Programming Shared Memory Computers part 1: introduction

Jan Verschelde[†]

University of Illinois at Chicago
Department of Mathematics, Statistics, and Computer Science
http://www.math.uic.edu/~jan
janv@uic.edu

https://pascal.math.uic.edu

Ada Europe 2021 Tutorial, 7 June 2021, online

[†]Supported by the National Science Foundation, grant DMS 1854513.

Outline

Programming Shared Memory Computers

1. Overview
   - parallel shared memory computing
   - processes, threads, and memories

2. Ideal Parallel Computations
   - no synchronization and no communication
   - hello tasks!
   - a package to estimate π

3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

parallel shared memory computing

Three main paradigms for parallel computing:

1. Distributed memory, e.g.: message passing.
2. Shared memory uses multithreading.
3. Accelerated, mainly with Graphics Processing Units.

This tutorial concerns multithreading. Our point of view is

$$\text{parallel} = \text{concurrent} + \text{speedup}$$

where

$$\text{speedup} = \frac{\text{sequential execution time}}{\text{parallel execution time}}.$$

Examples will run in a couple of seconds on one single thread.
Our goal: make the examples run faster.

computational setup

- The GNAT Community compiler is our main tool.
- A collection of compact code is posted on github:

`https://github.com/janverschelde/AdaEuropeTutorial`

- A 44-core computer is available at

  `https://pascal.math.uic.edu`

  after a sign up process which requires a valid email address.
- Launch a Terminal from the `new` pull down menu.

overview of the tutorial

The tutorial takes four hours:

1. ideal parallel computation,
2. load balancing with the work crew model,
3. pipelined computation,
4. synchronization with a shared data structure.

There are four folders, one for each part, at

`https://github.com/janverschelde/AdaEuropeTutorial`

and browsing through the code is needed to complete the slides.

Every hour has two parts:

1. a lecture on the main concepts;
2. suggested experimentation and exercises.

recommended reading

- Alan Burns and Andy Wellings:
  Concurrent and Real-Time Programming in Ada 2005.
  Cambridge University Press, 2007.
- John W. McCormick, Frank Singhoff, and Jerome Hugues:
  Building Parallel, Embedded, and Real-Time Applications with Ada. Cambridge University Press, 2011.
- John Barnes: *Ada 2012 Rationale.*
  The Language. The Standard Libraries.
  LNCS 8338, Springer, 2013.

Programming Shared Memory Computers

1. Overview
   - parallel shared memory computing
   - processes, threads, and memories

2. Ideal Parallel Computations
   - no synchronization and no communication
   - hello tasks!
   - a package to estimate π

3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

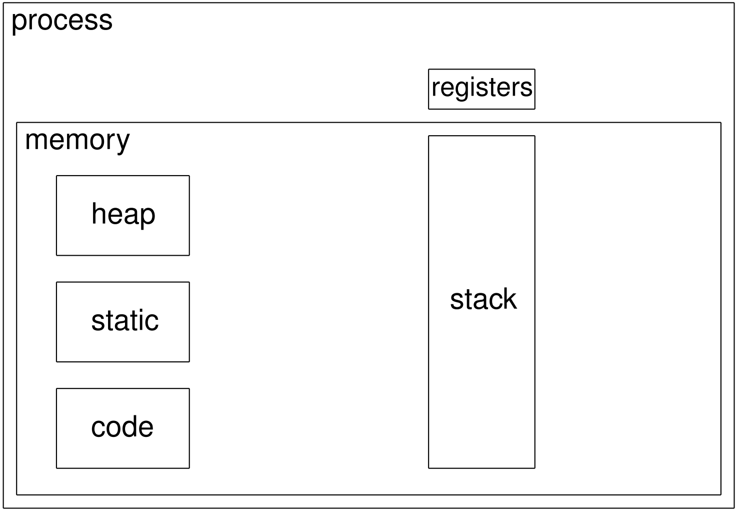
processes and threads

A thread is a single sequential flow within a process.

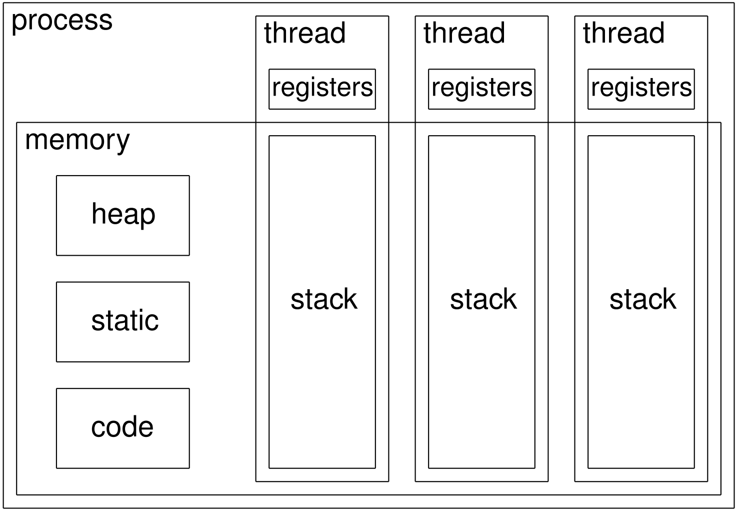
- Multiple threads within one process share heap storage, static storage, and code.
- Each thread has its own registers and stack.

Threads share the same single address space and synchronization is needed when threads access same memory locations.

single threaded process



multithreaded process



processes and threads

A thread is a single sequential flow within a process.

Multiple threads within one process share

- heap storage, for dynamic allocation and deallocation,
- static storage, fixed space,
- code.

Each thread has its own registers and stack.

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.

