Introduction to Supercomputing

Release 0.7.8

Jan Verschelde

Nov 23, 2016

Contents

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.

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.
- $\mathrm{R}_{\max}$ maximal performance achieved on the LINPACK benchmark (solving a dense linear system) for problem size $\mathrm{N}_{\max}$, measured in Gflop/s.
- $\mathrm{R}_{\mathrm{peak}}$ theoretical peak performance measured in Gflop/s.
- Power total power consumed by the system.

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.

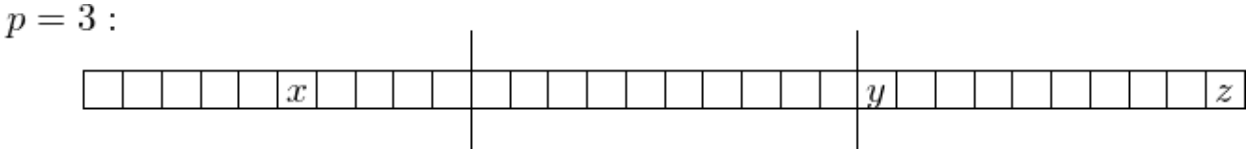


Fig. 1.1: A search illustrates superlinear speedup.

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.

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. 1.2.

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 $\frac{1}{R + \frac{1-R}{p}}$.

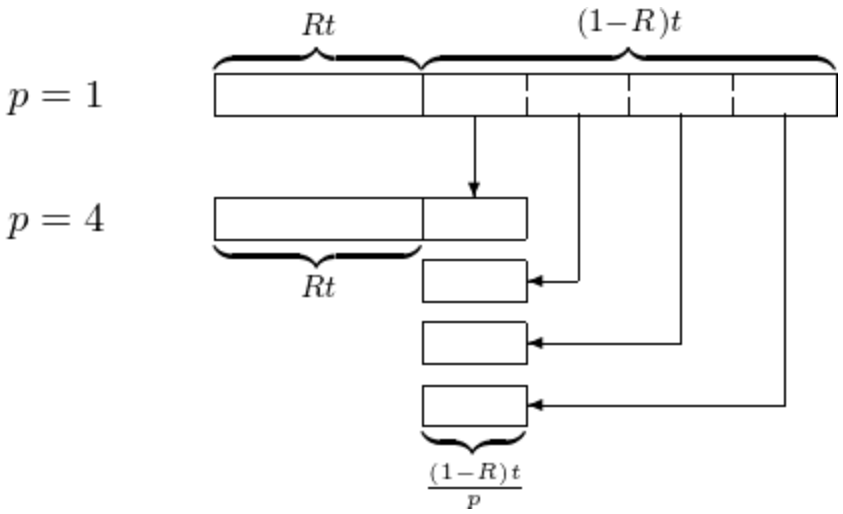


Fig. 1.2: Illustration of Amdahl's Law.

Corollary of Amdahl's Law

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

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.

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 $p + (1-p)s$.

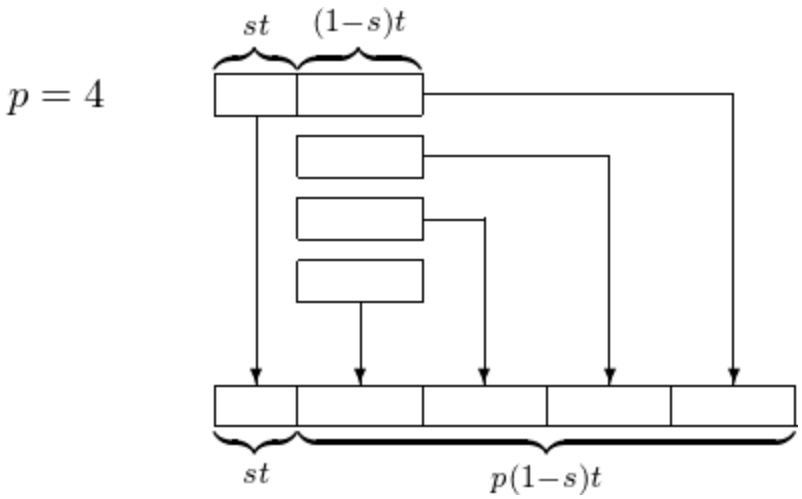


Fig. 1.3: Illustration of Gustafson's Law.

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. 1.4.

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.

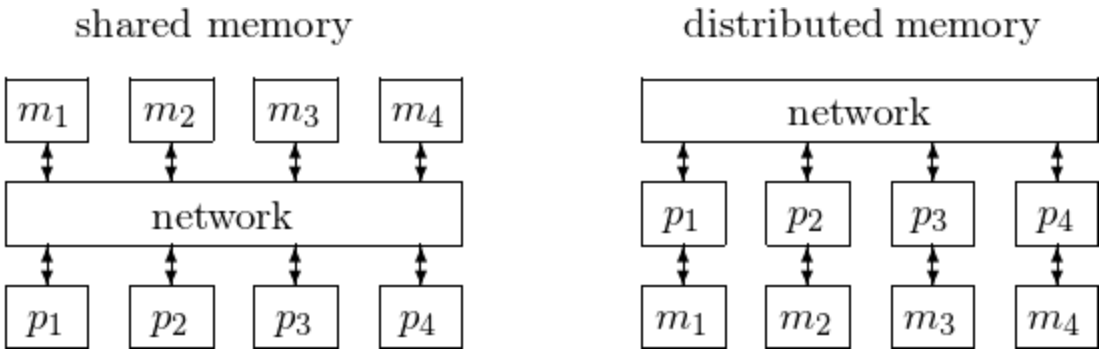


Fig. 1.4: A shared memory multicomputer has one single address space, accessible to every processor. In a distributed memory multicomputer, every processor has its own memory accessible via messages through that processor. Most nodes in a parallel computers have multiple cores.

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.

$$\text{total time} = \text{computation time} + \underbrace{\text{communication time}}_{\text{overhead}}$$

Because we want to reduce the overhead, the

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

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 $\to \infty$, 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. 1.5. A matrix and torus of 16 nodes is shown in Fig. 1.6.

Fig. 1.5: An array and ring of 4 notes.

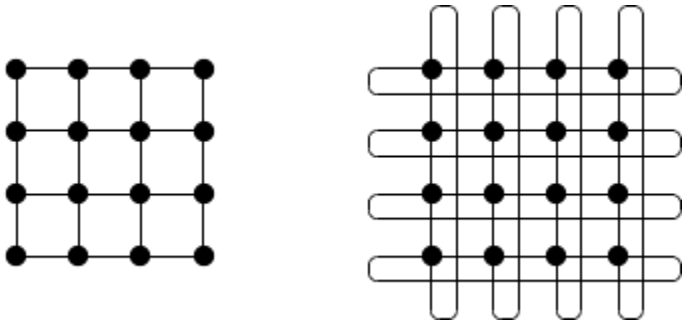


Fig. 1.6: A matrix and torus of 16 nodes.

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. 1.7.

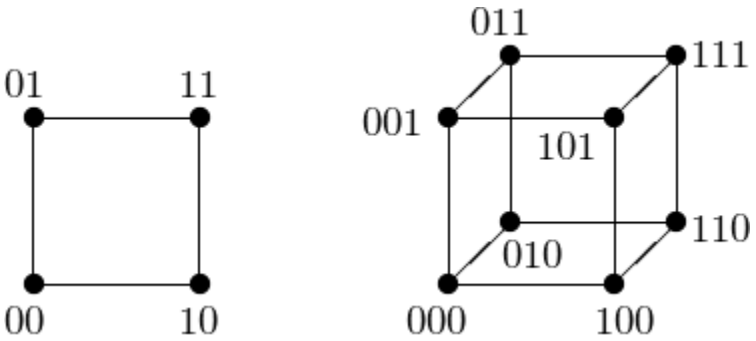


Fig. 1.7: Two special hypercubes: a square and cube.

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. 1.8.

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. 1.9.

A p-processor shared memory computer requires p^2 switches. 2-by-2 switches are shown in Fig. 1.10.

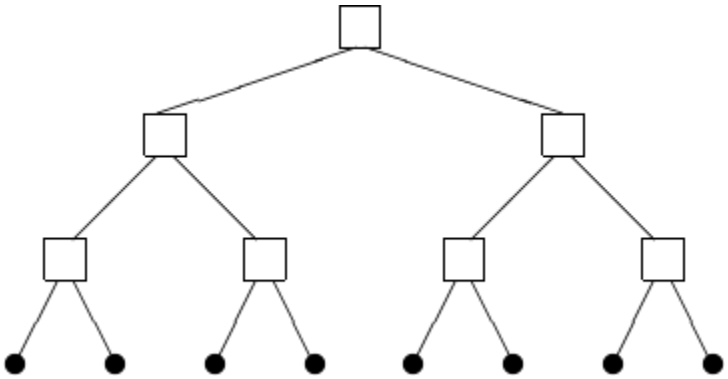


Fig. 1.8: A binary tree network.

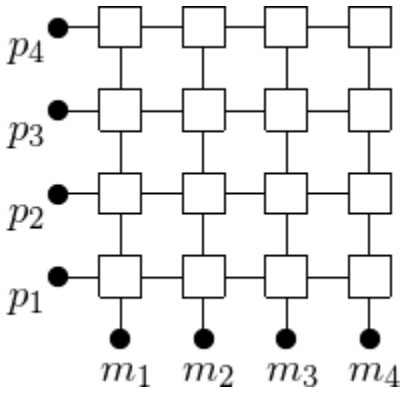


Fig. 1.9: Processors connected to memory modules via a crossbar switch.

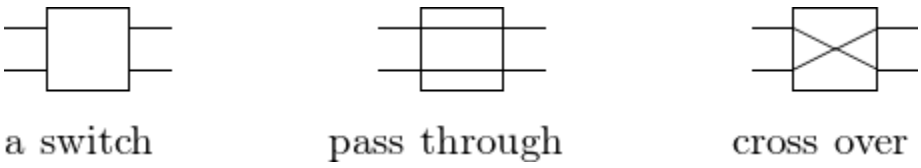


Fig. 1.10: 2-by-2 swiches.

Changing from *pass through* to *cross over* configuration changes the connections between the computers in the network, see Fig. 1.11.

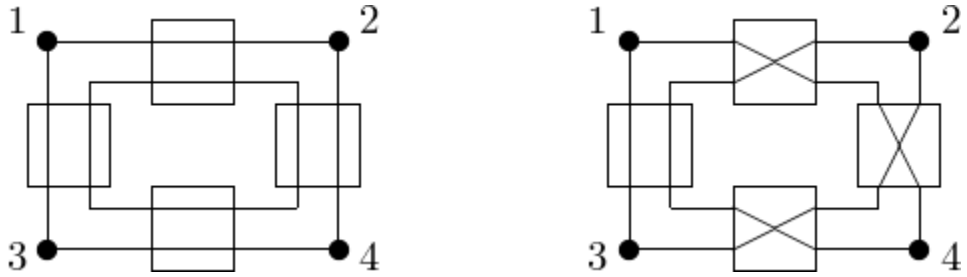


Fig. 1.11: Changing switches from pass through to cross over.

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. 1.12 shows a 2-stage network between 4 nodes.

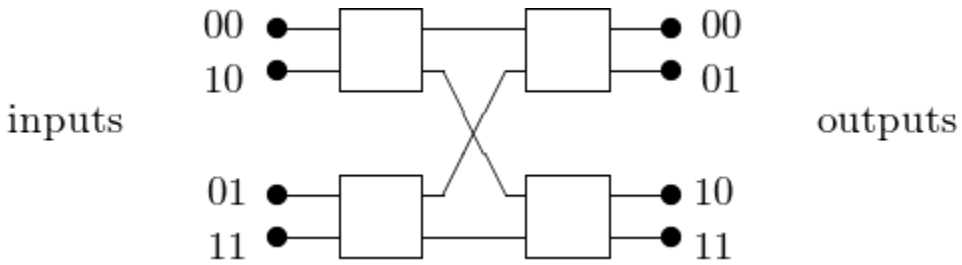


Fig. 1.12: A 2-stage network between 4 nodes.

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. 1.13 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. 1.14.

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.

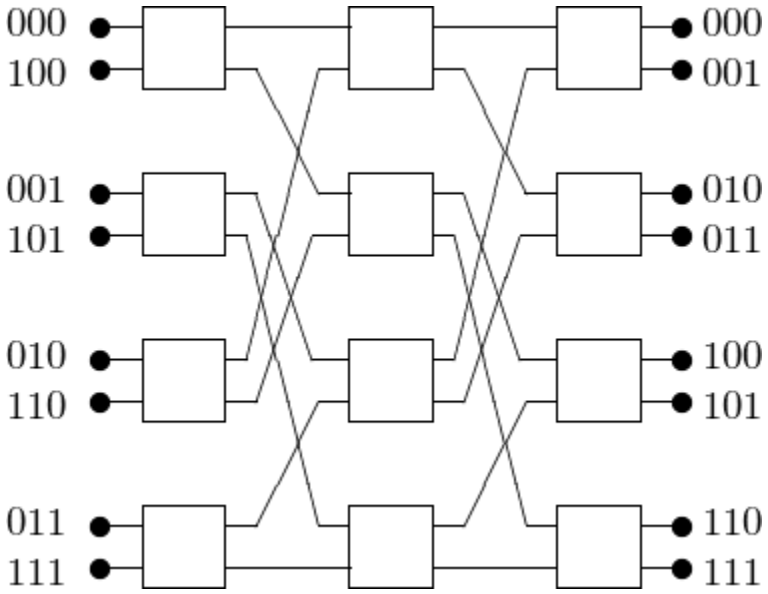


Fig. 1.13: A 3-stage Omega interconnection network.

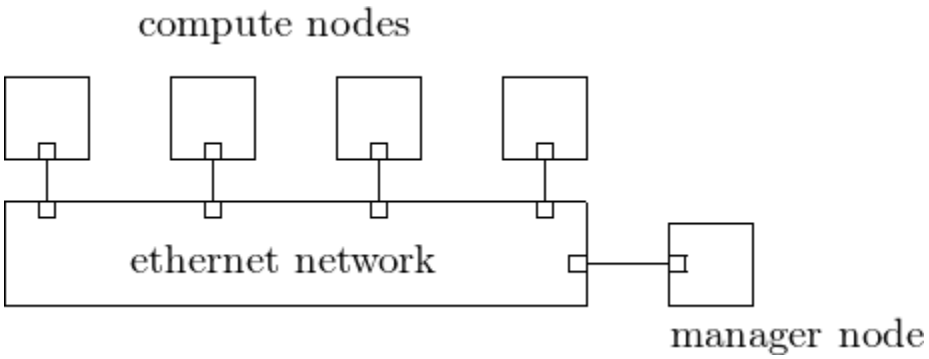


Fig. 1.14: A cluster connected via ethernet.

- 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 Maple, Python, 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 GPU computing with Maple

Maple is one of the big M's in scientific software. UIC has a campus wide license: available in labs. Well documented and supported. Since version 15, Maple 15 enables GPU computing and we can accelerate a matrix-matrix multiplication.

Experiments done on HP workstation Z800 with NVDIA Tesla C2050 general purpose graphic processing unit. The images in Fig. 1.15 and Fig. 1.16 are two screen shots of Maple worksheets.

