Introduction to Supercomputing

Release 1.2.5

Jan Verschelde

Nov 10, 2024

Contents

CHAPTER 0

Preface

This document contains the lecture notes for the course MCS 572, introduction to supercomputing, at the University of Illinois at Chicago.

The first runs of the course followed the book of Wilkinson and Allen. The book by Kirk, Hwu, and El Hajj is our main reference for acceleration on Graphics Processing Units (GPUs).

The goal of the course is to study the design and analysis of parallel algorithms and their implementation using message passing, multithreading, multitasking, and acceleration. We study the application of parallel programs to solve scientific problems.

0.1 Three Different Types of Parallelism

We distinguish between three different types of parallel computers:

1. Distributed Memory Parallel Computers
2. Shared Memory Parallel Computers
3. General Purpose Graphics Processing Units

for which we have three corresponding programming models:

1. Message Passing
2. Multithreading and Multitasking
3. Data Staging Algorithms

Load balancing algorithms are introduced for distributed memory and shared memory parallel computers. Partitioning and divide-and-conquer strategies are applied in the design of parallel algorithms. Tools and models to evaluate the performance of parallel programs are the task graph, isoefficiency, and the roofline model. Pipelining is a common technique to make parallel programs.

0.2 Programming Languages

The course is not a programming course, but a computational course. Familiarity with computers and programming is assumed.

1. Python allows for high level parallel programing The package `mpi4py` enables distributed memory parallel programming via message passing. Kernels can be launched for GPU execution using PyCUDA.
2. Julia has a MATLAB-like syntax and offers support for message passing via the package `MPI.jl`, has tools for multithreading and multitasking. One nice feature of the Julia ecosystem is the ability for vendor agnostic GPU acceleration.
3. C++ achieves performance portability. Using C and C++ as a programming model requires greater attention to the memory model.

It is important to emphasize that we are using programming languages to run short parallel programs, or to call code from software libraries.

0.3 Bibliography

1. Barry Wilkinson and Michael Allen: *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers* Pearson Prentice Hall, second edition, 2005.
2. David B. Kirk, Wen-mei W. Hwu, Izzat El Hajj: *Programming Massively Parallel Processors. A Hands-on Approach* Elsevier/Morgan Kaufmann Publishers, fourth edition, 2023.

CHAPTER 1

Introduction to Parallel Computing

This chapter collects some notes on the first three lectures in the first week of the course. We introduce some terminology and end with high level parallelism.

1.1 Introduction

In this first lecture we define supercomputing, speedup, and efficiency. Gustafson's Law reevaluates Amdahl's Law.

1.1.1 What is a Supercomputer?

Doing supercomputing means to use a supercomputer and is also called high performance computing.

Definition of Supercomputer

A *supercomputer* is a computing system (hardware, system & application software) that provides close to the best currently achievable sustained performance on demanding computational problems.

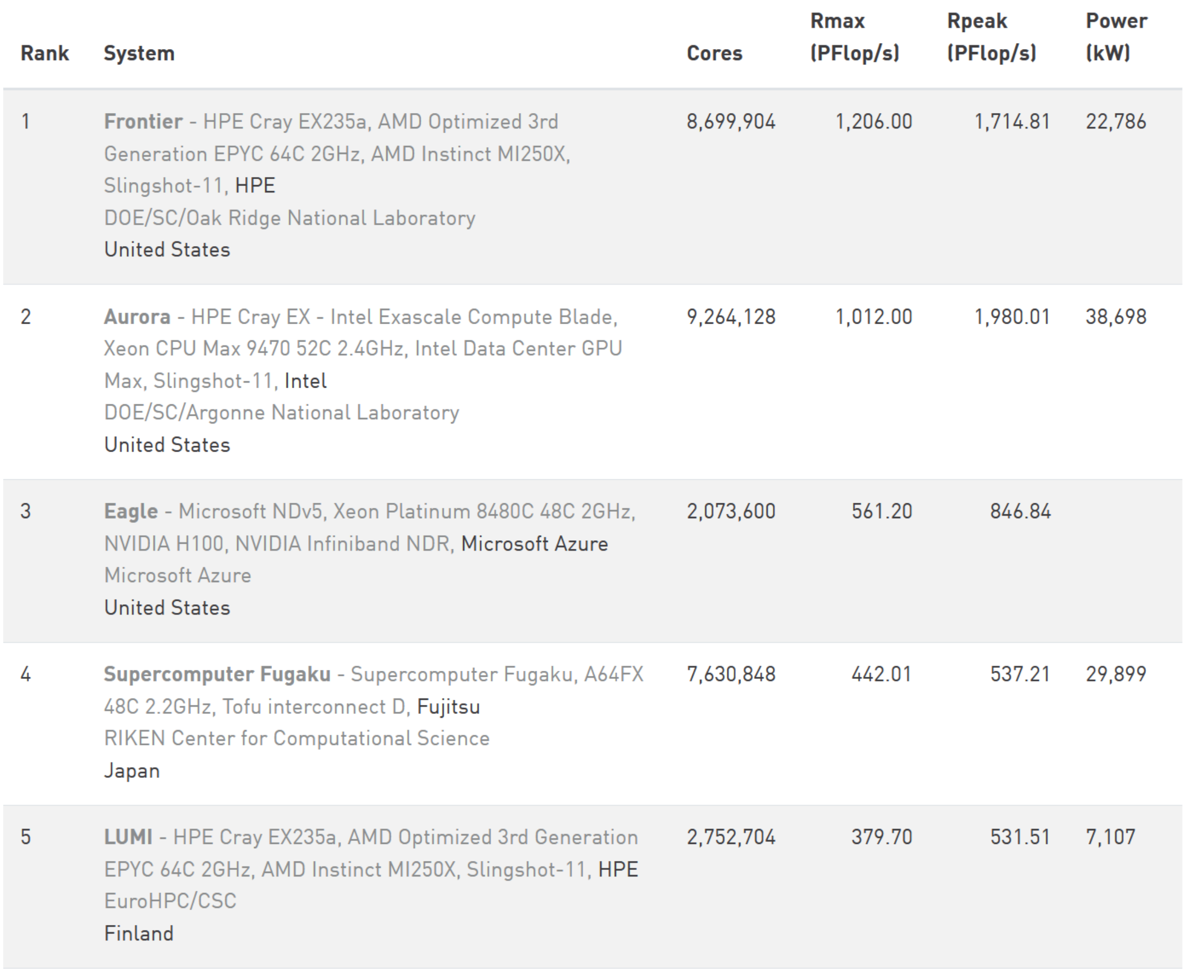
The current classification of supercomputers can be found at the TOP500 Supercomputer Sites.

The list of the top 5 supercomputers as of June 2024 is shown in Fig. 1.1.

A *flop* is a floating point operation. Performance is often measured in the number of flops per second. If two flops can be done per clock cycle, then a processor at 3GHz can theoretically perform 6 billion flops (6 gigaflops) per second. All computers in the top 10 achieve more than 1 petaflop per second.

Some system terms and architectures are listed below:

- core for a CPU: unit capable of executing a thread, for a GPU: a streaming multiprocessor.
- $\mathrm{R}_{\mathrm{max}}$ maximal performance achieved on the LINPACK benchmark (solving a dense linear system) for problem size $\mathrm{N}_{\mathrm{max}}$, measured in Gflop/s.
- $\mathrm{R}_{\mathrm{peak}}$ theoretical peak performance measured in Gflop/s.



Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107

Fig. 1.1: Top 5 Supercomputers, as of June 2024.

- Power total power consumed by the system.

In Fig. 1.2 the evolution of the performance of the top 500 supercomputers is illustrated.

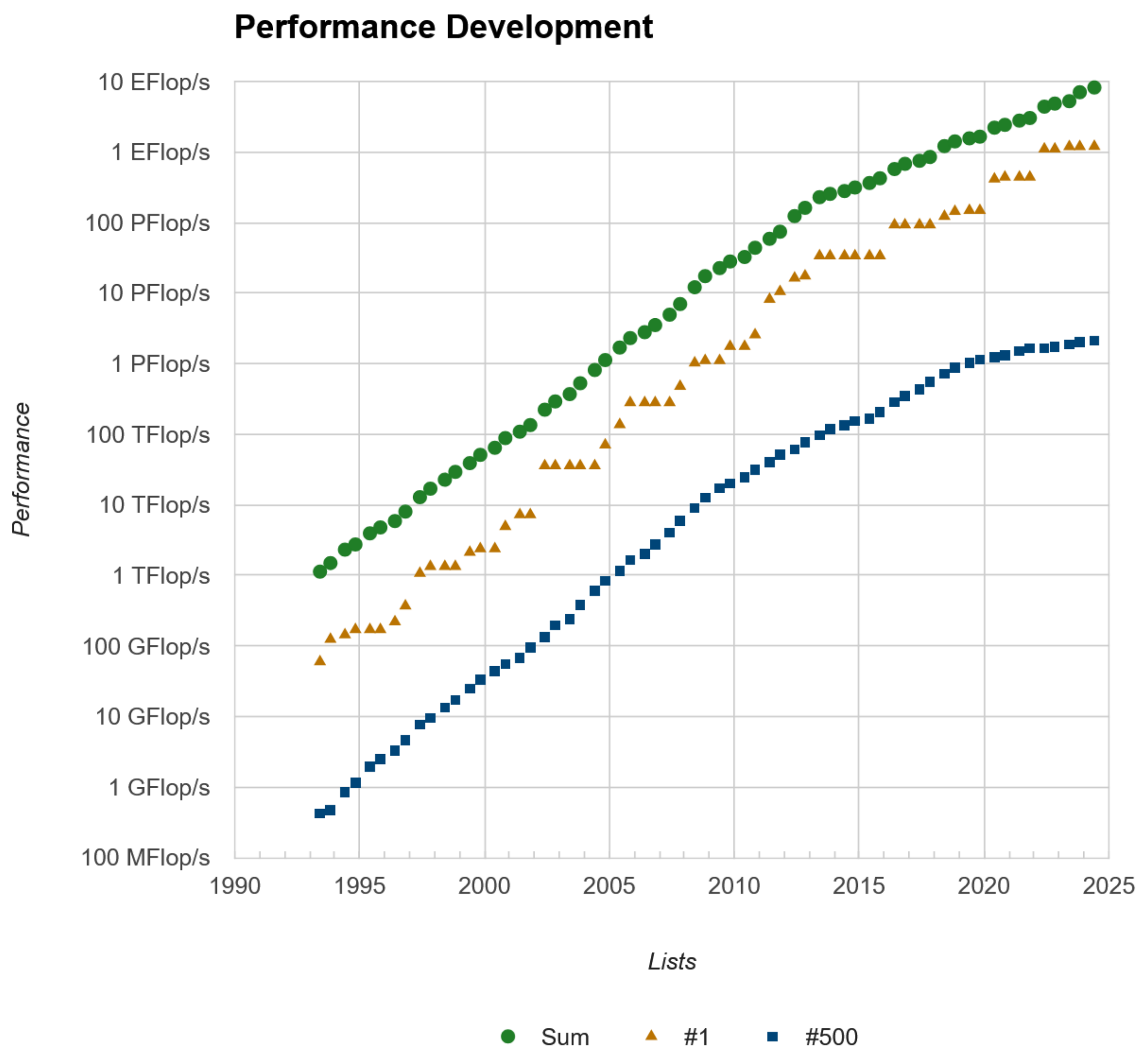


Fig. 1.2: Historical evolution of performance of supercomputers.

Concerning the types of architectures, we note the use of commodity leading edge microprocessors running at their maximal clock and power limits. Alternatively supercomputers use special processor chips running at less than maximal power to achieve high physical packaging densities. Thirdly, we observe mix of chip types and accelerators (GPUs).

1.1.2 Measuring Performance

The next definition links speedup and efficiency.

Definition of Speedup and Efficiency

By p we denote the number of processors.

$$\text{Speedup } S(p) = \frac{\text{sequential execution time}}{\text{parallel execution time}}.$$

Efficiency is another measure for parallel performance:

$$\text{Efficiency } E(p) = \frac{\text{speedup}}{\text{number of processors}} = \frac{S(p)}{p} \times 100\%.$$

In the best case, we hope: $S(p) = p$ and $E(p) = 100\%$. If $E = 50\%$, then on average processors are idle for half of the time.

While we hope for $S(p) = p$, we may achieve $S(p) > p$ and achieve *superlinear speedup*. Consider for example a sequential search in an unsorted list. A parallel search by p processors divides the list evenly in p sublists.

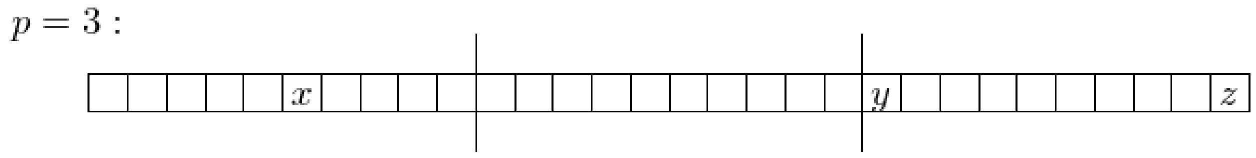


Fig. 1.3: A search illustrates superlinear speedup.

The sequential search time depends on position in list. The parallel search time depends on position in sublist. We obtain a huge speedup if the element we look for is for example the first element of the last sublist, as illustrated in Fig. 1.3.

1.1.3 Amdahl's and Gustafson's Law

Consider a job that takes time t on one processor. Let R be the fraction of t that must be done sequentially, $R \in [0, 1]$. Consider Fig. 1.4.

We then calculate the speedup on p processors as

$$S(p) \leq \frac{t}{Rt + \frac{(1-R)t}{p}} = \frac{1}{R + \frac{1-R}{p}} \leq \frac{1}{R}.$$

Amdahl's Law (1967)

Let R be the fraction of the operations which cannot be done in parallel. The speedup with p processors is bounded by $\dfrac{1}{R + \frac{1-R}{p}}$.

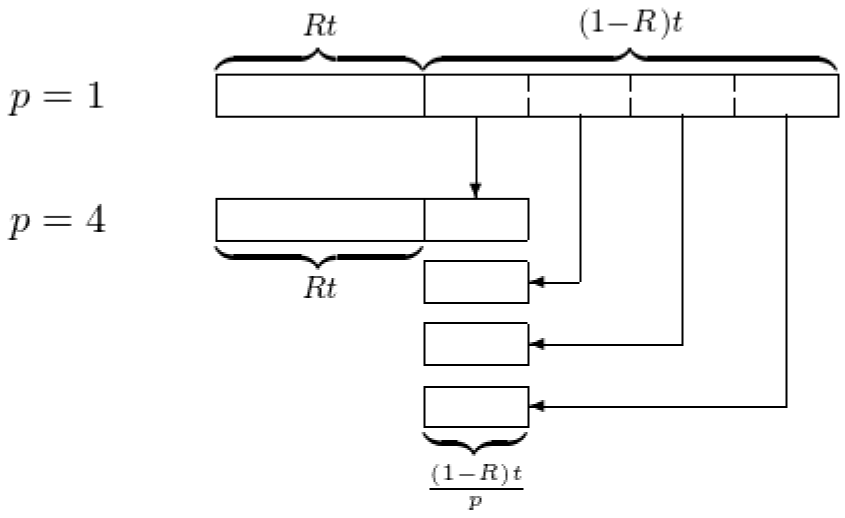


Fig. 1.4: Illustration of Amdahl's Law.

Corollary of Amdahl's Law

$S(p) \leq \dfrac{1}{R}$ as $p \rightarrow \infty$.

Example of Amdahl's Law

Suppose 90% of the operations in an algorithm can be executed in parallel. What is the best speedup with 8 processors? What is the best speedup with an unlimited amount of processors?

$$p = 8 : \frac{1}{\frac{1}{10} + \left(1 - \frac{1}{10}\right)\frac{1}{8}} = \frac{80}{17} \approx 4.7$$

$$p = \infty : \frac{1}{1/10} = 10.$$

In contrast to Ahmdahl's Law, we can start with the observation that many results obtained on supercomputers cannot be obtained one one processor. To derive the notion of *scaled speedup*, we start by considering a job that took time t on p processors. Let s be the fraction of t that is done sequentially. Consider Fig. 1.5.

The we computed the scaled speedup as follows:

$$S_s(p) \leq \frac{st + p(1-s)t}{t} = s + p(1-s) = p + (1-p)s.$$

We observe that the problem size scales with the number of processors!

Gustafson's Law (1988)

If s is the fraction of serial operations in a parallel program run on p processors, then the scaled speedup is bounded by $p + (1-p)s$.

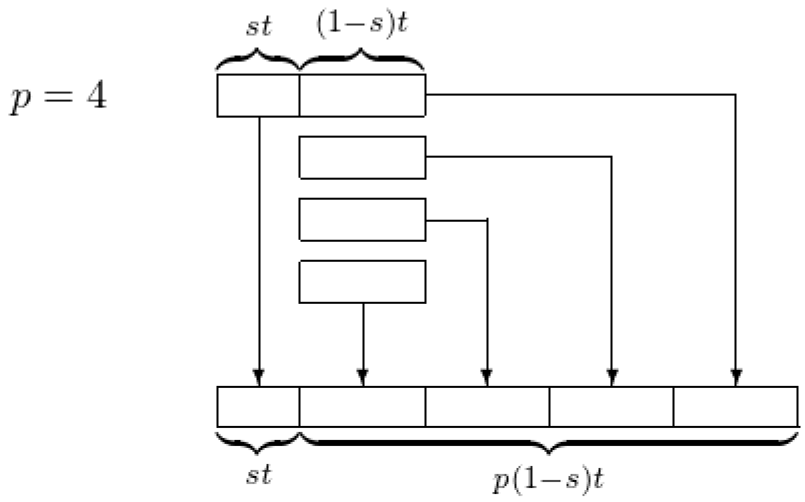


Fig. 1.5: Illustration of Gustafson's Law.

Example of Gustafson's Law

Suppose benchmarking reveals that 5% of time on a 64-processor machine is spent on one single processor (e.g.: root node working while all other processors are idle). Compute the scaled speedup.

$$p = 64, s = 0.05 : S_s(p) \leq 64 + (1 - 64)0.05 = 64 - 3.15 = 60.85.$$

More processing power often leads to better results, and we can achieve *quality up*. Below we list some examples.

- Finer granularity of a grid; e.g.: discretization of space and/or time in a differential equation.
- Greater confidence of estimates; e.g.: enlarged number of samples in a simulation.
- Compute with larger numbers (multiprecision arithmetic); e.g.: solve an ill-conditioned linear system.

If we can afford to spend the same amount of time on solving a problem then we can ask how much better we can solve the same problem with p processors? This leads to the notion of quality up.

$$\text{quality up } Q(p) = \frac{\text{quality on } p \text{ processors}}{\text{quality on 1 processor}}$$

$Q(p)$ measures improvement in quality using p procesors, keeping the computational time fixed.

1.1.4 Bibliography

1. S.G. Akl. **Superlinear performance in real-time parallel computation.** *The Journal of Supercomputing*, 29(1):89–111, 2004.
2. J.L. Gustafson. **Reevaluating Amdahl's Law.** *Communications of the ACM*, 31(5):532-533, 1988.
3. P.M. Kogge and T.J. Dysart. **Using the TOP500 to trace and project technology and architecture trends.** In *SC'11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM 2011.
4. B. Wilkinson and M. Allen. *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 2nd edition, 2005.
5. J.M. Wing. **Computational thinking.** *Communications of the ACM*, 49(3):33-35, 2006.

1.1.5 Exercises

1. How many processors whose clock speed runs at 3.0GHz does one need to build a supercomputer which achieves a theoretical peak performance of at least 4 Tera Flops? Justify your answer.
2. Suppose we have a program where 2% of the operations must be executed sequentially. According to Amdahl's law, what is the maximum speedup which can be achieved using 64 processors? Assuming we have an unlimited number of processors, what is the maximal speedup possible?
3. Benchmarking of a program running on a 64-processor machine shows that 2% of the operations are done sequentially, i.e.: that 2% of the time only one single processor is working while the rest is idle. Use Gustafson's law to compute the scaled speedup.

1.2 Classifications and Scalability

Parallel computers can be classified by instruction and data streams. Another distinction is between shared and distributed memory systems. We define clusters and the scalability of a problem. Network topologies apply both to hardware configurations and algorithms to transfer data.

1.2.1 Types of Parallel Computers

In 1966, Flynn introduced what is called the *MIMD and SIMD classification*:

- **SISD**: Single Instruction Single Data stream

  One single processor handles data sequentially. We use pipelining (e.g.: car assembly) to achieve parallelism.
- **MISD**: Multiple Instruction Single Data stream

  This is called systolic arrays and has been of little interest.
- **SIMD**: Single Instruction Multiple Data stream

  In graphics computing, one issues the same command for pixel matrix.

  One has vector and arrays processors for regular data structures.
- **MIMD**: Multiple Instruction Multiple Data stream

  This is the general purpose multiprocessor computer.

One model is **SPMD**: Single Program Multiple Data stream: All processors execute the same program. Branching in the code depends on the identification number of the processing node. Manager worker paradigm fits the SPMD model: manager (also called root) has identification zero; and workers are labeled $1, 2, \ldots, p-1$.

The distinction between shared and distributed memory parallel computers is illustrated with an example in Fig. 1.6.

1.2.2 Clusters and Scalability

Definition of Cluster

A *cluster* is an independent set of computers combined into a unified system through software and networking.

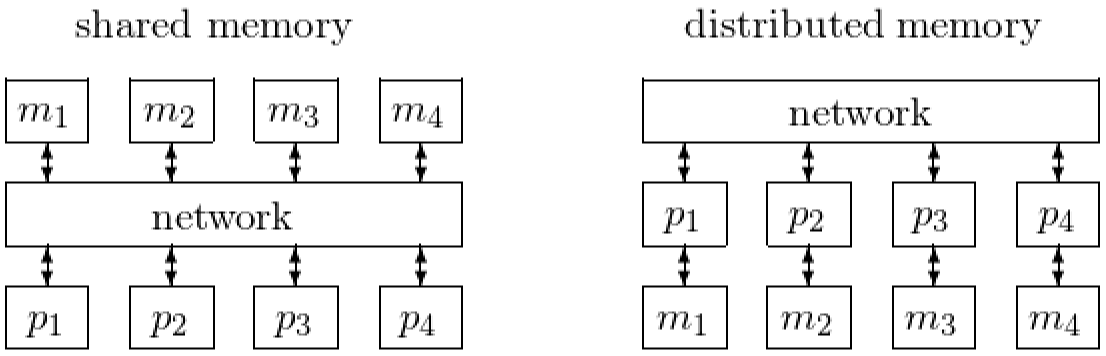


Fig. 1.6: A shared memory multicomputer has one single address space, accessible to every processor. In a distributed memory multicomputer, every processor has its own memory accessible via messages through that processor. Most nodes in a parallel computers have multiple cores.

Beowulf clusters are scalable performance clusters based on commodity hardware, on a private network, with open source software.

Three factors drove the clustering revolution in computing. First is the availability of commodity hardware: choice of many vendors for processors, memory, hard drives, etc… Second, concerning networking, Ethernet is dominating commodity networking technology, supercomputers have specialized networks. The third factor consists of open source software infrastructure: Linux and MPI.

We next discuss scalability as it relates to message passing in clusters.

$$\mbox{total time} = \mbox{computation time} + \underbrace{\mbox{communication time}}_{\mbox{o v e r h e a d}}$$

Because we want to reduce the overhead, the

$$\mbox{computation/communication ratio} = \frac{\mbox{computation time}}{\mbox{communication time}}$$

determines the *scalability* of a problem: The question is *How well can we increase the problem size n, keeping p, the number of processors fixed?* We desire that the order of overhead $\ll$ order of computation, so ratio $\rightarrow \infty$, Examples: $O(\log_2(n)) < O(n) < O(n^2)$. One remedy is to overlap the communication with computation.

In a distributed shared memory computer: the memory is physically distributed with each processor; and each processor has access to all memory in single address space. The benefits are that message passing often not attractive to programmers; and while shared memory computers allow limited number of processors, distributed memory computers scale well. The disadvantage is that access to remote memory location causes delays and the programmer does not have control to remedy the delays.

1.2.3 Network Topologies

We distinguish between static connections and dynamic network topologies enabled by switches. Below is some terminology.

- bandwidth: number of bits transmitted per second
- on latency, we distinguish tree types:
  - **message latency**: time to send zero length message (or startup time),
  - **network latency**: time to make a message transfer the network,
  - **communication latency**: total time to send a message including software overhead and interface delays.

- diameter of network: minimum number of links between nodes that are farthest apart
- on bisecting the network:

  bisection width: number of links needed to cut network
  in two equal parts,

  bisection bandwidth: number of bits per second which can
  be sent from one half of network to the other half.

Connecting p nodes in complete graph is too expensive. Small examples of an array and ring topology are shown in Fig. 1.7. A matrix and torus of 16 nodes is shown in Fig. 1.8.

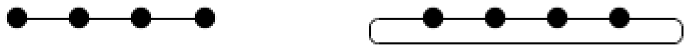


Fig. 1.7: An array and ring of 4 notes.

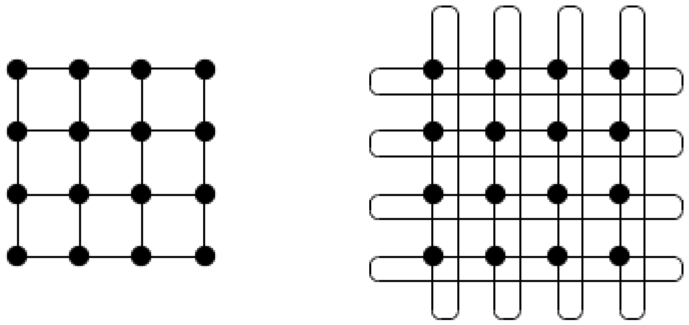


Fig. 1.8: A matrix and torus of 16 nodes.

A hypercube network is defined as follows. Two nodes are connected $\Leftrightarrow$ their labels differ in exactly one bit. Simple examples are shown in Fig. 1.9.

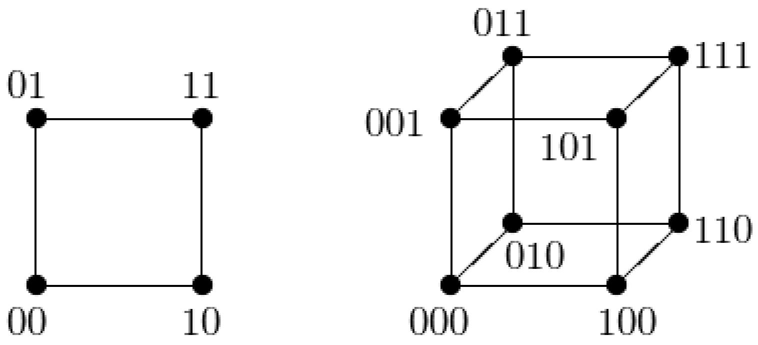


Fig. 1.9: Two special hypercubes: a square and cube.

e-cube or left-to-right routing: flip bits from left to right, e.g.: going from node 000 to 101 passes through 100. In a hypercube network with p nodes, the maximum number of flips is $\log_2(p)$, and the number of connections is . . .?

Consider a binary tree. The leaves in the tree are processors. The interior nodes in the tree are switches. This gives rise to a tree network, shown in Fig. 1.10.

Often the tree is *fat*: with an increasing number of links towards the root of the tree.

Dynamic network topologies are realized by switches. In a shared memory multicomputer, processors are usually connected to memory modules by a crossbar switch. An example, for $p = 4$, is shown in Fig. 1.11.

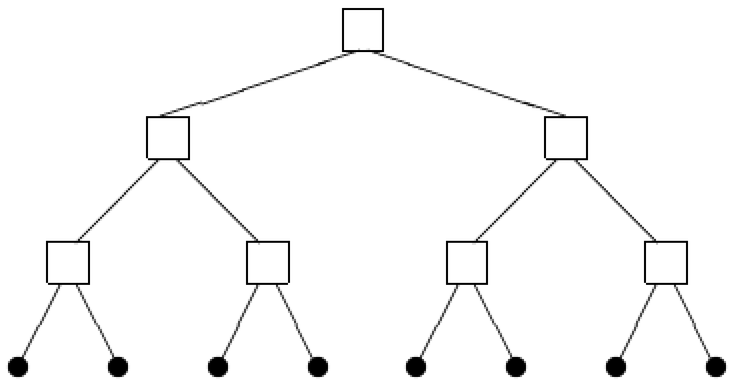


Fig. 1.10: A binary tree network.

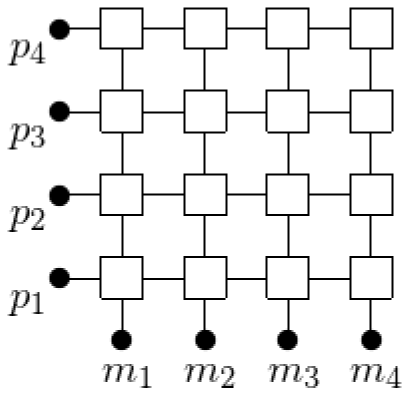


Fig. 1.11: Processors connected to memory modules via a crossbar switch.

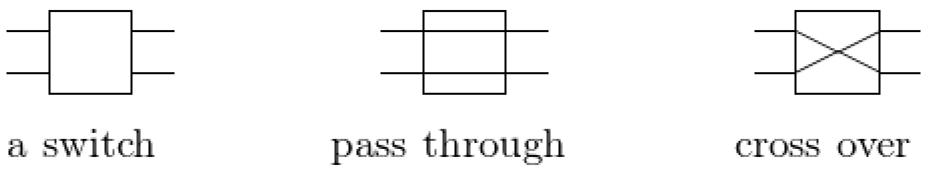


Fig. 1.12: 2-by-2 swiches.

A p-processor shared memory computer requires p^2 switches. 2-by-2 switches are shown in Fig. 1.12.

Changing from *pass through* to *cross over* configuration changes the connections between the computers in the network, see Fig. 1.13.

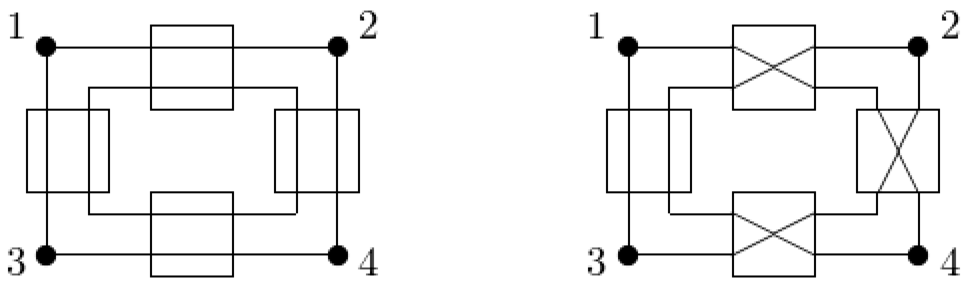


Fig. 1.13: Changing switches from pass through to cross over.

The rules in the routing algorithm in a multistage network are the following:

1. bit is zero: select upper output of switch; and
2. bit is one: select lower output of switch.

The first bit in the input determines the output of the first switch, the second bit in the input determines the output of the second switch. Fig. 1.14 shows a 2-stage network between 4 nodes.

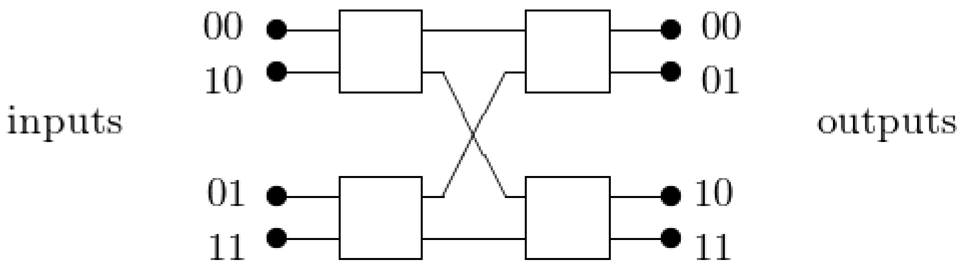


Fig. 1.14: A 2-stage network between 4 nodes.

The communication between 2 nodes using 2-by-2 switches causes *blocking*: other nodes are prevented from communicating. The number of switches for p processors equals $\log_2(p) \times \frac{p}{2}$. Fig. 1.15 shows the application of circuit switching for $p = 2^3$.

We distinguish between circuit and packet switching. If all circuits are occupied, communication is blocked. Alternative solution: packet switching: message is broken in packets and sent through network. Problems to avoid:

- **deadlock**: Packets are blocked by other packets waiting to be forwarded. This occurs when the buffers are full with packets. Solution: avoid cycles using e-cube routing algorithm.
- **livelock**: a packet keeps circling the network and fails to find its destination.

The network in a typical cluster is shown in Fig. 1.16.

Modern workstations are good for software development and for running modest test cases to investigate scalability. We give two examples. HP workstation Z800 RedHat Linux: two 6-core Intel Xeon at 3.47Ghz, 24GB of internal memory, and 2 NVDIA Tesla C2050 general purpose graphic processing units. Microway whisperstation RedHat Linux: two 8-core Intel Xeon at 2.60Ghz, 128GB of internal memory, and 2 NVDIA Tesla K20C general purpose graphic processing units.

The Hardware Specs of the new UIC Condo cluster is at <http://rc.uic.edu/hardware-specs>:

- Two login nodes are for managing jobs and file system access.

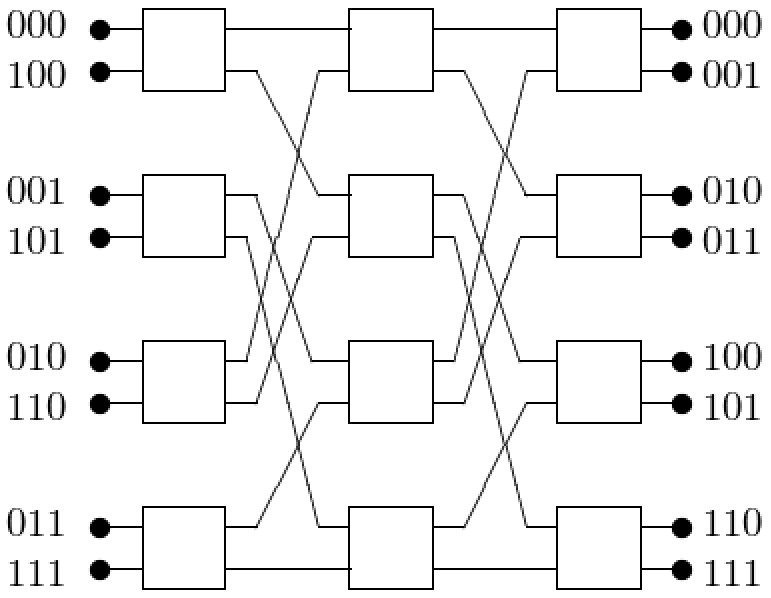


Fig. 1.15: A 3-stage Omega interconnection network.

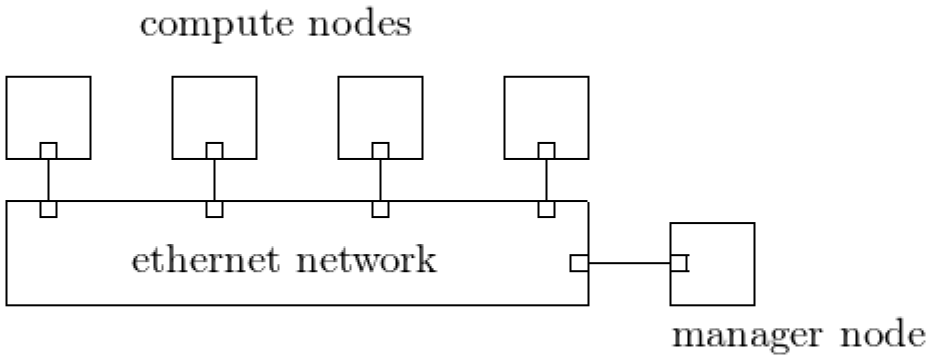


Fig. 1.16: A cluster connected via ethernet.

- 160 nodes, each node has 16 cores, running at 2.60GHz, 20MB cache, 128GB RAM, 1TB storage.
- 40 nodes, each node has 20 cores, running at 2.50GHz, 20MB cache, 128GB RAM, 1TB storage.
- 3 large memory compute nodes, each with 32 cores having 1TB RAM giving 31.25GB per core. Total adds upto 96 cores and 3TB of RAM.
- Total adds up to 3,456 cores, 28TB RAM, and 203TB storage.
- 288TB fast scratch communicating with nodes over QDR infiniband.
- 1.14PB of raw persistent storage.

1.2.4 Bibliography

1. M.J. Flynn and K. W. Rudd. **Parallel Architectures.** *ACM Computing Surveys* 28(1): 67-69, 1996.
2. A. Grama, A. Gupta, G. Karypis, V. Kumar. *Introduction to Parallel Computing*. Pearson. Addison-Wesley. Second edition, 2003.
3. G.K. Thiruvathukal. **Cluster Computing. Guest Editor's Introduction.** *Computing in Science and Engineering* 7(2): 11-13, 2005.
4. B. Wilkinson and M. Allen. *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 2nd edition, 2005.

1.2.5 Exercises

1. Derive a formula for the number of links in a hypercube with $p = 2^k$ processors for some positive number k.
2. Consider a network of 16 nodes, organized in a 4-by-4 mesh with connecting loops to give it the topology of a torus (or doughnut). Can you find a mapping of the nodes which give it the topology of a hypercube? If so, use 4 bits to assign labels to the nodes. If not, explain why.
3. We derived an Omega network for eight processors. Give an example of a configuration of the switches which is blocking, i.e.: a case for which the switch configurations prevent some nodes from communicating with each other.
4. Draw a multistage Omega interconnection network for $p = 16$.

1.3 High Level Parallel Processing

In this lecture we give three examples of what could be considered high level parallel processing. First we see how we may accelerate matrix-matrix multiplication using the computer algebra system Maple. Then we explore the multiprocessing module in Python and finally we show how multitasking in the object-oriented language Ada is effective in writing parallel programs.

In high level parallel processing we can use an existing programming environment to obtain parallel implementations of algorithms. In this lecture we give examples of three fundamentally different programming tools to achieve parallelism: multi-processing (distributed memory), multi-threading (shared memory), and use of accelerators (general purpose graphics processing units).

There is some sense of subjectivity with the above description of what high level means. If unfamiliar with Python, Julia, or Ada, then the examples in this lecture may also seem too technical. What does count as high level is that we do not worry about technical issues as communication overhead, resource utilitization, synchronization, etc., but we ask only two questions. Is the parallel code correct? Does the parallel code run faster?

1.3.1 High-Level Parallel Programming

In an attempt to define high-level parallel programming, we list some characteristics:

- familiar: no new language needed,
- interactive: receive quick feedback,
- personal: no supercomputer needed.

Rapid prototyping can help to decide if parallelism is feasible for a particular computation in an application.

The 17th international symposium on High-Level Parallel Programming and Applications (HLPP 2024), was held in Pisa, Italy, July 4-5, 2024.

Some of the topics include

- high-level programming and performance models,
- software synthesis, automatic code generation,
- applications using high-level languages and tools,
- formal models of verification.

While `high-level` also covers abstract and formal, there is a need for practical software and tools, so the `high-level` is not the opposite of technical.

1.3.2 Multiprocessing in Python

Some advantages of the scripting language Python are: educational, good for novice programmers, modules for scientfic computing: NumPy, SciPy, SymPy. Sage, a free open source mathematics software system, uses Python to interface many free and open source software packages. Our example: $\int_0^1 \sqrt{1-x^2}dx = \frac{\pi}{4}$. We will use the Simpson rule (available in SciPy) as a relatively computational intensive example.

We develop our scripts in an interactive Python shell:

```
$ python
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from scipy.integrate import simps
>>> from scipy import sqrt, linspace, pi
>>> f = lambda x: sqrt(1-x**2)
>>> x = linspace(0,1,1000)
>>> y = f(x)
>>> I = simps(y,x)
>>> 4*I
3.1415703366671113
```

The script `simpson4pi.py` is below:

```python
from scipy.integrate import simps
from scipy import sqrt, linspace, pi
f = lambda x: sqrt(1-x**2)
x = linspace(0,1,100); y = f(x)
I = 4*simps(y,x); print '10^2', I, abs(I - pi)
x = linspace(0,1,1000); y = f(x)
```

(continues on next page)

(continued from previous page)

```
I = 4*simps(y,x); print '10^3', I, abs(I - pi)
x = linspace(0,1,10000); y = f(x)
I = 4*simps(y,x); print '10^4', I, abs(I - pi)
x = linspace(0,1,100000); y = f(x)
I = 4*simps(y,x); print '10^5', I, abs(I - pi)
x = linspace(0,1,1000000); y = f(x)
I = 4*simps(y,x); print '10^6', I, abs(I - pi)
x = linspace(0,1,10000000); y = f(x)
I = 4*simps(y,x); print '10^7', I, abs(I - pi)
```

To run the script `simpson4pi.py`, We type at the command prompt `$`:

```
$ python simpson4pi.py
10^2 3.14087636133 0.000716292255311
10^3 3.14157033667 2.23169226818e-05
10^4 3.1415919489 7.04691599296e-07
10^5 3.14159263131 2.2281084977e-08
10^6 3.14159265289 7.04557745479e-10
10^7 3.14159265357 2.22573071085e-11
```

The slow convergence makes that this is certainly not a very good way to approximate π, but it fits our purposes. We have a slow computationally intensive process that we want to run in parallel.

We measure the time it takes to run a script as follows. Saving the content of

```
from scipy.integrate import simps
from scipy import sqrt, linspace, pi
f = lambda x: sqrt(1-x**2)
x = linspace(0,1,10000000); y = f(x)
I = 4*simps(y,x)
print I, abs(I - pi)
```

into the script `simpson4pi1.py`, we use the unix `time` command.

```
$ time python simpson4pi1.py
3.14159265357 2.22573071085e-11

real    0m2.853s
user    0m1.894s
sys     0m0.956s
```

The `real` is the so-called wall clock time, `user` indicates the time spent by the processor and `sys` is the system time.

Python has a multiprocessing module. The script belows illustrates its use.

```
from multiprocessing import Process
import os
from time import sleep

def say_hello(name,t):
    """
    Process with name says hello.
    """
    print 'hello from', name
```

(continues on next page)

(continued from previous page)

```
    print 'parent process :', os.getppid()
    print 'process id :', os.getpid()
    print name, 'sleeps', t, 'seconds'
    sleep(t)
    print name, 'wakes up'

pA = Process(target=say_hello, args = ('A',2,))
pB = Process(target=say_hello, args = ('B',1,))
pA.start(); pB.start()
print 'waiting for processes to wake up...'
pA.join(); pB.join()
print 'processes are done'
```

Running the script shows the following on screen:

```
$ python multiprocess.py
waiting for processes to wake up...
hello from A
parent process : 737
process id : 738
A sleeps 2 seconds
hello from B
parent process : 737
process id : 739
B sleeps 1 seconds
B wakes up
A wakes up
processes are done
```

Let us do numerical integration with multiple processes, with the script `simpson4pi2.py` listed below.

```
from multiprocessing import Process, Queue
from scipy import linspace, sqrt, pi
from scipy.integrate import simps

def call_simpson(fun, a,b,n,q):
    """
    Calls Simpson rule to integrate fun
    over [a,b] using n intervals.
    Adds the result to the queue q.
    """
    x = linspace(a, b, n)
    y = fun(x)
    I = simps(y, x)
    q.put(I)

def main():
    """
    The number of processes is given at the command line.
    """
    from sys import argv
    if len(argv) < 2:
```

(continues on next page)

(continued from previous page)

```
        print 'Enter the number of processes at the command line.'
        return
    npr = int(argv[1])
    crc = lambda x: sqrt(1-x**2)
    nbr = 20000000
    nbrsam = nbr/npr
    intlen = 1.0/npr
    queues = [Queue() for _ in range(npr)]
    procs = []
    (left, right) = (0, intlen)
    for k in range(1, npr+1):
        procs.append(Process(target=call_simpson, \
            args = (crc, left, right, nbrsam, queues[k-1])))
        (left, right) = (right, right+intlen)
    for process in procs:
        process.start()
    for process in procs:
        process.join()
    app = 4*sum([q.get() for q in queues])
    print app, abs(app - pi)
```

To check for speedup we run the script as follows:

```
$ time python simpson4pi2.py 1
3.14159265358 8.01003707807e-12

real    0m2.184s
user    0m1.384s
sys     0m0.793s
$ time python simpson4pi2.py 2
3.14159265358 7.99982302624e-12

real    0m1.144s
user    0m1.382s
sys     0m0.727s
$
```

We have as speedup 2.184/1.144 = 1.909.

1.3.3 Multithreading with Julia

Julia offers a fresh approach to numerical computing. The picture in Fig. 1.17 is taken from the Software Engineering Daily web site.

We apply multithreading in a Jupyter notebook, in a kernel installed with the environment variable set to 16 threads.

```
julia> using IJulia
julia> installkernel("Julia (16 threads)",
       env = Dict("JULIA_NUM_THREADS"=>"16"))
```

The matrix-matrix multiplication is executed by `mul!()` of BLAS, where BLAS stands for the Basic Linear Algebra Subroutines. Two issues we must consider:

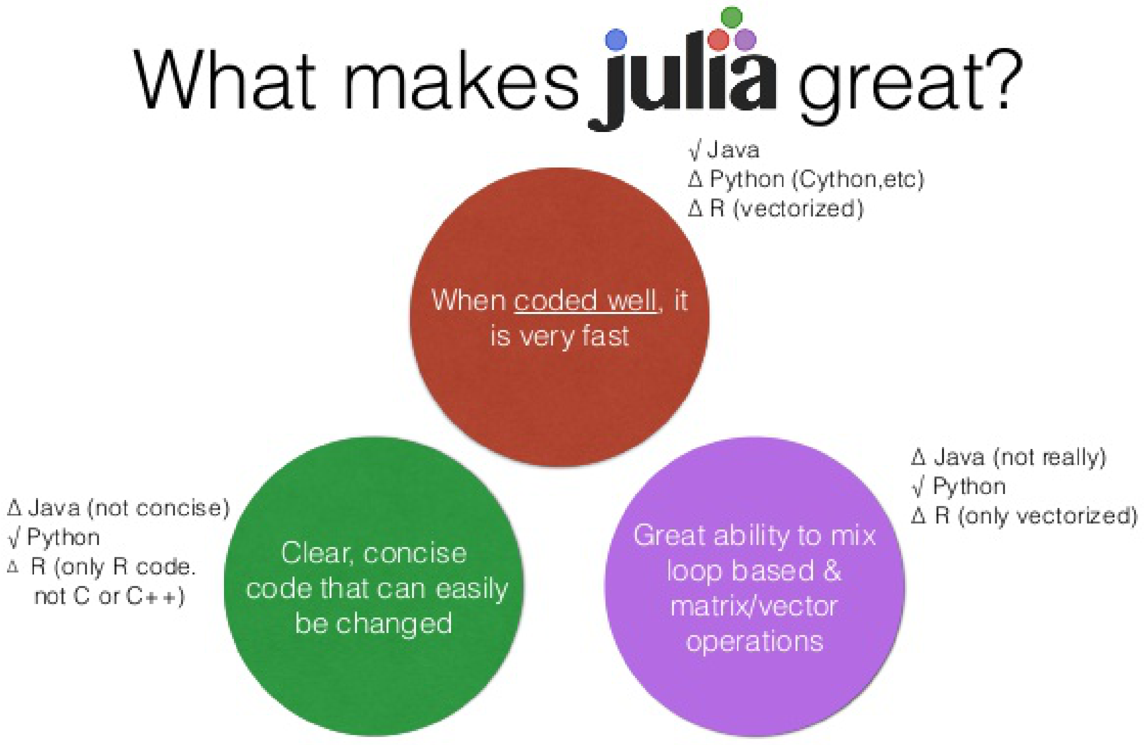

What makes julia great?
√ Java
Δ Python (Cython,etc)
Δ R (vectorized)
When coded well, it is very fast
Δ Java (not concise)
√ Python
Δ R (only R code. not C or C++)
Clear, concise code that can easily be changed
Δ Java (not really)
√ Python
Δ R (only vectorized)
Great ability to mix loop based & matrix/vector operations

Fig. 1.17: What makes Julia a great language.

1. Choose the size of the matrices large enough.
2. The time should not include the compilation time.

The instructions in code cells of a Jupyter notebook:

```
using LinearAlgebra
n = 8000
A = rand(n, n);
B = rand(n, n);
C = rand(n, n);
BLAS.set_num_threads(2)
@time mul!(C, A, B)
```

which reports `10.722 seconds (2.87 M allocations, 5.13% compilation time)`

Redo, and the second time: 10.359 seconds.

```
BLAS.set_num_threads(4)
@time mul!(C, A, B)
```

which prints `6.080 seconds`.

To illustrate parallel numerical integration, we can estimate π, via the area of the unit disk:

$$\int_0^1 \sqrt{1-x^2}dx = \frac{\pi}{4}$$

using the following steps:

1. Generate random uniformly distributed points with coordinates $(x, y) \in [0, +1] \times [0, +1]$.
2. We count a success when $x^2 + y^2 \leq 1$.

By the law of large numbers, the average of the observed successes converges to the expected value or mean, as the number of experiments increases.

Let us code this in Julia.

A dedicated random number generator is applied:

```
myrand(x::Int64) = (1103515245x + 12345) % 2^31
```

The function to estimate π is below:

```
"""
    function estimatepi(n)

Runs a simple Monte Carlo method
to estimate pi with n samples.
"""
function estimatepi(n)
    r = threadid()
    count = 0
    for i=1:n
        r = myrand(r)
        x = r/2^31
        r = myrand(r)
        y = r/2^31
```

(continues on next page)

(continued from previous page)

```
        count += (x^2 + y^2) <= 1
    end
    return 4*count/n
end
```

In a Jupyter notebook with 16 threads, we run a parallel for loop

```
nt = nthreads()
estimates = zeros(nt)
import Statistics
timestart = time()

@threads for i=1:nt
    estimates[i] = estimatepi(10_000_000_000/nt)
end

estpi = Statistics.mean(estimates)
elapsed16 = time() - timestart
```

The value for `elapsed16` is printed as `5.387`.

Running version 1.4.0-DEV.364 (2019-10-22) on two 22-core 2.2 GHz Intel Xeon E5-2699 processors in a CentOS Linux workstation with 256 GB RAM, produces the following table of timings.

Table 1.1: Running times with Julia multithreading.

p	wall clock time	elapsed time
1	1m 2.313s	62.060s
2	32.722s	32.418s
3	22.471s	22.190s
4	17.343s	17.042s
5	14.170s	13.896s
6	12.300s	11.997s
7	10.702s	10.442s

1.3.4 Tasking in Ada

Ada is a an object-oriented standardized language. Strong typing aims at detecting most errors during compile time. The tasking mechanism implements parallelism. The gnu-ada compiler produces code that maps tasks to threads. The main point is that shared-memory parallel programming can be done in a high level programming language as Ada.

The Simpson rule as an Ada function is shown below:

```
type double_float is digits 15;

function Simpson
  ( f : access function ( x : double_float )
        return double_float;
    a,b : double_float ) return double_float is

-- DESCRIPTION :
--   Applies the Simpson rule to approximate the
```

(continues on next page)

(continued from previous page)

```
--   integral of f(x) over the interval [a,b].

  middle : constant double_float := (a+b)/2.0;
  length : constant double_float := b - a;

begin
  return length*(f(a) + 4.0*f(middle) + f(b))/6.0;
end Simpson;
```

Calling Simpson in a main program:

```
with Ada.Numerics.Generic_Elementary_Functions;

package Double_Elementary_Functions is
new Ada.Numerics.Generic_Elementary_Functions(double_float);

function circle ( x : double_float ) return double_float is

-- DESCRIPTION :
--   Returns the square root of 1 - x^2.

begin
  return Double_Elementary_Functions.SQRT(1.0 - x**2);
end circle;

v : double_float := Simpson(circle'access,0.0,1.0);
```

The composite Simpson rule is written as

```
function Recursive_Composite_Simpson
  ( f : access function ( x : double_float )
        return double_float;
    a,b : double_float; n : integer ) return double_float is

-- DESCRIPTION :
--   Returns the integral of f over [a,b] with n subintervals,
--   where n is a power of two for the recursive subdivisions.

  middle : double_float;

begin
  if n = 1 then
    return Simpson(f,a,b);
  else
    middle := (a + b)/2.0;
    return Recursive_Composite_Simpson(f,a,middle,n/2)
         + Recursive_Composite_Simpson(f,middle,b,n/2);
  end if;
end Recursive_Composite_Simpson;
```

The main procedure is

```
procedure Main is

  v : double_float;
  n : integer := 16;

begin
  for k in 1..7 loop
    v := 4.0*Recursive_Composite_Simpson
               (circle'access,0.0,1.0,n);
    double_float_io.put(v);
    text_io.put("  error :");
    double_float_io.put(abs(v-Ada.Numerics.Pi),2,2,3);
    text_io.put("  for n = "); integer_io.put(n,1);
    text_io.new_line;
    n := 16*n;
  end loop;
end Main;
```

Compiling and executing at the command line (with a makefile):

```
$ make simpson4pi
gnatmake simpson4pi.adb -o /tmp/simpson4pi
gcc -c simpson4pi.adb
gnatbind -x simpson4pi.ali
gnatlink simpson4pi.ali -o /tmp/simpson4pi

$ /tmp/simpson4pi
3.13905221789359E+00  error : 2.54E-03  for n = 16
3.14155300930713E+00  error : 3.96E-05  for n = 256
3.14159203419701E+00  error : 6.19E-07  for n = 4096
3.14159264391183E+00  error : 9.68E-09  for n = 65536
3.14159265343858E+00  error : 1.51E-10  for n = 1048576
3.14159265358743E+00  error : 2.36E-12  for n = 16777216
3.14159265358976E+00  error : 3.64E-14  for n = 268435456
```

We define a worker task as follows:

```
task type Worker
  ( name : integer;
    f : access function ( x : double_float )
        return double_float;
    a,b : access double_float; n : integer;
    v : access double_float );

task body Worker is

  w : access double_float := v;

begin
  text_io.put_line("worker" & integer'image(name)
                            & " will get busy ...");
  w.all := Recursive_Composite_Simpson(f,a.all,b.all,n);
  text_io.put_line("worker" & integer'image(name)
```

(continues on next page)

(continued from previous page)

```
                            & " is done.");
end Worker;
```

Launching workers is done as

```
type double_float_array is
  array ( integer range <> ) of access double_float;

procedure Launch_Workers
  ( i,n,m : in integer; v : in double_float_array ) is

-- DESCRIPTION :
--   Recursive procedure to launch n workers,
--   starting at worker i, to apply the Simpson rule
--   with m subintervals.  The result of the i-th
--   worker is stored in location v(i).

  step : constant double_float := 1.0/double_float(n);
  start : constant double_float := double_float(i-1)*step;
  stop : constant double_float := start + step;
  a : access double_float := new double_float'(start);
  b : access double_float := new double_float'(stop);
  w : Worker(i,circle'access,a,b,m,v(i));

begin
  if i >= n then
    text_io.put_line("-> all" & integer'image(n)
                              & " have been launched");
  else
    text_io.put_line("-> launched " & integer'image(i));
    Launch_Workers(i+1,n,m,v);
  end if;
end Launch_Workers;
```

We get the number of tasks at the command line:

```
function Number_of_Tasks return integer is

-- DESCRIPTION :
--   The number of tasks is given at the command line.
--   Returns 1 if there are no command line arguments.

  count : constant integer
        := Ada.Command_Line.Argument_Count;

begin
  if count = 0 then
    return 1;
  else
    declare
      arg : constant string
          := Ada.Command_Line.Argument(1);
```

(continues on next page)

(continued from previous page)

```
    begin
      return integer'value(arg);
    end;
  end if;
end Number_of_Tasks;
```

The main procedure is below:

```
procedure Main is

  nbworkers : constant integer := Number_of_Tasks;
  nbintervals : constant integer := (16**7)/nbworkers;
  results : double_float_array(1..nbworkers);
  sum : double_float := 0.0;

begin
  for i in results'range loop
    results(i) := new double_float'(0.0);
  end loop;
  Launch_Workers(1,nbworkers,nbintervals,results);
  for i in results'range loop
    sum := sum + results(i).all;
  end loop;
  double_float_io.put(4.0*sum); text_io.put(" error :");
  double_float_io.put(abs(4.0*sum-Ada.Numerics.pi));
  text_io.new_line;
end Main;
```

Times in seconds obtained as `time /tmp/simpson4pitasking p` for p = 1, 2, 4, 8, 16, and 32 on kepler.

Table 1.2: Running times with Ada Tasking.

p	real	user	sys	speedup
1	8.926	8.897	0.002	1.00
2	4.490	8.931	0.002	1.99
4	2.318	9.116	0.002	3.85
8	1.204	9.410	0.003	7.41
16	0.966	12.332	0.003	9.24
32	0.792	14.561	0.009	11.27

Speedups are computed as $\dfrac{\text{real time with p = 1}}{\text{real time with p tasks}}$.

1.3.5 Performance Monitoring

perfmon2 is a hardware-based performance monitoring interface for the Linux kernel. To monitor the performance of a program, with the gathering of performance counter statistics, type

```
$ perf stat program
```

at the command prompt. To get help, type `perf help`. For help on `perf stat`, type `perf stat help`.

To count flops, we can select an event we want to monitor.

On the Intel Sandy Bridge processor, the codes for double operations are

- `0x530110` for `FP_COMP_OPS_EXE:SSE_FP_PACKED_DOUBLE`, and
- `0x538010` for `FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE`.

To count the number of double operations, we do

```
$ perf stat -e r538010 -e r530110 /tmp/simpson4pitasking 1
```

and the output contains

```
Performance counter stats for '/tmp/simpson4pitasking 1':

    4,932,758,276 r538010
    3,221,321,361 r530110

       9.116025034 seconds time elapsed
```

1.3.6 Bibliography

1. J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. **A Fresh Approach to Numerical Computing.** *SIAM Review*, Vol 59, No 1, pages 65-98, 2017.
2. Ivo Balbaert, Avik Sengupta, Malcom Sherrington: *Julia: High Performance Programming. Leverage the power of Julia to design and develop high performing programs.* Packt Publishing, 2016.

1.3.7 Exercises

1. A Monte Carlo method to estimate $\pi/4$ generates random tuples (x, y), with x and y uniformly distributed in $[0, 1]$. The ratio of the number of tuples inside the unit circle over the total number of samples approximates $\pi/4$.

   ```
   >>> from random import uniform as u
   >>> X = [u(0,1) for i in xrange(1000)]
   >>> Y = [u(0,1) for i in xrange(1000)]
   >>> Z = zip(X,Y)
   >>> F = filter(lambda t: t[0]**2 + t[1]**2 <= 1, Z)
   >>> len(F)/250.0
   3.1440000000000001
   ```

   Use the multiprocessing module to write a parallel version, letting processes take samples independently. Compute the speedup.

2. Develop a parallel Julia version for the `simpson4pi` code.

CHAPTER 2

Introduction to Message Passing

To program distributed memory parallel computers, we apply message passing.

2.1 Basics of MPI

Programming distributed memory parallel computers happens through message passing. In this lecture we give basic examples of using the Message Passing Interface, in C, Python, and Julia. The explanation using C requires careful attention to the data movements and this understanding is needed when applying the wrappers in Python or Julia.

2.1.1 One Single Program Executed by all Nodes

A parallel program is a collection of concurrent processes. A process (also called a job or task) is a sequence of instructions. Usually, there is a 1-to-1 map between processes and processors. If there are more processes than processors, then processes are executed in a time sharing environment. We use the SPMD model: Single Program, Multiple Data. Every node executes the same program. Every node has a unique identification number (id) — the root node has number zero — and code can be executed depending on the id. In a manager/worker model, the root node is the manager, the other nodes are workers.

The letters MPI stands for Message Passing Interface. MPI is a standard specification for interprocess communication for which several implementations exist. When programming in C, we include the header

```
#include <mpi.h>
```

to use the functionality of MPI. `Open MPI` is an open source implementation of all features of MPI-2. In this lecture we use MPI in simple interactive programs, e.g.: as `mpicc` and `mpirun` are available on laptop computers.

Our first parallel program is `mpi_hello_world`. We use a `makefile` to compile, and then run with 3 processes. Instead of `mpirun -np 3` we can also use `mpiexec -n 3`.

```
$ make mpi_hello_world
mpicc mpi_hello_world.c -o /tmp/mpi_hello_world
```

(continues on next page)

(continued from previous page)

```
$ mpirun -np 3 /tmp/mpi_hello_world
Hello world from processor 0 out of 3.
Hello world from processor 1 out of 3.
Hello world from processor 2 out of 3.
$
```

To pass arguments to the MCA modules (MCA stands for Modular Component Architecture) we can call `mpirun -np` (or `mpiexec -n`) with the option `--mca` such as

```
mpirun --mca btl tcp,self -np 4 /tmp/mpi_hello_world
```

MCA modules have direct impact on MPI programs because they allow tunable parameters to be set at run time, such as * which BTL communication device driver to use, * what parameters to pass to that BTL, etc. Note: BTL = Byte Transfer Layer.

The code of the program `mpi_hello_world.c` is listed below.

```
#include <stdio.h>
#include <mpi.h>

int main ( int argc, char *argv[] )
{
   int i,p;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&p);
   MPI_Comm_rank(MPI_COMM_WORLD,&i);

   printf("Hello world from processor %d out of %d.\n",i,p);

   MPI_Finalize();

   return 0;
}
```

2.1.2 Initialization, Finalization, and the Universe

Let us look at some MPI constructions that are part of any program that uses MPI. Consider the beginning and the end of the program.

```
#include <mpi.h>

int main ( int argc, char *argv[] )
{
   MPI_Init(&argc,&argv);
   MPI_Finalize();
   return 0;
}
```

The `MPI_Init` processes the command line arguments. The value of `argc` is the number of arguments at the command line and `argv` contains the arguments as strings of characters. The first argument, `argv[0]` is the name of the program. The cleaning up of the environment is done by `MPI_Finalize()`.

`MPI_COMM_WORLD` is a predefined named constant handle to refer to the universe of p processors with labels from 0 to $p-1$. The number of processors is returned by `MPI_Comm_size` and `MPI_Comm_rank` returns the label of a node. For example:

```
int i,p;

MPI_Comm_size(MPI_COMM_WORLD,&p);
MPI_Comm_rank(MPI_COMM_WORLD,&i);
```

2.1.3 Broadcasting Data

Many parallel programs follow a manager/worker model. In a *broadcast* the same data is sent to all nodes. A broadcast is an example of a collective communication. In a *collective communication*, all nodes participate in the communication.

As an example, we broadcast an integer. Node with id 0 (manager) prompts for an integer. The integer is *broadcasted* over the network and the number is sent to all processors in the universe. Every worker node prints the number to screen. The typical application of broadcasting an integer is the broadcast of the dimension of data before sending the data.

The compiling and running of the program goes as follows:

```
$ make broadcast_integer
mpicc broadcast_integer.c -o /tmp/broadcast_integer

$ mpirun -np 3 /tmp/broadcast_integer
Type an integer number...
123
Node 1 writes the number n = 123.
Node 2 writes the number n = 123.
$
```

The command `MPI_Bcast` executes the broadcast. An example of the `MPI_Bcast` command:

```
int n;
MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
```

There are five arguments:

1. the address of the element(s) to broadcast;
2. the number of elements that will be broadcasted;
3. the type of all the elements;
4. a message label; and
5. the universe.

The full source listing of the program is shown below.

```
#include <stdio.h>
#include <mpi.h>

void manager ( int* n );
/* code executed by the manager node 0,
 * prompts the user for an integer number n */
```

(continues on next page)

(continued from previous page)

```
void worker ( int i, int n );
/* code executed by the i-th worker node,
 * who will write the integer number n to screen */

int main ( int argc, char *argv[] )
{
   int myid,numbprocs,n;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numbprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);

   if (myid == 0) manager(&n);

   MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);

   if (myid != 0) worker(myid,n);

   MPI_Finalize();

   return 0;
}

void manager ( int* n )
{
   printf("Type an integer number... \n");
   scanf("%d",n);
}

void worker ( int i, int n )
{
   printf("Node %d writes the number n = %d.\n",i,n);
}
```

2.1.4 Moving Data from Manager to Workers

Often we want to broadcast an array of doubles. The situation before broadcasting the dimension n to all nodes on a 4-processor distributed memory computer is shown at the top left of Fig. 2.1. After broadcasting of the dimension, each node *must* allocate space to hold as many doubles as the dimension.

We go through the code step by step. First we write the headers and the subroutine declarations. We include `stdlib.h` for memory allocation.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void define_doubles ( int n, double *d );
/* defines the values of the n doubles in d */

void write_doubles ( int myid, int n, double *d );
```

(continues on next page)

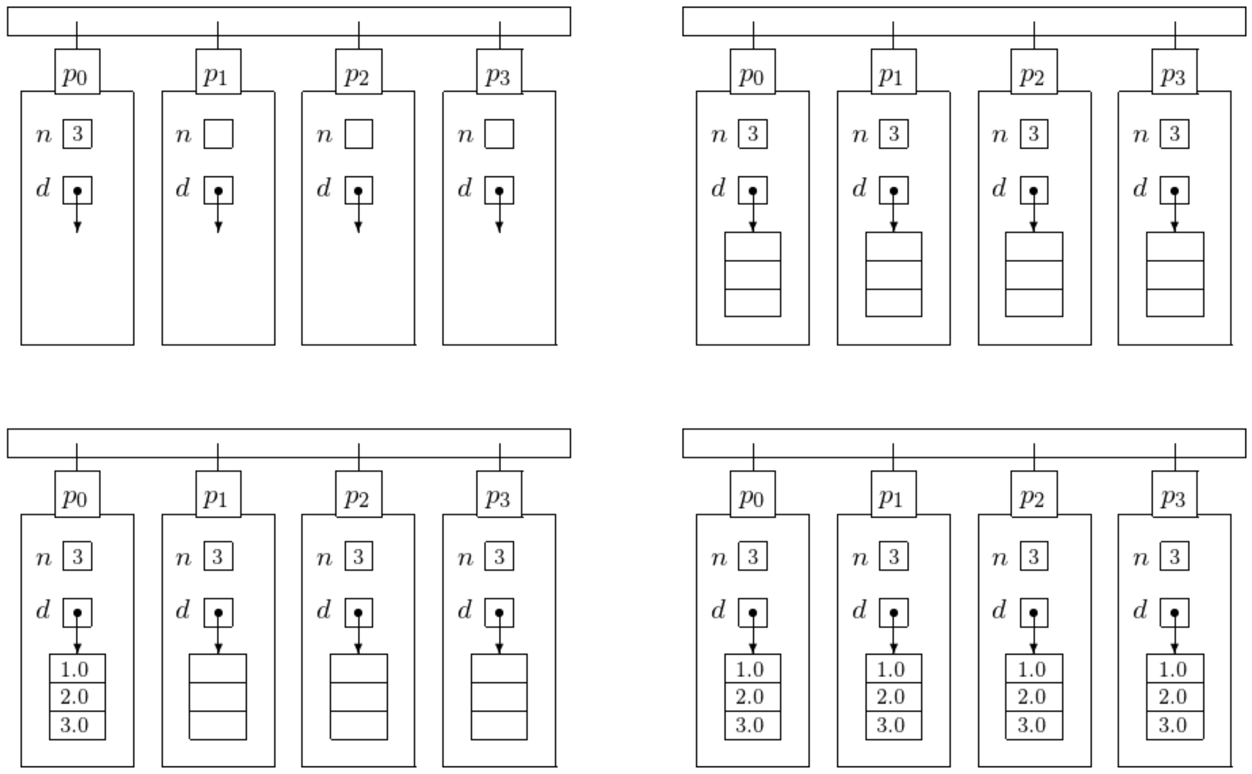


Fig. 2.1: On the schematic of a a distributed memory 4-processor computer, the top displays the situation before and after the broadcast of the dimension. After the broadcast of the dimension, each worker node allocates space for the array of doubles. The bottom two pictures display the situation before and after the broadcast of the array of doubles.

(continued from previous page)

```
/* node with id equal to myid
   writes the n doubles in d */
```

The main function starts by broadcasting the dimension.

```
int main ( int argc, char *argv[] )
{
   int myid,numbprocs,n;
   double *data;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numbprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);

   if (myid == 0)
   {
      printf("Type the dimension ...\n");
      scanf("%d",&n);
   }
   MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
```

The main program continues, allocating memory. It is very important that *every* node performs the memory allocation.

```
data = (double*)calloc(n,sizeof(double));

if (myid == 0) define_doubles(n,data);

MPI_Bcast(data,n,MPI_DOUBLE,0,MPI_COMM_WORLD);

if (myid != 0) write_doubles(myid,n,data);

MPI_Finalize();
return 0;
```

It is good programming practice to separate the code that does not involve any MPI activity in subroutines. The two subroutines are defined below.

```
void define_doubles ( int n, double *d )
{
   int i;

   printf("defining %d doubles ...\n", n);
   for(i=0; i < n; i++) d[i] = (double)i;
}

void write_doubles ( int myid, int n, double *d )
{
   int i;

   printf("Node %d writes %d doubles : \n", myid,n);
   for(i=0; i < n; i++) printf("%lf\n",d[i]);
}
```

2.1.5 MPI for Python

`MPI for Python` provides bindings of MPI for Python, allowing any Python program to exploit multiple processors. The code is available at github and it installs with `pip`. Version 4.0.0 was released on 28 July 2024.

The object oriented interface follows closely MPI-2 C++ bindings and supports point-to-point and collective communications of any pickable Python object, as well as numpy arrays and builtin bytes, strings. `mpi4py` gives the standard MPI *look and feel* in Python scripts to develop parallel programs. Often, only a small part of the code needs the efficiency of a compiled language. Python handles memory, errors, and user interaction.

Our first script is again a *hello world*, shown below.

```
from mpi4py import MPI

SIZE = MPI.COMM_WORLD.Get_size()
RANK = MPI.COMM_WORLD.Get_rank()
NAME = MPI.Get_processor_name()

MESSAGE = "Hello from %d of %d on %s." \
  % (RANK, SIZE, NAME)
print MESSAGE
```

Programs that run with MPI are executed with `mpiexec`. To run `mpi4py_hello_world.py` by 3 processes:

```
$ mpiexec -n 3 python mpi4py_hello_world.py
Hello from 2 of 3 on asterix.local.
Hello from 0 of 3 on asterix.local.
Hello from 1 of 3 on asterix.local.
$
```

Three Python interpreters are launched. Each interpreter executes the script, printing the hello message.

Let us consider again the basic MPI concepts and commands. `MPI.COMM_WORLD` is a predefined intracommunicator. An intracommunicator is a group of processes. All processes within an intracommunicator have a unique number. Methods of the intracommunicator `MPI.COMM_WORLD` are `Get_size()`, which returns the number of processes, and `Get_rank()`, which returns rank of executing process.

Even though every process runs the same script, the test `if MPI.COMM_WORLD.Get_rank() == i:` allows to specify particular code for the *i*-th process. `MPI.Get_processor_name() `` returns the name of the calling processor. A collective communication involves every process in the intracommunicator. A broadcast is a collective communication in which one process sends the same data to all processes, all processes receive the same data. In ``mpi4py`, a broadcast is done with the `bcast` method. An example:

```
$ mpiexec -n 3 python mpi4py_broadcast.py
0 has data {'pi': 3.1415926535897, 'e': 2.7182818284590}
1 has data {'pi': 3.1415926535897, 'e': 2.7182818284590}
2 has data {'pi': 3.1415926535897, 'e': 2.7182818284590}
$
```

To pass arguments to the MCA modules, we call `mpiexec` as `mpiexec --mca btl tcp,self -n 3 python mpi4py_broadcast.py`.

The script `mpi4py_broadcast.py` below performs a broadcast of a Python dictionary.

```
from mpi4py import MPI

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()

if(RANK == 0):
    DATA = {'e' : 2.7182818284590451,
            'pi' : 3.1415926535897931 }
else:
    DATA = None # DATA must be defined

DATA = COMM.bcast(DATA, root=0)
print RANK, 'has data', DATA
```

2.1.6 MPI wrappers for Julia

`MPI.jl` is a Julia interface to MPI, inspired by `mpi4py`.

Its installation requires a shared binary installation of a C MPI library, supporting the MPI 3.0 standard or later. The `MPI.jl` is a Julia package, install as `using MPI`.

To check if the installation works, run the Julia program `mpi_hello_world.jl`, shown below. The code below is adapted from `JuliaParallel/MPI.jl`, from the `docs/examples`:

```
using MPI
MPI.Init()

comm = MPI.COMM_WORLD
myid = MPI.Comm_rank(comm)
size = MPI.Comm_size(comm)

print("Hello from $myid of $size.\n")

MPI.Barrier(comm)
```

Run at the command prompt with `mpiexecjl`, after locating and adjusting the path.

2.1.7 Bibliography

1. S. Byrne, L.C. Wilcox, and V. Churavy: **MPI.jl: Julia bindings for the Message Passing Interface.** In *JuliaCon Proceedings*, 1(1), 68, 2021.
2. L. Dalcin, R. Paz, and M. Storti. **MPI for Python.** *Journal of Parallel and Distributed Computing*, 65:1108-1115, 2005.
3. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference Volume 1, The MPI Core*. Massachusetts Institute of Technology, second edition, 1998.

2.1.8 Exercises

0. Visit `http://www.mpi-forum.org/docs/` and look at the MPI book, available at

   `http://www.netlib.org/utk/papers/mpi-book/mpi-book.html`.

1. Adjust hello world so that after you type in your name once, when prompted by the manager node, every node salutes you, using the name you typed in.
2. We measure the wall clock time using `time mpirun` in the broadcasting of an array of doubles. To avoid typing in the dimension n, either define n as a constant in the program or redirect the input from a file that contains n. For increasing number of processes and n, investigate how the wall clock time grows.

2.2 Using MPI

We illustrate the collective communication commands to scatter data and gather results. Point-to-point communication happens via a send and a recv (receive) command.

2.2.1 Scatter and Gather

Consider the addition of 100 numbers on a distributed memory 4-processor computer. For simplicity of coding, we sum the first one hundred positive integers and compute

$$S = \sum_{i=1}^{100} i.$$

A parallel algorithm to sum 100 numbers proceeds in four stages:

1. distribute 100 numbers evenly among the 4 processors;
2. Every processor sums 25 numbers;
3. Collect the 4 sums to the manager node; and
4. Add the 4 sums and print the result.

Scattering an array of 100 number over 4 processors and gathering the partial sums at the 4 processors to the root is displayed in Fig. 2.2.

The scatter and gather are of the collective communication type, as every process in the universe participates in this operation. The MPI commands to scatter and gather are respectively `MPI_Scatter` and `MPI_Gather`.

The specifications of the MPI command to scatter data from one member to all members of a group are described in Table 2.1. The specifications of the MPI command to gather data from all members to one member in a group are listed Table 2.2.

Table 2.1: Arguments of the `MPI_Scatter` command.

MPI_SCATTER(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm)	
sendbuf	address of send buffer
sendcount	number of elements sent to each process
sendtype	data type of send buffer elements
recvbuf	address of receive buffer
recvcount	number of elements in receive buffer
recvtype	data type of receive buffer elements
root	rank of sending process
comm	communicator

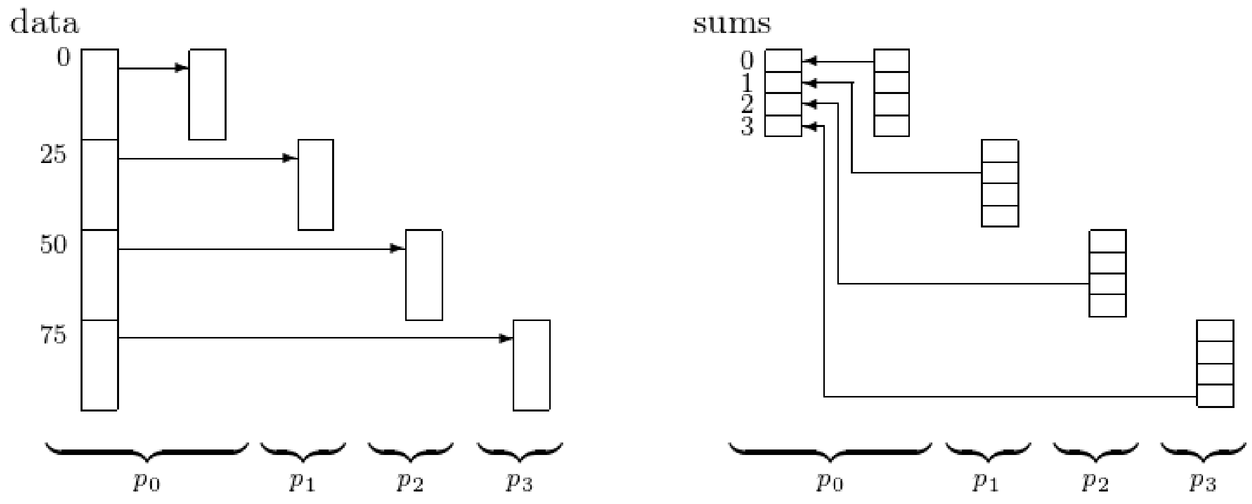


Fig. 2.2: Scattering data and gathering results.

Table 2.2: Arguments of the `MPI_Gather` command.

MPI_GATHER(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm)	
sendbuf	starting address of send buffer
sendcount	number of elements in send buffer
sendtype	data buffer of send buffer elements
recvbuf	address of receive buffer
recvcount	number of elements for any single receive
recvtype	data type of receive buffer elements
root	rank of receiving process
comm	communicator

The code for parallel summation, in the program `parallel_sum.c`, illustrates the scatter and the gather.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define v 1 /* verbose flag, output if 1, no output if 0 */

int main ( int argc, char *argv[] )
{
   int myid,j,*data,tosum[25],sums[4];

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);

   if(myid==0) /* manager allocates and initializes the data */
   {
      data = (int*)calloc(100,sizeof(int));
      for (j=0; j<100; j++) data[j] = j+1;
      if(v>0)
      {
```

(continues on next page)

(continued from previous page)

```
         printf("The data to sum : ");
         for (j=0; j<100; j++) printf(" %d",data[j]); printf("\n");
      }
   }

   MPI_Scatter(data,25,MPI_INT,tosum,25,MPI_INT,0,MPI_COMM_WORLD);

   if(v>0) /* after the scatter, every node has 25 numbers to sum */
   {
      printf("Node %d has numbers to sum :",myid);
      for(j=0; j<25; j++) printf(" %d", tosum[j]);
      printf("\n");
   }
   sums[myid] = 0;
   for(j=0; j<25; j++) sums[myid] += tosum[j];
   if(v>0) printf("Node %d computes the sum %d\n",myid,sums[myid]);

   MPI_Gather(&sums[myid],1,MPI_INT,sums,1,MPI_INT,0,MPI_COMM_WORLD);

   if(myid==0) /* after the gather, sums contains the four sums */
   {
      printf("The four sums : ");
      printf("%d",sums[0]);
      for(j=1; j<4; j++) printf(" + %d", sums[j]);
      for(j=1; j<4; j++) sums[0] += sums[j];
      printf(" = %d, which should be 5050.\n",sums[0]);
   }
   MPI_Finalize();
   return 0;
}
```

2.2.2 Send and Recv

To illustrate point-to-point communication, we consider the problem of squaring numbers in an array. An example of an input sequence is $2, 4, 8, 16, \ldots$ with corresponding output sequence $4, 16, 64, 256, \ldots$. Instead of squaring, we could apply a difficult function $y = f(x)$ to an array of values for x. A session with the parallel code with 4 processes runs as

```
$ mpirun -np 4 /tmp/parallel_square
The data to square :  2 4 8 16
Node 1 will square 4
Node 2 will square 8
Node 3 will square 16
The squared numbers :  4 16 64 256
$
```

Applying a parallel squaring algorithm to square p numbers runs in three stages:

1. The manager sends $p - 1$ numbers $x_1, x_2, \ldots, x_{p-1}$ to workers. Every worker receives: the i-th worker receives x_i in f. The manager copies x_0 to f: $f = x_0$.
2. Every node (manager and all workers) squares f.

3. Every worker sends f to the manager. The manager receives x_i from i-th worker, $i = 1, 2, \ldots, p-1$. The manager copies f to x_0: $x_0 = f$, and prints.

To perform point-to-point communication with MPI are `MPI_Send` and `MPI_Recv`. The syntax for the blocking send operation is in Table 2.3. Table 2.4 explains the blocking receive operation.

Table 2.3: The `MPI_SEND` command.

MPI_SEND(buf,count,datatype,dest,tag,comm)	
buf	initial address of the send buffer
count	number of elements in send buffer
datatype	data type of each send buffer element
dest	rank of destination
tag	message tag
comm	communication

Table 2.4: The `MPI_RECV` command.

MPI_RECV(buf,count,datatype,source,tag,comm,status)	
buf	initial address of the receive buffer
count	number of elements in receive buffer
datatype	data type of each receive buffer element
source	rank of source
tag	message tag
comm	communication
status	status object

Code for a parallel square is below. Every `MPI_Send` is matched by a `MPI_Recv`. Observe that there are two loops in the code. One loop is explicitly executed by the root. The other, implicit loop, is executed by the `mpiexec -n p` command.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

#define v 1 /* verbose flag, output if 1, no output if 0 */
#define tag 100 /* tag for sending a number */

int main ( int argc, char *argv[] )
{
   int p,myid,i,f,*x;
   MPI_Status status;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&p);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   if(myid == 0) /* the manager allocates and initializes x */
   {
      x = (int*)calloc(p,sizeof(int));
      x[0] = 2;
      for (i=1; i<p; i++) x[i] = 2*x[i-1];
      if(v>0)
      {
```

(continues on next page)

(continued from previous page)

```
         printf("The data to square : ");
         for (i=0; i<p; i++) printf(" %d",x[i]); printf("\n");
      }
   }
   if(myid == 0) /* the manager copies x[0] to f */
   {             /* and sends the i-th element to the i-th processor */
      f = x[0];
      for(i=1; i<p; i++) MPI_Send(&x[i],1,MPI_INT,i,tag,MPI_COMM_WORLD);
   }
   else /* every worker receives its f from root */
   {
      MPI_Recv(&f,1,MPI_INT,0,tag,MPI_COMM_WORLD,&status);
      if(v>0) printf("Node %d will square %d\n",myid,f);
   }
   f *= f;       /* every node does the squaring */
   if(myid == 0) /* the manager receives f in x[i] from processor i */
      for(i=1; i<p; i++)
         MPI_Recv(&x[i],1,MPI_INT,i,tag,MPI_COMM_WORLD,&status);
   else     /* every worker sends f to the manager */
      MPI_Send(&f,1,MPI_INT,0,tag,MPI_COMM_WORLD);
   if(myid == 0) /* the manager prints results */
   {
      x[0] = f;
      printf("The squared numbers : ");
      for(i=0; i<p; i++) printf(" %d",x[i]); printf("\n");
   }
   MPI_Finalize();
   return 0;
}
```

2.2.3 Reducing the Communication Cost

The *wall time* refers to the time elapsed on the clock that hangs on the wall, that is: the real time, which measures everything, not just the time the processors were busy. To measure the communication cost, we run our parallel program without any computations. `MPI_Wtime()` returns a double containing the elapsed time in seconds since some arbitrary time in the past. An example of its use is below.

```
double startwtime,endwtime,totalwtime;
startwtime = MPI_Wtime();
/* code to be timed */
endwtime = MPI_Wtime();
totalwtime = endwtime - startwtime;
```

A lot of time in a parallel program can be spent on communication. Broadcasting over 8 processors sequentially takes 8 stages. In a fan out broadcast, the 8 stages are reduced to 3. Fig. 2.3 illustrates the sequential and fan out broadcast.

The story of Fig. 2.3 can be told as follows. Consider the distribution of a pile of 8 pages among 8 people. We can do this in three stages:

1. Person 0 splits the pile keeps 4 pages, hands 4 pages to person 1.
2. Person 0 splits the pile keeps 2 pages, hands 2 pages to person 2.

   Person 1 splits the pile keeps 2 pages, hands 2 pages to person 3.

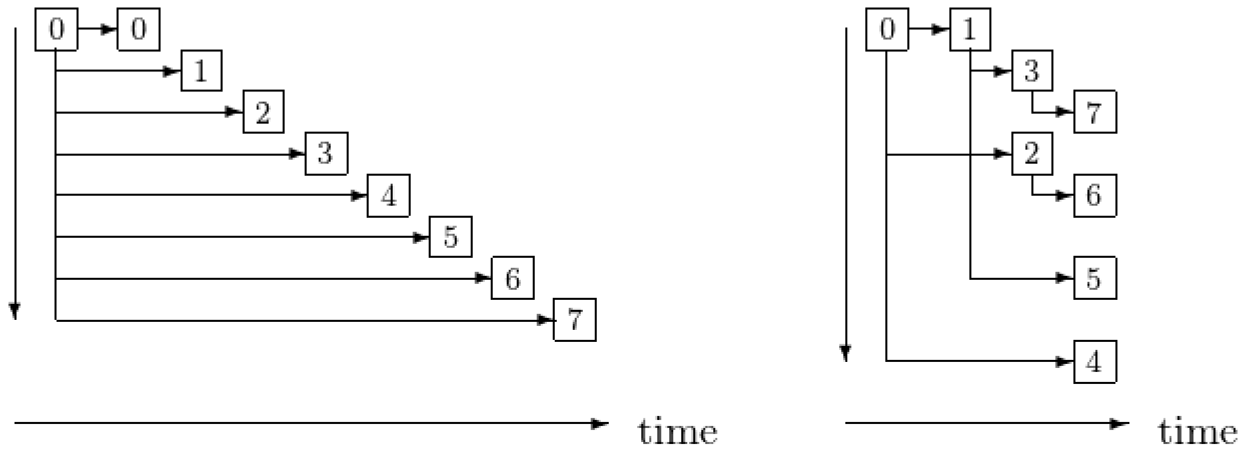


Fig. 2.3: Sequential (left) versus fan out (right) broadcast.

3. Person 0 splits the pile keeps 1 page, hands 1 pages to person 4.

   Person 1 splits the pile keeps 1 page, hands 1 pages to person 5.

   Person 2 splits the pile keeps 1 page, hands 1 pages to person 6.

   Person 3 splits the pile keeps 1 page, hands 1 pages to person 7.

Already from this simple example, we observe the pattern needed to formalize the algorithm. At stage k, processor i communicates with the processor with identification number $i + 2^k$.

The algorithm for fan out broadcast has a short description, shown below.