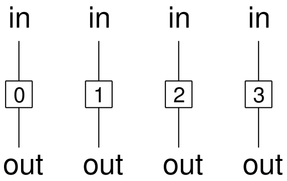
Threads share the same single address space and synchronization is needed when threads access same memory locations.

Programming Shared Memory Computers

1. Overview
   - parallel shared memory computing
   - processes, threads, and memories
2. Ideal Parallel Computations
   - no synchronization and no communication
   - hello tasks!
   - a package to estimate π
3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

ideal parallel computations

Suppose we have a disconnected computation graph for 4 tasks.



The main thread launches four tasks.

If the work load is well balanced so all tasks end at the same time, then we hope for an optimal speedup.

Monte Carlo simulations

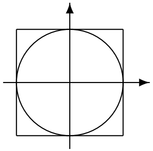
We simulate by

- 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:

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



Generate random uniformly distributed points with coordinates

$$(x, y) \in [0, 1] \times [0, 1].$$

We count a success when $x^2 + y^2 \leq 1$.

Programming Shared Memory Computers

1. Overview
   - parallel shared memory computing
   - processes, threads, and memories
2. Ideal Parallel Computations
   - no synchronization and no communication
   - hello tasks!
   - a package to estimate π
3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

hello tasks! *specification*

In a Terminal window, at the command prompt, type `./main`.

```
$ ./main
Number of CPUs : 88
Launching tasks ...
Task 1 says hello.
Task 2 says hello.
Task 3 says hello.
Task 4 says hello.
Launched 4 tasks.
$
```

The package `estimate_pi` exports `hello_tasks`:

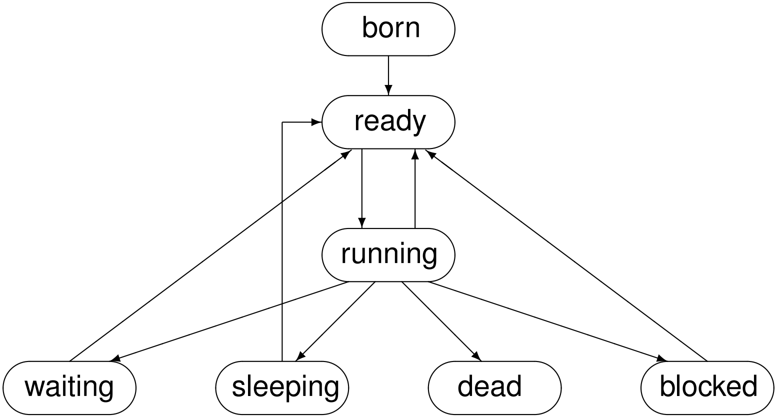
```
with number_types; use number_types;

package estimate_pi is

   procedure hello_tasks (p : in integer64 := 4);

   -- DESCRIPTION :
   --   A "Hello world!" version for launching p tasks.
```

hello tasks! *implementation* in `estimate_pi.adb`

```
procedure hello_tasks (p : in integer64 := 4) is

   task type worker (idnbr : integer64);

   task body worker is
   begin
      Text_IO.Put_Line ("Task" & integer64'image (idnbr)
                               & " says hello.");
   end worker;

   procedure launch (i : in integer64) is

      w : worker (i);

   begin
      if i < p then
         launch (i + 1);
      end if;
   end launch;

begin
   launch (1);
end hello_tasks;
```

the life cycle of a task

When do tasks start executing? When do tasks finish?
The life cycle of a task:



Why does `hello_tasks` work?

We consider three stages:

1. Creation: `task type worker (idnbr : integer64);`

   where the identification number `idnbr` is the task discriminant.

   The *i*-th task is created when the variable `w` in

   ```
   w : worker (i);
   ```

   is elaborated.

2. Tasks activate after the elaboration of the declarative part.

3. Tasks execute immediate after a successful activation, where execute means entering the ready state.

The parallelism happens because only the declarative part needs to be elaborated for a task to execute.

hello tasks! *in a main procedure*

The file `main.adb` contains the call to the procedure:

```
with estimate_pi;

procedure main is
begin
   estimate_pi.hello_tasks;
end main;
```

To obtain the code, type

```
git clone https://github.com/janverschelde/AdaEuropeTutorial
```

- to make the executable `main`, at the command prompt, type
  `gnatmake main`
- to remove all generate files generated, type `gnatclean main`

and these statements are also in a `makefile`.

Programming Shared Memory Computers

1. Overview
   - parallel shared memory computing
   - processes, threads, and memories

2. Ideal Parallel Computations
   - no synchronization and no communication
   - hello tasks!
   - a package to estimate π

3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

a package to estimate π

```
package estimate_pi is

-- DESCRIPTION :
--   Exports a parallel estimator for pi.

   freq1second : constant integer64 := 116_000_000;
   -- constant frequency so the wall clock time equals one second

   function incircle (x, y : float64) return Boolean;

   -- DESCRIPTION :
   --   Returns the outcome of (x*x + y*y <= 1.0).

   procedure count
     (seed      : in out integer64; sum : in out integer64;
      frequency : in     integer64 := freq1second);

   -- DESCRIPTION :
   --   Counts the number of times a pair of two random numbers (x,y
   --   made incircle(x,y) true, for i in range 1..frequency.
```

the parallel count procedure

```
procedure parallel_count
  (p         : in integer64;
   seeds     : in out vector; sums : in out vector;
   frequency : in integer64 := freq1second) is

   task type worker (idnbr : integer64);

   task body worker is
   begin
      count (seeds (idnbr), sums (idnbr), frequency);
   end worker;

   procedure launch (i : in integer64) is

      w : worker (i);

   begin
      if i < p then
         launch (i + 1);
      end if;
   end launch;

begin
   launch (1);
end parallel_count;
```

the command line arguments

To experience parallelism, we have two parameters:

1. a multiple of the frequency number `116_000_000`, and
2. the number of tasks,

passed at the command line, to measure the wall clock time.

```
$ time ./main 2 8
estimating 3.14159265358979E+00
estimate : 3.14157451293103E+00  error : 1.814E-05
frequency : 232000000  number of workers : 8

real    0m2.346s
user    0m17.892s
sys     0m0.004s
$
```

Programming Shared Memory Computers

1. Overview
   - parallel shared memory computing
   - processes, threads, and memories
2. Ideal Parallel Computations
   - no synchronization and no communication
   - hello tasks!
   - a package to estimate π
3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

compile, run, and evaluate

- Compile the code as `gnatmake -O3 -gnatp main` with full optimization, suppressing all checks.
- Compute the speedups with wall clock times.
- Are the speedups as expected?

Programming Shared Memory Computers

1. Overview
   - parallel shared memory computing
   - processes, threads, and memories
2. Ideal Parallel Computations
   - no synchronization and no communication
   - hello tasks!
   - a package to estimate π
3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

explorations and excursions

1. Consider applying a better random number generator.
2. Consider the estimation of the volume of the unit ball

$$\frac{\pi}{6} = \int_0^1 \int_0^1 \sqrt{1 - x^2 - y^2} \, dx \, dy$$

by generating random uniformly distributed points

$$(x, y, z) \in [0, 1] \times [0, 1] \times [0, 1].$$

Programming Shared Memory Computers part 2: the work crew model

Jan Verschelde[†]

University of Illinois at Chicago
Department of Mathematics, Statistics, and Computer Science
http://www.math.uic.edu/~jan
janv@uic.edu

https://pascal.math.uic.edu

Ada Europe 2021 Tutorial, 7 June 2021, online

[†]Supported by the National Science Foundation, grant DMS 1854513.

Outline

1. The Work Crew
   - collaborating tasks
   - an array of workers
2. Work Assignment
   - static: fix assignment before computations
   - dynamic: assign during the computations
   - semaphores to implement a critical section
3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

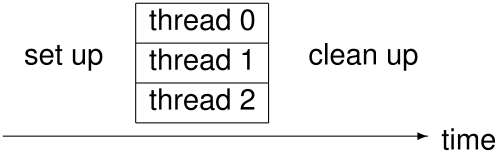
Programming Shared Memory Computers

1. The Work Crew
   - collaborating tasks
   - an array of workers
2. Work Assignment
   - static: fix assignment before computations
   - dynamic: assign during the computations
   - semaphores to implement a critical section
3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

the work crew model

Instead of the manager/worker model,
with threads we can apply a more collaborative model.

A computation performed by three 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.

Important for memory management:

- set up: all memory allocations, *before* the run,
- clean up: all memory deallocations, *after* the run.

Programming Shared Memory Computers

1. The Work Crew
   - collaborating tasks
   - an array of workers
2. Work Assignment
   - static: fix assignment before computations
   - dynamic: assign during the computations
   - semaphores to implement a critical section
3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

an array of workers

Running a *Hello tasks!* on an array of workers:

```
$ ./main
Task 1 says hello with id workers(4)_000000000F8B0E0.
Task 3 says hello with id workers(3)_000000000F87AA0.
Task 2 says hello with id workers(1)_000000000F80E20.
Task 4 says hello with id workers(2)_000000000F84460.
$
```

The last output is obtained via

```
     taskid : constant Ada.Task_Identification.Task_Id
            := Ada.Task_Identification.Current_Task;
```

which returns the identity of the task.

code for the *Hello tasks!*

```
procedure hello_tasks (p : in integer64 := 4) is

   task type worker;

   task body worker is

      idnbr : integer64;
      taskid : constant Ada.Task_Identification.Task_Id
             := Ada.Task_Identification.Current_Task;

   begin
      id_generator.get(idnbr);
      Text_IO.Put_Line ("Task" & integer64'image (idnbr)
                               & " says hello with id "
             & Ada.Task_Identification.Image(taskid) & ".");
   end worker;

   workers : array(1..p) of worker;

begin
   null;
end hello_tasks;
```

the id_generator assigns unique numbers

```
protected id_generator is

   procedure get ( id : out integer64 );
   -- returns a unique identification number

private
   next_id : integer64 := 1;
end id_generator;

protected body id_generator is
   procedure get ( id : out integer64 ) is
   begin
      id := next_id;
      next_id := next_id + 1;
   end get;
end id_generator;
```

Operations on data encapsulated by a protected object
are executed with mutually exclusive access.

Programming Shared Memory Computers

1. The Work Crew
   - collaborating tasks
   - an array of workers
2. Work Assignment
   - static: fix assignment before computations
   - dynamic: assign during the computations
   - semaphores to implement a critical section
3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

static work assignment

In a static work assignment, what a task computes is fixed in advance.