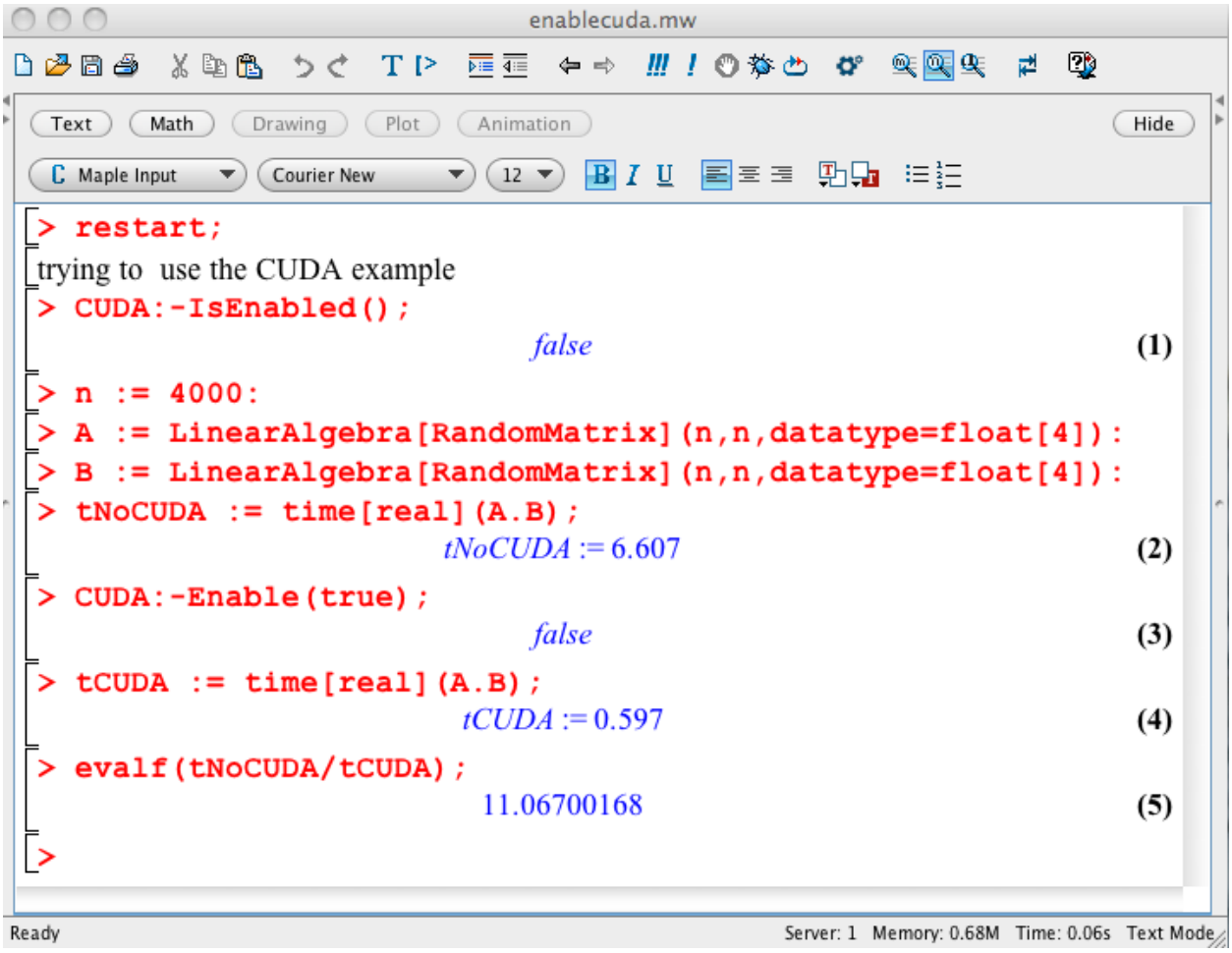


Fig. 1.15: First screen shot of a Maple worksheet.

As example application of matrix-matrix application, we consider a Markov chain. A stochastic process is a sequence of events depending on chance. A Markov process is a stochastic process with (1) a finite set of possible outcomes; (2) the probability of the next outcome depends only on the previous outcome; and (3) all probabilities are constant over time. Realization: $\mathbf{x}^{(k+1)} = A\mathbf{x}^{(k)}$, for $k = 0, 1, \ldots$, where A is an n-by-n matrix of probabilities and the vector $\mathbf{x}$ represents the state of the process. The sequence $\mathbf{x}^{(k)}$ is a Markov chain. We are interested in the long term behaviour: $\mathbf{x}^{(k+1)} = A^{k+1}\mathbf{x}(0)$.

As an application of Markov chains, consider the following model: UIC has about 3,000 new incoming freshmen each Fall. The state of each student measures time till graduation. Counting historical passing grades in gatekeeper courses give probabilities to transition from one level to the next. Our Goal is to model time till graduation based on rates of passing grades.

Although the number of matrix-matrix products is relatively small, to study sensitivity and what-if scenarios, many

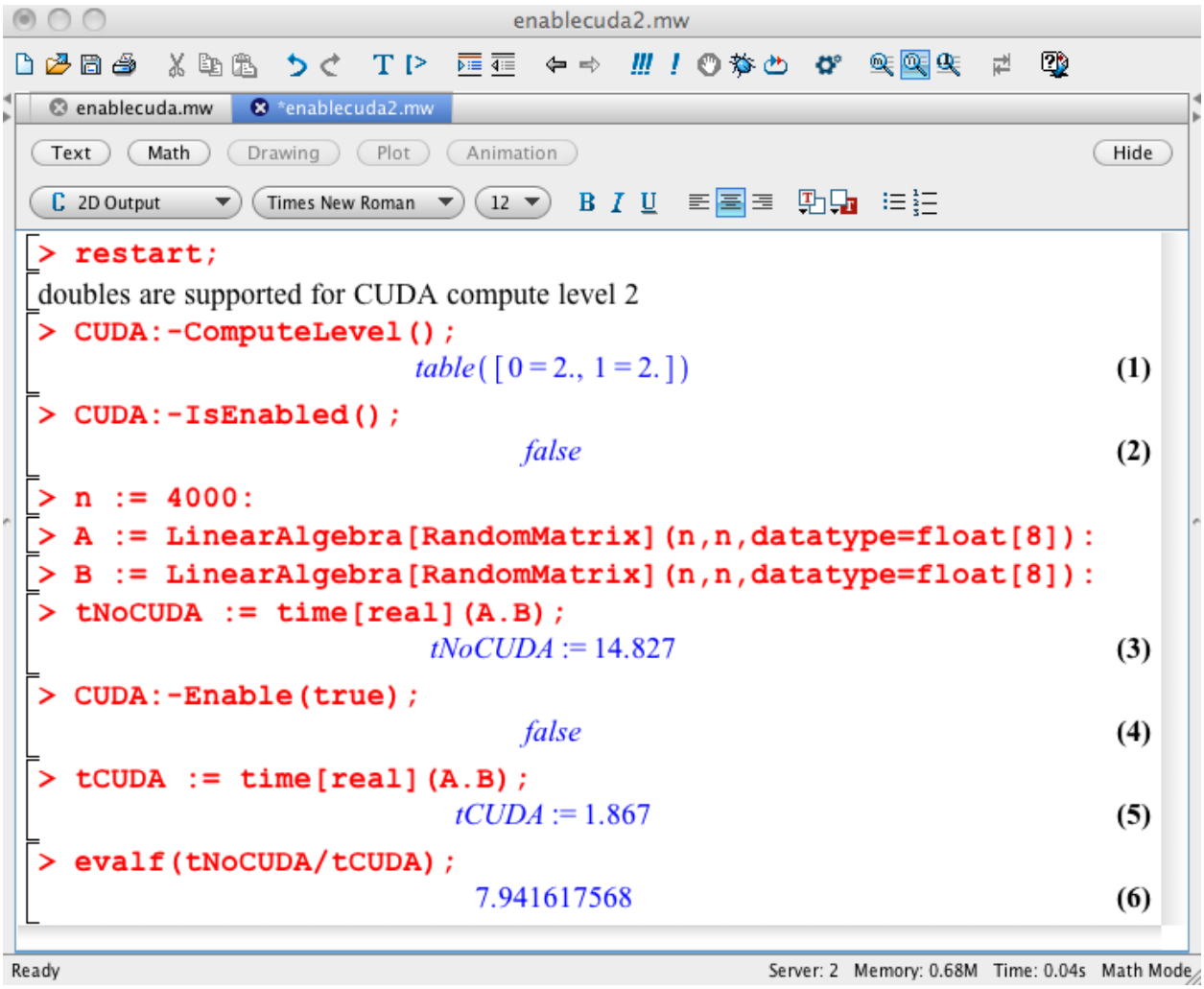


Fig. 1.16: Second screen shot of a Maple worksheet.

runs are needed.

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:

```
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)
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
    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:
        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 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
--   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)
                            & " 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);
    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 1.1: 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 $\frac{\text{real time with p = 1}}{\text{real time with p tasks}}$.

1.3.4 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
```

1.3.5 Exercises

1. For the Matrix-Matrix Multiplication of Maple with CUDA enabled investigate the importance of the size dimension n to achieve a good speedup. Experiment with values for $n \leq 4000$.

   For which values of n does the speedup drop to 2?

2. Write a Maple worksheet to generate matrices of probabilities for use in a Markov chain and compute at least 100 elements in the chain. For a large enough dimension, compare the elapsed time with and without CUDA enabled. To time code segments in a Maple worksheet, place the code segment between

   `start := time()` and `stop := time()` statements.

   The time spent on the code segment is then the difference between `stop` and `start`.

3. 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.

4. Compute the theoretical peak performance (expressed in giga or teraflops) of the two Intel Xeons E5-2670 in `kepler.math.uic.edu`. Justify your calculation.

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 or Python.

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

$ 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 */

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 );
/* 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");
```

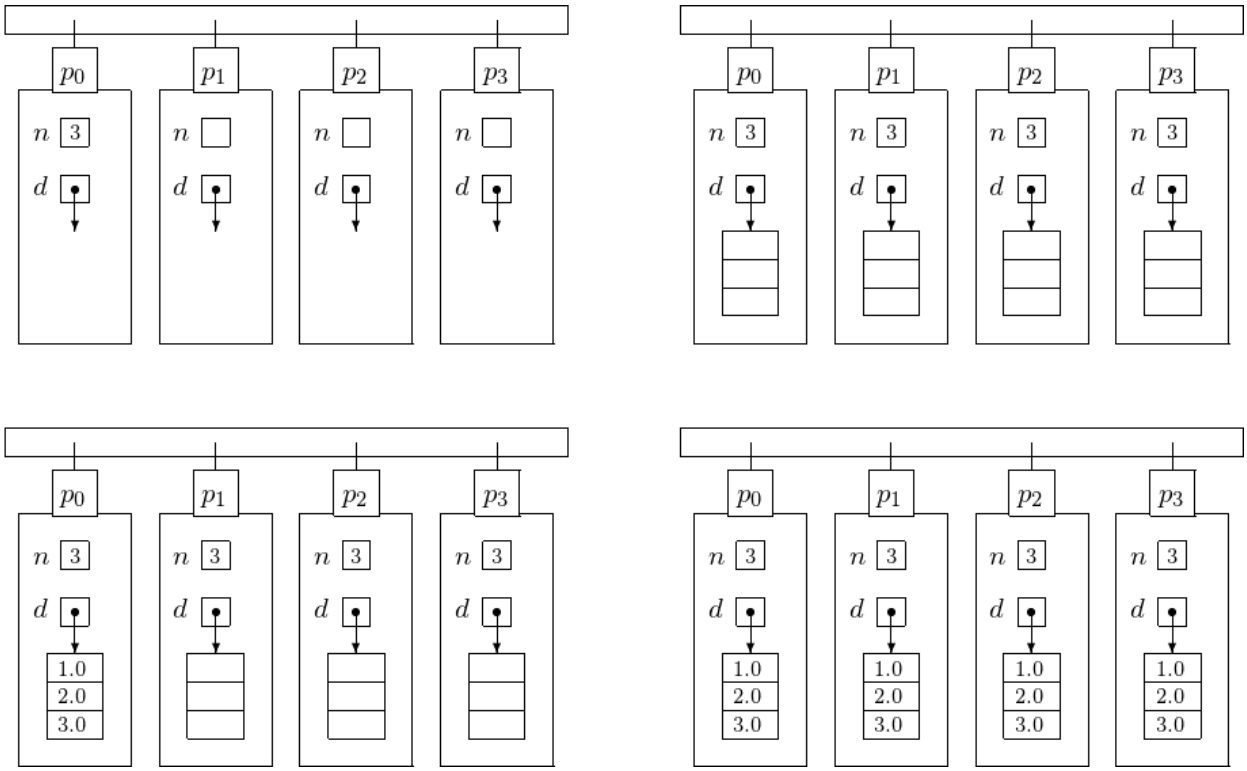


Fig. 2.1: On the schematic of a a distributed memory 4-processor computer, the top displays the situation before and after the broadcast of the dimension. After the broadcast of the dimension, each worker node allocates space for the array of doubles. The bottom two pictures display the situation before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

```
      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. It is available at `http://code.google.com/p/mpi4py`, with manual by Lisandro Dalcin: MPI for Python. The current Release 2.0.0 dates from July 2016.

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 Bibliography

1. L. Dalcin, R. Paz, and M. Storti. **MPI for Python.** *Journal of Parallel and Distributed Computing*, 65:1108-1115, 2005.
2. 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.

2.1.7 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

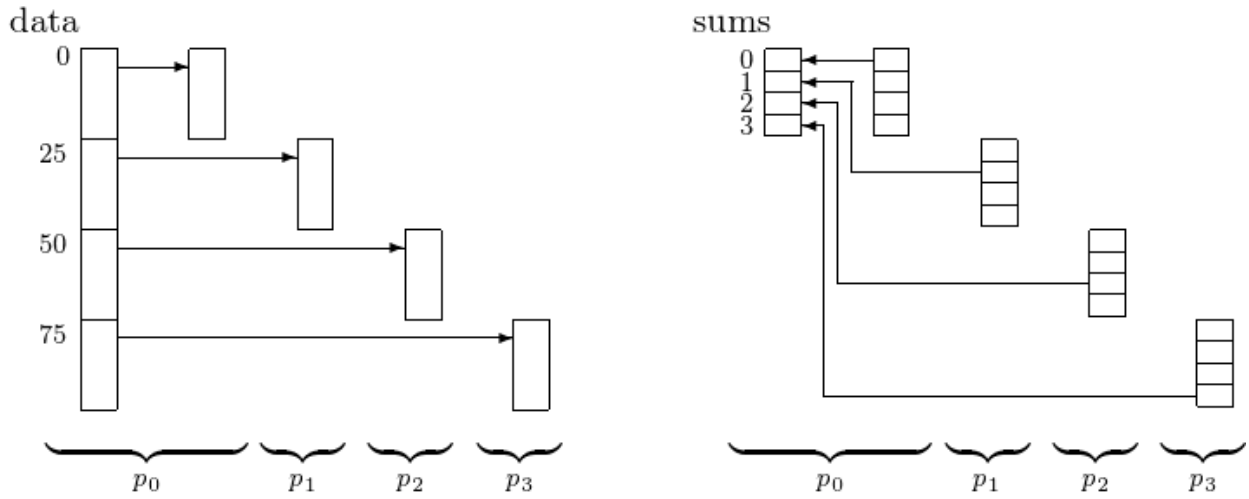


Fig. 2.2: Scattering data and gathering results.

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)
      {
         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)
      {
         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.