```
Algorithm: at step k, 2**(k-1) processors have data, and execute:

for j from 0 to 2**(k-1) do
    processor j sends to processor j + 2**(k-1);
    processor j+2**(k-1) receives from processor j.
```

The cost to broadcast of one item is $O(p)$ for a sequential broadcast, is $O(\log_2(p))$ for a fan out broadcast. The cost to scatter n items is $O(p \times n/p)$ for a sequential broadcast, is $O(\log_2(p) \times n/p)$ for a fan out broadcast.

2.2.4 Point-to-Point Communication with MPI for Python

In MPI for Python we call the methods `send` and `recv` for point-to-point communication. Process 0 sends `DATA` to process 1:

```
MPI.COMM_WORLD.send(DATA, dest=1, tag=2)
```

Every `send` must have a matching `recv`. For the script to continue, process 1 must do

```
DATA = MPI.COMM_WORLD.recv(source=0, tag=2)
```

mpi4py uses `pickle` on Python objects. The user can declare the MPI types explicitly.

What appears on screen running the Python script is below.

```
$ mpiexec -n 2 python mpi4py_point2point.py
0 sends {'a': 7, 'b': 3.14} to 1
1 received {'a': 7, 'b': 3.14} from 0
```

The script `mpi4pi_point2point.py` is below.

```
from mpi4py import MPI

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()

if(RANK == 0):
    DATA = {'a': 7, 'b': 3.14}
    COMM.send(DATA, dest=1, tag=11)
    print RANK, 'sends', DATA, 'to 1'
elif(RANK == 1):
    DATA = COMM.recv(source=0, tag=11)
    print RANK, 'received', DATA, 'from 0'
```

With `mpi4py` we can either rely on Python's dynamic typing or declare types explicitly when processing `numpy` arrays. To sum an array of numbers, we distribute the numbers among the processes that compute the sum of a slice. The sums of the slices are sent to process 0 which computes the total sum. The code for the script is `mpi4py_parallel_sum.py` and what appears on screen when the script runs is below.

```
$ mpiexec -n 10 python mpi4py_parallel_sum.py
0 has data [0 1 2 3 4 5 6 7 8 9] sum = 45
2 has data [20 21 22 23 24 25 26 27 28 29] sum = 245
3 has data [30 31 32 33 34 35 36 37 38 39] sum = 345
4 has data [40 41 42 43 44 45 46 47 48 49] sum = 445
5 has data [50 51 52 53 54 55 56 57 58 59] sum = 545
1 has data [10 11 12 13 14 15 16 17 18 19] sum = 145
8 has data [80 81 82 83 84 85 86 87 88 89] sum = 845
9 has data [90 91 92 93 94 95 96 97 98 99] sum = 945
7 has data [70 71 72 73 74 75 76 77 78 79] sum = 745
6 has data [60 61 62 63 64 65 66 67 68 69] sum = 645
total sum = 4950
```

The code for the script `mpi4py_parallel_sum.py` follows.

```
from mpi4py import MPI
import numpy as np

COMM = MPI.COMM_WORLD
RANK = COMM.Get_rank()
SIZE = COMM.Get_size()
N = 10

if(RANK == 0):
    DATA = np.arange(N*SIZE, dtype='i')
    for i in range(1, SIZE):
        SLICE = DATA[i*N:(i+1)*N]
        COMM.Send([SLICE, MPI.INT], dest=i)
    MYDATA = DATA[0:N]
```

(continues on next page)

(continued from previous page)

```
else:
    MYDATA = np.empty(N, dtype='i')
    COMM.Recv([MYDATA, MPI.INT], source=0)

S = sum(MYDATA)
print RANK, 'has data', MYDATA, 'sum =', S

SUMS = np.zeros(SIZE, dtype='i')
if(RANK > 0):
    COMM.send(S, dest=0)
else:
    SUMS[0] = S
    for i in range(1, SIZE):
        SUMS[i] = COMM.recv(source=i)
    print 'total sum =', sum(SUMS)
```

Recall that Python is case sensitive and the distinction between `Send` and `send`, and between `Recv` and `recv` is important. In particular, `COMM.send` and `COMM.recv` have no type declarations, whereas `COMM.Send` and `COMM.Recv` have type declarations.

2.2.5 Point-to-Point Communication with the MPI wrappers in Julia

The code below illustrates the sending and receiving of a dictionary between two nodes, using the MPI wrappers for Julia.

```
using MPI
MPI.Init()

comm = MPI.COMM_WORLD
myid = MPI.Comm_rank(comm)

if myid == 0
    data = Dict('a' => 7, 'b' => 3.14)
    println("$myid sends $data to 1")
    MPI.send(data, comm; dest=1, tag=11)
elseif myid == 1
    data = MPI.recv(comm; source=0, tag=11)
    println("$myid received $data from 0")
end
```

Running in a Terminal Windows, at the command prompt:

```
$ mpiexecjl -n 2 julia mpi_point2point.jl
0 sends Dict{Char, Real}('a' => 7, 'b' => 3.14) to 1
1 received Dict{Char, Real}('a' => 7, 'b' => 3.14) from 0
```

2.2.6 Bibliography

1. S. Byrne, L.C. Wilcox, and V. Churavy: **MPI.jl: Julia bindings for the Message Passing Interface.** In *JuliaCon Proceedings*, 1(1), 68, 2021.
2. L. Dalcin, R. Paz, and M. Storti. **MPI for Python**. *Journal of Parallel and Distributed Computing*, 65:1108–1115, 2005.
3. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference Volume 1, The MPI Core*. Massachusetts Institute of Technology, second edition, 1998.

2.2.7 Exercises

1. Adjust the parallel summation to work for p processors where the dimension n of the array is a multiple of p.
2. Use C or Julia to rewrite the program to sum 100 numbers using `MPI_Send` and `MPI_Recv` instead of `MPI_Scatter` and `MPI_Gather`. In Python, use the collective instead of point-to-point communication.
3. Use C, Python, or Julia to rewrite the program to square p numbers using `MPI_Scatter` and `MPI_Gather`.
4. Show that a hypercube network topology has enough direct connections between processors for a fan out broadcast.

2.3 Pleasingly Parallel Computations

Monte Carlo simulations are an example of a computation for which a parallel computation requires a constant amount of communication. In particular, at the start of the computations, the manager node gives every worker node a seed for its random numbers. At the end of the computations, the workers send their simulation result to the manager node. Between start and end, no communication occurred and we may expect an optimal speedup. This type of computation is called a pleasingly parallel computation.

2.3.1 Ideal Parallel Computations

Suppose we have a disconnected computation graph for 4 processes as in Fig. 2.4.

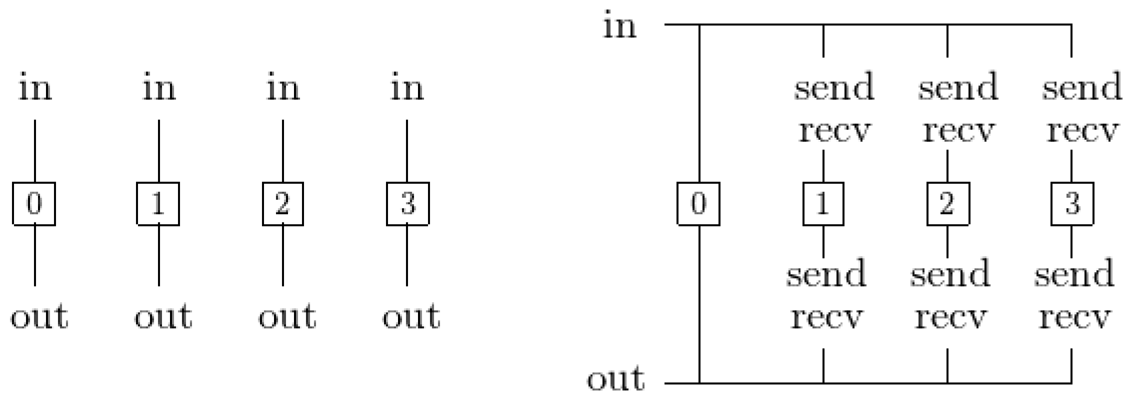


Fig. 2.4: One manager node distributes input data to the compute nodes and collects results from the compute nodes.

Even if the work load is well balanced and all nodes terminate at the same time, we still need to collect the results from each node. Without communication overhead, we hope for an optimal speedup.

Some examples of parallel computations without communication overhead are

1. Geometric transformations of images (section 3.2.1 in textbook): Given an n-by-n matrix of pixels with RGB color encodings, the communication overhead is $O(n^2)$. The cost of a transformation is at most $O(n^2)$. While good for parallel computing, this is *not* good for message passing on distributed memory!
2. The computation of the Mandelbrot set: Every pixel in the set may require up to 255 iterations. Pixels are computed *independently* from each other.
3. Monte Carlo simulations: Every processor generates a *different* sequence of random samples and process samples *independently*.

In this lecture we elaborate on the third example.

2.3.2 Monte Carlo Simulations

We count successes of simulated experiments. We simulate by

- repeatedly drawing samples along a distribution;
- counting the number of successful samples.

By the law of large numbers, the average of the observed successes converges to the expected value or mean, as the number of experiments increases.

Estimating π, the area of the unit disk as $\displaystyle\int_0^1 \sqrt{1-x^2}dx = \frac{\pi}{4}$. Generating random uniformly distributed points with coordinates $(x, y) \in [0, +1] \times [0, +1]$, We count a success when $x^2 + y^2 \leq 1$.

2.3.3 SPRNG: scalable pseudorandom number generator

We only have pseudorandom numbers as true random numbers do not exist on a computer… A multiplicative congruential generator is determined by multiplier a, additive constant c, and a modulus m:

$$x_{n+1} = (ax_n + c) \bmod m, \quad n = 0, 1, \ldots.$$

Assumed: the pooled results of p processors running p copies of a Monte Carlo calculation achieves variance p times smaller.

However, this assumption is true only if the results on each processor are statistically independent. Some problems are that the choice of the seed determines the period, and with repeating sequences, we have lattice effects. The SPRNG: Scalable PseudoRandom Number Generators library is designed to support parallel Monte Carlo applications. A simple use is illustrated below:

```
#include <stdio.h>

#define SIMPLE_SPRNG /* simple interface */
#include "sprng.h"

int main ( void )
{
   printf("hello SPRNG...\n");
   double r = sprng();
   printf("a random double: %.15lf\n",r);
   return 0;
}
```

Because g++ (the gcc c++ compiler) was used to build SPRNG, the makefile is as follows.

```
sprng_hello:
        g++ -I/usr/local/include/sprng sprng_hello.c -lsprng \
            -o /tmp/sprng_hello
```

To see a different random double with each run of the program, we generate a new seed, as follows:

```
#include <stdio.h>
#define SIMPLE_SPRNG
#include "sprng.h"

int main ( void )
{
   printf("SPRNG generates new seed...\n");
   /* make new seed each time program is run   */
   int seed = make_sprng_seed();
   printf("the seed : %d\n", seed);
   /* initialize the stream */
   init_sprng(seed, 0, SPRNG_DEFAULT);
   double r = sprng();
   printf("a random double: %.15f\n", r);
   return 0;
}
```

Consider the estimation of π with SPRNG and MPI. The progam `sprng_estpi.c` is below.

```
#include <stdio.h>
#include <math.h>
#define SIMPLE_SPRNG
#include "sprng.h"
#define PI 3.14159265358979

int main(void)
{
   printf("basic estimation of Pi with SPRNG...\n");
   int seed = make_sprng_seed();
   init_sprng(seed, 0, SPRNG_DEFAULT);

   printf("Give the number of samples : ");
   int n; scanf("%d", &n);

   int i, cnt=0;
   for(i=0; i<n; i++)
   {
      double x = sprng();
      double y = sprng();
      double z = x*x + y*y;
      if(z <= 1.0) cnt++;
   }
   double estimate = (4.0*cnt)/n;
   printf("estimate for Pi : %.15f", estimate);
   printf("  error : %.3e\n", fabs(estimate-PI));

   return 0;
```

(continues on next page)

(continued from previous page)

```
}
```

And some runs:

```
$ /tmp/sprng_estpi
basic estimation of Pi with SPRNG...
Give the number of samples : 100
estimate for Pi : 3.200000000000000  error : 5.841e-02
$ /tmp/sprng_estpi
basic estimation of Pi with SPRNG...
Give the number of samples : 10000
estimate for Pi : 3.131200000000000  error : 1.039e-02
$ /tmp/sprng_estpi
basic estimation of Pi with SPRNG...
Give the number of samples : 10000
estimate for Pi : 3.143600000000000  error : 2.007e-03
$ /tmp/sprng_estpi
basic estimation of Pi with SPRNG...
Give the number of samples : 1000000
estimate for Pi : 3.140704000000000  error : 8.887e-04
```

Using MPI, we run the program `sprng_estpi_mpi.c`:

```
#include <stdio.h>
#include <math.h>
#define SIMPLE_SPRNG
#include "sprng.h"
#define PI 3.14159265358979
#include <mpi.h>

double estimate_pi ( int i, int n );
/* Estimation of pi by process i,
 * using n samples. */

int main ( int argc, char *argv[] )
{
   int id,np;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&np);
   MPI_Comm_rank(MPI_COMM_WORLD,&id);

   int n;
   if(id == 0)
   {
      printf("Reading the number of samples...\n");
      scanf("%d",&n);
   }
   MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);

   double est4pi = estimate_pi(id, n);
   double sum = 0.0;
```

(continues on next page)

(continued from previous page)

```
   MPI_Reduce(&est4pi,&sum,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
   if(id == 0)
   {
      est4pi = sum/np;
      printf("Estimate for Pi : %.15lf",est4pi);
      printf("  error : %.3e\n",fabs(est4pi-PI));
   }
   MPI_Finalize();
   return 0;
}
```

`MPI_Reduce` is a collective communication function to reduce data gathered using some operations, e.g.: addition.

The syntax and arguments are in Table 2.5. The predefined reduction operation `op` we use is `MPI_SUM`.

Table 2.5: Syntax and arguments of MPI_Reduce.

MPI_REDUCE(sendbuf,recvbuf,count,datatype, op,root,comm)	
sendbuf	address of send buffer
recvbuf	address of receive buffer
count	number of elements in send buffer
datatype	data type of elements in send buffer
op	reduce operation
root	rank of root process
comm	communicator

The estimate function we use in `sprng_estpi_mpi.c` is below:

```
double estimate_pi ( int i, int n )
{
   int seed = make_sprng_seed();
   init_sprng(seed, 0, SPRNG_DEFAULT);

   int j,cnt=0;
   for(j=0; j<n; j++)
   {
      double x = sprng();
      double y = sprng();
      double z = x*x + y*y;
      if(z <= 1.0) cnt++;
   }
   double estimate = (4.0*cnt)/n;
   printf("Node %d estimate for Pi : %.15f",i,estimate);
   printf("  error : %.3e\n",fabs(estimate-PI));

   return estimate;
}
```

Because `g++` (the gcc C++ compiler) was used to build SPRNG, we must compile the code with `mpic++`. Therefore, the makefile contains the following lines:

```
sprng_estpi_mpi:
        mpic++ -I/usr/local/include/sprng \
               sprng_estpi_mpi.c -lsprng \
               -o /tmp/sprng_estpi_mpi
```

We end this section with the Mean Time Between Failures (MTBF) problem. The Mean Time Between Failures (MTBF) problem asks for the expected life span of a product made of components. Every component is critical. The multi-component product fails as soon as one of its components fails. For every component we assume that the life span follows a known normal distribution, given by μ and σ. For example, consider 3 components with respective means 11, 12, 13 and corresponding standard deviations 1, 2, 3. Running 100,000 simulations, we compute the average life span of the composite product.

If $f_i(t)$ is the cumulative distribution function of the *i*-th component, then we estimate the triple integral:

$$\mu = \sum_{i=1}^{3} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} t \prod_{j \neq i} (1 - f_j(t)) df_i(t).$$

We need to implement the normal distribution, as specified below.

```
int normal ( double *x, double *y );
/*
 * DESCRIPTION :
 *   Generates two independent normally distributed
 *   variables x and y, along Algorithm P (Knuth Vol 2),
 *   the polar method is due to Box and Muller.
 *
 * ON ENTRY :
 *   x,y     two independent random variables,
 *           uniformly distributed in [0,1].
 *
 * ON RETURN : fail = normal(&x,&y)
 *   fail    if 1, then x and y are outside the unit disk,
 *           if 0, then x and y are inside the unit disk;
 *   x,y     independent normally distributed variables. */
```

The function `normal` can be defined as

```
int normal ( double *x, double *y )
{
   double s;

   *x = 2.0*(*x) - 1.0;
   *y = 2.0*(*y) - 1.0;
   s = (*x)*(*x) + (*y)*(*y);

   if(s >= 1.0)
      return 1;
   else
   {
      double ln_s = log(s);
      double rt_s = sqrt(-2.0*ln_s/s);
      *x = (*x)*rt_s;
      *y = (*y)*rt_s;
      return 0;
```

(continues on next page)

(continued from previous page)

```
   }
}
```

To test the function `normal`, we proceed as follows. For the generated numbers, we compute the average μ and standard deviation σ. We count how many samples are in $[\mu - \sigma, \mu + \sigma]$.

```
$ /tmp/sprng_normal
normal variables with SPRNG ...
a normal random variable : 0.645521197140996
a normal random variable : 0.351776102906080
give number of samples : 1000000
mu = 0.000586448667516, sigma = 1.001564397361179
ratio of #samples in [-1.00,1.00] : 0.6822
generated 1572576 normal random numbers
```

To compile, we may need to link with the math library `-lm`. The header (specification) of the function is

```
double map_to_normal ( double mu, double sigma, double x );
/*
 * DESCRIPTION :
 *   Given a normally distributed number x with mean 0 and
 *   standard deviation 1, returns a normally distributed
 *   number y with mean mu and standard deviation sigma. */
```

The C code (implementation) of the function is

```
double map_to_normal ( double mu, double sigma, double x )
{
   return mu + sigma*x;
}
```

Running `sprng_mtbf` gives

```
$ /tmp/sprng_mtbf
MTBF problem with SPRNG ...
Give number of components : 3
average life span for part 0 ? 11
standard deviation for part 0 ? 1
average life span for part 1 ? 12
standard deviation for part 1 ? 2
average life span for part 2 ? 13
standard deviation for part 2 ? 3
mu[0] = 11.000  sigma[0] = 1.000
mu[1] = 12.000  sigma[1] = 2.000
mu[2] = 13.000  sigma[2] = 3.000
Give number of simulations : 100000
expected life span : 10.115057028769346
```

The header of the `mtbf` function is below

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

(continues on next page)

(continued from previous page)

```
#define SIMPLE_SPRNG
#include "sprng.h"

double mtbf ( int n, int m, double *mu, double *sigma );
/*
 * DESCRIPTION :
 *   Returns the expected life span of a composite product
 *   of m parts, where part i has expected life span mu[i],
 *   with standard deviation sigma[i],
 *   using n simulations. */
```

The `mtbf` function is implemented as

```
double mtbf ( int n, int m, double *mu, double *sigma )
{
   int i,cnt=0;
   double s[n+1];
   do
   {
      normald(mu[0],sigma[0],&s[cnt],&s[cnt+1]);
      for(i=1; i<m; i++)
      {
         double x,y;
         normald(mu[i],sigma[i],&x,&y);
         s[cnt] = min(s[cnt],x);
         s[cnt+1] = min(s[cnt+1],y);
      }
      cnt = cnt + 2;
   } while (cnt < n);

   double sum = 0.0;
   for(i=0; i<cnt; i++) sum = sum + s[i];
   return sum/cnt;
}
```

2.3.4 Bibliography

1. S.L. Anderson. Random number generators on vector supercomputers and other advanced architectures. *SIAM Review* 32(2): 221–251, 1990.
2. D.E. Knuth. *The Art of Computer Programming. Volume 2. Seminumerical Algorithms*. Third Edition. Addison-Wesley, 1997. Chapter Three.
3. M. Mascagni and A. Srinivasan. Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software* 26(3): 436–461, 2000.

2.3.5 Exercises

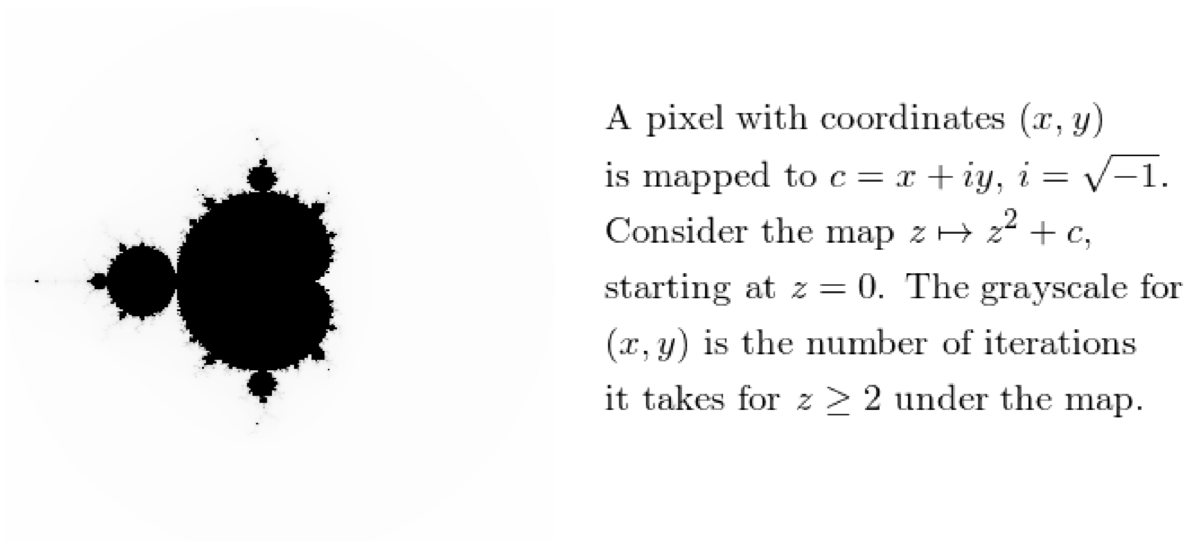
1. Consider the code for the estimation of π. For a fixed choice of the seed, examine the relationship between the error ϵ and the number of samples n. Make a plot relating n to $-\log_{10}(\epsilon)$, for sufficiently many experiments for different values of n so the trend becomes clear.
2. Consider the MPI code for the estimation of π. Fix the seeds so you can experimentally demonstrate the speedup. Execute the code for $p = 2, 4$, and 8 compute nodes.
3. Write a parallel version with MPI of `sprng_mtbf.c`. Verify the correctness by comparison with a sequential run.

2.4 Load Balancing

We distinguish between static and dynamic load balancing, using the computation of the Mandelbrot set as an example. For dynamic load balancing, we encounter the need for nonblocking communications. To check for incoming messages, we use MPI_Iprobe.

2.4.1 the Mandelbrot set

We consider computing the Mandelbrot set, shown in Fig. 2.5 as a grayscale plot.



A pixel with coordinates (x, y) is mapped to $c = x + iy$, $i = \sqrt{-1}$. Consider the map $z \mapsto z^2 + c$, starting at $z = 0$. The grayscale for (x, y) is the number of iterations it takes for $z \geq 2$ under the map.

Fig. 2.5: The Mandelbrot set.

The number n of iterations ranges from 0 to 255. The grayscales are plotted in reverse, as $255 - n$. Grayscales for different pixels are calculated independently. The prototype and definition of the function `iterate` is in the code below. We call `iterate` for all pixels (`x`, `y`), for `x` and `y` ranging over all rows and columns of a pixel matrix. In our plot we compute 5,000 rows and 5,000 columns.

```
int iterate ( double x, double y );
/*
 * Returns the number of iterations for z^2 + c
 * to grow larger than 2, for c = x + i*y,
 * where i = sqrt(-1), starting at z = 0. */
```

(continues on next page)

(continued from previous page)

```
int iterate ( double x, double y )
{
   double wx,wy,v,xx;
   int k = 0;

   wx = 0.0; wy = 0.0; v = 0.0;
   while ((v < 4) && (k++ < 254))
   {
      xx = wx*wx - wy*wy;
      wy = 2.0*wx*wy;
      wx = xx + x;
      wy = wy + y;
      v = wx*wx + wy*wy;
   }
   return k;
}
```

In the code for `iterate` we count 6 multiplications on doubles, 3 additions and 1 subtraction. On a Mac OS X laptop 2.26 Ghz Intel Core 2 Duo, for a 5,000-by-5,000 matrix of pixels:

```
$ time /tmp/mandelbrot
Total number of iterations : 682940922

real    0m15.675s
user    0m14.914s
sys     0m0.163s
```

The program performed $682,940,922 \times 10$ flops in 15 seconds or 455,293,948 flops per second. Turning on full optimization and the time drops from 15 to 9 seconds. After compilation with -O3, the program performed 758,823,246 flops per second.

```
$ make mandelbrot_opt
gcc -O3 -o /tmp/mandelbrot_opt mandelbrot.c

$ time /tmp/mandelbrot_opt
Total number of iterations : 682940922

real    0m9.846s
user    0m9.093s
sys     0m0.163s
```

The input parameters of the program define the intervals $[a,b]$ for x and $[c,d]$ for y, as $(x,y) \in [a,b] \times [c,d]$, e.g.: $[a,b] = [-2,+2] = [c,d]$; The number n of rows (and columns) in pixel matrix determines the resolution of the image and the spacing between points: $\delta x = (b-a)/(n-1)$, $\delta y = (d-c)/(n-1)$. The output is a postscript file, which is a standard format, direct to print or view, and allows for batch processing in an environment without visualization capabilities.

2.4.2 Granularity

Definition of Grain

A *grain* is a sequence of computational steps for sequential execution on a single processor.

Depending on the grain size, we distinguish between

- small grain size: *fine granularity*,
- large grain size: *coarse granularity*.

There is a tradeoff to make:

- Coarse granularity has little communication overhead, but may limit the amount of parallelism; while
- fine granularity promotes parallelism, but may lead to an excessive communication overhead.

2.4.3 Static Work Load Assignment

Static work load assignment means that the decision which pixels are computed by which processor is fixed *in advance* (before the execution of the program) by some algorithm. For the granularity in the communcation, we have two extremes:

1. Matrix of grayscales is divided up into p equal parts and each processor computes part of the matrix. For example: 5,000 rows among 5 processors, each processor takes 1,000 rows. The communication happens after all calculations are done, at the end all processors send their big submatrix to root node.
2. Matrix of grayscales is distributed pixel-by-pixel. Entry (i, j) of the n-by-n matrix is computed by processor with label $(i \times n + j) \bmod p$. The communication is completely interlaced with all computation.

In choosing the granularity between the two extremes:

1. Problem with all communication at end: Total cost = computational cost + communication cost. The communication cost is not interlaced with the computation.
2. Problem with pixel-by-pixel distribution: To compute the grayscale of one pixel requires at most 255 iterations, but may finish much sooner. Even in the most expensive case, processors may be mostly busy handling send/recv operations.

As compromise between the two extremes, we distribute the work load along the rows. Row i is computed by node $1 + (i \bmod (p - 1))$. THe root node 0 distributes row indices and collects the computed rows.

2.4.4 Static work load assignment with MPI

Consider a manager/worker algorithm for static load assignment: Given n jobs to be completed by p processors, $n \gg p$. Processor 0 is in charge of

1. distributing the jobs among the $p - 1$ compute nodes; and
2. collecting the results from the $p - 1$ compute nodes.

Assuming n is a multiple of $p - 1$, let $k = n/(p - 1)$.

The manager executes the following algorithm:

```
for i from 1 to k do
    for j from 1 to p-1 do
        send the next job to compute node j;
    for j from 1 to p-1 do
        receive result from compute node j.
```

The run of an example program is illustrated by what is printed on screen:

```
$ mpirun -np 3 /tmp/static_loaddist
reading the #jobs per compute node...
1
sending 0 to 1
sending 1 to 2
node 1 received 0
-> 1 computes b
node 1 sends b
node 2 received 1
-> 2 computes c
node 2 sends c
received b from 1
received c from 2
sending -1 to 1
sending -1 to 2
The result : bc
node 2 received -1
node 1 received -1
$
```

The main program is below, followed by the code for the worker and for the manager.

```
int main ( int argc, char *argv[] )
{
   int i,p;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&p);
   MPI_Comm_rank(MPI_COMM_WORLD,&i);
   if(i != 0)
      worker(i);
   else
   {
      printf("reading the #jobs per compute node...\n");
      int nbjobs; scanf("%d",&nbjobs);
      manager(p,nbjobs*(p-1));
   }
   MPI_Finalize();
   return 0;
}
```

Following is the code for each worker.

```
int worker ( int i )
{
   int myjob;
```

(continues on next page)

(continued from previous page)

```
   MPI_Status status;
   do
   {
      MPI_Recv(&myjob,1,MPI_INT,0,tag,
               MPI_COMM_WORLD,&status);
      if(v == 1) printf("node %d received %d\n",i,myjob);
      if(myjob == -1) break;
      char c = 'a' + ((char)i);
      if(v == 1) printf("-> %d computes %c\n",i,c);
      if(v == 1) printf("node %d sends %c\n",i,c);
      MPI_Send(&c,1,MPI_CHAR,0,tag,MPI_COMM_WORLD);
   }
   while(myjob != -1);
   return 0;
}
```

Following is the code for the manager.

```
int manager ( int p, int n )
{
   char result[n+1];
   int job = -1;
   int j;
   do
   {
      for(j=1; j<p; j++) /* distribute jobs */
      {
         if(++job >= n) break;
         int d = 1 + (job % (p-1));
         if(v == 1) printf("sending %d to %d\n",job,d);
         MPI_Send(&job,1,MPI_INT,d,tag,MPI_COMM_WORLD);
      }
      if(job >= n) break;
      for(j=1; j<p; j++) /* collect results */
      {
         char c;
         MPI_Status status;
         MPI_Recv(&c,1,MPI_CHAR,j,tag,MPI_COMM_WORLD,&status);
         if(v == 1) printf("received %c from %d\n",c,j);
         result[job-p+1+j] = c;
      }
   } while (job < n);

   job = -1;
   for(j=1; j < p; j++)  /* termination signal is -1 */
   {
      if(v==1) printf("sending -1 to %d\n",j);
      MPI_Send(&job,1,MPI_INT,j,tag,MPI_COMM_WORLD);
   }
   result[n] = '\0';
   printf("The result : %s\n",result);
   return 0;
```

(continues on next page)

(continued from previous page)

```
}
```

2.4.5 an implementation with mpi4py

The specifications of the Python functions in the problem are listed below.

```
from mpi4py import MPI
COMM = MPI.COMM_WORLD

def manager(npr, njobs, verbose=True):
    """
    The manager distributes njobs jobs to npr-1
    workers and prints the received results.
    The njobs must be a multiple of npr-1.
    """

def worker(i, verbose=True):
    """
    The i-th worker receives a number.
    The worker terminates if the number is -1,
    otherwise it sends to the manager the
    corresponding character following the letter 'a'.
    """
```

The main program is defined next.

```
def main(verbose=True):
    """
    Runs a manager/worker static load distribution.
    """
    rank = COMM.Get_rank()
    size = COMM.Get_size()
    if rank > 0:
        worker(rank, verbose)
    else:
        manager(size, size*(size-1), verbose)
```

Each workers receives a job and sends back a character corresponding to the received number.

```
def worker(i, verbose=True):
    if verbose:
        print('Hello from worker', i)
    while True:
        nbr = COMM.recv(source=0, tag=11)
        if verbose:
            print('-> worker', i, 'received', nbr)
        if nbr == -1:
            break
        chrnbr = chr(ord('a') + nbr)
        if verbose:
            print('-> worker', i, 'computed', chrnbr)
        COMM.send(chrnbr, dest=0, tag=11)
```

The manager distributes jobs, as defined below.

```python
def manager(npr, njobs, verbose=True):
    if verbose:
        print('Manager distributes', njobs,
              'jobs to', npr-1, 'workers')
    result = ''
    jobcnt = 0
    while jobcnt < njobs:
        for i in range(1, npr):
            jobcnt = jobcnt + 1
            nbr = 1 + (jobcnt % (npr-1))
            if verbose:
                print('-> manager sends job', jobcnt,
                      'to worker', i)
            COMM.send(nbr, dest=i, tag=11)
        for i in range(1, npr):
            data = COMM.recv(source=i, tag=11)
            if verbose:
                print('-> manager received', data,
                      'from worker', i)
            result = result + data
    for i in range(1, npr):
        if verbose:
            print('-> manager sends -1 to worker', i)
        COMM.send(-1, dest=i, tag=11)
    print('the result :', result)
    print('number of characters :', len(result))
    print('      number of jobs :', njobs)
```

2.4.6 Dynamic Work Load Balancing

Consider scheduling 8 jobs on 2 processors, as in Fig. 2.6.

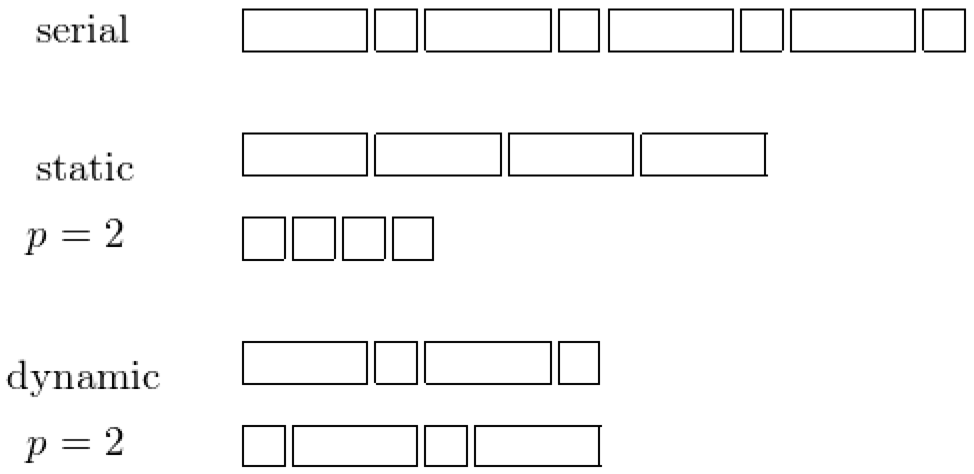


Fig. 2.6: Scheduling 8 jobs on 2 processors. In a worst case scenario, with static job scheduling, all the long jobs end up at one processor, while the short ones at the other, creating an uneven work load.

In scheduling n jobs on p processors, $n \gg p$, node 0 manages the job queue, nodes 1 to $p-1$ are compute nodes. The manager executes the following algorithm:

```
for j from 1 to p-1 do
    send job j-1 to compute node j;
while not all jobs are done do
    if a node is done with a job then
        collect result from node;
        if there is still a job left to do then
            send next job to node;
        else send termination signal.
```

To check for incoming messages, the nonblocking (or Immediate) MPI command is explained in Table 2.6.

Table 2.6: Syntax and arguments of MPI_Iprobe.

MPI_Iprobe(source,tag,comm,flag,status)	
source	rank of source or MPI_ANY_SOURCE
tag	message tag or MPI_ANY_TAG
comm	communicator
flag	address of logical variable
status	status object

If `flag` is true on return, then `status` contains the rank of the source of the message and can be received. The manager starts with distributing the first $p-1$ jobs, as shown below.

```
int manager ( int p, int n )
{
   char result[n+1];
   int j;

   for(j=1; j<p; j++) /* distribute first jobs */
   {
      if(v == 1) printf("sending %d to %d\n",j-1,j);
      MPI_Send(&j,1,MPI_INT,j,tag,MPI_COMM_WORLD);
   }
   int done = 0;
   int jobcount = p-1; /* number of jobs distributed */
   do                  /* probe for results */
   {
      int flag;
      MPI_Status status;
      MPI_Iprobe(MPI_ANY_SOURCE,MPI_ANY_TAG,
                 MPI_COMM_WORLD,&flag,&status);
      if(flag == 1)
      {              /* collect result */
         char c;
         j = status.MPI_SOURCE;
         if(v == 1) printf("received message from %d\n",j);
         MPI_Recv(&c,1,MPI_CHAR,j,tag,MPI_COMM_WORLD,&status);
         if(v == 1) printf("received %c from %d\n",c,j);
         result[done++] = c;
         if(v == 1) printf("#jobs done : %d\n",done);
         if(jobcount < n) /* send the next job */
```

(continues on next page)

(continued from previous page)

```
         {
            if(v == 1) printf("sending %d to %d\n",jobcount,j);
            jobcount = jobcount + 1;
            MPI_Send(&jobcount,1,MPI_INT,j,tag,MPI_COMM_WORLD);
         }
         else  /* send -1 to signal termination */
         {
            if(v == 1) printf("sending -1 to %d\n",j);
            flag = -1;
            MPI_Send(&flag,1,MPI_INT,j,tag,MPI_COMM_WORLD);
         }
      }
   } while (done < n);
   result[done] = '\0';
   printf("The result : %s\n",result);
   return 0;
}
```

The code for the worker is the same as in the static work load distribution, see the function `worker` above. To make the simulation of the dynamic load balancing more realistic, the code for the worker could be modified with a call to the `sleep` function with as argument a random number of seconds.

2.4.7 probing in Python and Julia

The probing is available in mpi4py as

```
iprobe(comm, source=ANY_SOURCE, tag=ANY_TAG, status=None)
```

and in MPI.jl, the probing is done as

```
ismessage, (status|nothing) = Iprobe(src::Integer, tag::Integer, comm::Comm)
```

To illustrate dynamic load balancing in a Julia program, the specifications are listed below.

```
using MPI
MPI.Init()
COMM = MPI.COMM_WORLD

"""
    function manager(p::Int, n::Int, verbose::Bool=true)

distributes n jobs to p-1 workers
and prints the received results.
Assumed is that n >= p-1.
"""

"""
    function worker(i::Int, verbose::Bool=true)

The i-th worker receives a number.
The worker terminates if the number is -1,
otherwise it sends to the manager
```

(continues on next page)

(continued from previous page)

```
the corresponding character following 'a'.
"""
```

The main program is defined next.

```
"""
    function main(verbose::Bool=true)

runs a manager/worker dynamic load distribution.
"""
function main(verbose::Bool=true)
    myid = MPI.Comm_rank(COMM)
    size = MPI.Comm_size(COMM)
    if myid == 0
        print("Give the number of jobs : ")
        line = readline(stdin)
        njobs = parse(Int, line)
    end
    MPI.Barrier(COMM)
    if myid == 0
        manager(size, njobs)
    else
        worker(myid)
    end
    MPI.Barrier(COMM)
end
```

The function which defines the actions of each worker is given below.

```
function worker(i::Int, verbose::Bool=true)
    println("Worker ", i, " says hello.")
    while true
        nbr = MPI.recv(COMM; source=0, tag=11)
        println("-> worker ", i, " received ", nbr)
        if nbr == -1
            break
        end
        chrnbr = Char(Int('a') + nbr)
        MPI.send(chrnbr, COMM; dest=0, tag=11)
    end
end
```

The manager sends the first jobs and then enters a loop to send the next jobs. Observe the use of `Iprobe`.

```
function manager(p::Int, n::Int, verbose::Bool=true)
    if verbose
        println("Manager distributes ", n,
                " jobs to ", p-1, " workers ...")
    end
    result = ""
    for j=1:p-1
        println("-> manager sends job ", j,
                " to worker ", j)
```

(continues on next page)

(continued from previous page)

```
        MPI.send(j, COMM; dest=j, tag=11)
    end
    jobcnt = p-1 # sent already p-1 jobs
    done = 0     # counts workers that are done
    while done < p-1
        for i=1:p-1
            messageSent = MPI.Iprobe(COMM; source=i)
            if messageSent
                data = MPI.recv(COMM; source=i)
                println("-> manager received ", data,
                        " from ", i)
            result = string(result, data)
            jobcnt = jobcnt + 1
            if jobcnt > n
                MPI.send(-1, COMM; dest=i, tag=11)
                done = done + 1
            else
                nbr = 1 + (jobcnt % p)
                MPI.send(nbr, COMM; dest=i, tag=11)
            end
        end
    end
end
println("result : ", result)
println("number of characters : ", length(result))
println("      number of jobs : ", n)
```

2.4.8 Scalability

Introducing load balancing we applied the manager/worker model to schedule jobs before executing (static) and during execution (dynamic). This model works well for a modest number of processors. For thousands of processors, one single manager may no longer be capable to obtain good load balancing.

Obtaining an optimal load balancing is NP-complete. The survey paper of Kwok and Ahmad describes several heuristics to statically schedule jobs.

Nearest neighbor load balancing methods iteratively strive to obtain a global optimal work load distribution. Among the deterministic algorithms are diffusion, dimension exchange, and the gradient model. The diffusion method was modeled by Cybenko using linear system theories. Taking into account the topologies of the networks, results from graph theory are applied in the convergence analysis, as explained in the book of Xu and Lau.

2.4.9 Bibliography

1. Selim G. Akl. **Superlinear performance in real-time parallel computation.** *The Journal of Supercomputing* 29(1):89–111, 2004.
2. George Cybenko. **Dynamic Load Balancing for Distributed Memory Processors**. *Journal of Parallel and Distributed Computing* 7, 279-301, 1989.
3. Yu-Kwong Kwok and Ishfaq Ahmad. **Static scheduling algorithms for allocating directed task graphs to multiprocessor.** *ACM Computing Surveys*, 31(4):406–469, 1999.

4. C. McCreary and H. Gill. **Automatic determination of grain size for efficient parallel processing.** *Communications of the ACM*, 32(9):1073–1078, 1989.

5. Chengzhong Xu and Francis C.M. Lau. *Load Balancing in Parallel Computers. Theory and Practice.* Kluwer Academic Publishers, 1997.

2.4.10 Exercises

1. Apply the manager/worker algorithm for static load assignment to the computation of the Mandelbrot set. What is the speedup for 2, 4, and 8 compute nodes? To examine the work load of every worker, use an array to store the total number of iterations computed by every worker.
2. Apply the manager/worker algorithm for dynamic load balancing to the computation of the Mandelbrot set. What is the speedup for 2, 4, and 8 compute nodes? To examine the work load of every worker, use an array to store the total number of iterations computed by every worker.
3. Compare the performance of static load assignment with dynamic load balancing for the Mandelbrot set. Compare both the speedups and the work loads for every worker.

2.5 Handson Supercomputing

This lecture introduces to the practical aspects of using fast workstations and a real supercomputer.

2.5.1 working on a fast workstation

The first workstation is `pascal.math.uic.edu`, a Microway numbersmasher Xeon + Tesla GPU server (2016):

- two 22-core Intel Xeon E5-2699v4 Broadwell at 2.20GHz,
- 256GB of internal memory at 2400MHz,
- NVIDIA Tesla P100 16GB Pascal GPU accelerators, 4.7 TFLOPS (FP64) peak performance.

The newer `ampere.math.uic.edu` is a Microway 2U Xeon + NVIDIA GPU server (2024):

- two 24-core Intel Xeon 5318Y Ice Lake-SP, up to 3.40GHz,
- 256GB of internal memory at 3200MHz,
- NVIDIA Ampere A100 80GB GPU accelerator, 8.6 TFLOPS (FP64) peak performance. TensorCore performance: up to 19.5 TFLOPS (FP64).

Login via `ssh` at a Terminal or PowerShell window. To transfer files use secure copy `scp`.

2.5.2 using a real supercomputer

Access has been granted through `https://access-ci.org`.

The workflow typically includes the following:

1. request access to the cluster
2. login to the cluster
3. user work spaces and directories
4. requesting and running software

5. submitting jobs
6. monitoring a job

2.6 Data Partitioning

To distribute the work load, we distinguish between functional and domain decomposition. To synchronize computations, we can use MPI_Barrier. We consider efficient scatter and gather implementations to fan out data and to fan in results. To overlap the communication with computation, we can use the nonblocking immediate send and receive operations.

2.6.1 functional and domain decomposition

To turn a sequential algorithm into a parallel one, we distinguish between functional and domain decomposition: In a *functional decomposition*, the arithmetical operations are distributed among several processors. The Monte Carlo simulations are example of a functional decomposition. In a *domain decomposition*, the data are distributed among several processors. The Mandelbrot set computation is an example of a domain decomposition. When solving problems, the entire data set is often too large to fit into the memory of one computer. Complete game trees (e.g.: the game of connect-4 or four in a row) consume an exponential amount of memory.

Divide and conquer used to solve problems:

1. break the problem in smaller parts;
2. solve the smaller parts; and
3. assemble the partial solutions.

Often, divide and conquer is applied in a recursive setting where the smallest nontrivial problem is the base case. Sorting algorithms which apply divide and conquer are mergesort and quicksort.

2.6.2 parallel summation

Applying divide and conquer, we sum a sequence of numbers with divide and conquer, as in the following formula.

$$\begin{aligned}\sum_{k=0}^{7} x_k &= (x_0+x_1+x_2+x_3)+(x_4+x_5+x_6+x_7) \\ &= ((x_0+x_1)+(x_2+x_3))+((x_4+x_5)+(x_6+x_7))\end{aligned}$$

The grouping of the summands in pairs is shown in Fig. 2.7.

The size of the problem is *n*, where $S = \sum_{k=0}^{n-1} x_k$. Assume we have 8 processors to make 8 partial sums:

$$\begin{aligned}S &= (S_0+S_1+S_2+S_3)+(S_4+S_5+S_6+S_7) \\ &= ((S_0+S_1)+(S_2+S_3))+((S_4+S_5)+(S_6+S_7))\end{aligned}$$

where $m = (n-1)/8$ and $S_i = \sum_{k=0}^{m} x_{k+im}$ The communication pattern goes along divide and conquer:

1. the numbers x_k are scattered in a *fan out* fashion,
2. summing the partial sums happens in a *fan in* mode.

Fanning out an array of data is shown in Fig. 2.8.

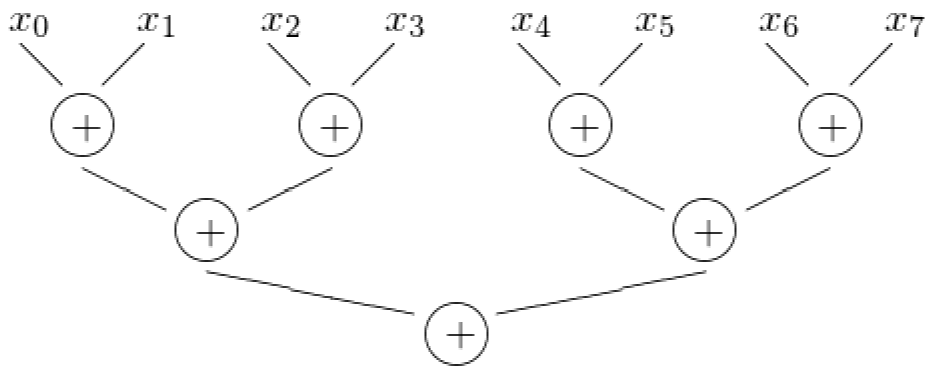
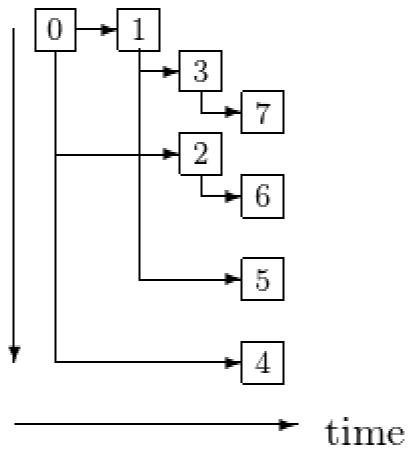


Fig. 2.7: With 4 processors, the summation of 8 numbers in done in 3 steps.



	step			
node	0	1	2	3
0	[0...7]	[0...3]	[0...1]	[0]
1		[4...7]	[4...5]	[4]
2			[2...3]	[2]
3			[6...7]	[6]
4				[1]
5				[5]
6				[3]
7				[7]

Fig. 2.8: Fanning out data.

```
Algorithm: at step k, 2**k processors have data, and execute:
for j from 0 to 2**(k-1) do
    processor j sends data/2**(k+1) to processor j + 2**k;
    processor j+2**k receives data/2**(k+1) from processor j.
```

In fanning out, we use the same array for all nodes, and use only one send/recv statement. Observe the bit patterns in nodes and data locations, as shown in Table 2.7.

Table 2.7: Bit patterns and data locations.

	step				
node	0	1	2	3	data
000	[0…7]	[0…3]	[0…1]	[0]	000
001		[4…7]	[4…5]	[4]	100
010			[2…3]	[2]	010
011			[6…7]	[6]	110
100				[1]	001
101				[5]	101
110				[3]	011
111				[7]	111

At step 3, the node with label in binary expansion $b_2 b_1 b_0$ has data starting at index $b_0 b_1 b_2$.

Fanning out with MPI is illustrated below, with 8 processes.

```
$ mpirun -np 8 /tmp/fan_out_integers
stage 0, d = 1 :
0 sends 40 integers to 1 at 40, start 40
1 received 40 integers from 0 at 40, start 40
stage 1, d = 2 :
0 sends 20 integers to 2 at 20, start 20
2 received 20 integers from 0 at 20, start 20
1 sends 20 integers to 3 at 60, start 60
3 received 20 integers from 1 at 60, start 60
stage 2, d = 4 :
0 sends 10 integers to 4 at 10, start 10
7 received 10 integers from 3 at 70, start 70
3 sends 10 integers to 7 at 70, start 70
4 received 10 integers from 0 at 10, start 10
1 sends 10 integers to 5 at 50, start 50
2 sends 10 integers to 6 at 30, start 30
6 received 10 integers from 2 at 30, start 30
5 received 10 integers from 1 at 50, start 50
data at all nodes :
1 has 10 integers starting at 40 with 40, 41, 42
2 has 10 integers starting at 20 with 20, 21, 22
7 has 10 integers starting at 70 with 70, 71, 72
5 has 10 integers starting at 50 with 50, 51, 52
0 has 10 integers starting at 0 with 0, 1, 2
6 has 10 integers starting at 30 with 30, 31, 32
3 has 10 integers starting at 60 with 60, 61, 62
4 has 10 integers starting at 10 with 10, 11, 12
```

To synchronize across all members of a group we apply `MPI_Barrier(comm)` where `comm` is the communicator

(MPI_COMM_WORLD). MPI_Barrier blocks the caller until all group members have called the statement. The call returns at any process only after all group members have entered the call.

The synchronization in each stage of the fan out must be done because the processors with high identification numbers may only start sending the data in later stages, when they have received the data from processors with low identification numbers.

The computation of the offset is done by the function parity_offset, as used in the program to fan out integers.

```
int parity_offset ( int n, int s );
/* returns the offset of node with label n
 * for data of size s based on parity of n */

int parity_offset ( int n, int s )
{
   int offset = 0;
   s = s/2;
   while(n > 0)
   {
      int d = n % 2;
      if(d > 0) offset += s;
      n = n/2;
      s = s/2;
   }
   return offset;
}
```

The main program to fan out integers is below.

```
int main ( int argc, char *argv[] )
{
   int myid,p,s,i,j,d,b;
   int A[size];

   MPI_Status status;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&p);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);

   if(myid == 0) /* manager initializes */
      for(i=0; i<size; i++) A[i] = i;

   s = size;
   for(i=0,d=1; i<3; i++,d*=2) /* A is fanned out */
   {
      s = s/2;
      if(v>0) MPI_Barrier(MPI_COMM_WORLD);
      if(myid == 0) if(v > 0) printf("stage %d, d = %d :\n",i,d);
      if(v>0) MPI_Barrier(MPI_COMM_WORLD);
      for(j=0; j<d; j++)
      {
         b = parity_offset(myid,size);
         if(myid == j)
         {
            if(v>0) printf("%d sends %d integers to %d at %d, start %d\n",
```

(continues on next page)

(continued from previous page)

```
                    j,s,j+d,b+s,A[b+s]);
            MPI_Send(&A[b+s],s,MPI_INT,j+d,tag,MPI_COMM_WORLD);
         }
         else if(myid == j+d)
         {
            MPI_Recv(&A[b],s,MPI_INT,j,tag,MPI_COMM_WORLD,&status);
            if(v>0)
               printf("%d received %d integers from %d at %d, start %d\n",
                      j+d,s,j,b,A[b]);
         }
      }
   }
   if(v > 0) MPI_Barrier(MPI_COMM_WORLD);
   if(v > 0) if(myid == 0) printf("data at all nodes :\n");
   if(v > 0) MPI_Barrier(MPI_COMM_WORLD);
   printf("%d has %d integers starting at %d with %d, %d, %d\n",
          myid,size/p,b,A[b],A[b+1],A[b+2]);
   MPI_Finalize();
   return 0;
}
```

The same program in Python uses mpi4py and numpy. As the program fans out an array of 80 integers over 8 processors, it must be executed as

```
mpiexec -n 8 python3 fan_out_integers.py
```

The print statements in verbose mode are omitted in the code below.

```
import numpy as np
from mpi4py import MPI
COMM = MPI.COMM_WORLD
SIZE = 80               # size of the problem


def parity_offset(n, s):
    """
    Returns the offset of node with label n
    for data of size s based on parity of n.
    """
    offset = 0
    dim = n
    wsz = s//2
    while dim > 0:
        d = dim % 2
        if d > 0:
            offset += wsz
        dim = dim//2
        wsz = wsz//2
    return offset

def main(verbose=True):
    """
```

(continues on next page)

(continued from previous page)

```
    Fans out 80 integers to 8 processors.
    """
    myid = COMM.Get_rank()
    p = COMM.Get_size()
    # manager initializes, workers allocate space
    if myid == 0:
        data = np.arange(SIZE, dtype='i')
    else:
        data = np.empty(SIZE, dtype='i')
    # the code below has no verbose statements
    d = 1                 # depth
    s = SIZE              # size of a slice
    b = 0                 # begin index
    for i in range(3):    # in 3 steps for 8 nodes
        s = s//2
        for j in range(d):
            b = parity_offset(myid, SIZE);
            if myid == j:
                slice = data[b+s: b+2*s]
                COMM.Send([slice, MPI.INT], dest=j+d)
            elif myid == j+d:
                slice = data[b: b+s]
                COMM.Recv([slice, MPI.INT], source=j)
        d = 2*d
```

The version in Julia with `mpi.jl` is listed next. To synchronize the printing in verbose mode, `MPI_Barrier` is applied.

```
using MPI
MPI.Init()

COMM = MPI.COMM_WORLD
# size of the problem
SIZE = 80

"""
    function parity_offset(n::Int, s::Int)

returns the offset of node with label n
for data of size s based on parity of n.
"""
function parity_offset(n::Int, s::Int)
    offset = 0
    dim = n
    wsz = div(s, 2)
    while dim > 0
        d = dim % 2
        if d > 0
            offset += wsz
        end
        dim = div(dim, 2)
        wsz = div(wsz, 2)
    end
```

(continues on next page)

(continued from previous page)

```
    return offset
end

"""
    function main(verbose=True)

fans out 80 integers to 8 processors.
"""
function main(verbose::Bool=true)
    myid = MPI.Comm_rank(COMM)
    p = MPI.Comm_size(COMM)
    # manager initializes, workers allocate space
    if myid == 0
        data = [i for i=1:SIZE]
    else
        data = zeros(SIZE)
    end
    d = 1
    s = SIZE
    b = 0
    for i=0:2
        s = div(s, 2)
        if verbose
            MPI.Barrier(COMM)
            if myid == 0
                println("stage ", i, " d = ", d)
            end
            MPI.Barrier(COMM)
        end
        for j=0:d-1
            b = parity_offset(myid, SIZE) + 1
            if myid == j
                if verbose
                    MPI.Barrier(COMM)
                    println(j, " sends ", s, " integers to ", j+d,
                            " at ", b+s, " start ", data[b+s],
                            " to ", data[b+2*s-1])
                    MPI.Barrier(COMM)
                end
                slice = data[b+s: b+2*s]
                MPI.send(slice, COMM; dest=j+d, tag=11)
            elseif myid == j+d
                data[b: b+s] = MPI.recv(COMM; source=j, tag=11)
                if verbose
                    MPI.Barrier(COMM)
                    println(j+d, " received ", s, " integers from ", j,
                            " at ", b, " start ", data[b],
                            " to ", data[b+s-1])
                    MPI.Barrier(COMM)
                end
            end
        end
```

(continues on next page)

(continued from previous page)

```
        d = 2*d
    end
    if verbose
        MPI.Barrier(COMM)
        println(myid, " has ", div(SIZE, p), " integers starting at ", b,
                " with ", data[b], data[b+1], " to ", data[b+div(SIZE, p)-1])
        MPI.Barrier(COMM)
    end
end

main()
```

To execute the above program, saved as `fan_out_integers.jl`, do

```
mpiexecjl -n 8 julia fan_out_integers.jl
```

Fanning in the results is illustrated in Fig. 2.9.

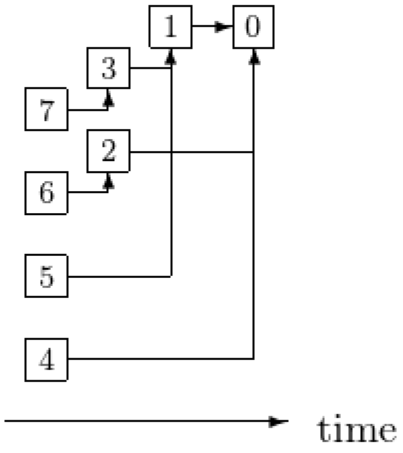


Fig. 2.9: Fanning in results.

```
Algorithm: at step k, 2**k processors send results and execute:
for j from 0 to 2**k-1 do
    processor j+2**k sends the result to processor j;
    processor j receives the result from processor j+2**k.
```

We run the algorithm for decreasing values of k: for example: k=2,1,0.

2.6.3 An Application

Computing π to trillions of digits is a benchmark problem for supercomputers.

One of the remarkable discoveries made by the PSLQ Algorithm (PSLQ = Partial Sum of Least Squares, or integer relation detection) is a simple formula that allows to calculating any binary digit of π without calculating the digits preceding it:

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i} \left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right).$$

BBP stands for Bailey, Borwein and Plouffe. Instead of adding numbers, we concatenate strings.

Some readings on calculations for pi are listed below:

- David H. Bailey, Peter B. Borwein and Simon Plouffe: **On the Rapid Computation of Various Polylogarithmic Constants.** *Mathematics of Computation* 66(218): 903–913, 1997.
- David H. Bailey: **the BBP Algorithm for Pi**. September 17, 2006. <http://crd-legacy.lbl.gov/~dhbailey/dhbpapers/>
- Daisuke Takahashi: **Parallel implementation of multiple-precision arithmetic and 2, 576, 980, 370, 000 decimal digits of pi calculation.** *Parallel Computing* 36(8): 439-448, 2010.

2.6.4 Nonblocking Point-to-Point Communication

The `MPI_SEND` and `MPI_RECV` are *blocking*:

- The sender must wait till the message is received.
- The receiver must wait till the message is sent.

For synchronized computations, this is desirable. To overlap the communication with the computation, we may prefer the use of *nonblocking* communication operations:

- `MPI_ISEND` for the Immediate send; and
- `MPI_IRECV` for the Immediate receive.

The status of the immediate send/receive

- can be queried with `MPI_TEST`; or
- we can wait for its completion with `MPI_WAIT`.

The specification of the `MPI_ISEND` command

```
MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
```

is in Table 2.8.

Table 2.8: Specification of the `MPI_ISEND` command.

parameter	description
buf	address of the send buffer
count	number of elements in send buffer
datatype	datatype of each send buffer element
dest	rank of the destination
tag	message tag
comm	communicator
request	communication request (output)

The sender should not modify any part of the send buffer after a nonblocking send operation is called, until the send completes.

The specification of the `MPI_IRECV` command

```
MPI_IRECV(buf, count, datatype, source, tag, comm, request)
```

is in Table 2.9.

Table 2.9: Specification of the `MPI_IRECV` command.

parameter	description
buf	address of the receive buffer
count	number of elements in receive buffer
datatype	datatype of each receive buffer element
source	rank of source or `MPI_ANY_SOURCE`
tag	message tag or `MPI_ANY_TAG`
comm	communicator
request	communication request (output)

The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.

After the call to `MPI_ISEND` or `MPI_IRECV`, the request can be used to query the status of the communication or wait for its completion.

To wait for the completion of a nonblocking communication:

```
MPI_WAIT (request, status)
```

with specifications in Table 2.10.

Table 2.10: Specification of the `MPI_WAIT` command.

parameter	description
request	communication request
status	status object

To test the status of the communication:

```
MPI_TEST (request, flag, status)
```

with specifications in Table 2.11.

Table 2.11: Specification of the `MPI_TEST` command.

parameter	description
request	communication request
flag	true if operation completed
status	status object

2.6.5 Exercises

1. Adjust the fanning out of the array of integers so it works for any number p of processors where $p = 2^k$ for some k. You may take the size of the array as an integer multiple of p. To illustrate your program, provide screen shots for $p = 8, 16$, and 32.
2. Run the program of the previous exercise on the supercomputer, for $p = 8, 16, 32, 64$, and 128.

   For each run, report the wall clock time.
3. Complete the summation and the fanning in of the partial sums, extending the program. You may leave $p = 8$.

CHAPTER 3

Introduction to Threading and Tasking

To program shared memory parallel computers, we can apply Open MP, the pthreads library in C, the Intel Threading Building blocks in C++, or the tools available in Julia.

3.1 Introduction to OpenMP

Although the focus of this lecture is on OpenMP, we start out by introducing the running example, and by illustrating its multithreaded solution in Julia.

3.1.1 programming shared memory parallel computers

We can approximate π via $\frac{\pi}{4} = \int_0^1 \sqrt{1-x^2}dx$, applying the trapezoidal rule: $\int_a^b f(x)dx \approx \frac{b-a}{2}(f(a)+f(b))$. Using n subintervals of $[a,b]$:

$$\int_a^b f(x)dx \approx \frac{h}{2}(f(a)+f(b)) + h\sum_{i=1}^{n-1} f(a+ih), \quad h = \frac{b-a}{n}.$$

The application of multithreading is illustrated in Fig. 3.1. In the example of Fig. 3.1, the interval $[0,1]$ in four equal parts, for the execution with four threads.

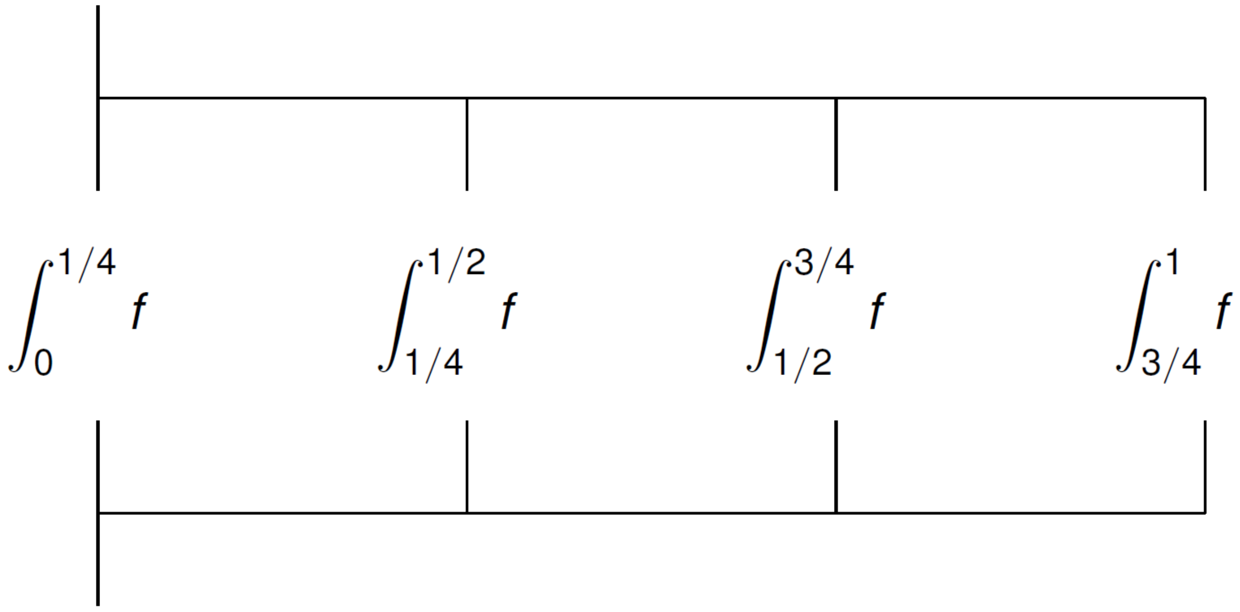


Fig. 3.1: Multithreaded integration with four threads Each thread has its own variables for the limits of the integration interval and for the value of the integral.

3.1.2 multithreading in Julia

The composite trapezoidal rule is defined in the Julia function below.

```
"""
    function traprule(f::Function,
                      a::Float64, b::Float64,
                      n::Int)

returns the composite trapezoidal rule to
approximate the integral of f over [a,b]
using n function evaluations.
"""
function traprule(f::Function,
                  a::Float64, b::Float64,
                  n::Int)
    h = (b-a)/n
    y = (f(a) + f(b))/2
    x = a + h
    for i=1:n-1
        y = y + f(x)
        x = x + h
    end
    return h*y
end
```

The program continues below, with the use of `Threads`.

```
using Printf
using Base.Threads

nt = nthreads()
println("The number of threads : $nt")
subapprox = zeros(nt)

f(x) = sqrt(1 - x^2)
dx = 1/nt
bounds = [i for i=0:dx:1]

timestart = time()
@threads for i=1:nt
    subapprox[i] = traprule(f, bounds[i], bounds[i+1], 1_000_000)
end
approxpi = 4*sum(subapprox)
elapsed = time() - timestart

println("The approximation for Pi : $approxpi")
err = @sprintf("%.3e", pi - approxpi)
println("with error : $err")
println("The elapsed time : $elapsed seconds")
```

Observe that in the parallel loop, every thread using one million function evaluations. Instead of a speedup, we should look for a quality up, and observe that the error of the approximation for pi decreases when using more threads.

To execute the code, at the command prompt in Linux, type

```
JULIA_NUM_THREADS=8 julia mtcomptrap.jl
```

to run the code with 8 threads. As an alternative to setting the environment variable `JULIA_NUM_THREADS`, launch the program as

```
julia -t 8 mtcomptrap.jl
```

to use 8 threads.

To introduce the notion of parallel region, consider the sequence Fig. 3.2. Suppose a_1, a_2, a_3 can be executed in parallel, and similarly for the execution of b_1, b_2, b_3, b_4, and c_1, c_2, c_3.

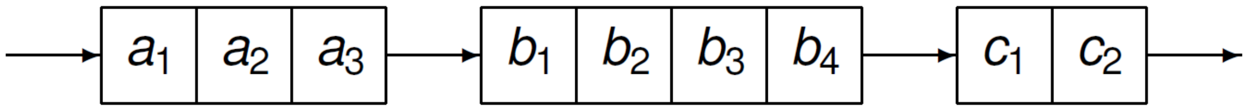


Fig. 3.2: A sequence of three blocks of statements.

The parallel execution of the sequence in Fig. 3.2 is shown in in Fig. 3.3.

The running of a crew of four threads using three parallel regions is illustrated in Fig. 3.4.

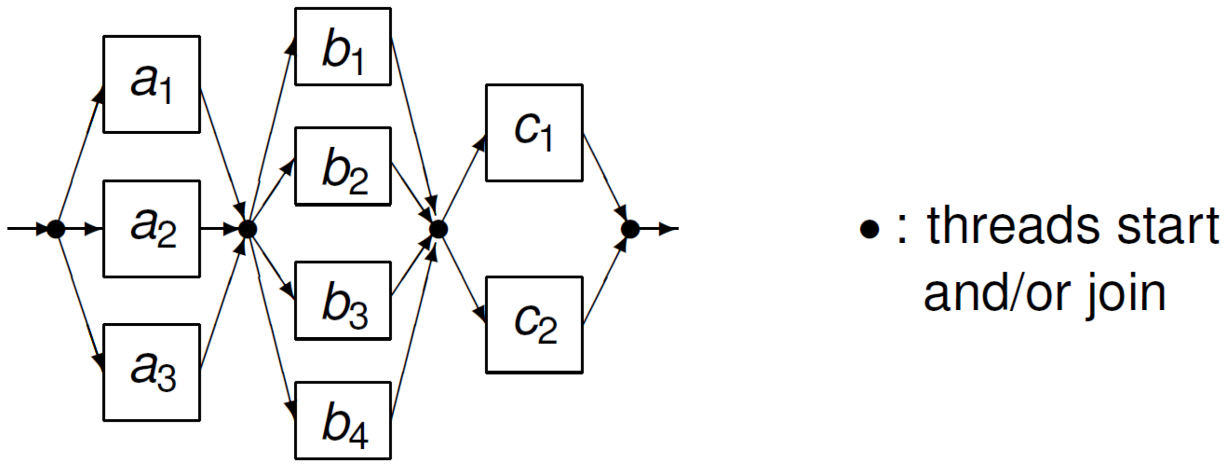


Fig. 3.3: Three parallel regions to execute a sequence of statements.

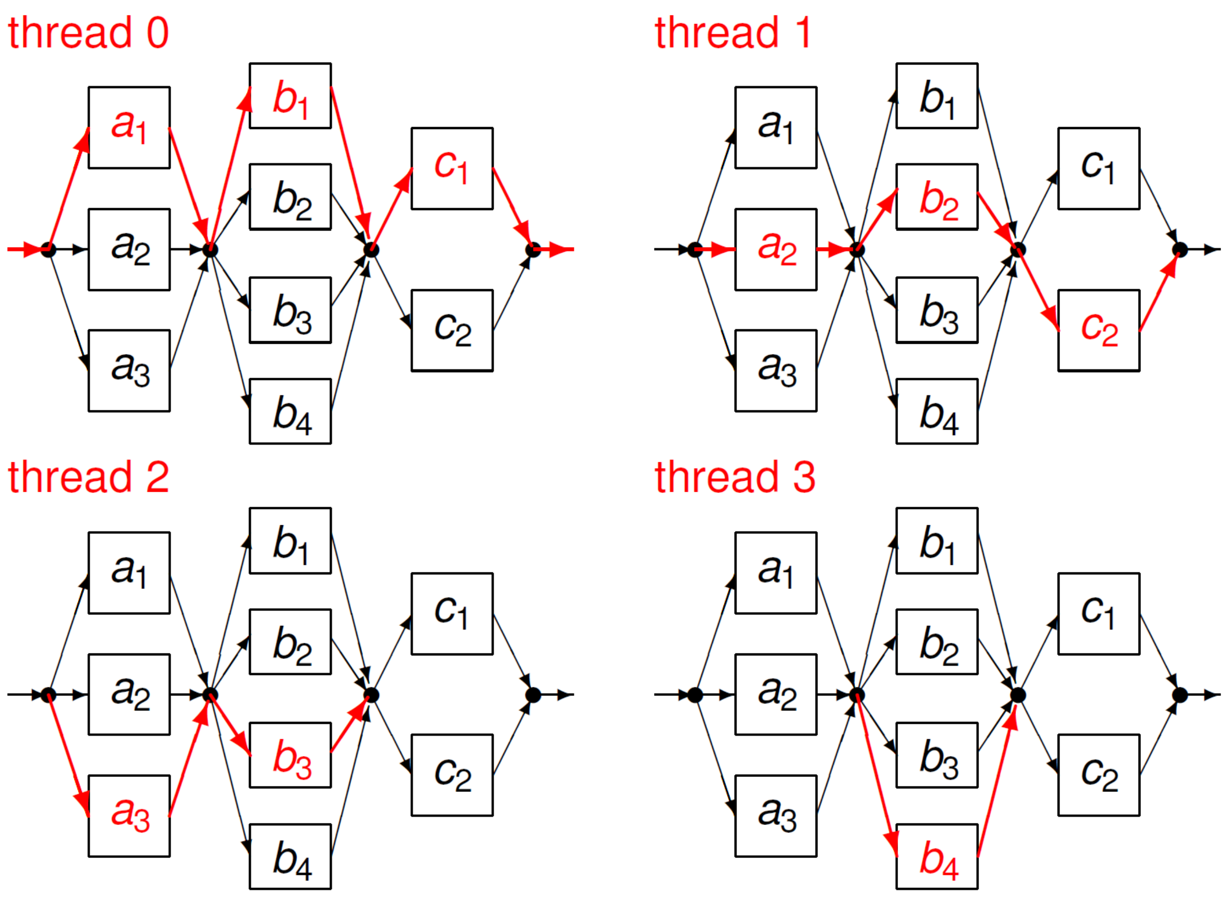


Fig. 3.4: Running a crew of four threads with three parallel regions.

3.1.3 the OpenMP Application Program Interface

OpenMP is an Application Program Interface which originated when a group of parallel computer vendors joined forces to provide a common means for programming a broad range of shared memory parallel computers.

The collection of

1. compiler directives (specified by `#pragma`)
2. library routines (call `gcc -fopenmp`) e.g.: to get the number of threads; and
3. environment variables (e.g.: number of threads, scheduling policies)

defines collectively the specification of the OpenMP API for shared-memory parallelism in C, C++, and Fortran programs. OpenMP offers a set of compiler directives to extend C/C++. The directives can be ignored by a regular C/C++ compiler…

With MPI, we identified processors with processes: in `mpirun -p` as `p` is larger than the available cores, as many as `p` processes are spawned. In comparing a process with a thread, we can consider a process as a completely separate program with its own variables and memory allocation. Threads share the same memory space and global variables between routines. A process can have many threads of execution.

3.1.4 using OpenMP

Our first program with OpenMP is below, `Hello World!`.

```
#include <stdio.h>
#include <omp.h>

int main ( int argc, char *argv[] )
{
   omp_set_num_threads(8);

   #pragma omp parallel
   {
      #pragma omp master
      {
         printf("Hello from the master thread %d!\n", omp_get_thread_num());
      }
      printf("Thread %d says hello.\n", omp_get_thread_num());
   }
   return 0;
}
```

If we save this code in the file `hello_openmp0`, then we compile and run the program as shown below.

```
$ make hello_openmp0
gcc -fopenmp hello_openmp0.c -o /tmp/hello_openmp0

$ /tmp/hello_openmp0
Hello from the master thread 0!
Thread 0 says hello.
Thread 1 says hello.
Thread 2 says hello.
Thread 3 says hello.
Thread 4 says hello.
```

(continues on next page)

(continued from previous page)

```
Thread 5 says hello.
Thread 6 says hello.
Thread 7 says hello.
$
```

Let us go step by step through the `hello_openmp0.c` program and consider first the use of library routines. We compile with `gcc -fopenmp` and put

```
#include <omp.h>
```

at the start of the program. The program `hello_openmp0.c` uses two OpenMP library routines:

1. `void omp_set_num_threads ( int n );`

   sets the number of threads to be used for subsequent parallel regions.

2. `int omp_get_thread_num ( void );`

   returns the thread number, within the current team, of the calling thread.

We use the `parallel` construct as

```
#pragma omp parallel
{
   S1;
   S2;
   ...
   Sm;
}
```

to execute the statements `S1`, `S2`, . . ., `Sm` in parallel.

The master construct specifies a structured block that is executed by the master thread of the team. The `master` construct is illustrated below:

```
#pragma omp parallel
{
   #pragma omp master
   {
      printf("Hello from the master thread %d!\n", omp_get_thread_num());
   }
   /* instructions omitted */
}
```

The single construct specifies that the associated block is executed by only one of the threads in the team (not necessarily the master thread), in the context of its implicit task. The other threads in the team, which do not execute the block, wait at an implicit barrier at the end of the single construct. Extending the `hello_openmp0.c` program with

```
#pragma omp parallel
{
   /* instructions omitted */
   #pragma omp single
   {
      printf("Only one thread %d says more ...\n", omp_get_thread_num());
   }
}
```

3.1.5 Numerical Integration with OpenMP

We consider the composite trapezoidal rule for π via $\frac{\pi}{4} = \int_0^1 \sqrt{1-x^2}dx$. The trapezoidal rule for $f(x)$ over $[a,b]$ is $\int_a^b f(x)dx \approx \frac{b-a}{2}(f(a)+f(b))$.

Using n subintervals of $[a,b]$:

$$\int_a^b f(x)dx \approx \frac{h}{2}(f(a)+f(b)) + h\sum_{i=1}^{n-1} f(a+ih), \quad h = \frac{b-a}{n}.$$

The first argument of the C function for the composite trapezoidal rule is the function that defines the integrand `f`. The complete C program follows.

```
double traprule
 ( double (*f) ( double x ), double a, double b, int n )
{
   int i;
   double h = (b-a)/n;
   double y = (f(a) + f(b))/2.0;
   double x;

   for(i=1,x=a+h; i < n; i++,x+=h) y += f(x);

   return h*y;
}

double integrand ( double x )
{
   return sqrt(1.0 - x*x);
}

int main ( int argc, char *argv[] )
{
   int n = 1000000;
   double my_pi = 0.0;
   double pi,error;

   my_pi = traprule(integrand,0.0,1.0,n);
   my_pi = 4.0*my_pi; pi = 2.0*asin(1.0); error = my_pi-pi;
   printf("Approximation for pi = %.15e with error = %.3e\n", my_pi,error);

   return 0;
}
```

On one core at 3.47 Ghz, running the program evaluates $\sqrt{1-x^2}$ one million times.

```
$ time /tmp/comptrap
Approximation for pi = 3.141592652402481e+00 with error = -1.187e-09

real    0m0.017s
user    0m0.016s
sys     0m0.001s
```

The private clause of parallel is illustrated below.

```
int main ( int argc, char *argv[] )
{
   int i;
   int p = 8;
   int n = 1000000;
   double my_pi = 0.0;
   double a,b,c,h,y,pi,error;

   omp_set_num_threads(p);

   h = 1.0/p;

   #pragma omp parallel private(i,a,b,c)
   /* each thread has its own i,a,b,c */
   {
      i = omp_get_thread_num();
      a = i*h;
      b = (i+1)*h;
      c = traprule(integrand,a,b,n);
      #pragma omp critical
      /* critical section protects shared my_pi */
         my_pi += c;
   }
   my_pi = 4.0*my_pi; pi = 2.0*asin(1.0); error = my_pi-pi;
   printf("Approximation for pi = %.15e with error = %.3e\n",my_pi,error);

   return 0;
}
```

A *private variable* is a variable in a parallel region providing access to a different block of storage for each thread.

```
#pragma omp parallel private(i,a,b,c)
/* each thread has its own i,a,b,c */
{
   i = omp_get_thread_num();
   a = i*h;
   b = (i+1)*h;
   c = traprule(integrand,a,b,n);
```

Thread i integrates from a to b, where h = 1.0/p and stores the result in c. The critical construct restricts execution of the associated structured block in a single thread at a time.

```
#pragma omp critical
/* critical section protects shared my_pi */
   my_pi += c;
```

A thread waits at the beginning of a *critical section* until no threads is executing a critical section. The critical construct enforces exclusive access. In the example, no two threads may increase my_pi simultaneously. Running on 8 cores:

```
$ make comptrap_omp
gcc -fopenmp comptrap_omp.c -o /tmp/comptrap_omp -lm
```

(continues on next page)

(continued from previous page)

```
$ time /tmp/comptrap_omp
Approximation for pi = 3.141592653497455e+00 \
with error = -9.234e-11

real    0m0.014s
user    0m0.089s
sys     0m0.001s
$
```

Compare on one core (`error = -1.187e-09`):

```
real    0m0.017s
user    0m0.016s
sys     0m0.001s
```

The results are summarized in Table 3.1. Summarizing the results:

Table 3.1: Multithreaded Composite Trapezoidal Rule.

	real time	error
1 thread	0.017s	-1.187e-09
8 threads	0.014s	-9.234e-11

In the multithreaded version, every thread uses 1,000,000 subintervals. The program with 8 threads does 8 times more work than the program with 1 thread.

In the book by Wilkinson and Allen, section 8.5 is on OpenMP.

3.1.6 Bibliography

1. Barbara Chapman, Gabriele Jost, and Ruud van der Pas. **Using OpenMP: Portable Shared Memory Parallel Programming**. The MIT Press, 2007.
2. OpenMP Architecture Review Board. **OpenMP Application Program Interface**. Version 4.0, July 2013. Available at <http://www.openmp.org>.

3.1.7 Exercises

0. Read the first chapter of the book **Using OpenMP** by Chapman, Jost, and van der Pas.
1. Modify the `hello world!` program with OpenMP so that the master thread prompts the user for a name which is used in the greeting displayed by thread 5. Note that only one thread, the one with number 5, greets the user.
2. Modify the `hello world!` program so that the number of threads is entered at the command line.
3. Consider the Monte Carlo simulations we have developed with MPI for the estimation of π. Write a version with OpenMP and examine the speedup.
4. Write an OpenMP program to simulate the management of a bank account, with the balance represented by a single shared variable. The program has two threads. Each thread shows the balance to the user and prompts for a debit (decrease) or a deposit (increase). Each thread then updates the balance in a critical section and displays the final the balance to the user.

3.2 The Crew of Threads Model

We illustrate the use of pthreads to implement the work crew model, working to process a sequence of jobs, given in a queue.

3.2.1 Multithreaded Processes

Before we start programming programming shared memory parallel computers, let us specify the relation between threads and processes.

A thread is a single sequential flow within a process. Multiple threads within one process share heap storage, static storage, and code. Each thread has its own registers and stack. Threads share the same single address space and synchronization is needed when threads access same memory locations. A single threaded process is depicted in Fig. 3.5 next to a multithreaded process.

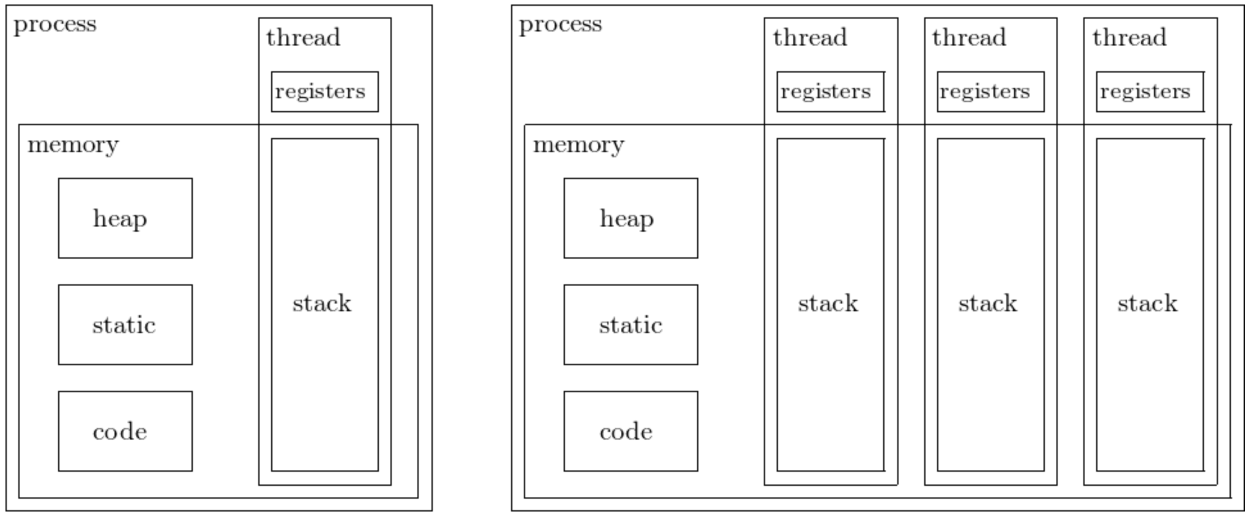


Fig. 3.5: At the left we see a process with one single thread and at the right a multithreaded process.

Threads share the same single address space and synchronization is needed when threads access same memory locations. Multiple threads within one process share heap storage, for dynamic allocation and deallocation; static storage, fixed space; and code. Each thread has its own registers and stack.

The difference between the stack and the heap:

- stack: Memory is allocated by reserving a block of fixed size on top of the stack. Deallocation is adjusting the pointer to the top.
- heap: Memory can be allocated at any time and of any size.

Every call to `calloc` (or `malloc`) and the deallocation with `free` involves the heap. Memory allocation or deallocation should typically happen respectively before or after the running of multiple threads. In a multithreaded process, the memory allocation and deallocation should otherwise occur in a critical section. Code is *thread safe* if its simultaneous execution by multiple threads is correct.

3.2.2 The Work Crew Model

Instead of the manager/worker model where one node is responsible for the distribution of the jobs and the other nodes are workers, with threads we can apply a more collaborative model. We call this the work crew model. Fig. 3.6 illustrates a computation performed by three threads in a work crew model.

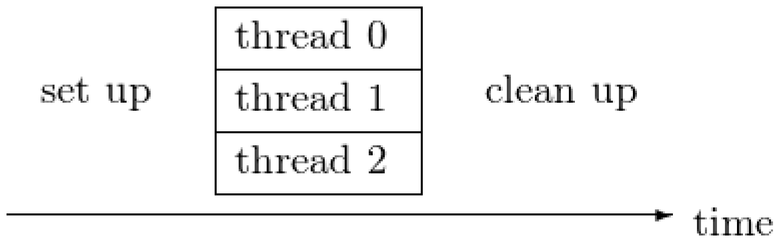


Fig. 3.6: A computation performed by 3 threads in a work crew model.

If the computation is divided into many jobs stored in a queue, then the threads grab the next job, compute the job, and push the result onto another queue or data structure.

We will simulate a work crew model:

- Suppose we have a queue of n jobs.
- Each job has a certain work load (computational cost).
- There are p threads working on the n jobs.

To distribute the jobs among the threads, we can choose between the following:

1. either each worker has its own queue of jobs,
2. or idle workers do the next jobs in the shared queue.

We consider the second type of distributing jobs, which corresponds to dynamic load balancing, because which job gets executed by which thread is determined during execution.

3.2.3 A Crew of Workers with Julia

The output of a simulation with Julia is below.