For example, to evaluate a definite integral, by four tasks:

$$\int_0^1 f(x)dx = \underbrace{\int_0^{1/4} f(x)dx}_{\text{task 1}} + \underbrace{\int_{1/4}^{1/2} f(x)dx}_{\text{task 2}} + \underbrace{\int_{1/2}^{3/4} f(x)dx}_{\text{task 3}} + \underbrace{\int_{3/4}^{1} f(x)dx.}_{\text{task 4}}$$

Definite integrals are approximated by the composite trapezoidal rule:

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

As our running example, we use $\displaystyle \frac{\pi}{4} = \int_0^1 \sqrt{1-x^2}\,dx$.

a recursive computation is more accurate

```
function recursive_rule
           ( f : access function ( x : float64 ) return float64;
             a,b : float64; n : integer64 ) return float64 is

   result,middle : float64;

begin
   if n = 1 then
      result := (b-a)*(f(a) + f(b))/2.0;
   else
      middle := a + (b-a)/2.0;
      result := recursive_rule(f,a,middle,n/2)
              + recursive_rule(f,middle,b,n/2);
   end if;
   return result;
end recursive_rule;
```

computing an approximation for π

```
package Double_Elementary_Functions is
   new Ada.Numerics.Generic_Elementary_Functions(float64);

function circle ( x : float64 ) return float64 is
begin
   return Double_Elementary_Functions.SQRT(1.0 - x**2);
end circle;

procedure run ( nsteps : in integer64 ) is

   approx : constant float64
          := recursive_rule(circle'access,0.0,1.0,nsteps);
   est4pi : constant float64 := 4.0*approx;
```

With `nsteps` equal to $2^{28} = 268,435,456$,
the approximation has an error of `2.673E-13`,
in about four seconds, 3.908 seconds wall clock time.

static work assignment

Typically, the granularity is coarse: #jobs = #tasks.

- What every task does can be hard coded.
- For the approximation of π, task *i* approximates

$$\int_{(i-1)h}^{i\,h} \sqrt{1 - x^2}\, dx, \quad h = \frac{1}{n},$$

  for *i* from 1 to *p*, where *p* is the number of tasks.
- The *i*-th task writes its approximation in the *i*-th location in a shared array.

encapsulation in a procedure

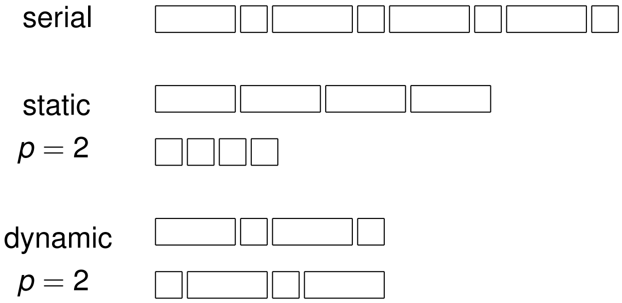
```
procedure run_workers ( p : in integer64; results : out vector ) is

   step : constant Float64 := 1.0/Float64(p);
   nsteps : constant integer64 := (2**28/p);

   task type worker;

   task body worker is

      myid : integer64;
      start,stop : float64;

   begin
      id_generator.get(myid);
      start := float64(myid-1)*step;
      stop := float64(myid)*step;
      results(myid) := recursive_rule(circle'access,start,stop,nsteps);
   end worker;

   workers : array(1..p) of worker;

begin
   null;
end run_workers;
```

Programming Shared Memory Computers

1. The Work Crew
   - collaborating tasks
   - an array of workers
2. Work Assignment
   - static: fix assignment before computations
   - dynamic: assign during the computations
   - semaphores to implement a critical section
3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

dynamic work assignment

Consider the scheduling of 8 jobs on 2 tasks:

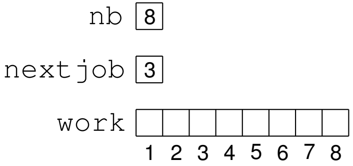


a job queue

The state of the job queue is defined by

1. the number of jobs: `nb`,
2. the index to the next job to be executed: `nextjob`,
3. the work to be done by every job: `work`.

Variables in a program can be values or references to values.



Each task grabs the index to the next job.

The update of `nextjob` happens in a *critical section*.

Programming Shared Memory Computers

using a semaphore

```
procedure test_job_queue (p : in integer64 := 4; n : in integer64 := 17) is

   nextjob : integer64 := 0;
   sem : Semaphore.Lock;

   task type worker;

   task body worker is

      idnbr,myjob : integer64;

   begin
      id_generator.get(idnbr);
      loop
         Semaphore.Request(sem);
         nextjob := nextjob + 1; myjob := nextjob;
         Semaphore.Release(sem);
         exit when (myjob > n);
         Text_IO.Put_Line ("Task" & integer64'image (idnbr) & " does job"
                                  & integer64'image (myjob) & ".");
      end loop;
   end worker;

   workers : array(1..p) of worker;

begin
   null;
end test_job_queue;
```

a binary semaphore to define a critical section

Following AdaCore Gem #81 (`www.adacore.com/gems/gem-81`), by Pat Rogers.

- The `GNAT.Semaphores` package defines semaphore abstractions, in terms of protected types.
- A binary semaphore is an abstraction for a flag.

```
        sem : Semaphore.Lock;
```

- The flag indicates whether or not the semaphore is available.

```
        Semaphore.Request(sem);
        nextjob := nextjob + 1; myjob := nextjob;
        Semaphore.Release(sem);
```

finer granularity of the jobs

Consider $n = 17$ jobs with $p = 4$ tasks.

- Instead of $2^{28}/p$ for the job size in the static work assignment, the job size is $2^{28}/n$ in the dynamic assignment.
- The j-th job is to approximate $\displaystyle\int_{(j-1)h}^{j\,h} \sqrt{1 - x^2} dx$, for $j = 1, 2, \ldots, n$, using $2^{28}/n$ for the number of steps.