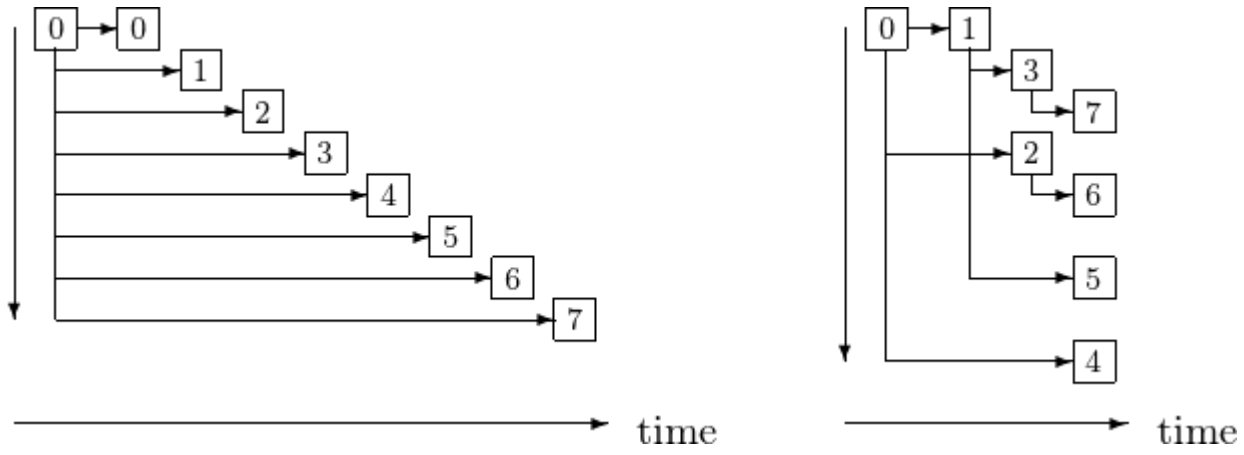


Fig. 2.3: Sequential (left) versus fan out (right) broadcast.

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.
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 ref{figmpi4pyparsum} 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]
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 Bibliography

1. L. Dalcin, R. Paz, and M. Storti. **MPI for Python**. *Journal of Parallel and Distributed Computing*, 65:1108–1115, 2005.
2. 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.

2.2.6 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 Python to rewrite the program to sum 100 numbers using `MPI_Send` and `MPI_Recv` instead of `MPI_Scatter` and `MPI_Gather`.
3. Use C or Python 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.

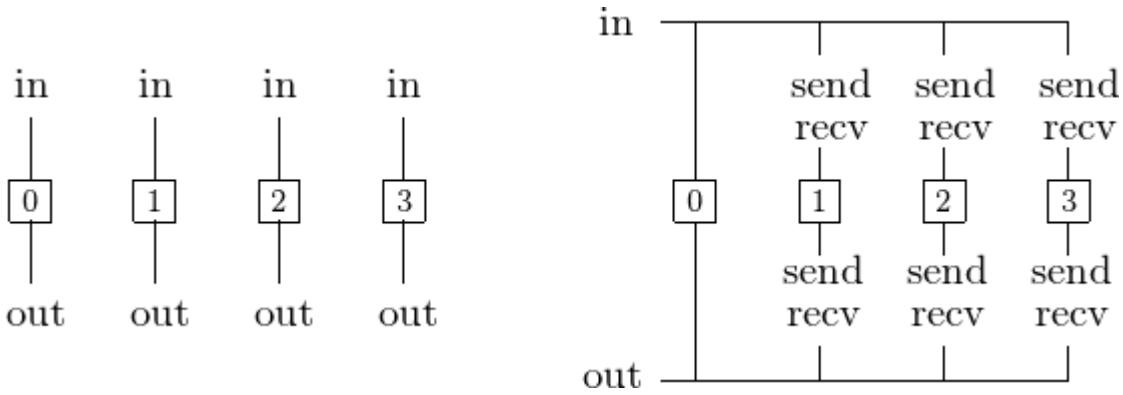


Fig. 2.4: One manager node distributes input data to the compute nodes and collects results from the compute nodes.

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

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;
   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;
   }
}
```

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>
#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.

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. */

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

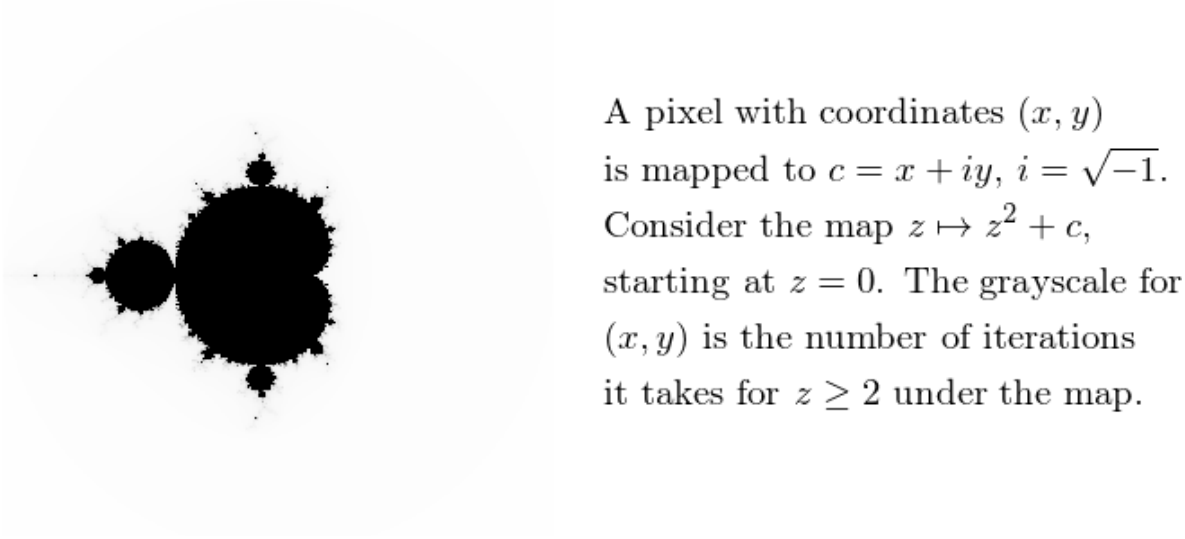


Fig. 2.5: The Mandelbrot set.

```
   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 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.

2.4.3 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;
   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;
}
```

2.4.4 Dynamic Work Load Balancing

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

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.

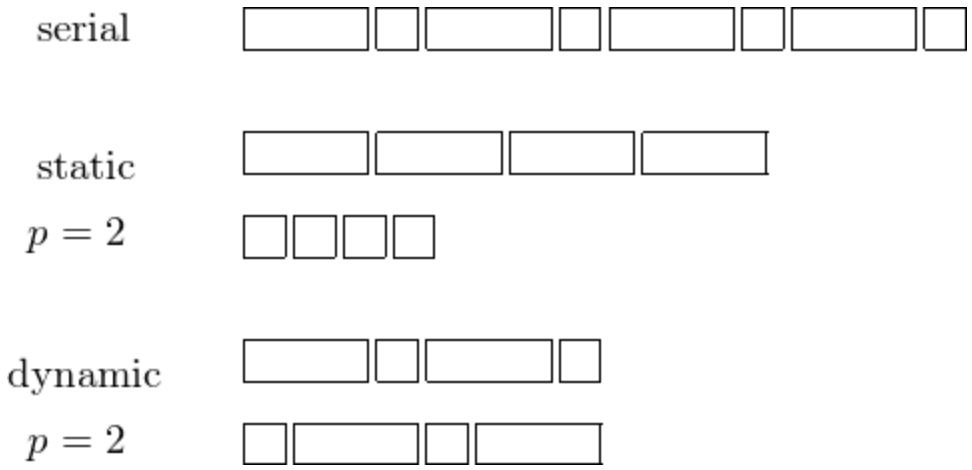


Fig. 2.6: Scheduling 8 jobs on 2 processors. In a worst case scenario, with static job scheduling, all the long jobs end up at one processor, while the short ones at the other, creating an uneven work load.

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 */
         {
            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.

2.4.5 Bibliography

1. George Cybenko. **Dynamic Load Balancing for Distributed Memory Processors**. *Journal of Parallel and Distributed Computing* 7, 279-301, 1989.

2.4.6 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 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.

2.5.1 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.

2.5.2 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.

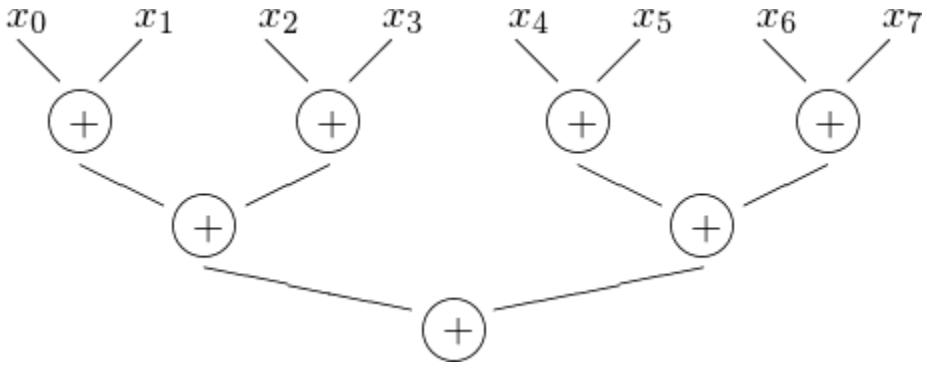


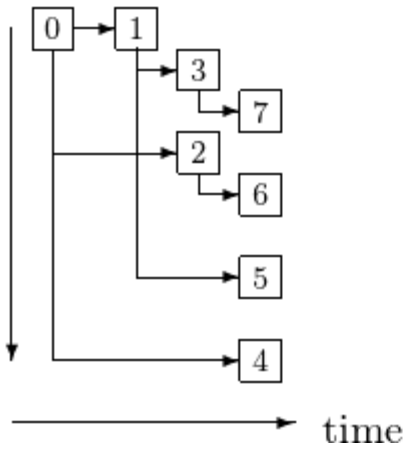
Fig. 2.7: With 4 processors, the summation of 8 numbers in done in 3 steps.

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.



	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.

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

```
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 $b2b1b0$ has data starting at index $b_0b_1b_2$.

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
1 sends 20 integers to 3 at 60, start 60
2 received 20 integers from 0 at 20, start 20
3 received 20 integers from 1 at 60, start 60
```

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

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 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",
                           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;
}
```

Fanning in the results is illustrated in Fig. 2.9.

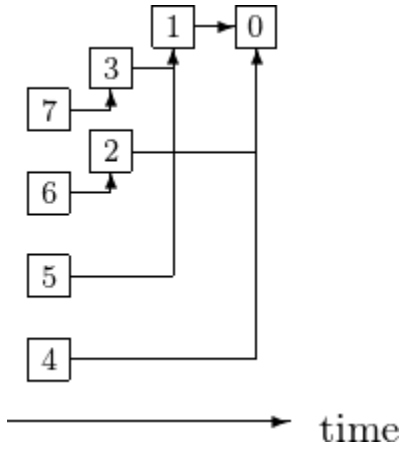


Fig. 2.9: Fanning in results.

```
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.

2.5.3 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.

2.5.4 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

2.5.5 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. Complete the summation and the fanning in of the partial sums, extending the program. You may leave $p = 8$.

CHAPTER 3

Introduction to Multithreading

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

3.1 Introduction to OpenMP

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.

3.1.1 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.
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());
   }
}
```

3.1.2 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

$ 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.

3.1.3 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>.

3.1.4 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 Introduction to Pthreads

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 the POSIX threads programming interface

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. 3.1 next to a multithreaded process.

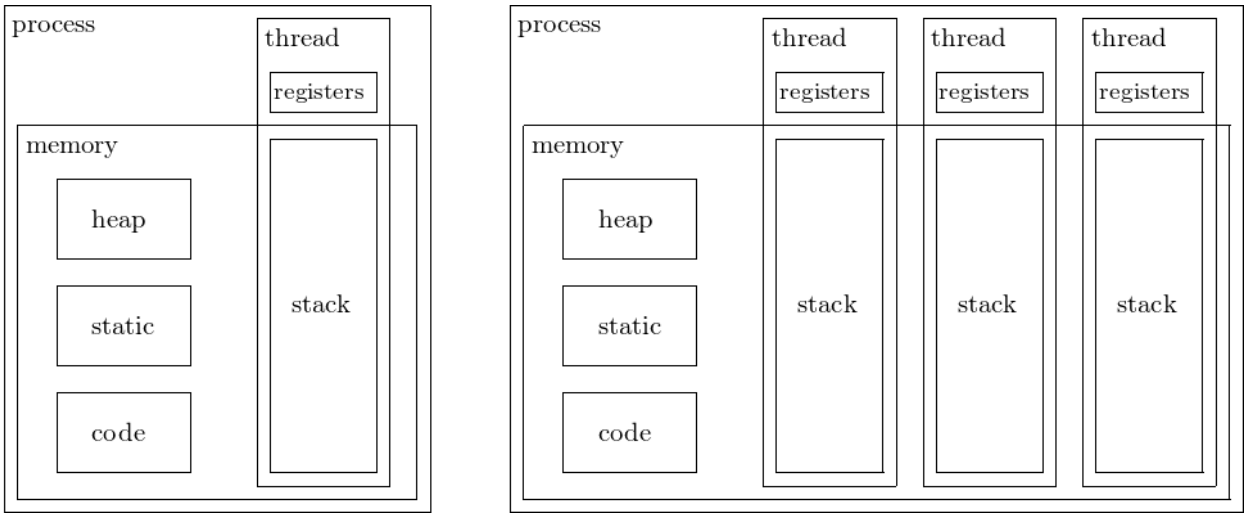


Fig. 3.1: At the left we see a process with one single thread and at the right a multithreaded process.

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.

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
```

3.2.2 Using Pthreads

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
   (a) the address of each thread,
   (b) the address of an attribute,
   (c) the function each thread executes, and
   (d) 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`.

3.2.3 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. 3.2 illustrates a task performed by three threads in a work crew model.

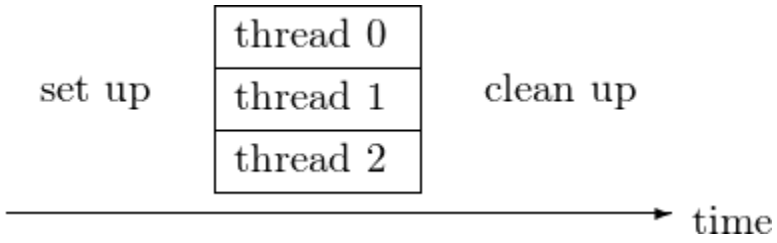


Fig. 3.2: A task performed by 3 threads in a work crew model.

If the task 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.

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];
   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);
}
```

3.2.4 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
$
```

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);

   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 $j = n \bmod 5,000$.

3.2.5 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
   (a) first picks up the left chop stick, and
   (b) 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, ...

3.2.6 Bibliography

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

3.2.7 Exercises

1. Modify the `hello world!` program with 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 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 for a work crew model of threads to compute the grayscales pixel by pixel. Compare the running time of your program using Pthreads with your MPI implementation.
4. Write a simulation for the dining philosophers problem. Could you observe starvation? Explain.

3.3 Introduction to the Intel Threading Building Blocks

Instead of working directly with threads, we can define tasks that are then mapped to threads. Work stealing is an alternative to load balancing.

3.3.1 the Intel Threading Building Blocks (TBB)

In this week we introduce programming tools for shared memory parallelism. Today we introduce a third tool:

1. OpenMP: programming shared memory parallel computers;
2. Pthreads: POSIX standard for Unix system programming; and
3. Intel Threading Building Blocks (TBB) for multicore processors.

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/>

3.3.2 task based programming and work stealing

Tasks are much lighter than threads. On Linux, starting and terminating a task is about 18 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. The TBB task scheduler uses *work stealing* for load balancing. 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.

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
$
```

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

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>
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
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;
}
```

speeding up the computations with parallel_for

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: $\frac{17.456}{1.579} = 11.055$ 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"

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));
```

3.3.4 using the parallel_reduce

We consider the summation of integers as an application of work stealing. Fig. 3.3 and Fig. 3.4 are taken from the Intel TBB tutorial.