```
$ julia -t 3 workcrew.jl
The jobs : [4, 5, 5, 2, 4, 6, 3, 6, 6, 4]
the number of threads : 3
Worker 1 is ready.
Worker 3 is ready.
Worker 2 is ready.
Worker 3 spends 5 seconds on job 2 ...
Worker 1 spends 4 seconds on job 1 ...
Worker 2 spends 5 seconds on job 3 ...
Worker 1 spends 2 seconds on job 4 ...
Worker 3 spends 6 seconds on job 6 ...
Worker 2 spends 4 seconds on job 5 ...
Worker 1 spends 3 seconds on job 7 ...
Worker 2 spends 6 seconds on job 8 ...
Worker 1 spends 6 seconds on job 9 ...
Worker 3 spends 4 seconds on job 10 ...
Jobs done : [1, 3, 2, 1, 2, 3, 1, 2, 1, 3]
```

The setup starts with generating a queue of jobs, done by the code below.

```
using Base.Threads

nbr = 10
jobs = rand((2, 3, 4, 5, 6), nbr)
println("The jobs : ", jobs)

nt = nthreads()
println("the number of threads : ", nt)

@threads for i=1:nt
    println("Worker ", threadid(), " is ready.")
end
```

The queue of jobs is shared between all threads. The value of the index to the next job is also shared. The next idle worker will take this value and update it. For the correctness of the program, it is critical that during the update of this value, no other thread accesses the value. Julia provides the mechanism of the atomic variable, which will be illustrated next.

```
jobidx = Atomic{Int}(1)
@threads for i=1:nt
    println("Worker ", threadid(), " is ready.")
    while true
        myjob = atomic_add!(jobidx, 1)
        if myjob > nbr
            break
        end
        println("Thread ", threadid(),
                " spends ", jobs[myjob], " seconds",
                " on job ", myjob, " ...")
        sleep(jobs[myjob])
        jobs[myjob] = threadid()
    end
end
println("Jobs done : ", jobs)
```

The job index is accessed in a *thread safe* manner using an atomic variable, and used as follows:

- The job index is declared and initialized to one: `jobidx = Atomic{Int}(1)`.
- Incrementing the job index goes via `myjob = atomic_add!(jobidx, 1)`, which returns the current value of `jobidx` and increments the value of `jobidx` by one.

The thread safe manner means that accessing the value of the job index can done by only one thread at the same time.

3.2.4 Processing a Job Queue

To define the simulation more precisely, consider that the state of the job queue is defined by

1. the number of jobs,
2. the index to the next job to be executed, and
3. the work to be done by every job.

Variables in a program can be values or references to values, as illustrated in Fig. 3.7, for a queue of 8 jobs. The current status of the queue is determined by the value of `nextjob`, the index to the next job.

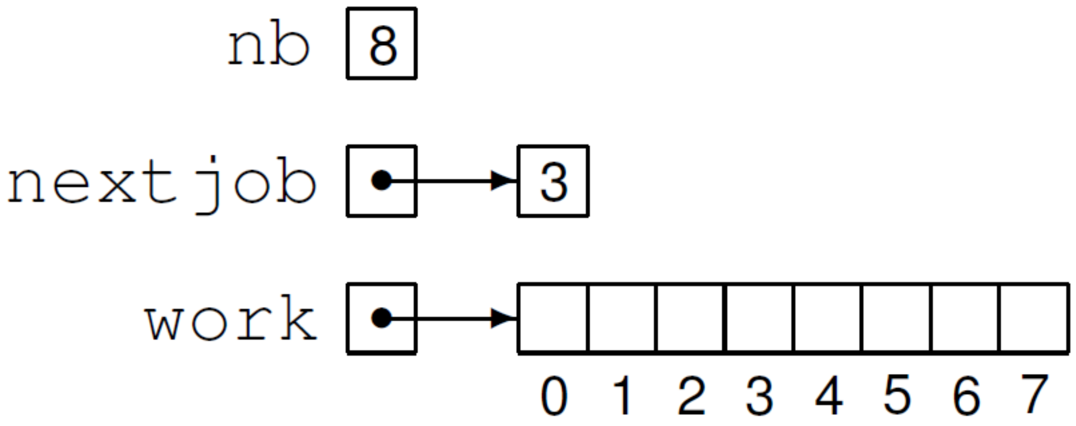


Fig. 3.7: Representing a job queue by the number of jobs, the next job and the work for each job.

In C, the above picture is realized by the statements:

```
int nb = 8;
int *nextjob;
int *work;

*nextjob = 3;
work = (int*)calloc(nb, sizeof(int));
```

To define the sharing of data between threads, we encapsulate the references. Every thread has as values

1. its thread identification number `id`; and
2. the number of jobs `nb`.

The shared data are

1. the reference to the next job; and
2. the cost for every job.

The use of values and pointers is illustrated in Fig. 3.8.

The definition of the data structure in C is shown below.

```
typedef struct
{
   int id;       /* identification label */
   int nb;       /* number of jobs */
   int *nextjob; /* index of next job */
```

(continues on next page)

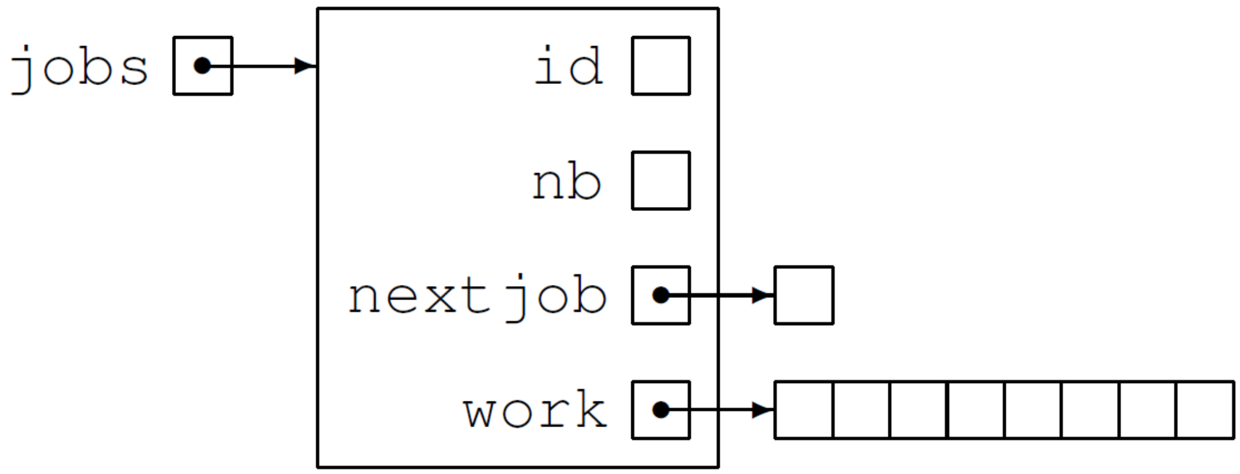


Fig. 3.8: Each thread has as values its identification number and the number of jobs. The shared values are in the memory locations referred to by the pointers `nextjob` and `work`.

(continued from previous page)

```
   int *work;    /* array of nb jobs */
} jobqueue;
```

For example, to share the data of the job queue between 8 threads, consider Fig. 3.9.

Thread i takes on input `q[`i`]`:

1. `q[`i`].id` $= i$,
2. `q[`i`].nb` $= 8$,
3. `*q[`i`].nextjob` $= 3$,
4. `q[`i`].work[`3`]` defines the next job.

The situation for $i = 2$ on the example with 3 threads is illustrated in Fig. 3.10.

The sequential C code to process the job queue is listed below:

```
void do_job ( jobqueue *q )
{
   int jobtodo;
   do
   {
      jobtodo = -1;
      int *j = q->nextjob;
      if(*j < q->nb) jobtodo = (*j)++;
      if(jobtodo == -1) break;
      int w = q->work[jobtodo];
      sleep(w);
   }
   while (jobtodo != -1);
}
```

The `q->nextjob` is equivalent to `(*q).nextjob`. The `jobtodo = (*j)++` dereferences `j`, assigns, and increments.

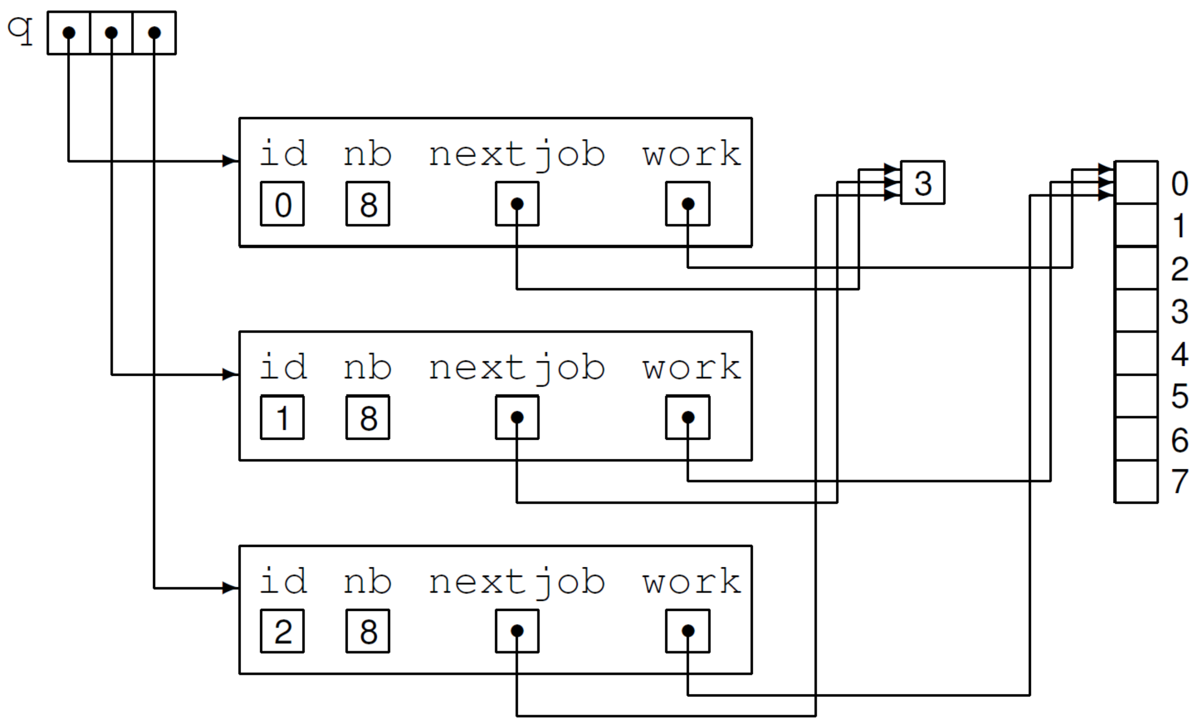


Fig. 3.9: A job queue q to distribute 8 jobs among 3 threads.

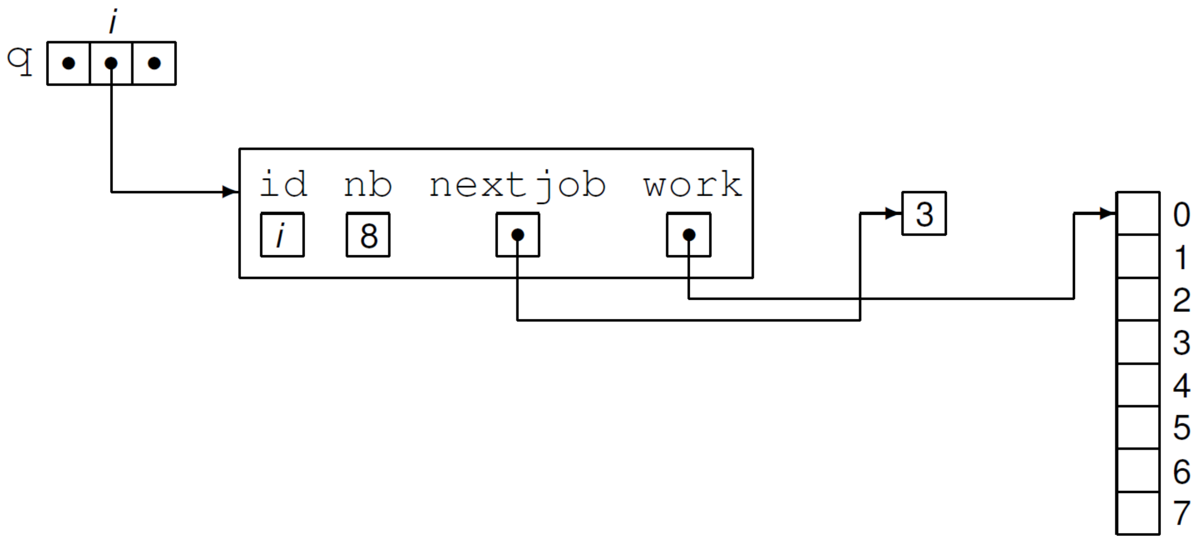


Fig. 3.10: A job queue q accessed by thread $i = 2$.

3.2.5 Processing the Jobs with OpenMP

With OpenMP, we take the sequential code and define parallel regions. Observe the critical section in the code below.

```
void do_job ( jobqueue *q )
{
   int jobtodo;
   do
   {
      jobtodo = -1;
      int *j = q->nextjob;

      #pragma omp critical
      if(*j < q->nb) jobtodo = (*j)++;

      if(jobtodo == -1) break;
      int w = q->work[jobtodo];
      sleep(w);
   }
   while (jobtodo != -1);
}
```

The do_job function is called in the function below, which defines the parallel region.

```
int process_jobqueue ( jobqueue *jobs, int nbt )
{
   jobqueue q[nbt];
   int i;

   for(i=0; i<nbt; i++)
   {
      q[i].nb = jobs->nb;
      q[i].id = i;
      q[i].nextjob = jobs->nextjob;
      q[i].work = jobs->work;
   }
   #pragma omp parallel
   {
      i = omp_get_thread_num();
      do_job(&q[i]);
   }
   return *(jobs->nextjob);
}
```

And then finally, we define the main program.

```
int main ( int argc, char* argv[] )
{
   int njobs,done,nbthreads;
   jobqueue *jobs;

   /* prompt for njobs and nbthreads */

   jobs = make_jobqueue(njobs);
```

(continues on next page)

(continued from previous page)

```
   omp_set_num_threads(nbthreads);

   done = process_jobqueue(jobs,nbthreads);

   printf("done %d jobs\n", jobs->nb);

   return 0;
}
```

This detailed definition of the job queue works as well with Pthreads, as explained in the next section.

3.2.6 The POSIX Threads Programming Interface

For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. POSIX stands for Portable Operating System Interface. Implementations of this POSIX threads programming interface are referred to as POSIX threads, or Pthreads. We can see that `gcc` supports posix threads when we ask for its version number:

```
$ gcc -v
... output omitted ...
Thread model: posix
... output omitted ...
```

In a C program we just insert

```
#include <pthread.h>
```

and compilation may require the switch `-pthread`

```
$ gcc -pthread program.c
```

Our first program with Pthreads is once again a hello world. We define the function each thread executes:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *say_hi ( void *args );
/*
 * Every thread executes say_hi.
 * The argument contains the thread id. */

int main ( int argc, char* argv[] ) { ... }

void *say_hi ( void *args )
{
   int *i = (int*) args;
   printf("hello world from thread %d!\n",*i);
   return NULL;
}
```

Typing `gcc -o /tmp/hello_pthreads hello_pthreads.c` at the command prompt compiles the program and execution goes as follows:

```
$ /tmp/hello_pthreads
How many threads ? 5
creating 5 threads ...
waiting for threads to return ...
hello world from thread 0!
hello world from thread 2!
hello world from thread 3!
hello world from thread 1!
hello world from thread 4!
$
```

Below is the main program:

```
int main ( int argc, char* argv[] )
{
   printf("How many threads ? ");
   int n; scanf("%d",&n);
   {
      pthread_t t[n];
      pthread_attr_t a;
      int i,id[n];
      printf("creating %d threads ...\n",n);
      for(i=0; i<n; i++)
      {
         id[i] = i;
         pthread_attr_init(&a);
         pthread_create(&t[i],&a,say_hi,(void*)&id[i]);
      }
      printf("waiting for threads to return ...\n");
      for(i=0; i<n; i++) pthread_join(t[i],NULL);
   }
   return 0;
}
```

In order to avoid sharing data between threads, To each thread we pass its unique identification label. To `say_hi` we pass the address of the label. With the array `id[n]` we have `n` distinct addresses:

```
pthread_t t[n];
pthread_attr_t a;
int i,id[n];
for(i=0; i<n; i++)
{
   id[i] = i;
   pthread_attr_init(&a);
   pthread_create(&t[i],&a,say_hi,(void*)&id[i]);
}
```

Passing `&i` instead of `&id[i]` gives to every thread the same address, and thus the same identification label. We can summarize the use of Pthreads in 3 steps:

1. Declare threads of type `pthread_t` and attribute(s) of type `pthread_attri_t`.
2. Initialize the attribute `a` as `pthread_attr_init(&a);` and create the threads with `pthreads_create`

providing

1. the address of each thread,
2. the address of an attribute,
3. the function each thread executes, and
4. an address with arguments for the function.

Variables are shared between threads if the same address is passed as argument to the function the thread executes.

3. The creating thread waits for all threads to finish using `pthread_join`.

To process a queue of jobs, we will simulate a work crew model with Pthreads. Suppose we have a queue with *n* jobs. Each job has a certain work load (computational cost). There are *t* threads working on the *n* jobs. A variable `nextjob` is an index to the next job. In a critical section, each thread reads the current value of `nextjob` and increments the value of `nextjob` with one.

The job queue is defined as a structure of constant values and pointers, which allows threads to share data.

```
typedef struct
{
   int id;       /* identification label */
   int nb;       /* number of jobs */
   int *nextjob; /* index of next job */
   int *work;    /* array of nb jobs */
} jobqueue;
```

Every thread gets a job queue with two constants and two adrresses. The constants are the identification number and the number of jobs. The identification number labels the thread and is different for each thread, whereas the number of jobs is the same for each thread. The two addresses are the index of the next job and the work array. Because we pass the addresses to each thread, each thread can change the data the addresses refer to.

The function to generate `n` jobs is defined next.

```
jobqueue *make_jobqueue ( int n )
{
   jobqueue *jobs;

   jobs = (jobqueue*) calloc(1,sizeof(jobqueue));
   jobs->nb = n;
   jobs->nextjob = (int*)calloc(1,sizeof(int));
   *(jobs->nextjob) = 0;
   jobs->work = (int*) calloc(n,sizeof(int));

   int i;
   for(i=0; i<n; i++)
      jobs->work[i] = 1 + rand() % 5;

   return jobs;
}
```

The function to process the jobs by `n` threads is defined below:

```
int process_jobqueue ( jobqueue *jobs, int n )
{
   pthread_t t[n];
```

(continues on next page)

(continued from previous page)

```
   pthread_attr_t a;
   jobqueue q[n];
   int i;
   printf("creating %d threads ...\n",n);
   for(i=0; i<n; i++)
   {
      q[i].nb = jobs->nb; q[i].id = i;
      q[i].nextjob = jobs->nextjob;
      q[i].work = jobs->work;
      pthread_attr_init(&a);
      pthread_create(&t[i],&a,do_job,(void*)&q[i]);
   }
   printf("waiting for threads to return ...\n");
   for(i=0; i<n; i++) pthread_join(t[i],NULL);
   return *(jobs->nextjob);
}
```

3.2.7 Implementing a Critical Section with mutex

Running the processing of the job queue can go as follows:

```
$ /tmp/process_jobqueue
How many jobs ? 4
4 jobs :  3 5 4 4
How many threads ? 2
creating 2 threads ...
waiting for threads to return ...
thread 0 requests lock ...
thread 0 releases lock
thread 1 requests lock ...
thread 1 releases lock
*** thread 1 does job 1 ***
thread 1 sleeps 5 seconds
*** thread 0 does job 0 ***
thread 0 sleeps 3 seconds
thread 0 requests lock ...
thread 0 releases lock
*** thread 0 does job 2 ***
thread 0 sleeps 4 seconds
thread 1 requests lock ...
thread 1 releases lock
*** thread 1 does job 3 ***
thread 1 sleeps 4 seconds
thread 0 requests lock ...
thread 0 releases lock
thread 0 is finished
thread 1 requests lock ...
thread 1 releases lock
thread 1 is finished
done 4 jobs
4 jobs :  0 1 0 1
```

(continues on next page)

(continued from previous page)

```
$
```

There are three steps to use a `mutex` (mutual exclusion):

1. initialization: `pthread_mutex_t L = PTHREAD_MUTEX_INITIALIZER;`
2. request a lock: `pthread_mutex_lock(&L);`
3. release the lock: `pthread_mutex_unlock(&L);`

The main function is defined below:

```
pthread_mutex_t read_lock = PTHREAD_MUTEX_INITIALIZER;

int main ( int argc, char* argv[] )
{
   printf("How many jobs ? ");
   int njobs; scanf("%d",&njobs);
   jobqueue *jobs = make_jobqueue(njobs);
   if(v > 0) write_jobqueue(jobs);

   printf("How many threads ? ");
   int nthreads; scanf("%d",&nthreads);
   int done = process_jobqueue(jobs,nthreads);
   printf("done %d jobs\n",done);
   if(v>0) write_jobqueue(jobs);

   return 0;
}
```

Below is the definition of the function `do_job`:

```
void *do_job ( void *args )
{
   jobqueue *q = (jobqueue*) args;
   int dojob;
   do
   {
      dojob = -1;
      if(v > 0) printf("thread %d requests lock ...\n",q->id);
      pthread_mutex_lock(&read_lock);
      int *j = q->nextjob;
      if(*j < q->nb) dojob = (*j)++;
      if(v>0) printf("thread %d releases lock\n",q->id);
      pthread_mutex_unlock(&read_lock);
      if(dojob == -1) break;
      if(v>0) printf("*** thread %d does job %d ***\n",
                     q->id,dojob);
      int w = q->work[dojob];
      if(v>0) printf("thread %d sleeps %d seconds\n",q->id,w);
      q->work[dojob] = q->id; /* mark job with thread label */
      sleep(w);
   } while (dojob != -1);
```

(continues on next page)

(continued from previous page)

```
    if(v>0) printf("thread %d is finished\n",q->id);

    return NULL;
}
```

Pthreads allow for the finest granularity. Applied to the computation of the Mandelbrot set: One job is the computation of the grayscale of one pixel, in a 5,000-by-5,000 matrix. The next job has number $n = 5,000*i+j$, where $i = n/5,000$ and $j = n \bmod 5,000$.

3.2.8 The Dining Philosophers Problem

A classic example to illustrate the synchronization problem in parallel program is the dining philosophers problem.

The problem setup, rules of the game:

1. Five philosophers are seated at a round table.
2. Each philosopher sits in front of a plate of food.
3. Between each plate is exactly one chop stick.
4. A philosopher thinks, eats, thinks, eats, …
5. To start eating, every philosopher
    1. first picks up the left chop stick, and
    2. then picks up the right chop stick.

Why is there a problem?

The problem of the starving philosophers:

- every philosoper picks up the left chop stick, at the same time,
- there is no right chop stick left, every philosopher waits, …

3.2.9 Bibliography

1. Compaq Computer Corporation. **Guide to the POSIX Threads Library**, April 2001.
2. Mac OS X Developer Library. **Threading Programming Guide**, 2010.

3.2.10 Exercises

1. Modify the `hello world!` program with Pthreads so that the master thread prompts the user for a name which is used in the greeting displayed by thread 5. Note that only one thread, the one with number 5, greets the user.
2. Consider the Monte Carlo simulations we have developed with MPI for the estimation of π. Write a version with Julia, or OpenMP, or Pthreads and examine the speedup.
3. Consider the computation of the Mandelbrot set as implemented in the program `mandelbrot.c` of lecture 7. Write code (with Julia, or OpenMP, or Pthreads) for a work crew model of threads to compute the grayscales. Does the grain size matter? Compare the running time of your program with your MPI implementation.
4. For some number `N`, array `x`, function `f`, consider:

```
#pragma omp parallel
#pragma omp for schedule(dynamic)
 {
    for(i=0; i<N; i++) x[i] = f(i);
 }
```

Define the simulation of the dynamic load balancing with the job queue using `schedule(dynamic)`.

5. Write a simulation for the dining philosophers problem. Could you observe starvation? Explain.

3.3 Tasking with OpenMP

A process can be viewed as a program. A process can have multiple threads of execution.

Task

A *task* provides a unit of work to a thread for execution.

Tasks are much lighter than threads. The main differences between threads and tasks are:

- Starting and terminating a task is much faster than starting and terminating a thread.
- A thread has its own process id and own resources, whereas a task is typically a small routine.

3.3.1 Parallel Recursive Functions

The sequence of Fibonacci numbers F_n are defined as

$$F_0 = 0, \quad F_1 = 1, \quad \text{and for } n > 1 : F_n = F_{n-1} + F_{n-2}.$$

This leads to a natural recursive function.

The recursion generates many function calls. While inefficient to compute F_n, this recursion serves as a parallel pattern.

The parallel version is part of the OpenMP Application Programming Interface Examples. The Fibonacci function with tasking demonstrates the generation of a large number of tasks with one thread. No parallelism will result from this example.

But it is instructive to introduce basic task constructs.

- The `task` construct defines an explicit task.
- The `taskwait` construct synchronizes sibling tasks.
- The `shared` clause of a task construct declares a variable to be shared by tasks.

The start of the program takes the number of threads as a command line argument, or, when that number is omitted, prompts the user for the number of threads:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int fib ( int n );
```

(continues on next page)

(continued from previous page)

```
/* Returns the n-th Fibonacci number,
 * computed recursively with tasking. */

int main ( int argc, char *argv[] )
{
   int n;

   if(argc > 1)
      n = atoi(argv[1]);
   else
   {
      printf("Give n : "); scanf("%d", &n);
   }
   omp_set_num_threads(8);

   #pragma omp parallel
   {
      #pragma omp single
      printf("F(%d) = %d\n",n,fib(n));
   }
   return 0;
}
```

The single construct specifies that the statement is executed by only one thread in the team. In this example, one thread generates many tasks. The definition of a parallel recursive Fibonacci function is below.

```
int fib ( int n )
{
   if(n < 2)
      return n;
   else
   {
      int left,right;    // shared by all tasks

      #pragma omp task shared(left)
      left = fib(n-1);

      #pragma omp task shared(right)
      right = fib(n-2);
                             // synchronize tasks
      #pragma omp taskwait
      return left + right;
   }
}
```

3.3.2 Parallel Recursive Quadrature

The recursive parallel computation of the Fibonacci number serves as a pattern to compute integrals by recursively dividing the integration interval.

Let us apply a numerical integration rule $R(f, a, b, n)$ to $\displaystyle\int_a^b f(x)dx$. The rule $R(f, a, b, n)$ takes on input

- the function f, bounds a, b of $[a, b]$, and
- the number n of function evaluations.

The rule returns and approximation A and an error estimate e.

If e is larger than some tolerance, then

1. $c = (b - a)/2$ is the middle of $[a, b]$,
2. compute $A_1, e_1 = R(f, a, c, n)$,
3. compute $A_2, e_2 = R(f, c, a, n)$,
4. return $A_1 + A_2, e_1 + e_2$.

As a rule, the composite trapezoidal rule is applied recursively. Using n subintervals of $[a,b]$, the rule is

$$R(f, a, b, n) = \frac{h}{2}(f(a) + f(b)) + h\sum_{i=1}^{n-1} f(a + ih), \quad h = \frac{b-a}{n}.$$

In our setup, let $f(x) = e^x$, $[a, b] = [0, 1]$, then $\displaystyle\int_0^1 e^x dx = e - 1$.

We keep n fixed. Let d be the depth of the recursion. The recursion level is ℓ. Pseudo code is below.

$\mathcal{F}(\ell, d, f, a, b, n)$:

```
1. If $\ell = d$ then
2. return $R(f, a, b, n)$
3. else
4. $c = (b - a)/2$
5. return $\mathcal{F}(\ell+1, d, f, a, c, n) + \mathcal{F}(\ell+1, d, f, c, b, n)$.
```

The tree of function calls is shown in Fig. 3.11. The root of the tree is the first call, omitting the value for n, the number of function calls.

At the leaves of the tree, the rule is applied. As all computations are concentrated at the leaves, we expect speedups from a parallel execution.

A recursive parallel integration function with OpenMP is defined below.

```
double rectraprule
 ( int level, int depth,
   double (*f) ( double x ), double a, double b, int n )
{
   if(level == depth)
      return traprule(f,a,b,n);
   else
   {
      double middle = (b - a)/2;
```

(continues on next page)

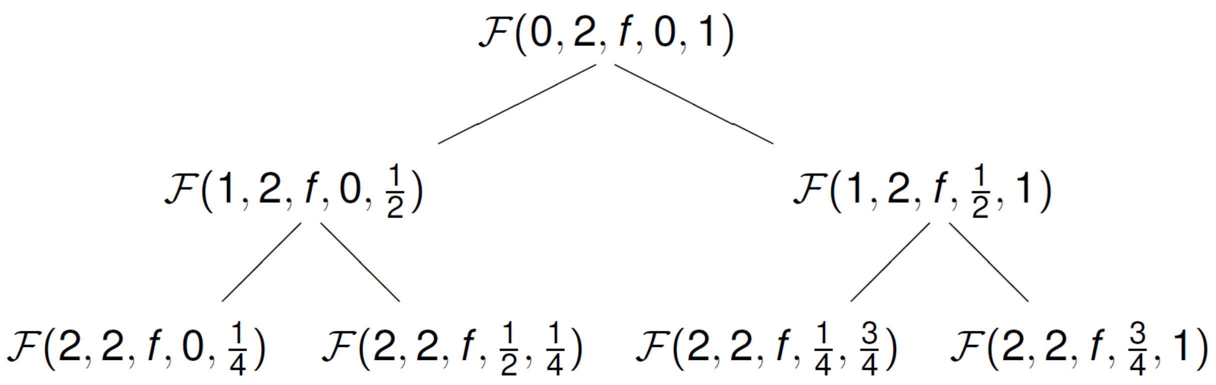

$\mathcal{F}(0, 2, f, 0, 1)$
$\mathcal{F}(1, 2, f, 0, \frac{1}{2})$
$\mathcal{F}(1, 2, f, \frac{1}{2}, 1)$
$\mathcal{F}(2, 2, f, 0, \frac{1}{4})$
$\mathcal{F}(2, 2, f, \frac{1}{2}, \frac{1}{4})$
$\mathcal{F}(2, 2, f, \frac{1}{4}, \frac{3}{4})$
$\mathcal{F}(2, 2, f, \frac{3}{4}, 1)$

Fig. 3.11: A tree of recursive function calls of depth 2.

(continued from previous page)

```
      double left,right;

      #pragma omp task shared(left)
      left = rectraprule(level+1,depth,f,a,middle,n);

      #pragma omp task shared(right)
      right = rectraprule(level+1,depth,f,middle,b,n);

      #pragma omp taskwait
      return left + right;
   }
}
```

Timing the running with 8 threads is shown below.

```
$ time ./comptraprec 200000 10
approximation = 1.7182818284620265e+00
   exp(1) - 1 = 1.7182818284590451e+00, error = 2.98e-12

real    0m3.299s
user    0m3.298s
sys     0m0.001s
$ time ./comptraprecomp 200000 10
approximation = 1.7182818284620265e+00
   exp(1) - 1 = 1.7182818284590451e+00, error = 2.98e-12

real    0m0.743s
user    0m4.003s
sys     0m0.004s
$
```

Observe the speedup, when comparing the wall clock times (`real`).

3.3.3 Bernstein's Conditions

Given a program, which statements can be executed in parallel? Let us do a dependency analysis.

Let u be an operation. Denote:

- $\mathcal{R}(u)$ is the set of memory cells u reads,
- $\mathcal{M}(u)$ is the set of memory cells u modifies.

Two operations u and v are independent if

1. $\mathcal{M}(u) \cap \mathcal{M}(v) = \emptyset$, and
2. $\mathcal{M}(u) \cap \mathcal{R}(v) = \emptyset$, and
3. $\mathcal{R}(u) \cap \mathcal{M}(v) = \emptyset$.

The above conditions are known as *Bernstein's conditions*. Checking Bernstein's conditions is easy for operations on scalars, is more difficult for array accesses, and is almost impossible for pointer dereferencing.

As an example, let `x` be some scalar and consider two statements:

1. u: `x = x + 1`,
2. v: `x = x + 2`.

We see that u and v are independent of each other, because u followed by v or v' followed by u is equivalent to

$$w: \texttt{x = x + 3}.$$

However, execution of u and v happens by a sequence of more elementary instructions:

- u: `r1 = x; r1 += 1; x = r1;`
- v: `r2 = x; r2 += 2; x = r2;`

where `r1` and `r2` are registers. The elementary instructions are no longer independent.

3.3.4 Task Dependencies

With the `depend` clause of OpenMP, the order of execution of tasks can be ordered.

In the `depend` clause, we consider two dependence types:

1. The `in` type: the task depends on the sibling task(s) that generates the item followed by the `in:`.
2. The `out` type: if an item appeared following an `in:` then there should be task with the clause `` out``.

The code below is copied from the OpenMP API Examples section.

```
#include <stdio.h>
#include <omp.h}

int main ( int argc, char *argv[] )
{
   int x = 1;

   #pragma omp parallel
   #pragma omp single
   {
       #pragma omp task shared(x) depend(out: x)
```

(continues on next page)

(continued from previous page)

```
        x = 2;
        #pragma omp task shared(x) depend(in: x)
        printf("x = %d\n", x);
    }
    return 0;
}
```

In the parallel region, the `single` construct indicates that every instruction needs to be execute only once.

- One task assigns 2 to `x`.
- Another task prints the value of `x`.

Without `depend`, tasks could execute in any order, and the program would have a race condition.

Race Condition

A *race condition* occurs in a parallel program execution when two or more threads access a common resource.

The depend clauses force the ordering of the tasks. The example always prints `x = 2`.

3.3.5 Parallel Blocked Matrix Multiplication

Our last example also comes from the OpenMP API Examples. Consider the product of two blocked matrices A with B:

$$\left[\begin{array}{cc} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{array}\right] \left[\begin{array}{cc} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{array}\right] = \left[\begin{array}{cc} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{array}\right].$$

where

$$C_{i,j} = A_{i,1}B_{1,j} + A_{i,2}B_{2,j},$$

for all i and j. The arguments of the depend clauses are blocked matrices.

Matrices are stored as pointers to rows. Allocating a matrix of dimension `dim`:

```
double **A;
int i;

A = (double**)calloc(dim, sizeof(double*));

for(i=0; i<dim; i++) A[i] = (double*)calloc(dim, sizeof(double));
```

Every row `A[i]` is allocated in the loop.

We consider multiplying blocked matrices of random doubles. At the command line, we specify

1. the block size, the size of each block,
2. the number of blocks in every matrix, and
3. the number of threads.

The dimension equals the block size times the number of blocks.

The parallel region:

```
#pragma omp parallel
#pragma omp single
matmatmul(dim,blocksize,A,B,C);
```

One single thread calls the function `matmatmul`. The `matmatmul` generates a large number of tasks. The function `matmatmul` begins as

```
void matmatmul
 ( int N, int BS,
   double **A, double **B, double **C )
{
   int i, j, k, ii, jj, kk;

   for(i=0; i<N; i+=BS)
   {
      for(j=0; j<N; j+=BS)
      {
         for(k=0; k<N; k+=BS)
         {
```

The triple loop computes the block $C_{i,j}$.

Each task has its own indices `ii`, `jj`, and `kk`.

```
#pragma omp task private(ii, jj, kk) \
        depend(in: A[i:BS][k:BS], B[k:BS][j:BS]) \
        depend(inout: C[i:BS][j:BS])
{
   for(ii=i; ii<i+BS; ii++)
      for(jj=j; jj<j+BS; jj++)
         for(kk=k; kk<k+BS; kk++)
            C[ii][jj] = C[ii][jj] + A[ii][kk]*B[kk][jj];
}
```

The `inout` dependence type `C[i:BS][j:BS]` expresses that the dependencies of the update of the block $C_{i,j}$.

Runs with 2 and 4 threads are shown below.

```
$ gcc -fopenmp -O3 -o matmulomp matmulomp.c

$ time ./matmulomp 500 2 2

real    0m0.828s
user    0m1.558s
sys     0m0.020s

$ time ./matmulomp 500 2 4

real    0m0.445s
user    0m1.575s
sys     0m0.017s
$
```

PLASMA (Parallel Linear Algebra Software for Multicore Architectures) is a numerical library intended as a successor to LAPACK for solving problems in dense linear algebra on multicore processors. As indicated in the bibliography section, the PLASMA developers used the OpenMP standard.

3.3.6 Bibliography

1. A. J. Bernstein: **Analysis of Programs for Parallel Processing.** *IEEE Transactions on Electronic Computers* 15(5):757-763, 1966.
2. P. Feautrier: **Bernstein's Conditions**. In *Encycopedia of Parallel Computing*, edited by David Padua, pages 130-133, Springer 2011.
3. C. von Praun: **Race Conditions**. In *Encycopedia of Parallel Computing*, edited by David Padua, pages 1691-1697, Springer 2011.
4. B. Wilkinson and M. Allen: **Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers.** 2nd Edition. Prentice-Hall 2005.
5. A. YarKhan, J. Kurzak, P. Luszczek, J. Dongarra: **Porting the PLASMA Numerical Library to the OpenMP Standard.** *International Journal of Parallel Programming*, May 2016.

3.3.7 Exercises

1. Label the six elementary operations in the example on the Bernstein's conditions as u_1, u_2, u_3, v_1, v_2, v_3.

   Write for each the sets $\mathcal{R}(\cdot)$ and $\mathcal{M}(\cdot)$.

   Based on the dependency analysis, arrange the six instructions for a correct parallel computation.

2. The block size, number of blocks, and number of threads are the three parameters in `matmulomp`.

   Explore experimentally with `matmulomp` the relationship between the number of blocks and the number of threads.

   For which values do you obtain a good speedup?

3.4 Tasking with Julia

Julia is a new programming language for scientific computing designed for performance. The tasking in Julia is inspired by parallel programming systems like Cilk, Intel Threading Building Blocks, and Go.

This lecture is based on a blogpost, of 23 July 2019, `https://julialang.org/blog/2019/07/multithreading` by Jeff Bezanson, Jameson Nash, and Kiran Pamnany, as an early preview of Julia version 1.3.0.

Tasks are units of work, mapped to threads. The next sections mirror the previous lecture on Tasking with OpenMP.

3.4.1 Parallel Recursive Functions

The sequence of Fibonacci numbers F_n are defined as

$$F_0 = 0, \quad F_1 = 1, \quad \text{and for } n > 1 : F_n = F_{n-1} + F_{n-2}.$$

This leads to a natural recursive function.

The recursion generates many function calls. While inefficient to compute F_n, this recursion serves as a parallel pattern.

The Fibonacci function with tasking demonstrates the generation of a large number of tasks with one thread. No parallelism will result from this example.

But it is instructive to introduce basic task constructs.

- With `t = @spawn F()` we start a task `t` to compute `F()`, for some function `F()`.
- The `fetch(t)` waits for `t` to complete and gets its return value.

In the multitasked program to compute the Fibonacci numbers, the number n for the n-th Fibonacci number will be gotten from the command line argument. The Julia program below prints all command line arguments.

```
print(PROGRAM_FILE, " has ", length(ARGS))
println(" arguments.")
println("The command line arguments :")
for x in ARGS
    println(x)
end
```

If the file `showthreads.jl` contains

```
using Base.Threads

nbt = nthreads()
println("The number of threads : ", nbt)
```

then run via typing

```
JULIA_NUM_THREADS=8 julia showthreads.jl
```

at the command prompt. Alternatively, type

```
julia -t 8 showthreads.jl
```

to run the program with 8 threads.

The recursive parallel computation of the Fibonacci numbers is then defined in the program below:

```
import Base.Threads.@spawn

function fib(n::Int)
    if n < 2
        return n
    end
    t = @spawn fib(n-2)
    return fib(n-1) + fetch(t)
end

if length(ARGS) > 0
    nbr = parse(Int64, ARGS[1])
    println(fib(nbr))
else
    println(fib(10))
end
```

If the program is saved in the file `fibmt.jl`, then we run it with 8 threads, typing

```
JULIA_NUM_THREADS=8 julia fibmt.jl 10
```

or alternatively

```
julia -t 8 fibmt.jl 10
```

at the command prompt to compute the 10-th Fibonacci number with tasks mapped to 8 threads.

The recursive function `fib` illustrates the starting of a task and the synchronization of the sibling task:

- `t = @spawn fib(n-2)` starts a task to compute `fib(n-2)`
- `fetch(t)` waits for `t` to complete and gets its return value.

There can not be any speedup because of the only computation, the + happens after the synchronization.

3.4.2 Parallel Recursive Quadrature

The recursive parallel computation of the Fibonacci number serves as a pattern to compute integrals by recursively dividing the integration interval. The setup is identical as the section on Parallel Recursive Quadrature in the Tasking with OpenMP lecture.

The recursive application of the composite Trapezoidal rule is defined in the Julia function *rectraprule*.

```
function rectraprule(level::Int64,depth::Int64,
                     f::Function,a::Float64,
                  b::Float64,n::Int64)
    if level == depth
        return traprule(f,a,b,n)
    else
        middle = (b-a)/2

        t = @spawn rectraprule(level+1,depth, \
                               f,a,middle,n)
        return rectraprule(level+1,depth, \
                           f,middle,b,n) + fetch(t)
    end
end
```

Using a depth of recursion of 4, the output of a couple of runs is shown below:

```
$ time JULIA_NUM_THREADS=2 julia traprulerecmt.jl 4
1.7182818284590451e+00
1.7182818292271964e+00   error : 7.68e-10

real    0m5.207s
user    0m9.543s
sys     0m0.734s
$ time JULIA_NUM_THREADS=4 julia traprulerecmt.jl 4
1.7182818284590451e+00
1.7182818292271964e+00   error : 7.68e-10

real    0m3.120s
user    0m9.872s
sys     0m0.727s
$ time JULIA_NUM_THREADS=8 julia traprulerecmt.jl 4
1.7182818284590451e+00
1.7182818292271964e+00   error : 7.68e-10
```

(continues on next page)

(continued from previous page)

```
real    0m1.985s
user    0m10.617s
sys     0m0.735s
$
```

Observe the decrease of the wall clock time as the number of threads doubles.

On a Windows computer, replace the `time` by `Measure-Command`, and type

```
Measure-Command { julia -t 8 traprulerecmt.jl }
```

at the prompt in a PowerShell window.

3.4.3 Parallel Merge Sort

Merge sort works by divide and conquer, recursively as:

1. If no or one element, then return.
2. Split in two equal halves.
3. Sort the first half.
4. Sort the second half.
5. Merge the sorted halves.

The two above sort statements are recursive.

The sort algorithm will work in place, modifying the input, without returning. Instead of `fetch`, we use `wait`. The `wait(t)` waits on task `t` to finish.

The Julia function `psort!` is defined below.

```
"""
Sorts the elements of v in place, from hi to lo.
"""
function psort!(v, lo::Int=1, hi::Int=length(v))
    if lo >= hi
        return v
    end
    if hi - lo < 100000         # no multithreading
        sort!(view(v, lo:hi), alg = MergeSort)
        return v
    end

    mid = (lo+hi)>>>1           # find the midpoint

    # task to sort the first half starts
    half = @spawn psort!(v, lo, mid)

    # runs with the current call below
    psort!(v, mid+1, hi)

    # wait for the lower half to finish
    wait(half)
```

(continues on next page)

(continued from previous page)

```
    temp = v[lo:mid]            # workspace for merging
    i, k, j = 1, lo, mid+1      # merge the two sorted sub-arrays

    @inbounds while k < j <= hi
         if v[j] < temp[i]
             v[k] = v[j]
             j += 1
         else
             v[k] = temp[i]
             i += 1
         end
         k += 1
    end
    @inbounds while k < j
         v[i] = temp[i]
         k += 1
         i += 1
    end

    return v
 end
```

The @inbounds skips the checking of the index bounds when accessing array elements.

The main function of the program which times the Julia code properly with @time is listed below.

```
"""
Calls the psort! once before the timing
to avoid compilation overhead.
"""
function main()
    a = rand(100)
    b = copy(a)
    psort!(b)
    a = rand(20000000)
    b = copy(a)
    @time psort!(b)
end
```

Runs in a Bourne shell are listed below.

```
$ for n in 1 2 4 8; do JULIA_NUM_THREADS=$n julia mergesortmt.jl; done
  2.219275 seconds (3.31 k allocations: 686.950 MiB, 3.34% gc time)
  1.439491 seconds (3.59 k allocations: 686.959 MiB, 6.41% gc time)
  0.920875 seconds (3.63 k allocations: 686.963 MiB, 3.90% gc time)
  0.625733 seconds (3.73 k allocations: 686.969 MiB, 4.45% gc time)
$
```

Compare to the wall clock time:

```
$ time JULIA_NUM_THREADS=8 julia mergesortmt.jl
  0.618549 seconds (3.72 k allocations: 686.969 MiB, 4.78% gc time)
```

(continues on next page)

(continued from previous page)

```

real	0m1.220s
user	0m3.579s
sys	0m1.015s
$
```

3.4.4 Basic Linear Algebra Subprograms

The builtin `LinearAlgebra` package of Julia offers access to BLAS (Basic Linear Algebra Subroutines) which allow for multithreaded computations. This section illustrates the application of multithreading to solving linear algebra problems.

The inplace matrix matrix multiplication is provided via `mul!` as illustrated in an interactive session below.

```
julia> using LinearAlgebra

julia> A=[1.0 2.0; 3.0 4.0]; B=[1.0 1.0; 1.0 1.0];

julia> C = similar(B); mul!(C, A, B)
2×2 Array{Float64,2}:
 3.0  3.0
 7.0  7.0
```

Basic Linear Algebra Subprograms (BLAS) specifies common elementary linear algebra operations.

```
help?> BLAS.set_num_threads
  set_num_threads(n)

  Set the number of threads the BLAS library should use.
```

Setting the number of threads provides a parallel matrix multiplication. Consider the program `matmatmulmt.jl` listed below.

```
using LinearAlgebra

if length(ARGS) < 2
    println("use as")
    print("       julia ", PROGRAM_FILE)
    println(" dimension nthreads")
else
    n = parse(Int, ARGS[1])
    p = parse(Int, ARGS[2])

    BLAS.set_num_threads(p)
    A = rand(n, n)
    B = rand(n, n)
    C = similar(B)
    @time mul!(C, A, B)
end
```

The output of runs is shown next:

```
$ julia matmatmulmt.jl 8000 1
 20.823673 seconds (2.70 M allocations: 130.252 MiB)

$ julia matmatmulmt.jl 8000 2
 11.338446 seconds (2.70 M allocations: 130.252 MiB)

$ julia matmatmulmt.jl 8000 4
  6.242092 seconds (2.70 M allocations: 130.252 MiB)

$ julia matmatmulmt.jl 8000 8
  3.853406 seconds (2.70 M allocations: 130.252 MiB)

$ julia matmatmulmt.jl 8000 16
  2.487637 seconds (2.70 M allocations: 130.252 MiB)

$ julia matmatmulmt.jl 8000 32
  1.864454 seconds (2.70 M allocations: 130.252 MiB)
$
```

The peak flops performance depends on the size of the problem. The function `peakflops` computes the peak flop rate of the computer by using double precision `gemm!`

```
julia> using LinearAlgebra

julia> peakflops(8000)
3.331289611013868e11

julia> peakflops(16000)
3.475269847112081e11

julia> peakflops(4000)
3.130204729573054e11
```

The size of the problem needs to be large enough to fully occupy the available computing resources.

3.4.5 Exercises

1. Execute the recursive trapezoidal rule for different number of evaluations and increasing depths of recursion.

   For which values do you observe the best speedups?

2. Run the `peakflops` on your computer.

   For which dimension do you see the highest value?

   Compute the number of flops and relate this to the specifications of your computer.

3.5 Evaluating Parallel Performance

When evaluating the performance of parallel programs, we start by measuring time. We distinguish between times directly from time measurements and those that are derived, e.g.: flops. When the number of processors grows, the size of the problem has to grow as well to achieve the same performance, which then leads to the notion of isoefficiency. For task based parallel programs the length of a critical path in a task graph provides an upper bound on the speedup. With the roofline model, we can distinguish between computations that are compute bound or memory bound.

3.5.1 Metrics

The goal is to characterize parallel performance. Metrics are determined from performance measures. Time metrics are obtained from time measurements.

Time measurements are

1. execution time which includes
   - CPU time and system time
   - I/O time
2. overhead time is caused by
   - communication
   - synchronization

The wall clock time measures execution time plus overhead time.

Time metrics come directly from time measurements. Derived metrics are results of arithmetical metric expressions.

Definition of flops

flops are the number of floating-point operations per second:

$$\frac{\text{number of floating-point operations done}}{\text{execution time}}.$$

Definition of communication-to-computation ratio:

The *communication-to-computation ratio* is

$$\frac{\text{communication time}}{\text{execution time}}.$$

Definition of memory access-to-computation ratio:

The *memory access-to-computation ratio* is

$$\frac{\text{time spent on memory operations}}{\text{execution time}}.$$

Speedup and efficiency depend on the number of processors and are called *parallelism metrics*. Metrics used in performance evaluation are

- Peak speed is the maximum flops a computer can attain. Fast Graphics Processing Units achieve teraflop performance.
- Benchmark metrics use representative applications. The LINPACK benchmark ranks the Top 500 supercomputers.
- Tuning metrics include bottleneck analysis. For task-based parallel programs, the application of critical path analysis techniques finds the longest path in the execution of a parallel program.

3.5.2 Isoefficiency

The notion of isoefficiency complements the scalabiliy treatments introduced by the laws of Ahmdahl and Gustafson. The law of Ahmdahl keeps the dimension of the problem fixed and increases the number of processors. In applying the law of Gustafson we do the opposite: we fix the number of processors and increase the dimension of the problem. In practice, to examine the scalability of a parallel program, we have to treat both the dimension and the number of processors as variables.

Before we examine how relates to scalability, recall some definitions. For p processors:

$$\text{Speedup} = \frac{\text{serial time}}{\text{parallel time}} = S(p) \rightarrow p.$$

As we desire the speedup to reach p, the efficiency goes to 1:

$$\text{Efficiency} = \frac{\text{Speedup}}{p} = \frac{S(p)}{p} = E(p) \rightarrow 1.$$

Let T_s denote the serial time, T_p the parallel time, and T_0 the overhead, then: $pT_p = T_s + T_0$.

$$E(p) = \frac{T_s}{pT_p} = \frac{T_s}{T_s + T_0} = \frac{1}{1 + T_0/T_s}$$

The scalability analysis of a parallel algorithm measures its capacity to effectively utilize an increasing number of processors.

Let W be the problem size, for FFT: $W = n\log(n)$. Let us then relate E to W and T_0. The overhead T_0 depends on W and p: $T_0 = T_0(W,p)$. The parallel time equals

$$T_p = \frac{W + T_0(W,p)}{p}, \qquad \text{Speedup } S(p) = \frac{W}{T_p} = \frac{Wp}{W + T_0(W,p)}.$$

The efficiency is

$$E(p) = \frac{S(p)}{p} = \frac{W}{W + T_0(W,p)} = \frac{1}{1 + T_0(W,p)/W}.$$

The goal is for $E(p) \rightarrow 1$ as $p \rightarrow \infty$. The algorithm scales badly if W must grow exponentially to keep efficiency from dropping. If W needs to grow only moderately to keep the overhead in check, then the algorithm scales well.

Isoefficiency relates work to overhead:

$$\begin{aligned} E = \frac{1}{1 + T_0(W,p)/W} \quad & \Rightarrow \quad \frac{1}{E} = \frac{1 + T_0(W,p)/W}{1} \\ & \Rightarrow \quad \frac{1}{E} - 1 = \frac{T_0(W,p)}{W} \\ & \Rightarrow \quad \frac{1-E}{E} = \frac{T_0(W,p)}{W}. \end{aligned}$$

The isoefficiency function is

$$W = \left(\frac{E}{1-E}\right) T_0(W,p) \quad \text{or} \quad W = KT_0(W,p).$$

Keeping K constant, isoefficiency relates W to T_0. We can relate isoefficiency to the laws we encountered earlier:

- Amdahl's Law: keep W fixed and let p grow.
- Gustafson's Law: keep p fixed and let W grow.

Let us apply the isoefficiency to the parallel FFT. The isoefficiency function: $W = K\,T_0(W,p)$. For FFT: $T_s = n\log(n)t_c$, where t_c is the time for complex multiplication and adding a pair. Let t_s denote the startup cost and t_w denote the time to transfer a word. The time for a parallel FFT:

$$T_p = \underbrace{t_c \left(\frac{n}{p} \right) \log(n)}_{\text{computation time}} + \underbrace{t_s \log(p)}_{\text{start up time}} + \underbrace{t_w \left(\frac{n}{p} \right) \log(p)}_{\text{transfer time}}.$$

Comparing start up cost to computation cost, using the expression for T_p in the efficiency $E(p)$:

$$\begin{aligned} E(p) = \frac{T_s}{pT_p} &= \frac{n\log(n)t_c}{n\log(n)t_c + p\log(p)t_s + n\log(p)t_w} \\ &= \frac{Wt_c}{Wt_c + p\log(p)t_s + n\log(p)t_w}, \quad W = n\log(n). \end{aligned}$$

Assume $t_w = 0$ (shared memory):

$$E(p) = \frac{Wt_c}{Wt_c + p\log(p)t_s}.$$

We want to express $K = \dfrac{E}{1-E}$, using $\dfrac{1}{K} = \dfrac{1-E}{E} = \dfrac{1}{E} - 1$:

$$\frac{1}{K} = \frac{Wt_c + p\log(p)t_s}{Wt_c} - \frac{Wt_c}{Wt_c} \quad \Rightarrow \quad W = K\left(\frac{t_s}{t_c}\right) p\log(p).$$

The plot in Fig. 3.12 shows by how much the work load must increase to keep the same efficiency for an increasing number of processors.

Comparing transfer cost to the computation cost, taking another look at the efficiency $E(p)$:

$$E(p) = \frac{Wt_c}{Wt_c + p\log(p)t_s + n\log(p)t_w}, \quad W = n\log(n).$$

Assuming $t_s = 0$ (no start up):

$$E(p) = \frac{Wt_c}{Wt_c + n\log(p)t_w}.$$

We want to express $K = \dfrac{E}{1-E}$, using $\dfrac{1}{K} = \dfrac{1-E}{E} = \dfrac{1}{E} - 1$:

$$\frac{1}{K} = \frac{Wt_c + n\log(p)t_w}{Wt_c} - \frac{Wt_c}{Wt_c} \quad \Rightarrow \quad W = K\left(\frac{t_w}{t_c}\right) n\log(p).$$

In Fig. 3.13 the efficiency function is displayed for an increasing number of processors and various values of the dimension.

3.5.3 Task Graph Scheduling

A task graph is a Directed Acyclic Graph (DAG):

- nodes are tasks, and
- edges are precedence constraints between tasks.

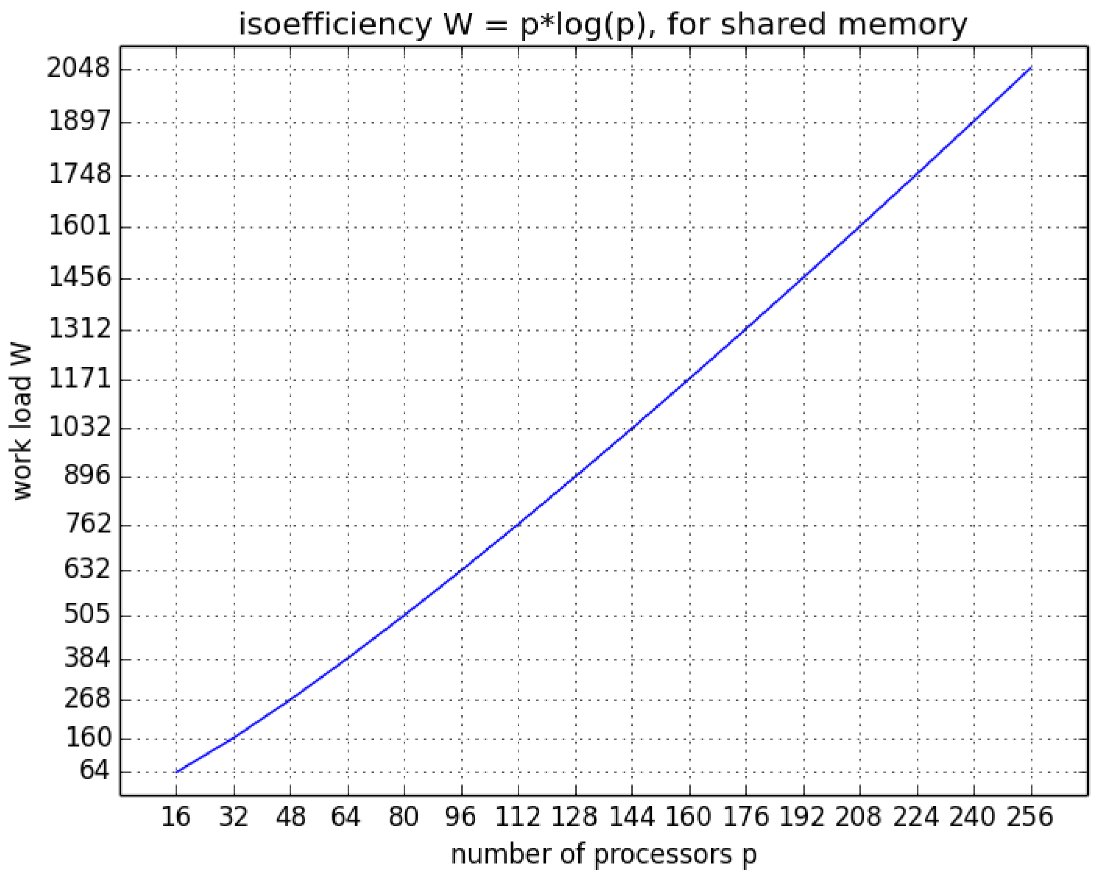


Fig. 3.12: Isoefficiency for a shared memory application.

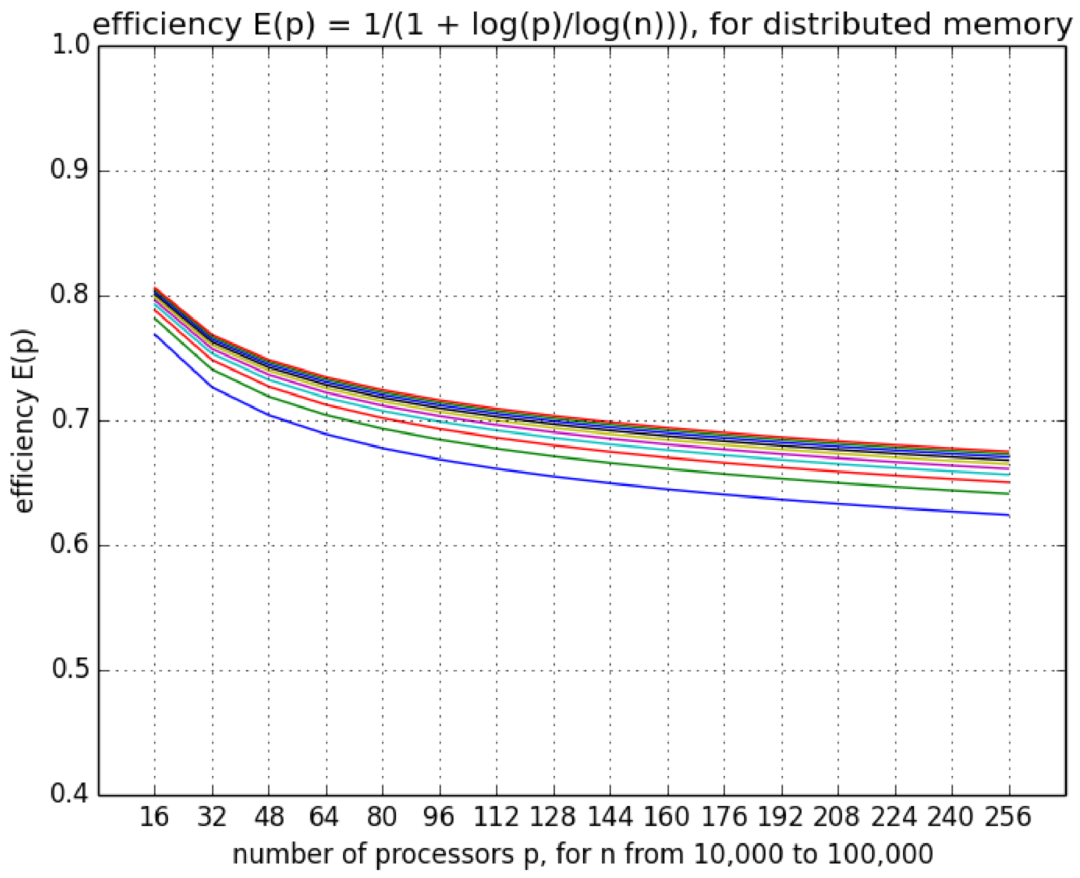


Fig. 3.13: Scalability analysis with a plot of the efficiency function.

Task graph scheduling or DAG scheduling maps the task graph onto a target platform.

The scheduler

1. takes a task graph as input,
2. decides which processor will execute what task,
3. with the objective to minimize the total execution time.

Let us consider the task graph of forward substitution.

Consider $L\mathbf{x} = \mathbf{b}$, an n-by-n lower triangular linear system, where $L = [\ell_{i,j}] \in \mathbb{R}^{n \times n}$, $\ell_{i,i} \neq 0$, $\ell_{i,j} = 0$, for $j > i$.

For $n = 3$, we compute:

$$\begin{array}{lcccl} \ell_{1,1}x_1 & = & b_1 & \Rightarrow & x_1 := b_1/\ell_{1,1} \\ \ell_{2,1}x_1 + \ell_{2,2}x_2 & = & b_2 & \Rightarrow & x_2 := (b_2 - \ell_{2,1}x_1)/\ell_{2,2} \\ \ell_{3,1}x_1 + \ell_{3,2}x_2 + \ell_{3,3}x_3 & = & b_3 & \Rightarrow & x_3 := (b_3 - \ell_{3,1}x_1 - \ell_{3,2}x_2)/\ell_{3,3} \end{array}$$

The formulas translate into pseudo code, with tasks labeled for each instruction:

```
task $T_{1,1}$ : $x_1 := b_1/\ell_{1,1}$
for $i$ from 2 to $n$ do
  for $j$ from 1 to $i-1$ do
    task $T_{i,j}$ : $b_i := b_i - \ell_{i,j}x_j$
  task $T_{i,i}$ : $x_i := b_i/\ell_{i,i}$
```

To decide which tasks depend on which other tasks, we apply Bernstein's conditions.

Each task T has an input set in(T), and an output set out(T).

Tasks T_1 and T_2 are independent if

$$\begin{array}{rcl} \text{in}(T_1) \cap \text{out}(T_2) & = & \emptyset, \text{ and} \\ \text{out}(T_1) \cap \text{in}(T_2) & = & \emptyset, \text{ and} \\ \text{out}(T_1) \cap \text{out}(T_2) & = & \emptyset. \end{array}$$

Applied to forward substitution:

```
task $T_{1,1}$ : $x_1 := b_1/\ell_{1,1}$           in$(T_{1,1}) = \{b_1, \ell_{1,1}\}$, out$(T_{1,1}) = \{x_1\}$
for $i$ from 2 to $n$ do
  for $j$ from 1 to $i-1$ do
    task $T_{i,j}$ : $b_i := b_i - \ell_{i,j}x_j$   in$(T_{i,j}) = \{x_j, b_i, \ell_{i,j}\}$, out$(T_{i,j}) = \{b_i\}$
  task $T_{i,i}$ : $x_i := b_i/\ell_{i,i}$         in$(T_{i,i}) = \{b_i, \ell_{i,i}\}$, out$(T_{i,i}) = \{x_i\}$
```

The task graph for a four dimensional linear system is shown in Fig. 3.14.

In the task graph of Fig. 3.14, a *critical path* is colored in red in Fig. 3.15.

Recall that $T_{i,i}$ computes x_i. The length of a critical path limits the speedup. For the above example, a sequential execution

$$T_{1,1}, T_{2,1}, T_{3,1}, T_{4,1}, T_{2,2}, T_{3,2}, T_{4,2}, T_{3,3}, T_{4,3}, T_{4,4}$$

takes 10 steps. The length of a critical path is 7. At most three threads can compute simultaneously. For $n = 4$, we found 7. For $n = 5$, the length of the critical path is 9, as can be seen from Fig. 3.16.

For any dimension n, the length of the critical path is $2n - 1$. At most $n - 1$ threads can compute simultaneously.

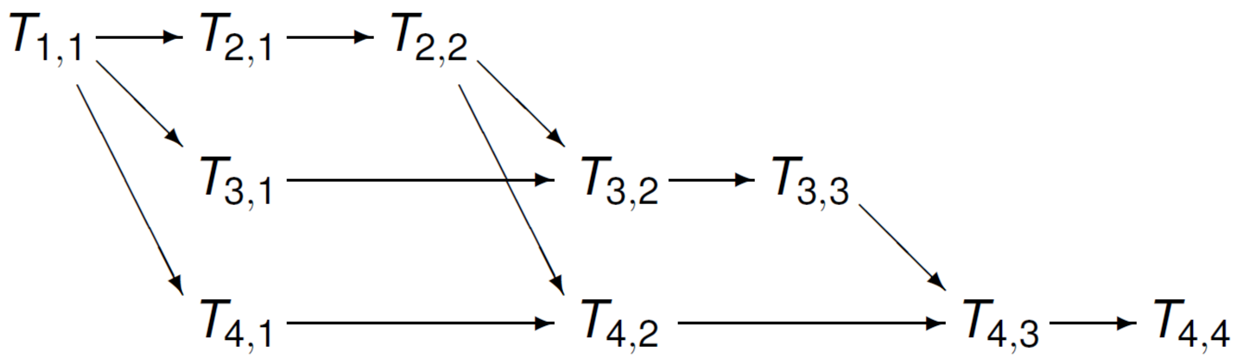


Fig. 3.14: Task graph for forward substition to solve a four dimensional lower triangular linear system.

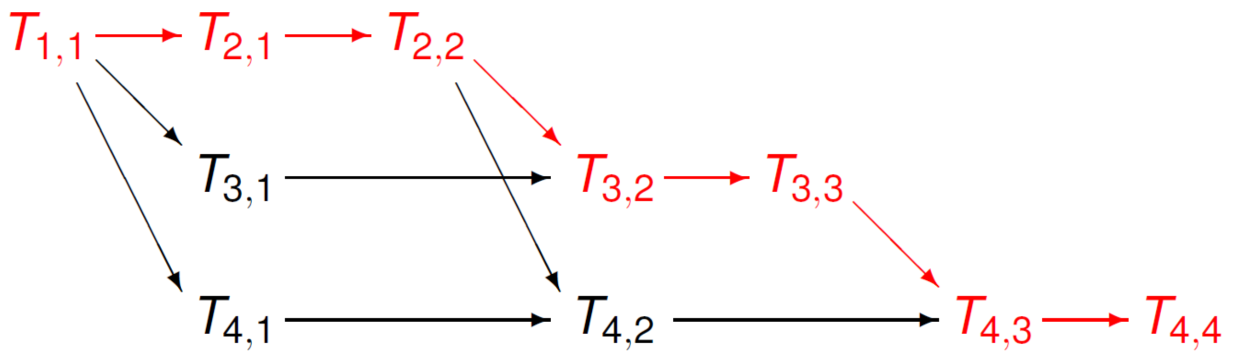


Fig. 3.15: A critical path is shown in red in the task graph for forward substition to solve a four dimensional lower triangular linear system.

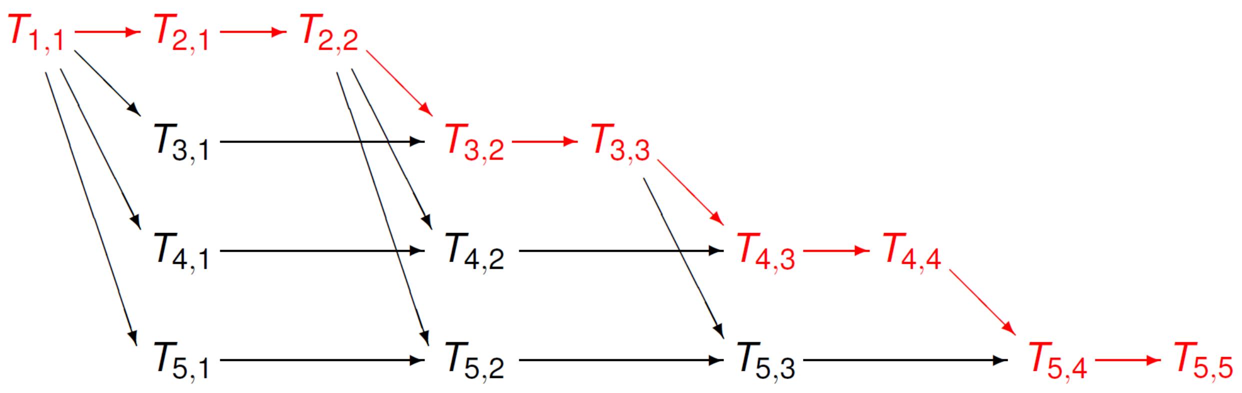


Fig. 3.16: A critical path is shown in red in the task graph for forward substition to solve a five dimensional lower triangular linear system.

3.5.4 The Roofline Model

Performance is typically measured in flops: the number of floating-point operations per second.

Definition of arithmetic intensity

The *arithmetic intensity* of a computation is the number of floating-point operations per byte.

For example, consider $z := x + y$, assign $x + y$ to z. One floating point operation involving 64-bit doubles, and each double occupies 8 bytes, so the arithmetic intensity is $1/24$.

Do you want faster memory or faster processors? To answer this question, we must decide if the computation if memory bound or compute bound.

Definition of memory bound

A computation is *memory bound* if the peak memory bandwidth determines the performance.

Memory bandwidth is the number of bytes per second that can be read or stored in memory.

Definition of compute bound

A computation is *compute bound* if the peak floating-point performance determines the performance.

A high arithmetic intensity is needed for a compute bound computation.

As an introduction to the roofline model, consider Fig. 3.17. The formula for attainable performance is

$$\begin{array}{c}\text{attainable}\\ \text{GFlops/sec}\end{array} = \min \left\{ \begin{array}{c} \text{peak floating point performance} \\ \\ \text{peak memory bandwidth} \times \text{operational intensity} \end{array} \right.$$

Observe the difference between arithmetic and operational intensity:

- arithmetic intensity measures the number of floating point operations per byte,
- operational intensity measures the number of operations per byte.

In applying the roofline model, in Fig. 3.17,

1. The horizontal line is the theoretical peak performance, expressed in gigaflops per second, the units of the vertical axis.
2. The units of the horizontal coordinate axis are flops per byte.

   The ridge point is the ratio of the theoretical peak performance and the memory bandwidth.
3. For any particular computation, record the pair (x, y)
   1. x is the arithmetic intensity, number of flops per byte,
   2. y is the performance defined by the number of flops per second.

If (x, y) lies under the horizontal part of the roof, then the computation is compute bound, otherwise, the computation is memory bound.

To summarize, to decide if a computation is memory bound or compute bound, consider Fig. 3.18.

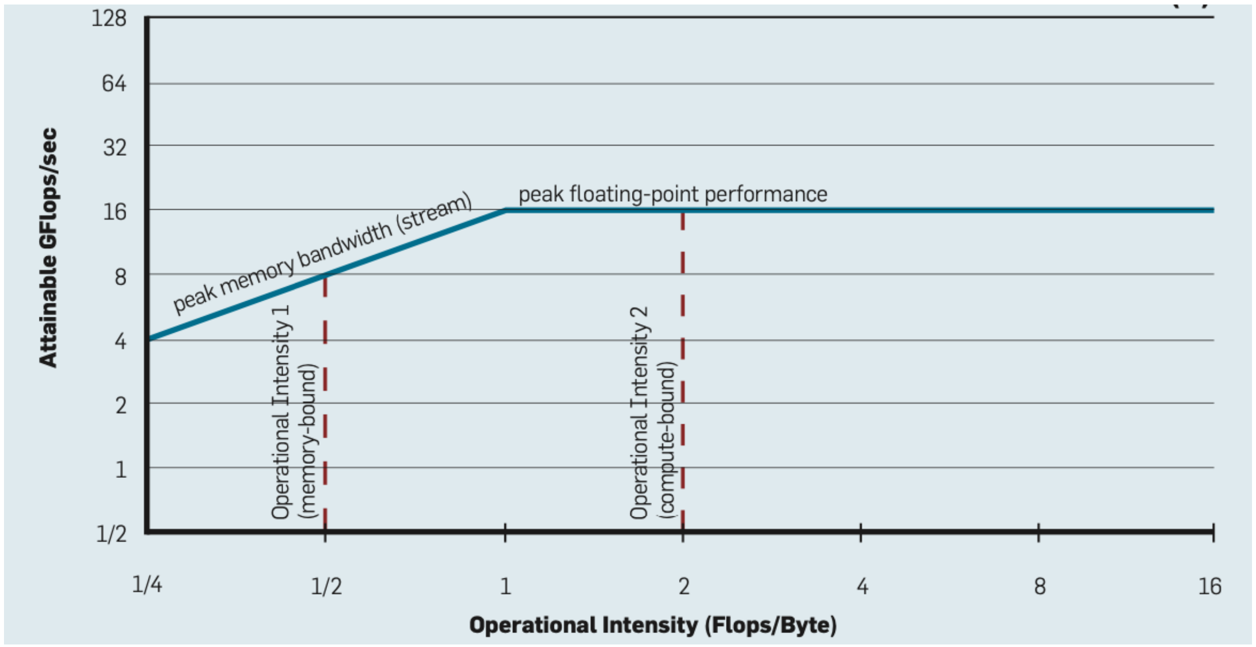


Fig. 3.17: The roofline model. Image copied from the paper by S. Williams, A. Waterman, and D. Patterson, 2009.

3.5.5 Bibliography

1. Thomas Decker and Werner Krandick: **On the Isoefficiency of the Parallel Descartes Method**. In *Symbolic Algebraic Methods and Verification Methods*, pages 55–67, Springer 2001. Edited by G. Alefeld, J. Rohn, S. Rump, and T. Yamamoto.
2. Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar: **Introduction to Parallel Computing**. 2nd edition, Pearson 2003.
3. Vipin Kumar and Anshul Gupta: **Analyzing Scalability of Parallel Algorithms and Architectures**. *Journal of Parallel and Distributed Computing* 22: 379–391, 1994.
4. Alan D. Malony: **Metrics.** In *Encycopedia of Parallel Computing*, edited by David Padua, pages 1124–1130, Springer 2011.
5. Yves Robert: **Task Graph Scheduling.** In *Encycopedia of Parallel Computing*, edited by David Padua, pages 2013–2024, Springer 2011.
6. S. Williams, A. Waterman, and D. Patterson: **Roofline: an insightful visual performance model for multicore architectures.** *Communications of the ACM*, 52(4):65-76, 2009.

3.6 Work Stealing

Work stealing is an alternative to load balancing. In parallel shared memory computing, we apply the work crew model.

We distinguish between static and dynamic work assignment:

1. Static: before the execution of the program. Each worker has *its own queue of jobs* to process.

   $+$ Ideal speedup if jobs are evenly distributed,

   $-$ if one worker gets all long jobs, then unbalanced.

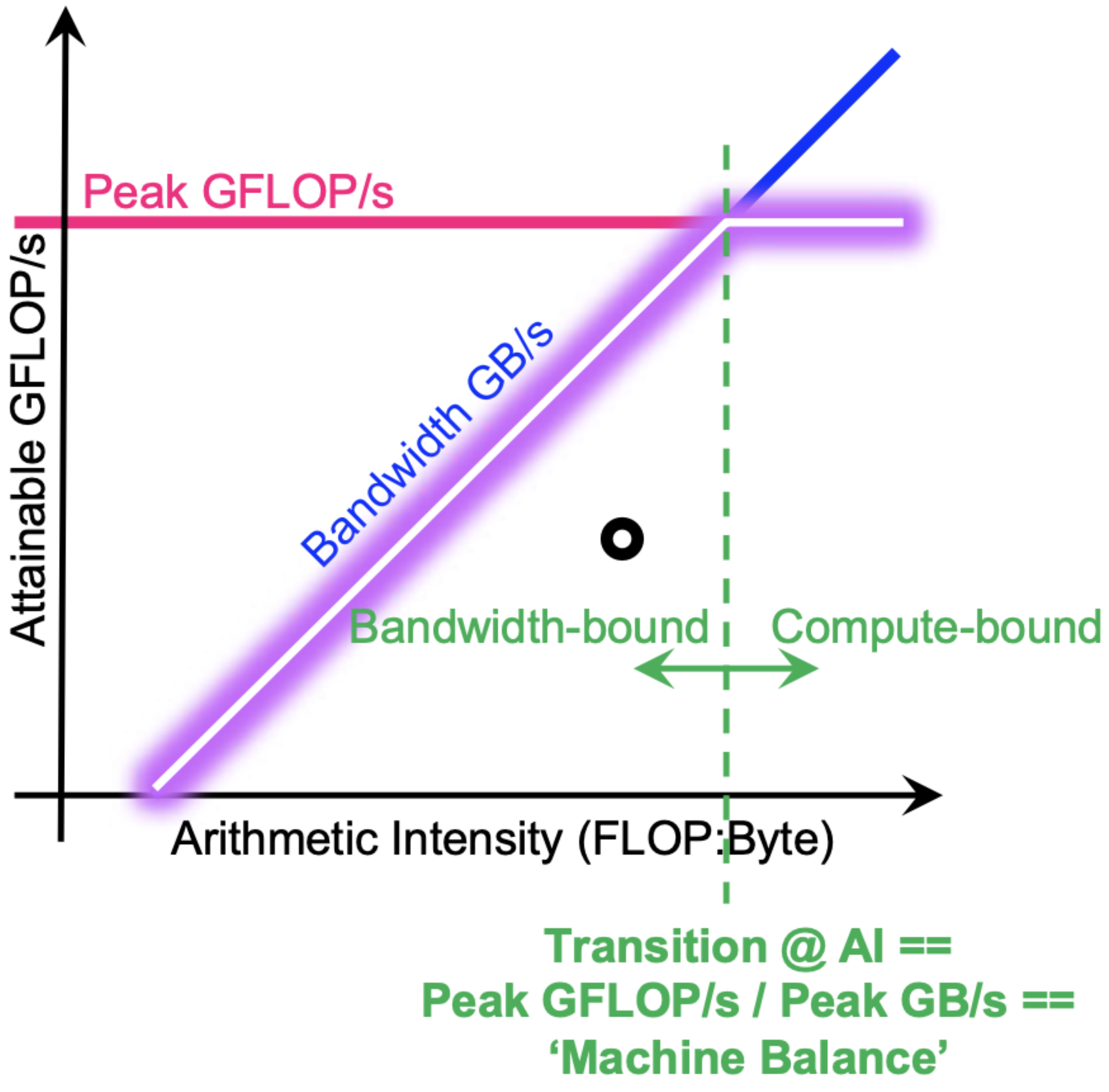


Fig. 3.18: Memory bound or compute bound? Image copied from the tutorial slides by Charlene Yang, LBNL, 16 June 2019.

2. Dynamic: during the execution of the program. Workers process *the same queue of jobs.*

   + The size of each job is taken into account,

   − synchronization overhead may dominate for small jobs and when there are many workers.

Tasks are much lighter than threads. starting and terminating a task is much faster than starting and terminating a thread; and a thread has its own process id and own resources, whereas a task is typically a small routine. In scheduling threads on processors, we distinguish between work sharing and work stealing. In work sharing, the scheduler attempts to migrate threads to under-utilized processors in order to distribute the work. In work stealing, under-utilized processors attempt to steal threads from other processors.

3.6.1 Work Stealing Simulated by a Julia Program

Work stealing is illustrated as a hybrid between static and dynamic work assignment:

1. Each worker starts with its own queue.
2. An idle worker will work on jobs of other queues.

Main benefit over dynamic work assignment: synchronization overhead occurs only at the end of the execution.

The setup of the Julia simulation is defined below.

1. As many queues as the number of threads are generated:

   - even indexed queues have small jobs,
   - odd indexed queues have large jobs.

   This generates unbalanced job queues to test the work stealing.

2. The i-th worker starts processing the i-th job queue.
3. Every queue has an index to the current job. In Julia, this index is of type `Atomic{Int}`, for mutual exclusive access.
4. After the i-th worker is done with its i-th job queue, it searches for jobs over all j-th queues, for $j \neq i$.

The Julia code to make the job queues is listed below.

```julia
using Base.Threads

nt = nthreads()

nbr = 10 # number of jobs in each queue
# allocate memory for all job queues
jobs = [zeros(nbr) for i=1:nt]

# every worker generates its own job queue
# even indexed queues have light work loads
@threads for i=1:nt
    if i % 2 == 0
        jobs[i] = rand((1, 2, 3), nbr)
    else
        jobs[i] = rand((4, 5, 6), nbr)
    end
    println("Worker ", threadid(), " has jobs ",
            jobs[i], " ", sum(jobs[i]))
end
```

The output of a run of the program with four threads is shown below. Each number in the job queue represents the time each job takes.

```
$ julia -t 4 worksteal.jl
Worker 1 has jobs [6.0, 6.0, 6.0, ... , 5.0] 53.0
Worker 3 has jobs [4.0, 4.0, 5.0, ... , 5.0] 48.0
Worker 4 has jobs [3.0, 2.0, 3.0, ... , 3.0] 24.0
Worker 2 has jobs [2.0, 2.0, 2.0, ... , 2.0] 14.0
```

The `...` represents omitted numbers for brevity.

The last number of the output is the sum of the times of the jobs. Workers 2 and 4 has clearly lighter loads, compared to workers 1 and 3.

In the code below, every worker starts processing its own queue:

```
jobidx = [Atomic{Int}(1) for i=1:nt]
@threads for i=1:nt
    while true
        myjob = atomic_add!(jobidx[i], 1)
        if myjob > length(jobs[i])
            break
        end
        println("Worker ", threadid(),
                " spends ", jobs[i][myjob], " seconds",
                " on job ", myjob, " ...")
        sleep(jobs[i][myjob])
        jobs[i][myjob] = threadid()
    end
```

Observe the use of the `Atomic{Int}` for the indices. The `myjob = atomic_add!(jobidx[i], 1)`

- increments the `jobidx[i]` after returning its value.
- This statement is executed in a critical section.

Then the code continues, idle threads steal work:

```
println("Worker ", threadid(), " will steal jobs ...")
more2steal = true
while more2steal
    more2steal = false
    for j=1:threadid()-1
        myjob = atomic_add!(jobidx[j], 1)
        if myjob <= length(jobs[j])
            println("Worker ", threadid(),
                    " spends ", jobs[j][myjob], " seconds",
                    " on job ", myjob, " of ", j, " ...")
            sleep(jobs[j][myjob])
            jobs[j][myjob] = threadid()
        end
        more2steal = (myjob < length(jobs[j]))
    end
    for j=threadid()+1:nt # is similar to previous code
```

An example of a part of an output is shown below.