We use the index i for a task and the index j for a job.

code for the worker task

```
task body worker is

   myid,myjob : integer64;
   start,stop : float64;

begin
   id_generator.get(myid);
   loop
      Semaphore.Request(sem);
      nextjob := nextjob + 1; myjob := nextjob;
      Semaphore.Release(sem);
      exit when (myjob > n);
      start := float64(myjob-1)*step;
      stop := float64(myjob)*step;
      results(myid) := results(myid)
         + recursive_rule(circle'access,start,stop,nsteps);
   end loop;
end worker;
```

a run with 17 jobs by 4 tasks

```
$ time ./main
The results :
1 : 1.97612345702311E-01
2 : 2.19402708891253E-01
3 : 1.90715989259139E-01
4 : 1.77667119544572E-01
Estimate for pi : 3.14159265358910E+00  error : 6.901E-13

real    0m0.681s
user    0m2.353s
sys     0m0.002s
$
```

Programming Shared Memory Computers

1. The Work Crew
   - collaborating tasks
   - an array of workers
2. Work Assignment
   - static: fix assignment before computations
   - dynamic: assign during the computations
   - semaphores to implement a critical section
3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

compile, run, and evaluate

- Compile the code as `gnatmake -O3 -gnatp main` with full optimization, suppressing all checks.
- Compute the speedups with wall clock times.
- Are the speedups as expected?

Programming Shared Memory Computers

1. The Work Crew
   - collaborating tasks
   - an array of workers

2. Work Assignment
   - static: fix assignment before computations
   - dynamic: assign during the computations
   - semaphores to implement a critical section

3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

explorations and excursions

- In both the static and dynamic work assignments, replace the computations with `delay x` statements, where `x` is a random number, between 1 and 6.
- Compare the wall clock time for the runs with the static and dynamic work assignments.
- Is the dynamic work assignment faster than the static one?

Programming Shared Memory Computers
part 3: pipelining

Jan Verschelde[†]

University of Illinois at Chicago
Department of Mathematics, Statistics, and Computer Science
http://www.math.uic.edu/~jan
janv@uic.edu

https://pascal.math.uic.edu

Ada Europe 2021 Tutorial, 7 June 2021, online

[†]Supported by the National Science Foundation, grant DMS 1854513.

Outline

Programming Shared Memory Computers

1. Pipelining
   - the producer/consumer model
   - a pyramid of workers
   - allocating tasks to processors
2. Pipelined Computations
   - linear pipelines
   - pyramid pipelines
3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

the producer/consumer model

A schematic of a 2-stage pipeline is shown below:

$$\begin{tikzcd}
\text{input} \arrow[r] & \boxed{P} \arrow[r] & \boxed{C} \arrow[r] & \text{output}
\end{tikzcd}$$

The output of P is the input for C:

- P is the producer,
- C is the consumer.

The producer/consumer pipeline is a model for heterogeneous parallelism, as P and C can do different types of computations.

It is also a model of synchronized computations:

- the consumer must wait if its input is empty,
- the producer must wait if its output if full.

the method of delegation

Keep some of the work, pass the rest to others in the pipeline.

A linear pipeline:

$$\begin{tikzcd}
\boxed{P_0} \arrow[r] & \boxed{P_1} \arrow[r] & \boxed{P_2} \arrow[r] & \boxed{P_3} \arrow[r] & \boxed{P_4} \arrow[r] & \boxed{P_5} \arrow[r] & \boxed{P_6}
\end{tikzcd}$$

A pyramid pipeline:

$$\begin{tikzcd}
 & & & \boxed{P_0} \arrow[dll] \arrow[drr] & & & \\
 & \boxed{P_1} \arrow[dl] \arrow[dr] & & & & \boxed{P_2} \arrow[dl] \arrow[dr] & \\
\boxed{P_3} & & \boxed{P_4} & & \boxed{P_5} & & \boxed{P_6}
\end{tikzcd}$$

Programming Shared Memory Computers

1. Pipelining
   - the producer/consumer model
   - a pyramid of workers
   - allocating tasks to processors
2. Pipelined Computations
   - linear pipelines
   - pyramid pipelines
3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

a pyramid of workers

Running a fan out launch of tasks.

```
$ ./main
Launching tasks ...
Hello from worker 1 at level 0 with number 1.
Hello from worker 1 at level 1 with number 2.
Hello from worker 2 at level 1 with number 3.
Hello from worker 1 at level 2 with number 4.
Hello from worker 2 at level 2 with number 5.
Hello from worker 3 at level 2 with number 6.
Hello from worker 4 at level 2 with number 7.
Launched 7 workers.
$
```

Every worker has a first name and a last name:

1. the last name is the level in the pyramid,
2. the first name is the rank at the level.

a leveled id generator

```
protected leveled_id_generator is

   procedure get ( lvl : in integer64; id : out integer64 );
   -- returns a unique identification number,
   -- within the current level lvl

private
   next_id : integer64 := 1;
   current_lvl : integer64 := -1;
end leveled_id_generator;

protected body leveled_id_generator is
   procedure get ( lvl : in integer64; id : out integer64 ) is
   begin
      if lvl /= current_lvl then -- reset the id counter
         current_lvl := lvl;
         next_id := 1;
      end if;
      id := next_id;
      next_id := next_id + 1;
   end get;
end leveled_id_generator;
```

code for a fan out launch

```
procedure launch (level : integer64) is

   task type worker;

   task body worker is ...

   nbrtasks : constant integer64 := 2**integer(level);
   workers : array(1..nbrtasks) of worker;

begin
   if level < pwr-1 then
      launch(level+1);
   end if;
end launch;
```

The `launch` combines

1. the recursion to activate a sequence of tasks, and
2. an array of increasing number of workers.