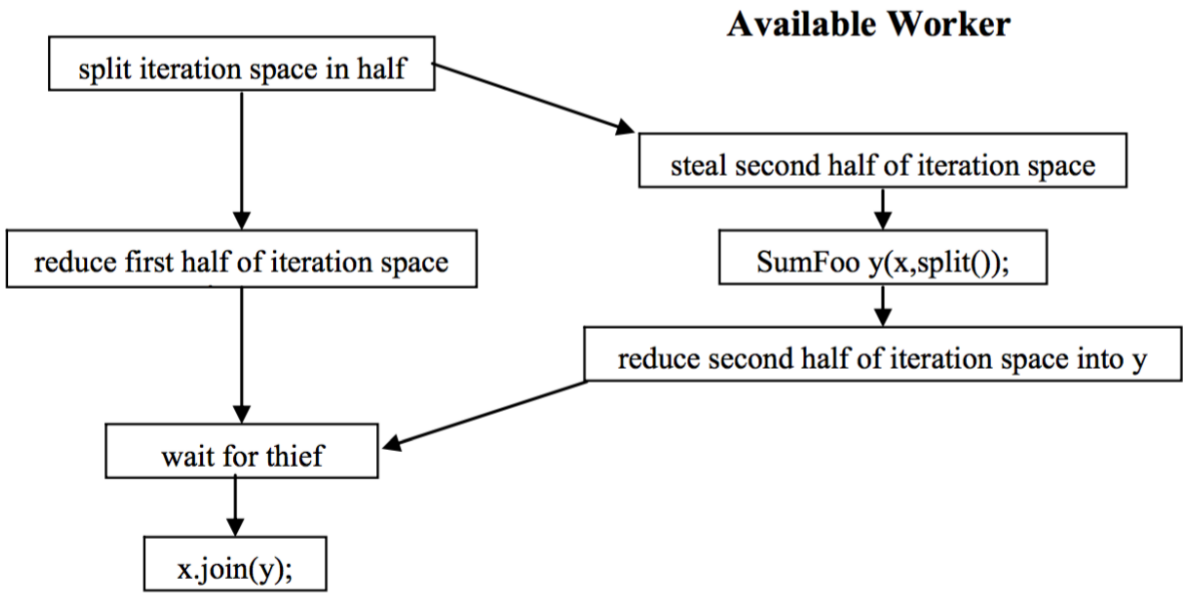


Fig. 3.3: An application of work stealing.

The definition of the class to sum a sequence of integers is below.

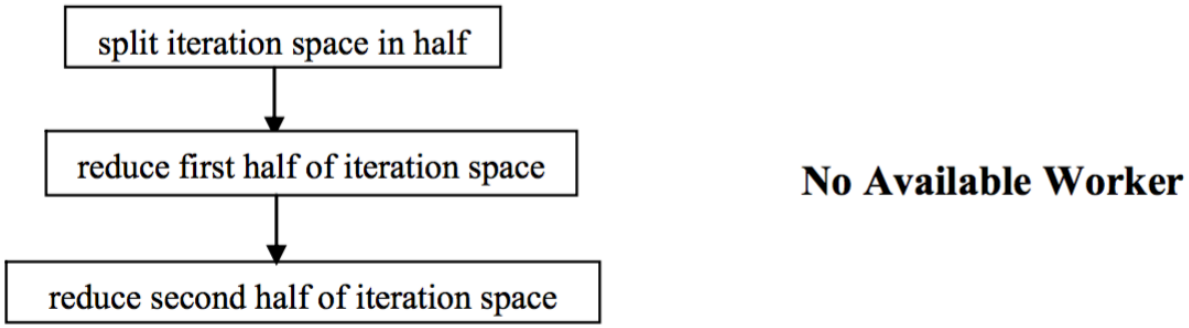


Fig. 3.4: What if no worker is available?

```
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
        << " and it should be " << n*(n+1)/2
        << endl;
   return 0;
}
```

3.3.5 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.

3.3.6 Exercises

1. Modify the `hello world!` program with so that the user is first prompted for a name. Two tasks are spawned and they use the given name in their greeting.
2. 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

Applications: Sorting, Integration, FFT

4.1 Parallel Sorting Algorithms

Sorting is one of the most fundamental problems. On distributed memory computers, we study parallel bucket sort. On shared memory computers, we examine parallel quicksort.

4.1.1 Sorting in C and C++

In C we have the function `qsort`, which implements quicksort. The prototype is

```
void qsort ( void *base, size_t count, size_t size,
             int (*compar)(const void *element1, const void *element2) );
```

`qsort` sorts an array whose first element is pointed to by base and contains count elements, of the given size. The function `compar` returns

- -1 if `element1` $<$ `element2`;
- 0 if `element1` $=$ `element2`; or
- $+1$ if `element1` $>$ `element2`.

We will apply `qsort` to sort a random sequence of doubles. Functions to generate an array of random numbers and to write the array are listed below.

```
void random_numbers ( int n, double a[n] )
{
   int i;
   for(i=0; i<n; i++)
      a[i] = ((double) rand())/RAND_MAX;
}
void write_numbers ( int n, double a[n] )
{
   int i;
   for(i=0; i<n; i++) printf("%.15e\n", a[i]);
}
```

To apply `qsort`, we define the `compare` function:

```
int compare ( const void *e1, const void *e2 )
{
   double *i1 = (double*)e1;
```

```
   double *i2 = (double*)e2;
   return ((*i1 < *i2) ? -1 : (*i1 > *i2) ? +1 : 0);
}
```

Then we can call `qsort` in the function `main()` as follows:

```
double *a = (double*)calloc(n,sizeof(double));
random_numbers(n,a);
qsort((void*)a,(size_t)n,sizeof(double),compare);
```

We use the command line to enter the dimension and to toggle off the output. To measure the CPU time for sorting:

```
clock_t tstart,tstop;
tstart = clock();
qsort((void*)a,(size_t)n,sizeof(double),compare);
tstop = clock();
printf("time elapsed : %.4lf seconds\n",
       (tstop - tstart)/((double) CLOCKS_PER_SEC));
```

Some timings with `qsort` on 3.47Ghz Intel Xeon are below:

```
$ time /tmp/time_qsort 1000000 0
time elapsed : 0.2100 seconds
real    0m0.231s
user    0m0.225s
sys     0m0.006s
$ time /tmp/time_qsort 10000000 0
time elapsed : 2.5700 seconds
real    0m2.683s
user    0m2.650s
sys     0m0.033s
$ time /tmp/time_qsort 100000000 0
time elapsed : 29.5600 seconds
real    0m30.641s
user    0m30.409s
sys     0m0.226s
```

Observe that $O(n \log_2(n))$ is almost linear in n. In C++ we apply the `sort` of the Standard Template Library (STL), in particular, we use the STL container `vector`. Functions to generate vectors of random numbers and to write them are given next.

```
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

vector<double> random_vector ( int n ); // returns a vector of n random doubles
void write_vector ( vector<double> v ); // writes the vector v

vector<double> random_vector ( int n )
{
   vector<double> v;
   for(int i=0; i<n; i++)
   {
      double r = (double) rand();
      r = r/RAND_MAX;
      v.push_back(r);
   }
```

```
   return v;
}
void write_vector ( vector<double> v )
{
   for(int i=0; i<v.size(); i++)
      cout << scientific << setprecision(15) << v[i] << endl;
}
```

To use the `sort` of the STL, we define the compare function, including the `algorithm` header:

```
#include <algorithm>
struct less_than // defines "<"
{
   bool operator()(const double& a, const double& b)
   {
      return (a < b);
   }
};
```

In the main program, to sort the vector `v` we write

```
sort(v.begin(), v.end(), less_than());
```

Timings of the STL `sort` on 3.47Ghz Intel Xeon are below.

```
$ time /tmp/time_stl_sort 1000000 0
time elapsed : 0.36 seconds
real    0m0.376s
user    0m0.371s
sys     0m0.004s
$ time /tmp/time_stl_sort 10000000 0
time elapsed : 4.09 seconds
real    0m4.309s
user    0m4.275s
sys     0m0.033s
$ time /tmp/time_stl_sort 100000000 0
time elapsed : 46.5 seconds
real    0m48.610s
user    0m48.336s
sys     0m0.267s
```

Different distributions may cause timings to fluctuate.

4.1.2 Bucket Sort for Distributed Memory

On distributed memory computers, we explain bucket sort. Given are n numbers, suppose all are in $[0, 1]$. The algorithm using p buckets proceeds in two steps:

1. Partition numbers x into p buckets: $x \in [i/p, (i+1)/p[\quad \Rightarrow \quad x \in (i+1)$-th bucket.
2. Sort all p buckets.

The cost to partition the numbers into p buckets is $O(n \log_2(p))$. Note: radix sort uses most significant bits to partition. In the best case: every bucket contains n/p numbers. The cost of Quicksort is $O(n/p \log_2(n/p))$ per bucket. Sorting p buckets takes $O(n \log_2(n/p))$. The total cost is $O(n(\log_2(p) + \log_2(n/p)))$.

parallel bucket sort

On p processors, all nodes sort:

1. The root node distributes numbers: processor *i* gets *i*-th bucket.
2. Processor *i* sorts *i*-th bucket.
3. The Root node collects sorted buckets from processors.

Is it worth it? Recall: the serial cost is $n(\log_2(p) + \log_2(n/p))$. The cost of parallel algorithm:

1. $n \log_2(p)$ to place numbers into buckets, and
2. $n/p \log_2(n/p)$ to sort buckets.

Then we compute the speedup:

$$\begin{aligned} \text{speedup} &= \frac{n(\log_2(p) + \log_2(n/p))}{n(\log_2(p) + \log_2(n/p)/p)} \\ &= \frac{1+L}{1+L/p} = \frac{1+L}{(p+L)/p} = \frac{p}{p+L}(1+L), \quad L = \frac{\log_2(n/p)}{\log_2(p)}. \end{aligned}$$

Comparing to quicksort, the speedup is

$$\begin{aligned} \text{speedup} &= \frac{n \log_2(n)}{n(\log_2(p) + n/p \log_2(n/p))} \\ &= \frac{\log_2(n)}{\log_2(p) + 1/p(\log_2(n) - \log_2(p))} \\ &= \frac{\log_2(n)}{1/p(\log_2(n) + (1-1/p)\log_2(p))} \end{aligned}$$

For example, $n = 2^{20}, \log_2(n) = 20, p = 2^2, \log_2(p) = 2$, then

$$\begin{aligned} \text{speed up} &= \frac{20}{1/4(20) + (1-1/4)2} \\ &= \frac{20}{5+3/2} = \frac{40}{13} \approx 3.08. \end{aligned}$$

communication versus computation

The scatter of n data elements costs $t_{\text{start up}} + nt_{\text{data}}$, where t_{data} is the cost of sending 1 data element. For distributing and collecting of all buckets, the total communication time is $2p\left(t_{\text{start up}} + \frac{n}{p}t_{\text{data}}\right)$. The computation/communication ratio is

$$\frac{(n \log_2(p) + n/p \log_2(n/p))t_{\text{compare}}}{2p\left(t_{\text{start up}} + \frac{n}{p}t_{\text{data}}\right)}$$

where t_{compare} is the cost for one comparison. The computation/communication ratio is

$$\frac{(n \log_2(p) + n/p \log_2(n/p))t_{\text{compare}}}{2p\left(t_{\text{start up}} + \frac{n}{p}t_{\text{data}}\right)}$$

where t_{compare} is the cost for one comparison. We view this ratio for $n \gg p$, for fixed p, so:

$$\frac{n}{p}\log_2\left(\frac{n}{p}\right) = \frac{n}{p}\left(\log_2(n) - \log_2(p)\right) \approx \frac{n}{p}\log_2(n).$$

The ratio then becomes $\frac{n}{p}\log_2(n)t_{\text{compare}} \gg 2nt_{\text{data}}$. Thus $\log_2(n)$ must be sufficiently high...

4.1.3 Quicksort for Shared Memory

A recursive algorithm partitions the numbers:

```
void quicksort ( double *v, int start, int end ) {
   if(start < end) {
      int pivot;
      partition(v,start,end,&pivot);
      quicksort(v,start,pivot-1);
      quicksort(v,pivot+1,end);
   }
}
```

where `partition` has the prototype:

```
void partition ( double *v, int lower, int upper, int *pivot );
/* precondition: upper - lower > 0
 * takes v[lower] as pivot and interchanges elements:
 * v[i] <= v[pivot] for all i < pivot, and
 * v[i] > v[pivot] for all i > pivot,
 * where lower <= pivot <= upper. */
```

A definition for the `partition` function is listed next:

```
void partition ( double *v, int lower, int upper, int *pivot )
{
   double x = v[lower];
   int up = lower+1;   /* index will go up */
   int down = upper;   /* index will go down */
   while(up < down)
   {
      while((up < down) && (v[up] <= x)) up++;
      while((up < down) && (v[down] > x)) down--;
      if(up == down) break;
      double tmp = v[up];
      v[up] = v[down]; v[down] = tmp;
   }
   if(v[up] > x) up--;
   v[lower] = v[up]; v[up] = x;
   *pivot = up;
}
```

Then in `main()` we can apply `partition` and `qsort` to sort an array of `n` doubles:

```
int lower = 0;
int upper = n-1;
int pivot = 0;
if(n > 1) partition(v,lower,upper,&pivot);
if(pivot != 0) qsort((void*)v,(size_t)pivot, sizeof(double),compare);
if(pivot != n) qsort((void*)&v[pivot+1],(size_t)(n-pivot-1),sizeof(double),compare);
```

quicksort with OpenMP

A parallel region in `main()`:

```
omp_set_num_threads(2);
#pragma omp parallel
```

```
{
   if(pivot != 0) qsort((void*)v,(size_t)pivot,sizeof(double),compare);
   if(pivot != n) qsort((void*)&v[pivot+1],(size_t)(n-pivot-1),sizeof(double),
↪compare);
}
```

Running on dual core Mac OS X at 2.26 GHz gives the following timings:

```
$ time /tmp/time_qsort 10000000 0
time elapsed : 4.0575 seconds
real    0m4.299s
user    0m4.229s
sys     0m0.068s

$ time /tmp/part_qsort_omp 10000000 0
pivot = 4721964
-> sorting the first half : 4721964 numbers
-> sorting the second half : 5278035 numbers
real    0m3.794s
user    0m7.117s
sys     0m0.066s
```

Speed up: 4.299/3.794 = 1.133, or 13.3% faster with one extra core.

parallel sort with Intel TBB

At the top of the program, add the lines

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

using namespace tbb;
```

To sort a number of random doubles:

```
int n;
double *v;
v = (double*)calloc(n,sizeof(double));
random_numbers(n,v);
parallel_sort(v, v+n);
```

The complete main program with the headers for the functions is below.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "tbb/parallel_sort.h"

using namespace tbb;

void random_numbers ( int n, double *v );
/* returns n random doubles in [0,1] */

void write_numbers ( double *v, int lower, int upper );
/* writes the numbers in v from lower to upper,
 * including upper */

int main ( int argc, char *argv[] )
```

```
{
   int n;
   if(argc > 1)
      n = atoi(argv[1]);
   else
   {
      printf("give n : ");
      scanf("%d",&n);
   }
   int vb = 1;
   if(argc > 2) vb = atoi(argv[2]);
   srand(time(NULL));
   double *v;
   v = (double*)calloc(n,sizeof(double));
   random_numbers(n,v);
   if(vb > 0)
   {
      printf("%d random numbers : \n",n);
      write_numbers(v,0,n-1);
   }
   parallel_sort(v, v+n);
   if(vb > 0)
   {
      printf("the sorted numbers :\n");
      write_numbers(v,0,n-1);
   }
   return 0;
}
```

A session with an interactive run to test the correctness goes as below.

```
$ /tmp/tbb_sort 4 1
4 random numbers :
3.696845319912231e-01
7.545582678888730e-01
6.707372915329120e-01
3.402865237278335e-01
the sorted numbers :
3.402865237278335e-01
3.696845319912231e-01
6.707372915329120e-01
7.545582678888730e-01
$
```

We can time the parallel runs as follows.

```
$ time /tmp/tbb_sort 10000000 0

real    0m0.479s
user    0m4.605s
sys     0m0.168s
$ time /tmp/tbb_sort 100000000 0

real    0m4.734s
user    0m51.063s
sys     0m0.386s
$ time /tmp/tbb_sort 1000000000 0
```

```
real    0m47.400s
user    9m32.713s
sys     0m2.073s
$
```

4.1.4 Bibliography

1. Edgar Solomonik and Laxmikant V. Kale: **Highly Scalable Parallel Sorting.** In the proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2010.
2. Mirko Rahn, Peter Sanders, and Johannes Singler: **Scalable Distributed-Memory External Sorting**. In the proceedings of the 26th IEEE International Conference on Data Engineering (ICDE), pages 685-688, IEEE, 2010.
3. Davide Pasetto and Albert Akhriev: **A Comparative Study of Parallel Sort Algorithms**. In SPLASH'11, the proceedings of the ACM international conference companion on object oriented programming systems languages and applications, pages 203-204, ACM 2011.

