Work Stealing

1. Threading and Tasking
   - mapping jobs to processors
   - task based programming and work stealing
   - combining static and dynamic work assignment
2. Multithreading in Python
   - Numba
   - Parsl
3. The Intel Threading Building Blocks
   - our first program with TBB
   - using the `parallel_for`
   - using the `parallel_reduce`

MCS 572 Lecture 15
Introduction to Supercomputing
Jan Verschelde, 30 September 2024

Work Stealing

1. Threading and Tasking
   - mapping jobs to processors
   - task based programming and work stealing
   - combining static and dynamic work assignment
2. Multithreading in Python
   - Numba
   - Parsl
3. The Intel Threading Building Blocks
   - our first program with TBB
   - using the `parallel_for`
   - using the `parallel_reduce`

mapping jobs to processors

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.

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.

Work Stealing

1. Threading and Tasking
   - mapping jobs to processors
   - task based programming and work stealing
   - combining static and dynamic work assignment
2. Multithreading in Python
   - Numba
   - Parsl
3. The Intel Threading Building Blocks
   - our first program with TBB
   - using the `parallel_for`
   - using the `parallel_reduce`

task based programming and work stealing

Tasks are much lighter than threads.

- starting and terminating a task is much times 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.

Work Stealing

1. Threading and Tasking
   - mapping jobs to processors
   - task based programming and work stealing
   - combining static and dynamic work assignment
2. Multithreading in Python
   - Numba
   - Parsl
3. The Intel Threading Building Blocks
   - our first program with TBB
   - using the `parallel_for`
   - using the `parallel_reduce`

work stealing as hybrid work assignment

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.

setup of the Julia program

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 $j \neq i$.

making the job queues

```
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
```

running the program

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.

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.

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 an output

```
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.

discussion

- 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.

Work Stealing

1. Threading and Tasking
   - mapping jobs to processors
   - task based programming and work stealing
   - combining static and dynamic work assignment
2. Multithreading in Python
   - Numba
   - Parsl
3. The Intel Threading Building Blocks
   - our first program with TBB
   - using the `parallel_for`
   - using the `parallel_reduce`

Numba for multithreading in Python

- 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 at `https://github.com/numba/numba`.
- To use, do `pip install numba`.
- The example on the next slide works on Windows.

an example from the wikipedia page

```
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)
```

Work Stealing

1. Threading and Tasking
   - mapping jobs to processors
   - task based programming and work stealing
   - combining static and dynamic work assignment
2. Multithreading in Python
   - Numba
   - Parsl
3. The Intel Threading Building Blocks
   - our first program with TBB
   - using the `parallel_for`
   - using the `parallel_reduce`

Parsl: 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 on the next slide was executed on WSL, Window Subsystem for Linux, Ubuntu 22.04.

an example from the parsl user guide

```
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())
```

Work Stealing

1. Threading and Tasking
   - mapping jobs to processors
   - task based programming and work stealing
   - combining static and dynamic work assignment
2. Multithreading in Python
   - Numba
   - Parsl
3. The Intel Threading Building Blocks
   - our first program with TBB
   - using the `parallel_for`
   - using the `parallel_reduce`

the Intel Threading Building Blocks

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++).

Open Source, download at
`http://threadingbuildingblocks.org/`
which redirects to a `github` page.

oneTBB and oneAPI

TBB is part of oneAPI which aims to offer
one single programming model for CPU, GPU, FPGA accelerators.

On M1 MacBook, the openAPI installs with brew, via
`brew install tbb` and `brew install onedpl`
where `onedpl` is formerly known as `parallelstl`
a C++ Standard library algorithms with support for execution policies.

To use TBB, look on your system for the location of header files and the libraries.

saying `hello`

```
#include <cstdio>
#include <tbb.h>
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);
      }
};
```

A `class` in C++ is a like a `struct` in C
for holding data attributes and functions (called methods).

the main function

```
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`.

Work Stealing

1. Threading and Tasking
   - mapping jobs to processors
   - task based programming and work stealing
   - combining static and dynamic work assignment
2. Multithreading in Python
   - Numba
   - Parsl
3. The Intel Threading Building Blocks
   - our first program with TBB
   - using the `parallel_for`
   - using the `parallel_reduce`

raising complex numbers to a large power

Consider the following problem:

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 $k = 1, 2, \ldots, n$.