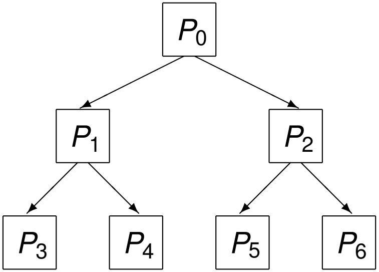
the body of the worker

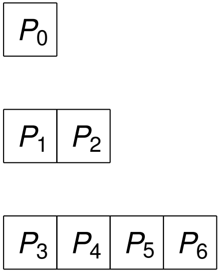
```
task body worker is

   i,n : integer64;

begin
   leveled_id_generator.get(level,i);
   n := i;
   for j in 0..level-1 loop
      n := n + 2**integer(j);
   end loop;
   text_io.put_line("Hello from worker" & integer64'image(i)
                  & " at level"         & integer64'image(level)
                  & " with number"      & integer64'image(n)
                  & ".");
end worker;
```

Worker i at level ℓ is uniquely identified by the number $n = i + \sum_{j=0}^{\ell} 2^j$.

how it works

The workers in the pyramid pipeline are in k arrays:





In this way, all $2^k - 1$ workers can compute in parallel.

Programming Shared Memory Computers

1. Pipelining
   - the producer/consumer model
   - a pyramid of workers
   - allocating tasks to processors
2. Pipelined Computations
   - linear pipelines
   - pyramid pipelines
3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

allocating tasks to processors

We can allocate all tasks at the same level
to the same processor:

```
procedure launch (level : integer64) is

   task type worker
      with CPU => System.Multiprocessors.CPU_Range(level);
```

As the number of workers increases with the level,
workers down the pyramid have to share the processors more.

Or equivalently, workers towards the top of the pyramid have more available processing power.

Programming Shared Memory Computers

a linear pipeline

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

What every task does is determined by its identification number:

1. For the approximation of π, task i approximates

$$\int_{(i-1)h}^{i\,h} \sqrt{1 - x^2}\, dx, \quad h = \frac{1}{n},$$

   for i from 1 to p, where p is the number of tasks.

2. The i-th task writes its approximation in the i-th location in a shared array.

code for the worker

Let p be the total number of tasks:

```
step : constant float64 := 1.0/float64(p);
nsteps : constant integer64 := (2**28/p);

use estimate_pi;
use trapezoidal_pi;

task type worker (i:integer64);

task body worker is

   a : constant float64 := float64(i-1)*step;
   b : constant float64 := float64(i)*step;

begin
   results(i) := recursive_rule(circle'access,a,b,nsteps);
end worker;
```

running a pipe of 64 workers

```
$ time ./main
Launching tasks ...
Launched 64 workers.
The results :
1 : 1.56243641938373E-02
2 : 1.56205487978238E-02
...
64 : 1.83710203264093E-03
Estimate for pi : 3.14159265358953E+00  error : 2.660E-13

real    0m0.219s
user    0m10.645s
sys     0m0.012s
$
```

Programming Shared Memory Computers

pyramid pipelines

In the pyramid pipeline:

1. worker i at level ℓ is uniquely identified by the number

$$n = i + \sum_{j=0}^{\ell} 2^j$$

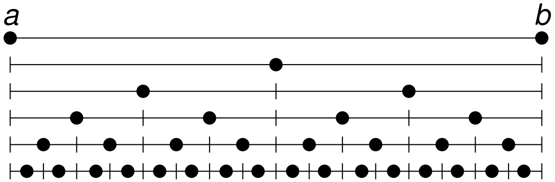
2. therefore, this worker with number n approximates

$$\int_{(n-1)h}^{n\,h} \sqrt{1 - x^2}\,dx.$$

But there is a better way to organize the workers. . .

computing new function evaluations

We can consider the composite trapezoidal rule as a sequence of rules which use more function evaluations in each step.



The number of new function evaluations is $1, 2, 4, 8, 16$.

At each level we compute more function evaluations.

$$T_{2n} = \frac{1}{2} T_n + \frac{b-a}{2n} \sum_{i=1}^{n-1} f\left(a + \frac{b-a}{2n} + i\frac{b-a}{n} \right)$$

error estimates with parallel computations

There is parallelism in the reformulated trapezoidal rule:

$$T_{2n} = \frac{1}{2} \underbrace{T_n}_{\text{task 1}} + \underbrace{\frac{b-a}{2n} \sum_{i=1}^{n-1} f\left(a + \frac{b-a}{2n} + i\frac{b-a}{n}\right)}_{\text{task 2}}$$

Application: $|T_{2n} - T_n|$ is an estimate for the error.

The reorganization of the composite trapezoidal rule into a pyramid pipeline provides information on the quality of the computations.

Programming Shared Memory Computers

1. Pipelining
   - the producer/consumer model
   - a pyramid of workers
   - allocating tasks to processors
2. Pipelined Computations
   - linear pipelines
   - pyramid pipelines
3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

compile, run, and evaluate

- Compile the code as `gnatmake -O3 -gnatp main` with full optimization, suppressing all checks.
- Compute the speedups with wall clock times.
- Are the speedups as expected?

Programming Shared Memory Computers

explorations and excursions

Using `pragma priority`, one can give priority to tasks.

In pyramid pipeline,
consider giving tasks on separate levels different priority values,
or, alternatively, give different priorities to tasks at the same level.

Programming Shared Memory Computers part 4: synchronization

Jan Verschelde[†]

University of Illinois at Chicago
Department of Mathematics, Statistics, and Computer Science
`http://www.math.uic.edu/~jan`
`janv@uic.edu`

`https://pascal.math.uic.edu`

Ada Europe 2021 Tutorial, 7 June 2021, online

[†]Supported by the National Science Foundation, grant DMS 1854513.