4.1.5 Exercises

1. Consider the fan out scatter and fan in gather operations and investigate how these operations will reduce the communication cost and improve the computation/communication ratio in bucket sort of n numbers on p processors.
2. Instead of OpenMP, use Pthreads to run Quicksort on two cores.
3. Instead of OpenMP, use the Intel Threading Building Blocks to run Quicksort on two cores.

4.2 Parallel Numerical Integration

The problem of numerical integeration is another illustration of an ideal parallel computation: the communication cost is constant and the amount of computational work increases as the number of function evaluations increases. The complexity of the function and the working precision is another factor which increases the computational cost.

4.2.1 Numerical Integration

In numerical integration we consider the problem of approximating the definite integral of a function over a domain. By domain decomposition we naturally arrive at parallel algorithms.

Let $a < b$ and consider $a = c_0 < c_1 < \cdots < c_{p-2} < c_{p-1} = b$, then:

$$\int_a^b f(x)dx = \sum_{k=1}^{p} \int_{c_{k-1}}^{c_k} f(x)dx.$$

We have p subintervals of $[a, b]$ and on each subinterval $[c_{k-1}, c_k]$ we apply a quadrature formula (weighted sum of function values):

$$\int_{c_{k-1}}^{c_k} f(x)dx \approx \sum_{j=1}^{n} w_j f(x_j)$$

where the weights w_j correspond to points $x_j \in [c_{k-1}, c_k]$. Let the domain D be partitioned as $\bigcup_{i=1}^{n} \Delta_i$:

$$\int_D f(x_1, x_2) dx_1 dx_2 = \sum_{k=1}^{n} \int_{\Delta_k} f(x_1, x_2) dx_1 dx_2.$$

For a triangle Δ, an approximation of the integral of f over Δ is to take the volume between the plane spanned by the function values at the corners of Δ the x_1x_2-plane. Finer domain decompositions of D lead to more triangles Δ, more function evaluations, and more accurate approximations.

Like Monte Carlo simulation, numerical integration is pleasingly parallel. The function evaluations can be computed independently from each other. No communication between processors needed once the subdomains have been distributed. The size of all communication is small. On input we have the definition of the subdomain, and on return is one weighted sum of function values.

To obtain highly accurate values when applying extrapolation on the trapezoidal rule (so-called Romberg integration), we use quad double arithmetic. A quad double is an unevaluated sum of 4 doubles, improves working precision from 2.2×10^{-16} to 2.4×10^{-63}. A quad double builds on the `double double`. 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. The QD library supports operator overloading in C++, as shown in the example code below.

```
#include <iostream>
#include <iomanip>
#include <qd/qd_real.h>
using namespace std;
int main ( void )
{
   qd_real q("2");
   cout << setprecision(64) << q << endl;
   for(int i=0; i<8; i++)
   {
      qd_real dq = (q*q - 2.0)/(2.0*q);
      q = q - dq; cout << q << endl;
   }
   cout << scientific << setprecision(4) << "residual : " << q*q - 2.0 << endl;
   return 0;
```

Compiling and running the program could go as below.

```
$ g++ -I/usr/local/qd-2.3.17/include qd4sqrt2.cpp /usr/local/lib/libqd.a -o /tmp/
↪qd4sqrt2
$ /tmp/qd4sqrt2
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
$
```

Instead of typing in all arguments to `g++` we better work with a `makefile` which contains

```
QD_ROOT=/usr/local/qd-2.3.17
QD_LIB=/usr/local/lib

qd4sqrt2:
        g++ -I$(QD_ROOT)/include qd4sqrt2.cpp \
            $(QD_LIB)/libqd.a -o qd4sqrt2
```

Then we can simply type `make qd4sqrt2` at the command prompt to build the executable `qd4sqrt2`.

Returning to the problem of approximating π we consider $\pi = \int_0^1 \frac{16x-16}{x^4-2x^3+4x-4}dx$. We apply the composite Trapezoidal rule doubling in each step the number of subintervals of $[0,1]$. Recycling the function evaluations, the next approximation requires as many function evaluations as in the previous step. To accelerate the convergence, extrapolate on the errors:

$$T[i][j] = \frac{T[i][j-1]2^{2j} - T[i-1][j-1]}{2^{2j}-1}, \quad T[i][0] = T\left(\frac{h}{2^i}\right).$$

where

$$T(h) = \frac{h}{2}(f(a)+f(b)) + h\sum_{k=1}^{n-1} f(a+kh), \quad h = \frac{b-a}{n}.$$

Running the program produces the following approximations for π, improved with Romberg integration.

```
$ /tmp/romberg4piqd
Give n : 20
Trapezoidal rule :
2.0000000000000000000000000000000000000000000000000000000000000000000000000000e+00  -1.1e+00
2.8285714285714285714285714285714285714285714285714285714285714285714285714286e+00  -3.1e-01
3.0599849140217096656334342993859535399012339858804671938901787924e+00  -8.2e-02
3.1208799149782192753090608934264402720707593687963124445027369600e+00  -2.1e-02
3.1363921117379842496423260900707118017245006984692475608300990573e+00  -5.2e-03
3.1402910615222379836674014569717873479608997416673717997690893828e+00  -1.3e-03
3.1412671635291865246174301601168385100304285866900551637643442050e+00  -3.3e-04
3.1415112753058333508223495008044322351701491319859264242480853344e+00  -8.1e-05
3.1415723086580001130741506654555989016008550279324178879004173905e+00  -2.0e-05
3.1415875673342908067733429372872267595388334364564318736724256675e+00  -5.1e-06
3.1415913820245079343519112519075857885126411600435940845928207730e+00  -1.3e-06
3.1415923356983838054575966637759377218669127222763583195386129930e+00  -3.2e-07
3.1415925741169353735102088131178638490686580423772000985781988075e+00  -7.9e-08
3.1415926337215784280554756890689392085987403031281282606384169231e+00  -2.0e-08
3.1415926486227395143502815854605592727657670704677224154970568916e+00  -5.0e-09
3.1415926523480298060901422591278382431441533032555099184797306336e+00  -1.2e-09
3.1415926532793523802854924341737272725049748286771339948938222906e+00  -3.1e-10
3.1415926535121830239131040417347661807897041512988564129230237347e+00  -7.8e-11
3.1415926535703906848249303226263308523017737071795040403161676085e+00  -1.9e-11
3.1415926535849426000531946040370197046578880393737659063268947933e+00  -4.9e-12
Romberg integration :
2.0000000000000000000000000000000000000000000000000000000000000000000000000000e+00  -1.1e+00
3.1047619047619047619047619047619047619047619047619047619047619047619047619048e+00  -3.7e-02
3.1392801316880188260437414797616101138912788116649184217866669809e+00  -2.3e-03
3.1414830360866343425236605718165706514196020802703167069117496616e+00  -1.1e-04
3.1415911260349351404190698539313113170056259534276006755868536015e+00  -1.5e-06
3.1415926431831900702282508859066437449426241659436386656679964387e+00  -1.0e-08
3.1415926535646358543989993705490112789767840363176063755945965454e+00  -2.5e-11
3.1415926535897718115554422419526551887771361680030439892483820211e+00  -2.1e-14
3.1415926535897932322804851779277884529440555741904175076351878046e+00  -6.2e-18
```

```
3.14159265358979323846206178951029046288663292765809377387833092870e+00  -5.8e-22
3.14159265358979323846264336587157647220816609616142031751236996050e+00  -1.7e-26
3.14159265358979323846264338327934082455883055775981969224993464280e+00  -1.6e-31
3.14159265358979323846264338327950288373651901114010052831545166960e+00  -4.6e-37
3.14159265358979323846264338327950288419716900588321601858149548700e+00  -3.9e-43
3.14159265358979323846264338327950288419716939937500618228596937800e+00  -1.0e-49
3.14159265358979323846264338327950288419716939937510582096755395090e+00  -7.4e-57
3.14159265358979323846264338327950288419716939937510582097494459220e+00  -5.7e-65
3.14159265358979323846264338327950288419716939937510582097494459040e+00  -1.9e-63
3.14159265358979323846264338327950288419716939937510582097494458900e+00  -3.3e-63
3.14159265358979323846264338327950288419716939937510582097494458750e+00  -4.8e-63
elapsed time : 2.040 seconds
```

The functions defining the composite Trapezoidal rule and Romberg integration in C++ are listed below.

```
vector<qd_real> comptrap ( qd_real f ( qd_real x ), qd_real a, qd_real b, int n )
{
   vector<qd_real> t(n);
   qd_real h = b - a;

   t[0] = (f(a) + f(b))*h/2;
   for(int i=1, m=1; i<n; i++, m=m*2) {
      h = h/2;
      t[i] = 0.0;
      for(int j=0; j<m; j++)
         t[i] += f(a+h+j*2*h);
      t[i] = t[i-1]/2 + h*t[i];
   }
   return t;
}
void romberg_extrapolation ( vector<qd_real>& t )
{
   int n = t.size();
   qd_real e[n][n];
   int m = 0;

   for(int i=0; i<n; i++) {
      e[i][0] = t[i];
      for(int j=1; j<n; j++) e[i][j] = 0.0;
   }
   for(int i=1; i<n; i++) {
      for(int j=1, m=2; j<=i; j++, m=m+2) {
         qd_real r = pow(2.0,m);
         e[i][j] = (r*e[i][j-1] - e[i-1][j-1])/(r-1);
      }
   }
   for(int i=1; i<n; i++) t[i] = e[i][i];
}
```

4.2.2 Parallel Numerical Integration

Running the OpenMP implementation using 2 cores on of a 2.60 GHz processor:

```
$ time /tmp/romberg4piqd 20
...
elapsed time : 2.040 seconds
```

```
real    0m2.046s
user    0m2.042s
sys     0m0.000s
$
```

Using two threads with OpenMP (speed up: `2.046/1.052 = 1.945`), run as below.

```
$ time /tmp/romberg4piqd_omp 20
...
elapsed time : 2.080 seconds

real    0m1.052s
user    0m2.081s
sys     0m0.003s
$
```

The code that needs to run in parallel is the most computational intensive stage, which is in the computation of the composite Trapezoidal rule.

```
for(int i=1, m=1; i<n; i++, m=m*2)
{
   h = h/2;
   t[i] = 0.0;
   for(int j=0; j<m; j++)
      t[i] += f(a+h+j*2*h);
   t[i] = t[i-1]/2 + h*t[i];
}
```

All function evaluations in the `j` loop can be computed independently from each other. The parallel code with OpenMP is listed below.

```
int id,jstart,jstop;
qd_real val;
for(int i=1, m=1; i<n; i++, m=m*2)
{
   h = h/2;
   t[i] = 0.0;
   #pragma omp parallel private(id,jstart,jstop,val)
   {
      id = omp_get_thread_num();
      jstart = id*m/2;
      jstop = (id+1)*m/2;
      for(int j=jstart; j<jstop; j++)
         val += f(a+h+j*2*h);
      #pragma omp critical
         t[i] += val;
   }
   t[i] = t[i-1]/2 + h*t[i];
}
```

The command `omp_set_num_threads(2);` is executed before `comptrap`, the function with the composite Trapezoidal rule is called. The benefits of OpenMP are twofold.

1. The threads are computing inside the `j` loop, inside the `i` loop of the function `comptrap`. ⇒ OpenMP does not create, join, destroy all threads for every different value of `i`, reducing system time.
2. The threads must wait at the end of each loop to update the approximation for the integral and to proceed to the

next `i`. ⇒ OpenMP takes care of the synchronization of the threads.

To build the executable, we have to tell the gcc compiler that we are using

- the OpenMP library: `gcc -fopenmp`; and
- the QD library, including headers and `libqd.a` file.

If the file `makefile` contains the lines:

```
QD_ROOT=/usr/local/qd-2.3.17
QD_LIB=/usr/local/lib

romberg4piqd_omp:
        g++ -fopenmp -I$(QD_ROOT)/include \
            romberg4piqd_omp.cpp \
            $(QD_LIB)/libqd.a \
            -o romberg4piqd_omp
```

then we can simply type `make romberg4piqd_omp` at the command prompt to build the executable.

using the Intel Threading Building Blocks

The Intel TBB implementation provides an opportunity to illustrate the `parallel_reduce` construction.

We adjust our previous class `SumIntegers` to sum a sequence of quad doubles. In the header file, we include the headers for the QD library and the header files we need from the Intel TBB.

```
#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <qd/qd_real.h>
#include "tbb/tbb.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_reduce.h"
#include "tbb/task_scheduler_init.h"

using namespace std;
using namespace tbb;
```

Then in the class definition, we start with the data attributes: the address to the start of the sequence of quad doubles and the sum of the sequence. The constructor copies the given address to the data attribute which stores the address of the sequence. The constructor initializes the sum to zero.

```
class SumQuadDoubles
{
   qd_real *data;
   public:
      qd_real sum;
      SumQuadDoubles ( qd_real *d ) : data(d), sum(0.0) {}
```

The parallelism is defined via the `blocked_range`, and by the operators to split and join.

```
void operator()
   ( const blocked_range<size_t>& r )
{
   qd_real s = qd_real(sum[0],sum[1],sum[2],sum[3]);
   // must accumulate !
   qd_real *d = data;
```

```
   size_t end = r.end();
   for(size_t i=r.begin(); i != end; ++i)
      s += d[i];
   sum = qd_real(s[0],s[1],s[2],s[3]);
}
// the splitting constructor
SumQuadDoubles ( SumQuadDoubles& x, split ) :
   data(x.data), sum(0.0) {}
// the join method does the merge
void join ( const SumQuadDoubles& x ) { sum += x.sum; }
```

The class definition is used in the following function:

```
qd_real ParallelSum ( qd_real *x, size_t n )
{
   SumQuadDoubles S(x);

   parallel_reduce(blocked_range<size_t>(0,n), S);

   return S.sum;
}
```

The `ParallelSum` is called in the main program as follows:

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

task_scheduler_init init(task_scheduler_init::automatic);
qd_real s = ParallelSum(d,n);
```

To compile `parsumqd_tbb.cpp`, we have to tell `g++` to

- use the QD library (include headers, link `libqd.a`); and
- use the Intel TBB library (headers and `-ltbb`).

Therefore, the `makefile` should contain the following:

```
QD_ROOT=/usr/local/qd-2.3.17
QD_LIB=/usr/local/lib
TBB_ROOT=/usr/local/tbb40_233oss

parsumqd_tbb:
        g++ -I$(TBB_ROOT)/include \
            -I$(QD_ROOT)/include \
            parsumqd_tbb.cpp -o /tmp/parsumqd_tbb \
            $(QD_LIB)/libqd.a -L$(TBB_ROOT)/lib -ltbb
```

For our numerical experiment, we sum as many as `n` numbers in an array of quad doubles, starting with 1,2,3 ... so the sum equals $n(n+1)/2$.