```
Worker 4 spends 1.0 seconds on job 7 ...
Worker 2 will steal jobs ...
Worker 2 spends 4.0 seconds on job 4 of 1 ...
Worker 4 spends 3.0 seconds on job 8 ...
Worker 1 spends 4.0 seconds on job 5 ...
Worker 4 spends 3.0 seconds on job 9 ...
Worker 2 spends 4.0 seconds on job 5 of 3 ...
Worker 3 spends 5.0 seconds on job 6 ...
Worker 4 spends 3.0 seconds on job 10 ...
Worker 1 spends 6.0 seconds on job 6 ...
Worker 3 spends 6.0 seconds on job 7 ...
Worker 4 will steal jobs ...
Worker 4 spends 6.0 seconds on job 7 of 1 ...
Worker 1 spends 4.0 seconds on job 8 ...
```

Worker 2 is done first, takes job 4 of worker 1. Worker 1 then continues with job 5. When worker 4 is done, it takes job 7 of worker 1. Worker 1 then continues with job 8.

To conclude, we make the following observations:

- Implementing a work crew with work stealing is not much more complicated than dynamic load balancing.
- The idle workers start at the first queue and then progress linearly, which may be good if the first queue contains all important jobs.
- In an alternative work stealing scheme, idle workers would start in the queue of their immediate neighbors.

3.6.2 Multithreading in Python with Numba

Numba is an open-source JIT compiler that translates a subset of Python and NumPy into fast machine code using LLVM, via the llvmlite Python package. It offers a range of options for parallelising Python code for CPUs and GPUs, often with only minor code changes. Started by Travis Oliphant in 2012, under active development `https://github.com/numba/numba`.

To use, do `pip install numba`. The example below (copied from the wikipedia page) works on Windows.

```python
import numba
import random

@numba.jit
def monte_carlo_pi(n_samples: int) -> float:
    """
    Applies Monte Carlo to estimate pi.
    """
    acc = 0
    for i in range(n_samples):
        x = random.random()
        y = random.random()
        if (x**2 + y**2) < 1.0:
            acc += 1
    return 4.0 * acc / n_samples

p = monte_carlo_pi(1000000)
print(p)
```

3.6.3 Multithreading in Python with Parsl

Parsl stands for Parallel Scripting in Python. Parsl provides an intuitive, pythonic way of parallelizing codes by annotating ''apps": Python functions or external applications that run concurrently. Parsl works seamlessly with Jupyter notebooks. Write once, run anywhere. From laptops to supercomputers.

To use, do `pip install parsl`. The example (copied from the parsl user guide) below was executed on WSL, Window Subsystem for Linux, Ubuntu 22.04.

```
from parsl import python_app
import parsl

parsl.load()

# Map function that returns double the input integer
@python_app
def app_double(x):
    return x*2

# Reduce function that returns the sum of a list
@python_app
def app_sum(inputs=()):
    return sum(inputs)

# Create a list of integers
items = range(0,4)

# Map phase: apply the double *app* function to each item in list
mapped_results = []
for i in items:
    x = app_double(i)
    mapped_results.append(x)

# Reduce phase: apply the sum *app* function to the set of results
total = app_sum(inputs=mapped_results)

print(total.result())
```

3.6.4 the Intel Threading Building Blocks (TBB)

The Intel TBB is a library that helps you leverage multicore performance *without having to be a threading expert*. The advantage of Intel TBB is that it works at a higher level than raw threads, yet does not require exotic languages or compilers. The library differs from others in the following ways:

- TBB enables you to specify logical parallelism instead of threads;
- TBB targets threading for performance;
- TBB is compatible with other threading packages;
- TBB emphasizes scalable, data parallel programming;
- TBB relies on generic programming, (e.g.: use of STL in C++).

The code is open source, free to download at < http://threadingbuildingblocks.org/>

The TBB task scheduler uses *work stealing* for load balancing.

Our first C++ program, similar to our previous *Hello world!* programs, using TBB is below. A `class` in C++ is a like a `struct` in C for holding data attributes and functions (called methods).

```
#include "tbb/tbb.h"
#include <cstdio>
using namespace tbb;

class say_hello
{
   const char* id;
   public:
      say_hello(const char* s) : id(s) { }
      void operator( ) ( ) const
      {
         printf("hello from task %s\n",id);
      }
};

int main( )
{
   task_group tg;
   tg.run(say_hello("1")); // spawn 1st task and return
   tg.run(say_hello("2")); // spawn 2nd task and return
   tg.wait( );             // wait for tasks to complete
}
```

The `run` method spawns the task immediately, but does not block the calling task, so control returns immediately. To wait for the child tasks to finish, the classing task calls `wait`. Observe the syntactic simplicity of `task_group`. When running the code, we see on screen:

```
$ ./hello_task_group
hello from task 2
hello from task 1
$
```

3.6.5 using the parallel_for

Consider the following problem of raising complex numbers to a large power.

- **Input** $n \in \mathbb{Z}_{>0}, d \in \mathbb{Z}_{>0}$,
  $\mathbf{x} \in \mathbb{C}^n$.
- **Output** $\mathbf{y} \in \mathbb{C}^n, y_k = x_k^d$,
  for $k = 1, 2, \ldots, n$.

Let us first develop the serial program.

To avoid overflow, we take complex numbers on the unit circle. In C++, complex numbers are defined as a template class. To instantiate the class `complex` with the type `double` we first declare the type `dcmplx`. Random complex numbers are generated as $e^{2\pi i\theta} = \cos(2\pi\theta) + i\sin(2\pi\theta)$, for random $\theta \in [0, 1]$.

```
#include <complex>
#include <cstdlib>
#include <cmath>
```

(continues on next page)

(continued from previous page)

```
using namespace std;

typedef complex<double> dcmplx;

dcmplx random_dcmplx ( void );
// generates a random complex number on the complex unit circle
dcmplx random_dcmplx ( void )
{
   int r = rand();
   double d = ((double) r)/RAND_MAX;
   double e = 2*M_PI*d;
   dcmplx c(cos(e),sin(e));
   return c;
}
```

We next define the function to write arrays. Observe the local declaration `int i` in the `for` loop, the scientific formatting, and the methods `real()` and `imag()`.

```
#include <iostream>
#include <iomanip>

void write_numbers ( int n, dcmplx *x ); // writes the array of n doubles in x
void write_numbers ( int n, dcmplx *x )
{
   for(int i=0; i<n; i++)
      cout << scientific << setprecision(4)
           << "x[" << i << "] = ( " << x[i].real()
           << " , " << x[i].imag() << ")\n";
}
```

Below it the prototype and the definition of the function to raise an array of `n` double complex number to some power. Because the builtin `pow` function applies repeated squaring, it is too efficient for our purposes and we use a plain loop.

```
void compute_powers ( int n, dcmplx *x, dcmplx *y, int d );
// for arrays x and y of length n, on return y[i] equals x[i]**d
void compute_powers ( int n, dcmplx *x, dcmplx *y, int d )
{
   for(int i=0; i < n; i++) // y[i] = pow(x[i],d); pow is too efficient
   {
      dcmplx r(1.0,0.0);
      for(int j=0; j < d; j++) r = r*x[i];
      y[i] = r;
   }
}
```

Without command line arguments, the main program prompts the user for the number of elements in the array and for the power. The three command line arguments are the dimension, the power, and the verbose level. If the third parameter is zero, then no numbers are printed to screen, otherwise, if the third parameter is one, the powers of the random numbers are shown. Running the program in silent mode is useful for timing purposes. Below are some example sessions with the program.

```
$ /tmp/powers_serial
how many numbers ? 2
```

(continues on next page)

(continued from previous page)

```
x[0] = ( -7.4316e-02 , 9.9723e-01)
x[1] = ( -9.0230e-01 , 4.3111e-01)
give the power : 3
x[0] = ( 2.2131e-01 , -9.7520e-01)
x[1] = ( -2.3152e-01 , 9.7283e-01)

$ /tmp/powers_serial 2 3 1
x[0] = ( -7.4316e-02 , 9.9723e-01)
x[1] = ( -9.0230e-01 , 4.3111e-01)
x[0] = ( 2.2131e-01 , -9.7520e-01)
x[1] = ( -2.3152e-01 , 9.7283e-01)

$ time /tmp/powers_serial 1000 1000000 0

real    0m17.456s
user    0m17.451s
sys     0m0.001s
```

The main program is listed below:

```
int main ( int argc, char *argv[] )
{
   int v = 1;     // verbose if > 0
   if(argc > 3) v = atoi(argv[3]);
   int dim;       // get the dimension
   if(argc > 1)
      dim = atoi(argv[1]);
   else
   {
      cout << "how many numbers ? ";
      cin >> dim;
   }
   // fix the seed for comparisons
   srand(20120203); //srand(time(0));
   dcmplx r[dim];
   for(int i=0; i<dim; i++)
      r[i] = random_dcmplx();
   if(v > 0) write_numbers(dim,r);
   int deg;       // get the degree
   if(argc > 1)
      deg = atoi(argv[2]);
   else
   {
      cout << "give the power : ";
      cin >> deg;
   }
   dcmplx s[dim];
   compute_powers(dim,r,s,deg);
   if(v > 0) write_numbers(dim,s);

   return 0;
}
```

To speed up the computations, `parallel_for` is used. We first illustrate the speedup that can be obtained with a parallel version of the code.

```
$ time /tmp/powers_serial 1000 1000000 0
real    0m17.456s
user    0m17.451s
sys     0m0.001s
$ time /tmp/powers_tbb 1000 1000000 0
real    0m1.579s
user    0m18.540s
sys     0m0.010s
```

The speedup: $\dfrac{17.456}{1.579} = 11.055$ with 12 cores. The class `ComputePowers` is defined below

```
class ComputePowers
{
   dcmplx *const c; // numbers on input
   int d;           // degree
   dcmplx *result;  // output
   public:
      ComputePowers(dcmplx x[], int deg, dcmplx y[])
         : c(x), d(deg), result(y) { }
      void operator()
         ( const blocked_range<size_t>& r ) const
      {
         for(size_t i=r.begin(); i!=r.end(); ++i)
         {
            dcmplx z(1.0,0.0);
            for(int j=0; j < d; j++) z = z*c[i];
            result[i] = z;
         }
      }
};
```

We next explain the use of `tbb/blocked_range.h`. A `blocked_range` represents a half open range $[i, j)$ that can be recursively split.

```
#include "tbb/blocked_range.h"

template<typename Value> class blocked_range

      void operator()
         ( const blocked_range<size_t>& r ) const
      {
         for(size_t i=r.begin(); i!=r.end(); ++i)
         {
```

Calling the `parallel_for` happens as follows:

```
#include "tbb/tbb.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"
#include "tbb/task_scheduler_init.h"

```

(continues on next page)

(continued from previous page)

```
using namespace tbb;
```

Two lines change in the main program:

```
task_scheduler_init init(task_scheduler_init::automatic);

parallel_for(blocked_range<size_t>(0,dim),
             ComputePowers(r,deg,s));
```

3.6.6 using the parallel_reduce

We consider the summation of integers as an application of work stealing. Fig. 3.19 and Fig. 3.20 are taken from the Intel TBB tutorial.

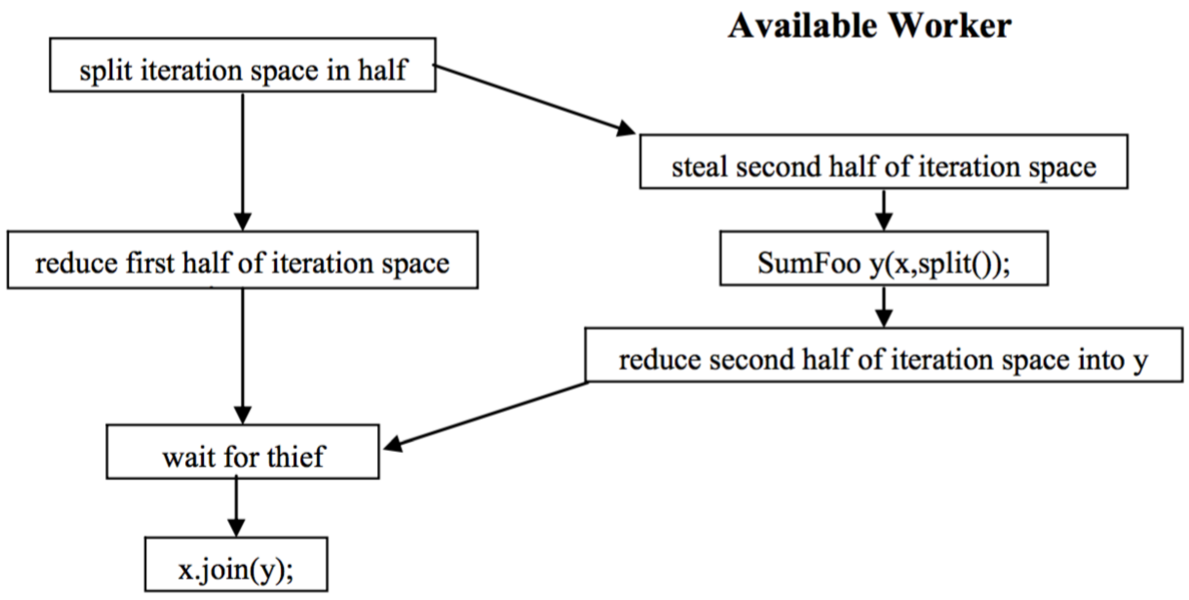

Available Worker
split iteration space in half
steal second half of iteration space
reduce first half of iteration space
SumFoo y(x,split());
reduce second half of iteration space into y
wait for thief
x.join(y);

Fig. 3.19: An application of work stealing.

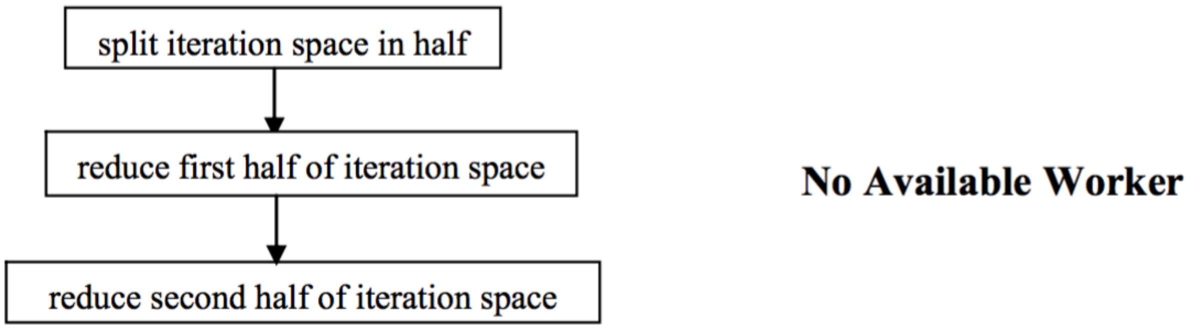

split iteration space in half
reduce first half of iteration space
reduce second half of iteration space
No Available Worker

Fig. 3.20: What if no worker is available?

The definition of the class to sum a sequence of integers is below.

```
class SumIntegers
{
   int *data;
   public:
      int sum;
      SumIntegers ( int *d ) : data(d), sum(0) {}
      void operator()
         ( const blocked_range<size_t>& r )
      {
         int s = sum;  // must accumulate !
         int *d = data;
         size_t end = r.end();
         for(size_t i=r.begin(); i != end; ++i)
            s += d[i];
         sum = s;
      }
      // the splitting constructor
      SumIntegers ( SumIntegers& x, split ) :
         data(x.data), sum(0) {}
      // the join method does the merge
      void join ( const SumIntegers& x ) { sum += x.sum; }
};

int ParallelSum ( int *x, size_t n )
{
   SumIntegers S(x);

   parallel_reduce(blocked_range<size_t>(0,n), S);

   return S.sum;
}
```

The main program is below:

```
int main ( int argc, char *argv[] )
{
   int n;
   if(argc > 1)
      n = atoi(argv[1]);
   else
   {
      cout << "give n : ";
      cin >> n;
   }
   int *d;
   d = (int*)calloc(n,sizeof(int));
   for(int i=0; i<n; i++) d[i] = i+1;

   task_scheduler_init init
      (task_scheduler_init::automatic);
   int s = ParallelSum(d,n);

   cout << "the sum is " << s
```

(continues on next page)

(continued from previous page)

```
          << " and it should be " << n*(n+1)/2
          << endl;
   return 0;
}
```

3.6.7 Bibliography

1. Intel Threading Building Blocks. Tutorial. Available online via <http://www.intel.com>.
2. Robert D. Blumofe and Charles E. Leiserson: **Scheduling Multithreaded Computations by Work-Stealing**. In the Proceedings of the 35th Annual IEEE Conference on Foundations of Computer Science (FoCS 1994), pages 356-368.
3. Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams and Katherine A. Yelick: **The Landscape of Parallel Computing Research: A View from Berkeley**. Technical Report No. UCB/EECS-2006-183 EECS Department, University of California, Berkeley, December 18, 2006.

3.6.8 Exercises

1. A permanent is similar to a determinant but then without the alternating signs. Develop a task-based parallel program to compute the permanent of a 0/1 matrix. Why is work stealing appropriate for this problem?
2. Modify the `hello world!` program with TBB so that the user is first prompted for a name. Two tasks are spawned and they use the given name in their greeting.
3. Modify `powers_tbb.cpp` so that the i-th entry is raised to the power $d - i$. In this way not all entries require the same work load. Run the modified program and compare the speedup to check the performance of the automatic task scheduler.

CHAPTER 4

Acceleration with Graphics Processing Units

A Graphics Processing Unit (GPU) is a massively parallel computer, capable of teraflop performance.

4.1 A Massively Parallel Processor: the GPU

4.1.1 Introduction to General Purpose GPUs

Thanks to the industrial success of video game development graphics processors became faster than general CPUs. General Purpose Graphic Processing Units (GPGPUs) are available, capable of double floating point calculations. Accelerations by a factor of 10 with one GPGPU are not uncommon. Comparing electric power consumption is advantageous for GPGPUs.

Thanks to the popularity of the PC market, millions of GPUs are available – every PC has a GPU. This is the first time that massively parallel computing is feasible with a mass-market product. Applications such as magnetic resonance imaging (MRI) use some combination of PC and special hardware accelerators.

In five weeks, we plan to cover the following topics:

1. architecture, programming models, scalable GPUs
2. introduction to CUDA and data parallelism
3. CUDA thread organization, synchronization
4. CUDA memories, reducing memory traffic
5. coalescing and applications of GPU computing

The lecture notes follow the book by David B. Kirk and Wen-mei W. Hwu: *Programming Massively Parallel Processors. A Hands-on Approach.* Elsevier 2010; fourth edition, 2023, with Izzat El Hajj as third author.

What are the expected learning outcomes from the part of the course?

1. We will study the design of massively parallel algorithms.
2. We will understand the architecture of GPUs and the programming models to accelerate code with GPUs.

3. We will use software libraries to accelerate applications.

The key questions we address are the following:

1. Which problems may benefit from GPU acceleration?
2. Rely on existing software or develop own code?
3. How to mix MPI, multicore, and GPU?

The textbook authors use the peach metaphor: much of the application code will remain sequential; but GPUs can dramatically improve easy to parallelize code.

Our Microway workstation `pascal` (acquired in 2016) has an NVIDIA GPU with the CUDA software development installed.

- NIVDIA P100 general purpose graphics processing unit
  1. number of CUDA cores: 3,584 (56×64)
  2. frequency of CUDA cores: 405MHz
  3. double precision floating point performance: 4.7 Tflops (peak)
  4. single precision floating point performance: 9.3 Tflops (peak)
  5. total global memory: 16275 MBytes
- CUDA programming model with `nvcc` compiler.

To compare the theoretical peak performance of the P100, consider the theoretical peak performance of the two Intel E5-2699v4 (2.2GHz 22 cores) CPUs in the workstation:

- 2.20 GHz $\times$ 8 flops/cycle = 17.6 GFlops/core;
- 44 core $\times$ 17.6 GFlops/core = 774.4 GFlops.

$\Rightarrow$ 4700/774.4 = 6.07. One P100 is as strong as $6 \times 44 = 264$ cores.

CUDA stands for Compute Unified Device Architecture, is a general purpose parallel computing architecture introduced by NVIDIA.

Consider the comparison of the specifications of three consecutive NVIDIA GPUs:

1. NVIDIA P100 16GB `Pascal` Accelerator
   - 3,586 CUDA cores, 3,586 = 56 SM $\times$ 64 cores/SM
   - GPU max clock rate: 1329 MHz (1.33 GHz)
   - 16GB Memory at 720GB/sec peak bandwidth
   - peak performance: 4.7 TFLOPS double precision
2. NVIDIA V100 `Volta` Accelerator
   - 5,120 CUDA cores, 5,120 = 80 SM $\times$ 64 cores/SM
   - GPU max clock rate: 1912 MHz (1.91 GHz)
   - Memory clock rate: 850 Mhz
   - 32GB Memory at 870GB/sec peak bandwidth
   - peak performance: 7.9 TFLOPS double precision
3. NVIDIA A100 `Ampere` Accelerator
   - 6,912 CUDA cores, 6,912 = 108 SM $\times$ 64 cores/SM

- GPU max clock rate: 1410 MHz (1.41 GHz)
- Memory clock rate: 1512 Mhz
- 80GB Memory at 1.94TB/sec peak bandwidth
- peak performance: 9.7 TFLOPS double precision
- peak FP64 Tensor Core performance: 19.5 TFLOPS

Tensor cores were already present in the P100 and V100, but those earlier tensor core were not capable of double precision arithmetic.

The full specifications are described in whitepapers available at the web site of NVIDIA:

- NVIDIA GeForce GTX 680, 2012.
- NVIDIA Tesla P100, 2016.
- NVIDIA Tesla V100 GPU Architecture, August 2017.
- NVIDIA A100 Tensor Core GPU Architecture, 2020.
- NVIDIA Ampere GA102 GPU Architecture, 2021.
- NVIDIA H100 Tensor Core GPU Architecture, 2022.

The evolution of bandwidth is shown in Fig. 4.1.

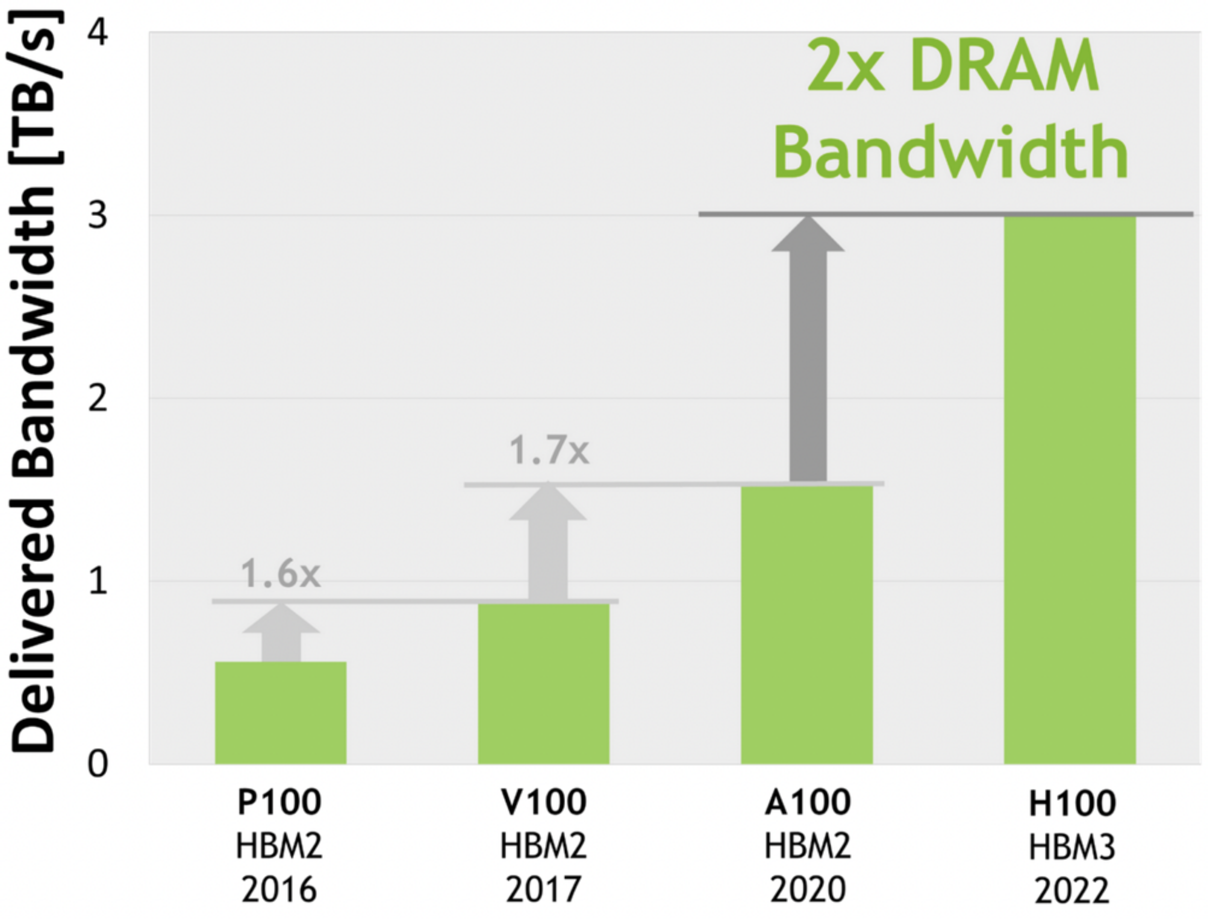
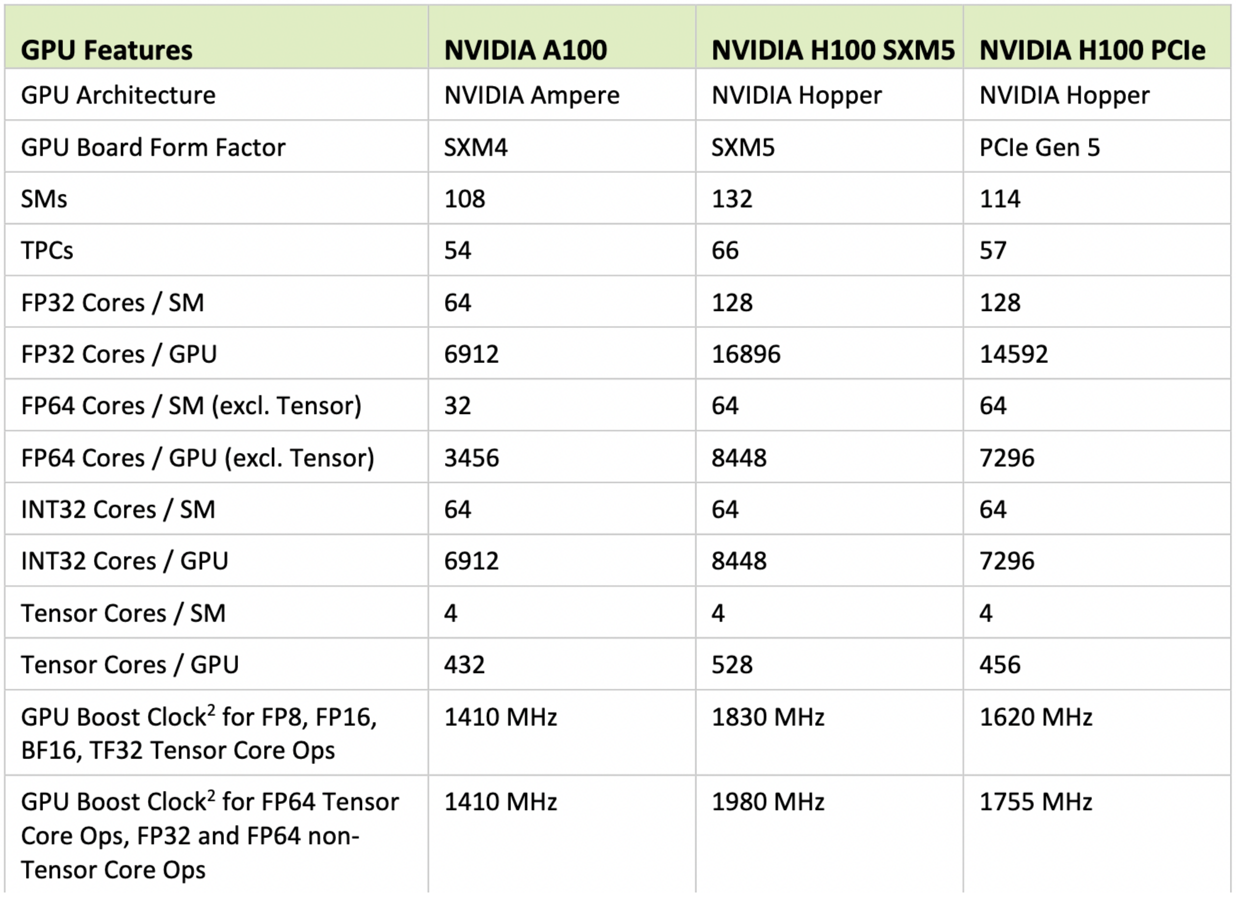


Fig. 4.1: Evolution of the bandwidth, from the NVIDIA developer documentation.

The evolution of core counts is shown in Fig. 4.2.



GPU Features	NVIDIA A100	NVIDIA H100 SXM5	NVIDIA H100 PCIe
GPU Architecture	NVIDIA Ampere	NVIDIA Hopper	NVIDIA Hopper
GPU Board Form Factor	SXM4	SXM5	PCIe Gen 5
SMs	108	132	114
TPCs	54	66	57
FP32 Cores / SM	64	128	128
FP32 Cores / GPU	6912	16896	14592
FP64 Cores / SM (excl. Tensor)	32	64	64
FP64 Cores / GPU (excl. Tensor)	3456	8448	7296
INT32 Cores / SM	64	64	64
INT32 Cores / GPU	6912	8448	7296
Tensor Cores / SM	4	4	4
Tensor Cores / GPU	432	528	456
GPU Boost Clock^2 for FP8, FP16, BF16, TF32 Tensor Core Ops	1410 MHz	1830 MHz	1620 MHz
GPU Boost Clock^2 for FP64 Tensor Core Ops, FP32 and FP64 non-Tensor Core Ops	1410 MHz	1980 MHz	1755 MHz

Fig. 4.2: Evolution of the core counts, from the NVIDIA developer documentation.

The evolution of the non-Tensor peak performance for various GPUs is as follows: Pascal: 4.7, Volta: 7.9, Ampere: 9.7, Hopper: 25.6 TFLOPS.

4.1.2 Graphics Processors as Parallel Computers

In this section we compare the performance between GPUs and CPU, explaining the difference between their architectures. The performance gap between GPUs and CPUs is illustrated by two figures, taken from the NVIDIA CUDA programming guide. We compare the flops in Fig. 4.3 and the memory bandwidth in Fig. 4.4.

Memory bandwidth is the rate at which data can be read from/stored into memory, expressed in bytes per second. Graphics chips operate at approximately 10 times the memory bandwidth of CPUs. For our Microway station `pascal`, the memory bandwidth of the CPU is 76.8GB/s, whereas the NVIDIA P100 has 720GB/s as peak bandwidth. Straightforward parallel implementations on GPGPUs often achieve directly a speedup of 10, saturating the memory bandwidth.

The main distinction between the CPU and GPU design is as follows:

- CPU: multicore processors have large cores and large caches using control for optimal serial performance.
- GPU: optimizing execution throughput of massive number of threads with small caches and minimized control units.

The distinction is illustrated in Fig. 4.5.

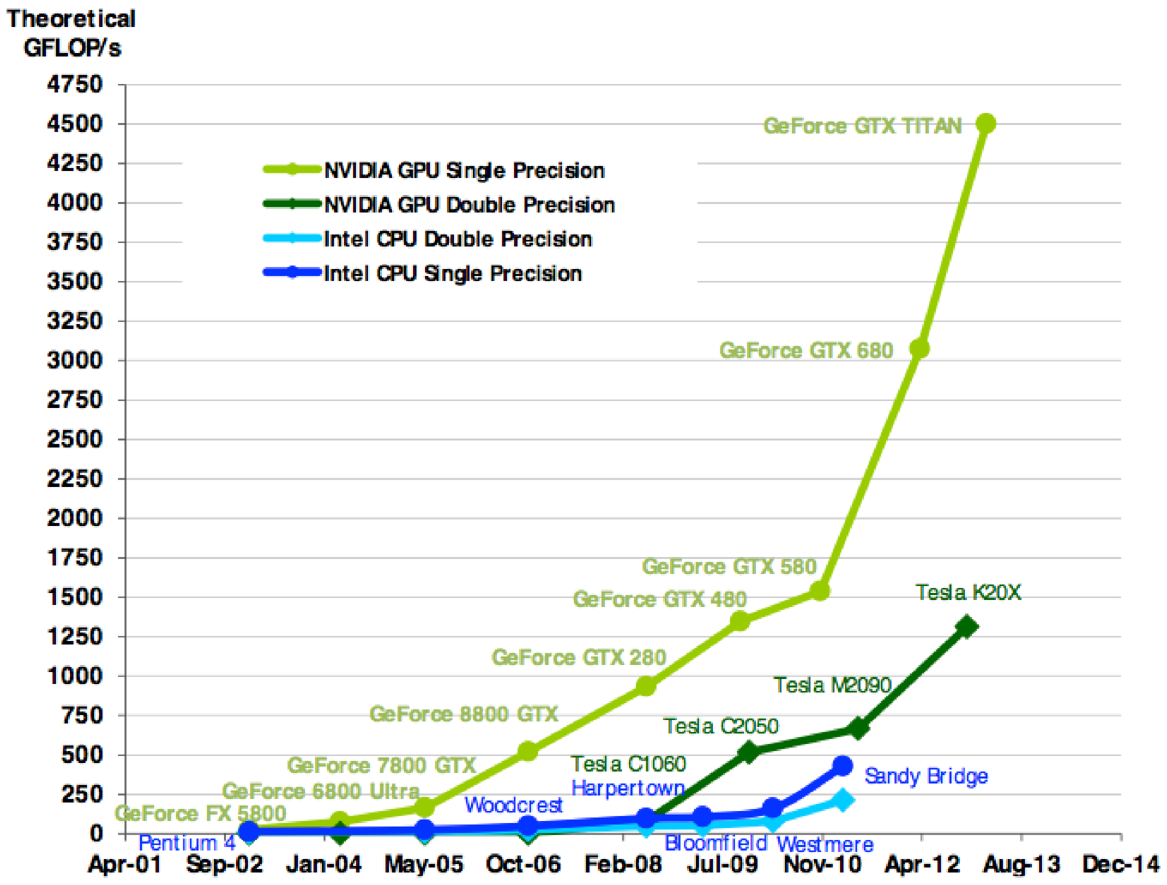


Figure 1 Floating-Point Operations per Second for the CPU and GPU

Fig. 4.3: Flops comparison taken from the NVIDIA programming guide.

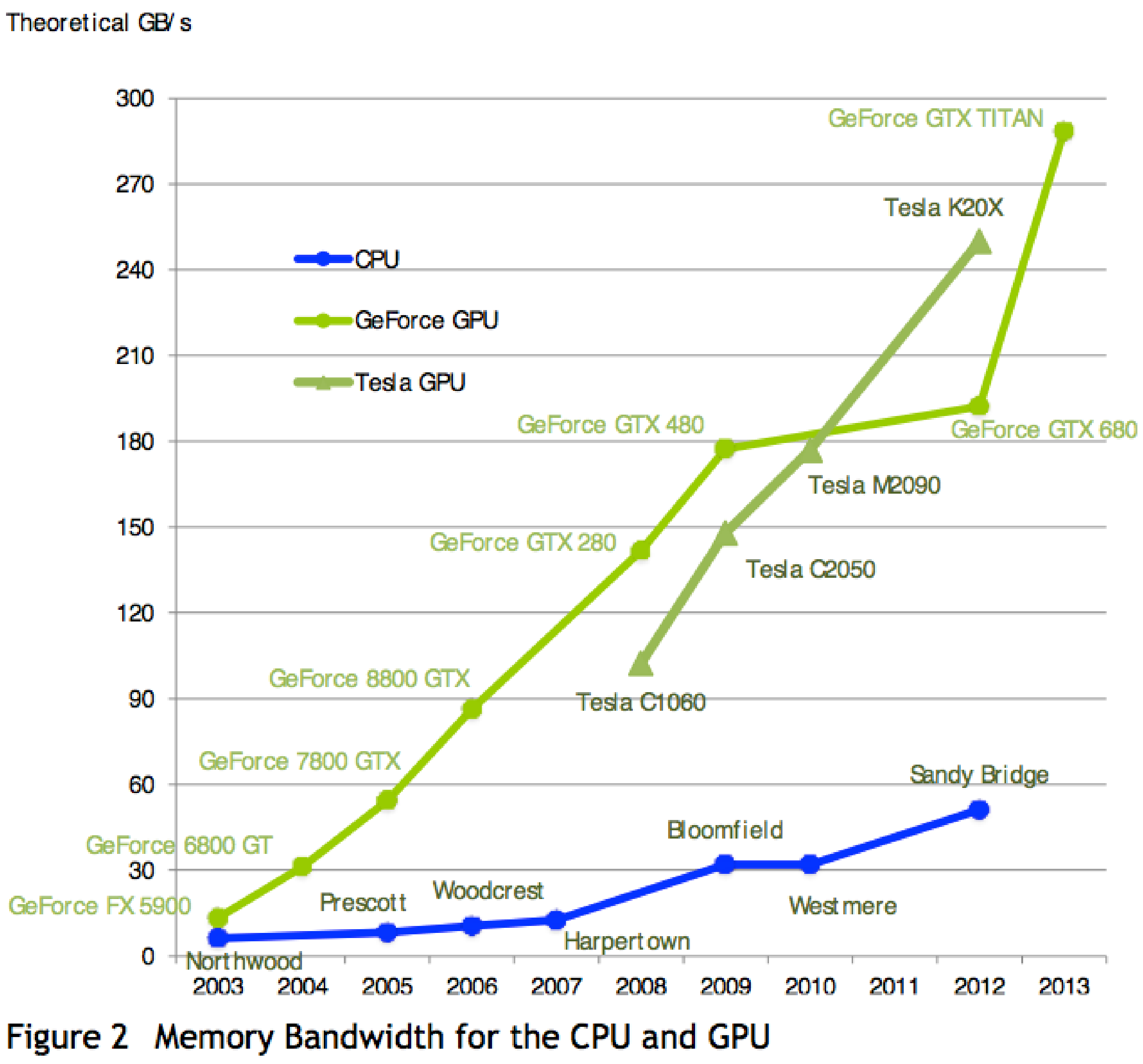


Fig. 4.4: Bandwidth comparision taken from the NVIDIA programming guide.

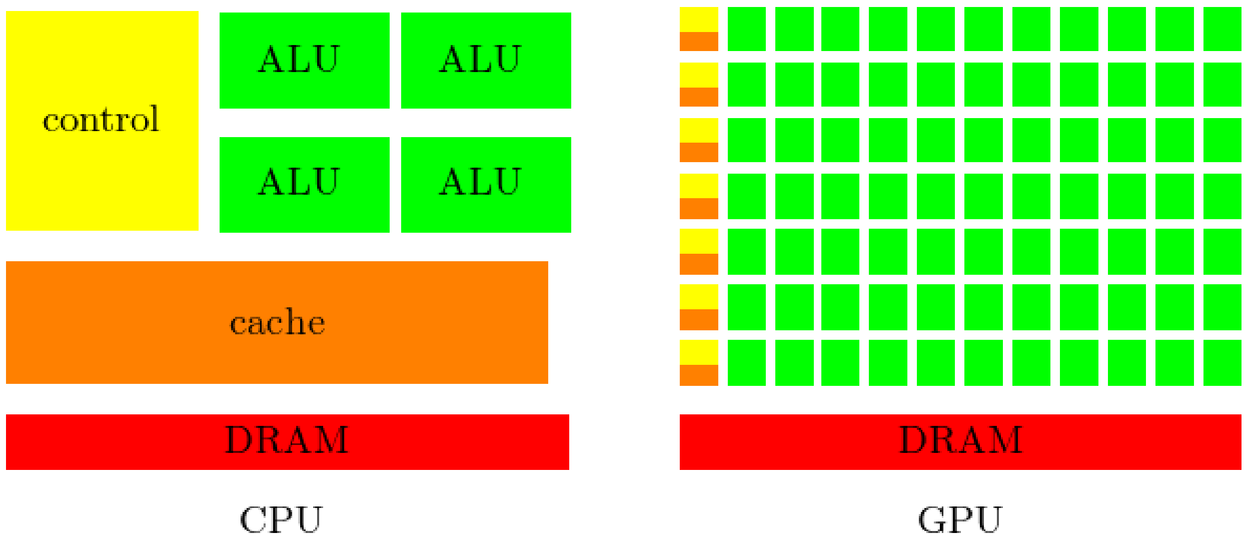


Fig. 4.5: Distinction between the design of a CPU and a GPU.

The architecture of a modern GPU is summarized in the following items:

- A CUDA-capable GPU is organized into an array of highly threaded Streaming Multiprocessors (SMs).
- Each SM has a number of Streaming Processors (SPs) that share control logic and an instruction cache.
- Global memory of a GPU consists of multiple gigabytes of Graphic Double Data Rate (GDDR) DRAM.
- Higher bandwidth makes up for longer latency.
- The growing size of global memory allows to keep data longer in global memory, with only occasional transfers to the CPU.
- A good application runs 10,000 threads simultaneously.

A concrete example of the GPU architecture is in Fig. 4.6.

Streaming multiprocessors support up to 2,048 threads. The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. Unlike CPU cores, threads are executed in order and there is no branch prediction, although instructions are pipelined.

According to David Kirk and Wen-mei Hwu (page 14): *Developers who are experienced with MPI and OpenMP will find CUDA easy to learn.* CUDA (Compute Unified Device Architecture) is a programming model that focuses on data parallelism.

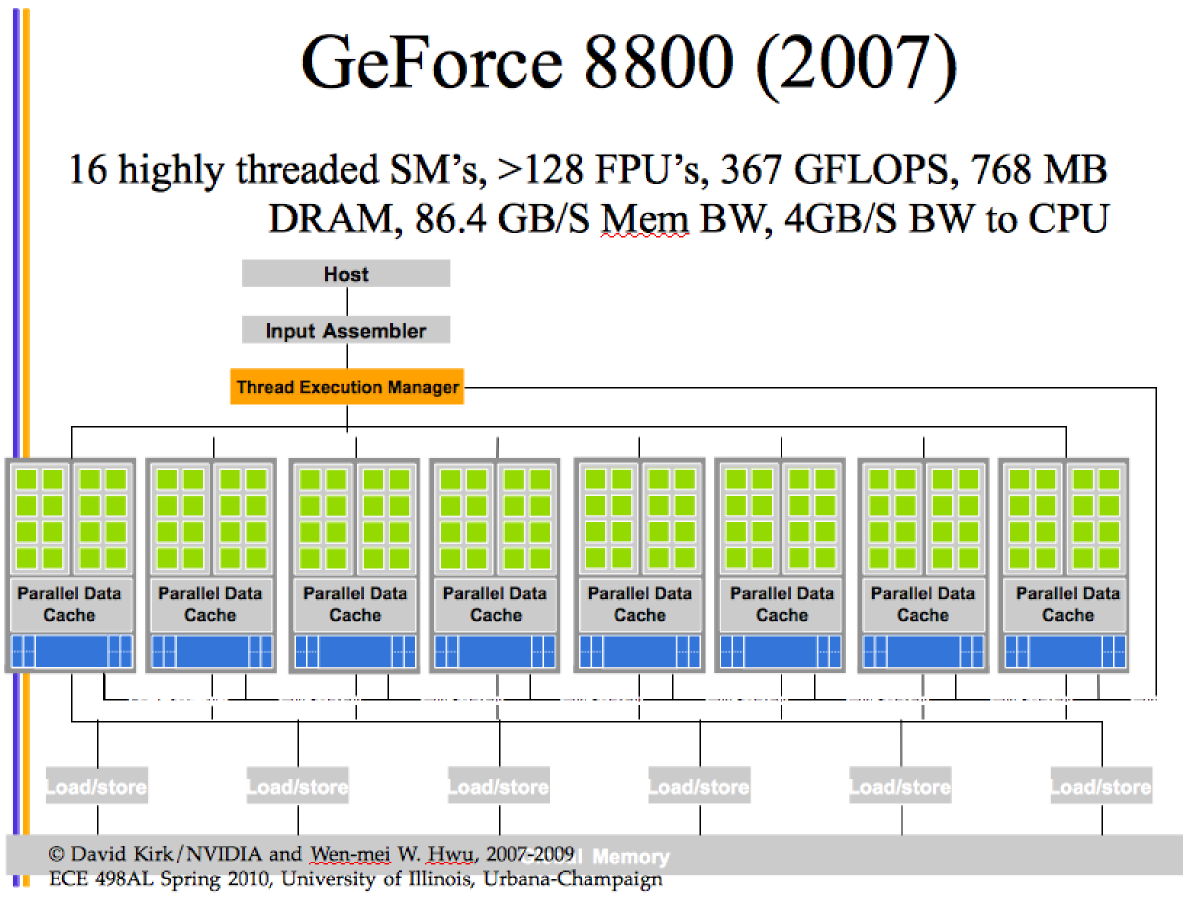
Data parallelism involves

- huge amounts of data on which
- the arithmetical operations are applied in parallel.

With MPI we applied the SPMD (Single Program Multiple Data) model. With GPGPU, the architecture is SIMT = Single Instruction Multiple Thread. An example with large amount of data parallelism is matrix-matrix multiplication in large dimensions. Available Software Development Tools (SDK), e.g.: BLAS, FFT are available for download at <http://www.nvidia.com>.

Alternatives to CUDA are

- OpenCL (chapter 14) for heterogeneous computing;
- OpenACC (chapter 15) uses directives like OpenMP;



GeForce 8800 (2007)

16 highly threaded SM's, >128 FPU's, 367 GFLOPS, 768 MB DRAM, 86.4 GB/S Mem BW, 4GB/S BW to CPU

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL Spring 2010, University of Illinois, Urbana-Champaign

Fig. 4.6: A concrete example of the GPU architecture.

- C++ Accelerated Massive Parallelism (chapter 18).

Extensions to CUDA are

- Thrust: productivity-oriented library for CUDA (chapter~16);
- CUDA FORTRAN (chapter 17);
- MPI/CUDA (chapter 19).

And then, of course, there is Julia, which provides packages for *vendor agnostic GPU computing*.

4.1.3 Bibliography

1. NVIDIA CUDA Programming Guide. Available at <http://developer.nvdia.com>.
2. Victor W. Lee et al: **Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU**. In *Proceedings of the 37th annual International Symposium on Computer Architecture (ISCA'10)*, ACM 2010.
3. W.W. Hwu (editor). **GPU Computing Gems: Emerald Edition**. Morgan Kaufmann, 2011.

4.1.4 Exercises

0. How strong is the graphics card in your computer?

4.2 Programming GPUs with PyCUDA and Julia

High level GPU programming can be done in Python or Julia. Before introducing PyCUDA and GPU programming with Julia, the data parallel model is described.

4.2.1 Data Parallelism

The programming model is Single Instruction Multiple Data (SIMD). In the application of data parallelism, blocks of threads read from memory, execute the same instruction(s), and then write the results back to memory. To fully occupy a massively parallel processor such as the GPU, at least 10,000 threads are needed.

Code that runs on the GPU is defined in a function, the so-called kernel.

The scalable programming model of the GPU is illustrated in Fig. 4.7.

Streaming multiprocessors schedule the blocks of threads for execution in groups of 32 threads, called a warp. The dual warp scheduler is illustrated in Fig. 4.8.

A kernel launch creates a grid of blocks, and each block has one or more threads. The organization of the grids and blocks can be 1D, 2D, or 3D. During the running of the kernel, threads in the same block are executed simultaneously.

The programming model for NVIDIA GPUs is called CUDA which stands for Compute Unified Device Architecture.

Programming GPUs is more complicated because it is not possible to quickly add print statements when debugging the code. As illustrated in Fig. 4.9 every GPU (called the device) is connected to a CPU (called the host). Kernels are lauched by the host for execution on the device.

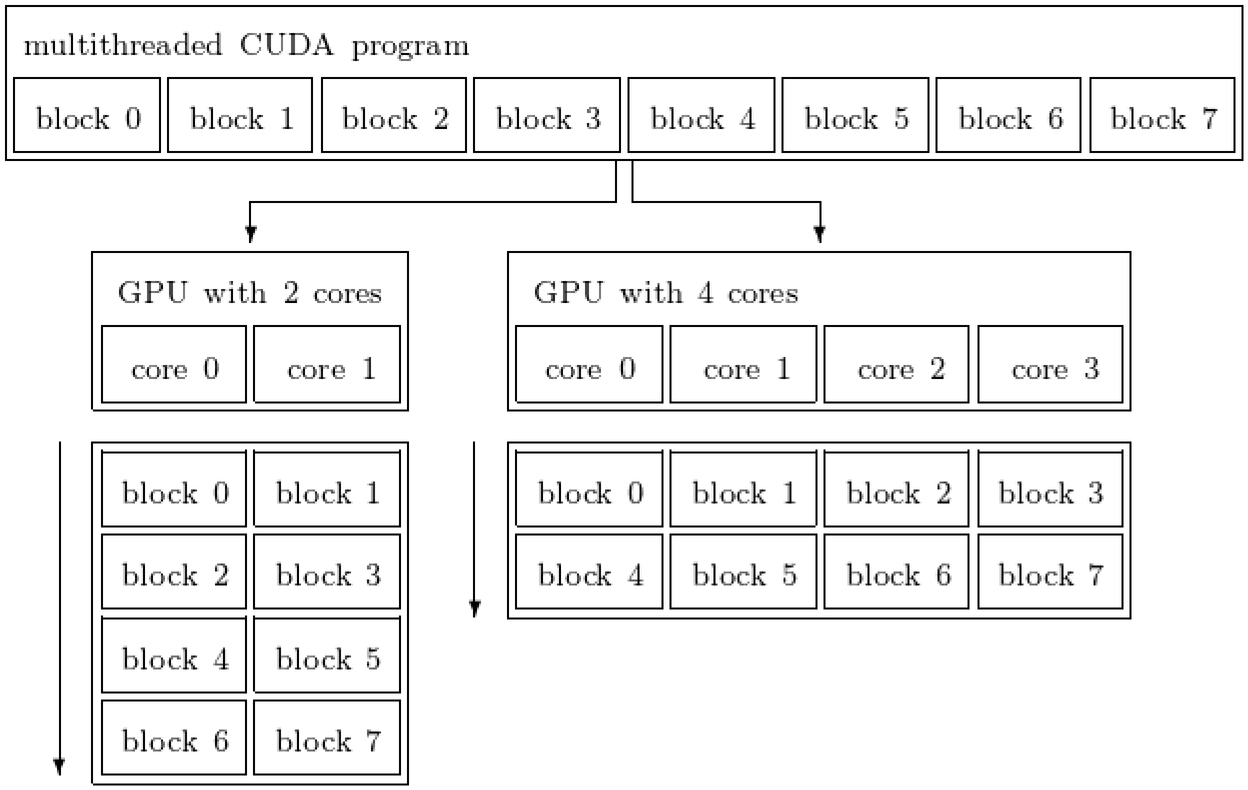


Fig. 4.7: A scalable programming model. Each core represents a streaming multiprocessor.

4.2.2 Matrix Matrix Multiplication

Our scientific running example of data parallel computations is the multiplication of two matrices. Consider the multiplication of matrices A and B which results in $C = A \cdot B$, with

$$A = [a_{i,j}] \in \mathbb{R}^{n \times m}, \quad B = [b_{i,j}] \in \mathbb{R}^{m \times p}, \quad C = [c_{i,j}] \in \mathbb{R}^{n \times p}.$$

$c_{i,j}$ is the inner product of the i-th row of A with the j-th column of B:

$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} \cdot b_{k,j}.$$

All $c_{i,j}$'s can be computed independently from each other. For $n = m = p = 1,024$ we have 1,048,576 inner products. The matrix multiplication is illustrated in Fig. 4.10.

4.2.3 PyCUDA

Code for the GPU can be generated in Python, see Fig. 4.11, as described in the following paper by A. Kloeckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih: **PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation.** *Parallel Computing*, 38(3):157–174, 2012.

To verify whether PyCUDA is correctly installed on our computer, we can run an interactive Python session as follows.

```
>>> import pycuda
>>> import pycuda.autoinit
```

(continues on next page)

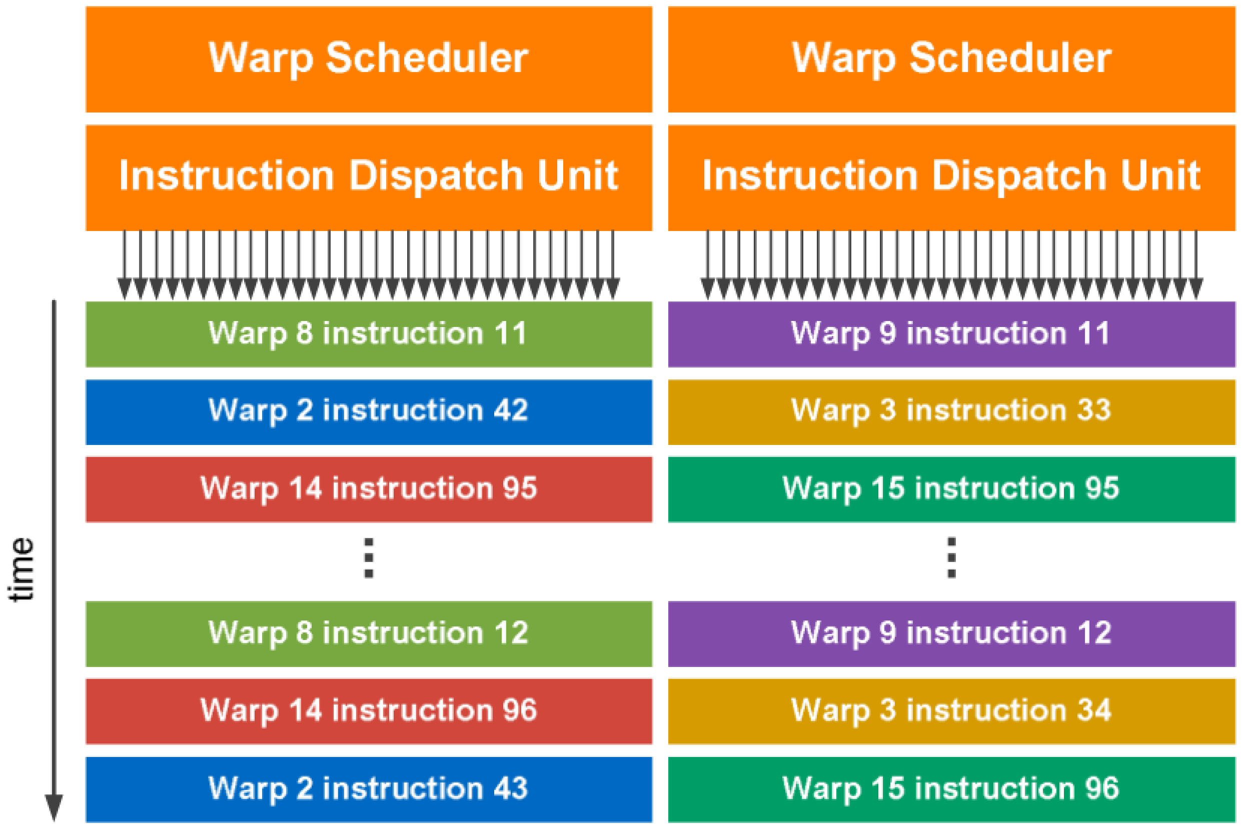


Fig. 4.8: Dual warp scheduler, copied from the NVIDIA documentation.

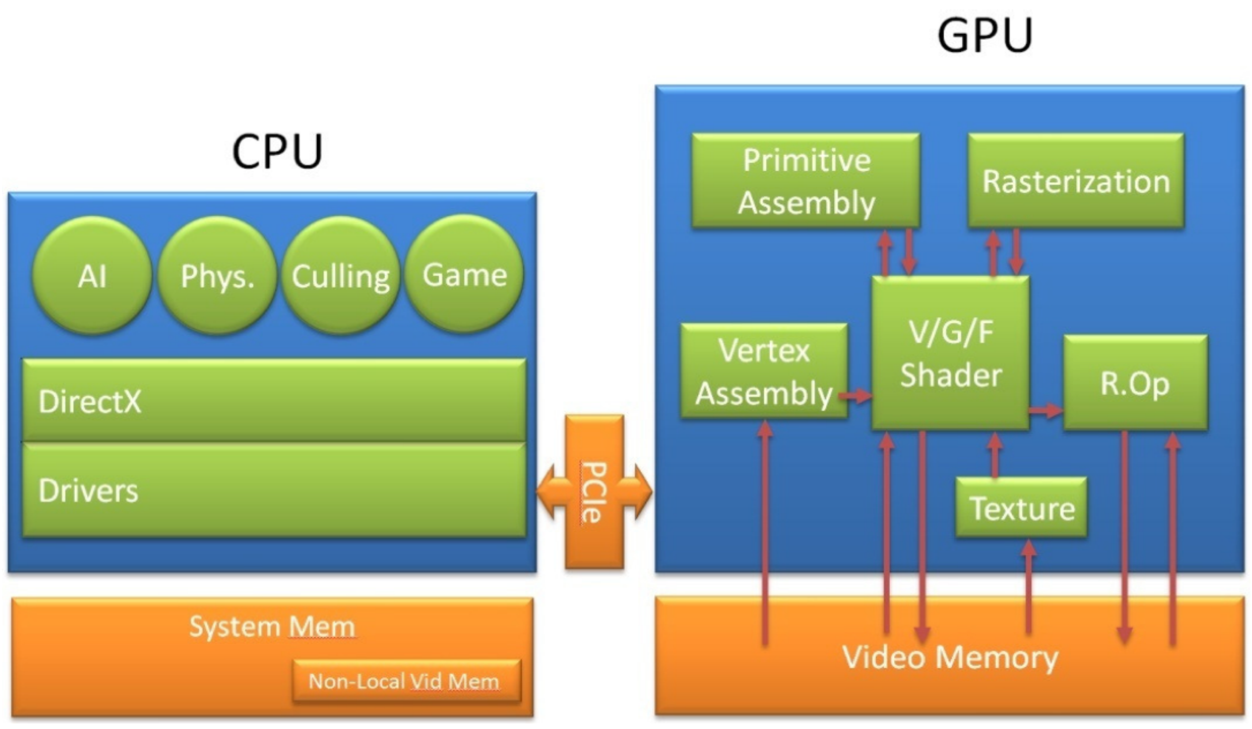


Fig. 4.9: From the GeForce 8 and 9 Series GPU Programming Guide (NVIDIA).

(continued from previous page)

```
>>> from pycuda.tools import make_default_context
>>> c = make_default_context()
>>> d = c.get_device()
>>> d.name()
'Tesla P100-PCIE-16GB'
```

We illustrate the matrix-matrix multiplication on the GPU with code generated in Python. We multipy an n-by-m matrix with an m-by-p matrix with a two dimensional grid of $n \times p$ threads. For testing we use 0/1 matrices.

```
$ python matmatmul.py
matrix A:
[[ 0.  0.  1.  0.]
 [ 0.  0.  1.  1.]
 [ 0.  1.  1.  0.]]
matrix B:
[[ 1.  1.  0.  1.  1.]
 [ 1.  0.  1.  0.  0.]
 [ 0.  0.  1.  1.  0.]
 [ 0.  0.  1.  1.  0.]]
multiplied A*B:
[[ 0.  0.  1.  1.  0.]
 [ 0.  0.  2.  2.  0.]
 [ 1.  0.  2.  1.  0.]]
$
```

The script starts with the import of the modules and type declarations.

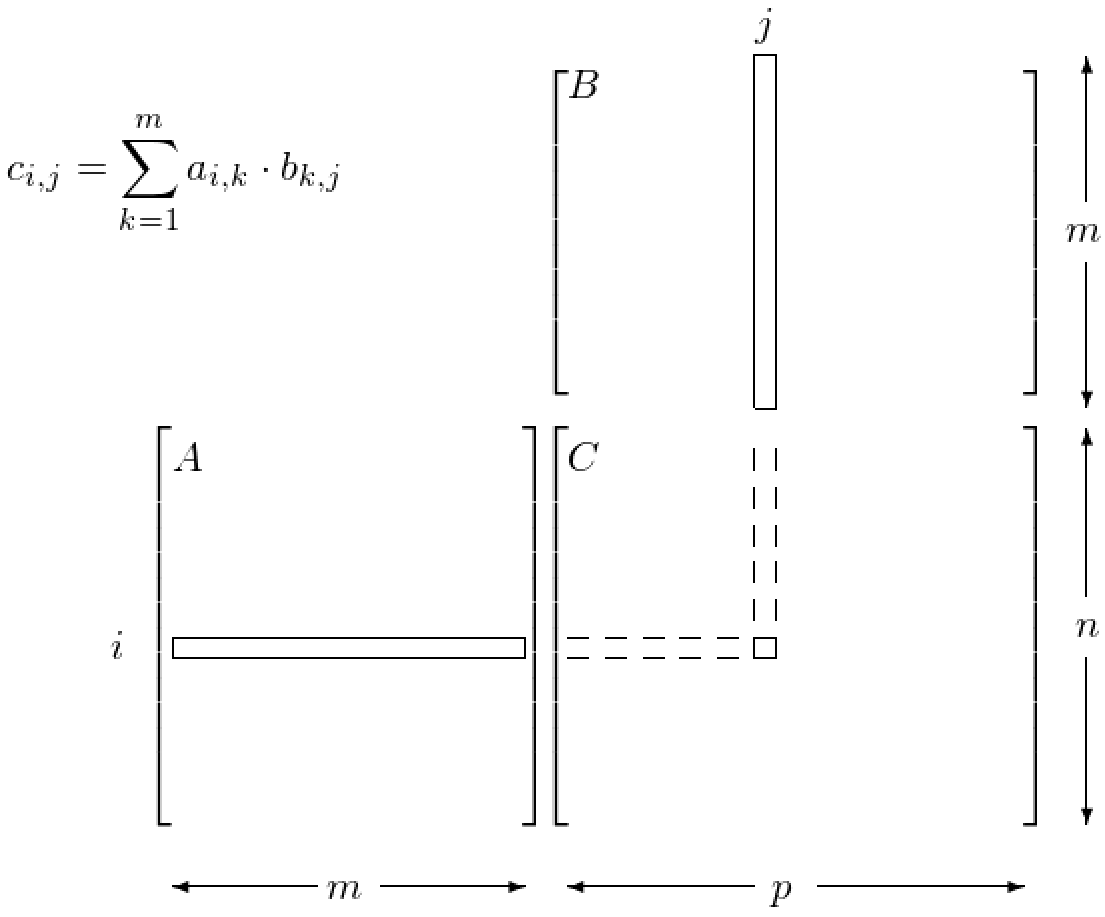


Fig. 4.10: Data parallelism in matrix multiplication.

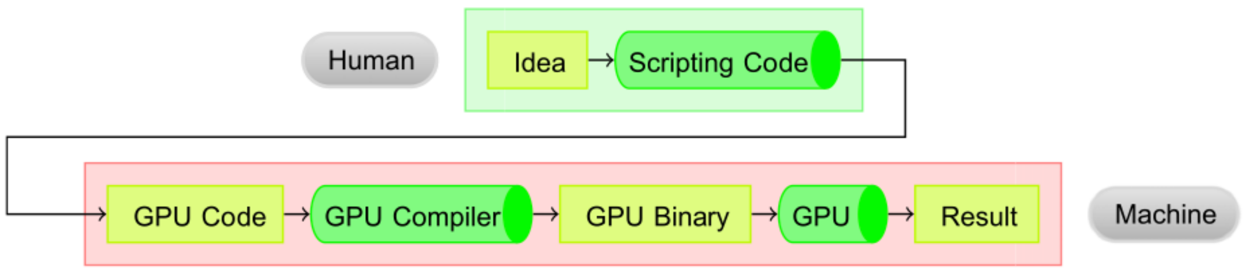


Fig. 4.11: The operating principle of GPU code generation.

```python
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy

(n, m, p) = (3, 4, 5)

n = numpy.int32(n)
m = numpy.int32(m)
p = numpy.int32(p)

a = numpy.random.randint(2, size=(n, m))
b = numpy.random.randint(2, size=(m, p))
c = numpy.zeros((n, p), dtype=numpy.float32)

a = a.astype(numpy.float32)
b = b.astype(numpy.float32)
```

The script then continues with the memory allocation and the copying from host to device.

```python
a_gpu = cuda.mem_alloc(a.size * a.dtype.itemsize)
b_gpu = cuda.mem_alloc(b.size * b.dtype.itemsize)
c_gpu = cuda.mem_alloc(c.size * c.dtype.itemsize)

cuda.memcpy_htod(a_gpu, a)
cuda.memcpy_htod(b_gpu, b)
```

The definition of the kernel follows:

```python
mod = SourceModule("""
    __global__ void multiply
      ( int n, int m, int p,
        float *a, float *b, float *c )
    {
        int idx = p*threadIdx.x + threadIdx.y;

        c[idx] = 0.0;
        for(int k=0; k<m; k++)
           c[idx] += a[m*threadIdx.x+k]
                    *b[threadIdx.y+k*p];
    }
    """)
```

To understand the code in the kernel, observe that a linear address system is used for the 2-dimensional array. Consider a 3-by-5 matrix stored row-wise (as in C), as shown in Fig. 4.12.

When defining the kernel, we assign inner products to threads. For example, consider a 3-by-4 matrix A and a 4-by-5 matrix B, as in Fig. 4.13.

The `i = blockIdx.x*blockDim.x + threadIdx.x` determines what entry in $C = A \cdot B$ will be computed:

- the row index in C is `i` divided by 5 and
- the column index in C is the remainder of `i` divided by 5.

The launching of the kernel and printing the result is the last stage.

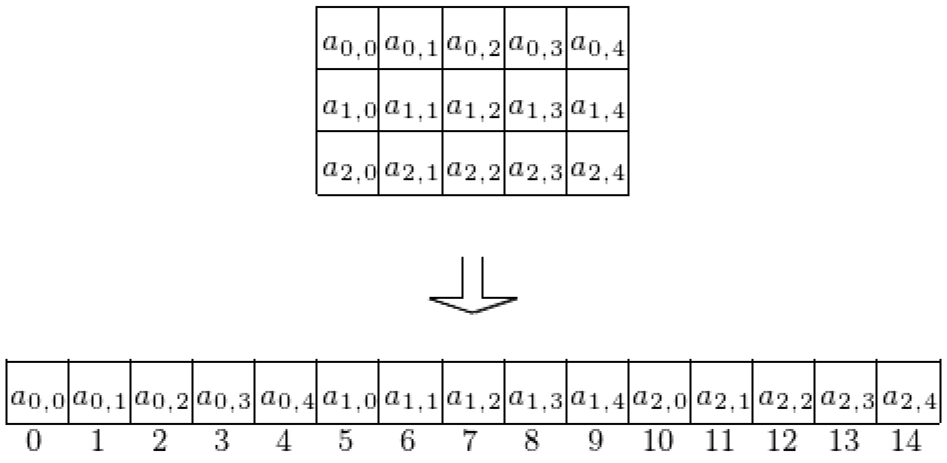

$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$ $a_{0,4}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$ $a_{0,4}$ $a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$ $a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Fig. 4.12: Storing a matrix as a one dimensional array.

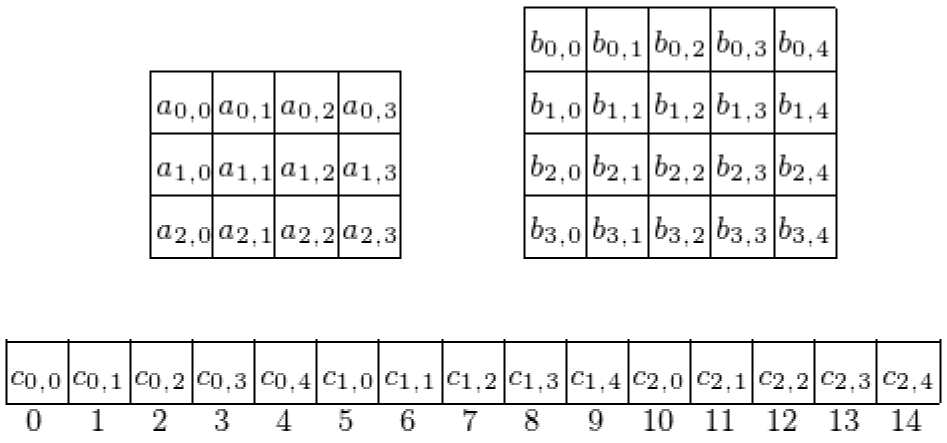

$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$ $b_{0,4}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$ $b_{1,4}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$ $b_{2,4}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$ $b_{3,4}$
$c_{0,0}$ $c_{0,1}$ $c_{0,2}$ $c_{0,3}$ $c_{0,4}$ $c_{1,0}$ $c_{1,1}$ $c_{1,2}$ $c_{1,3}$ $c_{1,4}$ $c_{2,0}$ $c_{2,1}$ $c_{2,2}$ $c_{2,3}$ $c_{2,4}$
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Fig. 4.13: Assigning inner products to threads.

```
func = mod.get_function("multiply")
func(n, m, p, a_gpu, b_gpu, c_gpu, \
     block=(int(n), int(p), 1), \
     grid=(1, 1), shared=0)

cuda.memcpy_dtoh(c, c_gpu)

print("matrix A:")
print(a)
print("matrix B:")
print(b)
print("multiplied A*B:")
print(c)
```

4.2.4 Vendor Agnostic GPU Computing in Julia

The package `CUDA.jl` allows GPU programming in Julia, on computers with an NVIDIA GPU and provided that the CUDA Software Development Kit is installed. The compilation process for Julia is illustrated in Fig. 4.14.

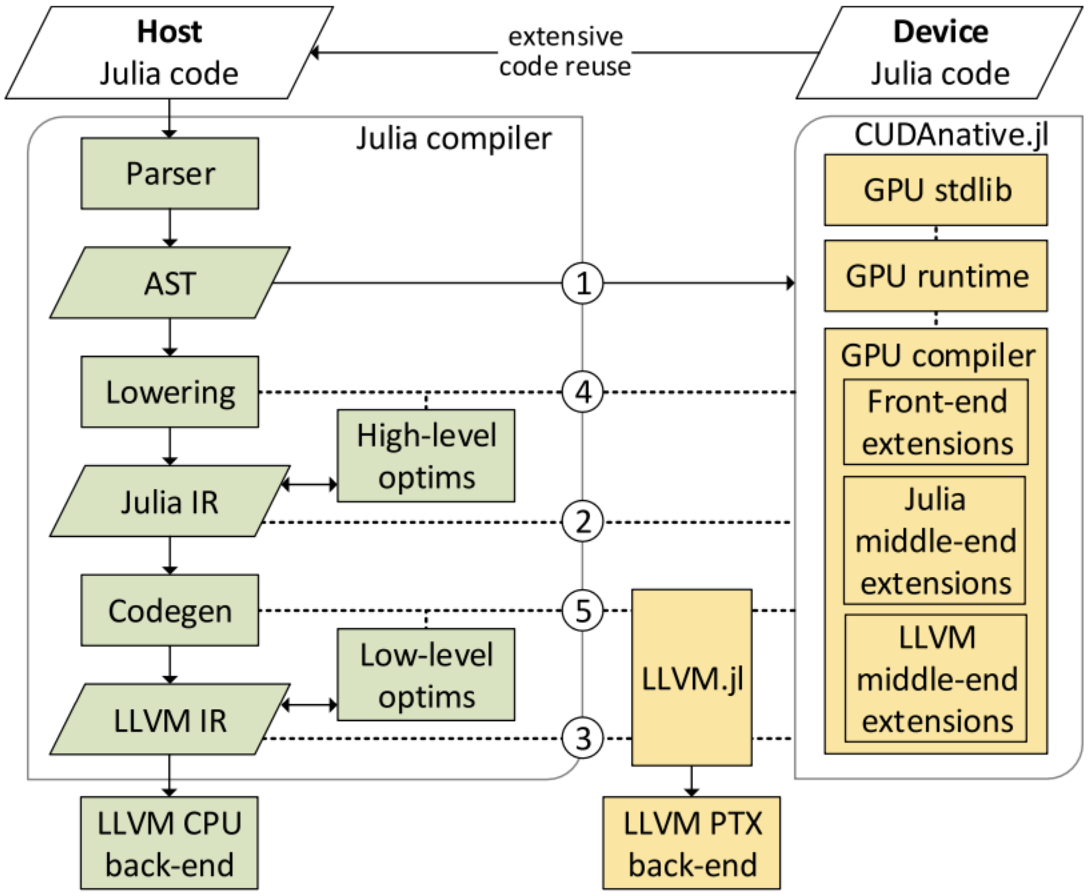


Fig. 4.14: The compilation process, copied from Besard et al., 2019.

The site JuliaGPU documents the organization to unify the many packages for programming GPUs in Julia. A first kernel is taken from the tutorials at the JuliaGPU site:

```
using CUDA
using Test

function gpu_add1!(y, x)
    for i = 1:length(y)
        @inbounds y[i] += x[i]
    end
    return nothing
end

N = 2^20             # adding one million Float32 numbers
x_d = CUDA.fill(1.0f0, N) # stored on GPU filled with 1.0
y_d = CUDA.fill(2.0f0, N) # stored on GPU filled with 2.0

fill!(y_d, 2)
@cuda gpu_add1!(y_d, x_d)
result = (@test all(Array(y_d) .== 3.0f0))
println(result)
```

Observe that, unlike with PyCUDA, the kernel is defined as a Julia function.

To make the point of {vendor agnostic GPU computing, in addition to CUDA.jl, the following packages are available:

- AMDGPU.jl for AMD GPUs,
- oneAPI.jl for the Intel oneAPI,
- Metal.jl to program GPUs in Apple hardware.

The code to add vectors on an M1 MacBook Air GPU with Metal looks very similar to the code with CUDA.jl.

```
using Metal
using Test

function gpu_add1!(y, x)
    for i = 1:length(y)
        @inbounds y[i] += x[i]
    end
    return nothing
end

N = 32
x_d = Metal.fill(1.0f0, N)  # filled with Float32 1.0 on GPU
y_d = Metal.fill(2.0f0, N)  # filled with Float32 2.0

# run with N threads

@metal threads=N gpu_add1!(y_d, x_d)
result = (@test all(Array(y_d) .== 3.0f0))

println(result)
```

As a last illustration, consider the multiplying of matrices with `Metal`. The CUDA version of the example is copied from the Julia for High-Performance Scientific Computing web site, adjusted

1. using `Metal` instead of using `CUDA`,

2. work with `Float32` instead of `Float64`,
3. use `MtlArray` instead of `CuArray`.

The code is below, using the operator *:

```
using Metal
using BenchmarkTools

dim = 2^10
A_h = rand(Float32, dim, dim);
A_d = MtlArray(A_h);

@btime $A_h * $A_h;
@btime $A_d * $A_d;
```

When executed on a 2020 M1 Macbook Air, what is printed is 6.229 ms and 23.625 μs for CPU and GPU respectively. Observe the difference in orders of magnitude, milliseconds versus microseconds. Even as the GPU on this laptop is not intended for high performance scientific computations, it outperforms the CPU.

4.2.5 Bibliography

1. T. Besard, C. Foket, and B. De Sutter: **Effective Extensible Programming: Unleashing Julia on GPUs.** *IEEE Transactions on Parallel and Distributed Systems*, Vol. 30, No. 4, pages 827–841, 2019.
2. A. Kloeckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. **PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation.** *Parallel Computing*, 38(3):157–174, 2012.
3. U. Utkarsh, V. Churavy, Y. Ma, T. Besard, P. Srisuma, T. Gymnich, A. R. Gerlach, A. Edelman, G. Barbastathis, R. D. Braatz, and C. Rackauckas: **Automated Translation and Accelerated Solving of Differential Equations on Multiple GPU Platforms.** *Computer Methods in Applied Mechanics and Engineering*, Vol. 419, 2024, article 116591.

4.2.6 Exercises

1. The matrix matrix multiplication example with PyCUDA uses $(3, 4, 5)$ as values for (n, m, p). Considering massive parallelism, what are the largest dimensions you could consider for one block of threads on the P100 and/or the A100? Illustrate your values for the dimensions experimentally.
2. In the PyCUDA matrix matrix multiplication, change the `float32` types into `float64` and redo the previous exercise. Time the code. Do you notice a difference?
3. On your own computer, check the vendor of the GPU and run the equivalent `gpuadd.jl` after installing the proper Julia package. Report on the performance, relative to the CPU in your computer.

4.3 Introduction to CUDA

CUDA (Compute Unified Device Architecture) is the parallel programming platform for the NVIDIA GPUs, for general purpose processing.

4.3.1 Our first GPU Program

We will run Newton's method in complex arithmetic as our first CUDA program.

To compute $\sqrt{c}$ for $c \in c$, we apply Newton's method on $x^2 - c = 0$:

$$x_0 := c, \quad x_{k+1} := x_k - \frac{x_k^2 - c}{2x_k}, \quad k = 0, 1, \ldots$$

Five iterations suffice to obtain an accurate value for $\sqrt{c}$.

Finding roots is relevant for scientific computing. But, is this computation suitable for the GPU? The data parallelism we can find in this application is that we can run Newton's method for many different c's. With a little more effort, the code in this section can be extended to a complex root finder for polynomials in one variable.

Definition of Compute Capability

The *compute capability* of an NVIDIA GPU is represented by a version number in the format x.y, it identifies the features supported by the hardware.

What does compute capability mean for the programmer? Some examples of the supported features:

- 1.3: double-precision floating-point operations
- 2.0: synchronizing threads
- 3.5: dynamic parallelism
- 5.3: half-precision floating-point operations
- 6.0: atomic addition operation on 64-bit floats
- 8.0: tensor cores supporting double float precision

The compute capability is not the same as the CUDA version.

To examine the CUDA Compute Capability, we check the card with `deviceQuery`. Below is the result on a computer with a Pascal P100 NVIDIA GPU.

```
$ /usr/local/cuda/samples/1_Utilities/deviceQuery/deviceQuery
/usr/local/cuda/samples/1_Utilities/deviceQuery/deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 2 CUDA Capable device(s)

Device 0: "Tesla P100-PCIE-16GB"
  CUDA Driver Version / Runtime Version          11.0 / 8.0
  CUDA Capability Major/Minor version number:    6.0
  Total amount of global memory:                 16276 MBytes (17066885120 bytes)
  (56) Multiprocessors, ( 64) CUDA Cores/MP:     3584 CUDA Cores
```

(continues on next page)

(continued from previous page)

```
  GPU Max Clock rate:                            405 MHz (0.41 GHz)
  Memory Clock rate:                             715 Mhz
  Memory Bus Width:                              4096-bit
  L2 Cache Size:                                 4194304 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536),␣
↪3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 2 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device␣
↪simultaneously) >
```

Another standard check is the bandwidthTest, which runs as below on the same computer.

```
$ /usr/local/cuda/samples/1_Utilities/bandwidthTest/bandwidthTest
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: Tesla P100-PCIE-16GB
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)     Bandwidth(MB/s)
   33554432                  11530.1

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)     Bandwidth(MB/s)
   33554432                  12848.3

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)     Bandwidth(MB/s)
```

(continues on next page)

(continued from previous page)

```
   33554432                  444598.8

Result = PASS

$
```

It is interesting to compare with a run on a computer which hosts an Ampere A100 NVIDIA GPU. The output of `DeviceQuery` follows.

```
$ /usr/local/cuda/samples/bin/x86_64/linux/release/deviceQuery
/usr/local/cuda/samples/bin/x86_64/linux/release/deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA A100 80GB PCIe"
  CUDA Driver Version / Runtime Version          12.4 / 12.4
  CUDA Capability Major/Minor version number:    8.0
  Total amount of global memory:                 81038 MBytes (84974239744 bytes)
  (108) Multiprocessors, (064) CUDA Cores/MP:    6912 CUDA Cores
  GPU Max Clock rate:                            1410 MHz (1.41 GHz)
  Memory Clock rate:                             1512 Mhz
  Memory Bus Width:                              5120-bit
  L2 Cache Size:                                 41943040 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536),␣
↪3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        167936 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 202 / 0
```

(continues on next page)

(continued from previous page)

```
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device␣
↪simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 12.4, CUDA Runtime
Version = 12.4, NumDevs = 1
Result = PASS
```

The output of `bandwidthTest` is below:

```
$ /usr/local/cuda/samples/bin/x86_64/linux/release/bandwidthTest
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: NVIDIA A100 80GB PCIe
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(GB/s)
   32000000                     25.2

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(GB/s)
   32000000                     26.3

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(GB/s)
   32000000                     1313.8

Result = PASS

$
```

Observe the differences in magnitude between the A100 and the P100.

4.3.2 CUDA Program Structure

There are five steps to get GPU code running:

1. C and C++ functions are labeled with CUDA keywords `__device__`, `__global__`, or `__host__`.
2. Determine the data for each thread to work on.
3. Transferring data from/to host (CPU) to/from the device (GPU).
4. Statements to launch data-parallel functions, called *kernels*.
5. Compilation with `nvcc`.

The compilation process is illustrated in Fig. 4.15.

We will now examine every step in greater detail.

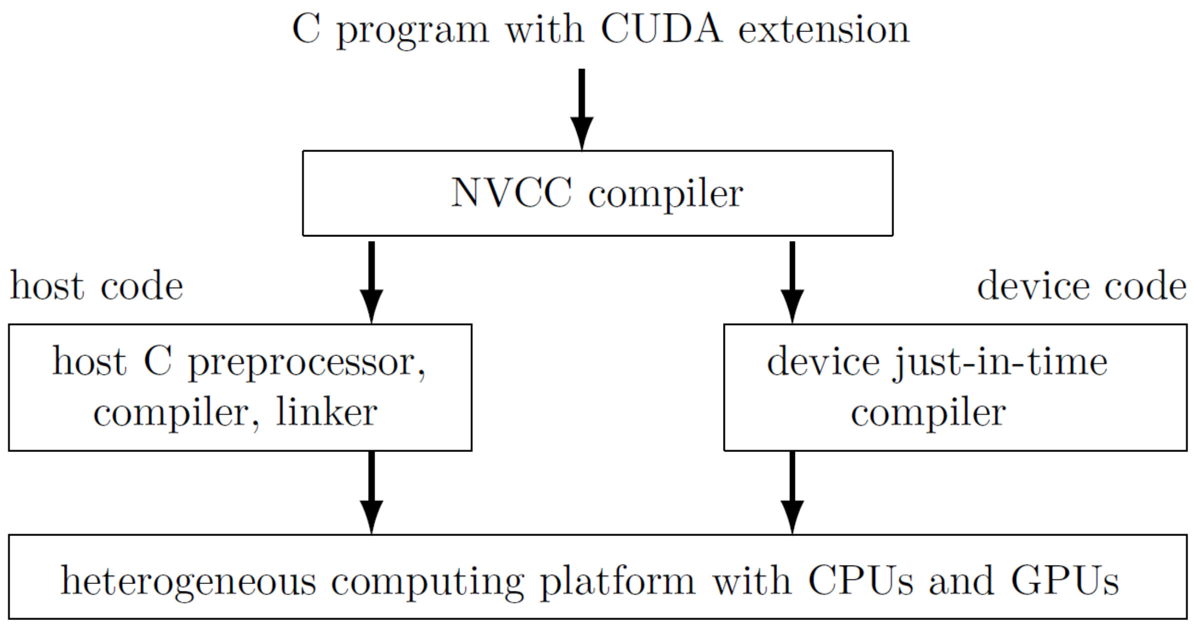

C program with CUDA extension
NVCC compiler
host code
device code
host C preprocessor, compiler, linker
device just-in-time compiler
heterogeneous computing platform with CPUs and GPUs

Fig. 4.15: The compilation process with the NVIDIA compiler.

In the first step, we add CUDA extensions to functions. We distinguish between three keywords before a function declaration:

- `__host__`: The function will run on the host (CPU).
- `__device__`: The function will run on the device (GPU).
- `__global__`: The function is called from the host but runs on the device. This function is called a *kernel*.

The CUDA extensions to C function declarations are summarized in Table 4.1.

Table 4.1: CUDA extensions to C function declarations.

	executed on	callable from
`__device__ double D()`	device	device
`__global__ void K()`	device	host
`__host__ int H()`	host	host

In the second step, we determine the data for each thread. The grid consists of N blocks, with `blockIdx.x` $\in \{0, N-1\}$. Within each block, `threadIdx.x` $\in \{0,$ `blockDim.x` $- 1\}$. This second step is illustrated in Fig. 4.16, taken from the techical blog on *An Even Easier Introduction to CUDA* by Mark Harris.

In the third step, data is allocated and transferred from the host to the device. We illustrate this step with the code below.