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 declare

```
#include <complex>

using namespace std;

typedef complex<double> dcmplx;
```

random complex doubles

```
#include <cstdlib>
#include <cmath>

dcmplx random_dcmplx ( void );
// generates a random complex number
// on the complex unit circle
```

We compute $e^{2\pi i\theta} = \cos(2\pi\theta) + i\sin(2\pi\theta)$, for random $\theta \in [0, 1]$:

```
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;
}
```

writing arrays

```
#include <iostream>
#include <iomanip>

void write_numbers ( int n, dcmplx *x );
// writes the array of n doubles in x
```

Observe the local declaration `int i` in the `for` loop,
the scientific formatting, and the methods `real()` and `imag()`:

```
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";
}
```

computing powers

```
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
```

The plain `for(int j` loop avoids repeated squaring:

```
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;
   }
}
```

command line arguments

```
$ ./powers_serial
how many numbers ? 2
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)

$ ./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 ./powers_serial 1000 1000000 0

real    0m20.139s
user    0m20.101s
sys     0m0.000s
```

the main program

```
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);
```

the main program continued

```
   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;
}
```

the speedup

```
$ time ./powers_serial 1000 1000000 0

real    0m20.139s
user    0m20.101s
sys     0m0.000s

$ time ./powers_tbb 1000 1000000 0

real    0m1.191s
user    0m35.170s
sys     0m0.043s
```

The speedup: $\frac{20.139}{1.191} = 16.909$ on two 8-core CPUs.

on M1 MacBook Air

```
% /usr/bin/time /tmp/powers_serial 1000 1000000 0
      16.23 real        13.15 user         0.02 sys

% /usr/bin/time /tmp/powers_tbb 1000 1000000 0
       2.30 real        16.96 user         0.04 sys
```

the class `ComputePowers`

```
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;
         }
      }
};
```

tbb/blocked_range.h

```
#include "tbb/blocked_range.h"

template<typename Value> class blocked_range
```

A `blocked_range` represents a half open range $[i, j)$ that can be recursively split.

```
void operator()
   ( const blocked_range<size_t>& r ) const
{
   for(size_t i=r.begin(); i!=r.end(); ++i)
   {
```

calling the `parallel_for`

```
#include "tbb/tbb.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"

using namespace tbb;
```

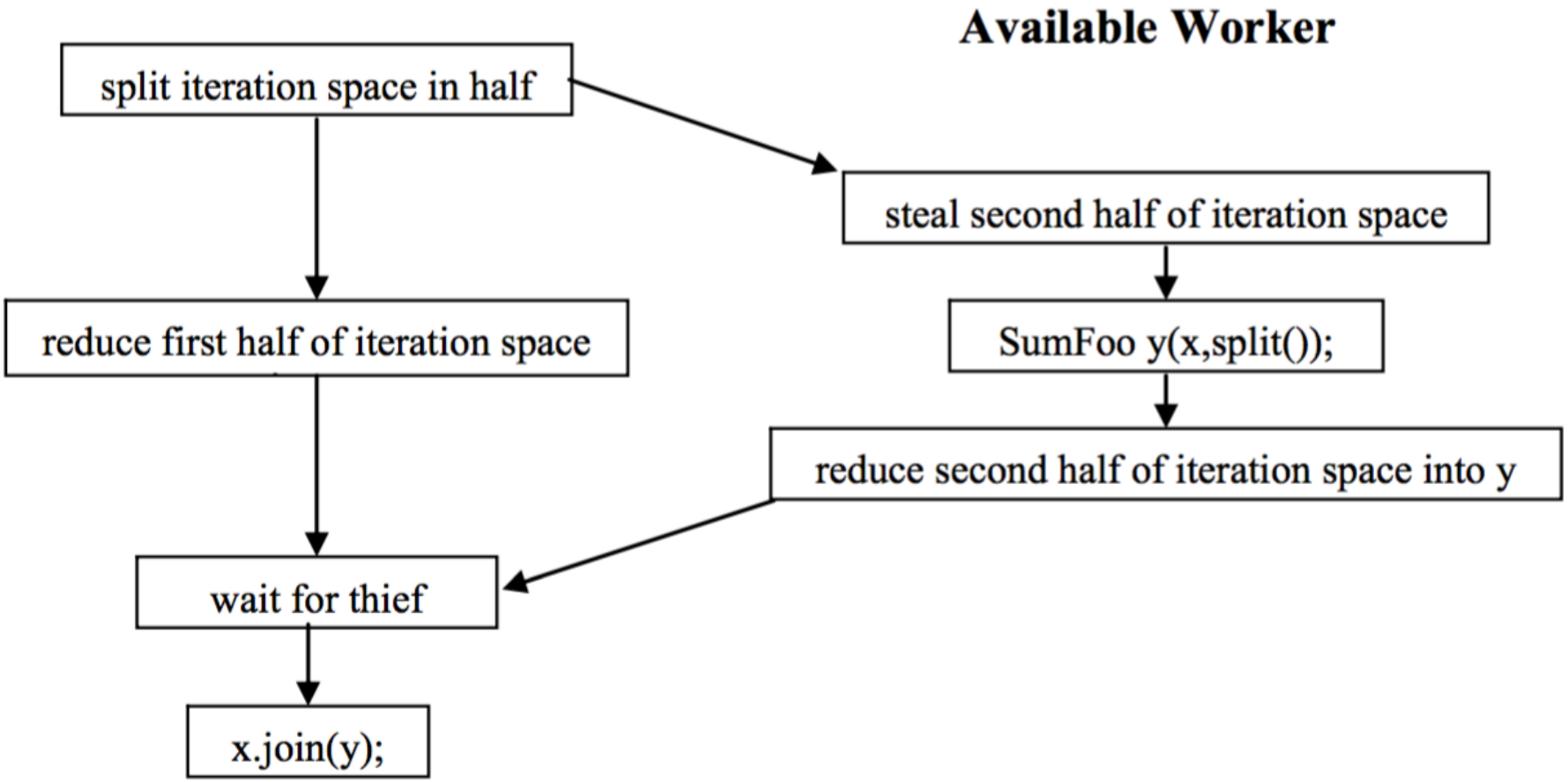
One line changes in the main program:

```
parallel_for(blocked_range<size_t>(0,dim),
             ComputePowers(r,deg,s));
```