```
$ time /tmp/parsumqd_tbb 20160921
 S = 2.03231377864581000 ... omitted ... 000e+14
 T = 2.03231377864581000 ... omitted ... 000e+14

real    0m0.765s
user    0m8.231s
```

```
sys     0m0.146s
$
```

We estimate the speed up (done on a 16-core computer) comparing user time to wall clock time: `8.231/0.765 = 10.759`.

The work stealing scheme for `parallel_reduce` is explained in the Intel Threading Building Blocks tutorial, in section 3.3.

4.2.3 Bibliography

1. 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.
2. A. Yazici. **The Romberg-like Parallel Numerical Integration on a Cluster System**. In the proceedings of the 24th International Symposium on Computer and Information Sciences, ISCIS 2009, pages 686-691, IEEE 2009.
3. D.H. Bailey and J.M. Borwein. **Highly Parallel, High-Precision Numerical Integration**. April 2008. Report LBNL-57491. Available at <http://crd-legacy.lbl.gov/}sim{tt dhbailey/dhbpapers/>.

4.2.4 Exercises

1. Make the OpenMP implementation of `romberg4piqd_omp.cpp` more general by prompting the user for a number of threads and then using those threads in the function `comptrap`. Compare the speed up for 2, 4, 8, and 16 threads.
2. Write an elaborate description on the thread creation and synchronization issues with Pthreads to achieve a parallel version of `romberg4piqd.cpp`.
3. Use the Intel Threading Building Blocks to write a parallel version of the composite trapezoidal rule in quad double arithmetic.

4.3 Parallel FFT and Isoefficiency

As our third application for parallel algorithms, we consider the Fast Fourier Transform (FFT). Instead of investigating the algorithmic aspects in detail, we introduce the FFT with an excellent software library FFTW which supports for OpenMP. The FFT is then used in the second part to illustrate the notion of isoefficiency, to complement the scalabiliy treatments in the laws of Ahmdahl and Gustafson.

4.3.1 The Fast Fourier Transform in Parallel

A periodic function $f(t)$ can be written as a series of sinusoidal waveforms of various frequencies and amplitudes: the Fourier series. The Fourier transform maps f from the time to the frequency domain. In discrete form:

$$F_k = \frac{1}{n} \sum_{j=0}^{n-1} f_j e^{-2\pi i(jk/n)}, \quad k = 0, 1, \ldots, n-1, f_k = f(x_k).$$

The Discrete Fourier Transform (DFT) maps a convolution into a componentwise product. The Fast Fourier Transform is an algorithm that reduces the cost of the DFT from $O(n^2)$ to $O(n \log(n))$, for length n. The are many applications, for example: signal and image processing.

FFTW (the Fastest Fourier Transform in the West) is a library for the Discrete Fourier Transform (DFT), developed at MIT by Matteo Frigo and Steven G. Johnson available under the GNU GPL license at <http://www.fftw.org>. FFTW received the 1999 J. H. Wilkinson Prize for Numerical Software. FFTW 3.3.3 supports MPI and comes with multithreaded versions: with Cilk, Pthreads and OpenMP are supported. Before `make install`, do

```
./configure --enable-mpi --enable-openmp --enable-threads
```

We illustrate the use of FFTW for signal processing.

Consider $f(t) = 2\cos(4 \times 2\pi t) + 5\sin(10 \times 2\pi t)$, for $t \in [0, 1]$. Its plot is in Fig. 4.1.

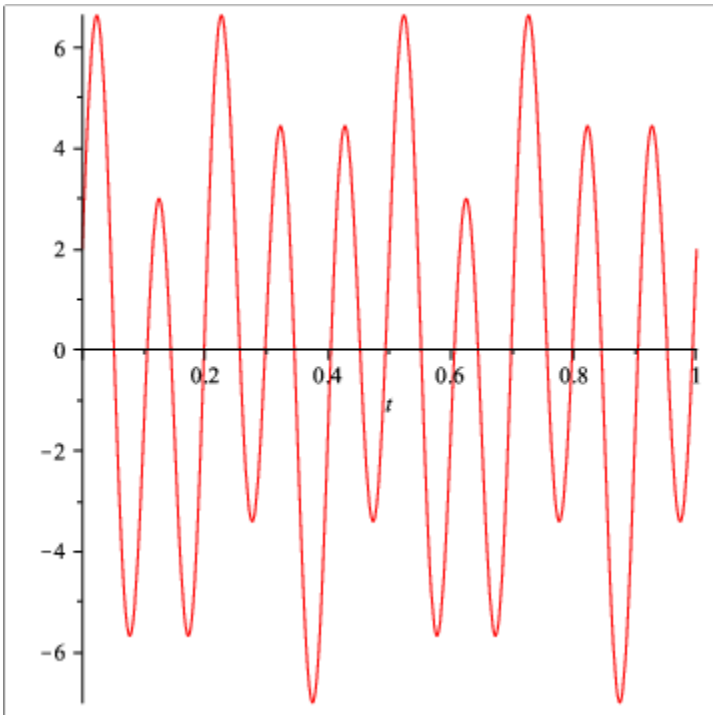


Fig. 4.1: A simple test signal.

Compiling and running the code below gives the frequencies and amplitudes of the test signal.

```
$ make fftw_use
gcc fftw_use.c -o /tmp/fftw_use -lfftw3 -lm
$ /tmp/fftw_use
scanning through the output of FFTW...
at 4 : (2.56000e+02,-9.63913e-14)
=> frequency 4 and amplitude 2.000e+00
at 10 : (-7.99482e-13,-6.40000e+02)
=> frequency 10 and amplitude 5.000e+00
$
```

Calling the FFTW software on a test signal:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <fftw3.h>
```

```
double my_signal ( double t );
/*
 * Defines a signal composed of a cosine of frequency 4 and amplitude 2,
 * a sine of frequency 10 and amplitude 5. */

void sample_signal ( double f ( double t ), int m, double *x, double *y );
/*
 * Takes m = 2^k samples of the signal f, returns abscisses in x and samples in y. */

int main ( int argc, char *argv[] )
{
   const int n = 256;
   double *x, *y;
   x = (double*)calloc(n,sizeof(double));
   y = (double*)calloc(n,sizeof(double));
   sample_signal(my_signal,n,x,y);

   int m = n/2+1;
   fftw_complex *out;
   out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*m);

   fftw_plan p;
   p = fftw_plan_dft_r2c_1d(n,y,out,FFTW_ESTIMATE);
   fftw_execute(p);
   printf("scanning through the output of FFTW...\n");
   double tol = 1.0e-8;  /* threshold on noise */
   int i;
   for(i=0; i<m; i++)
   {
      double v = fabs(out[i][0])/(m-1)
               + fabs(out[i][1])/(m-1);
      if(v > tol)
      {
         printf("at %d : (%.5e,%.5e)\n",
                i,out[i][0],out[i][1]);
         printf("=> frequency %d and amplitude %.3e\n",
                i,v);
      }
   }
   return 0;
}
```

running the OpenMP version of FFTW

For timing purposes, we define a program that takes in the parameters at the command line. The parameters are `m` and `n`: We will run the fftw `m` times for dimension `n`. The setup of our experiment is in the code below followed by the parallel version with OpenMP.

Setup of the experiment to execute the fftw:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <fftw3.h>

void random_sequence ( int n, double *re, double *im );
```

```
/*
 * Returns a random sequence of n complex numbers
 * with real parts in re and imaginary parts in im. */

int main ( int argc, char *argv[] )
{
   int n,m;
   if(argc > 1)
      n = atoi(argv[1]);
   else
      printf("please provide dimension on command line\n");
   m = (argc > 2) ? atoi(argv[2]) : 1;
   double *x, *y;
   x = (double*)calloc(n,sizeof(double));
   y = (double*)calloc(n,sizeof(double));
   random_sequence(n,x,y);
   fftw_complex *in,*out;
   in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*n);
   out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*n);
   fftw_plan p;
   p = fftw_plan_dft_1d(n,in,out,FFTW_FORWARD,FFTW_ESTIMATE);
   clock_t tstart,tstop;
   tstart = clock();
   int i;
   for(i=0; i<m; i++) fftw_execute(p);
   tstop = clock();
   printf("%d iterations took %.3f seconds\n",m,(tstop-tstart)/((double) CLOCKS_PER_
↪SEC));
   return 0;
}
```

Code to runt the OpenMP version of FFTW is below.

```
#include <omp.h>
#include <fftw3.h>

int main ( int argc, char *argv[] )
{
   int t;  /* number of threads */

   t = (argc > 3) ? atoi(argv[3]) : 1;

   int okay = fftw_init_threads();
   if(okay == 0)
      printf("error in thread initialization\n");
   omp_set_num_threads(t);
   fftw_plan_with_nthreads(t);
```

To define the compilation in the makefile, we must link with the OpenMP library version fftw3 and with the fftw3 library:

```
fftw_timing_omp:
        gcc -fopenmp fftw_timing_omp.c -o /tmp/fftw_timing_omp \
            -lfftw3_omp -lfftw3 -lm
```

Results are in Table 4.1 and Table 4.2.

Table 4.1: Running times (with the unix command `time`), for computing 1,000 DFTs for $n = 100,000$, with p threads, where real = wall clock time, user = cpu time, sys = system time, obtained on two 8-core processors at 2.60 Ghz.

p	real	user	sys	speed up
1	2.443s	2.436s	0.004s	1
2	1.340s	2.655s	0.010s	1.823
4	0.774s	2.929s	0.008s	3.156
8	0.460s	3.593s	0.017s	5.311
16	0.290s	4.447s	0.023s	8.424

Table 4.2: Running times (with the unix command `time`), for computing 1,000 DFTs for $n = 1,000,000$, with p threads, where real = wall clock time, user = cpu time, sys = system time, obtained on two 8-core processors at 2.60 Ghz.

p	real	user	sys	speed up
1	44.806s	44.700s	0.014s	1
2	22.151s	44.157s	0.020s	2.023
4	10.633s	42.336s	0.019s	4.214
8	6.998s	55.630s	0.036s	6.403
16	3.942s	62.250s	0.134s	11.366

Comparing the speedups in Table 4.1 with those in Table 4.2, we observe that speedups improve with a ten fold increase of n. Next we introduce another way to think about scalability.

4.3.2 Isoefficiency

Before we examine how efficiency 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

$$T_p = \frac{W + T_0(W,p)}{p}, \quad \text{Speedup } S(p) = \frac{W}{T_p} = \frac{Wp}{W + T_0(W,p)}.$$

The efficiency

$$E(p) = \frac{S(p)}{p} = \frac{W}{W + T_0(W,p)} = \frac{1}{1 + T_0(W,p)/W}.$$

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:

$$\begin{aligned} E = \frac{1}{1+T_0(W,p)/W} &\Rightarrow \frac{1}{E} = \frac{1+T_0(W,p)/W}{1} \\ &\Rightarrow \frac{1}{E} - 1 = \frac{T_0(W,p)}{W} \\ &\Rightarrow \frac{1-E}{E} = \frac{T_0(W,p)}{W}. \end{aligned}$$

The isoefficiency function is

$$W = \left(\frac{E}{1-E}\right) T_0(W,p) \quad \text{or} \quad W = K\,T_0(W,p).$$

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. 4.2 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):

$$E(p) = \frac{Wt_c}{Wt_c + n\log(p)t_w}.$$

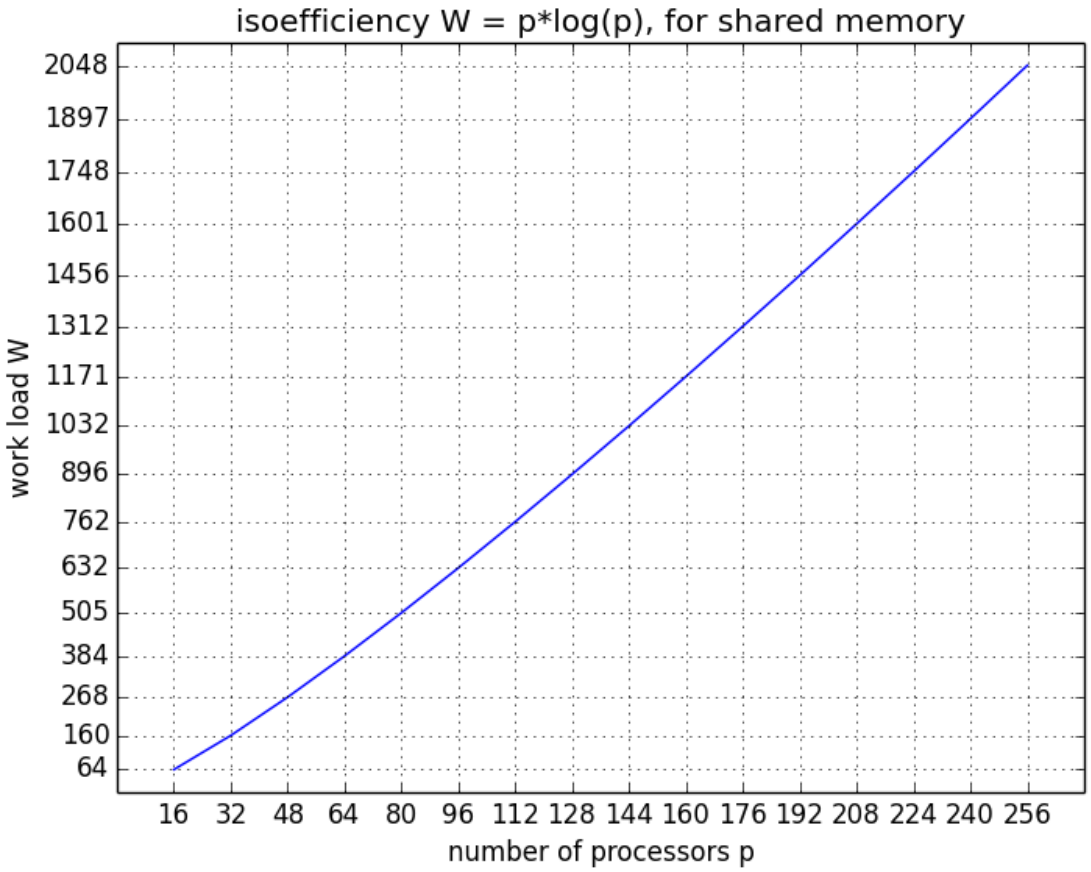


Fig. 4.2: Isoefficiency for a shared memory application.

We want to express $K = \frac{E}{1-E}$, using $\frac{1}{K} = \frac{1-E}{E} = \frac{1}{E} - 1$:

$$\frac{1}{K} = \frac{Wt_c + n\log(p)t_w}{Wt_c} - \frac{Wt_c}{Wt_c} \quad \Rightarrow \quad W = K\left(\frac{t_w}{t_c}\right) n\log(p).$$

In Fig. 4.3 the efficiency function is displayed for an increasing number of processors and various values of the dimension.

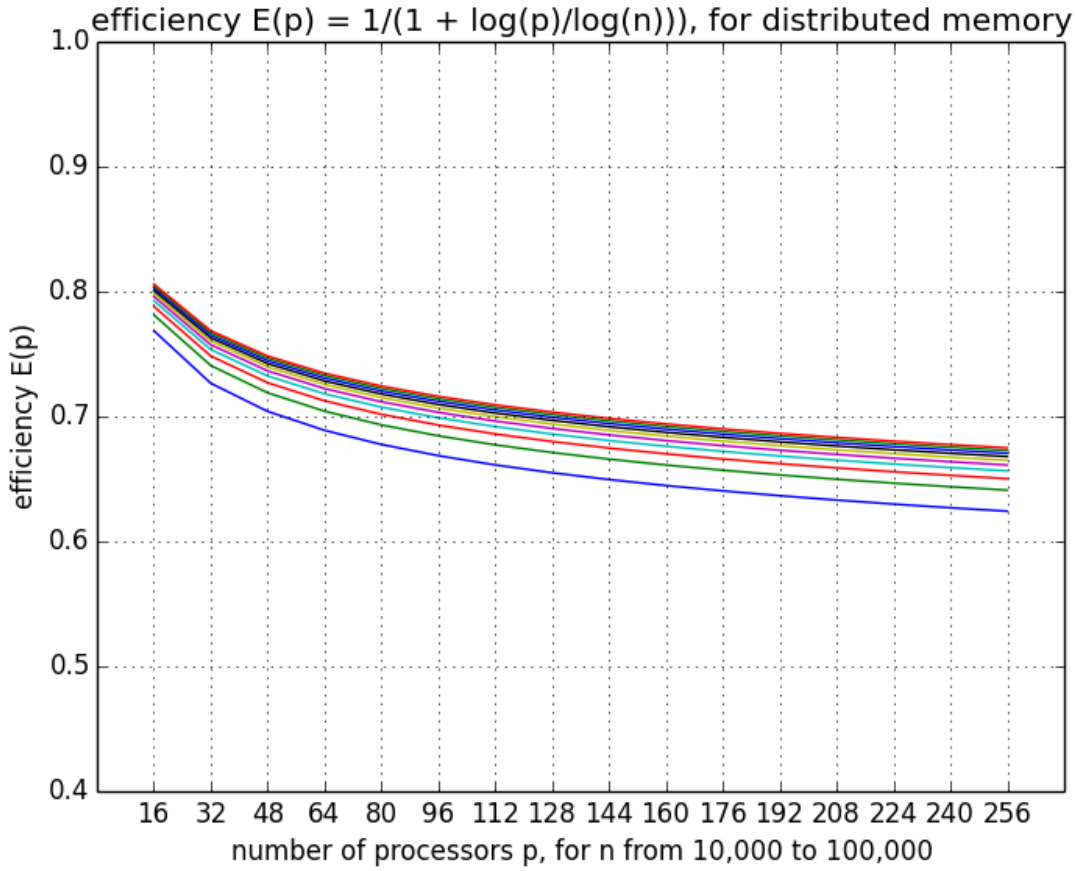


Fig. 4.3: Scalability analysis with a plot of the efficiency function.

4.3.3 Bibliography

1. Matteo Frigo and Steven G. Johnson: **FFTW for version 3.3.3, 25 November 2012**. Manual available at <http://www.fftw.org>.
2. Matteo Frigo and Steven G. Johnson: **The Design and Implementation of FFTW3**. *Proc. IEEE* 93(2): 216–231, 2005.
3. Vipin Kumar and Anshul Gupta: **Analyzing Scalability of Parallel Algorithms and Architectures**. *Journal of Parallel and Distributed Computing* 22: 379–391, 1994.
4. Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar: **Introduction to Parallel Computing**. 2nd edition, Pearson 2003. Chapter 13 is devoted to the Fast Fourier Transform.

5. 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.

4.3.4 Exercises

1. Investigate the speed up for the parallel FFTW with threads.
2. Consider the isoefficiency formulas we derived for a parallel version of the FFT. Suppose an efficiency of 0.6 is desired. For values $t_c = 1$, $t_s = 25$ and $t_w = 4$, make plots of the speedup for increasing values of n, taking $p = 64$. Interpret the plots relating to the particular choices of the parameters t_c, t_s, t_w, and the desired efficiency.
3. Relate the experimental speed ups with the theoretical isoefficiency formulas.

CHAPTER 5

Pipelining

5.1 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.

5.1.1 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. 5.1.

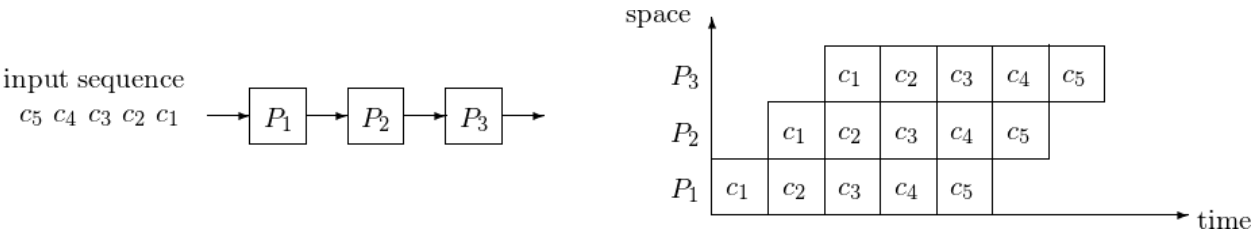


Fig. 5.1: A schematic of a 3-stage pipeline at the left, with the corresponding space-time diagram at the right. After 3 time units, one car per time unit is completed. It takes 7 time units to complete 5 cars.

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:

$$\lim_{n \to \infty} \frac{p \times n}{n+p-1} = p.$$

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. 5.2.

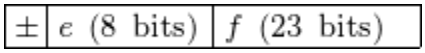


Fig. 5.2: A floating-point number has a sign bit, exponent, and fraction.

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. 5.3.

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. 5.4. Observe: the consumption of a signal is sequential.

5.1.2 Pipeline Implementations

A ring topology of processors is a natural way to implement a pipeline. In Fig. 5.5, the stages in a pipeline are performed by the processes organized 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.

5.1.3 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 mod p. A session of an MPI implementation of one pipeline cycle for this calculation shows the following:

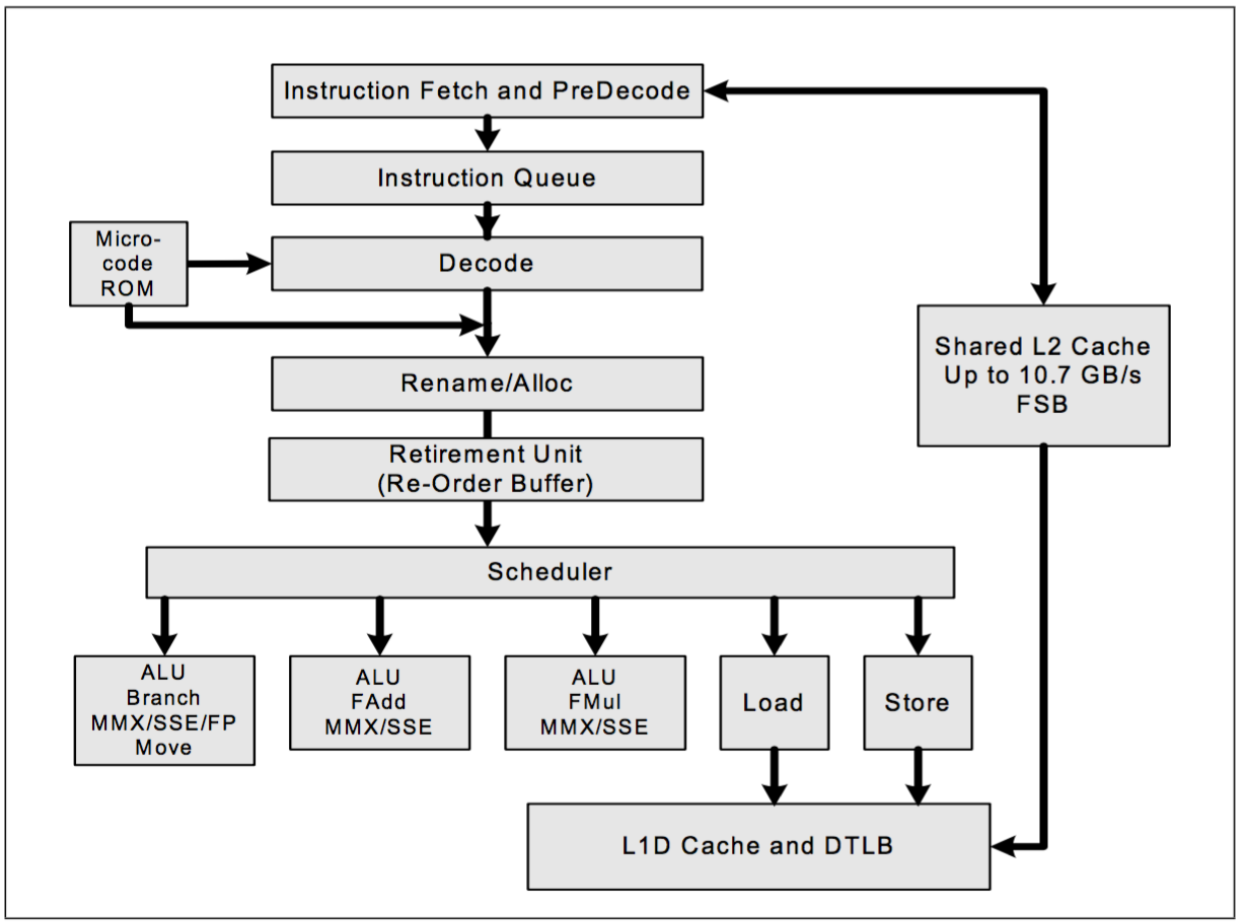


Figure 2-3. The Intel Core Microarchitecture Pipeline Functionality

Fig. 5.3: Copied from the Intel Architecture Software Developer's Manual.

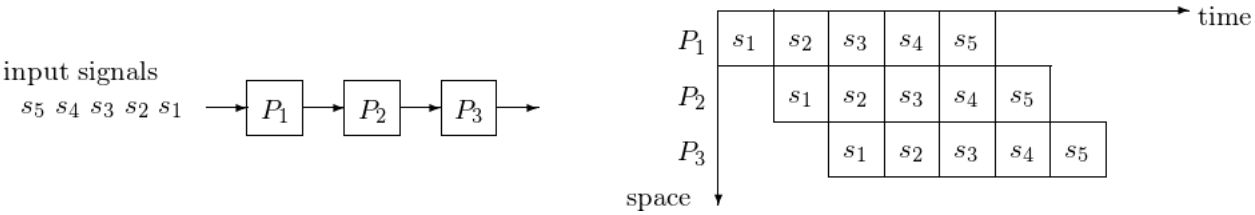


Fig. 5.4: Pipeline to denoising a signal at the left, with space-diagram at the right.

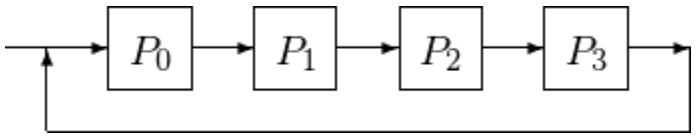


Fig. 5.5: Four stages in a pipeline executed by four processes in a ring.

```
$ 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);
   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);
}
```

pipelined addition

Consider 4 processors in a ring topology as in Fig. 5.5. To add a sequence of 32 numbers, with data partitioning:

$$\underbrace{a_0, a_1, \ldots, a_7}_{k}, \underbrace{b_0, b_1, \ldots, b_7}_{k}, \underbrace{c_0, c_1, \ldots, c_7}_{k}, \underbrace{d_0, d_1, \ldots, d_7}_{k}.$$

$$A_k = \sum_{j=0}^{k} a_j \quad B_k = \sum_{j=0}^{k} b_j \quad C_k = \sum_{j=0}^{k} c_j \quad D_k = \sum_{j=0}^{k} d_j$$

The final sum is $S = A_7 + B_7 + C_7 + D_7$. Fig. 5.6 shows the space-time diagram for pipeline addition.

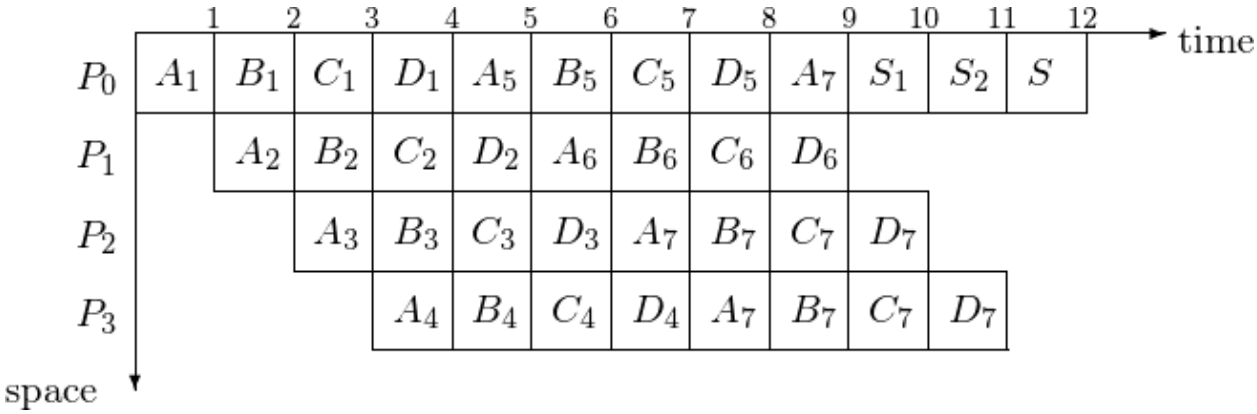


Fig. 5.6: Space-time diagram for pipelined addition, where $S_1 = A_7 + B_7, S_2 = S_1 + C_7, S = S_2 + D_7$.

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*: $\lim_{n \to \infty} S(p) = p$.

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 */
   {
      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]);
   }
}
```

5.1.4 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 d given by its coefficient vector $(a_0, a_1, a_2, \ldots, a_d)$, using Horner's method, e.g., for $d = 4$: $f(x) = (((a_4x + a_3)x + a_2)x + a_1)x + a_0$. Give MPI code of this algorithm to evaluate f at a sequence of n values for x by a p-stage pipeline.

5.2 Pipelined Sorting

We continue our study of pipelined computations, but now for shared memory parallel computers. The Intel Threading Building Blocks provide support for pipeline patterns.

5.2.1 Pipelines with Intel Threading Building Blocks (TBB)

The Intel Threading Building Blocks (TBB) classes `pipeline` and `filter` implement the pipeline pattern. A 3-stage pipeline is shown in Fig. 5.7.

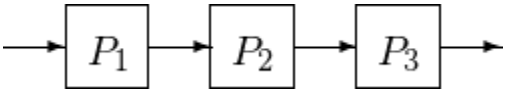


Fig. 5.7: A 3-stage pipeline.