```
cudaDoubleComplex *xhost = new cudaDoubleComplex[n];

// we copy n complex numbers to the device
size_t s = n*sizeof(cudaDoubleComplex);
cudaDoubleComplex *xdevice;
cudaMalloc((void**)&xdevice,s);

```

(continues on next page)

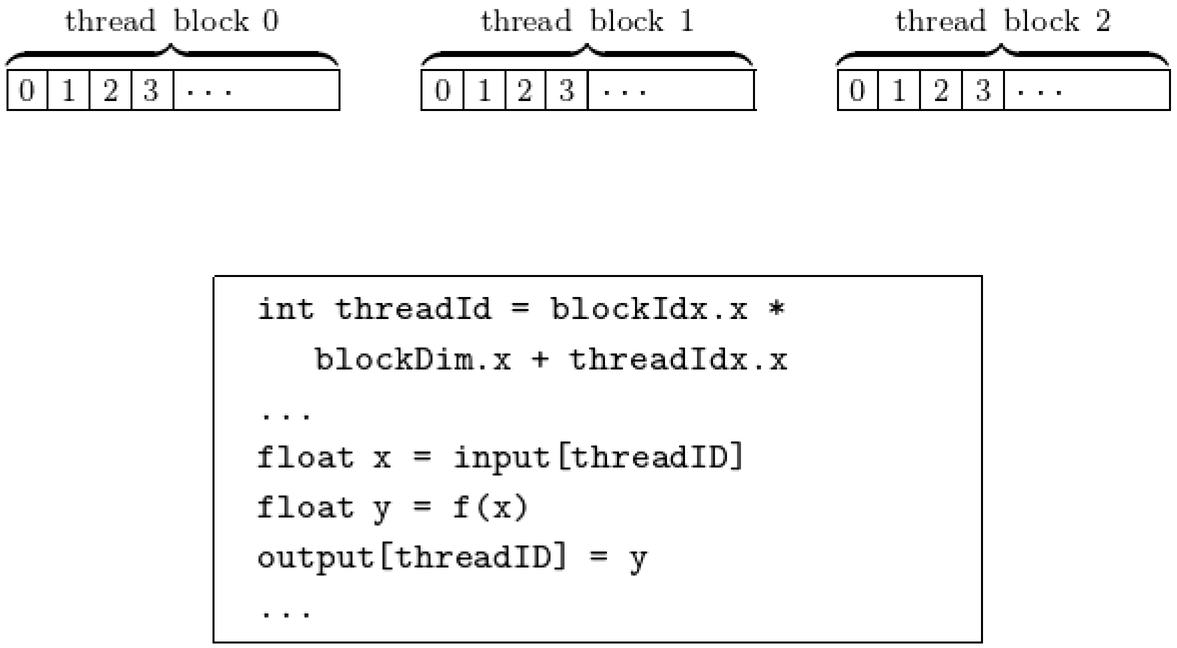


Fig. 4.16: Defining the data for each thread.

(continued from previous page)

```
cudaMemcpy(xdevice,xhost,s,cudaMemcpyHostToDevice);

// allocate memory for the result
cudaDoubleComplex *ydevice;
cudaMalloc((void**)&ydevice,s);

// copy results from device to host
cudaDoubleComplex *yhost = new cudaDoubleComplex[n];

cudaMemcpy(yhost,ydevice,s,cudaMemcpyDeviceToHost);
```

In the fourth step, the kernel is launched. The kernel is declared as

```
__global__ void squareRoot
 ( int n, cudaDoubleComplex *x, cudaDoubleComplex *y )
// Applies Newton's method to compute the square root
// of the n numbers in x and places the results in y.
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   ...
```

For frequency `f`, dimension `n`, and block size `w`, we do:

```
// invoke the kernel with n/w blocks per grid
// and w threads per block
for(int i=0; i<f; i++)
   squareRoot<<<n/w,w>>>(n,xdevice,ydevice);
```

In the fifth step, the code is compiled with `nvcc`. Then, if the `makefile` contains

```
runCudaComplexSqrt:
        nvcc -o /tmp/run_cmpsqrt -arch=sm_13 \
                runCudaComplexSqrt.cu
```

typing `make runCudaComplexSqrt` does

```
nvcc -o /tmp/run_cmpsqrt -arch=sm_13 runCudaComplexSqrt.cu
```

The `-arch=sm_13` is needed for `double` arithmetic.

The code to compute complex roots is below. Complex numbers and their arithmetic is defined on the host and on the device.

```
#ifndef __CUDADOUBLECOMPLEX_CU__
#define __CUDADOUBLECOMPLEX_CU__

#include <cmath>
#include <cstdlib>
#include <iomanip>
#include <vector_types.h>
#include <math_functions.h>

typedef double2 cudaDoubleComplex;
```

We use the `double2` of `vector_types.h` to define complex numbers because `double2` is a native CUDA type allowing for coalesced memory access.

Random complex numbers are generated with the function below.

```
__host__ cudaDoubleComplex randomDoubleComplex()
// Returns a complex number on the unit circle
// with angle uniformly generated in [0,2*pi].
{
   cudaDoubleComplex result;
   int r = rand();
   double u = double(r)/RAND_MAX;
   double angle = 2.0*M_PI*u;
   result.x = cos(angle);
   result.y = sin(angle);
   return result;
}
```

Calling `sqrt` of `math_functions.h` is done in the function below.

```
__device__ double radius ( const cudaDoubleComplex c )
// Returns the radius of the complex number.
{
   double result;
   result = c.x*c.x + c.y*c.y;
   return sqrt(result);
}
```

We overload the output operator.

```
__host__ std::ostream& operator<<
 ( std::ostream& os, const cudaDoubleComplex& c)
// Writes real and imaginary parts of c,
// in scientific notation with precision 16.
{
   os << std::scientific << std::setprecision(16)
      << c.x << "  " << c.y;
   return os;
}
```

Complex addition is defined with operator overloading, as in the function below.

```
__device__ cudaDoubleComplex operator+
 ( const cudaDoubleComplex a, const cudaDoubleComplex b )
// Returns the sum of a and b.
{
   cudaDoubleComplex result;
   result.x = a.x + b.x;
   result.y = a.y + b.y;
   return result;
}
```

The rest of the arithmetical operations are defined in a similar manner. All definitions related to complex numbers are stored in the file `cudaDoubleComplex.cu`.

The kernel function to compute the square root is listed below.

```
#include "cudaDoubleComplex.cu"

__global__ void squareRoot
 ( int n, cudaDoubleComplex *x, cudaDoubleComplex *y )
// Applies Newton's method to compute the square root
// of the n numbers in x and places the results in y.
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   cudaDoubleComplex inc;
   cudaDoubleComplex c = x[i];
   cudaDoubleComplex r = c;
   for(int j=0; j<5; j++)
   {
      inc = r + r;
      inc = (r*r - c)/inc;
      r = r - inc;
   }
   y[i] = r;
}
```

The main function takes command line arguments as defined below.

```
int main ( int argc, char*argv[] )
{
   if(argc < 5)
   {
      cout << "call with 4 arguments : " << endl;
```

(continues on next page)

(continued from previous page)

```
      cout << "dimension, block size, frequency, and check (0 or 1)"
           << endl;
   }
   else
   {
      int n = atoi(argv[1]); // dimension
      int w = atoi(argv[2]); // block size
      int f = atoi(argv[3]); // frequency
      int t = atoi(argv[4]); // test or not
```

The main program generates n random complex numbers with radius one. After the generation of the data, the data is transferred and the kernel is launched.

```
// we generate n random complex numbers on the host
cudaDoubleComplex *xhost = new cudaDoubleComplex[n];
for(int i=0; i<n; i++) xhost[i] = randomDoubleComplex();
// copy the n random complex numbers to the device
size_t s = n*sizeof(cudaDoubleComplex);
cudaDoubleComplex *xdevice;
cudaMalloc((void**)&xdevice,s);
cudaMemcpy(xdevice,xhost,s,cudaMemcpyHostToDevice);
// allocate memory for the result
cudaDoubleComplex *ydevice;
cudaMalloc((void**)&ydevice,s);
// invoke the kernel with n/w blocks per grid
// and w threads per block
for(int i=0; i<f; i++)
   squareRoot<<<n/w,w>>>(n,xdevice,ydevice);
// copy results from device to host
cudaDoubleComplex *yhost = new cudaDoubleComplex[n];
cudaMemcpy(yhost,ydevice,s,cudaMemcpyDeviceToHost);
```

To verify the correctness, there is the option to test one random number.

```
      if(t == 1) // test the result
      {
         int k = rand() % n;
         cout << "testing number " <<  k << endl;
         cout << "        x = " << xhost[k] << endl;
         cout << "  sqrt(x) = " << yhost[k] << endl;
         cudaDoubleComplex z = Square(yhost[k]);
         cout << "sqrt(x)^2 = " << z << endl;
      }
   }
   return 0;
}
```

To run the code, we first test the correctness.

```
$ /tmp/run_cmpsqrt 1 1 1 1
testing number 0
        x = 5.3682227446949737e-01  -8.4369535119816541e-01
  sqrt(x) = 8.7659063264145631e-01  -4.8123680528950746e-01
```

(continues on next page)

(continued from previous page)

```
sqrt(x)^2 = 5.3682227446949726e-01  -8.4369535119816530e-01
```

On 64,000 numbers, 32 threads in a block, doing it 10,000 times:

```
$ time /tmp/run_cmpsqrt 64000 32 10000 1
testing number 50325
        x = 7.9510606509728776e-01  -6.0647039931517477e-01
  sqrt(x) = 9.4739275517002119e-01  -3.2007337822967424e-01
sqrt(x)^2 = 7.9510606509728765e-01  -6.0647039931517477e-01

real    0m1.618s
user    0m0.526s
sys     0m0.841s
```

Then we change the number of thread in a block. The output below are from the runs are on the NVIDIA P100.

```
$ time ./run_cmpsqrt 128000 32 100000 0

real    0m1.639s
user    0m0.989s
sys     0m0.650s

$ time ./run_cmpsqrt 128000 64 100000 0

real    0m1.640s
user    0m1.001s
sys     0m0.639s

$ time ./run_cmpsqrt 128000 128 100000 0

real    0m1.652s
user    0m0.952s
sys     0m0.700s
```

In five steps we wrote our first complete CUDA program. We started chapter 3 of the textbook by Kirk & Hwu, covering more of the CUDA Programming Guide. Available in `/usr/local/cuda/doc` and at <http://www.nvidia.com> are the *CUDA C Best Practices Guide* and the *CUDA Programming Guide*. Many examples of CUDA applications are available in `/usr/local/cuda/samples`.

4.3.3 using CUDA.jl

To illustrate the launching of blocks of threads in Julia with CUDA.jl, we consider again the addition of two vectors, using the example copied from the CUDA.jl tutorial. The complete code is listed below.

```
using CUDA
using Test

function gpu_add3!(y, x)
    index = (blockIdx().x - 1) * blockDim().x
          + threadIdx().x
    stride = gridDim().x * blockDim().x
    for i = index:stride:length(y)
```

(continues on next page)

(continued from previous page)

```
        @inbounds y[i] += x[i]
    end
    return
end

N = 2^20
x_d = CUDA.fill(1.0f0, N) # N Float32 1.0 on GPU
y_d = CUDA.fill(2.0f0, N) # N Float32 2.0

# run with 256 threads per block

numblocks = ceil(Int, N/256)
@cuda threads=256 blocks=numblocks gpu_add3!(y_d, x_d)
result = (@test all(Array(y_d) .== 3.0f0))

println(result)
```

Observe the launching of the kernel with 256 threads per block after the computation of the number of blocks.

4.3.4 Exercises

1. Instead of 5 Newton iterations in `runCudaComplexSqrt.cu` use `k` iterations where `k` is entered by the user at the command line. What is the influence of `k` on the timings?
2. Modify the kernel for the complex square root so it takes on input an array of complex coefficients of a polynomial of degree d. Then the root finder applies Newton's method, starting at random points. Test the correctness and experiment to find the rate of success, i.e.: for polynomials of degree d how many random trials are needed to obtain $d/2$ roots of the polynomial?
3. Use the kernel in a python script with PyCUDA.
4. Use CUDA.jl (or Metal.jl, oneAPI.jl, AMDGPU.jl on your GPU) for the square roots example.

4.4 Data Parallelism and Matrix Multiplication

Matrix multiplication is one of the fundamental building blocks in numerical linear algebra, and therefore in scientific computation. In this section, we investigate how data parallelism may be applied to solve this problem.

4.4.1 Data Parallelism

Many applications process large amounts of data. Data parallelism refers to the property where many arithmetic operations can be safely performed on the data simultaneously. Consider the multiplication of matrices A and B which results in $C = A \cdot B$, with

$$A = [a_{i,j}] \in \mathbb{R}^{n \times m}, \quad B = [b_{i,j}] \in \mathbb{R}^{m \times p}, \quad C = [c_{i,j}] \in \mathbb{R}^{n \times p}.$$

$c_{i,j}$ is the inner product of the *i*-th row of A with the *j*-th column of B:

$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} \cdot b_{k,j}.$$

All $c_{i,j}$'s can be computed independently from each other. For $n = m = p = 1,024$ we have 1,048,576 inner products. The matrix multiplication is illustrated in Fig. 4.10.

4.4.2 Code for Matrix-Matrix Multiplication

Code for a device (the GPU) is defined in functions using the keyword `__global__` before the function definition. Data parallel functions are called *kernels*. Kernel functions generate a large number of threads.

In matrix-matrix multiplication, the computation can be implemented as a kernel where each thread computes one element in the result matrix. To multiply two 1,000-by-1,000 matrices, the kernel using one thread to compute one element generates 1,000,000 threads when invoked.

CUDA threads are much lighter weight than CPU threads: they take very few cycles to generate and schedule thanks to efficient hardware support whereas CPU threads may require thousands of cycles.

A CUDA program consists of several phases, executed on the host: if no data parallelism, or on the device: for data parallel algorithms. The NVIDIA C compiler `nvcc` separates phases at compilation. Code for the host is compiled on host's standard C compilers and runs as ordinary CPU process. device code is written in C with keywords for data parallel functions and further compiled by `nvcc`.

A CUDA program has the following structure:

```
CPU code
kernel<<<numb_blocks,numb_threads_per_block>>>(args)
CPU code
```

The number of blocks in a grid is illustrated in Fig. 4.17.

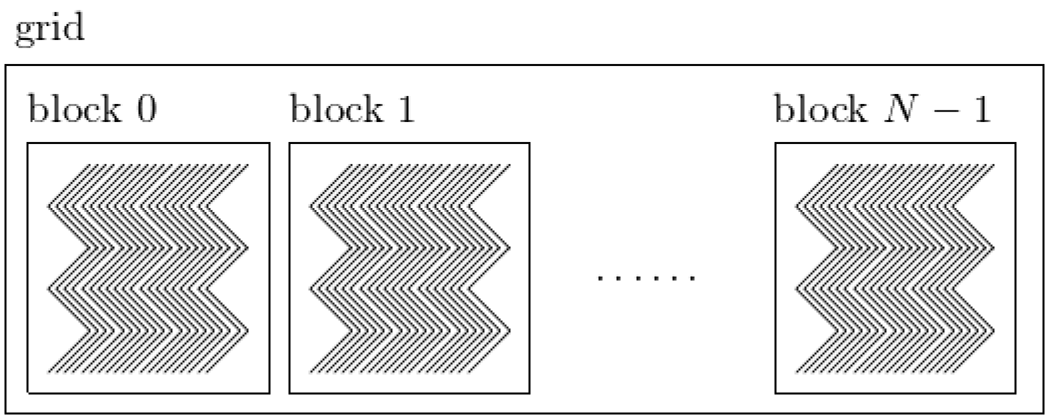


Fig. 4.17: The organization of the threads of execution in a CUDA program.

For the matrix multiplication $C = A \cdot B$, we run through the following stages:

1. Allocate device memory for A, B, and C.
2. Copy A and B from the host to the device.
3. Invoke the kernel to have device do $C = A \cdot B$.
4. Copy C from the device to the host.
5. Free memory space on the device.

A linear address system is used for the 2-dimensional array. Consider a 3-by-5 matrix stored row-wise (as in C), as shown in Fig. 4.12. Code to generate a random matrix is below:

```
#include <stdlib.h>

__host__ void randomMatrix ( int n, int m, float *x, int mode )
```

(continues on next page)

(continued from previous page)

```
/*
 * Fills up the n-by-m matrix x with random
 * values of zeroes and ones if mode == 1,
 * or random floats if mode == 0. */
{
   int i,j,r;
   float *p = x;

   for(i=0; i<n; i++)
      for(j=0; j<m; j++)
      {
         if(mode == 1)
            r = rand() % 2;
         else
            r = ((float) rand())/RAND_MAX;
         *(p++) = (float) r;
      }
}
```

The writing of a matrix is defined by the following code:

```
#include <stdio.h>

__host__ void writeMatrix ( int n, int m, float *x )
/*
 * Writes the n-by-m matrix x to screen. */
{
   int i,j;
   float *p = x;

   for(i=0; i<n; i++,printf("\n"))
      for(j=0; j<m; j++)
         printf(" %d", (int)*(p++));
}
```

In defining the kernel, we assign inner products to threads. For example, consider a 3-by-4 matrix A and a 4-by-5 matrix B, as in Fig. 4.13. The `i = blockIdx.x*blockDim.x + threadIdx.x` determines what entry in $C = A \cdot B$ will be computed:

- the row index in C is `i` divided by 5 and
- the column index in C is the remainder of `i` divided by 5.

The kernel function is defined below:

```
__global__ void matrixMultiply
 ( int n, int m, int p, float *A, float *B, float *C )
/*
 * Multiplies the n-by-m matrix A
 * with the m-by-p matrix B into the matrix C.
 * The i-th thread computes the i-th element of C. */
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   C[i] = 0.0;
```

(continues on next page)

(continued from previous page)

```
   int rowC = i/p;
   int colC = i%p;
   float *pA = &A[rowC*m];
   float *pB = &B[colC];
   for(int k=0; k<m; k++)
   {
      pB = &B[colC+k*p];
      C[i] += (*(pA++))*(*pB);
   }
}
```

Running the program in a terminal window is shown below.

```
$ /tmp/matmatmul 3 4 5 1
a random 3-by-4 0/1 matrix A :
 1 0 1 1
 1 1 1 1
 1 0 1 0
a random 4-by-5 0/1 matrix B :
 0 1 0 0 1
 0 1 1 0 0
 1 1 0 0 0
 1 1 0 1 0
the resulting 3-by-5 matrix C :
 2 3 0 1 1
 2 4 1 1 1
 1 2 0 0 1
$
```

The main program takes four command line arguments: the dimensions of the matrices, that is: the number of rows and columns of A, and the number of columns of B. The fourth element is the mode, whether output is needed or not. The parsing of the command line arguments is below:

```
int main ( int argc, char*argv[] )
{
   if(argc < 4)
   {
      printf("Call with 4 arguments :\n");
      printf("dimensions n, m, p, and the mode.\n");
   }
   else
   {
      int n = atoi(argv[1]);    /* number of rows of A */
      int m = atoi(argv[2]);    /* number of columns of A */
                                /* and number of rows of B */
      int p = atoi(argv[3]);    /* number of columns of B */
      int mode = atoi(argv[4]); /* 0 no output, 1 show output */
      if(mode == 0)
         srand(20140331)
      else
         srand(time(0));
```

The next stage in the main program is the allocation of memories, on the host and on the device, as listed below:

```
float *Ahost = (float*)calloc(n*m,sizeof(float));
float *Bhost = (float*)calloc(m*p,sizeof(float));
float *Chost = (float*)calloc(n*p,sizeof(float));
randomMatrix(n,m,Ahost,mode);
randomMatrix(m,p,Bhost,mode);
if(mode == 1)
{
   printf("a random %d-by-%d 0/1 matrix A :\n",n,m);
   writeMatrix(n,m,Ahost);
   printf("a random %d-by-%d 0/1 matrix B :\n",m,p);
   writeMatrix(m,p,Bhost);
}
/* allocate memory on the device for A, B, and C */
float *Adevice;
size_t sA = n*m*sizeof(float);
cudaMalloc((void**)&Adevice,sA);
float *Bdevice;
size_t sB = m*p*sizeof(float);
cudaMalloc((void**)&Bdevice,sB);
float *Cdevice;
size_t sC = n*p*sizeof(float);
cudaMalloc((void**)&Cdevice,sC);
```

After memory allocation, the data is copied from host to device and the kernels are launched.

```
/* copy matrices A and B from host to the device */
cudaMemcpy(Adevice,Ahost,sA,cudaMemcpyHostToDevice);
cudaMemcpy(Bdevice,Bhost,sB,cudaMemcpyHostToDevice);

/* kernel invocation launching n*p threads */
matrixMultiply<<<n*p,1>>>(n,m,p,
                          Adevice,Bdevice,Cdevice);

/* copy matrix C from device to the host */
cudaMemcpy(Chost,Cdevice,sC,cudaMemcpyDeviceToHost);
/* freeing memory on the device */
cudaFree(Adevice); cudaFree(Bdevice); cudaFree(Cdevice);
if(mode == 1)
{
   printf("the resulting %d-by-%d matrix C :\n",n,p);
   writeMatrix(n,p,Chost);
}
```

4.4.3 Two Dimensional Arrays of Threads

Using `threadIdx.x` and `threadIdx.y` instead of a one dimensional organization of the threads in a block we can make the (i, j)-th thread compute $c_{i,j}$. The main program is then changed into

```
/* kernel invocation launching n*p threads */
dim3 dimGrid(1,1);
dim3 dimBlock(n,p);
matrixMultiply<<<dimGrid,dimBlock>>>
   (n,m,p,Adevice,Bdevice,Cdevice);
```

The above construction creates a grid of one block. The block has $n \times p$ threads:

- `threadIdx.x` will range between 0 and $n - 1$, and
- `threadIdx.y` will range between 0 and $p - 1$.

The new kernel is then:

```
__global__ void matrixMultiply
 ( int n, int m, int p, float *A, float *B, float *C )
/*
 * Multiplies the n-by-m matrix A
 * with the m-by-p matrix B into the matrix C.
 * The (i,j)-th thread computes the (i,j)-th element of C. */
{
   int i = threadIdx.x;
   int j = threadIdx.y;
   int ell = i*p + j;
   C[ell] = 0.0;
   float *pB;
   for(int k=0; k<m; k++)
   {
      pB = &B[j+k*p];
      C[ell] += A[i*m+k]*(*pB);
   }
}
```

4.4.4 Examining Performance

Performance is often expressed in terms of flops.

- 1 flops = one floating-point operation per second;
- use `perf`: Performance analysis tools for Linux
- run the executable, with `perf stat`

  ```
  perf stat ./matmatmul0 1024 1024 1024 0
  ```

- with the events following the `-e` flag we count the floating-point operations.

  ```
  perf stat -e fp_arith_inst_retired.scalar_single \
  ./matmatmul0 1024 1024 1024 0
  ```

Executables are compiled with the option `-O2`.

Running on one Intel Xeon E5-2699v4 Broadwell core, the output is listed below:

```
$ perf stat -e fp_arith_inst_retired.scalar_single \
  ./matmatmul0 768 768 768 0

 Performance counter stats for './matmatmul0 768 768 768 0':

       905,969,664      fp_arith_inst_retired.scalar_single:u

       1.039681742 seconds time elapsed

       1.036818000 seconds user
       0.002999000 seconds sys
```

Did 905,969,664 operations in 1.037 seconds:

$$\Rightarrow (905, 969, 664/1.037)/(2^{30}) = 0.81\text{GFlops}.$$

Let us compared this run with the P100:

```
$ perf stat -e fp_arith_inst_retired.scalar_single
  ./matmatmul1 768 768 768 0

 Performance counter stats for './matmatmul1 768 768 768 0':

             6,123      fp_arith_inst_retired.scalar_single:u

       0.207871212 seconds time elapsed

       0.039441000 seconds user
       0.167880000 seconds sys
```

The drop from 1.037 seconds to 0.28 seconds is not impressive. The dimension 768 is too small for the GPU to be able to improve much. Let us run this on larger dimensions. First on the CPU:

```
$ perf stat -e fp_arith_inst_retired.scalar_single
  ./matmatmul0 4096 4096 4096 0

 Performance counter stats for './matmatmul0 4096 4096 4096 0':

   137,438,953,472      fp_arith_inst_retired.scalar_single:u

     416.494934403 seconds time elapsed

     416.466205000 seconds user
       0.047003000 seconds sys
```

and then on the GPU:

```
$ perf stat  ./matmatmul1 4096 4096 4096 0
```

which shows `0.569705088 seconds time elapsed`.

The speedup is 416.495/0.570 = 730, which is significant. Counting flops, f = 137,438,953,472, the performance is

- $t_{\text{cpu}} = 415.495$ and $f/t_{\text{cpu}}/(2^{30}) = 0.3$ GFlops.

- $t_{\rm gpu} = 0.570$ and $f/t_{\rm gpu}/(2^{30}) = 224.5$ GFlops.

The performance is far from optimal, both for CPU and GPU. Therefore, we will examine improved versions in the next lectures.

4.4.5 using CUDA.jl and Metal.jl

A plain matrix matrix multiplication in Julia with `CUDA.jl` is listed below:

```
using CUDA

function matmul!(C, A, B)
    i = threadIdx().x
    j = threadIdx().y
    for k=1:size(A, 2)
        @inbounds C[i, j] = C[i, j] + A[i, k]*B[k, j]
    end
end

dim = 2^2
A_h = rand(dim, dim)
B_h = rand(dim, dim)
C_h = A_h * B_h
A_d = CuArray(A_h)
B_d = CuArray(B_h)
C_d = CuArray(zeros(dim, dim))

@cuda threads=(dim, dim) matmul!(C_d, A_d, B_d)

println(C_h)
println(C_d)
```

On a macOS GPU using Apple's Metal framework, the equivalent code uses `Metal.jl`, listed below.

```
using Metal

function matmul!(C, A, B)
    threadpos = thread_position_in_grid_2d()
    i = threadpos[1]
    j = threadpos[2]
    for k=1:size(A, 2)
        @inbounds C[i, j] = C[i, j] + A[i, k]*B[k, j]
    end
end
```

Observe the `threadpos` to work with the two dimensional grid of threads. The code continues below. Because the GPU in an M1 MacBook Air does not support 64-bit floats, we use `Float32` instead of the default `Float64`:

```
dim = 2^2
A_h = rand(Float32, dim, dim)
B_h = rand(Float32, dim, dim)
C_h = A_h * B_h
A_d = MtlArray(A_h)
```

(continues on next page)

(continued from previous page)

```
B_d = MtlArray(B_h)
C_d = MtlArray(zeros(Float32, dim, dim))

@metal threads=(dim, dim) matmul!(C_d, A_d, B_d)

println(C_h)
println(C_d)
```

Observe the launching of the kernel.

4.4.6 Exercises

1. The `perf` was illustrated on on older computer. Redo the illustrations on `ampere`.
2. Modify `matmatmul0.c` and `matmatmul1.cu` to work with doubles instead of floats. Examine the performance.
3. Modify `matmatmul2.cu` to use double indexing of matrices, e.g.: `C[i][j] += A[i][k]*B[k][j]`.
4. Compare the performance of `matmatmul1.cu` and `matmatmul2.cu`, taking larger and larger values for n, m, and p. Which version scales best?
5. Compare the performance of `matmatmul1.cu` and `mmmulcuda2.jl`. Does the Julia code achieve the same performance as the C CUDA program?

4.5 Device Memories and Matrix Multiplication

The performance of our first, straightforward definition of an accelerated matrix matrix multiplication was disappointing. Tiling the matrices and paying attention to the memory hierarchies dramatically improves the performance.

4.5.1 Device Memories

Before we launch a kernel, we have to allocate memory on the device, and to transfer data from the host to the device. By default, memory on the device is *global memory*. In addition to global memory, we distinguish between

- registers for storing local variables,
- shared memory for all threads in a block,
- constant memory for all blocks on a grid.

The importance of understanding different memories is in the calculation of the expected performance level of kernel code.

the Compute to Global Memory Access (CGMA) ratio

The *Compute to Global Memory Access (CGMA) ratio* is the number of floating-point calculations performed for each access to the global memory within a region of a CUDA program.

If the CGMA ratio is 1.0, then the memory clock rate determines the upper limit for the performance. This corresponds to the roofline model discussed earlier, see Fig. 3.17 and Fig. 3.18. While memory bandwidth on a GPU is superior to that of a CPU, we will miss the theoretical peak performance by a factor of ten.

The different types of memory are schematically presented in Fig. 4.18.

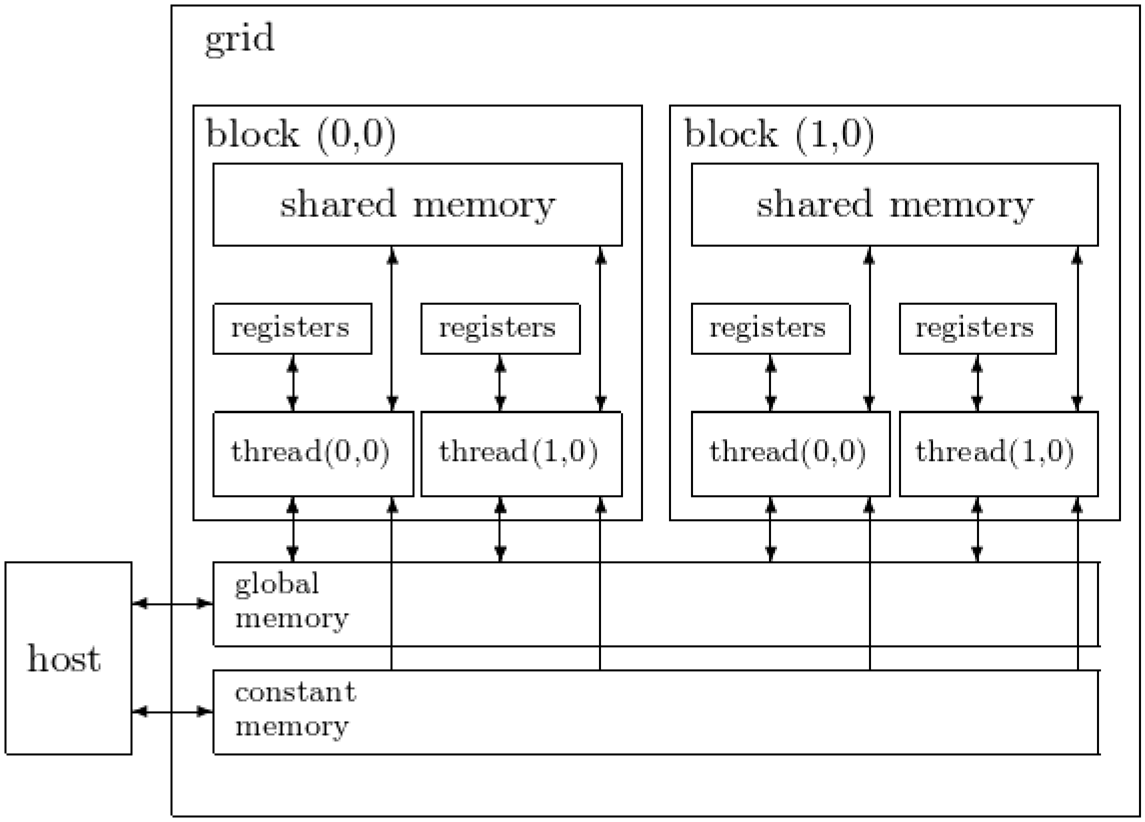


Fig. 4.18: The CUDA device memory types.

Registers are allocated to individual threads. Each thread can access only its own registers. A kernel function typically uses registers to hold frequently accessed variables that are private to each thread.

Number of 32-bit registers available per block:

- 8,192 on the GeForce 9400M,
- 32,768 on the Tesla C2050/C2070,
- 65,536 on the K20C, P100, V100, and A100.

A typical CUDA kernel may launch thousands of threads. However, having too many local variables in a kernel function may prevent all blocks from running in parallel.

Like registers, shared memory is an on-chip memory. Variables residing in registers and shared memory can be accessed at very high speed in a highly parallel manner. Unlike registers, which are private to each thread, all threads in the same block have access to shared memory.

Amount of shared memory per block:

- 16,384 byes on the GeForce 9400M,
- 49,152 bytes on the Tesla C2050/C2070,
- 49,152 bytes on the K20c, P100, V100, and A100.

The constant memory supports short-latency, high-bandwidth, read-only access by the device when all threads simultaneously access the same location.

- The GeForce 9400M has 65,536 bytes of constant memory, the total amount of global memory is 254 MBytes.
- The Tesla C2050/C2070 has 65,536 bytes of constant memory, the total amount of global memory is 2,687 MBytes, with 786,432 bytes of L2 Cache.
- The K20c has 65,536 bytes of constant memory, the total amount of global memory is 4,800 MBytes (5,032,706,048 bytes) with 1,310,720 bytes of L2 Cache.
- The constant memory remained the same for the P100, V100, and A100, but the global memory and L2 cache increased, see the summary in Table 4.2.

Table 4.2: constant, global, and cache memory, in bytes (b) and megabytes (Mb)

GPU type	constant	global	L2 cache
GeForce 9400 M	65,536 b	254 Mb	
Tesla C2050	65,536 b	2,687 Mb	786,432 b
Kepler K20C	65,536 b	4,800 Mb	1,310,720 b
Pascal P100	65,536 b	16,376 Mb	4,194,304 b
Volta V100	65,536 b	32,505 Mb	6,291,456 b
Ampere A100	65,536 b	81,038 Mb	41,943,040 b

The relationshiop between thread organization and different types of device memories is shown in Fig. 4.19, copied from the NVIDIA Whitepaper on Kepler GK110.

Each variable is stored in a particular type of memory, has a scope and a lifetime.

Scope is the range of threads that can access the variable. If the scope of a variable is a single thread, then a private version of that variable exists for every single thread. Each thread can access only its private version of the variable.

Lifetime specifies the portion of the duration of the program execution when the variable is available for use. If a variable is declared in the kernel function body, then that variable is available for use only by the code of the kernel. If the kernel is invoked several times, then the contents of that variable will not be maintained across these invocations.

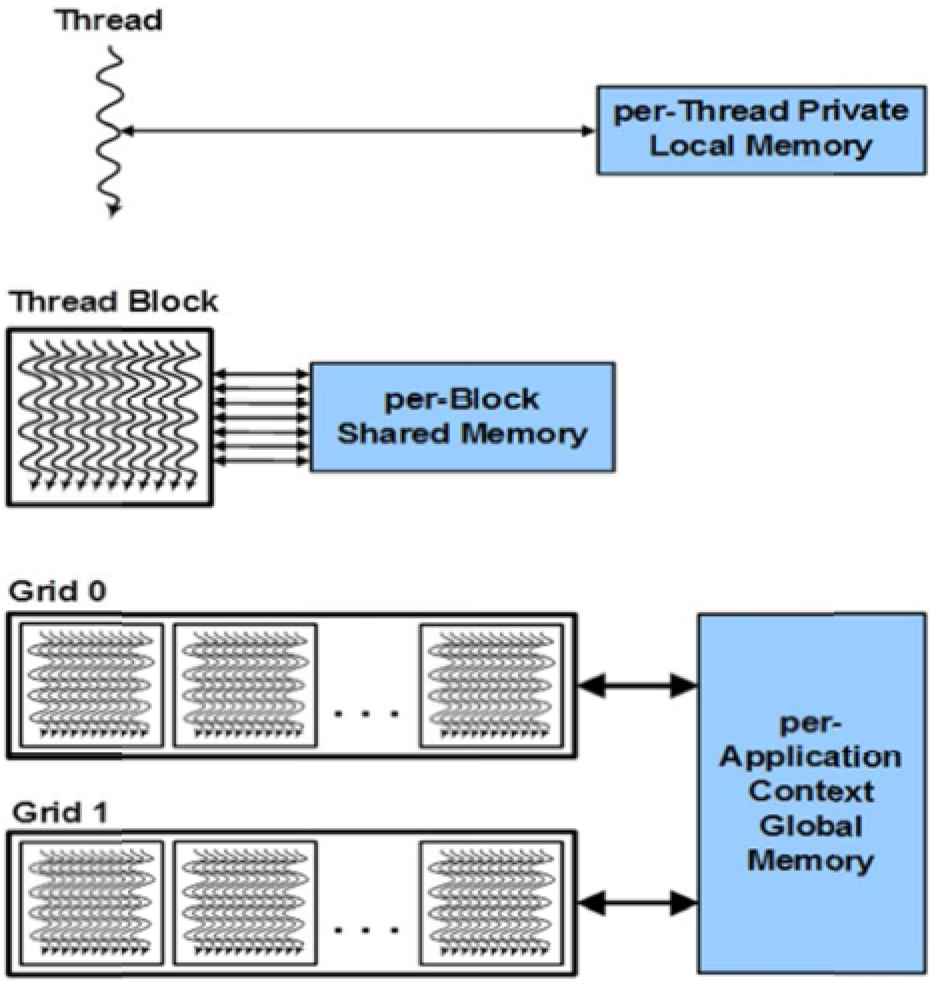

Thread
per-Thread Private Local Memory
Thread Block
per-Block Shared Memory
Grid 0
Grid 1
per-Application Context Global Memory

Fig. 4.19: Registers, shared, and global memory per thread, thread block, and grid.

We distinguish between five different variable declarations, based on their memory location, scope, and lifetime, summarized in Table 4.3.

Table 4.3: CUDA variable declarations.

variable declaration	memory	scope	lifetime
atomic variables	register	thread	kernel
array variables	local	thread	kernel
`__device__.__shared__.int v`	shared	block	kernel
`__device__.int v`	global	grid	program
`__device__.__constant__.int v`	constant	grid	program

The `__device__` in front of `__shared__` is optional.

4.5.2 Matrix Multiplication

In an application of tiling, let us examine the CGMA ratio. In our simple implementation of the matrix-matrix multiplication $C = A \cdot B$, we have the statement

```
C[i] += (*(pA++))*(*pB);
```

where

- `C` is a float array; and
- `pA` and `pB` are pointers to elements in a float array.

For the statement above, the CGMA ratio is 2/3:

- for one addition and one multiplication,
- we have three memory accesses.

To improve the CGMA ratio, we apply tiling. For $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times p}$, the product $C = A \cdot B \in \mathbb{R}^{n \times p}$. Assume that n, m, and p are multiples of some w, e.g.: $w = 8$. We compute C in tiles of size $w \times w$:

- Every block computes one tile of C.
- All threads in one block operate on submatrices:

$$C_{i,j} = \sum_{k=1}^{m/w} A_{i,k} \cdot B_{k,j}.$$

- The submatrices $A_{i,k}$ and $B_{k,j}$ are loaded from global memory into shared memory of the block.

The tiling of matrix multiplication as it relates to shared memory is shown in Fig. 4.20.

The GPU computing SDK contains as one of the examples `matrixMul` and this `matrixMul` is explained in great detail in the CUDA programming guide. We run it on the GeForce 9400M, the Tesla C2050/C2070, and the K20C, P100, V100, A100.

A session on the A100 is below:

```
$ /usr/local/cuda/samples/bin/x86_64/linux/release/matrixMul
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "Ampere" with compute capability 8.0
```

(continues on next page)

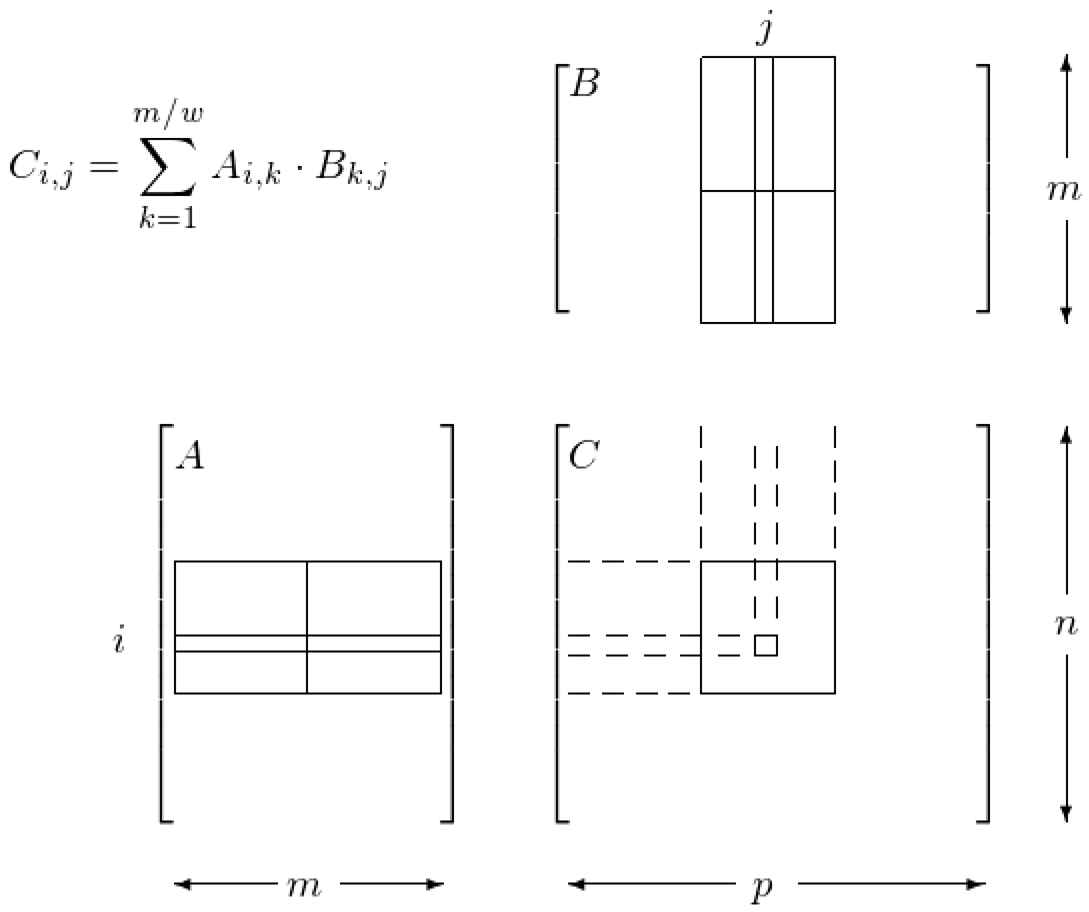


Fig. 4.20: Tiled matrix multiplication and shared memory.

(continued from previous page)

```
MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done

Performance= 4303.49 GFlop/s, Time= 0.030 msec, Size= 131072000 Ops,
WorkgroupSize= 1024 threads/block

Checking computed result for correctness: Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements.

Results may vary when GPU Boost is enabled.
```

Observe the 4.303 TFlop/s performance. The theoretical peak performance (with GPU Boost) is 78 TFlop/s (half), 19.5 TFlop/s (single), 9.7 TFlop/s (double). The performance improves with CUBLAS, the CUDA libraries for the Basic Linear Algebra Software. A session on the A100 with CUBLAS is below:

```
$ /usr/local/cuda/samples/bin/x86_64/linux/release/matrixMulCUBLAS
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Ampere" with compute capability 8.0

GPU Device 0: "NVIDIA A100 80GB PCIe" with compute capability 8.0

MatrixA(640,480), MatrixB(480,320), MatrixC(640,320)
Computing result using CUBLAS...done.
Performance= 11076.92 GFlop/s, Time= 0.018 msec, Size= 196608000 Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS

NOTE: The CUDA Samples are not meant for performance measurements.

Results may vary when GPU Boost is enabled.
```

Observe the 11.076 TFlop/s performance. For single floats, the theoretical peak precision is 17.6 TFlop/s, or 19.5 TFlop/s with GPU Boost.

The comparison of several generations of NVIDIA GPUs is summarized in table Table 4.4. The units in Table 4.4 are GFlop/s for the performance of `matrixMul` and `CUBLAS`, but then TFlop/s for the peak performance in the last column.

Table 4.4: evolution of matrixMul performance.

GPU	matrixMul	CUBLAS	peak
K20C	264	1,171	3.5
P100	1,909	3,089	9.3
V100	2,974	7,146	14.8
A100	4,303	11,076	19.5

The code of the kernel of `matrixMul` is listed next.

```
template <int BLOCK_SIZE> __global__ void
matrixMul( float* C, float* A, float* B, int wA, int wB)
{
```

(continues on next page)

(continued from previous page)

```
    int bx = blockIdx.x;   // Block index
    int by = blockIdx.y;
    int tx = threadIdx.x;  // Thread index
    int ty = threadIdx.y;
    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;
    // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE;
    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;
    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * wB;
    // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep) {

        // Declaration of the shared memory array As used to
        // store the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Declaration of the shared memory array Bs used to
        // store the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        // Load the matrices from device memory
        // to shared memory; each thread loads
        // one element of each matrix
        AS(ty, tx) = A[a + wA * ty + tx];
        BS(ty, tx) = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
#pragma unroll
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += AS(ty, k) * BS(k, tx);

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }
```

(continues on next page)

(continued from previous page)

```
    // Write the block sub-matrix to device memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

The emphasis in this lecture is on

1. the use of device memories; and
2. data organization (tiling) and transfer.

In the next lecture we will come back to this code, and cover thread scheduling

1. the use of `blockIdx`; and
2. thread synchronization.

4.5.3 using shared memory with CUDA.jl

We end the lecture with an illustration of the use of shared memory with `CUDA.jl`, using an example from the documentation. The example computes a dot product. The kernel computes

$$c_i = \sum_{k=1}^{n} a_k b_k,$$

where n equals the number of threads per block. The main program adds up the c_i for each block. In this section we highlight the syntax. The code starts as follows:

```
using CUDA
"""
    function dot(a,b,c, N, threadsPerBlock, blocksPerGrid)

computes the dot product of two vectors a and b of length N
and places the result in c, using shared memory.
"""
function dot(a,b,c, N, threadsPerBlock, blocksPerGrid)

    # Set up shared memory cache for this current block.
    cache = @cuDynamicSharedMem(Int64, threadsPerBlock)
```

Then, inside the kernel, the shared memory `cache` is used in the reduction as follows:

```
i::Int = blockDim().x/2
while i!=0
   if cacheIndex < i
       cache[cacheIndex+1] += cache[cacheIndex+i+1]
   end
   sync_threads()
   i = i/2
end
```

At the end of the kernel, the first element in the `cache` is what is computed by each thread block and that first element is stored in global memory as follows:

```
if cacheIndex == 0
    c[blockIdx().x] = cache[1]
end
```

The main program launches the kernel. We start with the dimensions:

```
N::Int64 = 33 * 1024
threadsPerBlock::Int64 = 256
blocksPerGrid::Int64 = min(32, (N + threadsPerBlock - 1) / threadsPerBlock)
```

Then comes the setup of the data (omitted). The launching of the kernels happens as

```
@cuda blocks = blocksPerGrid
      threads = threadsPerBlock
      shmem = (threadsPerBlock * sizeof(Int64))
dot(a,b,c, N, threadsPerBlock, blocksPerGrid)
```

The purpose of this short subsection is to demonstrate that a Julia program can define and use shared memory. Reduction algorithms will be covered in a later lecture.

4.5.4 Bibliography

1. Vasily Volkov and James W. Demmel:

   Benchmarking GPUs to tune dense linear algebra.

   In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, 2008. Article No. 31.

4.5.5 Exercises

1. Compile the `matrixMul` of the GPU Computing SDK on your laptop and desktop and run the program.
2. Consider the matrix multiplication code of last lecture and compute the CGMA ratio.
3. Adjust the code for matrix multiplication we discussed last time to use shared memory.

4.6 Thread Organization and Matrix Multiplication

In this lecture we look at the problem of multiplying two matrices, from the perspective of the thread organization.

4.6.1 Thread Organization

The code that runs on the GPU is defined in a function, the kernel. A kernel launch creates a grid of blocks, and each block has one or more threads. The organization of the grids and blocks can be 1D, 2D, or 3D.

During the running of the kernel:

- Threads in the same block are executed simultaneously.
- Blocks are scheduled by the streaming multiprocessors.

The P100 has 56 Streaming Multiprocessors (SMs) and threads are executed in groups of 32 (the warp size). Each SM has 64 cores. This implies: $56 \times 64 = 3584$ threads can run simultaneously. The A100 has 108 SMs, also with 64 cores each. A picture of the scalable programming model was shown in Fig. 4.7.

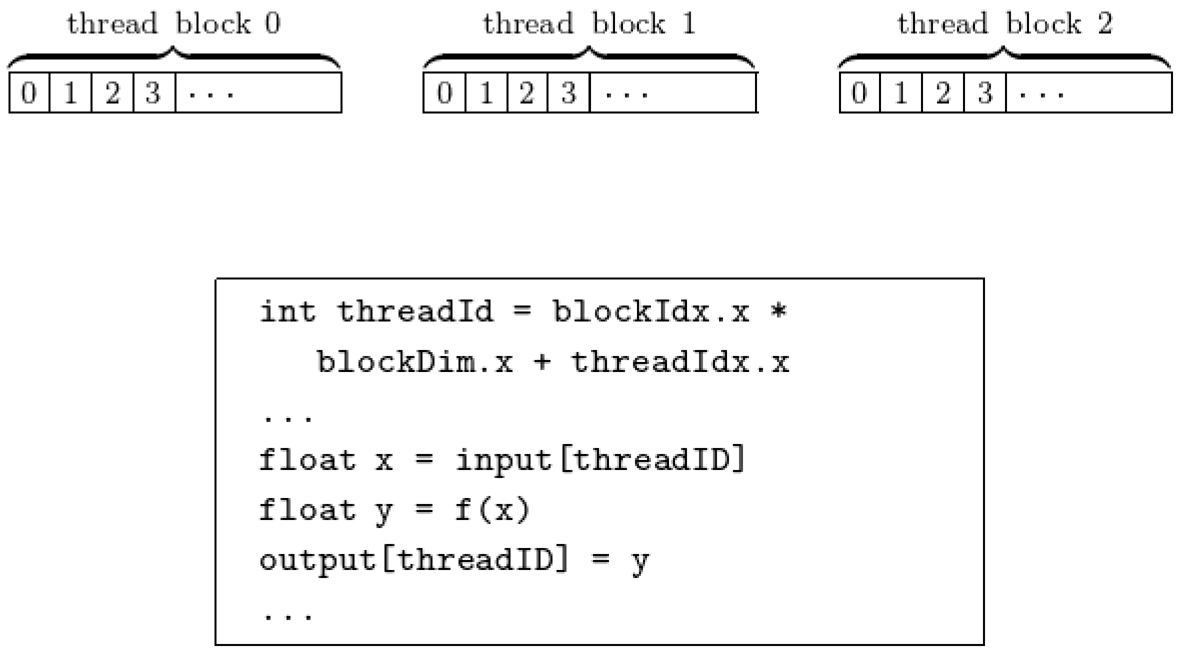
All threads execute the same code, defined by the kernel. The builtin variable `threadIdx`

- identifies every thread in a block uniquely; and
- defines the data processed by the thread.

The builtin variable `blockDim` holds the number of threads in a block. In a one dimensional organization, we use only `threadIdx.x` and `blockDim.x`. For 2D and 3D, the other components

- `threadIdx.y` belongs to the range 0 .. `blockDim.y`;
- `threadIdx.z` belongs to the range 0 .. `blockDim.z`.

The grid consists of N blocks, with $\texttt{blockIdx.x} \in \{0, N-1\}$. Within each block, $\texttt{threadIdx.x} \in \{0, \texttt{blockDim.x} - 1\}$. The organization of the data for each thread is in Fig. 4.21.



```
int threadId = blockIdx.x *
   blockDim.x + threadIdx.x
...
float x = input[threadID]
float y = f(x)
output[threadID] = y
...
```

Fig. 4.21: Data mapped to threads with block and thread indices.

Suppose the kernel is defined by the function `F` with input arguments `x` and output arguments `y`, then the execution configuration parameters are set as below:

```
dim3 dimGrid(128,1,1);
dim3 dimBlock(32,1,1);
F<<<dimGrid,dimBlock>>>(x,y);
```

which launches a grid of 128 blocks. The grid is a one dimensional array. Each block in the grid is also one dimensional and has 32 threads.

begin{frame}{multidimensional thread organization}

The limitations of the P100 and V100 are as follows:

- Maximum number of threads per block: 1,024.

- Maximum sizes of each dimension of a block: $1,024 \times 1,024 \times 64$.

  Because 1,024 is the upper limit for the number of threads in a block, the largest square 2D block is 32×32, as $32^2 = 1,024$.

- Maximum sizes of each dimension of a grid: $2,147,483,647 \times 65,535 \times 65,535$.

  2,147,483,647 is the upper limit for the builtin variable `gridDim.x`, while 65,535 is the upper limit for the builtin variables `gridDim.y` and `gridDim.z`.

The same limitations apply for the A100.

Consider the following 3D example. Suppose the function `F` defines the kernel, with argument `x`, then

```
dim3 dimGrid(3,2,4);
dim3 dimBlock(5,6,2);
F<<<dimGrid,dimBlock>>>(x);
```

launches a grid with

- $3 \times 2 \times 4$ blocks; and
- each block has $5 \times 6 \times 2$ threads.

4.6.2 Matrix Matrix Multiplication

With a three dimensional grid we can define submatrices. Consider for example a grid of dimension $2 \times 2 \times 1$ to store a 4-by-4 matrix in tiles of dimensions $2 \times 2 \times 1$, as in Fig. 4.22.

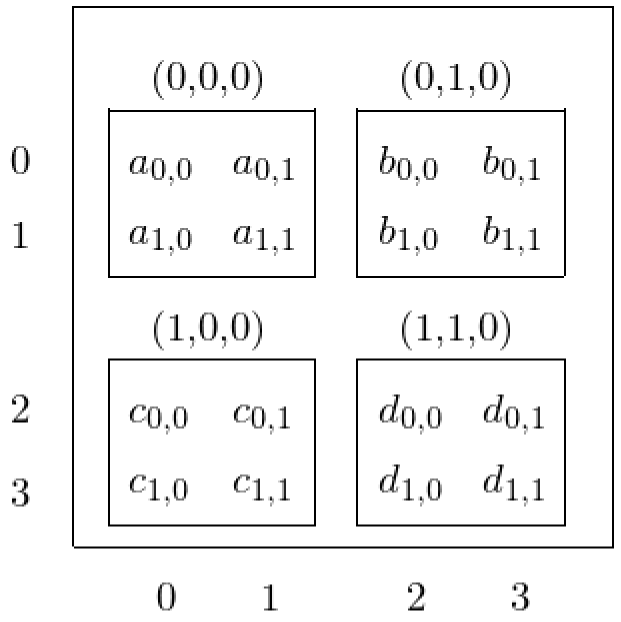

(0,0,0)
(0,1,0)
(1,0,0)
(1,1,0)
$a_{0,0}$ $a_{0,1}$ $a_{1,0}$ $a_{1,1}$
$b_{0,0}$ $b_{0,1}$ $b_{1,0}$ $b_{1,1}$
$c_{0,0}$ $c_{0,1}$ $c_{1,0}$ $c_{1,1}$
$d_{0,0}$ $d_{0,1}$ $d_{1,0}$ $d_{1,1}$
0
1
2
3
0
1
2
3

Fig. 4.22: Storing a tiled matrix in a grid.

A kernel launch with a grid of dimensions $2 \times 2 \times 1$ where each block has dimensions $2 \times 2 \times 1$ creates 16 threads. The mapping of the entries in the matrix to threads is illustrated in Fig. 4.23.

A kernel launch with a grid of dimensions $2 \times 2 \times 1$ where each block has dimensions $2 \times 2 \times 1$ creates 16 threads. The linear address calculation is illustrated in Fig. 4.24.

The main function in the CUDA code to organized the threads is listed below.

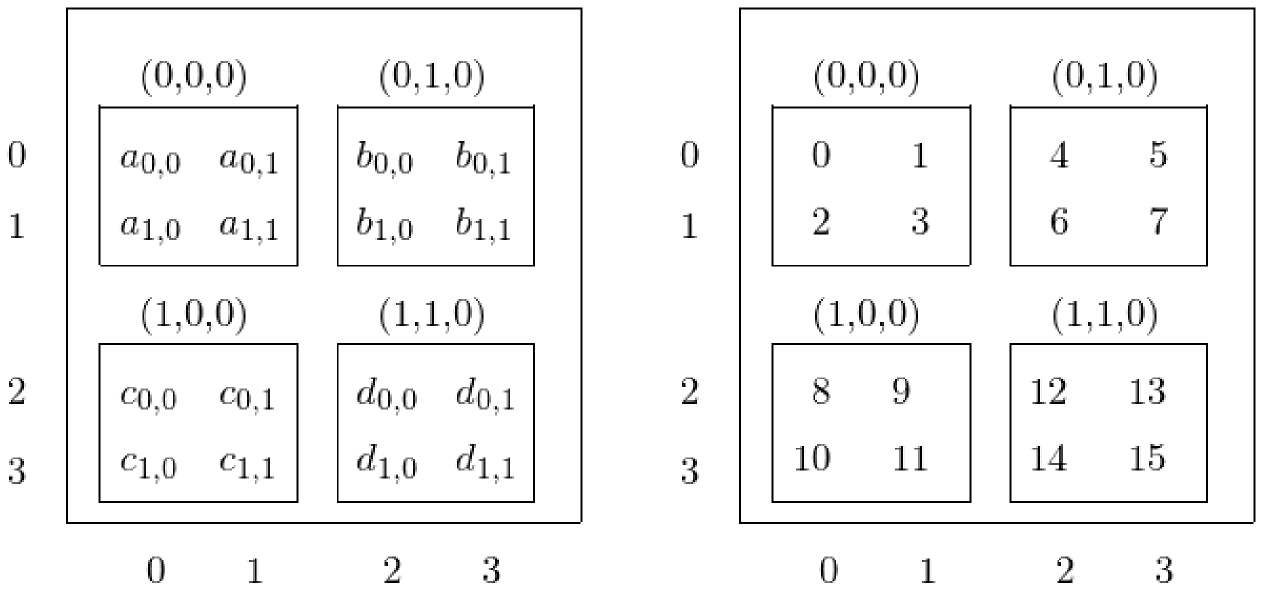
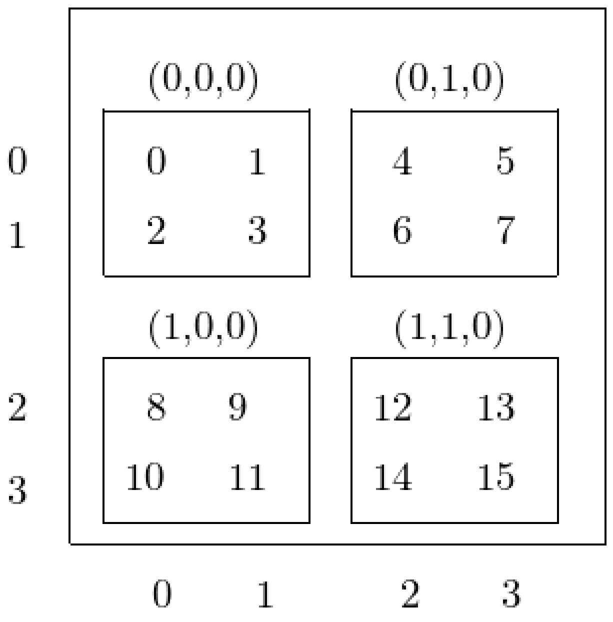


Fig. 4.23: Mapping threads to entries in the matrix.



```
x[0][0][0][0][0][0] = 0
x[0][0][0][0][1][0] = 1
x[0][0][0][1][0][0] = 2
x[0][0][0][1][1][0] = 3
x[0][1][0][0][0][0] = 4
x[0][1][0][0][1][0] = 5
x[0][1][0][1][0][0] = 6
x[0][1][0][1][1][0] = 7
x[1][0][0][0][0][0] = 8
x[1][0][0][0][1][0] = 9
x[1][0][0][1][0][0] = 10
x[1][0][0][1][1][0] = 11
x[1][1][0][0][0][0] = 12
x[1][1][0][0][1][0] = 13
x[1][1][0][1][0][0] = 14
x[1][1][0][1][1][0] = 15
```

Fig. 4.24: Linear address calculation for threads and submatrices.

```
int main ( int argc, char* argv[] )
{
   const int xb = 2; /* gridDim.x */
   const int yb = 2; /* gridDim.y */
   const int zb = 1; /* gridDim.z */
   const int xt = 2; /* blockDim.x */
   const int yt = 2; /* blockDim.y */
   const int zt = 1; /* blockDim.z */
   const int n = xb*yb*zb*xt*yt*zt;

   printf("allocating array of length %d...\n",n);

   /* allocating and initializing on the host */

   int *xhost = (int*)calloc(n,sizeof(int));
   for(int i=0; i<n; i++) xhost[i] = -1.0;

   /* copy to device and kernel launch */

   int *xdevice;
   size_t sx = n*sizeof(int);
   cudaMalloc((void**)&xdevice,sx);
   cudaMemcpy(xdevice,xhost,sx,cudaMemcpyHostToDevice);

   /* set the execution configuration for the kernel */

   dim3 dimGrid(xb,yb,zb);
   dim3 dimBlock(xt,yt,zt);
   matrixFill<<<dimGrid,dimBlock>>>(xdevice);
```

The kernel is defined in the code below.

```
__global__ void matrixFill ( int *x )
/*
 * Fills the matrix using blockIdx and threadIdx. */
{
   int bx = blockIdx.x;
   int by = blockIdx.y;
   int tx = threadIdx.x;
   int ty = threadIdx.y;
   int row = by*blockDim.y + ty;
   int col = bx*blockDim.x + tx;
   int dim = gridDim.x*blockDim.x;
   int i = row*dim + col;
   x[i] = i;
}
```

Then the main program continues with the copying to host and writing the result.

```
   /* copy data from device to host */
   cudaMemcpy(xhost,xdevice,sx,cudaMemcpyDeviceToHost);
   cudaFree(xdevice);
```

(continues on next page)

(continued from previous page)

```
   int *p = xhost;
   for(int i1=0; i1 < xb; i1++)
     for(int i2=0; i2 < yb; i2++)
       for(int i3=0; i3 < zb; i3++)
         for(int i4=0; i4 < xt; i4++)
           for(int i5=0; i5 < yt; i5++)
              for(int i6=0; i6 < zt; i6++)
                printf("x[%d][%d][%d][%d][%d][%d] = %d\n",
                        i1,i2,i3,i4,i5,i6,*(p++));
   return 0;
}
```

4.6.3 Submatrices with Threads in CUDA.jl

The equivalent computation in Julia, on an NVIDIA GPU with CUDA.jl uses the kernel below:

```
using CUDA

"""
    function matFill!(A)

fills the array using the blockIdx and threadIdx.
"""
function matFill!(A)
    bx = blockIdx().x - 1
    by = blockIdx().y - 1
    tx = threadIdx().x - 1
    ty = threadIdx().y - 1
    row = by*blockDim().y + ty
    col = bx*blockDim().x + tx
    dim = gridDim().x*blockDim().x
    idx = 1 + row*dim + col
    A[idx] = idx
    return nothing
end
```

Observe the `- 1` after the block and thread indices. Similar to arrays in Julia starting at index 1, the block and thread indices in CUDA.jl also start at 1. For `- 1` happens then for the index calculation to be the same as in the C code.

The kernel `matFill!` is launched as follows:

```
xb = 2 # gridDim.x
yb = 2 # gridDim.y
zb = 1 # gridDim.z
xt = 2 # blockDim.x
yt = 2 # blockDim.y
zt = 1 # blockDim.z

dim = xb*yb*zb*xt*yt*zt
A_h = zeros(dim)
A_d = CuArray(A_h)
```

(continues on next page)

(continued from previous page)

```
@cuda threads=(xt, yt, zt) blocks=(xb, yb, zb) matFill!(A_d)

A_h = Array(A_d)
println(A_d)
println(A_h)
```

4.6.4 Thread Synchronization

In a block all threads run independently. CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function: `__syncthreads()`. The thread executing `__syncthreads()` will be held at the calling location in the code until every thread in the block reaches the location. Placing a `__syncthreads()` ensures that all threads in a block have completed a task before moving on.

Consider the tiled matrix multiplication, as shown in Fig. 4.25.

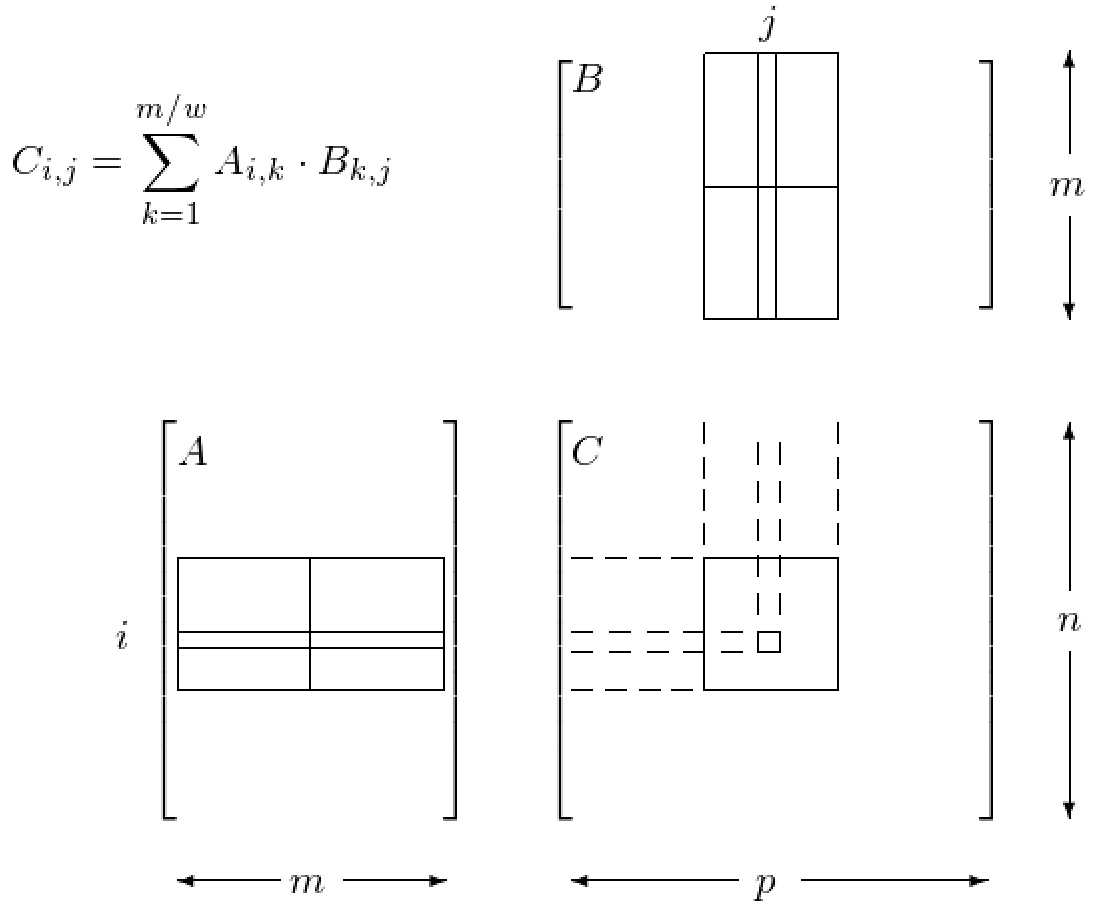


Fig. 4.25: Tiled matrix multiplication with shared memory.

With tiled matrix matrix multiplication using shared memory, all threads in the block collaborate to copy the tiles $A_{i,k}$ and $B_{k,j}$ from global memory to shared memory. Here is the need for thread synchronization. Before the calculation of the inner products, all threads must finish their copy statement: they all execute the `__syncthreads()`. Every thread computes one inner product. After this computation, another synchronization is needed. Before moving on to the next tile, all threads must finish, therefore, they all execute the `__syncthreads()` after computing their inner product and moving on to the next phase.

Let us then revisit the kernel of matrixMul and consider the code below.

```
template <int BLOCK_SIZE> __global__ void
matrixMul( float* C, float* A, float* B, int wA, int wB)
{
    int bx = blockIdx.x;   // Block index
    int by = blockIdx.y;
    int tx = threadIdx.x;  // Thread index
    int ty = threadIdx.y;
    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;
    // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE;
    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;
    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * wB;

    // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep) {

        // Declaration of the shared memory array As used to
        // store the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Declaration of the shared memory array Bs used to
        // store the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from device memory
        // to shared memory; each thread loads
        // one element of each matrix
        AS(ty, tx) = A[a + wA * ty + tx];
        BS(ty, tx) = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
#pragma unroll
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += AS(ty, k) * BS(k, tx);
```

(continues on next page)

(continued from previous page)

```
        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to device memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

4.6.5 Bibliography

1. NVIDIA CUDA Programming Guide. Available at <http://developer.nvdia.com>.
2. Vasily Volkov and James W. Demmel: **Benchmarking GPUs to tune dense linear algebra.** In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008. Article No. 31.

4.6.6 Exercises

1. Investigate the performance for the matrix-matrix multiplication with PyCUDA, comparing with the `numpy` implementation.
2. Find the limitations of the grid and block sizes for the graphics card on your laptop or desktop.
3. Extend the simple code with the three dimensional thread organization to a tiled matrix-vector multiplication for numbers generated at random as 0 or 1.

4.7 Warps and Reduction Algorithms

We discuss warp scheduling, latency hiding, SIMT, and thread divergence. To illustrate the concepts we discussed two reduction algorithms.

4.7.1 More on Thread Execution

The grid of threads are organized in a two level hierarchy:

- the grid is 1D, 2D, or 3D array of blocks; and
- each block is 1D, 2D, or 3D array of threads.

Blocks can execute in any order. Threads are bundled for execution. Each block is partitioned into *warps*.

Definition of warp

A *warp* is a unit of 32 threads, executed simultaneously by a streaming multiprocessor.

On the Tesla C2050/C2070, K20C, P100, V100, and A100, each warp consists of 32 threads.

The scheduling of threads is represented schematically in Fig. 4.26.

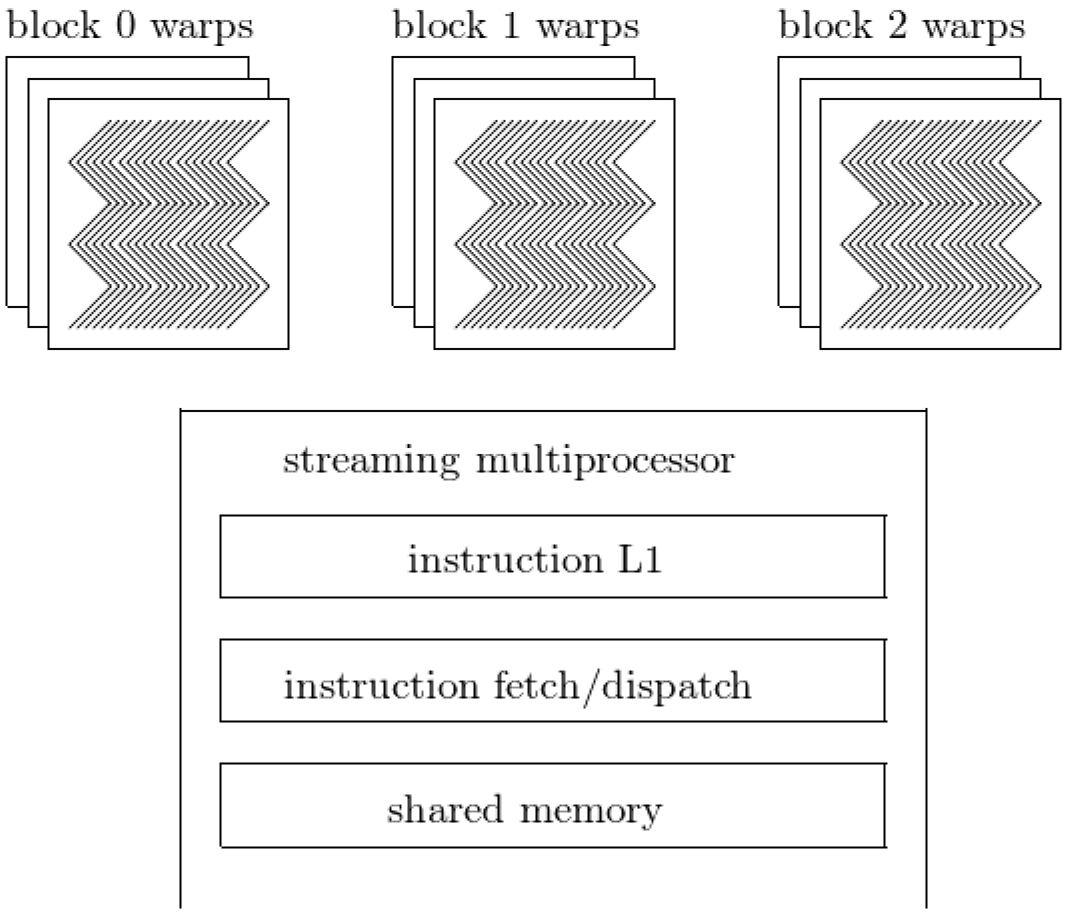


Fig. 4.26: Scheduling of threads by a streaming multiprocessor.

Let us consider the thread indices of warps. All threads in the same warp run at the same time. The partitioning of threads in a one dimensional block, for warps of size 32:

- warp 0 consists of threads 0 to 31 (value of `threadIdx`),
- warp w starts with thread $32w$ and ends with thread $32(w+1)-1$,
- the last warp is padded so it has 32 threads.

In a two dimensional block, threads in a warp are ordered along a lexicographical order of (`threadIdx.x`, `threadIdx.y`). For example, an 8-by-8 block has 2 warps (of 32 threads):

- warp 0 has threads $(0,0),(0,1),\ldots,(0,7)$, $(1,0),(1,1),\ldots,(1,7)$, $(2,0),(2,1),\ldots,(2,7)$, $(3,0),(3,1),\ldots,(3,7)$; and
- warp 1 has threads $(4,0),(4,1),\ldots,(4,7)$, $(5,0),(5,1),\ldots,(5,7)$, $(6,0),(6,1),\ldots,(6,7)$, $(7,0),(7,1),\ldots,(7,7)$.

As shown in Fig. 4.8, each streaming multiprocessor of the Fermi architecture has a dual warp scheduler.

Why give so many warps to a streaming multiprocessor if there only 32 can run at the same time? The answer is to efficiently execute long latency operations. What is this latency?

- A warp must often wait for the result of a global memory access and is therefore not scheduled for execution.

- If another warp is ready for execution, then that warp can be selected to execute the next instruction.

Definition of latency hiding

The mechanism of filling the latency of an expensive operation with work from other threads is known as *latency hiding*.

Warp scheduling is used for other types of latency operations, for example: pipelined floating point arithmetic and branch instructions. With enough warps, the hardware will find a warp to execute, in spite of long latency operations. The selection of ready warps for execution introduces no idle time and is referred to as *zero overhead thread scheduling*. The long waiting time of warp instructions is hidden by executing instructions of other warps. In contrast, CPUs tolerate latency operations with cache memories, and branch prediction mechanisms.

Let us consider how this applies to matrix-matrix multiplication For matrix-matrix multiplication, what should the dimensions of the blocks of threads be? We narrow the choices to three: 8×8, 16×16, or 32×32?

Considering that the C2050/C2070 has 14 streaming multiprocessors:

1. $32 \times 32 = 1,024$ equals the limit of threads per block.
2. $8 \times 8 = 64$ threads per block and $1,024/64 = 12$ blocks.
3. $16 \times 16 = 256$ threads per block and $1,024/256 = 4$ blocks.

Note that we must also take into account the size of shared memory when executing tiled matrix matrix multiplication.

In multicore CPUs, we use Single-Instruction, Multiple-Data (SIMD): the multiple data elements to be processed by a single instruction must be first collected and packed into a single register.

In SIMT, all threads process data in their own registers. In SIMT, the hardware executes an instruction for all threads in the same warp, before moving to the next instruction. This style of execution is motivated by hardware costs constraints. The cost of fetching and processing an instruction is amortized over a large number of threads.

Single-Instruction, Multiple-Thread works well when all threads within a warp follow the same control flow path. For example, for an *if-then-else* construct, it works well

- when either all threads execute the *then* part,
- or all execute the *else* part.

If threads within a warp take different control flow paths, then the SIMT execution style no longer works well.

Considering the *if-then-else* example, it may happen that

- some threads in a warp execute the *then* part,
- other threads in the same warp execute the *else* part.

In the SIMT execution style, multiple passes are required:

- one pass for the *then* part of the code, and
- another pass for the *else* part.

These passes are sequential to each other and thus increase the execution time.

Definition of thread divergence

If threads in the same warp follow different paths of control flow, then we say that these threads *diverge* in their execution.

Next are other examples of thread divergence. Consider an iterative algorithm with a loop some threads finish in 6 iterations, other threads need 7 iterations. In this example, two passes are required: * one pass for those threads that do the 7th iteration, * another pass for those threads that do not.

In some code, decisions are made on the `threadIdx` values:

- For example: `if(threadIdx.x > 2){ ... }`.
- The loop condition may be based on `threadIdx`.

An important class where thread divergence is likely to occur is the class of reduction algorithms.

4.7.2 Parallel Reduction Algorithms

Typical examples of reduction algorithms are the computation of the sum or the maximum of a sequence of numbers. Another example is a tournament, shown in Fig. 4.27. A reduction algorithm extracts one value from an array, e.g.: the sum of an array of elements, the maximum or minimum element in an array. A reduction algorithm visits every element in the array, using a current value for the sum or the maximum/minimum. Large enough arrays motivate parallel execution of the reduction. To reduce n elements, $n/2$ threads take $\log_2(n)$ steps.

Reduction algorithms take only 1 flop per element loaded. They are

- not *compute bound*, that is: limited by flops performance,
- but *memory bound*, that is: limited by memory bandwidth.

When judging the performance of code for reduction algorithms, we have to compare to the peak memory bandwidth and not to the theoretical peak flops count.

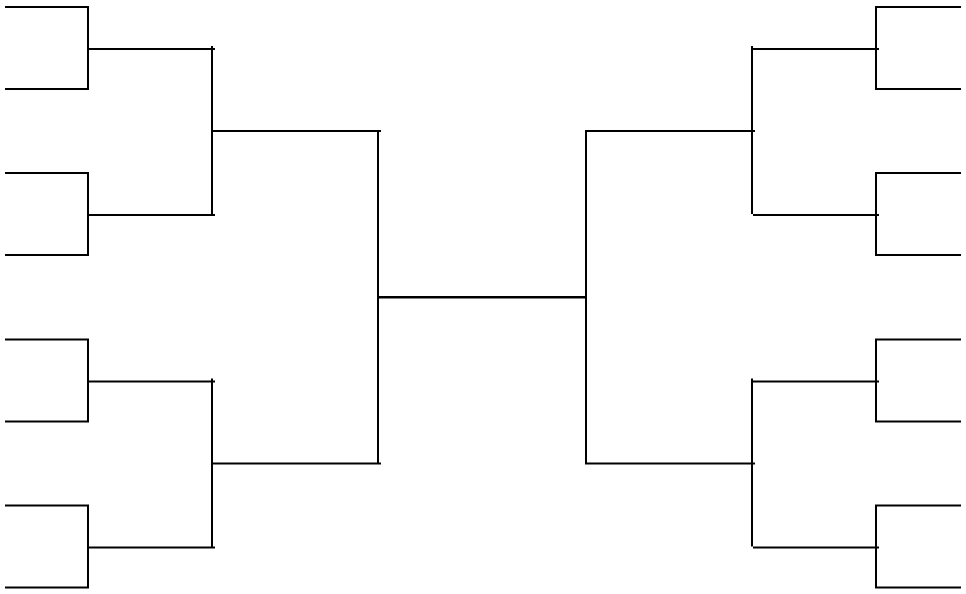


Fig. 4.27: A example of a reduction: a tournament.

As an introduction to a kernel for the parallel sum, consider the summation of 32 numbers, see Fig. 4.28.

The original array is in the global memory and copied to shared memory for a thread block to sum. A code snippet in the kernel to sum number follows.

```
__shared__ float partialSum[];
```

(continues on next page)

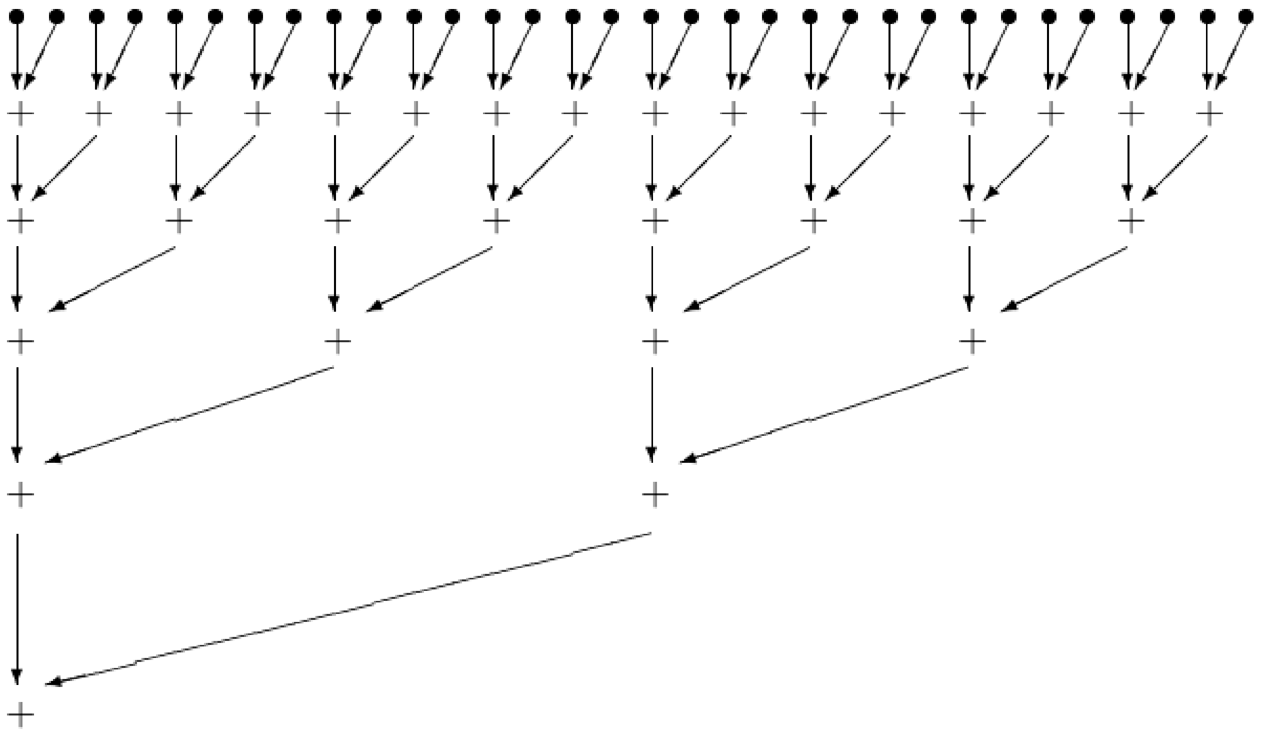


Fig. 4.28: Summing 32 numbers in a parallel reduction.

(continued from previous page)

```
int t = threadIdx.x;
for(int stride = 1; stride < blockDim.x; stride *= 2)
{
   __syncthreads();
   if(t % (2*stride) == 0)
      partialSum[t] += partialSum[t+stride];
}
```

The reduction is done *in place*, replacing elements. The `__syncthreads()` ensures that all partial sums from the previous iteration have been computed.

Because of the statement

```
if(t % (2*stride) == 0)
   partialSum[t] += partialSum[t+stride];
```

the kernel clearly has thread divergence. In each iteration, two passes are needed to execute all threads, even though fewer threads will perform an addition. Let us see if we can develop a kernel with less thread divergence.

Consider again the example of summing 32 numbers, but now with a different organization, as shown in Fig. 4.29.

The original array is in the global memory and copied to shared memory for a thread block to sum. The kernel for the revised summation is below.