Work Stealing

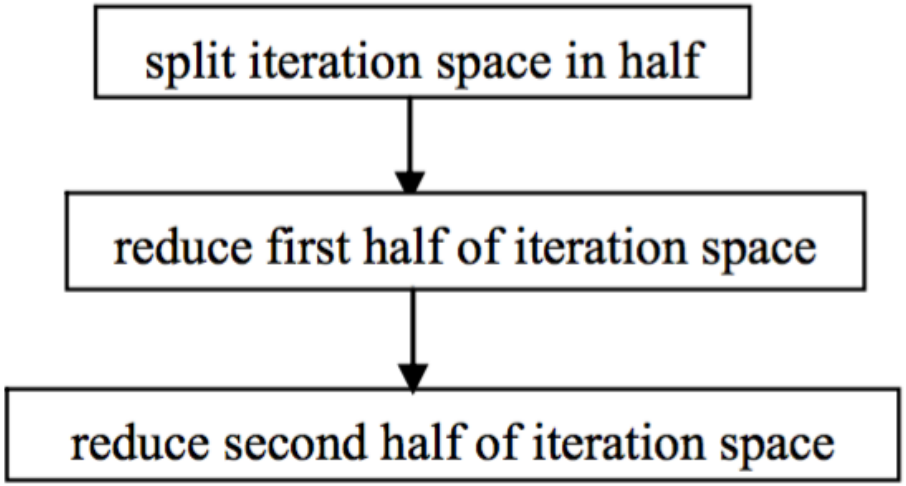
1. Threading and Tasking
   - mapping jobs to processors
   - task based programming and work stealing
   - combining static and dynamic work assignment
2. Multithreading in Python
   - Numba
   - Parsl
3. The Intel Threading Building Blocks
   - our first program with TBB
   - using the `parallel_for`
   - using the `parallel_reduce`

an application of work stealing



from the Intel Threading Building Blocks Tutorial

what if no worker is available?



No Available Worker

from the Intel Threading Building Blocks Tutorial

the class SumIntegers

```
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;
      }
```

split and join methods

```
      // 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;
}
```

code in the main program

```
int *d;
d = (int*)calloc(n,sizeof(int));
for(int i=0; i<n; i++) d[i] = i+1;

int s = ParallelSum(d,n);
```

Bibliography

- **Intel Threading Building Blocks. Tutorial.**
  Available online via http://www.intel.com.
- 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.
- 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.

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 ith 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.