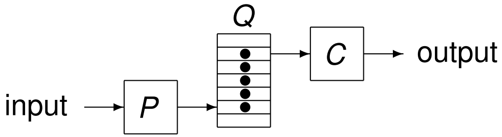
Outline

Programming Shared Memory Computers

1. Synchronization
   - the buffered producer/consumer model
   - running a producer/consumer simulation
2. Adding Buffered Function Values
   - a 2-stage pipeline for π
   - arrays of evaluators and adders
3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

the buffered producer/consumer model

A schematic of a 2-stage pipeline, with a buffer, is shown below:



The output of *P* is the input for *C*:

- *P* is the producer, *C* is the consumer,
- *Q* is the queue, acts as buffer between *P* and *C*.

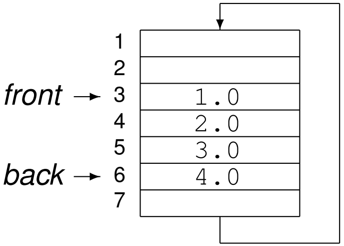
The producer/consumer pipeline is a model for heterogeneous parallelism, as *P* and *C* can do different types of computations, *at a different pace*.

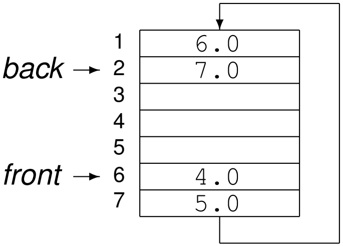
- The consumer must wait if its input is empty,
- the producer must wait if its output if full.

The buffer *Q* helps to synchronize the computations:

a circular buffer of fixed size

Consider an array to represent a circular buffer.





Three important numbers define the state of the queue:

- The index *front* points to the front of the queue.
- The *back* points to the end of the queue.
- The index calculation happens modulo the *size* of the queue.

The queue may become empty or may become full.

operations and exceptions

The `Float64_Queue` defines four operations and two exceptions:

1. Initialize the queue with the `size`, allocating space.
2. Push a number to the back of the queue, with `push_back`.
   The exception `queue_full` will be raised is the queue is full.
3. Pop a number from the front of the queue, with `pop_front`.
   The exception `queue_empty` will be raised if the queue is empty.
4. Deallocate the space occupied by the queue.

In the body of the package, the update of the `front` and `back` happens in a critical section, using a semaphore.

- Producers push to the back of the queue.
- Consumers pop from the front of the queue.

definition of the `pop_front` function

```
function pop_front return Float64 is

   result : Float64 := 0.0;

begin
   if front = 0 then                    -- zero front means empty queue
      raise queue_empty;
   else
      Semaphore.Request(sem);           -- enter the critical section
      result := data(front);
      if front = back then              -- we popped the last item
         front := 0;                    -- set the front to zero
      else
         front := front + 1;
         if front > data'last           -- circular update of front
          then front := 1;
         end if;
         if full_queue                  -- the queue is no longer full
          then full_queue := false;
         end if;
      end if;
      Semaphore.Release(sem);           -- leave the critical section
   end if;
   return result;
end pop_front;
```

Programming Shared Memory Computers

1. Synchronization
   - the buffered producer/consumer model
   - running a producer/consumer simulation
2. Adding Buffered Function Values
   - a 2-stage pipeline for π
   - arrays of evaluators and adders
3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

running a producer/consumer simulation

Consider a simulation of n pipeline cycles.

1. The producer pushes the numbers $1, 2, \ldots, n$ to the queue.
2. The consumer adds the popped numbers.

After a correct run, the total sum should be $n(n+1)/2$.

```
procedure test_producer_consumer
             ( cycles_number : in integer64 := 5;
               queue_size    : in integer64 := 2;
               producer_pace : in duration  := 1.0;
               consumer_pace : in duration  := 1.0 );
```

1. `cycles_number` is the number of producer/consumer steps,
2. `queue_size` is the size of the buffer queue,
3. `producer_pace` is the time needed to produce a new number,
4. `consumer_pace` is the time needed to consume a new number.

algorithms for the producer and consumer

The give and take interaction is symmetrical:

- The producer and consumer do the same number of steps.
- A busy-wait loop is executed when pushing or popping fails.
- The interaction is correct because of the critical sections in the `pop_front` and `push_back` operations.

the producer code

```
task body producer is

   nbr : float64 := 0.0;
   fail : boolean;

begin
   for k in 1..cycles_number loop
      delay producer_pace;
      nbr := nbr + 1.0;
      loop
         fail := false;
         declare
         begin
            float64_queue.push_back(nbr);
         exception
            when float64_queue.queue_full => fail := true;
         end;
         exit when not fail;
         delay producer_pace/2.0;
      end loop;
   end loop;
end producer;
```

the consumer code

```
task body consumer is

   nbr : float64;
   fail : boolean;

begin
   for k in 1..cycles_number loop
      loop
         fail := false;
         declare
         begin
            nbr := float64_queue.pop_front;
         exception
            when float64_queue.queue_empty => fail := true;
         end;
         exit when not fail;
         delay consumer_pace/2.0;
      end loop;
      delay consumer_pace;
      result := result + nbr;
   end loop;
end consumer;
```

many producers and consumers

Modifications to run with *p* producers and *c* consumers:

- Two arrays of tasks, one array of producers, and another array of consumers.
- Two separate identification number generators, one to identify producers, the other for the consumers.
- For *n* cycles and *p* producers, the *i*-th producer pushes

  $$i, i + p, i + 2p, \ldots, i + (n - 1)p,$$

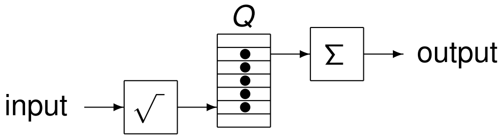
  so the entire sequence is $1, 2, \ldots, np$.
- The *i*-th consumer accumulates the sum of popped numbers at the *i*-th spot in an array of numbers.
- *How to stop the consumers?*