```
__shared__ float partialSum[];

int t = threadIdx.x;
for(int stride = blockDim.x >> 1; stride > 0;
```

(continues on next page)

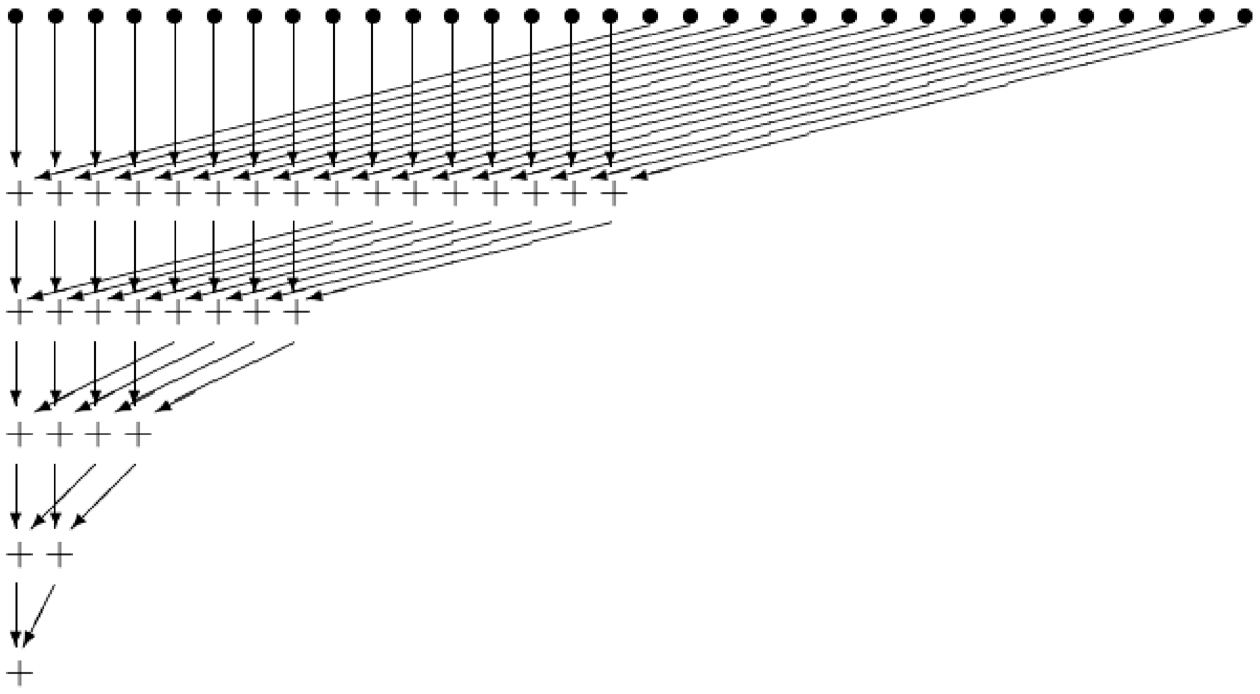


Fig. 4.29: The parallel summation of 32 number revised.

(continued from previous page)

```
    stride >> 1)
{
   __syncthreads();
   if(t < stride)
      partialSum[t] += partialSum[t+stride];
}
```

The division by 2 is done by shifting the stride value to the right by 1 bit.

Why is there less thread divergence? Af first, there seems no improvement, because of the `if`. Consider a block of 1,024 threads, partitioned in 32 warps. A warp consists of 32 threads with consecutive `threadIdx` values:

- all threads in warp 0 to 15 execute the add statement,
- all threads in warp 16 to 31 skip the add statement.

All threads in each warp take the same path ⇒ no thread divergence. If the number of threads that execute the add drops below 32, then thread divergence still occurs. Thread divergence occurs in the last 5 iterations.

4.7.3 Julia Defined Kernels

The summing of 32 numbers in five steps with Metal on a M1 Macbook Air can be coded as below.

```
using Metal

"""
    function gpusum32!(a)

```

(continues on next page)

(continued from previous page)

```
sums the 32 numbers in the array a.
On return a[1] contains the sum.
"""
function gpusum32!(a)
    i = thread_position_in_grid_1d()
    a[i] += a[i+16]
    a[i] += a[i+8]
    a[i] += a[i+4]
    a[i] += a[i+2]
    a[i] += a[i+1]
    return nothing
end

a_h = [convert(Float32, k) for k=1:32]
z_h = [0.0f0 for k=1:32] # padding with zeros
x_h = vcat(a_h, z_h)
println("the numbers to sum : ", x_h)
x_d = MtlArray(x_h)

@metal threads=32 gpusum32!(x_d)

println("the summed numbers : ", Array(x_d))
```

which prints the numbers `528.0`, `527.0`, `525.0`, `522.0`, etc.

The equivalent code to add 32 numbers in five steps with CUDA on an NVIDIA GPU is below.

```
using CUDA

"""
    function gpusum32!(a)

sums the 32 numbers in the array a.
On return a[1] contains the sum.
"""
function gpusum32!(a)
    i = threadIdx().x
    a[i] += a[i+16]
    a[i] += a[i+8]
    a[i] += a[i+4]
    a[i] += a[i+2]
    a[i] += a[i+1]
    return nothing
end

a_h = [convert(Float32, k) for k=1:32]
z_h = [0.0f0 for k=1:32] # padding with zeros
x_h = vcat(a_h, z_h)
println("the numbers to sum : ", x_h)
x_d = CuArray(x_h)

@cuda threads=32 gpusum32!(x_d)
```

(continues on next page)

(continued from previous page)

```
println("the summed numbers : ", Array(x_d))
```

which prints the same numbers as in the other program.

To illustrate shared memory with `CUDA.jl` consider the summation of the first N natural numbers. The output of running `firstsum.jl` on an NVIDIA GPU is as follows:

```
$ julia firstsum.jl
size of the vector : 33792
  number of blocks : 32
 threads per block : 256
 number of threads : 8192
Does GPU value 570966528 = 570966528 ? true
$
```

In the check, we use $\displaystyle\sum_{k=1}^{N} k = \frac{N(N+1)}{2}$. The definition of the kernel is below.

```
using CUDA

"""
    function sum(x, y, N, threadsPerBlock, blocksPerGrid)

computes the sum product of N numbers in x and
places the results in y, using shared memory.
"""
function sum(x, y, N, threadsPerBlock, blocksPerGrid)
    # set up shared memory cache for this current block
    cache = @cuDynamicSharedMem(Int64, threadsPerBlock)
    # initialise the indices
    tid = (threadIdx().x - 1) + (blockIdx().x - 1) * blockDim().x
    totalThreads = blockDim().x * gridDim().x
    cacheIndex = threadIdx().x - 1
    # run over the vector
    temp = 0
    while tid < N
        temp += x[tid + 1]
        tid += totalThreads
    end
    # set cache values
    cache[cacheIndex + 1] = temp
    # synchronise threads
    sync_threads()
    # we add up all of the values stored in the cache
    i::Int = blockDim().x ÷ 2
    while i!=0
        if cacheIndex < i
            cache[cacheIndex + 1] += cache[cacheIndex + i + 1]
        end
        sync_threads()
        i = i ÷ 2
    end
    # cache[1] now contains the sum of the numbers in the block
```

(continues on next page)

(continued from previous page)

```
    if cacheIndex == 0
        y[blockIdx().x] = cache[1]
    end
    return nothing
end
```

The launching of the kernels happens in the main program below.

```
"""
    Tests the kernel on the first N natural numbers.
"""
function main()
    N::Int64 = 33 * 1024
    threadsPerBlock::Int64 = 256
    blocksPerGrid::Int64 = min(32, (N + threadsPerBlock - 1) / threadsPerBlock)
    println("size of the vector : ", N)
    println("  number of blocks : ", blocksPerGrid)
    println(" threads per block : ", threadsPerBlock)
    println(" number of threads : ", blocksPerGrid*threadsPerBlock)
    # input arrays on the host
    x_h = [i for i=1:N]
    # make the arrays on the device
    x_d = CuArray(x_h)
    y_d = CuArray(fill(0, blocksPerGrid))
    # execute the kernel. Note the shmem argument - this is necessary to allocate
    # space for the cache we allocate on the gpu with @cuDynamicSharedMem
    @cuda blocks = blocksPerGrid threads = threadsPerBlock shmem =
    (threadsPerBlock * sizeof(Int64)) sum(x_d, y_d, N, threadsPerBlock, blocksPerGrid)
    # copy the result from device to the host
    y_h = Array(y_d)
    local result = 0
    for i in 1:blocksPerGrid
        result += y_h[i]
    end
    # check whether output is correct
    print("Does GPU value ", result, " = ", N*(N+1) ÷ 2, " ? ")
    println(result == N*(N+1) ÷ 2)
end
```

4.7.4 Bibliography

- S. Sengupta, M. Harris, and M. Garland. **Efficient parallel scan algorithms for GPUs.** Technical Report NVR-2008-003, NVIDIA, 2008.
- M. Harris. **Optimizing parallel reduction in CUDA.** White paper available at <http://docs.nvidia.com>.

4.7.5 Exercises

1. Consider the code `matrixMul` of the GPU computing SDK. Look up the dimensions of the grid and blocks of threads. Can you (experimentally) justify the choices made?
2. Write code for the two summation algorithms we discussed. Do experiments to see which algorithm performs better.
3. Apply the summation algorithm to the composite trapezoidal rule. Use it to estimate π via $\displaystyle\frac{\pi}{4} = \int_0^1 \sqrt{1 - x^2} dx$.

CHAPTER 5

Review for the Midterm Exam

At the middle of the course, there is a midterm exam. We consider some representative questions.

5.1 Four Sample Questions

The four questions in this lecture are representative for some of the topics covered in the course.

5.1.1 Scaled Speedup

Benchmarking of a program running on a 12-processor machine shows that 5% of the operations are done sequentially, i.e.: that 5% of the time only one single processor is working while the rest is idle.

Compute the scaled speedup.

The formula for scaled speedup is $S_s(p) \leq \dfrac{st + p(1-s)t}{t} = s + p(1-s) = p + (1-p)s$. Evaluating this formula for $s = 0.05, p = 12$ yields

$$S_s(12) = 12 + (1-12)0.05 = 11.45.$$

5.1.2 Network Topologies

Show that a hypercube network topology has enough connections for a fan-in gathering of results.

Consider Fig. 5.1, which illustrates the fan-in algorithm for 8 nodes.

For the example in Fig. 5.1, three steps are executed:

1. 001 $\rightarrow$ 000; 011 $\rightarrow$ 010; 101 $\rightarrow$ 100; 111 $\rightarrow$ 110
2. 010 $\rightarrow$ 000; 110 $\rightarrow$ 100
3. 100 $\rightarrow$ 000

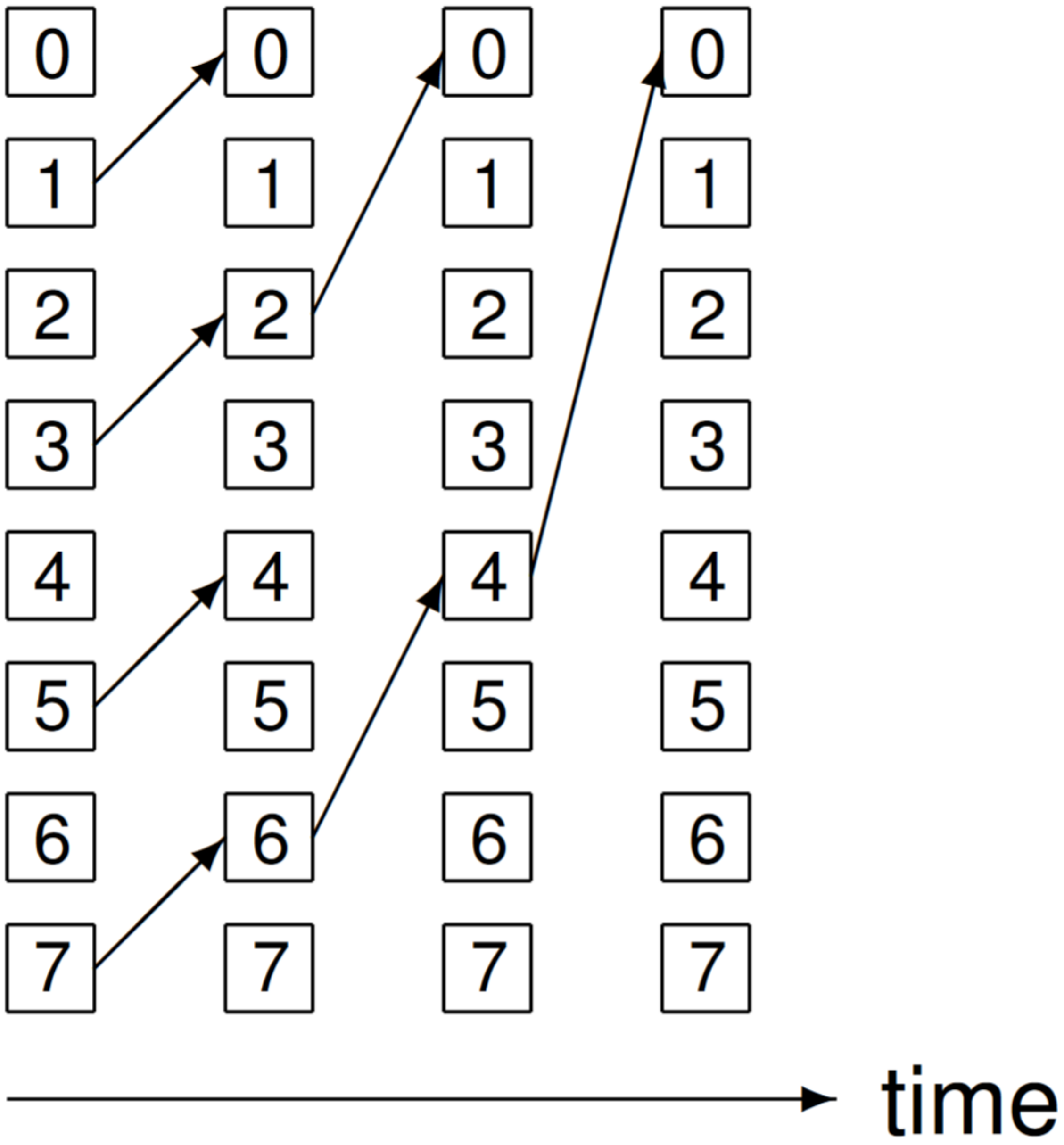


Fig. 5.1: Fanning in the result for 8 nodes.

To show a hypercube network has sufficiently many connections for the fan-in algorithm, we proceed via a proof by induction.

- The base case: we verified for 1, 2, 4, and 8 nodes.
- Assume we have enough connections for a 2^k hypercube.

  We need to show that we have enough connections for a 2^{k+1} hypercube:

  1. In the first k steps:
     - node 0 gathers from nodes $1, 2, \ldots 2^k - 1$;
     - node 2^k gathers from nodes $2^k + 1, 2^k + 2, \ldots, 2^{k+1} - 1$.
  2. In step $k + 1$: node 2^k can send to node 0, because only one bit in 2^k is different from 0.

5.1.3 Task Graph Scheduling

Given are two vectors $\mathbf{x}$ and $\mathbf{y}$, both of length n, with $x_i \neq x_j$ for all $i \neq j$. Consider the code below:

```
for i from 2 to n do
    for j from i to n do
        y[j] = (y[i-1] - y[j])/(x[i-1] - x[j])
```

1. Define the task graph for a parallel computation of $\mathbf{y}$.
2. Do a critical path analysis on the graph to determine the upper limit of the speedup.

For $n = 4$, the numbers are in the table below:

$$\begin{array}{lll} y_2 = \dfrac{y_1 - y_2}{x_1 - x_2} & & \\ y_3 = \dfrac{y_1 - y_3}{x_1 - x_3} & y_3 = \dfrac{y_2 - y_3}{x_2 - x_3} & \\ y_4 = \dfrac{y_1 - y_4}{x_1 - x_4} & y_4 = \dfrac{y_2 - y_4}{x_2 - x_4} & y_4 = \dfrac{y_3 - y_4}{x_3 - x_4} \end{array}$$

If the computations happen row by row, then there is no parallelism. Observe that the elements in each column can be computed independently from each other. Label the computation on row i and column j by $T_{i,j}$ and consider the graph in Fig. 5.2.

For $n = 4$, with 3 processors, it takes three steps to compute the table. The speedup is 6/3 = 2. Each path leading to $T_{4,4}$ has two edges or three nodes. So, the length of the critical path is 2.

For any n, with $n - 1$ processors, it takes $n - 1$ steps, leading to a speedup of $n(n - 1)/2 \times 1/(n - 1) = n/2$.

5.1.4 Compute Bound or Memory Bound

A kernel performs 36 floating-point operations and seven 32-bit global memory accesses per thread.

Consider two GPUs A and B, with the following properties:

- A has peak FLOPS of 200 GFLOPS and 100 GB/second as peak memory bandwidth;
- B has peak FLOPS of 300 GFLOPS and 250 GB/second as peak memory bandwidth.

For each GPU, is the kernel compute bound or memory bound?

The CGMA ratio of the kernel is $\dfrac{36}{7 \times 4} = \dfrac{36}{28} = \dfrac{9}{7} \dfrac{\text{operations}}{\text{byte}}$.

$$\begin{tikzcd}
T_{2,2} \arrow[dr] \arrow[ddr] & & \\
T_{3,2} \arrow[r] & T_{3,3} \arrow[dr] & \\
T_{4,2} \arrow[r] & T_{4,3} \arrow[r] & T_{4,4}
\end{tikzcd}$$

Fig. 5.2: The task graph for $n = 4$.

Taking the ratio of the peak performance and peak memory bandwidth of GPU A gives $200/100 = 2$ operations per byte. As $9/7 < 2$, the kernel is memory bound on GPU A.

Alternatively, it takes GPU A per thread $\dfrac{36}{200 \times 2^{30}}$ seconds for the operations and $\dfrac{28}{100 \times 2^{30}}$ seconds for the memory transfers. As $0.18 < 0.28$, more time is spent on transfers than on operations.

For GPU B, the ratio is $300/250 = 6/5$ operations per byte. As $9/7 > 6/5$, the kernel is compute bound on GPU B.

Alternatively, it takes GPU B per thread $\dfrac{36}{300 \times 2^{30}}$ seconds for the operations and $\dfrac{28}{250 \times 2^{30}}$ seconds for the memory transfers. As $0.12 > 0.112$, more time is spent on computations than on transfers.

5.2 Fall 2024 Midterm Questions

The four questions below appeared on the midterm exam of Fall 2024.

5.2.1 Question 1 : Isoefficiency

Consider the communication and computation costs (as functions of dimension n and number of processors p) in the running times $t^{(A)}(n,p)$ and $t^{(B)}(n,p)$ of two programs, respectively A and B:

$$t^{(A)}(n,p) = t^{(A)}_{\rm comm}(n,p) + t^{(A)}_{\rm comp}(n,p), \quad t^{(A)}_{\rm comm}(n,p) = n\log(p), \quad t^{(A)}_{\rm comp}(n,p) = n^2/p,$$

$$t^{(B)}(n,p) = t^{(B)}_{\rm comm}(n,p) + t^{(B)}_{\rm comp}(n,p), \quad t^{(B)}_{\rm comm}(n,p) = p + n^2 p, \quad t^{(B)}_{\rm comp}(n,p) = n^3/p.$$

Which program scales best? Use isoefficiency to justify your answer.

Illustrate with values for p between from 2 and 256, and for n between 10,000 and 100,000.

Evaluating the efficiencies for both programs gives Table 5.1 and Table 5.2, indexed by the number of processors in 2, 4, 8, 16, 32, 64, 128, 256, in the rows, and for the columns by the dimensions in 10000, 20000, 50000, 100000.

Table 5.1: efficiency of program A

	10000	20000	50000	100000
2	99.98	99.99	100.00	100.00
4	99.92	99.96	99.98	99.99
8	99.76	99.88	99.95	99.98
16	99.36	99.68	99.87	99.94
32	98.43	99.21	99.68	99.84
64	96.30	98.12	99.24	99.62
128	91.78	95.71	98.24	99.11
256	83.00	90.71	96.07	97.99

Table 5.2: efficiency of program B

	10000	20000	50000	100000
2	99.97	99.99	99.99	100.00
4	99.85	99.93	99.97	99.99
8	99.37	99.69	99.87	99.94
16	97.51	98.74	99.49	99.75
32	90.72	95.13	98.00	98.99
64	70.95	83.01	92.43	96.07
128	37.91	54.97	75.32	85.92
256	13.24	23.38	43.28	60.41

As the efficiency of program A is better than program B for increasing number of processors and dimensions, program A scales best.

5.2.2 Question 2 : the roofline model

A processor has 800 GFLOPS as peak performance and 400 GB/s as peak memory bandwidth.

The arithmetic intensity of a computation is 3 flops per byte.

1. Given the processor specifications, draw the roofline model.

   Apply your model to justify whether the computation is compute bound or memory bound.

2. If the arithmetic intensity of the computation is 1 flop per byte, then what is the maximal performance the computation can attain?

The roofline model is drawn in Fig. 5.3.

On Fig. 5.3 we see two vertical lines. The first vertical line at 1 flop per byte contains the points of the possible performance, capped at 400 Gigaflops. At 1 flop per byte, the computation is memory bound. At 3 flops per byte, the maximum performance is 800 Gigaflops and the computation is compute bound.

5.2.3 Question 3 : tasking for enumeration

Consider three sets $A = \{a_1, a_2, \ldots, a_n\}$, $B = \{b_1, b_2, \ldots, b_n\}$, $C = \{c_1, c_2, \ldots, c_n\}$ of vertices in a tripartite graph G with edges in the set E.

Each edge $e \in E$ is a pair of the form (a_i, b_j) or (b_i, c_j), for i and j in $\{1, 2, \ldots, n\}$.

A matching is a subset of E where every element of A, B, and C occurs exactly once.

Consider the straightforward enumeration of all triplets to compute all matchings in G.

1. Explain how tasking can be applied to speed up the enumeration.
2. What speedup do you expect? Relate the speedup to n and the size of E.

Below is the outline of a possible solution.

1. The matching consists of a set of triplets (a, b, c), where each a, b, c occurs exactly once in each triplet, (a, b) and (b, c) are edges.

   In exploring the search space, we can enumerate the triplets as follows, for each edge $(u, v) \in E$, consider the matching once with, and once without (u, v), spawning two tasks for each case, with $(u, v) \in A \times B$ or $(u, v) \in B \times C$.

   Each task maintains its path of selected edges and set of triplets.

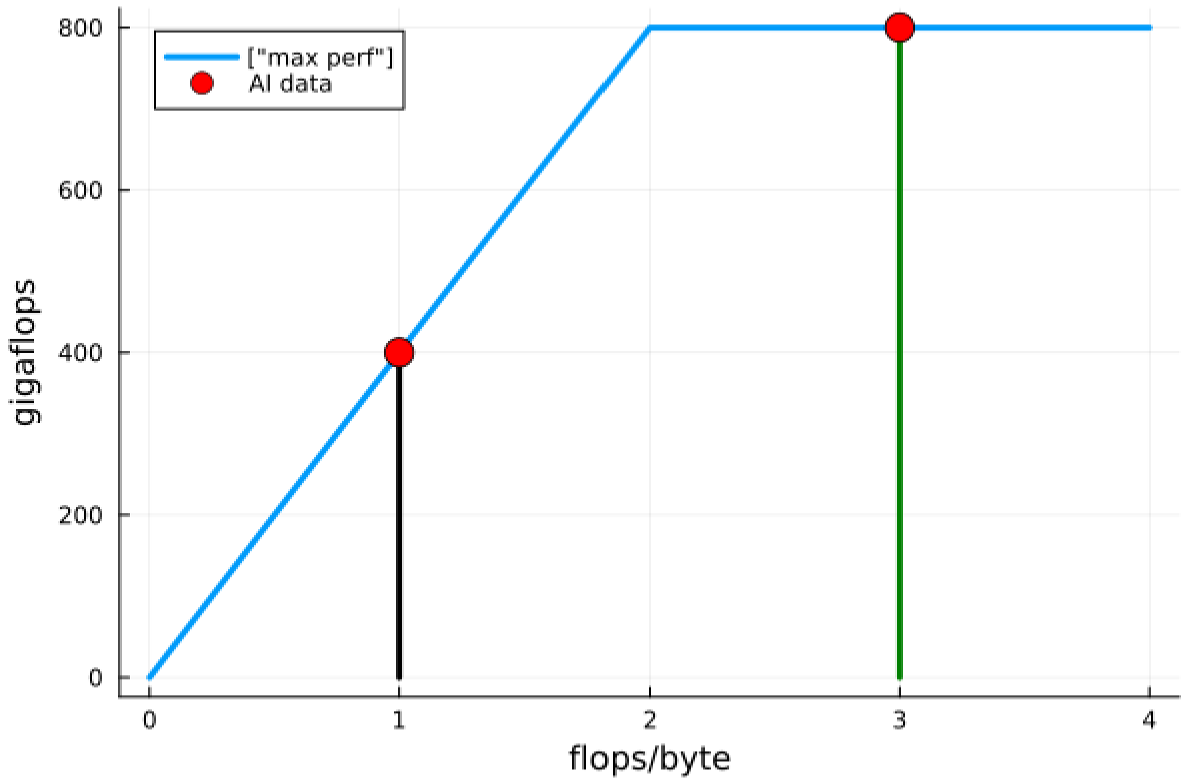


Fig. 5.3: Roofline model for 800 GFLOP/s and 400 GB/s.

2. The number of possible edges grows as n choose 2 squared, as an upper bound between the edges between vertices of sets A and B, and combined with all edges of vertices of sets B and C. The search space grows much faster than we can scale the number of threads and the search space can be explored independently by many threads.

   With p threads, we expect the search space to be explored p times faster, and we may hope for an optimal speedup.

   Superlinear speedup could occur if one task finds a matching when less than $1/p$-th of the search space has been explored, where p is the number of threads, but then if we want only one matching, and the question asked for all matchings.

5.2.4 Question 4 : CGMA ratio

Consider the kernel below.

```
__global__ SumOfSquares ( int n, float *x, float *y )
{
    int bdx = blockIdx.x;
    int tdx = threadIdx.x;
    int offset = bdx*n;

    y[idx] = 0.0;
    for(int i=0; i<n; i++)
       y[idx] = y[idx] + x[offset+i]*x[offset+i];
}
```

1. Compute the CGMA ratio for this kernel.
2. Explain how the use of registers and shared memory improves the CGMA ratio.

Below is an outline of a possible solution.

1. The CGMA ratio is $\dfrac{2n}{4n+1}$.
2. Using two registers `xi` and `yr`, as below

   ```
   __global__ SumOfSquares ( int n, float *x, float *y )
   {
       int bdx = blockIdx.x;
       int tdx = threadIdx.x;
       int offset = bdx*n;
       float xi;
       float yr = 0.0;

       for(int i=0; i<n; i++)
       {
          xi = x[offset+i];
          yr = yr + xi*xi;
       }
       y[idx] = yr;
   }
   ```

   improves the CGMA ratio to $\dfrac{2n}{n+1}$.

CHAPTER 6

Pipelining and Synchronized Computations

6.1 Pipelined Computations

Although a process may consist in stages that have to be executed in order and thus there may not be much speedup possible for the processing of one item, arranging the stages in a pipeline speeds up the processing of many items.

6.1.1 Functional Decomposition

Car manufacturing is a successful application of pipelines. Consider a simplified car manufacturing process in three stages: (1) assemble exterior, (2) fix interior, and (3) paint and finish, as shown schematically Fig. 6.1.

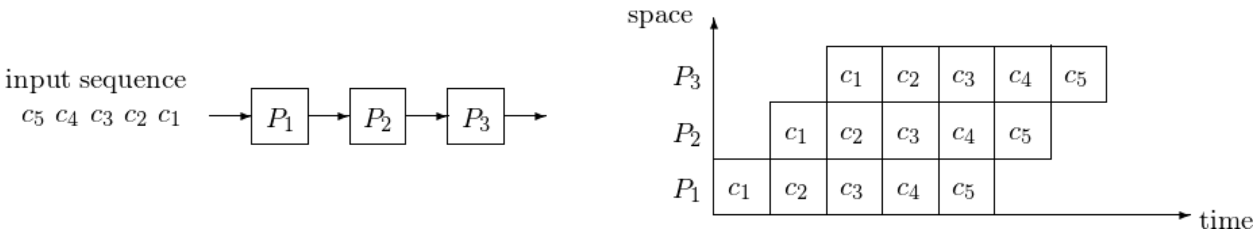


Fig. 6.1: A schematic of a 3-stage pipeline at the left, with the corresponding space-time diagram at the right. After 3 time units, one car per time unit is completed. It takes 7 time units to complete 5 cars.

Definition of a Pipeline

A pipeline with p processors is *a p-stage pipeline*. A time unit is called a *pipeline cycle*. The time taken by the first *p-1* cycles is the *pipeline latency*.

Suppose every process takes one time unit to complete. How long does it take till a p-stage pipeline completes n inputs? A p-stage pipeline on n inputs. After p time units the first input is done. Then, for the remaining $n-1$ items, the pipeline completes at a rate of one item per time unit. So, it takes $p + n - 1$ time units for the p-stage pipeline to complete n inputs. The speedup `S(p)` for n inputs in a p-stage pipeline is thus

$$S(p) = \frac{n \times p}{p + n - 1}.$$

For a fixed number p of processors:

$$\lim_{n \rightarrow \infty} \frac{p \times n}{n + p - 1} = p.$$

Pipelining is a *functional decomposition* method to develop parallel programs. Recall the classification of Flynn: MISD = Multiple Instruction Single Data stream.

Another successful application of pipelining is floating-point addition. The parts of a floating-point number are shown in Fig. 6.2.

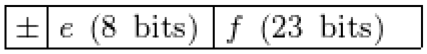


Fig. 6.2: A floating-point number has a sign bit, exponent, and fraction.

Adding to floats could be done in 6 cycles:

1. unpack fractions and exponents;
2. compare the exponents;
3. align the fractions;
4. add the fractions;
5. normalize the result; and
6. pack the fraction and the exponent of the result.

Adding two vectors of n floats with 6-stage pipeline takes $n + 6 - 1$ pipeline cycles, instead of $6n$ cycles.

The functionality of the pipelines in Intel Core processors is shown in Fig. 6.3.

Our third example of a successful application of pipelining is the denoising a signal. Every second we take 256 samples of a signal:

- P_1: apply FFT,
- P_2: remove low amplitudes, and
- P_3: inverse FFT,

as shown in Fig. 6.4. Observe: the consumption of a signal is sequential.

6.1.2 Loop Unrolling

The example below is taken from section 3.2.2 on loop unrolling in *Scientific Programming and Computer Architecture* by Divakar Viswanath, Springer-Verlag, 2017.

The Leibniz series

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots$$

converges very slowly.

The branching in the straightforward code in Julia prevents a pipelined execution of the floating-point operations.

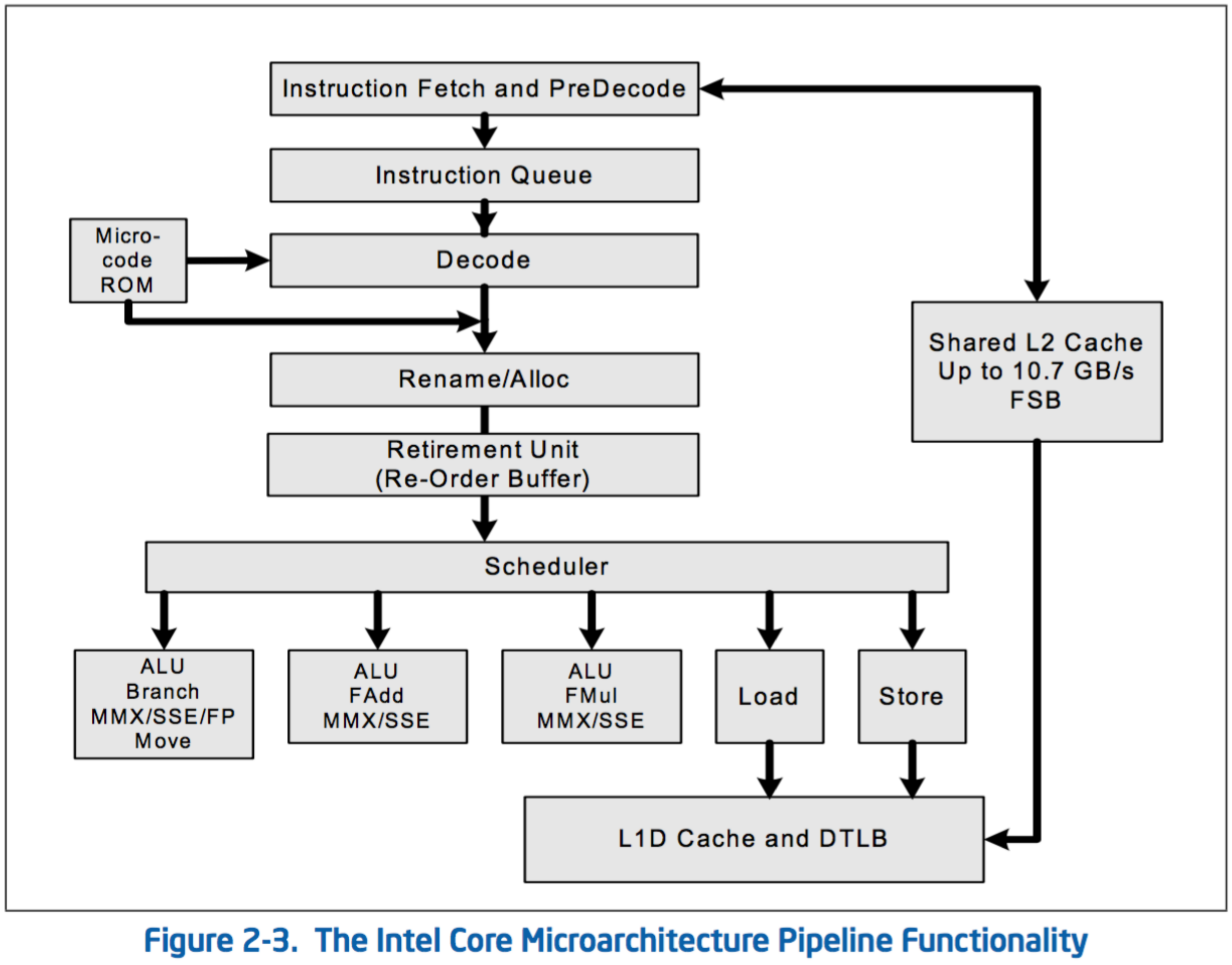

Instruction Fetch and PreDecode
Instruction Queue
Micro-code ROM
Decode
Rename/Alloc
Retirement Unit (Re-Order Buffer)
Scheduler
ALU Branch MMX/SSE/FP Move
ALU FAdd MMX/SSE
ALU FMul MMX/SSE
Load
Store
L1D Cache and DTLB
Shared L2 Cache Up to 10.7 GB/s FSB
Figure 2-3. The Intel Core Microarchitecture Pipeline Functionality

Fig. 6.3: Copied from the Intel Architecture Software Developer's Manual.

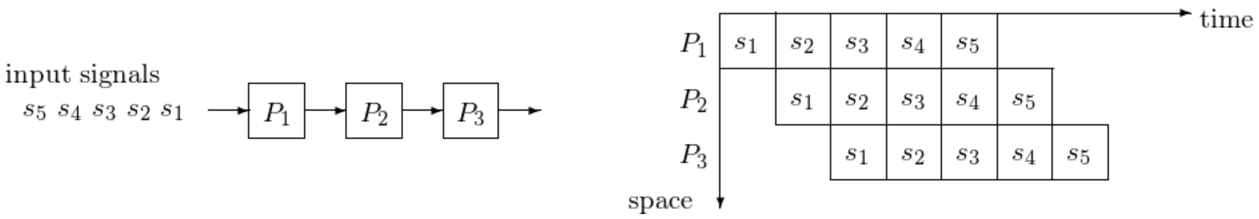


Fig. 6.4: Pipeline to denoising a signal at the left, with space-diagram at the right.

```
function leibniz1(N::Int)
    s = 1.0
    for i=1:N
        if(i%2 == 1)
            s = s - 1.0/(2.0*i + 1.0)
        else
            s = s + 1.0/(2.0*i + 1.0)
        end
    end
    return s
end
```

Applying loop unrolling, summing the even and odd terms separately avoids branching, allows a pipelined executions of the floating-point operations.

```
function leibniz2(N::Int)
    s = 1.0
    for i=2:2:N
        s = s + 1.0/(2.0*i + 1.0)
    end
    for i=1:2:N
        s = s - 1.0/(2.0*i + 1.0)
    end
    return s
end
```

The second function `leibniz2` rewrites `leibniz1` as

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots = 1 + \frac{1}{5} + \frac{1}{9} + \cdots - \frac{1}{3} - \frac{1}{7} - \frac{1}{11} - \cdots$$

The code below benchmarks the two different functions.

```
using BenchmarkTools

println(4.0*leibniz1(10^8))
@btime leibniz1(10^8)

println(4.0*leibniz2(10^8))
@btime leibniz2(10^8)
```

which gives the output:

```
3.141592663589326
239.600 ms (0 allocations: 0 bytes)
3.1415926635801443
125.266 ms (0 allocations: 0 bytes)
```

Executed with Julia 1.8.5 on two 22-core Intel Xeon E5-2699v4 Broadwell at 2.20GHz, 256GB of internal memory at 2400MHz.

On more recent versions of Julia, and in particular 1.11, there is no longer a difference in performance between the two functions.

6.1.3 Pipeline Implementations

A ring topology of processors is a natural way to implement a pipeline. In Fig. 6.5, the stages in a pipeline are performed by the processes organized in a ring.

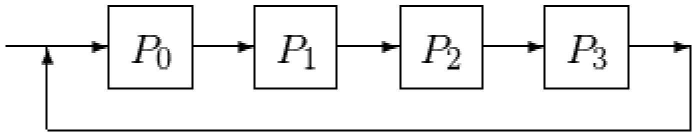


Fig. 6.5: Four stages in a pipeline executed by four processes in a ring.

In a manager/worker organization, node 0 receives the input and sends it to node~1. Every node i, for $i = 1, 2, \ldots, p{-}1$, does the following.

- It receives an item from node $i - 1$,
- performs operations on the item, and
- sends the processed item to node $(i + 1) \bmod p$.

At the end of one cycle, node 0 has the output.

6.1.4 Using MPI to implement a pipeline

Consider the following calculation with p processes. Process 0 prompts the user for a number and sends it to process 1. For $i > 0$: process i receives a number from process $i - 1$, doubles the number and sends it to process $i \bmod p$. A session of an MPI implementation of one pipeline cycle for this calculation shows the following:

```
$ mpirun -np 4 /tmp/pipe_ring
One pipeline cycle for repeated doubling.
Reading a number...
2
Node 0 sends 2 to the pipe...
Processor 1 receives 2 from node 0.
Processor 2 receives 4 from node 1.
Processor 3 receives 8 from node 2.
Node 0 received 16.
$
```

This example *is a type 1 pipeline*: efficient only if we have more than one instance to compute. The MPI code for the manager is below:

```
void manager ( int p )
/*
 * The manager prompts the user for a number and passes this number to node 1 for␣
↪doubling.
 * The manager receives from node p-1 the result. */
{
   int n;
   MPI_Status status;

   printf("One pipeline cycle for repeated doubling.\n");
   printf("Reading a number...\n"); scanf("%d",&n);
   printf("Node 0 sends %d to the pipe...\n",n);
```

(continues on next page)

(continued from previous page)

```
   fflush(stdout);
   MPI_Send(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD);
   MPI_Recv(&n,1,MPI_INT,p-1,tag,MPI_COMM_WORLD,&status);
   printf("Node 0 received %d.\n",n);
}
```

Following is the MPI code for the workers.

```
void worker ( int p, int i )
/*
 * Worker with identification label i of p receives a number,
 * doubles it, and sends it to node i+1 mod p. */
{
   int n;
   MPI_Status status;

   MPI_Recv(&n,1,MPI_INT,i-1,tag,MPI_COMM_WORLD,&status);
   printf("Processor %d receives %d from node %d.\n",i,n,i-1);
   fflush(stdout);
   n *= 2;                        /* double the number */
   if(i < p-1)
      MPI_Send(&n,1,MPI_INT,i+1,tag,MPI_COMM_WORLD);
   else
      MPI_Send(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD);
}
```

Let us consider the pipelined addition. Consider 4 processors in a ring topology as in Fig. 6.5. To add a sequence of 32 numbers, with data partitioning:

$$\underbrace{a_0, a_1, \ldots, a_7}_{A_k = \sum_{j=0}^{k} a_j}, \underbrace{b_0, b_1, \ldots, b_7}_{B_k = \sum_{j=0}^{k} b_j}, \underbrace{c_0, c_1, \ldots, c_7}_{C_k = \sum_{j=0}^{k} c_j}, \underbrace{d_0, d_1, \ldots, d_7}_{D_k = \sum_{j=0}^{k} d_j}.$$

The final sum is $S = A_7 + B_7 + C_7 + D_7$. Fig. 6.6 shows the space-time diagram for pipeline addition.

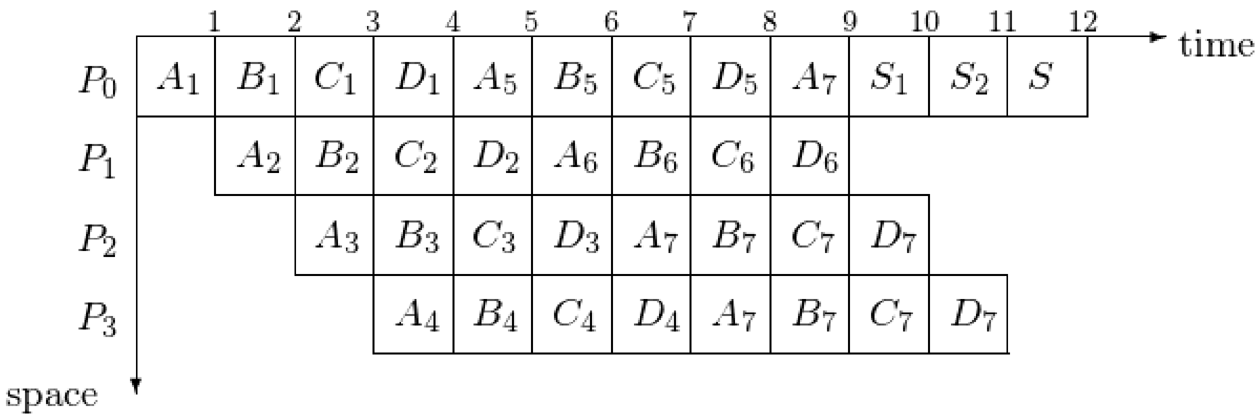

1
2
3
4
5
6
7
8
9
10
11
12
time
space
P_0
P_1
P_2
P_3
A_1 B_1 C_1 D_1 A_5 B_5 C_5 D_5 A_7 S_1 S_2 S
A_2 B_2 C_2 D_2 A_6 B_6 C_6 D_6
A_3 B_3 C_3 D_3 A_7 B_7 C_7 D_7
A_4 B_4 C_4 D_4 A_7 B_7 C_7 D_7

Fig. 6.6: Space-time diagram for pipelined addition, where $S_1 = A_7 + B_7, S_2 = S_1 + C_7, S = S_2 + D_7$.

Let us compute the speedup for this pipelined addition. We finished addition of 32 numbers in 12 cycles: 12 = 32/4 +

4. In general, with p-stage pipeline to add n numbers:

$$S(p) = \frac{n-1}{\frac{n}{p} + p}$$

For fixed p: $\lim_{n \rightarrow \infty} S(p) = p$.

A pipelined addition implemented with MPI using 5-stage pipeline shows the following on screen:

```
mpirun -np 5 /tmp/pipe_sum
The data to sum :  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26␣
↪27 28 29 30
Manager starts pipeline for sequence 0...
Processor 1 receives sequence 0 :  3 3 4 5 6
Processor 2 receives sequence 0 :  6 4 5 6
Processor 3 receives sequence 0 :  10 5 6
Processor 4 receives sequence 0 :  15 6
Manager received sum 21.
Manager starts pipeline for sequence 1...
Processor 1 receives sequence 1 :  15 9 10 11 12
Processor 2 receives sequence 1 :  24 10 11 12
Processor 3 receives sequence 1 :  34 11 12
Processor 4 receives sequence 1 :  45 12
Manager received sum 57.
Manager starts pipeline for sequence 2...
Processor 1 receives sequence 2 :  27 15 16 17 18
Processor 2 receives sequence 2 :  42 16 17 18
Processor 3 receives sequence 2 :  58 17 18
Processor 4 receives sequence 2 :  75 18
Manager received sum 93.
Manager starts pipeline for sequence 3...
Processor 1 receives sequence 3 :  39 21 22 23 24
Processor 2 receives sequence 3 :  60 22 23 24
Processor 3 receives sequence 3 :  82 23 24
Processor 4 receives sequence 3 :  105 24
Manager received sum 129.
Manager starts pipeline for sequence 4...
Processor 1 receives sequence 4 :  51 27 28 29 30
Processor 2 receives sequence 4 :  78 28 29 30
Processor 3 receives sequence 4 :  106 29 30
Processor 4 receives sequence 4 :  135 30
Manager received sum 165.
The total sum : 465
$
```

The MPI code is defined in the function below.

```
void pipeline_sum ( int i, int p ) /* performs a pipeline sum of p*(p+1) numbers */
{
   int n[p][p-i+1];
   int j,k;
   MPI_Status status;

   if(i==0)  /* manager generates numbers */
```

(continues on next page)

(continued from previous page)

```
   {
      for(j=0; j<p; j++)
         for(k=0; k<p+1; k++) n[j][k] = (p+1)*j+k+1;
      if(v>0)
      {
         printf("The data to sum : ");
         for(j=0; j<p; j++)
            for(k=0; k<p+1; k++) printf(" %d",n[j][k]);
         printf("\n");
      }
   }
   for(j=0; j<p; j++)
      if(i==0)  /* manager starts pipeline of j-th sequence */
      {
         n[j][1] += n[j][0];
         printf("Manager starts pipeline for sequence %d...\n",j);
         MPI_Send(&n[j][1],p,MPI_INT,1,tag,MPI_COMM_WORLD);
         MPI_Recv(&n[j][0],1,MPI_INT,p-1,tag,MPI_COMM_WORLD,&status);
         printf("Manager received sum %d.\n",n[j][0]);
      }
      else      /* worker i receives p-i+1 numbers */
      {
         MPI_Recv(&n[j][0],p-i+1,MPI_INT,i-1,tag,MPI_COMM_WORLD,&status);
         printf("Processor %d receives sequence %d : ",i,j);
         for(k=0; k<p-i+1; k++) printf(" %d", n[j][k]);
         printf("\n");
         n[j][1] += n[j][0];
         if(i < p-1)
            MPI_Send(&n[j][1],p-i,MPI_INT,i+1,tag,MPI_COMM_WORLD);
         else
            MPI_Send(&n[j][1],1,MPI_INT,0,tag,MPI_COMM_WORLD);
      }
   if(i==0)  /* manager computes the total sum */
   {
      for(j=1; j<p; j++) n[0][0] += n[j][0];
      printf("The total sum : %d\n",n[0][0]);
   }
}
```

6.1.5 Exercises

1. Describe the application of pipelining technique for grading n copies of an exam that has p questions. Explain the stages and make a space-time diagram.
2. Write code to use the 4-stage pipeline to double numbers for a sequence of 10 consecutive numbers starting at 2.
3. Consider the evaluation of a polynomial $f(x)$ of degree n given by its coefficient vector $(a_0, a_1, a_2, \ldots, a_n)$, using Horner's method, e.g., for $n = 4$: $f(x) = (((a_4 x + a_3)x + a_2)x + a_1)x + a_0$. Give code of this algorithm to evaluate f at a sequence of n values for x by a p-stage pipeline.

6.2 Pipelined Sorting, Sieving, Substitution

We continue our study of pipelined computations, for sorting, prime number generation, and solving triangular linear systems.

6.2.1 Sorting Numbers

As a data archival application, consider a pipeline with four computers in Fig. 6.7.

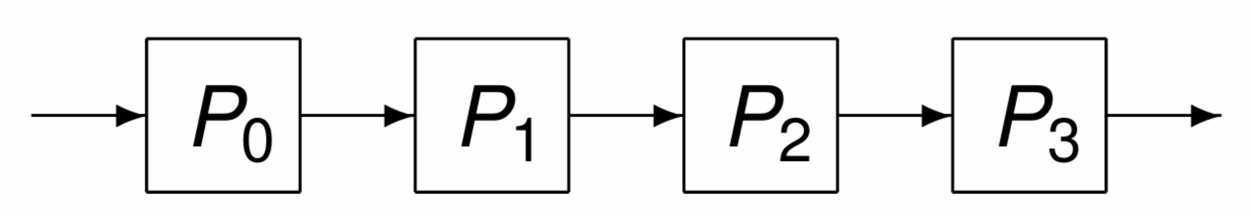


Fig. 6.7: A 4-stage pipeline in a data archival application.

The most recent data is stored on P_0.

1. When P_0 receives new data, its older data is moved to P_1.
2. When P_1 receives new data, its older data is moved to P_2.
3. When P_2 receives new data, its older data is moved to P_3.
4. When P_3 receives new data, its older data is archived to tape.

This is a type 1 pipeline. Every processor does the same three steps:

1. receive new data,
2. sort data,
3. send old data.

medskip

This leads to a pipelined sorting of numbers.

We consider a parallel version of insertion sort, sorting `p` numbers with `p` processors. Processor `i` does `p-i` steps in the algorithm:

```
for step 0 to p-i-1 do
    manager receives number
    worker i receives number from i-1
    if step = 0 then
       initialize the smaller number
    else if number > smaller number then
       send number to i+1
    else
       send smaller number to i+1;
       assign number to smaller number;
    end if;
end for.
```

A pipeline session with MPI can go as below.

```
$ mpirun -np 4 /tmp/pipe_sort
The 4 numbers to sort :  24 19 25 66
Manager gets 24.
Manager gets 19.
Node 0 sends 24 to 1.
Manager gets 25.
Node 0 sends 25 to 1.
Manager gets 66.
Node 0 sends 66 to 1.
Node 1 receives 24.
Node 1 receives 25.
Node 1 sends 25 to 2.
Node 1 receives 66.
Node 1 sends 66 to 2.
Node 2 receives 25.
Node 2 receives 66.
Node 2 sends 66 to 3.
Node 3 receives 66.
The sorted sequence :  19 24 25 66
```

MPI code for a pipeline version of insertion sort is in the program `pipe_sort.c` below:

```
int main ( int argc, char *argv[] )
{
   int i,p,*n,j,g,s;
   MPI_Status status;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&p);
   MPI_Comm_rank(MPI_COMM_WORLD,&i);
   if(i==0) /* manager generates p random numbers */
   {
      n = (int*)calloc(p,sizeof(int));
      srand(time(NULL));
      for(j=0; j<p; j++) n[j] = rand() % 100;
      if(v>0)
      {
         printf("The %d numbers to sort : ",p);
         for(j=0; j<p; j++) printf(" %d", n[j]);
         printf("\n"); fflush(stdout);
      }
   }
   for(j=0; j<p-i; j++)  /* processor i performs p-i steps */
      if(i==0)
      {
         g = n[j];
         if(v>0) { printf("Manager gets %d.\n",n[j]); fflush(stdout); }
         Compare_and_Send(i,j,&s,&g);
      }
      else
      {
         MPI_Recv(&g,1,MPI_INT,i-1,tag,MPI_COMM_WORLD,&status);
         if(v>0) { printf("Node %d receives %d.\n",i,g); fflush(stdout); }
         Compare_and_Send(i,j,&s,&g);
```

(continues on next page)

(continued from previous page)

```
      }
   MPI_Barrier(MPI_COMM_WORLD); /* to synchronize for printing */
   Collect_Sorted_Sequence(i,p,s,n);
   MPI_Finalize();
   return 0;
}
```

The function `Compare_and_Send` is defined next.

```
void Compare_and_Send ( int myid, int step, int *smaller, int *gotten )
/* Processor "myid" initializes smaller with gotten at step zero,
 * or compares smaller to gotten and sends the larger number through. */
{
   if(step==0)
      *smaller = *gotten;
   else
      if(*gotten > *smaller)
      {
         MPI_Send(gotten,1,MPI_INT,myid+1,tag,MPI_COMM_WORLD);
         if(v>0)
         {
            printf("Node %d sends %d to %d.\n",
                   myid,*gotten,myid+1);
            fflush(stdout);
         }
      }
      else
      {
         MPI_Send(smaller,1,MPI_INT,myid+1,tag,
                  MPI_COMM_WORLD);
         if(v>0)
         {
            printf("Node %d sends %d to %d.\n",
                   myid,*smaller,myid+1);
            fflush(stdout);
         }
         *smaller = *gotten;
      }
}
```

The function `Collect_Sorted_Sequence` follows:

```
void Collect_Sorted_Sequence ( int myid, int p, int smaller, int *sorted )
/* Processor "myid" sends its smaller number to the manager who collects
 * the sorted numbers in the sorted array, which is then printed. */
{
   MPI_Status status;
   int k;
   if(myid==0) {
      sorted[0] = smaller;
      for(k=1; k<p; k++)
         MPI_Recv(&sorted[k],1,MPI_INT,k,tag,
```

(continues on next page)

(continued from previous page)

```
                MPI_COMM_WORLD,&status);
      printf("The sorted sequence : ");
      for(k=0; k<p; k++) printf(" %d",sorted[k]);
      printf("\n");
   }
   else
      MPI_Send(&smaller,1,MPI_INT,0,tag,MPI_COMM_WORLD);
}
```

6.2.2 Prime Number Generation

The sieve of Erathostenes is shown in Fig. 6.8.

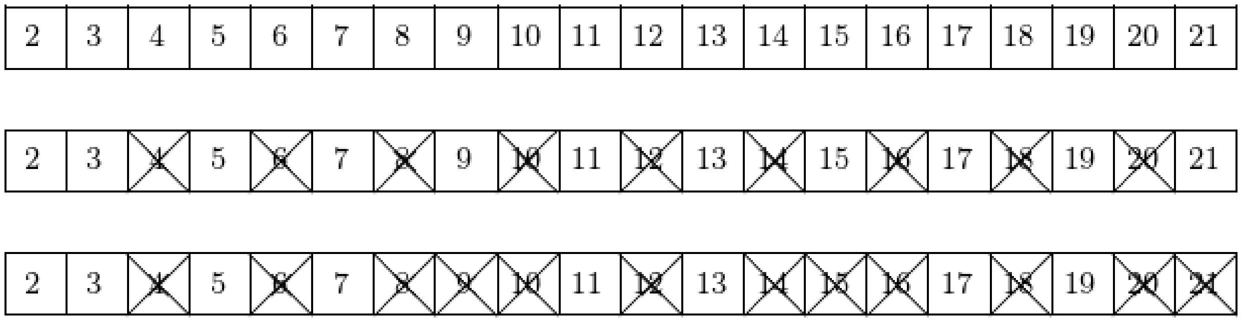


Fig. 6.8: Wiping out all multiples of 2 and 3 gives all prime numbers between 2 and 21.

A pipelined sieve algorithm is defined as follows. One stage in the pipeline

1. receives a prime,
2. receives a sequence of numbers,
3. extracts from the sequence all multiples of the prime, and
4. sends the filtered list to the next stage.

This pipeline algorithm is of *type 2*. As in type 1, multiple input items are needed for speedup; but the amount of work in every stage will complete fewer steps than in the preceding stage.

For example, consider a 2-stage pipeline to compute all primes ≤ 21 with the sieve algorithm:

1. wipe out all multiples of 2, in nine multiplications;
2. wipe out all multiples of 3, in five multiplications.

Although the second stage in the pipeline starts only after we determined that 3 is not a multiple of 2, there are fewer multiplications in the second stage. The space-time diagram with the multiplications is in Fig. 6.9.

A parallel implementation of the sieve of Erathostenes is in the examples collection of the Intel TBB distribution, in `/usr/local/tbb40_20131118oss/examples/parallel_reduce/primes`. Computations on a 16-core computer `kepler`:

```
$ make
g++ -O2 -DNDEBUG  -o primes main.cpp primes.cpp -ltbb -lrt
./primes
#primes from [2..100000000] = 5761455 (0.106599 sec with serial code)
```

(continues on next page)

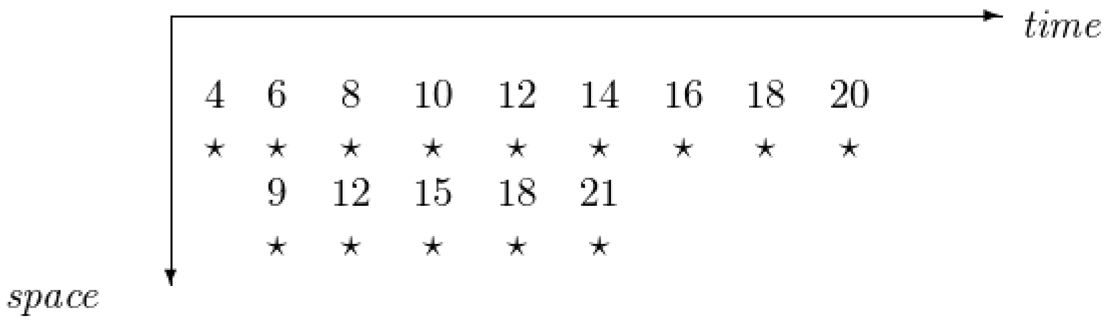


Fig. 6.9: A 2-stage pipeline to compute all primes ≤ 21.

(continued from previous page)

```
#primes from [2..100000000] = 5761455 (0.115669 sec with 1-way parallelism)
#primes from [2..100000000] = 5761455 (0.059511 sec with 2-way parallelism)
#primes from [2..100000000] = 5761455 (0.0393051 sec with 3-way parallelism)
#primes from [2..100000000] = 5761455 (0.0287207 sec with 4-way parallelism)
#primes from [2..100000000] = 5761455 (0.0237532 sec with 5-way parallelism)
#primes from [2..100000000] = 5761455 (0.0198929 sec with 6-way parallelism)
#primes from [2..100000000] = 5761455 (0.0175456 sec with 7-way parallelism)
#primes from [2..100000000] = 5761455 (0.0168987 sec with 8-way parallelism)
#primes from [2..100000000] = 5761455 (0.0127005 sec with 10-way parallelism)
#primes from [2..100000000] = 5761455 (0.0116965 sec with 12-way parallelism)
#primes from [2..100000000] = 5761455 (0.0104559 sec with 14-way parallelism)
#primes from [2..100000000] = 5761455 (0.0109771 sec with 16-way parallelism)
#primes from [2..100000000] = 5761455 (0.00953452 sec with 20-way parallelism)
#primes from [2..100000000] = 5761455 (0.0111944 sec with 24-way parallelism)
#primes from [2..100000000] = 5761455 (0.0107475 sec with 28-way parallelism)
#primes from [2..100000000] = 5761455 (0.0151389 sec with 32-way parallelism)
elapsed time : 0.520726 seconds
$
```

6.2.3 Solving Triangular Systems

We apply a type 3 pipeline to solve a triangular linear system. With forward substitution formulas we solve a lower triangular system.

The LU factorization of a matrix A reduces the solving of a linear system to solving two triangular systems. To solve an n-dimensional linear system $A\mathbf{x} = \mathbf{b}$ we factor A as a product of two triangular matrices, $A = LU$:

- L is lower triangular, $L = [\ell_{i,j}]$, $\ell_{i,j} = 0$ if $j > i$ and $\ell_{i,i} = 1$.
- U is upper triangular $U = [u_{i,j}]$, $u_{i,j} = 0$ if $i > j$.

Solving $A\mathbf{x} = \mathbf{b}$ is equivalent to solving $L(U\mathbf{x}) = \mathbf{b}$:

1. Forward substitution: $L\mathbf{y} = \mathbf{b}$.
2. Backward substitution: $U\mathbf{x} = \mathbf{y}$.

Factoring A costs $O(n^3)$, solving triangular systems costs $O(n^2)$.

Expanding the matrix-vector product $L\mathbf{y}$ in $L\mathbf{y} = \mathbf{b}$ leads to formulas for forward substitution:

$$\left\{ \begin{array}{lcl} y_1 & = & b_1 \\ \ell_{2,1} y_1 + y_2 & = & b_2 \\ \ell_{3,1} y_1 + \ell_{3,2} y_2 + y_3 & = & b_3 \\ & \vdots & \\ \ell_{n,1} y_1 + \ell_{n,2} y_2 + \ell_{n,3} y_3 + \cdots + \ell_{n,n-1} y_{n-1} + y_n & = & b_n \end{array} \right.$$

and solving for the diagonal elements gives

$$\begin{array}{lcl} y_1 & = & b_1 \\ y_2 & = & b_2 - \ell_{2,1} y_1 \\ y_3 & = & b_3 - \ell_{3,1} y_1 - \ell_{3,2} y_2 \\ & \vdots & \\ y_n & = & b_n - \ell_{n,1} y_1 - \ell_{n,2} y_2 - \cdots - \ell_{n,n-1} y_{n-1} \end{array}$$

The formulas lead to an algorithm. For $k = 1, 2, \ldots, n$:

$$y_k = b_k - \sum_{i=1}^{k-1} \ell_{k,i} y_i.$$

Formulated as an algorithm, in pseudocode:

```
for k from 1 to n do
    y[k] := b[k]
    for i from 1 to k-1 do
        y[k] := y[k] - L[k][i]*y[i].
```

We count $1 + 2 + \cdots + n - 1 = \dfrac{n(n-1)}{2}$ multiplications and subtractions.

Pipelines are classified into three types:

1. Type 1: Speedup only if multiple instances. Example: instruction pipeline.
2. Type 2: Speedup already if one instance. Example: pipeline sorting.
3. Type 3: Worker continues after passing information through. Example: solve $L\mathbf{y} = \mathbf{b}$.

Typical for the 3rd type of pipeline is the varying length of each job, as exemplified in Fig. 6.10.

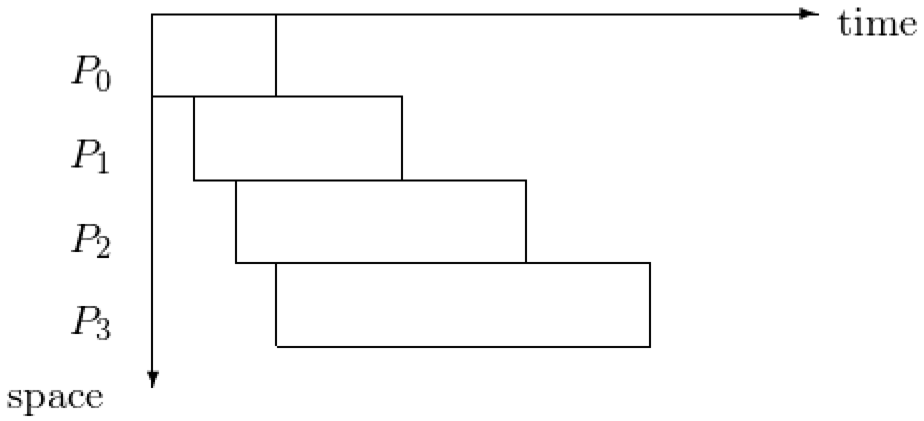


Fig. 6.10: Space-time diagram for pipeline with stages of varying length.

Using an n-stage pipeline, we assume that L is available on every processor.

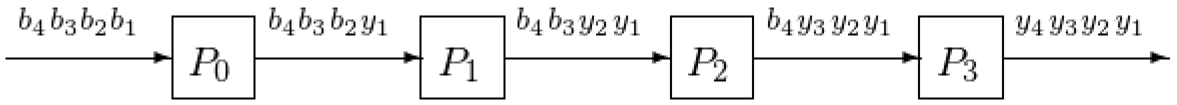


Fig. 6.11: 4-stage pipeline to solve a 4-by-4 lower triangular system.

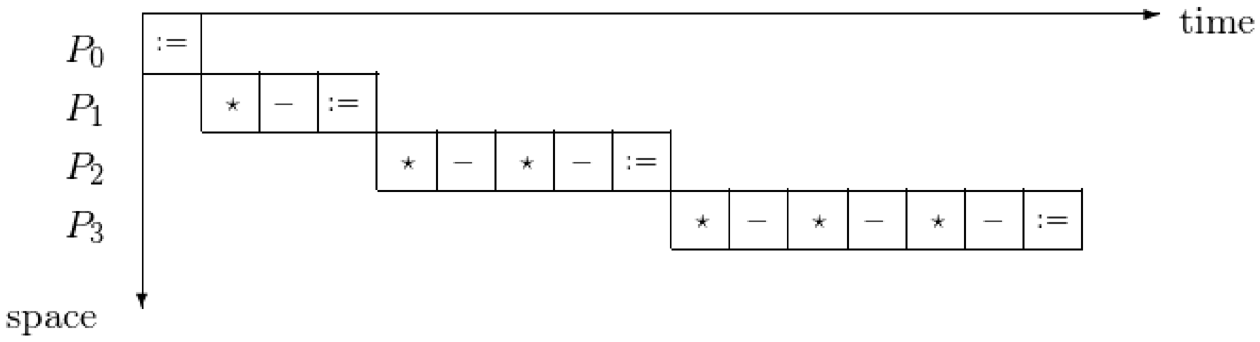


Fig. 6.12: Space-time diagram for solving a 4-by-4 lower triangular system.

The corresponding 4-stage pipeline is shown in Fig. 6.11 with the space-time diagram in Fig. 6.12.

In type 3 pipelining, a worker continues after passing results through. The making of y_1 available in the next pipeline cycle is illustrated in Fig. 6.13. The corresponding space-time diagram is in Fig. 6.14 and the space-time diagram in Fig. 6.15 shows at what time step which component of the solution is.

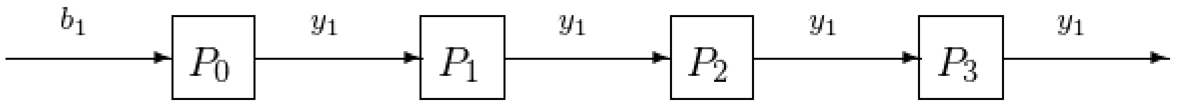


Fig. 6.13: Passing y_1 through the type 3 pipeline.

We count the steps for $p = 4$ or in general, for $p = n$ as follows. The latency takes 4 steps for y_1 to be at P_4, or in general: n steps for y_1 to be at P_n. It takes then 6 additional steps for y_4 to be computed by P_4, or in general: $2n - 2$ additional steps for y_n to be computed by P_n. So it takes $n + 2n - 2 = 3n - 2$ steps to solve an n-dimensional triangular system by an n-stage pipeline.

```
y := b
for i from 2 to n do
    for j from i to n do
        y[j] := y[j] - L[j][i-1]*y[i-1]
```

Consider for example the solving of $L\mathbf{y} = \mathbf{b}$ for $n = 5$.

1. $y := \mathbf{b}$;
2. $y_2 := y_2 - \ell_{2,1} \star y_1$;

   $y_3 := y_3 - \ell_{3,1} \star y_1$;

   $y_4 := y_4 - \ell_{4,1} \star y_1$;

   $y_5 := y_5 - \ell_{5,1} \star y_1$;
3. $y_3 := y_3 - \ell_{3,2} \star y_2$;

   $y_4 := y_4 - \ell_{4,2} \star y_2$;

   $y_5 := y_5 - \ell_{5,2} \star y_2$;

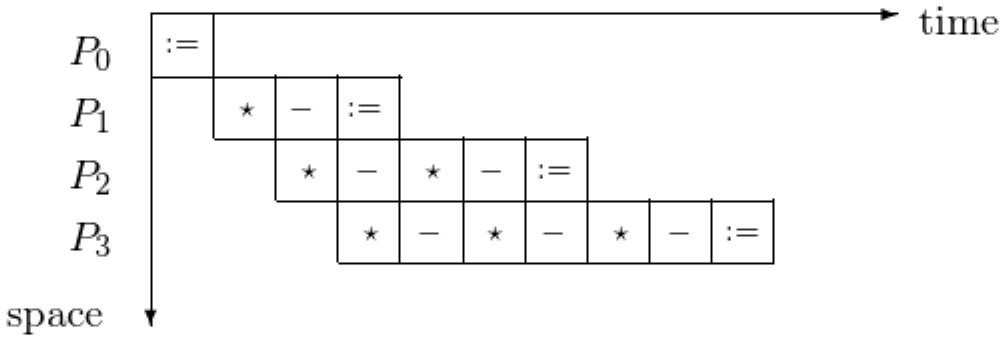


Fig. 6.14: Space-time diagram of a type 3 pipeline.

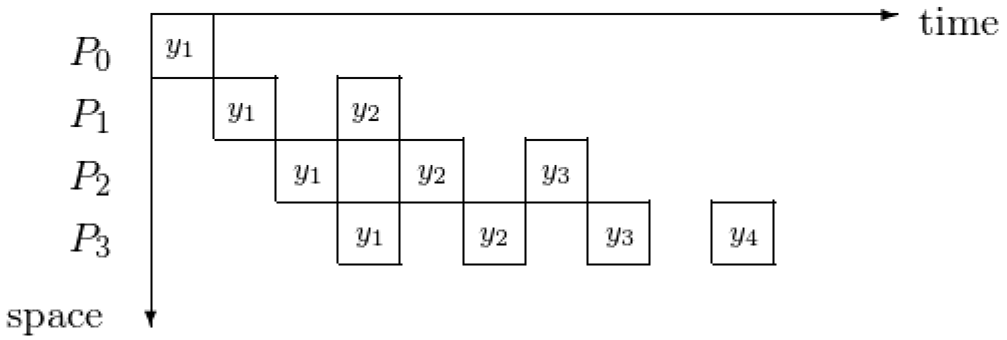


Fig. 6.15: Space-time diagram illustrates the component of the solutions.

4. $y_4 := y_4 - \ell_{4,3} \star y_3$;

   $y_5 := y_5 - \ell_{5,3} \star y_3$;

5. $y_5 := y_5 - \ell_{5,4} \star y_4$;

Observe that all instructions in the *j* loop are independent from each other!

Consider the inner loop in the algorithm to solve $L\mathbf{y} = \mathbf{b}$. We distribute the update of $y_i, y_{i+1}, \ldots, y_n$ among p processors. If $n \gg p$, then we expect a close to optimal speedup.

6.2.4 Bibliography

1. Shahid H. Bokhari. **Multiprocessing the Sieve of Eratosthenes.** *Computer* 20(4):50-58, 1987.
2. B. Wilkinson and M. Allen. *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 2nd edition, 2005.

6.2.5 Exercises

1. Write the pipelined sorting algorithm with OpenMP or Julia.

   Demonstrate the correctness of your implementation with some good examples.

2. Use message passing to implement the pipelined sieve algorithm.

   Relate the number of processors in the network to the number of multiples which must be computed before sending off the sequence to the next processor.

3. Implement the pipelined sieve algorithm with OpenMP and Julia.

   Can the constraint on the number of computed multiples be formulated with dependencies?

4. Consider the upper triangular system $U\mathbf{x} = \mathbf{y}$, with $U = [u_{i,j}]$, $u_{i,j} = 0$ if $i > j$.

   Derive the formulas and general algorithm to compute the components of the solution $\mathbf{x}$.

   For $n = 4$, draw the third type of pipeline.

6.3 Solving Triangular Systems

Triangular linear system occur as the result of an LU or a QR decomposition.

6.3.1 Ill Conditioned Matrices and Quad Doubles

Consider the 4-by-4 lower triangular matrix

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ -2 & -2 & 1 & 0 \\ -2 & -2 & -2 & 1 \end{bmatrix}.$$

What we know from numerical analysis:

1. The condition number of a matrix magnifies roundoff errors.
2. The hardware double precision is $2^{-52} \approx 2.2 \times 10^{-16}$.
3. We get no accuracy from condition numbers larger than 10^{16}.

An experiment in an interactive Julia session illustate that ill conditioned matrices occur already in modest dimensions.

```
julia> using LinearAlgebra

julia> A = ones(32,32);

julia> D = Diagonal(A);

julia> L = LowerTriangular(A);

julia> LmD = L - D;

julia> L2 = D - 2*LmD;

julia> cond(L2)
2.41631630569077e16
```

The condition number is estimated at 2.4×10^{16}.

A floating-point number consists of a sign bit, exponent, and a fraction (also known as the mantissa). Almost all microprocessors follow the IEEE 754 standard. GPU hardware supports 32-bit (single float) and for compute capability ≥ 1.3 also double floats.

Numerical analysis studies algorithms for continuous problems, problems for their sensitivity to errors in the input; and algorithms for their propagation of roundoff errors.

The floating-point addition is *not* associative! Parallel algorithms compute and accumulate the results in an order that is different from their sequential versions. For example, adding a sequence of numbers is more accurate if the numbers are sorted in increasing order.

Instead of speedup, we can ask questions about quality up:

- If we can afford to keep the total running time constant, does a faster computer give us more accurate results?
- How many more processors do we need to guarantee a result?

A quad double is an unevaluated sum of 4 doubles, improves the working precision from 2.2×10^{-16} to 2.4×10^{-63}. The software QDlib is presented in the paper *Algorithms for quad-double precision floating point arithmetic* by Y. Hida, X.S. Li, and D.H. Bailey, published in the 15th IEEE Symposium on Computer Arithmetic, pages 155-162. IEEE, 2001. The software is available at <http://crd.lbl.gov/~dhbailey/mpdist>.

A quad double builds on `double double`. Some features of working with doubles double are:

- The least significant part of a double double can be interpreted as a compensation for the roundoff error.
- Predictable overhead: working with double double is of the same cost as working with complex numbers.

Consider Newton's method to compute :math:sqrt{x}: as defined in the code below.

```
#include <iostream>
#include <iomanip>
#include <qd/qd_real.h>
using namespace std;

qd_real newton ( qd_real x )
{
   qd_real y = x;
   for(int i=0; i<10; i++)
      y -= (y*y - x)/(2.0*y);
   return y;
}
```

The main program is as follows.

```
int main ( int argc, char *argv[] )
{
   cout << "give x : ";
   qd_real x; cin >> x;
   cout << setprecision(64);
   cout << "         x : " << x << endl;

   qd_real y = newton(x);
   cout << "  sqrt(x) : " << y << endl;

   qd_real z = y*y;
   cout << "sqrt(x)^2 : " << z << endl;
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

If the program is in the file `newton4sqrt.cpp` and the makefile contains

```
QD_ROOT=/usr/local/qd-2.3.13
QD_LIB=/usr/local/lib

newton4sqrt:
        g++ -I$(QD_ROOT)/include newton4sqrt.cpp \
              $(QD_LIB)/libqd.a -o /tmp/newton4sqrt
```

then we can create the executable, simply typing `make newton4sqrt`. A run with the code for *sqrt2* is shown below.

```
2.0000000000000000000000000000000000000000000000000000000000000000e+00
1.5000000000000000000000000000000000000000000000000000000000000000e+00
1.4166666666666666666666666666666666666666666666666666666666666667e+00
1.4142156862745098039215686274509803921568627450980392156862745098e+00
1.4142135623746899106262955788901349101165596221157440445849050192e+00
1.4142135623730950488016896235025302436149819257761974284982894987e+00
1.4142135623730950488016887242096980785696718753772340015610131332e+00
1.4142135623730950488016887242096980785696718753769480731766797380e+00
1.4142135623730950488016887242096980785696718753769480731766797380e+00
residual : 0.0000e+00
```

6.3.2 On a Parallel Shared Memory Computer with OpenMP

We will rewrite the formulas for forward substitution. Expanding the matrix-vector product $L\mathbf{y}$ in $L\mathbf{y} = \mathbf{b}$ leads to

$$\left\{ \begin{array}{lcl} y_1 & = & b_1 \\ \ell_{2,1} y_1 + y_2 & = & b_2 \\ \ell_{3,1} y_1 + \ell_{3,2} y_2 + y_3 & = & b_3 \\ & \vdots & \\ \ell_{n,1} y_1 + \ell_{n,2} y_2 + \ell_{n,3} y_3 + \cdots + \ell_{n,n-1} y_{n-1} + y_n & = & b_n \end{array} \right.$$

and solving for the diagonal elements gives

$$\begin{array}{lcl} y_1 & = & b_1 \\ y_2 & = & b_2 - \ell_{2,1} y_1 \\ y_3 & = & b_3 - \ell_{3,1} y_1 - \ell_{3,2} y_2 \\ & \vdots & \\ y_n & = & b_n - \ell_{n,1} y_1 - \ell_{n,2} y_2 - \cdots - \ell_{n,n-1} y_{n-1} \end{array}$$

In rewriting the formulas, consider the case for $n = 5$. Solving $L\mathbf{y} = \mathbf{b}$ for $n = 5$:

1. $\mathbf{y} := \mathbf{b}$
2. $y_2 := y_2 - \ell_{2,1} \star y_1$

   $y_3 := y_3 - \ell_{3,1} \star y_1$

   $y_4 := y_4 - \ell_{4,1} \star y_1$

   $y_5 := y_5 - \ell_{5,1} \star y_1$

3. $y_3 := y_3 - \ell_{3,2} \star y_2$

   $y_4 := y_4 - \ell_{4,2} \star y_2$

   $y_5 := y_5 - \ell_{5,2} \star y_2$
4. $y_4 := y_4 - \ell_{4,3} \star y_3$

   $y_5 := y_5 - \ell_{5,3} \star y_3$
5. $y_5 := y_5 - \ell_{5,4} \star y_4$

In the algorithm

```
$\mathbf{y} := \mathbf{b}$
for $i$ from 2 to $n$ do
    for $j$ from $i$ to $n$ do
        $y_j := y_j - \ell_{j,i-1} \star y_{i-1}$
```

Observe that all instructions in the j loop are independent from each other! Considering the inner loop in the algorithm to solve $L\mathbf{y} = \mathbf{b}$, we distribute the update of $y_i, y_{i+1}, \ldots, y_n$ among p processors. If $n \gg p$, then we expect a close to optimal speedup.

For our parallel solver for triangular systems:

- For $L = [\ell_{i,j}]$, we generate random numbers for $\ell_{i,j} \in [0, 1]$.

  The exact solution $\mathbf{y}$: $y_i = 1$, for $i = 1, 2, \ldots, n$.

  We compute the right hand side $\mathbf{b} = L\mathbf{y}$.
- Even already in small dimensions, the condition number may grow exponentially.

  Hardware double precision is insufficient. Therefore, we use quad double arithmetic.
- We use a straightforward OpenMP implementation.

In solving random lower triangular systems, relying on hardware doubles is problematic:

```
$ time ./trisol 10
last number : 1.0000000000000009e+00

real   0m0.003s    user   0m0.001s    sys   0m0.002s

$ time ./trisol 100
last number : 9.9999999999974221e-01

real   0m0.005s    user   0m0.001s    sys   0m0.002s

$ time ./trisol 1000
last number : 2.7244600009080568e+04

real   0m0.036s    user   0m0.025s    sys   0m0.009s
```

For a matrix of quad doubles, allocating data in the main program is done in the code snippet below.

```
qd_real b[n],y[n];

qd_real **L;
L = (qd_real**) calloc(n,sizeof(qd_real*));
for(int i=0; i<n; i++)
```

(continues on next page)

(continued from previous page)

```
   L[i] = (qd_real*) calloc(n,sizeof(qd_real));

srand(time(NULL));
random_triangular_system(n,L,b);
```

Generating a random triangular system happens through the function defined next.

```
void random_triangular_system
 ( int n, qd_real **L, qd_real *b )
{
   for(int i=0; i<n; i++)
   {
      L[i][i] = 1.0;
      for(j=0; j<i; j++)
      {
         double r = ((double) rand())/RAND_MAX;
         L[i][j] = qd_real(r);
      }
      for(int j=i+1; j<n; j++)
         L[i][j] = qd_real(0.0);
   }
   for(int i=0; i<n; i++)
   {
      b[i] = qd_real(0.0);
      for(int j=0; j<n; j++)
         b[i] = b[i] + L[i][j];
   }
}
```

Then the triangular system is solved by the following code.

```
void solve_triangular_system_swapped
 ( int n, qd_real **L, qd_real *b, qd_real *y )
{
   for(int i=0; i<n; i++) y[i] = b[i];

   for(int i=1; i<n; i++)
   {
      for(int j=i; j<n; j++)
         y[j] = y[j] - L[j][i-1]*y[i-1];
   }
}
```

Using OpenMP, we add directives, creating a parallel section, declaring which variables are shared, and which are not.

```
void solve_triangular_system_swapped
 ( int n, qd_real **L, qd_real *b, qd_real *y )
{
   int j;

   for(int i=0; i<n; i++) y[i] = b[i];

   for(int i=1; i<n; i++)
```

(continues on next page)

(continued from previous page)

```
    {
       #pragma omp parallel shared(L,y) private(j)
       {
          #pragma omp for
          for(j=i; j<n; j++)
             y[j] = y[j] - L[j][i-1]*y[i-1];
       }
    }
}
```

For dimension $n = 8,000$, for varying number p of cores, Table 6.1 summarizes the running of `time ./trisol_qd_omp n p` on 12-core Intel X5690, 3.47 GHz.

Table 6.1: solving a triangular system for p cores.

p	cpu time	real	user	sys
1	21.240s	35.095s	34.493s	0.597s
2	22.790s	25.237s	36.001s	0.620s
4	22.330s	19.433s	35.539s	0.633s
8	23.200s	16.726s	36.398s	0.611s
12	23.260s	15.781s	36.457s	0.626s

The serial part is the generation of the random numbers for L and the computation of $\mathbf{b} = L\mathbf{y}$. Recall Amdahl's Law.

We can compute the serial time, subtracting for $p = 1$, from the real time the cpu time spent in the solver, i.e.: $35.095 - 21.240 = 13.855$. For $p = 12$, time spent on the solver is $15.781 - 13.855 = 1.926$. Compare 1.926 to 21.240/12 = 1.770.

6.3.3 Accelerated Back Substitution

Consider a 3-by-3-tiled upper triangular system $U\mathbf{x} = \mathbf{b}$.

$$U = \left[\begin{array}{ccc} U_1 & A_{1,2} & A_{1,3} \\ & U_2 & A_{2,3} \\ & & U_3 \end{array} \right], \quad \mathbf{x} = \left[\begin{array}{c} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{array} \right], \quad \mathbf{b} = \left[\begin{array}{c} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \end{array} \right],$$

where U_1, U_2, U_3 are upper triangular, with nonzero diagonal elements.

Invert all diagonal tiles:

$$\left[\begin{array}{ccc} U_1^{-1} & A_{1,2} & A_{1,3} \\ & U_2^{-1} & A_{2,3} \\ & & U_3^{-1} \end{array} \right].$$

- The inverse of an upper triangular matrix is upper triangular.
- Solve an upper triangular system for each column of the inverse.
- The columns of the inverse can be computed independently.

$\Rightarrow$ Solve many smaller upper triangular systems in parallel.

After inverting the diagonal tiles, in the second stage, we solve $U\mathbf{x} = \mathbf{b}$ for

$$U = \left[\begin{array}{ccc} U_1^{-1} & A_{1,2} & A_{1,3} \\ & U_2^{-1} & A_{2,3} \\ & & U_3^{-1} \end{array} \right]$$

in the following steps:

1. $\mathbf{x}_3 := U_3^{-1}\mathbf{b}_3$,
2. $\mathbf{b}_2 := \mathbf{b}_2 - A_{2,3}\mathbf{x}_3$, $\mathbf{b}_1 := \mathbf{b}_1 - A_{1,3}\mathbf{x}_3$,
4. $\mathbf{x}_2 := U_2^{-1}\mathbf{b}_2$,
5. $\mathbf{b}_1 := \mathbf{b}_1 - A_{1,2}\mathbf{x}_2$,
6. $\mathbf{x}_1 := U_1^{-1}\mathbf{b}_1$.

Statements on the same line can be executed in parallel. In multiple double precision, several blocks of threads collaborate in the computation of one matrix-vector product.

The two stages are executed by three kernels.

Algorithm 1: *Tiled Accelerated Back Substitution*.

On input are the following :

- N is the number of tiles,
- n is the size of each tile,
- U is an upper triangular Nn-by-Nn matrix,
- $\mathbf{b}$ is a vector of size Nn.

The output is $\mathbf{x}$ is a vector of size Nn: $U\mathbf{x} = \mathbf{b}$.

1. Let $U_1, U_2, \ldots, U_N$ be the diagonal tiles.

   The *k*-th thread solves $U_i\mathbf{v}_k = \mathbf{e}_k$, computing the *k*-th column U_i^{-1}.
2. For $i = N, N-1, \ldots, 1$ do
   1. n threads compute $\mathbf{x}_i = U^{-1}\mathbf{b}_i$;
   2. simultaneously update $\mathbf{b}_j$ with $\mathbf{b}_j - A_{j,i}\mathbf{x}_i$, $j \in \{1, 2, \ldots, i-1\}$ with $i-1$ blocks of n threads.

A parallel execution could run in time proportional to Nn.

For efficiency, we must stage the data right. A matrix U of multiple doubles is stored as $[U_1, U_2, \ldots, U_m]$,

- U_1 holds the most significant doubles of U,
- U_m holds the least significant doubles of U.

Similarly, $\mathbf{b}$ is an array of m arrays $[\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_m]$, sorted in the order of significance. In complex data, real and imaginary parts are stored separately.

The main advantages of this representation are twofold:

- facilitates staggered application of multiple double arithmetic,
- benefits efficient memory coalescing, as adjacent threads in one block of threads read/write adjacent data in memory, avoiding bank conflicts.

In the experimental setup, about the input matrices:

- Random numbers are generated for the input matrices.
- Condition numbers of random triangular matrices almost surely grow exponentially *[Viswanath and Trefethen, 1998]*.
- In the standalone tests, the upper triangular matrices are the Us of an LU factorization of a random matrix, computed by the host.

Two input parameters are set for every run:

- The size of each tile is the number of threads in a block. The tile size is a multiple of 32.
- The number of tiles equals the number of blocks. As the V100 has 80 streaming multiprocessors, the number of tiles is at least 80.

The units of the flops in Table 6.2 and Table 6.3 are Gigaflops.

Table 6.2: Back substitution in double double precision on the V100.

stage in Algorithm 1	64×80	128×80	256×80
invert diagonal tiles	1.2	9.3	46.3
multiply with inverses	1.7	3.3	8.9
back substitution	7.9	4.7	12.2
time spent by kernels	5.0	17.3	67.4
wall clock time	82.0	286.0	966.0
kernel time flops	190.6	318.7	525.1
wall clock flops	11.7	19.2	36.7

Table 6.3: Back substitution in quad double precision on the V100.

stage in Algorithm 1	64×80	128×80	256×80
invert diagonal tiles	6.2	38.3	137.4
multiply with inverses	12.2	23.8	63.1
back substitution	13.3	26.7	112.2
time spent by kernels	31.7	88.8	312.7
wall clock time	187.0	619.0	2268.0
kernel time flops	299.4	614.2	1122.3
wall clock flops	50.8	88.1	154.8

Consider the doubling of the dimension and the precision.

1. Double the dimension, expect the time to quadruple.
2. From double double to quad double: 11.7 is multiplier, from quad double to octo double: 5.4 times longer.

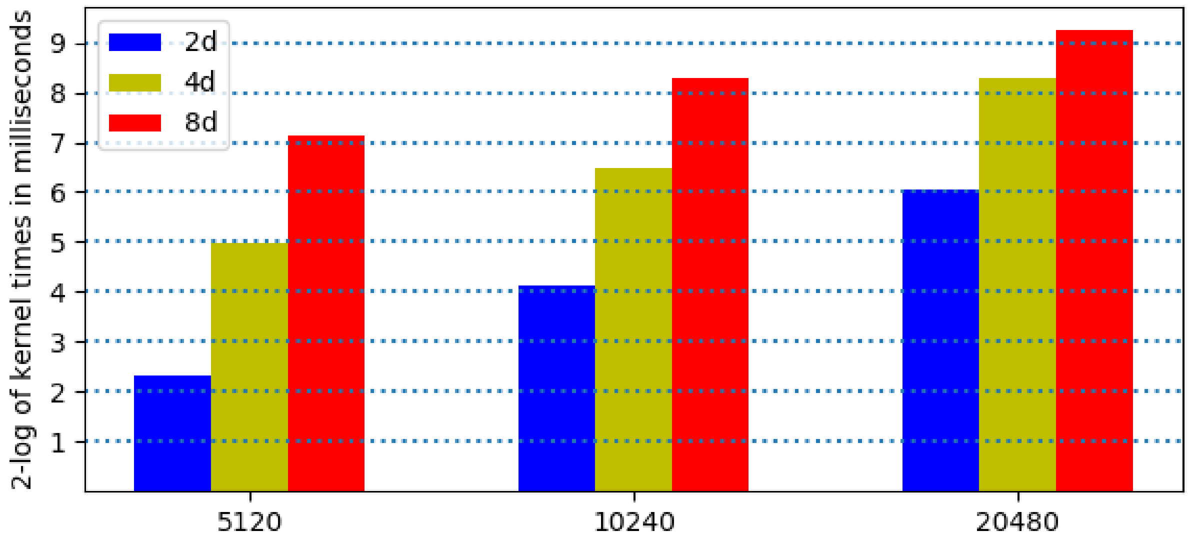


Fig. 6.16: 2-logarithms of times on the V100 in 3 precisions

In Fig. 6.16, the heights of the bars are closer to each other in higher dimensions.

The V100 has 80 multiprocessors, its theoretical peak performance is 1.68 times that of the P100.

The value for N is fixed at 80, n runs from 32 to 256, see Fig. 6.17.

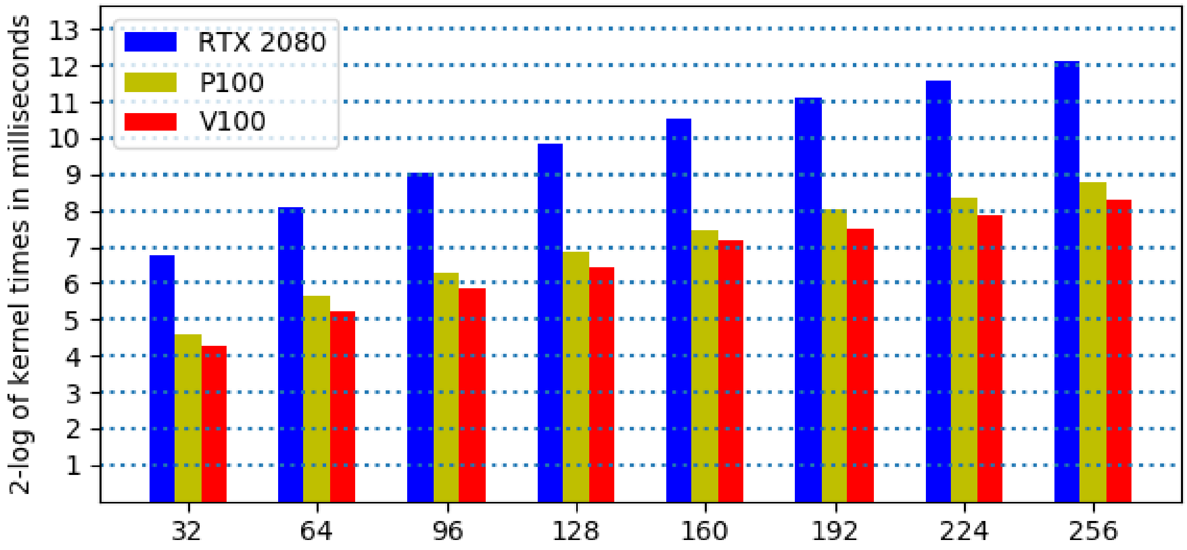


Fig. 6.17: kernel times in quad double precision on 3 GPUs

In Fig. 6.17, observe the heights of the bars as the dimensions double and the relative performance of the three different GPUs.

Considering $20480 = 320 \times 64 = 160 \times 128 = 80 \times 256$, we run back substitution in quad double precision, for $20480 = N \times n$, for three different combinations of N and n, on the V100. The results are summarized in Table 6.4.

Table 6.4: Back substitution in quad double precision.

stage in Algorithm 1	320×64	160×128	80×256
invert diagonal tiles	13.5	35.8	132.3
multiply with inverses	49.0	47.5	64.3
back substitution	84.6	91.7	112.3
time spent by kernels	147.1	175.0	308.9
wall clock time	2620.0	2265.0	2071.0
kernel time flops	683.0	861.1	1136.1
wall clock flops	38.3	66.5	169.5

The units of all times in Table 6.4 are milliseconds, flops unit is Gigaflops.

6.3.4 Bibliography

- T.J. Dekker. **A floating-point technique for extending the available precision.** *Numerische Mathematik*, 18(3):224-242, 1971.
- D. H. Heller. **A survey of parallel algorithms in numerical linear algebra.** *SIAM Review*, 20(4):740-777, 1978.
- Y. Hida, X.S. Li, and D.H. Bailey. **Algorithms for quad-double precision floating point arithmetic.** In *15th IEEE Symposium on Computer Arithmetic*, pages 155-162. IEEE, 2001. Software at <http://crd.lbl.gov/~dhbailey/mpdist>.
- N. J. Higham. **Stability of parallel triangular system solvers.** *SIAM J. Sci. Comput.*, 16(2):400-413, 1995.
- M. Joldes, J.-M. Muller, V. Popescu, W. Tucker. **CAMPARY: Cuda Multiple Precision Arithmetic Library and Applications.** In *Mathematical Software – ICMS 2016, the 5th International Conference on Mathematical Software*, pages 232-240, Springer-Verlag, 2016.
- M. Lu, B. He, and Q. Luo. **Supporting extended precision on graphics processors.** In A. Ailamaki and P.A. Boncz, editors, *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010), June 7, 2010, Indianapolis, Indiana*, pages 19-26, 2010. Software at <https://code.google.com/archive/p/gpuprec>.
- W. Nasri and Z. Mahjoub. **Optimal parallelization of a recursive algorithm for triangular matrix inversion on MIMD computers.** *Parallel Computing*, 27:1767-1782, 2001.
- J. Verschelde. **Least squares on GPUs in multiple double precision.** In *The 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 828-837. IEEE, 2022. Code at <https://github.com/janverschelde/PHCpack/src/GPU>.
- D. Viswanath and L. N. Trefethen. **Condition numbers of random triangular matrices.** *SIAM J. Matrix Anal. Appl.*, 19(2):564-581, 1998.

6.3.5 Exercises

1. Write a parallel solver with OpenMP to solve $U\mathbf{x} = \mathbf{y}$.

   Take for U a matrix with random numbers in $[0, 1]$, compute $\mathbf{y}$ so all components of $\mathbf{x}$ equal one. Test the speedup of your program, for large enough values of n and a varying number of cores.

2. Describe a parallel solver for upper triangular systems $U\mathbf{y} = \mathbf{b}$ for distributed memory computers. Write a prototype implementation using MPI and discuss its scalability.

3. Consider a tiled lower triangular system $L\mathbf{x} = \mathbf{b}$.

6.4 Barriers for Synchronizations

For message passing, we distinguish between a linear, a tree, and a butterfly barrier, introducing the `sendrecv` in MPI. As an example of a data parallel algorithm, we describe the prefix sum algorithm. In the third subsection, Brent's theorem is stated and applied to the parallel summation problem.

6.4.1 Synchronizing Computations

In synchronized computations, processors pass through a number of stages in an algorithm.

Definition of synchronization barrier

A *synchronization barrier* guarantees that no processor continues to the next stage until all processors have finished the current stage.

In the above definition, the *processor* stands for a process, thread, or task. Examples in distributed memory, shared memory, and accelerated parallel processing are

- Message passing defines `MPI_Barrier(MPI_Comm comm)`.
- OpenMP has the `#pragma omp barrier` construct.
- CUDA provides the instruction `__syncthreads()`.

A barrier has two phases. The arrival or trapping phase is followed by the departure or release phase. The manager maintains a counter: only when all workers have sent to the manager, does the manager send messages to all workers. Pseudo code for a linear barrier in a manager/worker model is shown below.

```
code for manager                code for worker

for i from 1 to p-1 do
    receive from i              send to manager
for i from 1 to p-1 do
    send to i                   receive from manager
```

The counter implementation of a barrier or linear barrier is effective but it takes $O(p)$ steps. A schematic of the steps to synchronize 8 processes is shown in Fig. 6.18 for a linear and a tree barrier.

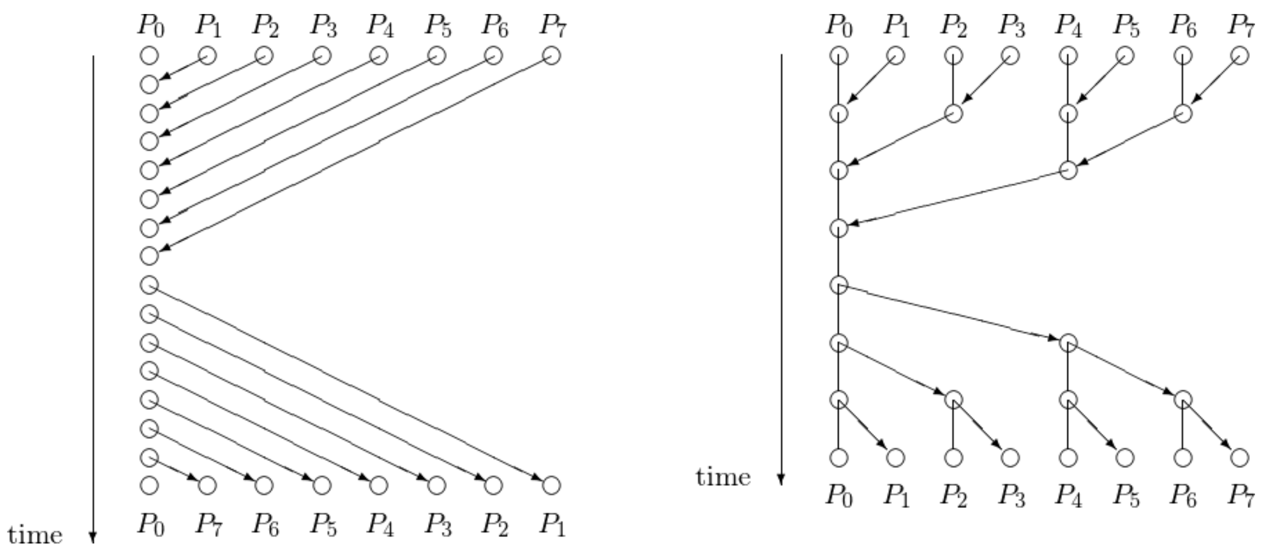

P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
P_0 P_7 P_6 P_5 P_4 P_3 P_2 P_1
time
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
time
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7

Fig. 6.18: A linear next to a tree barrier to synchronize 8 processes. For 8 processes, the linear barrier takes twice as many time steps as the tree barrier.

Implementing a tree barrier we write pseudo code for the trapping and the release phase, for $p = 2^k$ (recall the fan in gather and the fan out scatter):

The *trapping phase* is defined below:

```
for i from k-1 down to 0 do
    for j from 2**i to 2**(i+1) do
        node j sends to node j - 2**i
        node j - 2**i receives from node j.
```

The *release phase* is defined below

```
for i from 0 to k-1 do
    for j from 0 to 2**i-1  do
        node j sends to j + 2**i
        node j + 2**i receives from node j.
```

Observe that two processes can synchronize in one step. We can generalize this into a tree barrier so there are no idle processes. This leads to a butterfly barrier shown in Fig. 6.19.

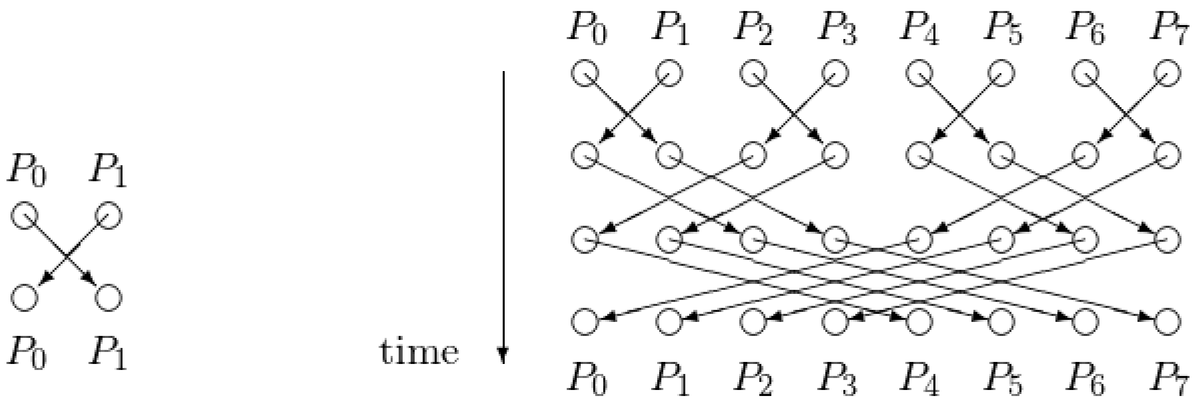


Fig. 6.19: Two processes can synchronize in one step as shown on the left. At the right is a schematic of the time steps for a tree barrier to synchronize 8 processes.

The algorithm for a butterfly barrier, for $p = 2^k$, is described is pseudo code below.

```
for i from 0 to k-1 do
    s := 0
    for j from 0 to p-1 do
        if (j mod 2**(i+1) = 0) s := j
        node j sends to node ((j + 2**i) mod 2**(i+1)) + s
        node ((j + 2**i) mod 2^(i+1)) + s receives from node j
```

To avoid deadlock, ensuring that every send is matched with a corresponding receive, we can work with a `sendrecv`, as shown in Fig. 6.20.

The `sendrecv` in MPI has the following form:

```
MPI_Sendrecv(sendbuf,sendcount,sendtype,dest,sendtag,
             recvbuf,recvcount,recvtype,source,recvtag,comm,status)
```

where the parameters are in Table 6.5.

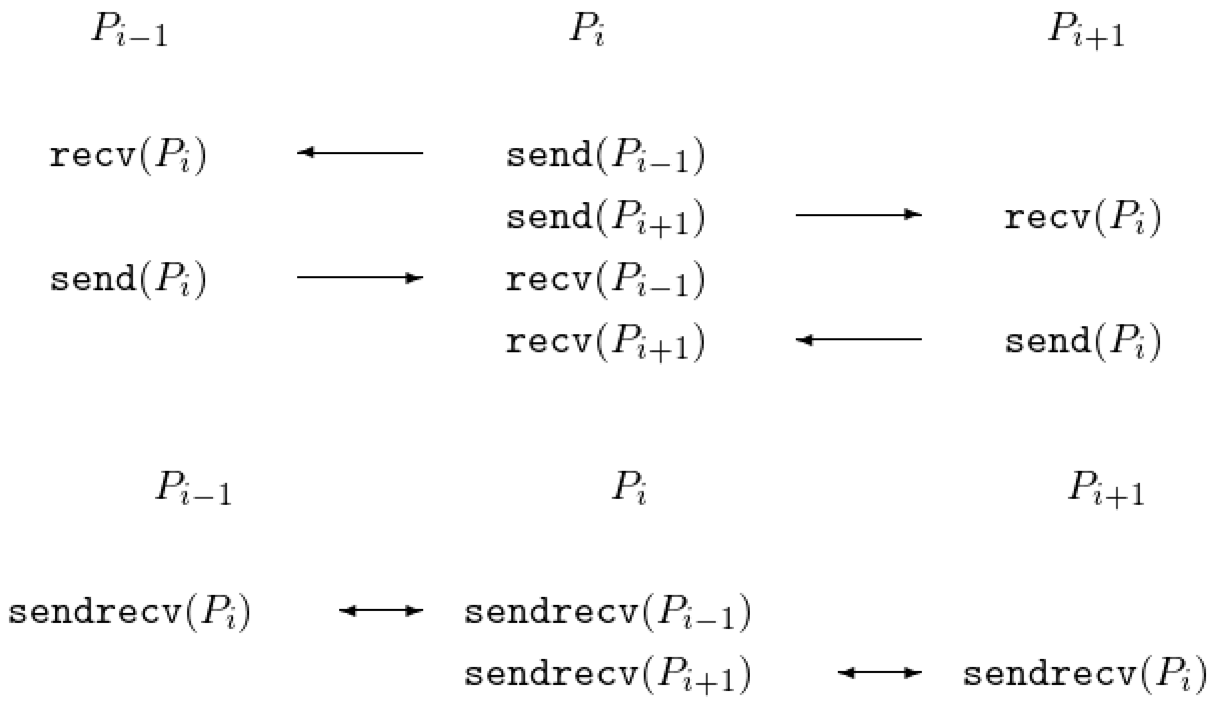


Fig. 6.20: The top picture is equivalent to the bottom picture.

Table 6.5: Parameters of sendrecv in MPI.

parameter	description
sendbuf	initial address of send buffer
sendcount	number of elements in send buffer
sendtype	type of elements in send buffer
dest	rank of destination
sendtag	send tag
recvbuf	initial address of receive buffer
recvcount	number of elements in receive buffer
sendtype	type of elements in receive buffer
source	rank of source or MPI_ANY_SOURCE
recvtag	receive tag or MPI_ANY_TAG
comm	communicator
status	status object

We illustrate `MPI_Sendrecv` to synchronize two nodes. Processors 0 and 1 swap characters in a bidirectional data transfer.

```
$ mpirun -np 2 /tmp/use_sendrecv
Node 0 will send a to 1
Node 0 received b from 1
Node 1 will send b to 0
Node 1 received a from 0
$
```

with code below:

```
#include <stdio.h>
#include <mpi.h>

#define sendtag 100

int main ( int argc, char *argv[] )
{
   int i,j;
   MPI_Status status;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&i);

   char c = 'a' + (char)i; /* send buffer */
   printf("Node %d will send %c to %d\n",i,c,j);
   char d;                 /* receive buffer */

   MPI_Sendrecv(&c,1,MPI_CHAR,j,sendtag,&d,1,MPI_CHAR,MPI_ANY_SOURCE,
                MPI_ANY_TAG,MPI_COMM_WORLD,&status);

   printf("Node %d received %c from %d\n",i,d,j);
}

MPI_Finalize();
return 0;
```

6.4.2 The Prefix Sum Algorithm

A data parallel computation is a computation where the *same* operations are preformed on *different* data *simultaneously*. The benefits of data parallel computations is that they are easy to program, scale well, and are fit for SIMD computers.

The problem we consider is to compute $\displaystyle\sum_{i=0}^{n-1} a_i$ for $n = p = 2^k$. This problem is related to the composite trapezoidal rule.

For $n = 8$ and $p = 8$, the prefix sum algorithm is illustrated in Fig. 6.21.

Pseudo code for the prefix sum algorithm for $n = p = 2^k$ is below. Processor i executes:

```
s := 1
x := a[i]
for j from 0 to k-1 do
    if (j < p - s + 1) send x to processor i+s
    if (j > s-1) receive y from processor i-s
                 add y to x: x := x + y
    s := 2*s
```

The speedup: $\dfrac{p}{\log_2(p)}$. Communication overhead: one send/recv in every step.

The prefix sum algorithm can be coded up in MPI as in the program below.

```
#include <stdio.h>
#include "mpi.h"
```

(continues on next page)

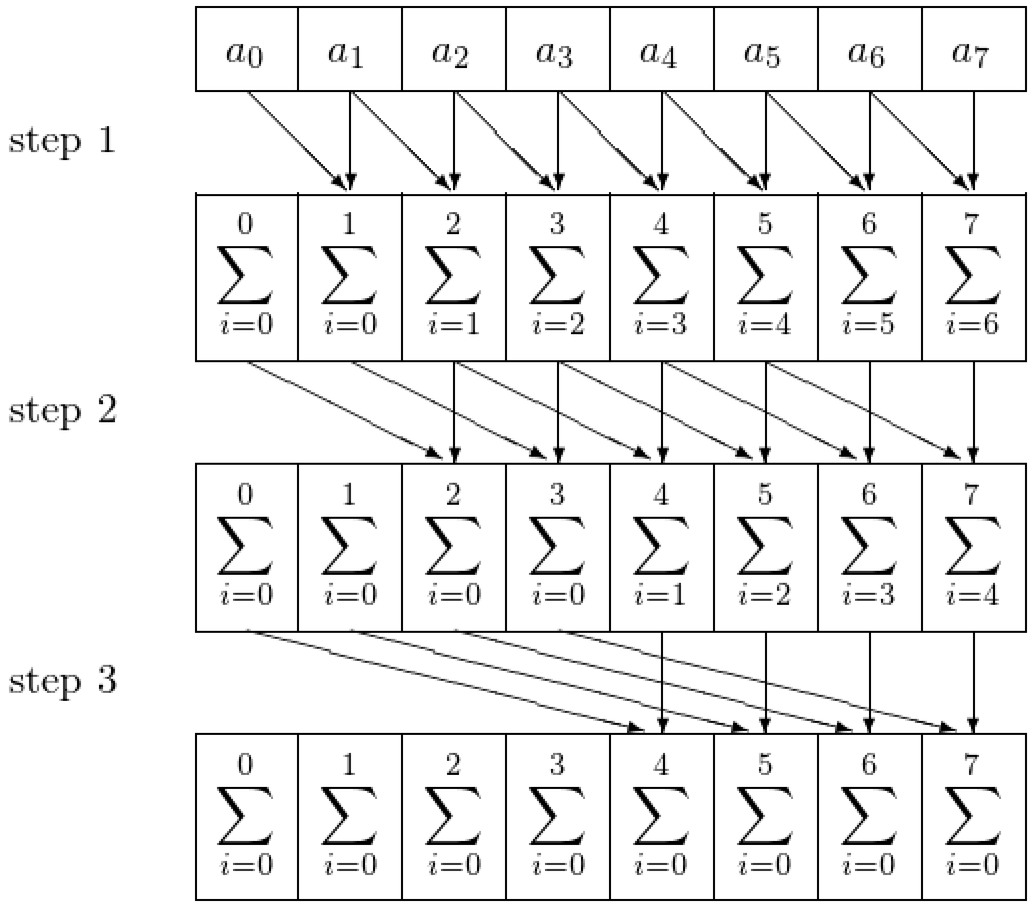

a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7
step 1
$\sum_{i=0}^{0}$ $\sum_{i=0}^{1}$ $\sum_{i=1}^{2}$ $\sum_{i=2}^{3}$ $\sum_{i=3}^{4}$ $\sum_{i=4}^{5}$ $\sum_{i=5}^{6}$ $\sum_{i=6}^{7}$
step 2
$\sum_{i=0}^{0}$ $\sum_{i=0}^{1}$ $\sum_{i=0}^{2}$ $\sum_{i=0}^{3}$ $\sum_{i=1}^{4}$ $\sum_{i=2}^{5}$ $\sum_{i=3}^{6}$ $\sum_{i=4}^{7}$
step 3
$\sum_{i=0}^{0}$ $\sum_{i=0}^{1}$ $\sum_{i=0}^{2}$ $\sum_{i=0}^{3}$ $\sum_{i=0}^{4}$ $\sum_{i=0}^{5}$ $\sum_{i=0}^{6}$ $\sum_{i=0}^{7}$

Fig. 6.21: The prefix sum for $n = 8 = p$.

(continued from previous page)

```
#define tag 100                /* tag for send/recv */

int main ( int argc, char *argv[] )
{
   int i,j,nb,b,s;
   MPI_Status status;
   const int p = 8;         /* run for 8 processors */

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&i);

   nb = i+1;             /* node i holds number i+1 */
   s = 1;     /* shift s will double in every step */

   for(j=0; j<3; j++)              /* 3 stages, as log2(8) = 3 */
   {
      if(i < p - s)     /* every one sends, except last s ones */
         MPI_Send(&nb,1,MPI_INT,i+s,tag,MPI_COMM_WORLD);
      if(i >= s)     /* every one receives, except first s ones */
      {
         MPI_Recv(&b,1,MPI_INT,i-s,tag,MPI_COMM_WORLD,&status);
         nb += b;       /* add received value to current number */
      }
      MPI_Barrier(MPI_COMM_WORLD);  /* synchronize computations */
      if(i < s)
         printf("At step %d, node %d has number %d.\n",j+1,i,nb);
      else
         printf("At step %d, Node %d has number %d = %d + %d.\n",
                j+1,i,nb,nb-b,b);
      s *= 2;                                /* double the shift */
   }
   if(i == p-1) printf("The total sum is %d.\n",nb);

   MPI_Finalize();
   return 0;
}
```

Running the code prints the following to screen:

```
$ mpirun -np 8 /tmp/prefix_sum
At step 1, node 0 has number 1.
At step 1, Node 1 has number 3 = 2 + 1.
At step 1, Node 2 has number 5 = 3 + 2.
At step 1, Node 3 has number 7 = 4 + 3.
At step 1, Node 7 has number 15 = 8 + 7.
At step 1, Node 4 has number 9 = 5 + 4.
At step 1, Node 5 has number 11 = 6 + 5.
At step 1, Node 6 has number 13 = 7 + 6.
At step 2, node 0 has number 1.
At step 2, node 1 has number 3.
At step 2, Node 2 has number 6 = 5 + 1.
At step 2, Node 3 has number 10 = 7 + 3.
```

(continues on next page)

(continued from previous page)

```
At step 2, Node 4 has number 14 = 9 + 5.
At step 2, Node 5 has number 18 = 11 + 7.
At step 2, Node 6 has number 22 = 13 + 9.
At step 2, Node 7 has number 26 = 15 + 11.
At step 3, node 0 has number 1.
At step 3, node 1 has number 3.
At step 3, node 2 has number 6.
At step 3, node 3 has number 10.
At step 3, Node 4 has number 15 = 14 + 1.
At step 3, Node 5 has number 21 = 18 + 3.
At step 3, Node 6 has number 28 = 22 + 6.
At step 3, Node 7 has number 36 = 26 + 10.
The total sum is 36.
```

6.4.3 Brent's Theorem

PRAM stands for Parallel Random Access Machine The PRAM model is an idealized construct.

- It assumes any number of processors can access any items in memory instantly.
- An operation takes one unit time.

The PRAM model helps to derive bounds on the theoretical time of a parallel algorithm.

Brent's theorem

Assume

1. a parallel computer where each processor can perform an arithmetic operation in unit time; and
2. the computer has exactly enough processors to exploit the maximum concurrency in an algorithm with N operations, such that T time steps suffice,

then a computer with P processors can perform the algorithm in time

$$T_P \leq T + \frac{N - T}{P},$$

where P is less than or equal to the number of processors needed to exploit the maximum concurrency in the algorithm.

As an application of Brent's theorem, we look at parallel summation. Consider the sum of n numbers.

If $n = 2^T$, then the PRAM can do the sum in T steps.

If the PRAM has P processors and $P \leq n/2$, then

$$T_P \leq \lceil \log_2(n) \rceil + \frac{(n-1) - \log_2(n)}{P},$$

where T_P is the execution time with P processors.

Typically, the number of processors is fixed, and then we want to find the best size n of the problem so the theoretical bounds on the execution time are within reach.

6.4.4 Bibliography

1. R. P. Brent: **The parallel evaluation of general arithmetic expressions.** *Journal of the ACM* 12(2): 201-206, 1974.
2. John Gustafson: **Brent's Theorem.** In *Enclopedia of Parallel Computing*, edited by David Padua, pages 182-185, Springer 2011.
3. W. Daniel Hillis and Guy L. Steele. *Data Parallel Algorithms*. **Communications of the ACM**, vol. 29, no. 12, pages 1170-1183, 1986.

6.4.5 Exercises

1. Does the hypercube topology support a butterfly barrier? If not, explain with an example. Otherwise, show that the hypercube topology has sufficiently many connections for a butterfly barrier.
2. Write code using `MPI_sendrecv` for a butterfly barrier. Show that your code works for $p = 8$.
3. Rewrite `prefix_sum.c` using `MPI_sendrecv`.
4. Consider the composite trapezoidal rule for the approximation of π (see lecture 13), doubling the number of intervals in each step. Can you apply the prefix sum algorithm so that at the end, processor i holds the approximation for π with 2^i intervals?

6.5 Parallel Iterative Methods for Linear Systems

We consider the method of Jacobi and introduce the `MPI_Allgather` command for the synchronization of the iterations. In the analysis of the communication and the computation cost, we determine the optimal value for the number of processors which minimizes the total cost.

6.5.1 Jacobi Iterations

We derive the formulas for Jacobi's method, starting from a fixed point formula. We want to solve $A\mathbf{x} = \mathbf{b}$ for A an n-by-n matrix, and $\mathbf{b}$ an n-dimensional vector, for **very large** n. Consider $A = L + D + U$, where

- $L = [\ell_{i,j}], \ell_{i,j} = a_{i,j}, i > j, \ell_{i,j} = 0, i \leq j$. L is lower triangular.
- $D = [d_{i,j}], d_{i,i} = a_{i,i} \neq 0, d_{i,j} = 0, i \neq j$. D is diagonal.
- $U = [u_{i,j}], u_{i,j} = a_{i,j}, i < j, u_{i,j} = 0, i \geq j$. U is upper triangular.

Then we rewrite $A\mathbf{x} = \mathbf{b}$ as

$$\begin{array}{rcl} A\mathbf{x} = \mathbf{b} & \Leftrightarrow & (L + D + U)\mathbf{x} = \mathbf{b} \\ & \Leftrightarrow & D\mathbf{x} = \mathbf{b} - L\mathbf{x} - U\mathbf{x} \\ & \Leftrightarrow & D\mathbf{x} = D\mathbf{x} + \mathbf{b} - L\mathbf{x} - U\mathbf{x} - D\mathbf{x} \\ & \Leftrightarrow & D\mathbf{x} = D\mathbf{x} + \mathbf{b} - A\mathbf{x} \\ & \Leftrightarrow & \mathbf{x} = \mathbf{x} + D^{-1}(\mathbf{b} - A\mathbf{x}). \end{array}$$

The fixed point formula $\mathbf{x} = \mathbf{x} + D^{-1}(\mathbf{b} - A\mathbf{x})$ is well defined if $a_{i,i} \neq 0$. The fixed point formula $\mathbf{x} = \mathbf{x} + D^{-1}(\mathbf{b} - A\mathbf{x})$ leads to

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \underbrace{D^{-1}\left(\mathbf{b} - A\mathbf{x}^{(k)}\right)}_{\Delta\mathbf{x}}, \quad k = 0, 1, \ldots$$

Writing the formula as an algorithm:

```
Input: A, b, x(0), eps, N.
Output: x(k), k is the number of iterations done.

for k from 1 to N do
    dx := D**(-1) ( b - A x(k) )
    x(k+1) := x(k) + dx
    exit when (norm(dx) <= eps)
```

Counting the number of operations in the algorithm above, we have a cost of $O(Nn^2)$, $O(n^2)$ for $A\mathbf{x}^{(k)}$, if A is dense.

Convergence of the Jacobi method

The Jacobi method converges for strictly row-wise or column-wise diagonally dominant matrices, i.e.: if

$$|a_{i,i}| > \sum_{j\neq i} |a_{i,j}| \quad \text{or} \quad |a_{i,i}| > \sum_{j\neq i} |a_{j,i}|, \quad i = 1, 2, \ldots, n.$$

To run the code above with p processors:

- The n rows of A are distributed evenly (e.g.: $p = 4$):

$$D \star \left[\begin{array}{c} \Delta\mathbf{x}^{[0]} \\ \hline \Delta\mathbf{x}^{[1]} \\ \hline \Delta\mathbf{x}^{[2]} \\ \hline \Delta\mathbf{x}^{[3]} \end{array}\right] = \left[\begin{array}{c} \mathbf{b}^{[0]} \\ \hline \mathbf{b}^{[1]} \\ \hline \mathbf{b}^{[2]} \\ \hline \mathbf{b}^{[3]} \end{array}\right] - \left[\begin{array}{cccc} A^{[0,0]} & A^{[0,1]} & A^{[0,2]} & A^{[0,3]} \\ \hline A^{[1,0]} & A^{[1,1]} & A^{[1,2]} & A^{[1,3]} \\ \hline A^{[2,0]} & A^{[2,1]} & A^{[2,2]} & A^{[2,3]} \\ \hline A^{[3,0]} & A^{[3,1]} & A^{[3,2]} & A^{[3,3]} \end{array}\right] \star \left[\begin{array}{c} \mathbf{x}^{[0],(k)} \\ \hline \mathbf{x}^{[1],(k)} \\ \hline \mathbf{x}^{[2],(k)} \\ \hline \mathbf{x}^{[3],(k)} \end{array}\right]$$

- Synchronization is needed for $(|| Delta {bf x} || leq epsilon)$.

For $||\cdot||$, use $||\Delta\mathbf{x}||_1 = |\Delta x_1| + |\Delta x_2| + \cdots + |\Delta x_n|$, the butterfly synchronizations are displayed in Fig. 6.22.

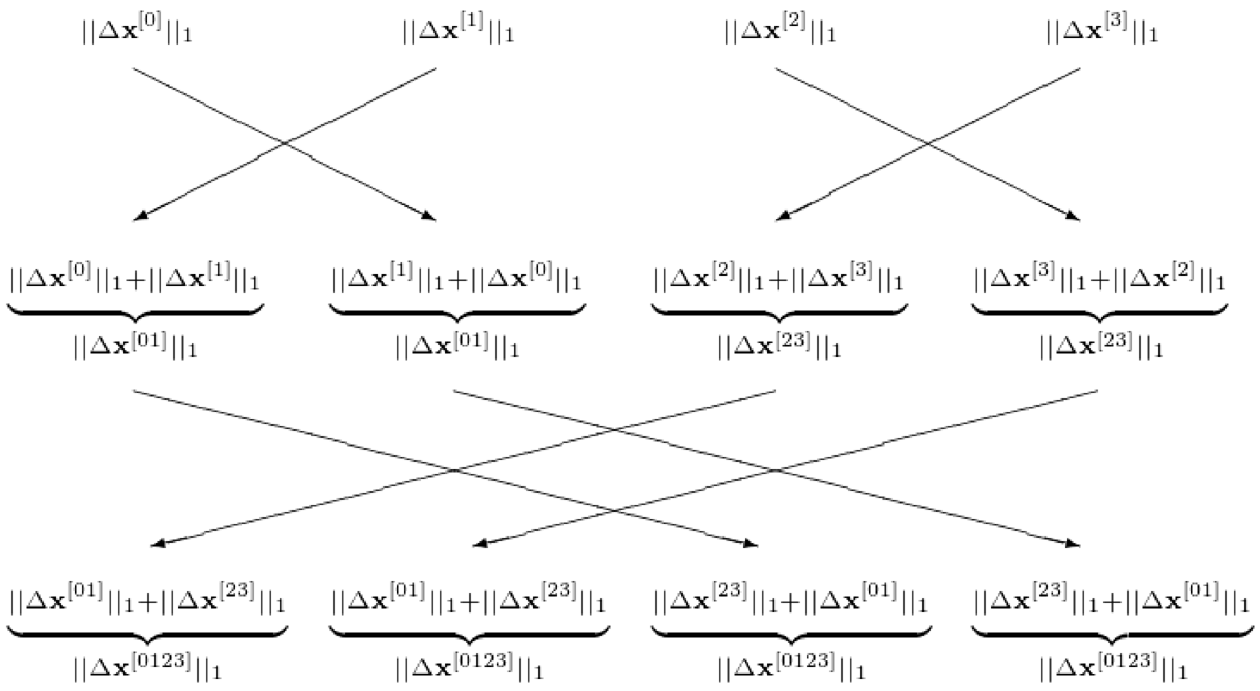

$||\Delta\mathbf{x}^{[0]}||_1$
$||\Delta\mathbf{x}^{[1]}||_1$
$||\Delta\mathbf{x}^{[2]}||_1$
$||\Delta\mathbf{x}^{[3]}||_1$
$\underbrace{||\Delta\mathbf{x}^{[0]}||_1+||\Delta\mathbf{x}^{[1]}||_1}_{||\Delta\mathbf{x}^{[01]}||_1}$
$\underbrace{||\Delta\mathbf{x}^{[1]}||_1+||\Delta\mathbf{x}^{[0]}||_1}_{||\Delta\mathbf{x}^{[01]}||_1}$
$\underbrace{||\Delta\mathbf{x}^{[2]}||_1+||\Delta\mathbf{x}^{[3]}||_1}_{||\Delta\mathbf{x}^{[23]}||_1}$
$\underbrace{||\Delta\mathbf{x}^{[3]}||_1+||\Delta\mathbf{x}^{[2]}||_1}_{||\Delta\mathbf{x}^{[23]}||_1}$
$\underbrace{||\Delta\mathbf{x}^{[01]}||_1+||\Delta\mathbf{x}^{[23]}||_1}_{||\Delta\mathbf{x}^{[0123]}||_1}$
$\underbrace{||\Delta\mathbf{x}^{[01]}||_1+||\Delta\mathbf{x}^{[23]}||_1}_{||\Delta\mathbf{x}^{[0123]}||_1}$
$\underbrace{||\Delta\mathbf{x}^{[23]}||_1+||\Delta\mathbf{x}^{[01]}||_1}_{||\Delta\mathbf{x}^{[0123]}||_1}$
$\underbrace{||\Delta\mathbf{x}^{[23]}||_1+||\Delta\mathbf{x}^{[01]}||_1}_{||\Delta\mathbf{x}^{[0123]}||_1}$

Fig. 6.22: Butterfly synchronization of a parallel Jacobi iteration with 4 processors.

The communication stages are as follows. At the start, every node must have $\mathbf{x}^{(0)}$, ϵ, N, a number of rows of A and the corresponding part of the right hand side $\mathbf{b}$. After each update n/p elements of $\mathbf{x}^{(k+1)}$ must be scattered. The butterfly synchronization takes $\log_2(p)$ steps. The scattering of $\mathbf{x}^{(k+1)}$ can coincide with the butterfly synchronization. The computation effort: $O(n^2/p)$ in each stage.

6.5.2 A Parallel Implementation with MPI

For dimension n, we consider the diagonally dominant system:

$$\left[\begin{array}{cccc} n+1 & 1 & \cdots & 1 \\ 1 & n+1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & n+1 \end{array} \right] \left[\begin{array}{c} x_1 \\ x_2 \\ \vdots \\ x_n \end{array} \right] = \left[\begin{array}{c} 2n \\ 2n \\ \vdots \\ 2n \end{array} \right].$$

The exact solution is $\mathbf{x}$: for $i = 1, 2, \ldots, n$, $x_i = 1$. We start the Jacobi iteration method at $\mathbf{x}^{(0)} = \mathbf{0}$. The parameters are $\epsilon = 10^{-4}$ and $N = 2n^2$. A session where we run the program displays on screen the following:

```
$ time /tmp/jacobi 1000
   0 : 1.998e+03
   1 : 1.994e+03
...
8405 : 1.000e-04
8406 : 9.982e-05
computed 8407 iterations
error : 4.986e-05

real    0m42.411s
user    0m42.377s
sys     0m0.028s
```

C code to run Jacobi iterations is below.

```
void run_jacobi_method
 ( int n, double **A, double *b, double epsilon, int maxit, int *numit, double *x );
/*
 * Runs the Jacobi method for A*x = b.
 *
 * ON ENTRY :
 *   n        the dimension of the system;
 *   A        an n-by-n matrix A[i][i] /= 0;
 *   b        an n-dimensional vector;
 *   epsilon  accuracy requirement;
 *   maxit    maximal number of iterations;
 *   x        start vector for the iteration.
 *
 * ON RETURN :
 *   numit    number of iterations used;
 *   x        approximate solution to A*x = b. */
```

```
void run_jacobi_method
 ( int n, double **A, double *b, double epsilon, int maxit, int *numit, double *x )
{
```

(continues on next page)

(continued from previous page)

```
   double *dx,*y;
   dx = (double*) calloc(n,sizeof(double));
   y = (double*) calloc(n,sizeof(double));
   int i,j,k;

   for(k=0; k<maxit; k++)
   {
      double sum = 0.0;
      for(i=0; i<n; i++)
      {
         dx[i] = b[i];
         for(j=0; j<n; j++)
            dx[i] -= A[i][j]*x[j];
         dx[i] /= A[i][i];
         y[i] += dx[i];
         sum += ( (dx[i] >= 0.0) ? dx[i] : -dx[i]);
      }
      for(i=0; i<n; i++) x[i] = y[i];
      printf("%3d : %.3e\n",k,sum);
      if(sum <= epsilon) break;
   }
   *numit = k+1;
   free(dx); free(y);
}
```

6.5.3 Gather-to-All with MPI_Allgather

Gathering the four elements of a vector to four processors is schematically depicted in Fig. 6.23.

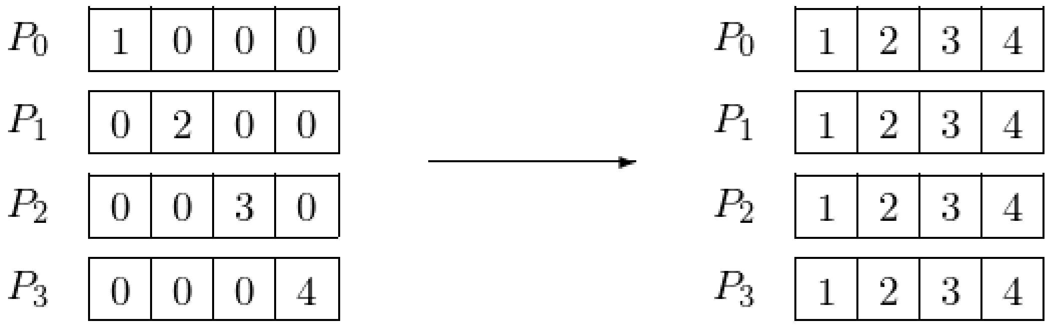


Fig. 6.23: Gathering 4 elements to 4 processors.

The syntax of the MPI gather-to-all command is

```
MPI_Allgather(sendbuf,sendcount,sendtype,
              recvbuf,recvcount,recvtype,comm)
```

where the parameters are in Table 6.6.

Table 6.6: The parameters of MPI_Allgather.

parameter	description
sendbuf	starting address of send buffer
sendcount	number of elements in send buffer
sendtype	data type of send buffer elements
recvbuf	address of receive buffer
recvcount	number of elements received from any process
recvtype	data type of receive buffer elements
comm	communicator

A program that implements the situation as in Fig. 6.23 will print the following to screen:

```
$ mpirun -np 4 /tmp/use_allgather
data at node 0 : 1 0 0 0
data at node 1 : 0 2 0 0
data at node 2 : 0 0 3 0
data at node 3 : 0 0 0 4
data at node 3 : 1 2 3 4
data at node 0 : 1 2 3 4
data at node 1 : 1 2 3 4
data at node 2 : 1 2 3 4
$
```

The code of the program `use_allgather.c` is below:

```
int i,j,p;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&i);
MPI_Comm_size(MPI_COMM_WORLD,&p);
{
   int data[p];
   for(j=0; j<p; j++) data[j] = 0;
   data[i] = i + 1;
   printf("data at node %d :",i);
   for(j=0; j<p; j++) printf(" %d",data[j]); printf("\n");
   MPI_Allgather(&data[i],1,MPI_INT,data,1,MPI_INT,MPI_COMM_WORLD);
   printf("data at node %d :",i);
   for(j=0; j<p; j++) printf(" %d",data[j]); printf("\n");
}
```

Applying the `MPI_Allgather` to a parallel version of the Jacobi method shows the following on screen:

```
$ time mpirun -np 10 /tmp/jacobi_mpi 1000
...
8405 : 1.000e-04
8406 : 9.982e-05
computed 8407 iterations
error : 4.986e-05

real    0m5.617s
user    0m45.711s
sys     0m0.883s
```

Recall that the wall clock time of the run with the sequential program equals `42.411.s`. The speedup is thus `42.411/5.617 = 7.550`. Code for the parallel `run_jacobi_method` is below.

```
void run_jacobi_method
 ( int id, int p, int n, double **A, double *b, double epsilon, int maxit, int *numit,␣
↪double *x )
{
   double *dx,*y;
   dx = (double*) calloc(n,sizeof(double));
   y = (double*) calloc(n,sizeof(double));
   int i,j,k;
   double sum[p];
   double total;
   int dnp = n/p;
   int istart = id*dnp;
   int istop = istart + dnp;
   for(k=0; k<maxit; k++)
   {
      sum[id] = 0.0;
      for(i=istart; i<istop; i++)
      {
         dx[i] = b[i];
         for(j=0; j<n; j++)
            dx[i] -= A[i][j]*x[j];
         dx[i] /= A[i][i];
         y[i] += dx[i];
         sum[id] += ( (dx[i] >= 0.0) ? dx[i] : -dx[i]);
      }
      for(i=istart; i<istop; i++) x[i] = y[i];
      MPI_Allgather(&x[istart],dnp,MPI_DOUBLE,x,dnp,MPI_DOUBLE,MPI_COMM_WORLD);
      MPI_Allgather(&sum[id],1,MPI_DOUBLE,sum,1,MPI_DOUBLE,MPI_COMM_WORLD);
      total = 0.0;
      for(i=0; i<p; i++) total += sum[i];
      if(id == 0) printf("%3d : %.3e\n",k,total);
      if(total <= epsilon) break;
   }
   *numit = k+1;
   free(dx);
}
```

Let us do an analysis of the computation and communication cost. Computing $\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} + D^{-1}(\mathbf{b} - A\mathbf{x}^{(k)})$ with p processors costs

$$t_{\rm comp} = \frac{n(2n+3)}{p}.$$

We count $2n+3$ operations because of

- one $-$ and one $\star$ when running over the columns of A; and
- one $/$, one $+$ for the update and one $+$ for the $|| \cdot ||_1$.

The communication cost is

$$t_{\rm comm} = p\left(t_{\rm startup} + \frac{n}{p} t_{\rm data}\right).$$

In the examples, the time unit is the cost of one arithmetical operation. Then the costs $t_{\rm startup}$ and $t_{\rm data}$ are multiples of this unit.

Finding the p with the minimum total cost is illustrated in Fig. 6.24 and Fig. 6.25.

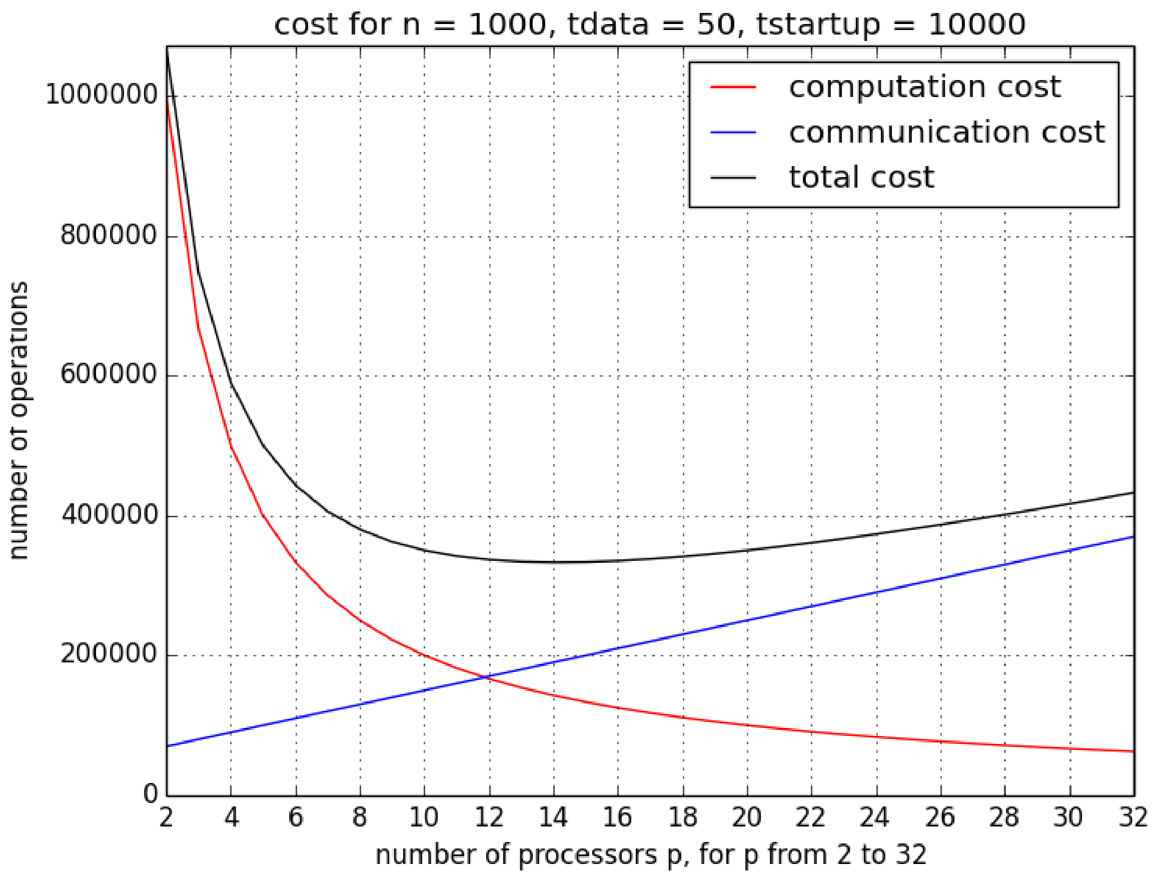


Fig. 6.24: With increasing p, the (red) computation cost decreases, while the (blue) communication cost increases. The minimum of the (black) total cost is the optimal value for p.

In Fig. 6.24, the communication, computation, and total cost is shown for p ranging from 2 to 32, for one iteration, with $n = 1,000$, $t_{\rm startup} = 10,000$, and $t_{\rm data} = 50$. We see that the total cost starts to increase once p becomes larger than 16. For a larger dimension, after a ten-fold increase, $n = 10,000$, $t_{\rm startup} = 10,000$, and $t_{\rm data} = 50$, the scalability improves, as in Fig. 6.25, p ranges from 16 to 256.

6.5.4 Strip Partitioning and Reduce Barriers in Julia

If the dimension of the matrix is a multiple of the number of threads, for some matrix `A` and vectors `x`, `y`:

```
nt = nthreads()
size = 10
dim = nt*size

@threads for i=1:nt
    tdx = threadid()
```

(continues on next page)

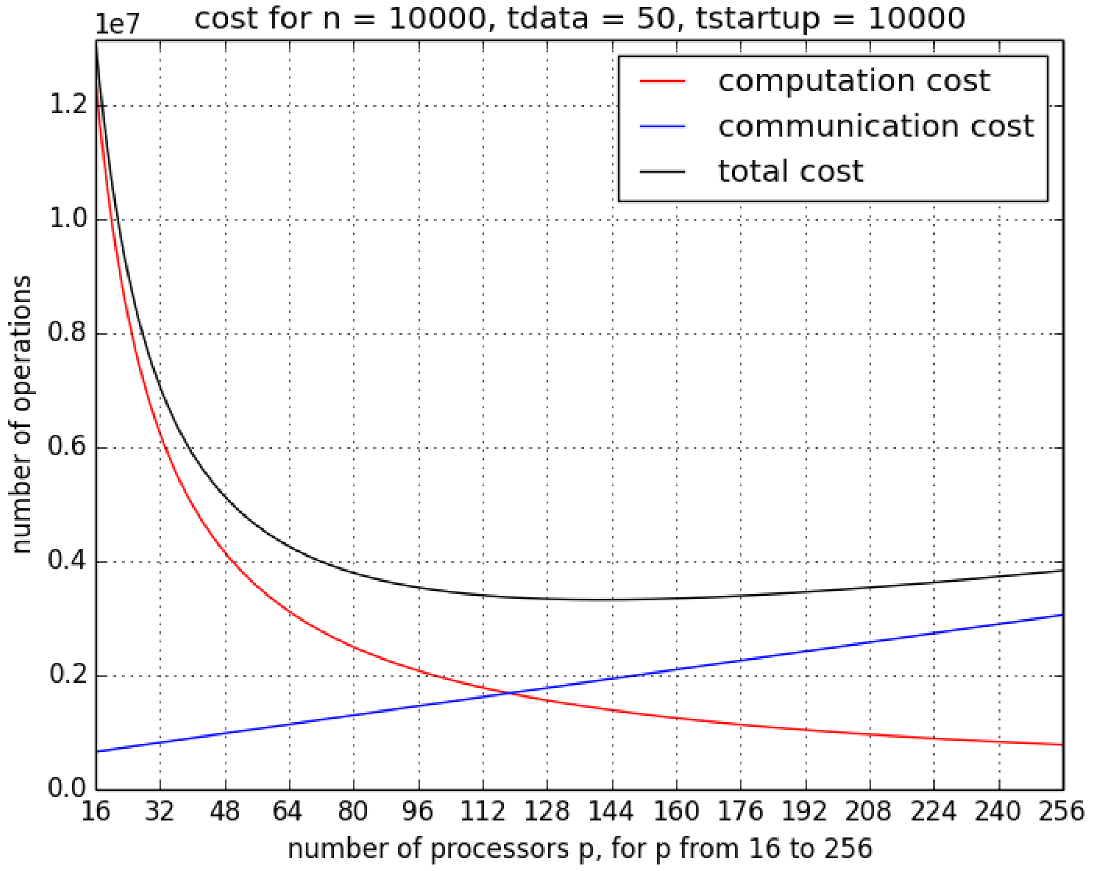


Fig. 6.25: With increasing p, the (red) computation cost decreases, while the (blue) communication cost increases. The minimum of the (black) total cost is the optimal value for p.

(continued from previous page)

```
    idxstart = 1 + (tdx-1)*size
    idxend = tdx*size
    @inbounds y[idxstart:idxend] = A[idxstart:idxend, :]*x
end
```

Reduce barriers are provided by the package `SyncBarriers` in Julia, and can be applied as illustrated below.

```
using Base.Threads
using SyncBarriers

nt = nthreads()
nb = [k for k=1:nt]
barrier = reduce_barrier(+, Int, nt)
s = 0
@threads for i=1:nt
    tdx = threadid()
    global s = reduce!(barrier[tdx], nb[tdx])
end
println("The sum of ", nb, " is ", s, ".")
```

The output of `julia -t 4 mtreduce.jl` is

```
The sum of [1, 2, 3, 4] is 10.
```

In a multithreaded Jacobi method, with p threads:

1. The i-th thread
    1. computes the i-th strip of the update $\Delta \mathbf{x}_i$,
    2. updates the i-th strip of $\mathbf{x}_i$ with $\Delta \mathbf{x}_i$,
    3. computes the norm of the i-th update $\|\Delta \mathbf{x}_i\|$.
2. Given $(\|\Delta \mathbf{x}_1\|, \|\Delta \mathbf{x}_2\|, \ldots, \|\Delta \mathbf{x}_p\|)$, a reduce barrier computes

$$\|\Delta \mathbf{x}\|_1 = \|\Delta \mathbf{x}_1\| + \|\Delta \mathbf{x}_2\| + \cdots + \|\Delta \mathbf{x}_p\|$$

and that $\|\Delta \mathbf{x}\|_1$ is used by every thread.

The full Julia program `mtjacobi.jl` is posted at the course web site.

The output of three runs on pascal are below.

```
time julia -t 2 mtjacobi.jl 8000
number of iterations : 40
the error : 1.9681077347290746e-5

real    0m15.390s
user    11m35.441s
sys     4m51.916s
$ time julia -t 4 mtjacobi.jl 8000
number of iterations : 20
the error : 2.3621495916454325e-5

real    0m5.400s
```

(continues on next page)

(continued from previous page)

```
user    2m13.138s
sys     1m18.059s
$ time julia -t 8 mtjacobi.jl 8000
number of iterations : 39
the error : 1.7918058060438385e-5

real    0m5.400s
user    2m10.425s
sys     1m13.413s
```

We covered section 6.3.1 in the book of Wilkinson and Allen. *Because of its slow convergence, the Jacobi method is seldomly used.*

6.5.5 Exercises

1. Use mpi4py or MPI.jl for the parallel Jacobi method. Compare with the C version to demonstrate the correctness.
2. Use OpenMP to write a parallel version of the Jacobi method. Do you observe a better speedup than with MPI?
3. The power method to compute the largest eigenvalue of a matrix A uses the formulas

$$\mathbf{y} := A\mathbf{x}^{(k)} \quad \text{and} \quad \mathbf{x}^{(k+1)} := \mathbf{y}/||\mathbf{y}||.$$

   Describe a parallel implementation of the power method.
4. Consider the formula for the total cost of the Jacobi method for an n-dimensional linear system with p processors. Derive an analytic expression for the optimal value of p. What does this expression tell about the scalability?

6.6 Domain Decomposition Methods

THe method of Jacobi is an interative method which is not in place: we do not overwrite the current solution with new components as soon as these become available. In contrast, the method of Gauss-Seidel does update the current solution with newly computed components of the solution as soon as these are computed.

Domain decomposition methods to solve partial differential equations are another important class of synchronized parallel computations, explaining the origin for the need to solve large linear systems. This chapter ends with an introduction to the software PETSc, the Portable, Extensible Toolkit for Scientific Computation.

6.6.1 Gauss-Seidel Relaxation

The method of Gauss-Seidel is an iterative method for solving linear systems. We want to solve $A\mathbf{x} = \mathbf{b}$ for a very large dimension n. Writing the method of Jacobi componentwise:

$$x_i^{(k+1)} := x_i^{(k)} + \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{n} a_{i,j} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n$$

We observe that we can already use $x_j^{(k+1)}$ for $j < i$. This leads to the following formulas

$$x_i^{(k+1)} := x_i^{(k)} + \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(k+1)} - \sum_{j=i}^{n} a_{i,j} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n.$$

The method of Gauss-Seidel is an *in-place method*: old values are overwritten by new ones as soon as computed. In translating the formulas, we end up with two loops:

$$\begin{array}{l} \text{for } j \text{ from } 1 \text{ to } i-1 \text{ do } \Delta x_i := \Delta x_i - a_{i,j} x_j^{(k+1)} \\ \text{for } j \text{ from } i \text{ to } n \text{ do } \Delta x_i := \Delta x_i - a_{i,j} x_j^{(k)} \end{array}$$

The two loops are fused into one loop as done below:

$$\text{for } j \text{ from } 1 \text{ to } n \text{ do } \Delta x_i := \Delta x_i - a_{i,j} x_j$$

C code for the Gauss-Seidel method is below.

```
void run_gauss_seidel_method
 ( int n, double **A, double *b, double epsilon, int maxit, int *numit, double *x )
/*
 * Runs the  method of Gauss-Seidel for A*x = b.
 *
 * ON ENTRY :
 *   n        the dimension of the system;
 *   A        an n-by-n matrix A[i][i] /= 0;
 *   b        an n-dimensional vector;
 *   epsilon  accuracy requirement;
 *   maxit    maximal number of iterations;
 *   x        start vector for the iteration.
 *
 * ON RETURN :
 *   numit    number of iterations used;
 *   x        approximate solution to A*x = b. */
{
   double *dx = (double*) calloc(n,sizeof(double));
   int i,j,k;
   for(k=0; k<maxit; k++)
   {
      double sum = 0.0;
      for(i=0; i<n; i++)
      {
         dx[i] = b[i];
         for(j=0; j<n; j++)
            dx[i] -= A[i][j]*x[j];
         dx[i] /= A[i][i]; x[i] += dx[i];
         sum += ( (dx[i] >= 0.0) ? dx[i] : -dx[i]);
      }
      printf("%4d : %.3e\n",k,sum);
      if(sum <= epsilon) break;
   }
   *numit = k+1; free(dx);
}
```

Running on the same example as in the previous chapter goes much faster:

```
$ time /tmp/gauss_seidel 1000
   0 : 1.264e+03
   1 : 3.831e+02
   2 : 6.379e+01
```

(continues on next page)

(continued from previous page)

```
   3 : 1.394e+01
   4 : 3.109e+00
   5 : 5.800e-01
   6 : 1.524e-01
   7 : 2.521e-02
   8 : 7.344e-03
   9 : 1.146e-03
  10 : 3.465e-04
  11 : 5.419e-05
computed 12 iterations     <----- 8407 with Jacobi
error : 1.477e-05

real    0m0.069s           <----- 0m42.411s
user    0m0.063s           <----- 0m42.377s
sys     0m0.005s           <----- 0m0.028s
```

6.6.2 Parallel Gauss-Seidel with OpenMP

The method of Jacobi is suitable for strip partitioning of the (dense) matrix and in a parallel distributed memory implementation, every processor can keep its own portion of the solution vector $\mathbf{x}$. The Gauss-Seidel method makes the new x_i directly available which leads to communication overhead on distributed memory computers.

In a parallel shared memory implementation, consider:

1. Threads compute inner products of matrix rows with $\mathbf{x}$.
2. Each Δx_i is updated in a critical section.

So, many threads compute one inner product. For example, three threads, assuming n is divisible by 3, compute:

$$\left[\begin{array}{c|c|c} a_{i,1} \cdots a_{i,n/3} & a_{i,n/3+1} \cdots a_{i,2n/3} & a_{i,2n/3+1} \cdots a_{i,n} \end{array}\right] \left[\begin{array}{c} x_1 \\ \vdots \\ x_{n/3} \\ \hline x_{n/3+1} \\ \vdots \\ x_{2n/3} \\ \hline x_{2n/3+1} \\ \vdots \\ x_n \end{array}\right]$$

Each thread has its own variable to accumulate its portion of the inner product.

Using p threads:

```
void run_gauss_seidel_method
 ( int p, int n, double **A, double *b, double epsilon, int maxit, int *numit, double *x␣
↪)
{
   double *dx;
   dx = (double*) calloc(n,sizeof(double));
   int i,j,k,id,jstart,jstop;
```

(continues on next page)

(continued from previous page)

```
   int dnp = n/p;
   double dxi;

   for(k=0; k<maxit; k++)
   {
      double sum = 0.0;
      for(i=0; i<n; i++)
      {
         dx[i] = b[i];
         #pragma omp parallel shared(A,x) private(id,j,jstart,jstop,dxi)
         {
            id = omp_get_thread_num();
            jstart = id*dnp;
            jstop = jstart + dnp;
            dxi = 0.0;
            for(j=jstart; j<jstop; j++)
               dxi += A[i][j]*x[j];
            #pragma omp critical
               dx[i] -= dxi;
         }
         dx[i] /= A[i][i];
         x[i] += dx[i];
         sum += ( (dx[i] >= 0.0) ? dx[i] : -dx[i]);
      }
      printf("%4d : %.3e\n",k,sum);
      if(sum <= epsilon) break;
   }
   *numit = k+1;
   free(dx);
}
```

The update instructions

```
dx[i] /= A[i][i];
x[i] += dx[i];
sum += ( (dx[i] >= 0.0) ? dx[i] : -dx[i]);
```

are executed after each parallel region. This ensures the synchronization and the execution of the stop test:

```
if(sum <= epsilon) break;
```

Observe that although the entire matrix `A` is shared between all threads, each threads needs only `n/p` columns of the matrix. In the MPI version of the method of Jacobi, entire rows of the matrix were distributed among the processors. If we were to make a distributed memory version of the OpenMP code, then we would distribute entire columns of the matrix `A` over the processors.

Running times obtained via the command `time` on a 12-core Intel X5690 at 3.47 GHz, are in Table 6.7.

Table 6.7: Times of a parallel Gauss-Seidel with OpenMP

p	*n*	real	user	sys	speedup
1	10,000	7.165s	6.921s	0.242s	
	20,000	28.978s	27.914s	1.060s	
	30,000	1m 6.491s	1m 4.139s	2.341s	
2	10,000	4.243s	7.621s	0.310s	1.689
	20,000	16.325s	29.556s	1.066s	1.775
	30,000	36.847s	1m 6.831s	2.324s	1.805
5	10,000	2.415s	9.440s	0.420s	2.967
	20,000	8.403s	32.730s	1.218s	3.449
	30,000	18.240s	1m 11.031s	2.327s	3.645
10	10,000	2.173s	16.241s	0.501s	3.297
	20,000	6.524s	45.629s	1.521s	4.442
	30,000	13.273s	1m 29.687s	2.849s	5.010

6.6.3 Solving the Heat Equation

We will be applying a time stepping method to the heat or diffusion equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{\partial u}{\partial t}$$

models the temperature distribution $u(x, y, t)$ evolving in time t for (x, y) in some domain.

Related Partial Differential Equations (PDEs) are

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad \text{and} \quad \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y),$$

respectively called the Laplace and Poisson equations.

For the discretization of the derivatives, consider that at a point (x_0, y_0, t_0), we have

$$\left.\frac{\partial u}{\partial x}\right|_{(x_0,y_0,t_0)} = \lim_{h \rightarrow 0} \underbrace{\frac{u(x_0 + h, y_0, t_0) - u(x_0, y_0, h)}{h}}_{u_x(x_0,y_0,t_0)}$$

so for positive $h \approx 0$, $u_x(x_0, y_0, t_0) \approx \left.\frac{\partial u}{\partial x}\right|_{(x_0,y_0,t_0)}$.

For the second derivative we use the finite difference $u_{xx}(x_0, y_0, t_0)$

$$\begin{aligned} &= \frac{1}{h}\left(\frac{u(x_0 + h, y_0, t_0) - u(x_0, y_0, t_0)}{h} - \frac{u(x_0, y_0, t_0) - u(x_0 - h, y_0, t_0)}{h}\right) \\ &= \frac{u(x_0 + h, y_0, t_0) - 2u(x_0, y_0, t_0) + u(x_0 - h, y_0, t_0)}{h^2}. \end{aligned}$$

Time stepping is then done along the formulas:`

$$\begin{aligned} u_t(x_0, y_0, t_0) &= \frac{u(x_0, y_0, t_0 + h) - u(x_0, y_0, t_0)}{h} \\ u_{xx}(x_0, y_0, t_0) &= \frac{u(x_0 + h, y_0, t_0) - 2u(x_0, y_0, t_0) + u(x_0 - h, y_0, t_0)}{h^2} \\ u_{yy}(x_0, y_0, t_0) &= \frac{u(x_0, y_0 + h, t_0) - 2u(x_0, y_0, t_0) + u(x_0, y_0 - h, t_0)}{h^2} \end{aligned}$$

Then the equation $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$ becomes

$$\begin{array}{rcl} u(x_0, y_0, t_0 + h) & = & u(x_0, y_0, t_0) \\ & + & \frac{1}{h} \left[\, u(x_0 + h, y_0, t_0) + u(x_0 - h, y_0, t_0) \right. \\ & + & \left. \quad u(x_0, y_0 + h, t_0) + u(x_0, y_0 - h, t_0) - 4u(x_0, y_0, t_0) \, \right] \end{array}$$

Locally, the error of this approximation is $O(h^2)$.

The algorithm performs synchronous iterations on a grid. For $(x, y) \in [0, 1] \times [0, 1]$, the division of $[0, 1]$ in n equal subintervals, with $h = 1/n$, leads to a grid $(x_i = ih, y_j = jh)$, for $i = 0, 1, \ldots, n$ and $j = 0, 1, \ldots, n$. For t, we use the same step size h: $t_k = kh$. Denote $u_{i,j}^{(k)} = u(x_i, y_j, t_k)$, then

$$u_{i,j}^{(k+1)} = u_{i,j}^{(k)} + \frac{1}{h} \left[u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)} - 4u_{i,j}^{(k)} \right].$$

Fig. 6.26 shows the labeling of the grid points.

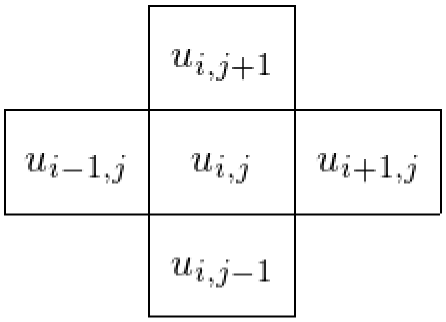


Fig. 6.26: In every step, we update $u_{i,j}$ based on $u_{i-1,j}$, $u_{i+1,j}$, $u_{i,j-1}$, and $u_{i,j+1}$.

We divide the grid in red and black points, as in Fig. 6.27.

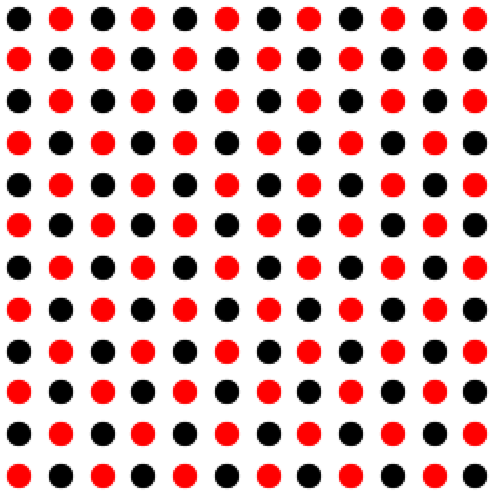


Fig. 6.27: Organization of the grid $u_{i,j}$ in red and black points.

The computation is organized in two phases:

1. In the first phase, update all black points simultaneously; and then
2. in the second phase, update all red points simultaneously.

We can decompose a domain in strips, but then there are n/p boundaries that must be shared. To reduce the overlapping, we partition in squares, as shown in Fig. 6.28.

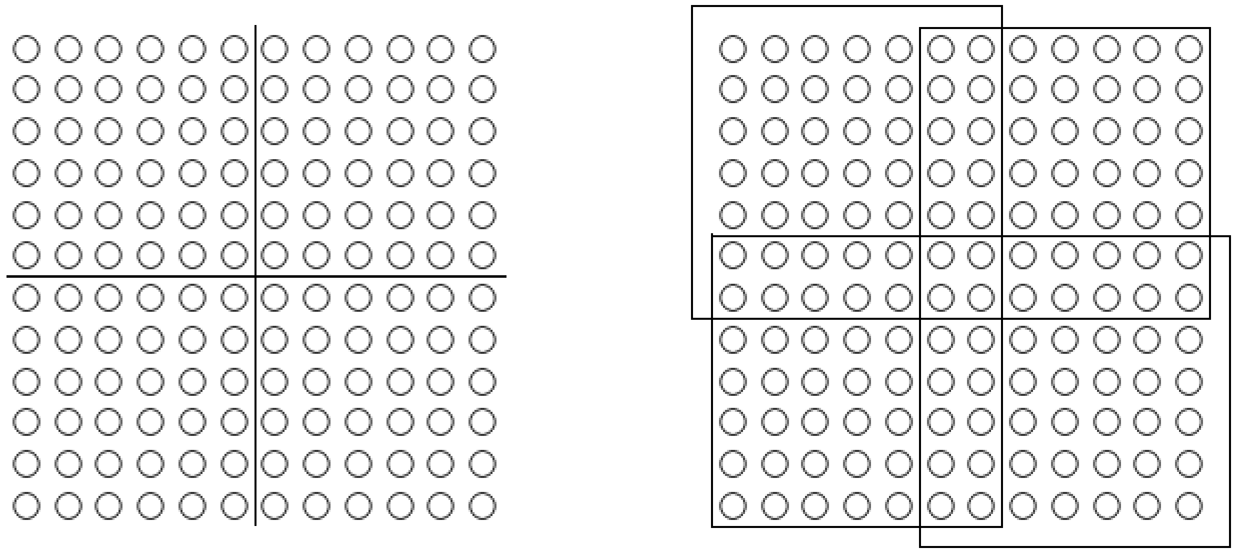


Fig. 6.28: Partitioning of the grid in squares.

Then the boundary elements are proportional to $n/\sqrt{p}$.

In Fig. 6.28, two rows and two columns are shared between two partitions. To reduce the number of shared rows and columns to one, we can take an odd number of rows and columns. In the example of Fig. 6.28, instead of 12 rows and columns, we could take 11 or 13 rows and columns. Then only the middle row and column is shared between the partitions.

Comparing communication costs, we make the following observations. In a square partition, every square has 4 edges, whereas a strip has only 2 edges. For the communication cost, we multiply by 2 because for every send there is a receive. Comparing the communication cost for a strip partitioning

$$t_{\rm comm}^{\rm strip} = 4\left(t_{\rm startup} + n t_{\rm data}\right)$$

to the communication cost for a square partitioning (for $p \geq 9$):

$$t_{\rm comm}^{\rm square} = 8\left(t_{\rm startup} + \frac{n}{\sqrt{p}} t_{\rm data}\right).$$

A strip partition is best if the startup time is large and if we have only very few processors.

If the startup time is low, and for $p \geq 4$, a square partition starts to look better.

This subsection ends with some numerical considerations. The discretization of the heat equation is the simplest one.

- The explicit forward difference method is conditionally stable: in order for the method to converge, the step size in time depends on the step size in space.
- Methods that are unconditionally stable are implicit and require the solving of a linear system in each time step.

6.6.4 Solving the Heat Equation with PETSc

The acronym PETSc stands for Portable, Extensible Toolkit for Scientific Computation. PETSc provides data structures and routines for large-scale application codes on parallel (and serial) computers, using MPI. It supports Fortran, C, C++, Python, and MATLAB (serial) and is free and open source, available at <https://petsc.org>.

6.6.5 Bibliography

1. S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. Curfman McInnes, B. Smith, and H. Zhang. *PETSc Users Manual. Revision 3.2*. Mathematics and Computer Science Division, Argonne National Laboratory, September 2011.
2. Ronald F. Boisvert, L. A. Drummond, Osni A. Marques: Introduction to the special issue on the Advanced CompuTational Software (ACTS) collection. *ACM TOMS* 31(3):281–281, 2005. Special issue on the Advanced CompuTational Software (ACTS) Collection.

6.6.6 Exercises

1. Take the running times of the OpenMP version of the method of Gauss-Seidel and compute the efficiency for each of the 9 cases. What can you conclude about the scalability?
2. Use MPI to write a parallel version of the method of Gauss-Seidel. Compare the speedups with the OpenMP version.
3. Run an example of the PETSc tutorials collection with an increasing number of processes to investigate the speedup.
4. Cellular automata (e.g.: Conway's game of life) are synchronized computations. Discuss a parallel implementation of Conway's game of life and illustrate your discussion with a computation.

6.7 Memory Coalescing Techniques

To take full advantage of the high memory bandwidth of the GPU, the reading from global memory must also run in parallel. We consider memory coalescing techniques to organize the execution of load instructions by a warp.

6.7.1 Accessing Global and Shared Memory

Accessing data in the global memory is critical to the performance of a CUDA application. In addition to tiling techniques utilizing shared memories we discuss memory coalescing techniques to move data efficiently from global memory into shared memory and registers. Global memory is implemented with dynamic random access memories (DRAMs). Reading one DRAM is a very slow process.

Modern DRAMs use a parallel process: Each time a location is accessed, many consecutive locations that includes the requested location are accessed. If an application uses data from consecutive locations before moving on to other locations, the DRAMs work close to the advertised peak global memory bandwidth.

Recall that all threads in a warp execute the same instruction. When all threads in a warp execute a load instruction, the hardware detects whether the threads access consecutive memory locations. The most favorable global memory access is achieved when the same instruction for all threads in a warp accesses global memory locations. In this favorable case, the hardware *coalesces* all memory accesses into a consolidated access to consecutive DRAM locations.

definition of memory coalescing

If, in a warp, thread 0 accesses location n, thread 1 accesses location $n + 1$, ... thread 31 accesses location $n + 31$, then all these accesses are *coalesced*, that is: combined into one single access.

The CUDA C Best Practices Guide gives a high priority recommendation to coalesced access to global memory. An example is shown in Fig. 6.29, extracted from Figure G-1 of the NVIDIA Programming Guide.

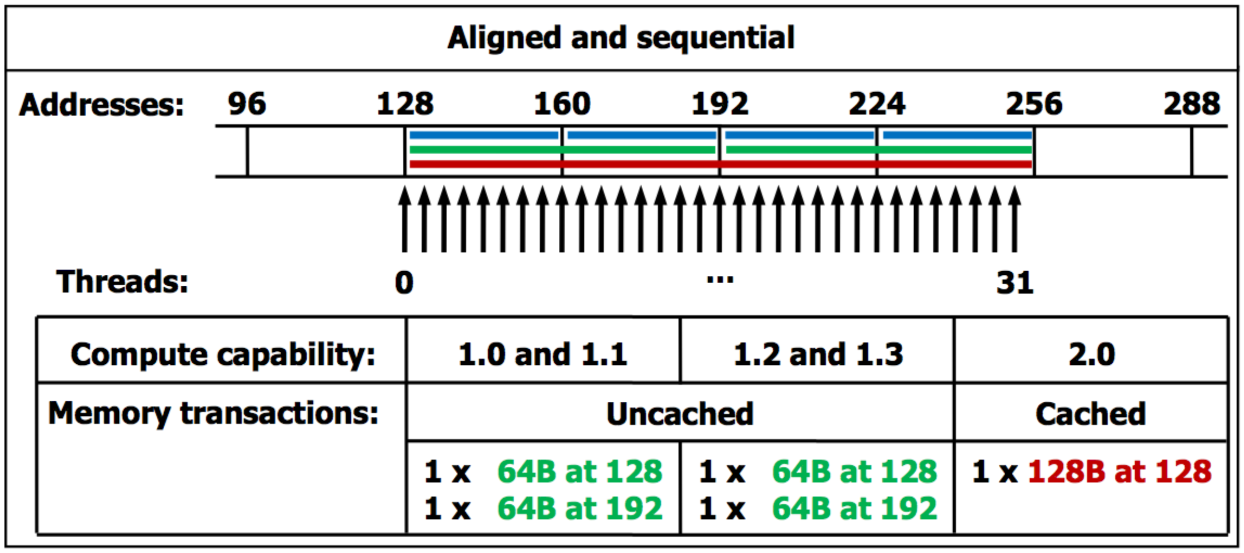


Fig. 6.29: An example of a global memory access by a warp.

More recent examples from the 2016 NVIDIA Programming guide are in Fig. 6.30 and Fig. 6.31.

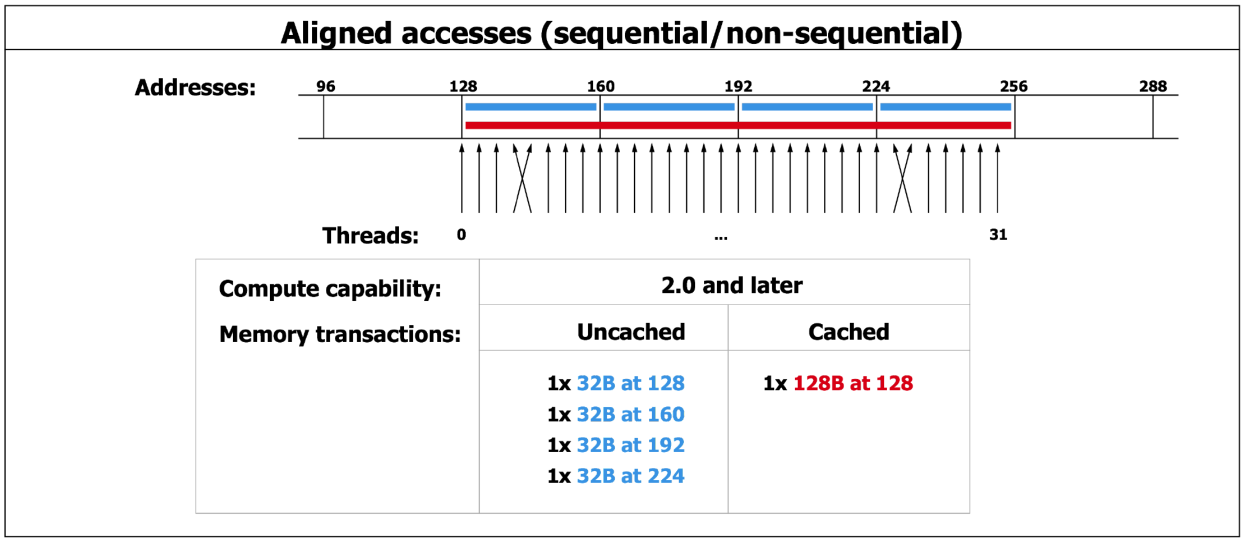


Fig. 6.30: An example of aligned memory access by a ward.

In `/usr/local/cuda/include/vector_types.h` we find the definition of the type `double2` as

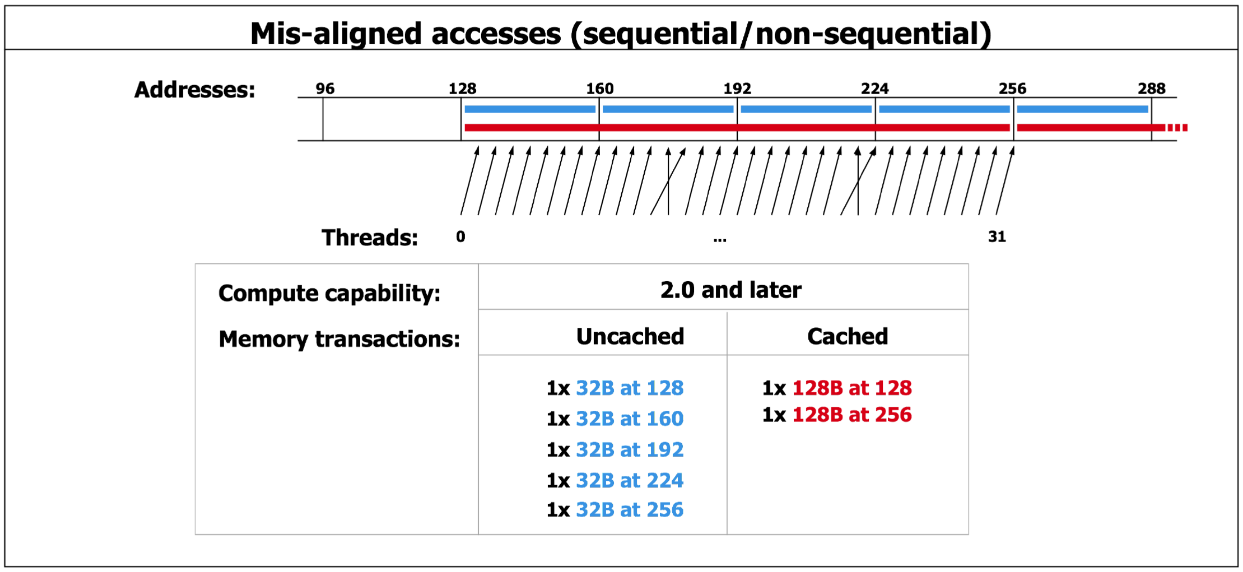


Fig. 6.31: An example of mis-aligned memory access by a ward.

```
struct __device_builtin__ __builtin_align__(16) double2
{
    double x, y;
};
```

The `__align__(16)` causes the doubles in `double2` to be 16-byte or 128-bit aligned. Using the `double2` type for the real and imaginary part of a complex number allows for coalesced memory access.

With a simple copy kernel we can explore what happens when access to global memory is misaligned:

```
__global__ void copyKernel
 ( float *output, float *input, int offset )
{
   int i = blockIdx.x*blockDim.x + threadIdx.x + offset;
   output[i] = input[i];
}
```

The bandwidth will decrease significantly for `offset` > 1.

Shared memory has 32 banks that are organized such that successive 32-bit words are assigned to successive banks, i.e.: interleaved. The bandwidth of shared memory is 32 bits per bank per clock cycle. Because shared memory is on chip, uncached shared memory latency is roughly 100 times slower than global memory.

definition of bank conflict

A *bank conflict* occurs if two or more threads access any bytes within *different* 32-bit words belonging to the *same* bank.

If two or more threads access any bytes within the same 32-bit word, then there is no bank conflict between these threads. The CUDA C Best Practices Guide gives a medium priority recommendation to shared memory access without bank conflicts.

Memory accesses are illustrated in Fig. 6.32 and Fig. 6.33.