Programming Shared Memory Computers

1. Synchronization
   - the buffered producer/consumer model
   - running a producer/consumer simulation
2. Adding Buffered Function Values
   - a 2-stage pipeline for π
   - arrays of evaluators and adders
3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

a 2-stage pipeline for π

To approximate $\frac{\pi}{4} = \int_0^1 \sqrt{1-x^2}\,dx$ with the composite trapezoidal rule, we apply:

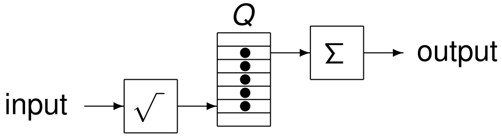


The two stages in this pipeline are

1. $\sqrt{}$ is the evaluation of $\sqrt{1-x^2}$ at many points; and
2. Σ is the summation of the function values.

a space-time diagram

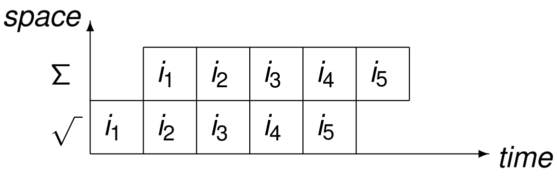
In the 2-stage pipeline schematic



each • represent a stored sequence of function values.

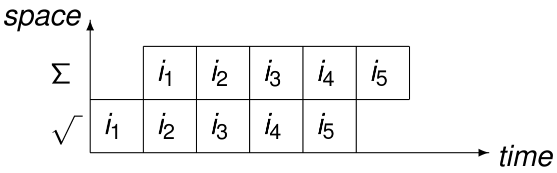
Each sequence is defined by an input range and output integral.

The space-time diagram for five instances is below:



definition of the inputs and outputs

The five instances in the space-time diagram



are for $\displaystyle\frac{\pi}{4} = \int_0^1 \sqrt{1 - x^2}\, dx$ the five subintervals of $[0, 1]$:

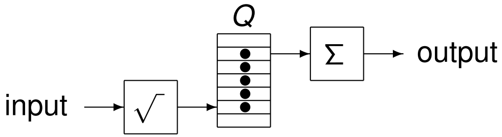
$$i_1 = \left[0, \frac{1}{5}\right], \quad i_2 = \left[\frac{1}{5}, \frac{2}{5}\right], \quad i_3 = \left[\frac{2}{5}, \frac{3}{5}\right], \quad i_4 = \left[\frac{3}{5}, \frac{4}{5}\right], \quad i_5 = \left[\frac{4}{5}, 1\right].$$

and the outputs are the integrals over those intervals.

Assuming evaluation and summation take the same amount of time, the theoretical speedup is $10/6 \approx 1.67$.

shared data structures

The queue Q stores data represented by • in



Each • represents a sequence of function values.

Instead of copying arrays of 64-bit floats,

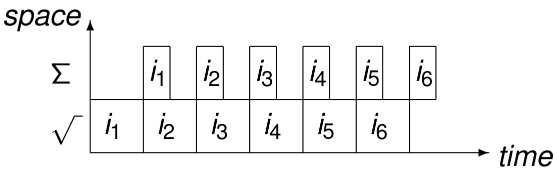
1. pushing an index to the queue means: values are ready, and
2. each index popped from the queue is the input for a summation.

In the data structure for Q,
each • is an index to an array of 64-bit float arrays.

Programming Shared Memory Computers

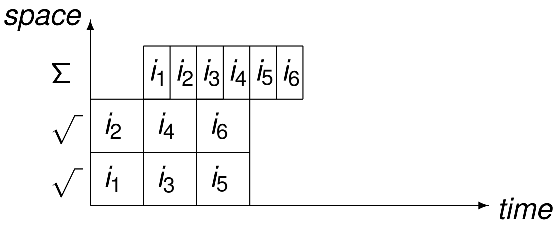
two evaluators and one adder

As the addition is likely to be much faster than evaluation, the space-time diagram may look as follows:



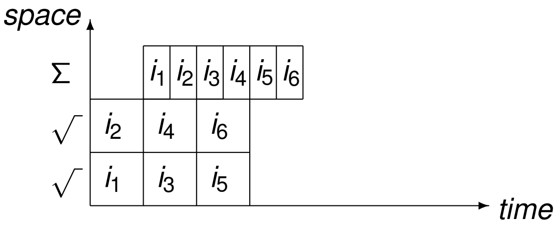
space
Σ
i_1 i_2 i_3 i_4 i_5 i_6
$\sqrt{}$
i_1 i_2 i_3 i_4 i_5 i_6
time

With two evaluators:



arrays of evaluators and adders

If the adders are twice as fast than the evaluators:



Alternatively, two adders work as fast as one evaluator.

If p adders are as fast as q evaluators,
then use an array of q adders and an array of p evaluators.

Programming Shared Memory Computers

1 Synchronization
- the buffered producer/consumer model
- running a producer/consumer simulation

2 Adding Buffered Function Values
- a 2-stage pipeline for π
- arrays of evaluators and adders

3 Exercises
- compile, run, and evaluate
- explorations and excursions

compile, run, and evaluate

- Compile the code as `gnatmake -O3 -gnatp main` with full optimization, suppressing all checks.
- Compute the speedups with wall clock times.
- Are the speedups as expected?

Programming Shared Memory Computers

1. Synchronization
   - the buffered producer/consumer model
   - running a producer/consumer simulation

2. Adding Buffered Function Values
   - a 2-stage pipeline for π
   - arrays of evaluators and adders

3. Exercises
   - compile, run, and evaluate
   - explorations and excursions

explorations and excursions

Instead of the synchronization with a queue,
consider the direct synchronization with Ada's rendezvous.