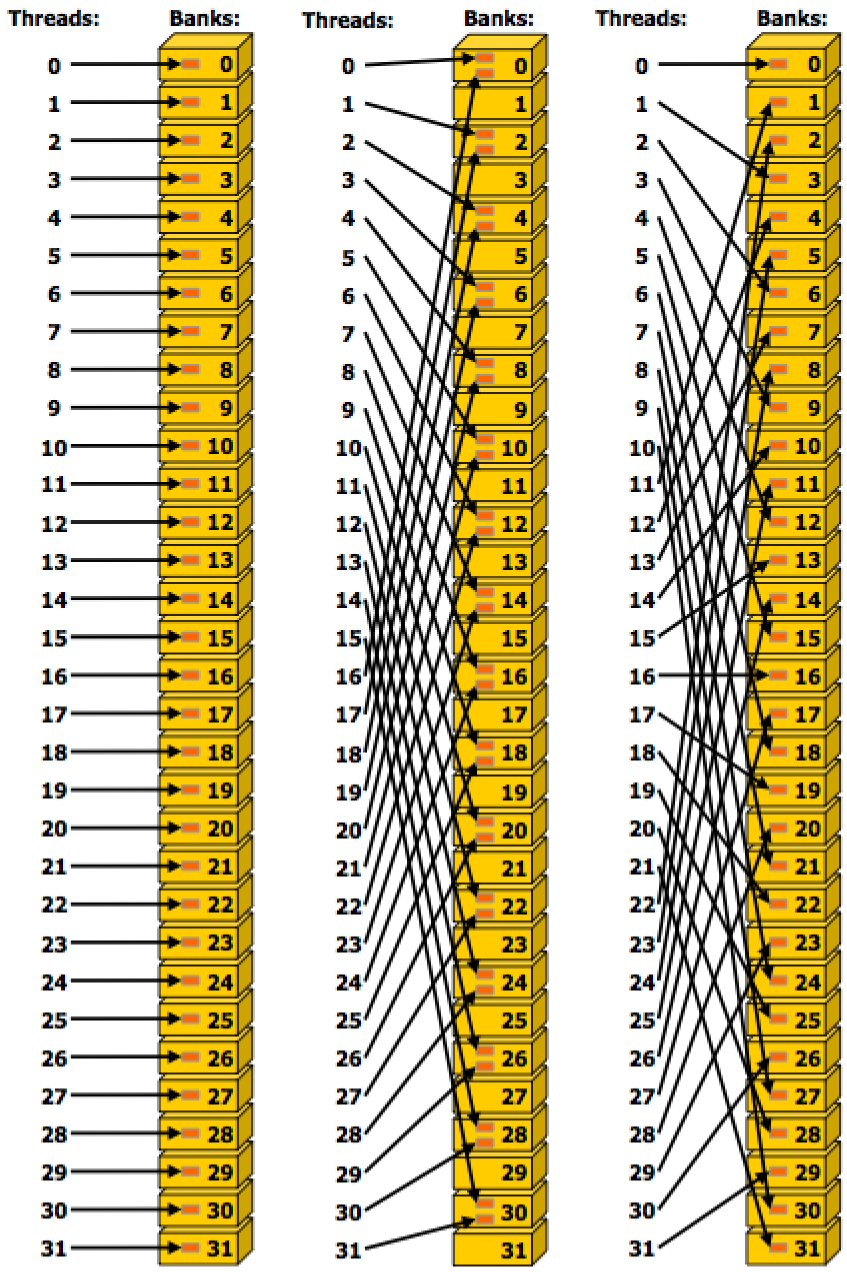


Fig. 6.32: Examples of strided shared memory accesses, copied from Figure G-2 of the NVIDIA Programming Guide.

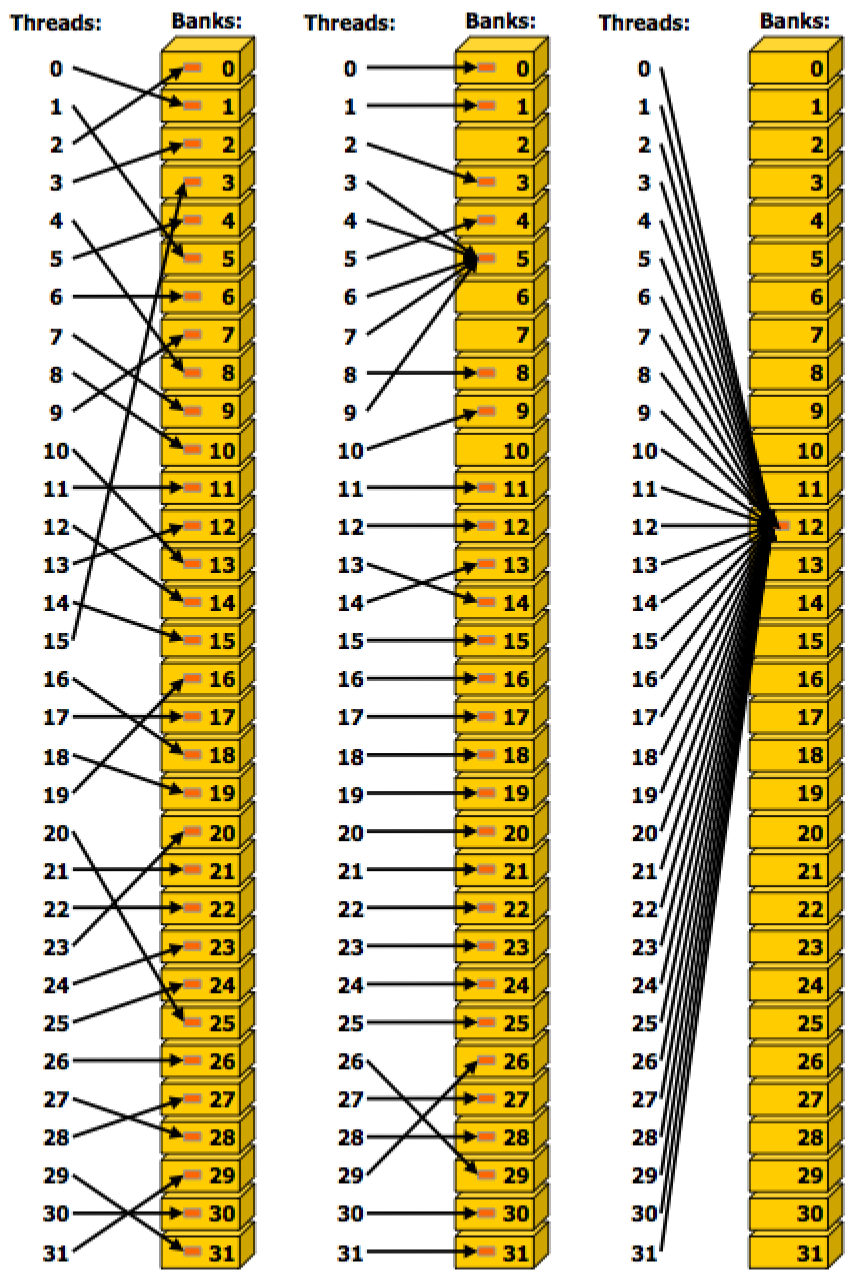


Fig. 6.33: Irregular and colliding shared memory accesses, is Figure G-3 of the NVIDIA Programming Guide.

6.7.2 Memory Coalescing Techniques

Consider two ways of accessing the elements in a matrix:

- elements are accessed row after row; or
- elements are accessed column after column.

These two ways are shown in Fig. 6.34.

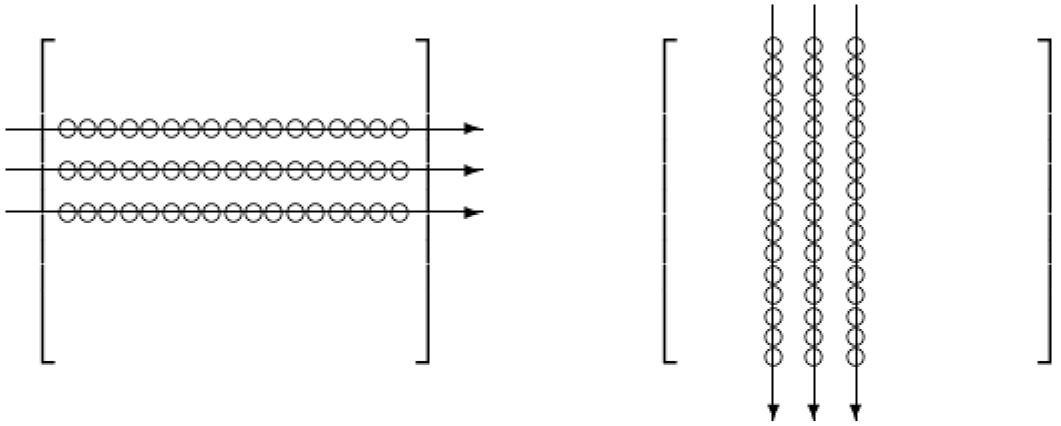


Fig. 6.34: Two ways of accessing elements in a matrix.

Recall the linear address system to store a matrix. In C, the matrix is stored row wise as a one dimensional array, see Fig. 4.12.

Threads t_0, t_1, t_2, and t_3 access the elements on the first two columns, as shown in Fig. 6.35.

Four threads t_0, t_1, t_2, and t_3 access elements on the first two rows, as shown in Fig. 6.36.

The differences between uncoalesced and coalesced memory accesses are shown in Fig. 6.37.

We can use shared memory for coalescing. Consider Fig. 4.25 for the tiled matrix-matrix multiplication.

For $C_{i,j} = \sum_{k=1}^{m/w} A_{i,k} \cdot B_{k,j}$, $A \in \mathbb{R}^{n \times m}$, $B \in \mathbb{R}^{m \times p}$, $A_{i,k}, B_{k,j}, C_{i,j} \in \mathbb{R}^{w \times w}$, every warp reads one tile $A_{i,k}$ of A and one tile $B_{k,j}$ of B: every thread in the warp reads one element of $A_{i,k}$ and one element of $B_{k,j}$.

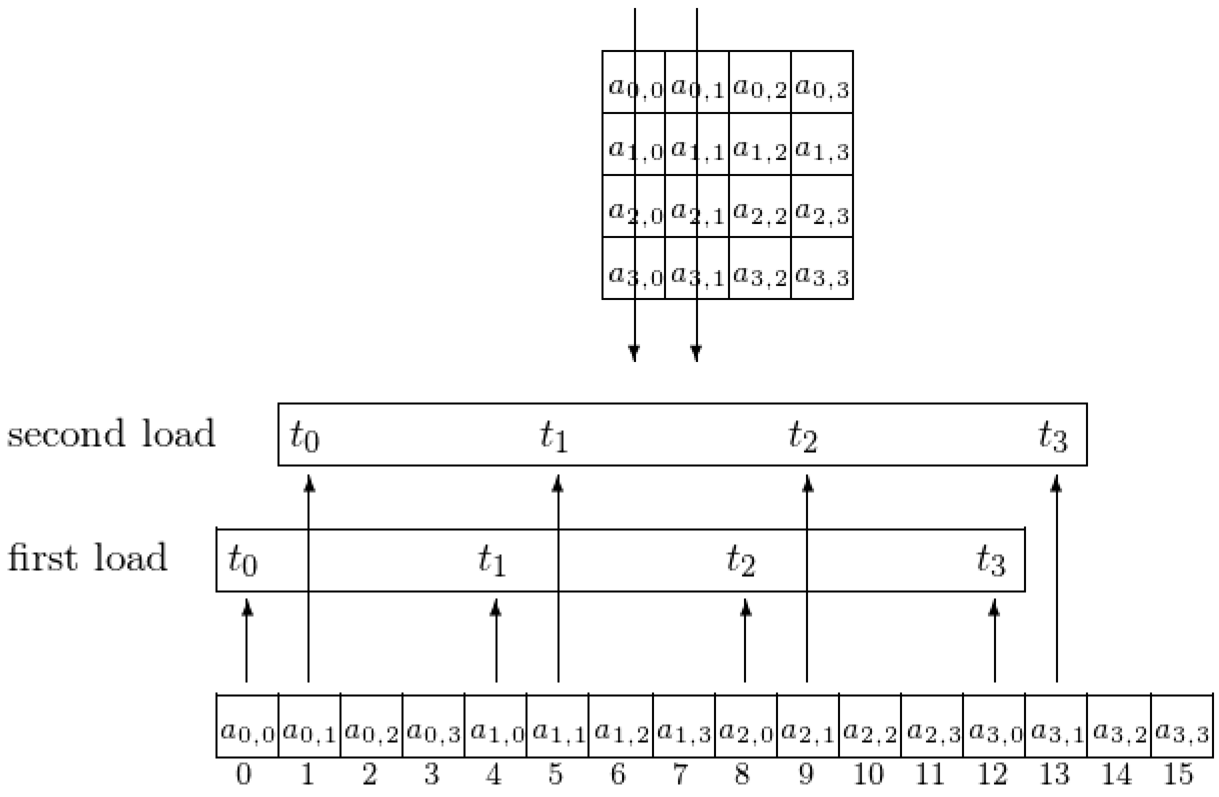
The number of threads equals w, the width of one tile, and threads are identified with `tx = threadIdx.x` and `ty = threadIdx.y`. The `by = blockIdx.y` and `bx = blockIdx.x` correspond respectively to the first and the second index of each tile, so we have `row = by*` w `+ ty` and `col = bx*` w `+ tx`.

Row wise access to A uses A `[row*m + (k*w + tx)]`. For B: B `[(k*w+ty)*m + col]` = B `[(k*w+ty)*m + bx*w+tx]`. Adjacent threads in a warp have adjacent `tx` values so we have coalesced access also to B.

The tiled matrix multiplication kernel is below:

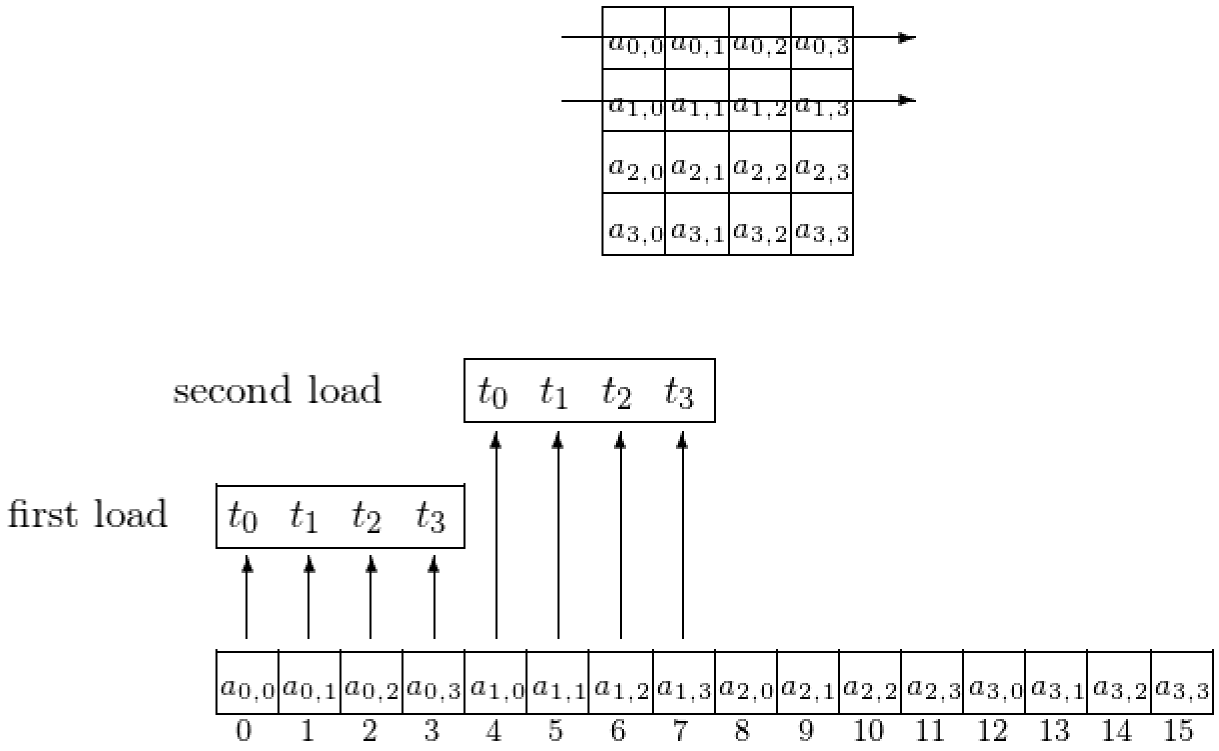
```
__global__ void mul ( float *A, float *B, float *C, int m )
{
   __shared__ float As[w][w];
   __shared__ float Bs[w][w];
   int bx = blockIdx.x;        int by = blockIdx.y;
   int tx = threadIdx.x;       int ty = threadIdx.y;
   int col = bx*w + tx;        int row = by*w + ty;
   float Cv = 0.0;
```

(continues on next page)



second load
first load
t_0 t_1 t_2 t_3
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$ $a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Fig. 6.35: Accessing elements column after column.



second load
first load
t_0 t_1 t_2 t_3
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$ $a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Fig. 6.36: Accesing elements row after row.

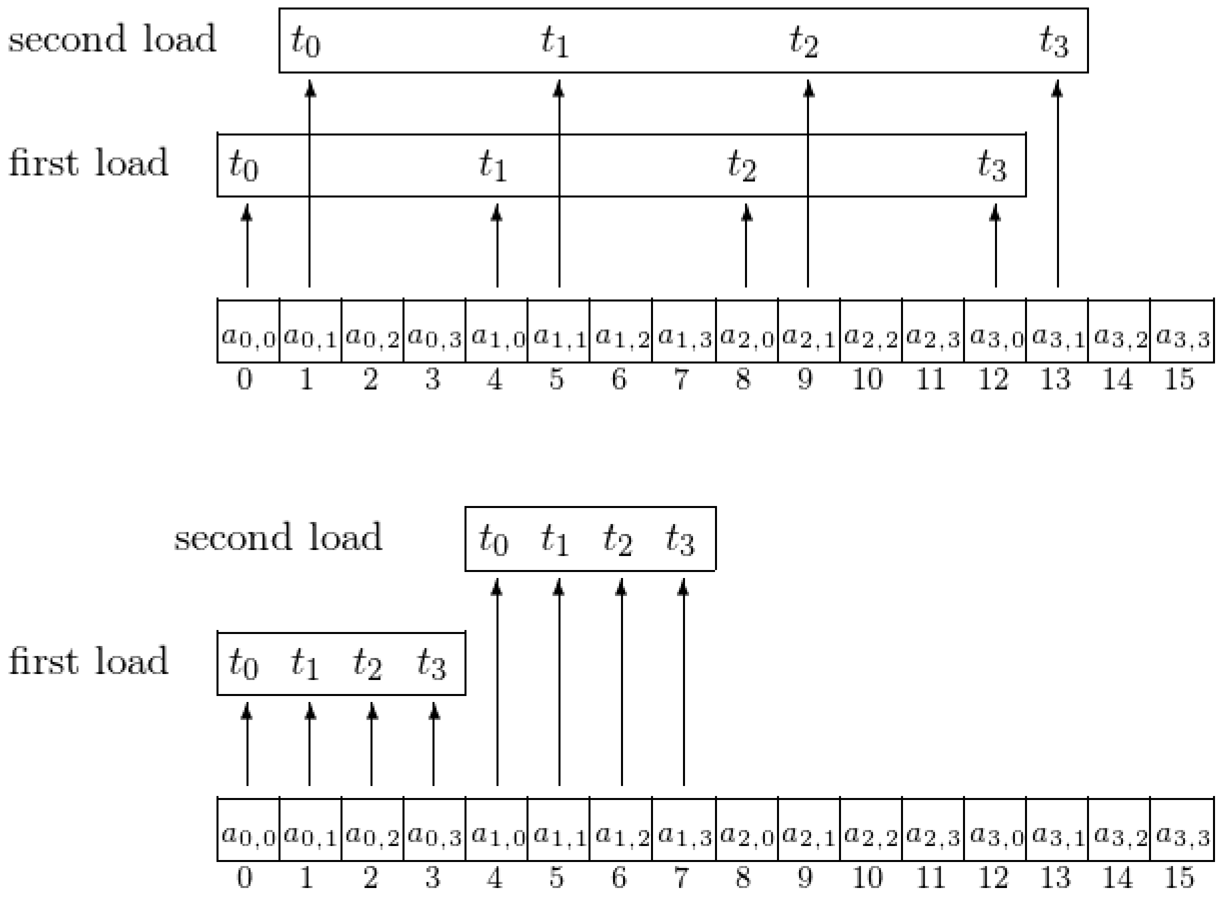

second load
t_0 t_1 t_2 t_3
first load
t_0 t_1 t_2 t_3
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$ $a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
second load
t_0 t_1 t_2 t_3
first load
t_0 t_1 t_2 t_3
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$ $a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Fig. 6.37: Uncoalesced versus coalesced access.

(continued from previous page)

```
   for(int k=0; k<m/w; k++)
   {
      As[ty][tx] = A[row*m + (k*w + tx)];
      Bs[ty][tx] = B[(k*w + ty)*m + col];
      __syncthreads();
      for(int ell=0; ell<w; ell++)
         Cv += As[ty][ell]*Bs[ell][tx];
      C[row][col] = Cv;
   }
}
```

6.7.3 Avoiding Bank Conflicts

All threads in the same warp execute the same instruction. When retrieving/storing data from global memory, one instruction in a kernel defines the retrieval/storage of 32 data elements.

With memory coalescing, retrieving/storing 32 data elements requires as much time as retrieving/storing one data element.

definition of data staging

A *data staging algorithm* arranges the data for memory coalescing.

Arranging data involves positioning the input and output data so that adjacent data elements are accessed by adjacent threads.

Consider an array of complex numbers and/or multiple doubles.

The elements of such arrays are composite.

- Every complex number has a real and imaginary part.
- These parts can be one double, or a multiple double.
- A quad double consists of a most significant double, the second most, third most, fourth most significant double.

Using the straighforward representation will lead to bank conflicts.

Instead of an array of complex doubles, use two arrays:

1. one array with the real doubles,
2. another array with the imaginary doubles.

An array of complex quad doubles is stored in 8 arrays.

Consider the following problem:

On input are $x_0, x_1, x_2, \ldots x_{31}$, all of type float.

The output is

$$\begin{array}{ccccc}
x_0^2, & x_0^3, & x_0^4, & \ldots, & x_0^{33}, \\
x_1^2, & x_1^3, & x_1^4, & \ldots, & x_1^{33}, \\
x_2^2, & x_2^3, & x_2^4, & \ldots, & x_2^{33}, \\
\vdots & \vdots & \vdots & & \vdots \\
x_{31}^2, & x_{31}^3, & x_{31}^4, & \ldots, & x_{31}^{33}.
\end{array}$$

This gives 32 threads in a warp 1,024 multiplications to do. Assume the input and output resides in shared memory. How to compute without bank conflicts?

Suppose we observe the order of the output sequence. If thread i computes $x_i^2, x_i^3, x_i^4, \ldots, x_i^{33}$, then after the first step, all threads write $x_0^2, x_1^2, x_2^2, \ldots, x_{31}^2$ to shared memory. If the stride is 32, all threads write into the same bank. Instead of a simultaneous computation of 32 powers at once, the writing to shared memory will be serialized.

Suppose we alter the order in the output sequence.

$$\begin{array}{ccccc}
x_0^2, & x_1^2, & x_1^2, & \ldots, & x_{31}^2, \\
x_0^3, & x_1^3, & x_2^3, & \ldots, & x_{31}^3, \\
x_0^4, & x_1^4, & x_2^4, & \ldots, & x_{31}^4, \\
\vdots & \vdots & \vdots & & \vdots \\
x_0^{33}, & x_1^{33}, & x_2^{33}, & \ldots, & x_{31}^{33}.
\end{array}$$

After the first step, thread i writes x_i^2 in adjacent memory, next to x_{i-1}^2 (if $i > 0$) and x_{i+1}^2 (if $i < 31$). Without bank conflicts, the speedup will be close to 32.

Below is a basic Julia version of the kernel to solve this problem.

```
using CUDA

"""
    function gpupwr32!(a, b)

raises the elements in the array a
to the powers 2, 3, .., 33,
writing the results in the array b.
"""
function gpupwr32!(a, b)
    i = threadIdx().x    # starts at 1
    idx = 1 + 32*(i-1)
    b[idx] = a[i]*a[i]
    idx = idx + 1
    for p=3:33
        b[idx] = a[i]*b[idx-1]
        idx = idx + 1
    end
    return nothing
end
```

The main program to launch the kernel follows:

```
dx = convert(Float32, 0.2/31)
x_h = [0.9f0 + (k-1)*dx for k=1:32]
y_h = [0.0f0 for k=1:32*32] # output
println("the input numbers : ", x_h)
x_d = CuArray(x_h)
y_d = CuArray(y_h)

# run with 32 threads

@cuda threads=32 gpupwr32!(x_d, y_d)
```

For correctness, the code continues with a comparision with the vector computed on the host.

6.7.4 Exercises

1. Run `copyKernel` for large enough arrays for zero `offset` and an `offset` equal to two. Measure the timings and deduce the differences in memory bandwidth between the two different values for `offset`.
2. Consider the kernel of `matrixMul` in the GPU computing SDK. Is the loading of the tiles into shared memory coalesced? Justify your answer.
3. Write a CUDA program for the computation of consecutive powers, using coalesced access of the values for the input elements. Compare the two orders of storing the output sequence in shared memory: once with and once without bank conflicts.

6.8 Introduction to Tensor Cores

Training deep neural networks is computationally expensive. Tensor cores accelerate convolutions and matrix operations, for use to accelerate high performance computing, data center, and machine learning applications.

While targeted to General Matrix Multiply (GEMM), convolution operations can be reduced to GEMM. The tensor core peak performance in double precision increased from 19.5 on Ampere A100 to 134 TFLOPS on Hopper H100.

6.8.1 High Throughput Computing

The Volta V100 gives a 12-fold increase in throughput, compared to the Pascal P100.

definition of throughput

Throughput measures how much information a system can process in a given amount of time.

High Performance Computing (HPC) measures FLOPS. High Throughput Computing (HTC) measures the number of jobs that can be completed over a long period. While related, HPC is concerned with speed, HTC is also concerned with robustness and reliability.

6.8.2 Volta, Ampere, Hopper Architectures

Each Streaming Multiprocessor (SM) on Volta, shown in Fig. 6.38 has 8 Tensor cores. With 80 SMs, there are 640 tensor cores in total, yielding 125 tensor TFLOPS of mixed precision.

The comparison between Volta V100 and Ampere A100 is illustrated in Fig. 6.39 and Fig. 6.40 compare the Ampere A100 with the Hopper H100.

Tensor cores execute mixed precision matrix operations, illustrated in Fig. 6.41.

Reading Release 12.1 of the CUDA C++ Programming Guide, section 10.2.4, 28 February 2023, about Warp Matrix Functions:

- For compute capability 7.0 or higher.
- Double precision is supported for compute capability at least 8.0.
- All threads in a warp must execute the same code. Code execution is likely to hang otherwise.
- A *fragment* is a templated type with template parameters describing which matrix the fragment holds (A, B or accumulator), the shape of the overall WMMA operation, the data type and, for A and B matrices, whether the data is row or column major.

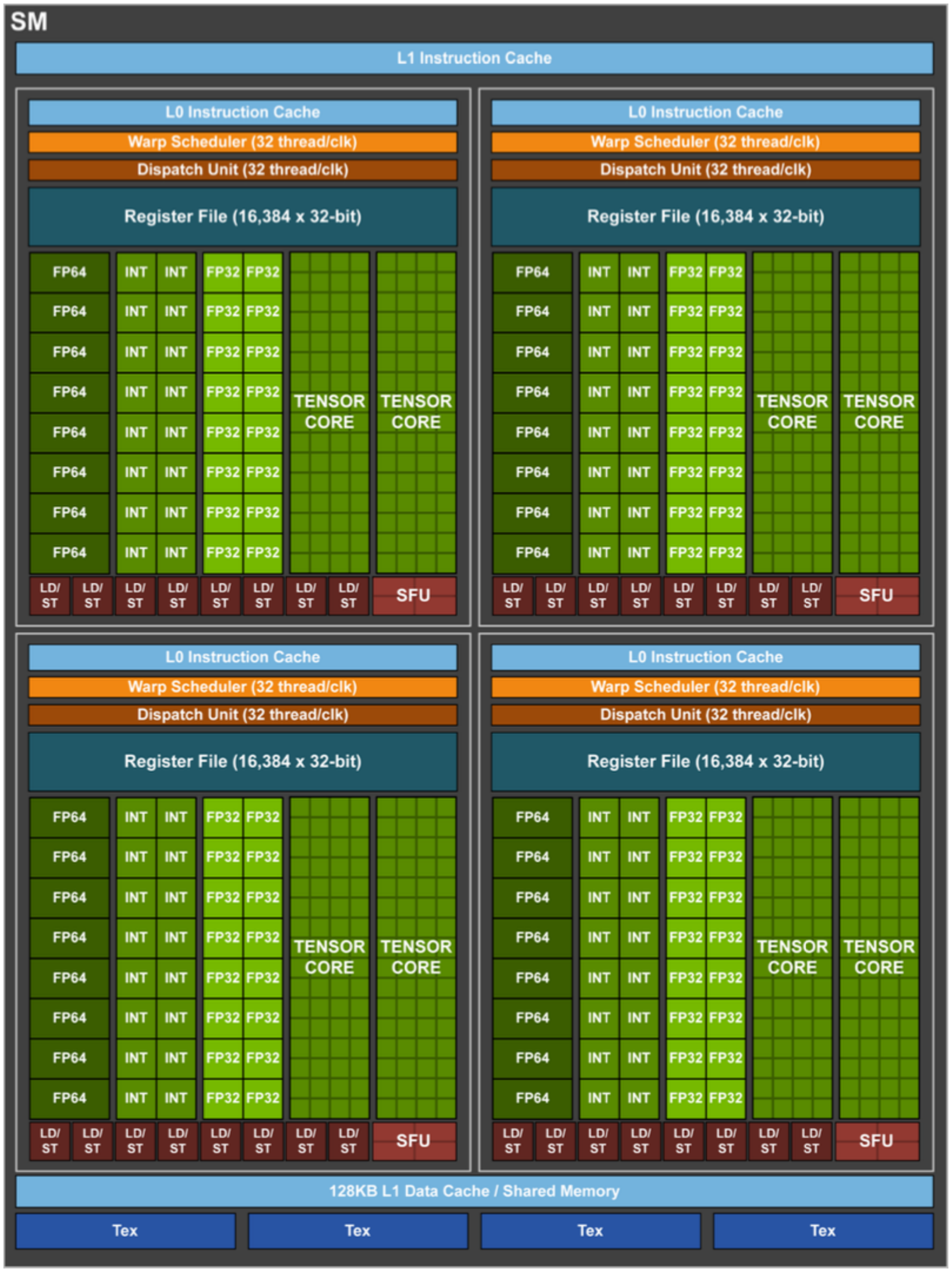

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

Fig. 6.38: The Volta Streaming Multiprocessor, picture from the NVIDIA Volta Architecture white paper.

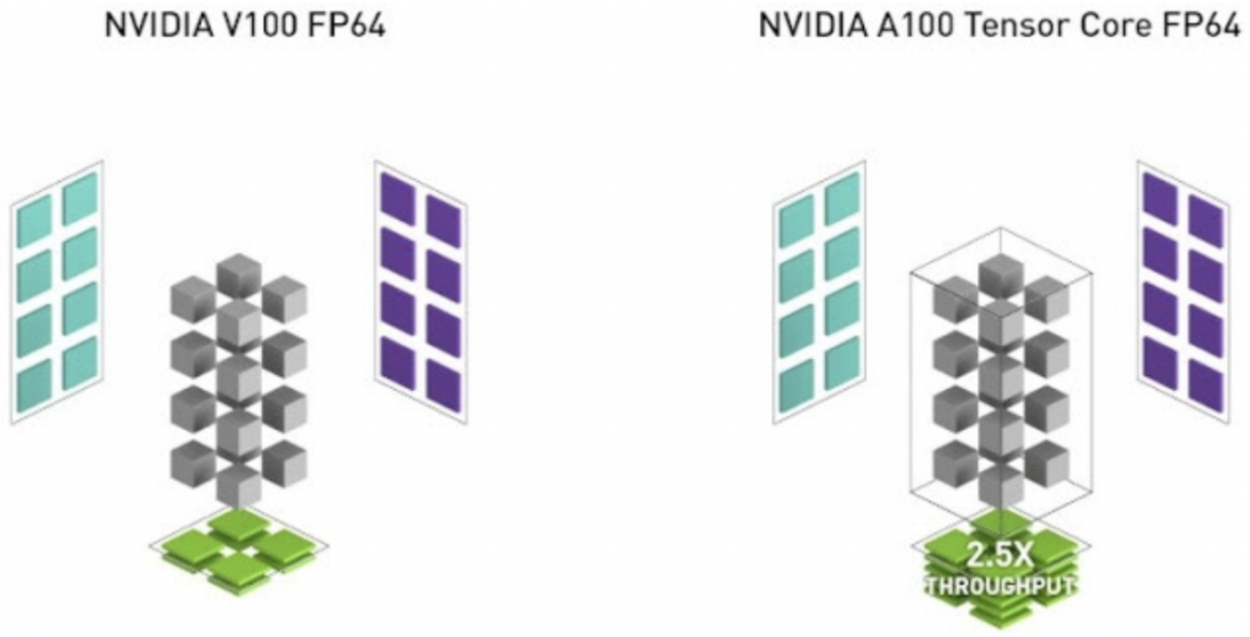


Fig. 6.39: The V100 versus the A100, picture from the NVIDIA Ampere Architecture white paper.

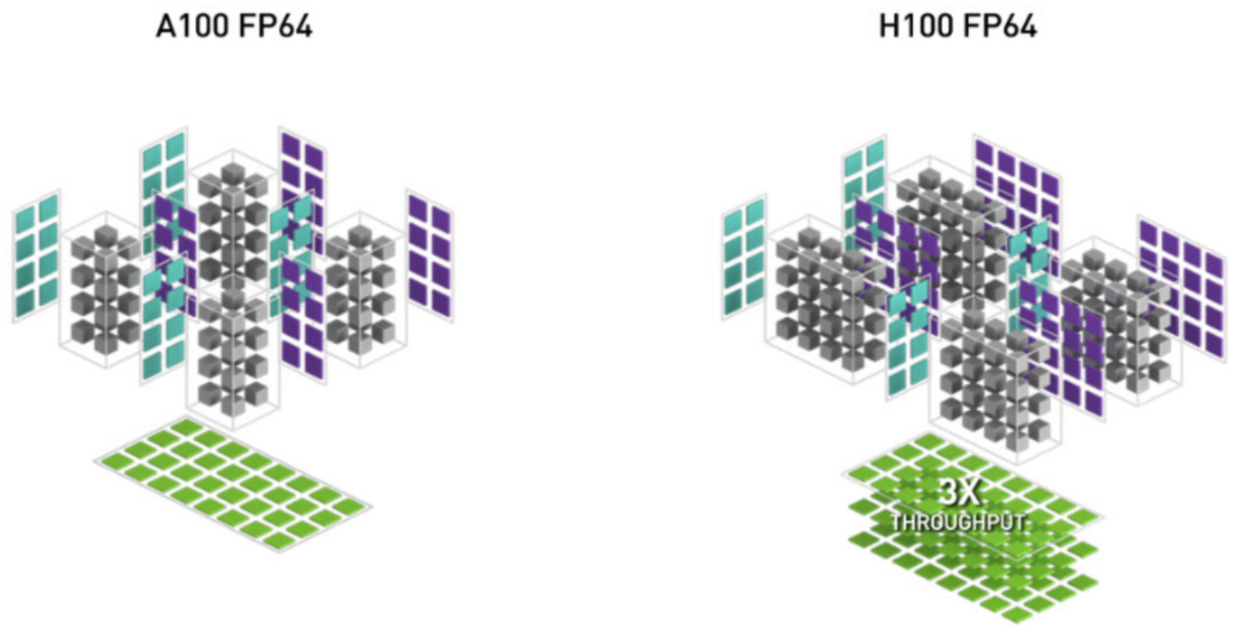


Fig. 6.40: The A100 versus the H100, picture from the NVIDIA Hopper Architecture white paper.

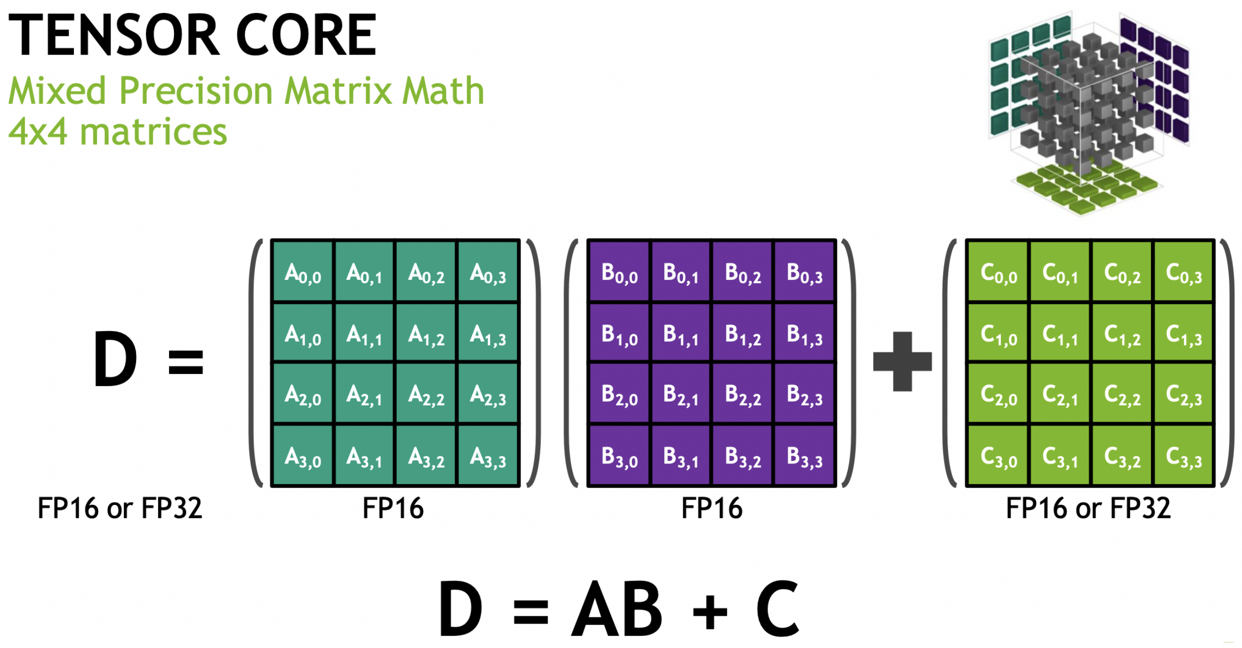


Fig. 6.41: Mixed Precision Matrix Math, picture taken from presentation at SC 2019 by Vishal Mehta on getting started with tensor cores in HPC.

On a Volta computer, in the folder `/usr/local/cuda/samples/0_Simple/cudaTensorCoreGemm` the output of running an example is below:

```
$ ./cudaTensorCoreGemm
Initializing...
GPU Device 0: "Quadro GV100" with compute capability 7.0

M: 4096 (16 x 256)
N: 4096 (16 x 256)
K: 4096 (16 x 256)
Preparing data for GPU...
Required shared memory size: 64 Kb
Computing... using high performance kernel compute_gemm
Time: 2.768896 ms
TFLOPS: 49.64
$
```

On `ampere`, in the folder `cudaTensorCoreGemm` of `/usr/local/cuda/samples/Samples/3_CUDA_Features` the same example is found and below is the output on a a run on the A100.

```
$ ./cudaTensorCoreGemm
Initializing...
GPU Device 0: "Ampere" with compute capability 8.0

M: 4096 (16 x 256)
N: 4096 (16 x 256)
K: 4096 (16 x 256)
Preparing data for GPU...
Required shared memory size: 64 Kb
```

(continues on next page)

(continued from previous page)

```
Computing... using high performance kernel compute_gemm
Time: 1.756160 ms
TFLOPS: 78.26
$
```

Observe the increase in performance of the A100 over the V100. A short explanation of what was computed follows. The Warp Matrix Multipy and Accumulate (WMMA) computes

$$D = \alpha AB + \beta C$$

where

- matrix A is M-by-K row major,
- matrix B is K-by-N column major, and
- matrices C and D are M-by-N.

Each Cooperative Thread Array (CTA) consists of 8 warps and computes one 128-by-128 tile, using shared memory for the matrix C.

6.8.3 Simple Matrix Multiplication

The explanations in this section are based on a 2017 NVIDIA Technical Blog, by Jeremy Appleyard and Scott Yokim, available from <https://developer.nvidia.com/blog> on Programming Tensor Cores in CUDA 9.

The demonstration code is available on github via <https://github.com/NVIDIA-developer-blog/code-samples> posted with a `Makefile`. It show the use of the WMMA (Warp Matrix Multiply Accumulate) API to perform a matrix multiplication.

- For performance, use the `cudaTensorCoreGemm` in the CUDA Toolkit.
- For highest performance, use cuBLAS.

There are four steps in the demonstration code:

1. Use headers and namespaces.
2. Declarations and initialization:

   A simple warp is responsible for a single 16-by-16 section of the output matrix. Tiling happens with a 2D grid:

   ```
   int warpM = (blockIdx.x*blockDim.x+threadIdx.x)/ warpSize;
   int warpN = (blockIdx.y*blockDim.y+threadIdx.y);
   ```

3. The inner loop performs the matrix multiplication.
4. Finishing up: store the accumulated data to memory.

A fragment is a templated type with parameters as follows:

1. which matrix the fragment holds, A, B, or accumulator;
2. the shape of the overall WMMA operation;
3. the data type;
4. for A and B matrices, whether the data is row or column major.

The parameters are specified at the declaration of the fragment. Accumulator fragments are filled with zeros at initialization. Declaration of the fragments is done in the code below:

```
wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;
```

Initialization of the accumulator fragment happens as follows:

```
wmma::fill_fragment(acc_frag, 0.0f);
```

One tile of the output matrix is computed by one warp.

- The loop runs over the rows of A and columns of B, to produce an m-by-n output tile.
- Data is loaded from global memory into a fragment.
- If a tile is discontinous in memory, the stride must be provided to the load function.
- The Matrix Multiply Accumulate (MMA) accumulates in place, so both first and last arguments are the accumulator fragment previously initialized to zero.

The headers and declarations are in the code blocks below:

```
#include <mma.h>
using namespace nvcuda;
```

```
// Must be multiples of 16 for wmma code to work
#define MATRIX_M 16384
#define MATRIX_N 16384
#define MATRIX_K 16384
```

```
// The only dimensions currently supported by WMMA
const int WMMA_M = 16;
const int WMMA_N = 16;
const int WMMA_K = 16;
```

Below is the listing of the start of the kernel:

```
// Performs an MxNxK GEMM (C=alpha*A*B + beta*C) assuming:
//  1) Matrices are packed in memory.
//  2) M, N and K are multiples of 16.
//  3) Neither A nor B are transposed.
__global__ void wmma_example
  ( half *a, half *b, float *c,
    int M, int N, int K, float alpha, float beta)
{
    // Leading dimensions.
    int lda = M;
    int ldb = K;
    int ldc = M;
```

That there is only a single loop is because of the two dimensional organization of the threads. Code for the loop is below:

```
for (int i = 0; i < K; i += WMMA_K)
{
```

(continues on next page)

(continued from previous page)

```
   int aRow = warpM * WMMA_M;
   int aCol = i;

   int bRow = i;
   int bCol = warpN * WMMA_N;

   // Bounds checking
   if (aRow < M && aCol < K && bRow < K && bCol < N)
   {
      // Load the inputs
      wmma::load_matrix_sync(a_frag, a+aRow+aCol*lda, lda);
      wmma::load_matrix_sync(b_frag, b+bRow+bCol*ldb, ldb);

      // Perform the matrix multiplication
      wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
   }
}
```

Observe the computation of the indices for the arguments of `load_matrix_sync`.

The code below loads the current value of `c`, scales it by `beta` and adds this to our result scaled by `alpha`.

```
int cRow = warpM * WMMA_M;
int cCol = warpN * WMMA_N;

if (cRow < M && cCol < N)
{
   wmma::load_matrix_sync(c_frag, c + cRow + cCol * ldc,
                          ldc, wmma::mem_col_major);

#pragma unroll
   for(int i=0; i < c_frag.num_elements; i++)
   {
      c_frag.x[i] = alpha * acc_frag.x[i] + beta * c_frag.x[i];
   }
   // Store the output
   wmma::store_matrix_sync(c + cRow + cCol * ldc, c_frag,
                           ldc, wmma::mem_col_major);
}
```

The output of runs on volta and ampere are listed next:

```
$ ./TCGemm

M = 16384, N = 16384, K = 16384.  alpha = 2.000000, beta = 2.000000

Running with wmma...
Running with cuBLAS...

Checking results...
Results verified: cublas and WMMA agree.
```

On the Volta V100, we obtain:

```
wmma took 631.051270ms
cublas took 99.577888ms
```

On the Ampere A100:

```
wmma took 501.762054ms
cublas took 38.711296ms
```

While A100 is (at least) twice as fast as V100, the performance of the simple matrix multiplication is disappointing, which emphasizes the point that running simple code on faster computer is often pointless.

6.8.4 Bibliography

1. NVIDIA. CUDA C++ Programming Guide.
2. Da Yan, Wei Wang, Xiaowen Chu: **Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply.** In the Proceedings of the 2020 *IEEE International Parallel and Distributed Processing Symposium* (IPDPS), pages 634–643.
3. Thomas Faingnaert, Tim Besard, and Bjorn De Sutter: **Flexible Performant GEMM Kernels on GPUs.** *IEEE Transactions on Parallel and Distributed Systems* 33:(9): 2230–2248, 2022.
4. Massimiliano Fasi, Nicholas J. Higham, Florent Lopez, Theo Mary, and Mantas Mikaitis: **Matrix Multiplication in Multiword Arithmetic: Error Analysis and Application to GPU Tensor Cores.** *SIAM Journal on Scientific Computing* 45(1): C1–C19, 2023.

6.8.5 Exercises

1. Let $\mathbf{x} = (x_0, x_1, x_2, x_3, x_4)$ and $\mathbf{y} = (y_0, y_1, y_2, y_3, y_4)$ be two vectors and consider its convolution $x_0y_4 + x_1y_3 + x_2y_2 + x_3y_1 + x_4y_0$.

   Demonstrate how to rewrite convolutions as matrix products.

2. Install the Julia package `GemmKernels.jl`.

   Read the paper by Faingnaert et al. and run an example of matrix multiplication with the package.

6.9 Performance Considerations

Our goal is to fully occupy the GPU. When launching a kernel, we set the number of blocks and number of threads per block. For full occupancy, we want to reach the largest number of resident blocks and threads. The number of threads ready for execution may be limited by constraints on the number of registers and shared memory.

6.9.1 Dynamic Partitioning of Resources

In Table 6.8 we compare the compute capabilities of a Streaming Multiprocessor (SM) for the graphics cards with respective compute capabilities 1.1, 2.0, 3.5, and 6.0: GeForce 9400M, Tesla C2050/C2070, K20C, and P100.

Table 6.8: Compute Capabilities 1.1, 2.0, 3.5, 6.0.

compute capability	1.1	2.0	3.5	6.0
maximum number of threads per block	512	1,024	1,024	1,024
maximum number of blocks per SM	8	8	16	32
warp size	32	32	32	32
maximum number of warps per SM	24	48	64	64
maximum number of threads per SM	768	1,536	2,048	2,048

During runtime, thread slots are partitioned and assigned to thread blocks. Streaming multiprocessors are versatile by their ability to dynamically partition the thread slots among thread blocks. They can either execute many thread blocks of few threads each, or execute a few thread blocks of many threads each. In contrast, fixed partitioning where the number of blocks and threads per block are fixed will lead to waste.

We consider the interactions between resource limitations on the C2050. The Tesla C2050/C2070 has 1,536 thread slots per streaming multiprocessor. As $1,536 = 32 \times 48$, we have

$$\text{number of thread slots } = \text{warp size } \times \text{ number of warps per block.}$$

For 32 threads per block, we have 1,536/32 = 48 blocks. However, we can have at most 8 blocks per streaming multiprocessor. Therefore, to fully utilize both the block and thread slots, to have 8 blocks, we should have

- $1,536/8 = 192$ threads per block, or
- $192/32 = 6$ warps per block.

On the K20C, the interaction between resource liminations differ. The K20C has 2,048 thread slots per streaming multiprocessor. The total number of thread slots equals $2,048 = 32 \times 64$. For 32 threads per block, we have 2,048/32 = 64 blocks. However, we can have at most 16 blocks per streaming multiprocessor. Therefore, to fully utilize both the block and thread slots, to have 16 blocks, we should have

- $2,048/16 = 128$ threads per block, or
- $128/32 = 4$ warps per block.

On the P100, there is another slight difference in the resource limitation, which leads to another outcome. In particular, we now can have at most 32 blocks per streaming multiprocessor. To have 32 blocks, we should have

- $2,048/32 = 64$ threads per block, or
- $64/32 = 2$ warps per block.

The memory resources of a streaming multiprocessor are compared in Table 6.9, for the graphics cards with respective compute capabilities 1.1, 2.0, 3.5, and 6.0: GeForce 9400M, Tesla C2050/C2070, K20C, and P100.

Table 6.9: memory resources for several compute capabilities.

compute capability	1.1	2.0	3.5	6.0
number of 32-bit registers per SM	8K	32KB	64KB	64KB
maximum amount of shared memory per SM	16KB	48KB	48KB	64KB
number of shared memory banks	16	32	32	32
amount of local memory per thread	16KB	512KB	512KB	512KB
constant memory size	64KB	64KB	64KB	64KB
cache working set for constant memory per SM	8KB	8KB	8KB	10KB

Local memory resides in device memory, so local memory accesses have the same high latency and low bandwidth as global memory.

Registers hold frequently used programmer and compiler-generated variables to reduce access latency and conserve memory bandwidth. Variables in a kernel that are not arrays are automatically placed into registers.

By dynamically partitioning the registers among blocks, a streaming multiprocessor can accommodate more blocks if they require few registers, and fewer blocks if they require many registers. As with block and thread slots, there is a potential interaction between register limitations and other resource limitations.

Consider the matrix-matrix multiplication example. Assume

- the kernel uses 21 registers, and
- we have 16-by-16 thread blocks.

How many threads can run on each streaming multiprocessor?

1. We calculate the number of registers for each block: $16 \times 16 \times 21 = 5,376$ registers.
2. We have $32 \times 1,024$ registers per SM: $32 \times 1,024/5,376 = 6$ blocks; and $6 < 8 =$ the maximum number of blocks per SM.
3. We calculate the number of threads per SM: $16 \times 16 \times 6 = 1,536$ threads; and we can have at most 1,536 threads per SM.

We now introduce the performance cliff, assuming a slight increase in one resource. Suppose we use one extra register, 22 instead of 21. To answer how many threads now can run on each SM, we follow the same calculations.

1. We calculate the number of registers for each block: $16 \times 16 \times 22 = 5,632$ registers.
2. We have $32 \times 1,024$ registers per SM: $32 \times 1,024/5,632 = 5$ blocks.
3. We calculate the number of threads per SM: $16 \times 16 \times 5 = 1,280$ threads; and with 21 registers we could use all 1,536 threads per SM.

Adding one register led to a reduction of 17% in the parallelism.

Definition of performance cliff

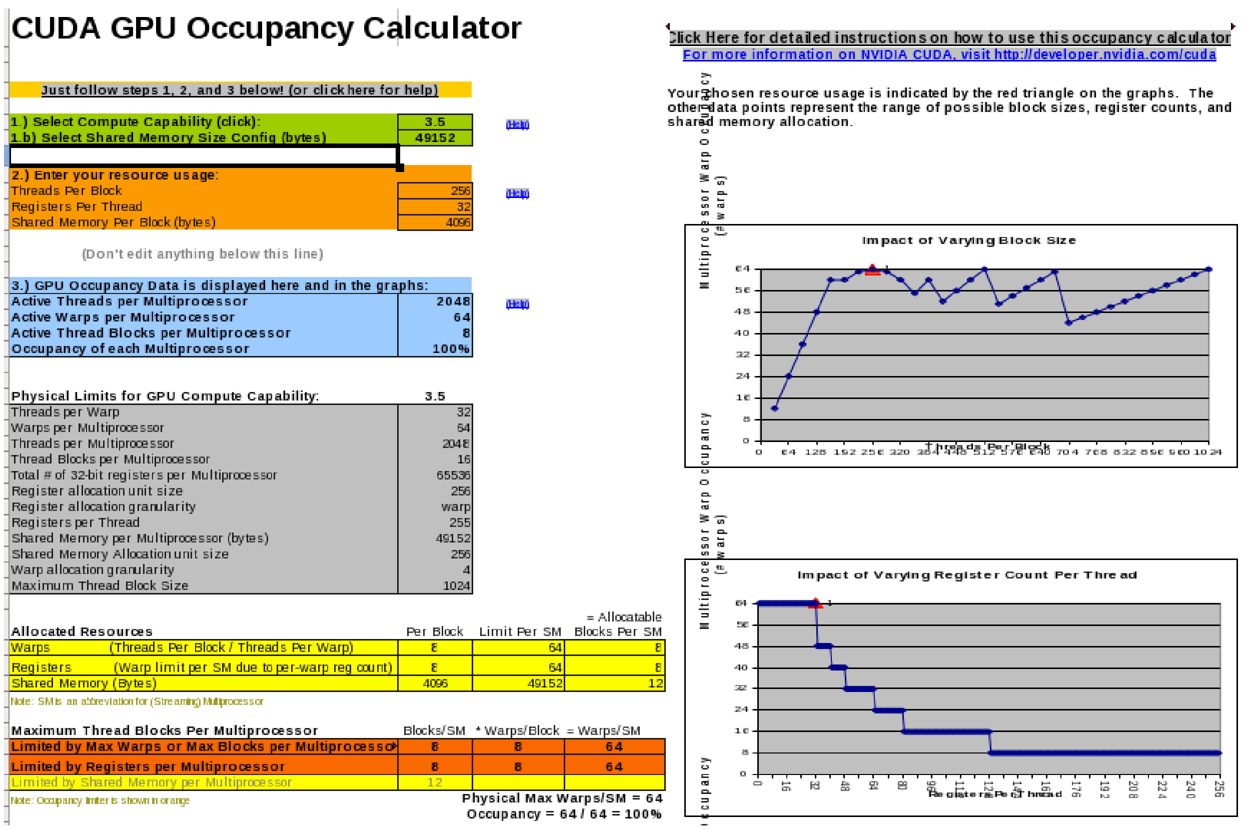
When a slight increase in one resource leads to a dramatic reduction in parallelism and performance, one speaks of a *performance cliff*.

The CUDA compiler tool set contains a spreadsheet to compute the occupancy of the GPU, as shown in Fig. 6.42.

6.9.2 The Compute Visual Profiler

The Compute Visual Profiler is a graphical user interface based profiling tool to measure performance and to find potential opportunities for optimization in order to achieve maximum performance.

We look at one of the example projects `matrixMul`. The analysis of the kernel `matrixMul` is displayed in Fig. 6.43, Fig. 6.44, Fig. 6.45, Fig. 6.46, and Fig. 6.47.



CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

1.) Select Compute Capability (click):	3.5
1.b) Select Shared Memory Size Config (bytes)	49152

2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	32
Shared Memory Per Block (bytes)	4096

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	8
Occupancy of each Multiprocessor	100%

Physical Limits for GPU Compute Capability:	3.5
Threads per Warp	32
Warps per Multiprocessor	64
Threads per Multiprocessor	2048
Thread Blocks per Multiprocessor	16
Total # of 32-bit registers per Multiprocessor	65536
Register allocation unit size	256
Register allocation granularity	warp
Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	256
Warp allocation granularity	4
Maximum Thread Block Size	1024

Allocated Resources		Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps	(Threads Per Block / Threads Per Warp)	8	64	8
Registers	(Warp limit per SM due to per-warp reg count)	8	64	8
Shared Memory (Bytes)		4096	49152	12

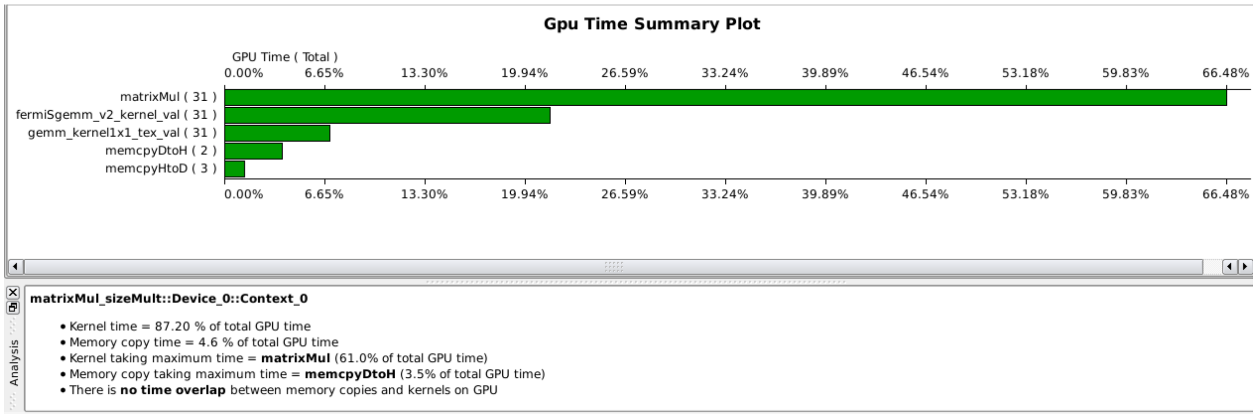
Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	8	8	64
Limited by Registers per Multiprocessor	8	8	64
Limited by Shared Memory per Multiprocessor	12		

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64
Occupancy = 64 / 64 = 100%

Fig. 6.42: The CUDA occupancy calculator.



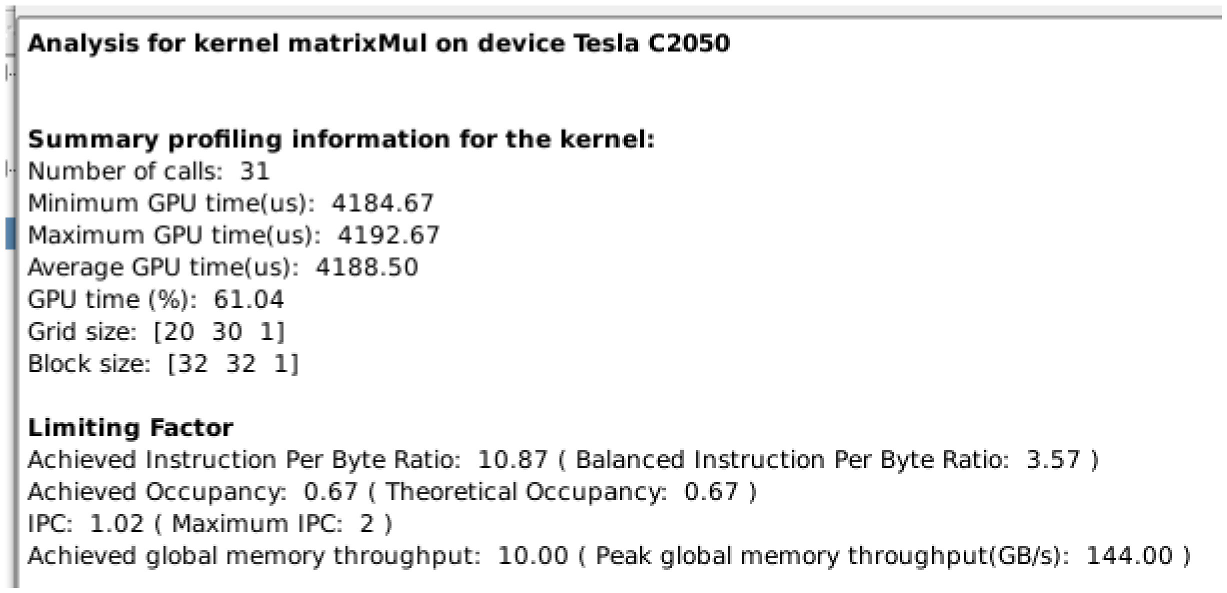
Gpu Time Summary Plot

Kernel	GPU Time (Total)
matrixMul (31)	
fermiSgemm_v2_kernel_val (31)	
gemm_kernel1x1_tex_val (31)	
memcpyDtoH (2)	
memcpyHtoD (3)	

matrixMul_sizeMult::Device_0::Context_0

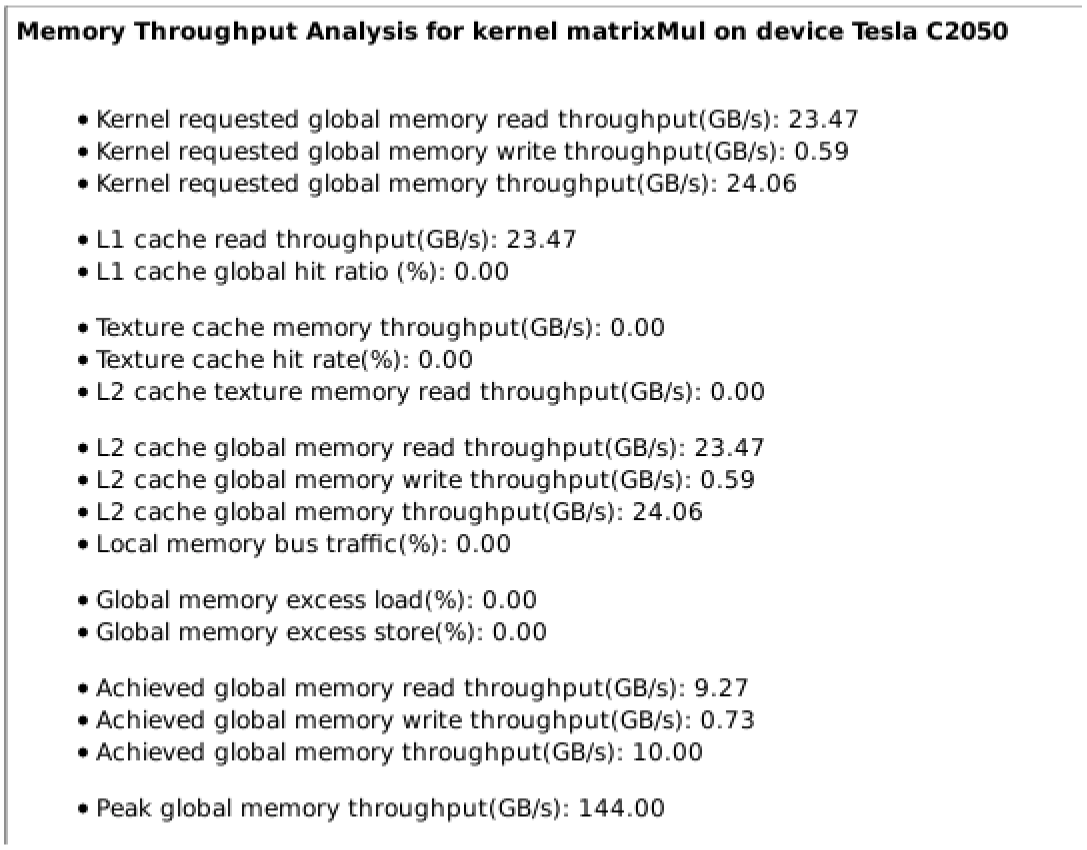
- Kernel time = 87.20 % of total GPU time
- Memory copy time = 4.6 % of total GPU time
- Kernel taking maximum time = **matrixMul** (61.0% of total GPU time)
- Memory copy taking maximum time = **memcpyDtoH** (3.5% of total GPU time)
- There is **no time overlap** between memory copies and kernels on GPU

Fig. 6.43: GPU time summary of the matrixMul kernel.



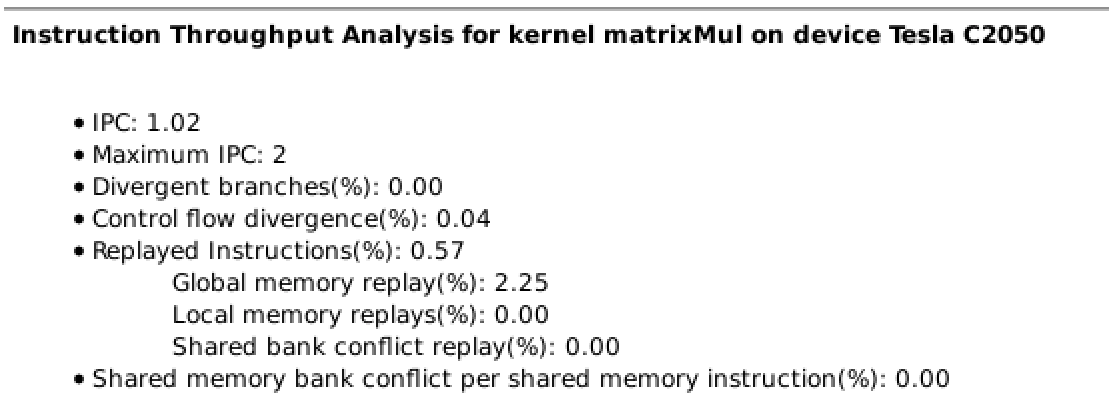
```
Analysis for kernel matrixMul on device Tesla C2050


Summary profiling information for the kernel:
Number of calls:  31
Minimum GPU time(us):  4184.67
Maximum GPU time(us):  4192.67
Average GPU time(us):  4188.50
GPU time (%):  61.04
Grid size:  [20  30  1]
Block size:  [32  32  1]

Limiting Factor
Achieved Instruction Per Byte Ratio:  10.87 ( Balanced Instruction Per Byte Ratio:  3.57 )
Achieved Occupancy:  0.67 ( Theoretical Occupancy:  0.67 )
IPC:  1.02 ( Maximum IPC:  2 )
Achieved global memory throughput:  10.00 ( Peak global memory throughput(GB/s):  144.00 )
```

Fig. 6.44: Limiting factor identification of the matrixMul kernel, IPC = Instructions Per Cycle.



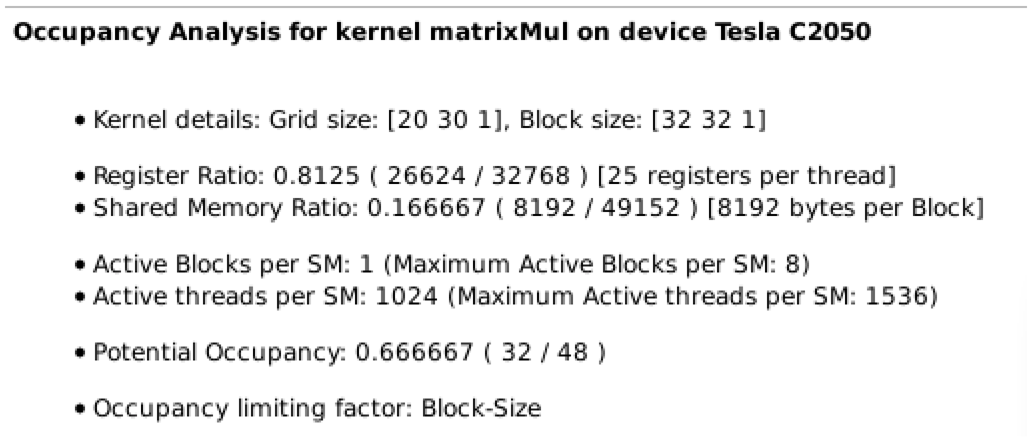
```
Memory Throughput Analysis for kernel matrixMul on device Tesla C2050


    • Kernel requested global memory read throughput(GB/s): 23.47
    • Kernel requested global memory write throughput(GB/s): 0.59
    • Kernel requested global memory throughput(GB/s): 24.06

    • L1 cache read throughput(GB/s): 23.47
    • L1 cache global hit ratio (%): 0.00

    • Texture cache memory throughput(GB/s): 0.00
    • Texture cache hit rate(%): 0.00
    • L2 cache texture memory read throughput(GB/s): 0.00

    • L2 cache global memory read throughput(GB/s): 23.47
    • L2 cache global memory write throughput(GB/s): 0.59
    • L2 cache global memory throughput(GB/s): 24.06
    • Local memory bus traffic(%): 0.00

    • Global memory excess load(%): 0.00
    • Global memory excess store(%): 0.00

    • Achieved global memory read throughput(GB/s): 9.27
    • Achieved global memory write throughput(GB/s): 0.73
    • Achieved global memory throughput(GB/s): 10.00

    • Peak global memory throughput(GB/s): 144.00
```

Fig. 6.45: Memory throughput analysis of the matrixMul kernel.



```
Instruction Throughput Analysis for kernel matrixMul on device Tesla C2050

  • IPC: 1.02
  • Maximum IPC: 2
  • Divergent branches(%): 0.00
  • Control flow divergence(%): 0.04
  • Replayed Instructions(%): 0.57
          Global memory replay(%): 2.25
          Local memory replays(%): 0.00
          Shared bank conflict replay(%): 0.00
  • Shared memory bank conflict per shared memory instruction(%): 0.00
```

Fig. 6.46: Instruction throughput analysis of the matrixMul kernel, IPC = Instructions Per Cycle.



```
Occupancy Analysis for kernel matrixMul on device Tesla C2050

  • Kernel details: Grid size: [20 30 1], Block size: [32 32 1]

  • Register Ratio: 0.8125 ( 26624 / 32768 ) [25 registers per thread]
  • Shared Memory Ratio: 0.166667 ( 8192 / 49152 ) [8192 bytes per Block]

  • Active Blocks per SM: 1 (Maximum Active Blocks per SM: 8)
  • Active threads per SM: 1024 (Maximum Active threads per SM: 1536)

  • Potential Occupancy: 0.666667 ( 32 / 48 )

  • Occupancy limiting factor: Block-Size
```

Fig. 6.47: Occupancy analysis of the matrixMul kernel.

6.9.3 Data Prefetching and Instruction Mix

One of the most important resource limitations is access to global memory and long latencies. Scheduling other warps while waiting for memory access is powerful, but often not enough. A complementary to warp scheduling solution is to prefetch the next data elements while processing the current data elements. Combined with tiling, data prefetching provides extra independent instructions to enable the scheduling of more warps to tolerate long memory access latencies.

For the tiled matrix-matrix multiplication, the pseudo code below combines prefetching with tiling:

```
load first tile from global memory into registers;
loop
{
   deposit tile from registers to shared memory;
   __syncthreads();
   load next tile from global memory into registers;
   process current tile;
   __syncthreads();
}
```

The prefetching adds independent instructions between loading the data from global memory and processing the data.

The data in Table 6.10 is copied from Table 2 of the CUDA C Programming Guide. The ftp in Table 6.10 stands for floating-point and int for integer.

Table 6.10: Number of operations per clock cycle per multiprocessor.

compute capability	1.x	2.0	3.5	6.0
32-bit fpt add, multiply, multiply-add	8	32	192	64
64-bit fpt add, multiply, multiply-add	1	16	64	4
32-bit int add, logical operation, shift, compare	8	32	160	128
32-bit fpt reciprocal, sqrt, log, exp, sin, cos	2	4	32	32

Consider the following code snippet:

```
for(int k = 0; k < m; k++)
   C[i][j] += A[i][k]*B[k][j];
```

Counting all instructions:

- 1 loop branch instruction (`k < m`);
- 1 loop counter update instruction (`k++`);
- 3 address arithmetic instructions (`[i][j], [i][k], [k][j]`);
- 2 floating-point arithmetic instructions (+ and *).

Of the 7 instructions, only 2 are floating point.

Loop unrolling reduces the number of loop branch instructions, loop counter updates, address arithmetic instructions. Note: `gcc -funroll-loops` is enabled with `gcc -O2`.

6.9.4 Thread Coarsening

Acceleration by GPUs applies fine grained parallelism, often at the instruction level, following the single instruction multiple data model.

definition of thread coarsening

By *thread coarsening*, each thread is given more work, to reduce the overhead caused by parallelism.

One typical situation occurs with the block size limitation, when the number of threads is insufficient. As a consequence of thread coarsening, the number of threads in a block decreases, overcoming the block size limitation.

The application of thread coarsening to tiled matrix matrix multiplication is illustrated in a sequence of three pictures, in Fig. 6.48, in Fig. 6.49, and in Fig. 6.49.

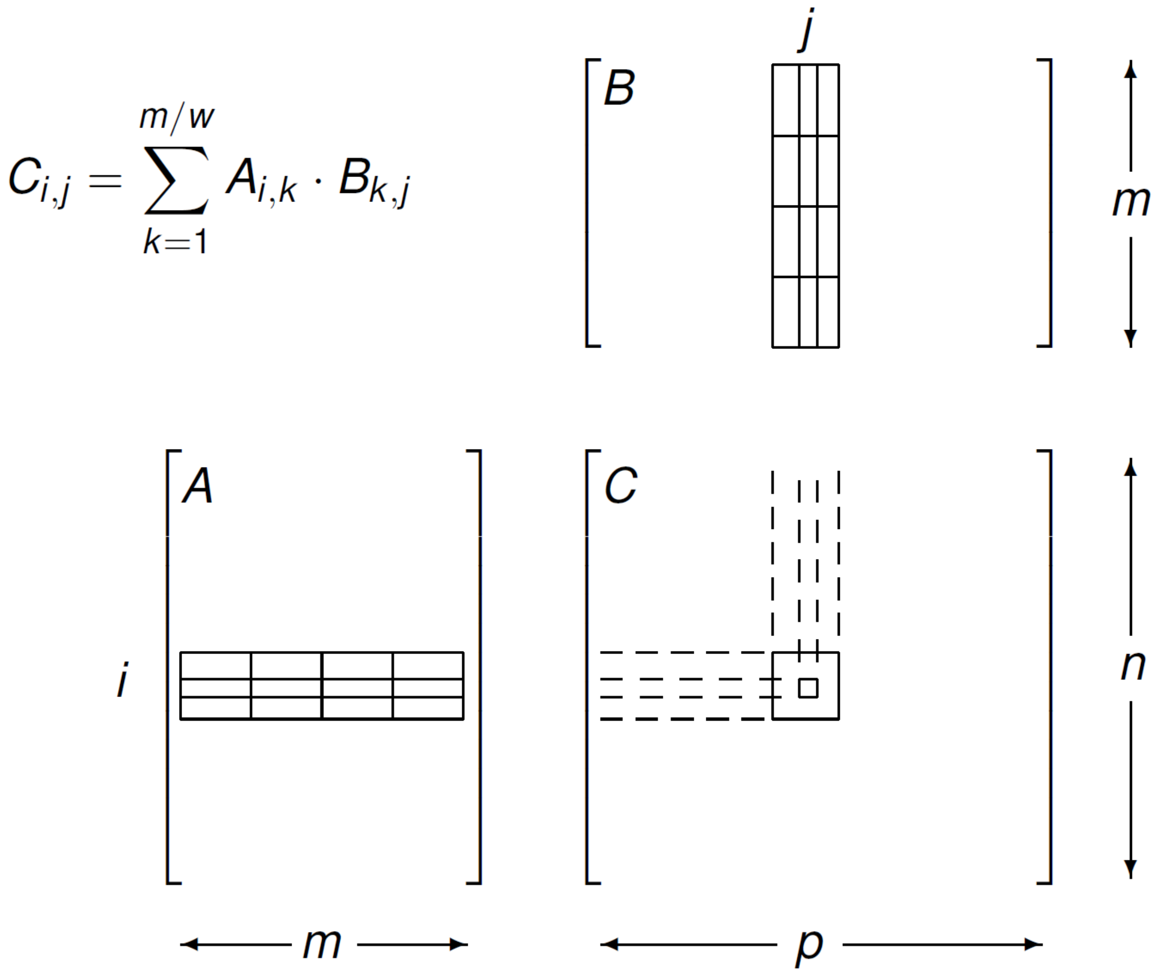


Fig. 6.48: Tiled matrix matrix multiplication.

In the matrix matrix multiplication with shared memory, one output tile is computed by one block of threads:

- Each block loads one tile of A and one tile of B.
- Shared memory is not shared among the blocks.

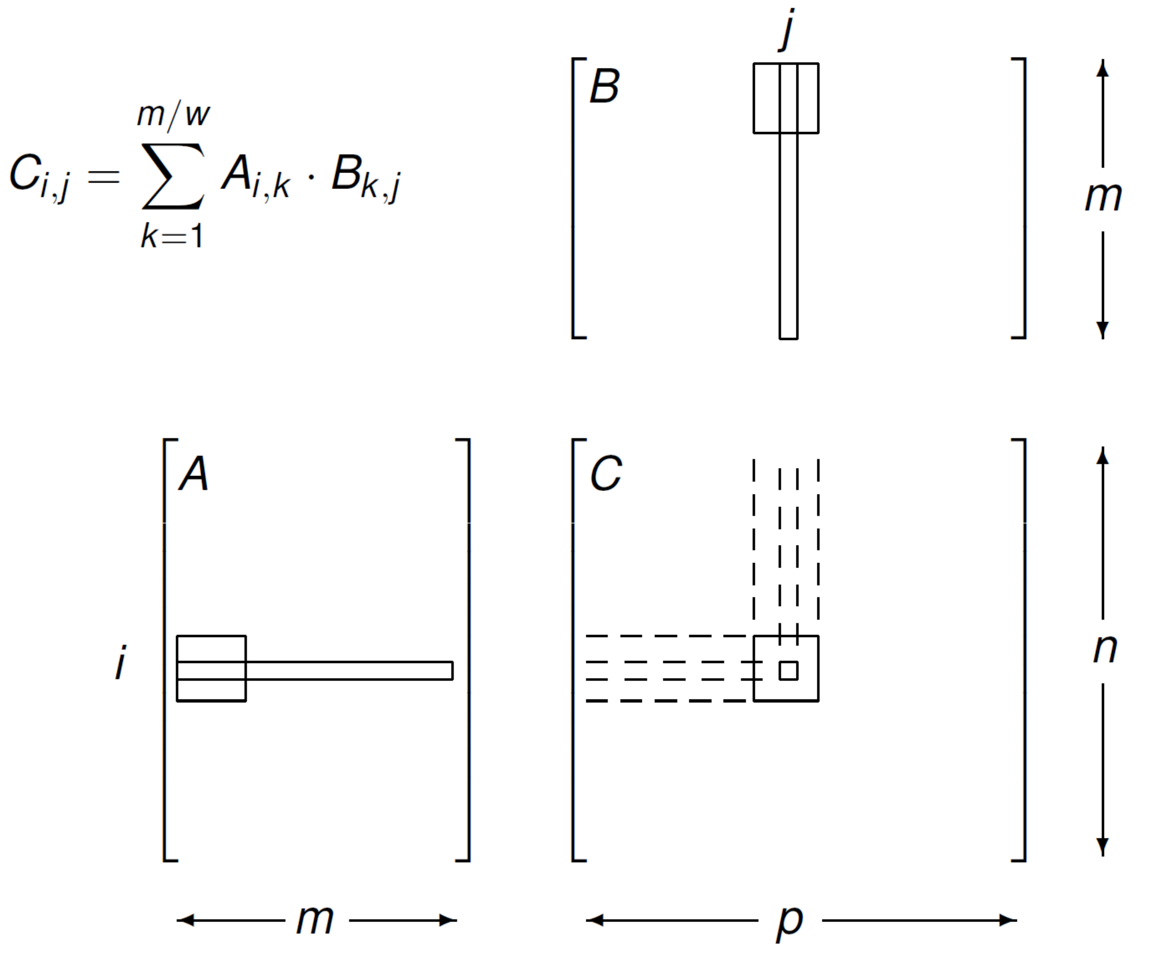


Fig. 6.49: In tiled matrix matrix multiplication, one block of threads computes one output tile.

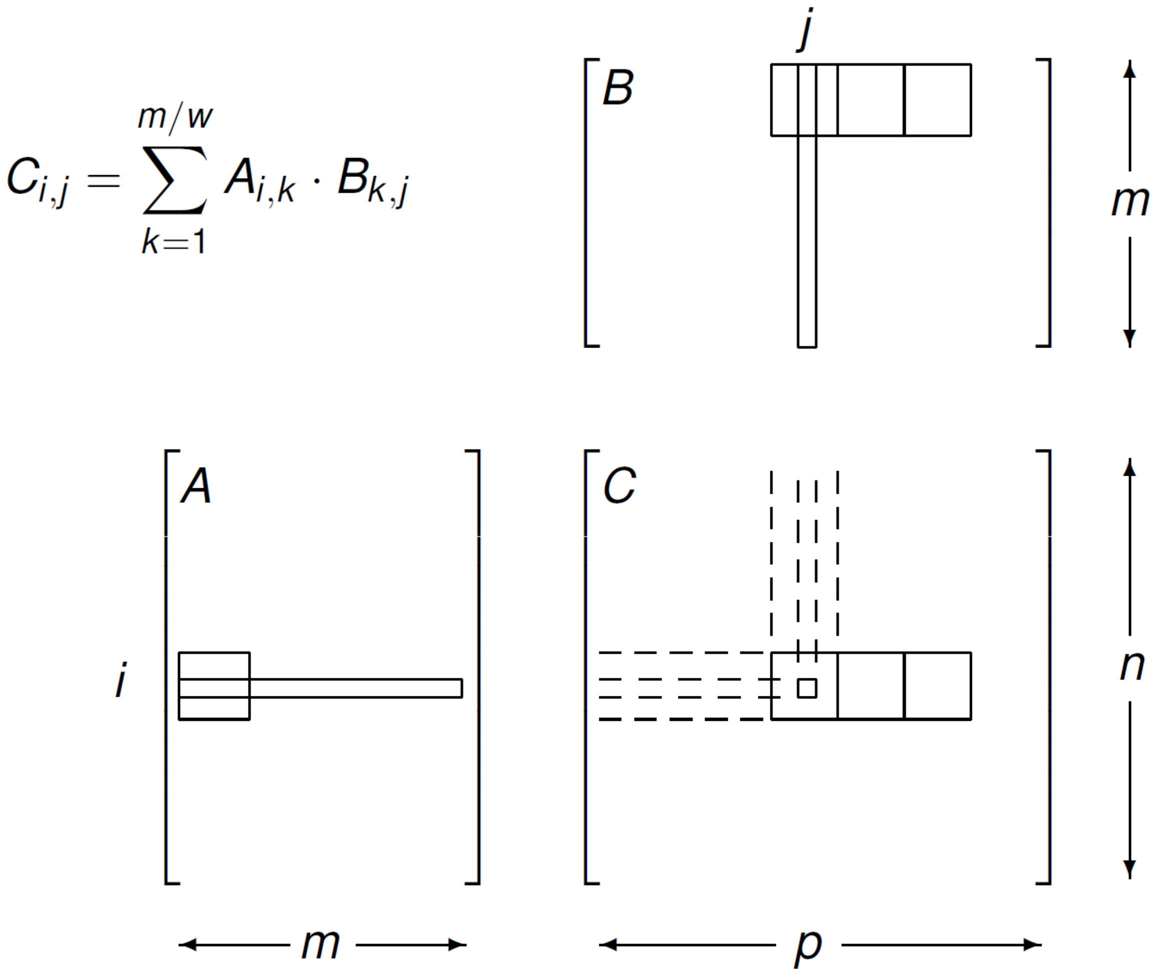


Fig. 6.50: In tiled matrix matrix multiplication, with thread coarsening, adjacent tiles of the second matrix are loaded by the same block of threads.

Each output tile is processed by a different block. The same input tiles for A are loaded for output tiles.

With thread coarsening, one block of threads loads one tile of A, and several vertically adjacent tiles of B. The *coarse factor* equals the number of tiles of B that are multiplied in the inner loop of the new kernel.

To clarify, pseudo code is below, for the tiled matrix multiplication, to multiply matrices `A` and `B` to make `C`:

```
block of threads loads a tile of A
block of threads loads a tile of B
block of threads updates a tile of C
```

With thread coarsening, the code is expanded into:

```
block of threads loads a tile of A
for k in 1, 2, ..., coarse factor do
    block of threads loads the next tile of B
    block of threads updates the next tile of C
```

The fourth edition of *Programming Massively Parallel Processors* by Wen-mei Hwu, David B. Kirk, and Izzat El Hajj contains explicit C code.

Thread coarsening is similar to the topic of granularity and while it is a powerful optimization, there are pitfalls:

1. Do not apply when not needed. Example: vector addition.
2. Thread coarsening may lead to underutilization. Coarsening factors depend on the type of a device and/or the specifics of the data that is processed.
3. Thread coarsening may reduce the occupancy. After thread coarsening, threads may use more registers and/or too much shared memory reducing the occupancy of the device.

It is important to know the performance bottleneck of a computation.

definition of performance bottleneck

The resource that limits the performance of a computation is a *performance bottleneck*.

If an optimization does not target the performance bottleneck, then the optimization attempt may even hurt performance. For example, ask the following questions: Is the computation compute or memory bound? Is the performance limited by occupancy? In answering those questions, understand the GPU architecture, and familiarize yourself with profiling tools.

At this point in the course, we have covered the fundamental topics of GPU acceleration.

6.9.5 Exercises

1. Consider a GPU with 2048 threads/SM, 32 blocks/SM, 64K registers/SM, and 96KB of shared memory/SM.
   - Kernel A uses 64 threads/block, 27 registers per thread, and 4KB of shared memory/SM.
   - Kernel B uses 256 threads/block, 31 registers per thread, and 8KB of shared memory/SM.

   Determine if the kernels achieve full occupancy. If not, specify the limiting factor(s).
2. Read the user guide of the compute visual profiler and perform a run on GPU code you wrote (of some previous exercise or your code for the third project). Explain the analysis of the kernel.
3. Redo the first interactions between resource limitations of this lecture using the specifications for compute capability 1.1.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index