A session with a program using the Intel TBB goes as follows:

```
$ /tmp/pipe_tbb
 the input sequence : 1 -2 3 -4
the output sequence : 8 -16 24 -32
$
```

The `makefile` contains

```
TBB_ROOT = /usr/local/tbb40_20131118oss
pipe_tbb:
       g++ -I$(TBB_ROOT)/include -L$(TBB_ROOT)/lib \
            pipe_tbb.cpp -o /tmp/pipe_tbb -ltbb
```

and `pipe_tbb.cpp` starts with

```
#include <iostream>
#include "tbb/pipeline.h"
#include "tbb/compat/thread"
#include "tbb/task_scheduler_init.h"
using namespace tbb;
int Sequence[] = {1,-2,3,-4,0}; // 0 is sentinel
```

The inheriting from the `filter` class is done as below:

```
class DoublingFilter: public filter {
   int* my_ptr;
   public:
      DoublingFilter() :
         filter(serial_in_order), my_ptr(Sequence) {}
         // process items one at a time in the given order
      void* operator()(void*) {
         if(*my_ptr) {
            *my_ptr = (*my_ptr)*2;  // double item
            return (void*)my_ptr++; // pass to next filter
         }
         else
            return NULL;
      }
};
```

A `thread_bound_filter` is a filter explicitly serviced by a particular thread, in this case the main thread:

```
class OutputFilter: public thread_bound_filter
{
   public:
      OutputFilter() :
         thread_bound_filter(serial_in_order) {}
      void* operator()(void* item)
      {
         int *v = (int*)item;
         std::cout << " " << (*v)*2;
         return NULL;
      }
};
```

The argument of `run` is the maximum number of live tokens.

```
void RunPipeline ( pipeline* p )
{
   p->run(8);
}
```

The pipeline runs until the first filter returns `NULL` and each subsequent filter has processed all items from its predecessor. In the function `main()`:

```
// another thread initiates execution of the pipeline
std::thread t(RunPipeline,&p);
```

The creation of a pipeline in the main program happens as follows:

```
int main ( int argc, char* argv[] )
{
   std::cout << " the input sequence :";
```

```
   for(int* s = Sequence; (*s); s++)
      std::cout << " " << *s;
   std::cout << "\nthe output sequence :";

   DoublingFilter f;   // construct the pipeline
   DoublingFilter g;
   OutputFilter h;

   pipeline p;
   p.add_filter(f); p.add_filter(g); p.add_filter(h);
   // another thread initiates execution of the pipeline
   std::thread t(RunPipeline,&p);
   // process the thread_bound_filter
   // with the current thread
   while(h.process_item()
         != thread_bound_filter::end_of_stream)
      continue;
   // wait for pipeline to finish on the other thread
   t.join();
   std::cout << "\n";

   return 0;
}
```

5.2.2 Sorting 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);
      }
   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,
                  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);
}
```

5.2.3 Prime Number Generation

The sieve of Erathostenes is shown in Fig. 5.8.

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.

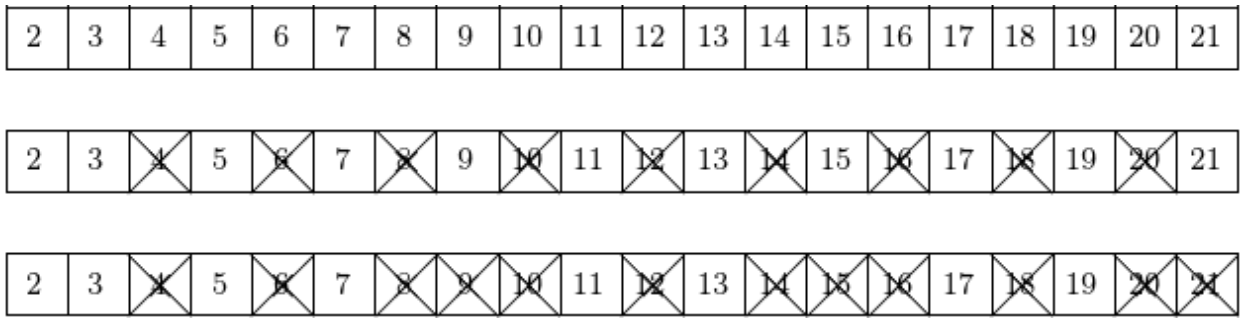


Fig. 5.8: Wiping out all multiples of 2 and 3 gives all prime numbers between 2 and 21.

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. 5.9.

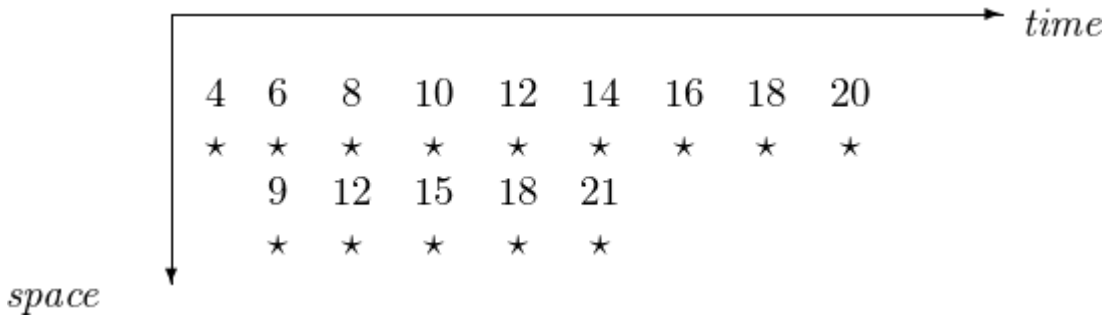


Fig. 5.9: A 2-stage pipeline to compute all primes ≤ 21.

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)
#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
$
```

5.2.4 Exercises

1. Consider the evaluation of a polynomial $f(x)$ of degree d given by its coefficient vector $(a_0, a_1, a_2, \ldots, a_d)$, using Horner's method, e.g., for $d = 4$: $f(x) = (((a_4x + a_3)x + a_2)x + a_1)x + a_0$. Give code of this algorithm to evaluate f at a sequence of n values for x by a p-stage pipeline, using the Intel TBB.
2. Write a pipeline with the Intel TBB to implement the parallel version of insertion sort we have done with MPI.
3. Use Pthreads to implement the parallel version of insertion sort we have done with MPI.

5.3 Solving Triangular Systems

We apply a type 3 pipeline to solve a triangular linear system. Rewriting the formulas for the forward substitution we arrive at an implementation for shared memory computers with OpenMP. For accurate results, we compute with quad double arithmetic.

5.3.1 Forward Substitution Formulas

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:

$$
\begin{cases}
y_1 & = & b_1 \\
\ell_{2,1}y_1 + y_2 & = & b_2 \\
\ell_{3,1}y_1 + \ell_{3,2}y_2 + y_3 & = & b_3 \\
 & \vdots & \\
\ell_{n,1}y_1 + \ell_{n,2}y_2 + \ell_{n,3}y_3 + \cdots + \ell_{n,n-1}y_{n-1} + y_n & = & b_n
\end{cases}
$$

and solving for the diagonal elements gives

$$
\begin{array}{rcl}
y_1 & = & b_1 \\
y_2 & = & b_2 - \ell_{2,1}y_1 \\
y_3 & = & b_3 - \ell_{3,1}y_1 - \ell_{3,2}y_2 \\
 & \vdots & \\
y_n & = & b_n - \ell_{n,1}y_1 - \ell_{n,2}y_2 - \cdots - \ell_{n,n-1}y_{n-1}
\end{array}
$$

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. 5.10.

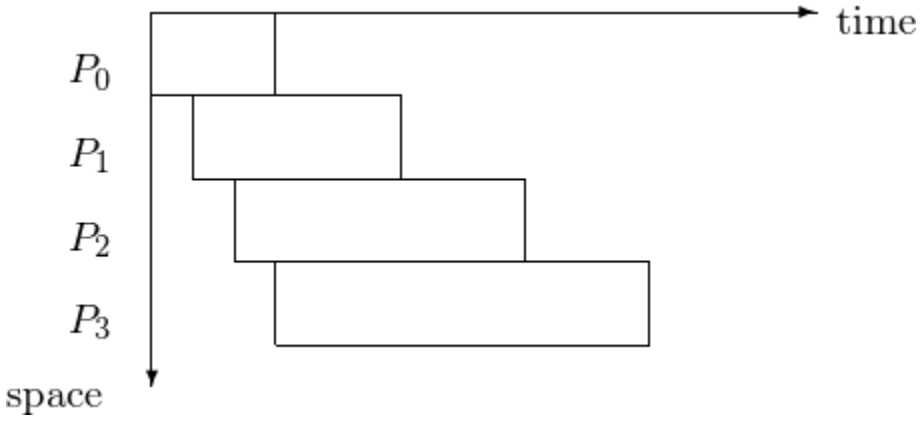


Fig. 5.10: Space-time diagram for pipeline with stages of varying length.

5.3.2 Parallel Solving

Using an n-stage pipeline, we assume that L is available on every processor.

The corresponding 4-stage pipeline is shown in Fig. 5.11 with the space-time diagram in Fig. 5.12.

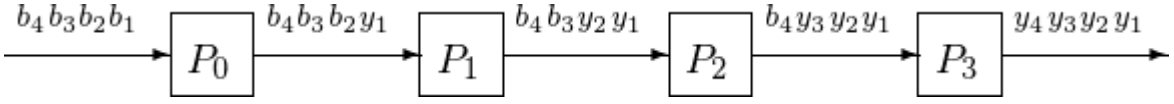


Fig. 5.11: 4-stage pipeline to solve a 4-by-4 lower triangular system.

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. 5.13. The corresponding space-time diagram is in Fig. 5.14 and the space-time diagram in Fig. 5.15 shows at what time step which component of the solution is.

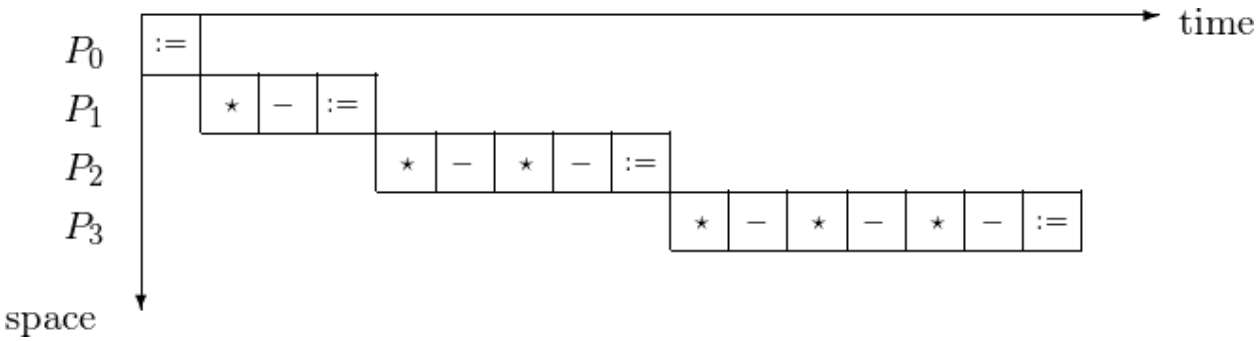


Fig. 5.12: Space-time diagram for solving a 4-by-4 lower triangular system.

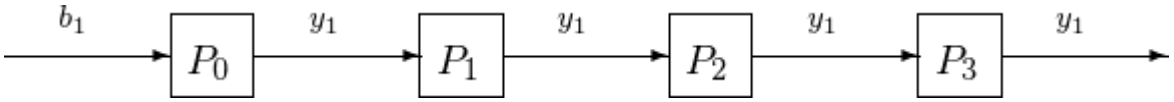


Fig. 5.13: Passing y_1 through the type 3 pipeline.

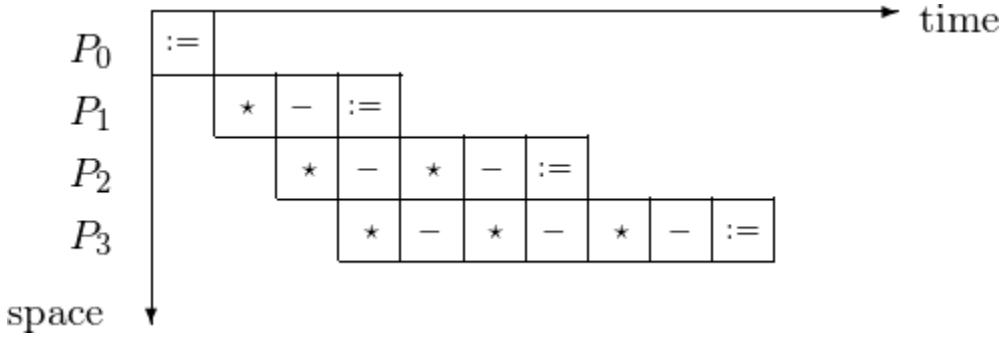


Fig. 5.14: Space-time diagram of a type 3 pipeline.

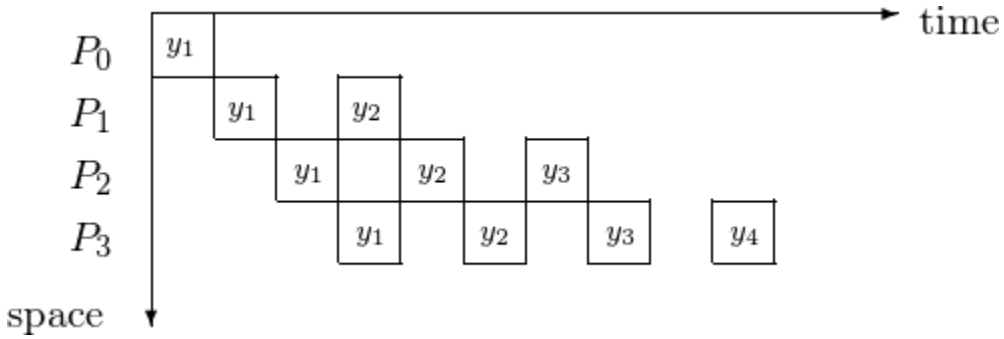


Fig. 5.15: Space-time diagram illustrates the component of the solutions.

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. $y_3 := y_3 - \ell_{3,2} \star y_2$;

   $y_4 := y_4 - \ell_{4,2} \star y_2$;

   $y_5 := y_5 - \ell_{5,2} \star y_2$;
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.

5.3.3 Parallel Solving with OpenMP

For our parallel solver for triangular systems, the setup is as follows. 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 $\mathbf{b} = L\mathbf{y}$.

For dimensions $n > 800$, hardware doubles are insufficient. With hardware doubles, the accumulation of round off is such that we loose all accuracy in y_n. We recall that condition numbers of n-dimensional triangular systems can grow as large as 2^n. Therefore, we use quad double arithmetic.

Relying on hardware doubles is problematic:

```
$ time /tmp/trisol 10
last number : 1.0000000000000009e+00

real    0m0.003s    user    0m0.001s    sys    0m0.002s

$ time /tmp/trisol 100
last number : 9.9999999999974221e-01

real    0m0.005s    user    0m0.001s    sys    0m0.002s

$ time /tmp/trisol 1000
last number : 2.7244600009080568e+04
```

```
real    0m0.036s    user    0m0.025s    sys    0m0.009s
```

Allocating a matrix of quad double in the main program is done as follows:

```
{
   qd_real b[n],y[n];
   int i,j;

   qd_real **L;
   L = (qd_real**) calloc(n,sizeof(qd_real*));
   for(i=0; i<n; i++)
      L[i] = (qd_real*) calloc(n,sizeof(qd_real));

   srand(time(NULL));
   random_triangular_system(n,L,b);
```

The function for a random triangular system is defined next:

```
void random_triangular_system ( int n, qd_real **L, qd_real *b )
{
   int i,j;
   for(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(j=i+1; j<n; j++)
         L[i][j] = qd_real(0.0);
   }
   for(i=0; i<n; i++)
   {
      b[i] = qd_real(0.0);
      for(j=0; j<n; j++)
         b[i] = b[i] + L[i][j];
   }
}
```

Solving the system is done by the function below.

```
void solve_triangular_system_swapped ( int n, qd_real **L, qd_real *b, qd_real *y )
{
   int i,j;

   for(i=0; i<n; i++) y[i] = b[i];

   for(i=1; i<n; i++)
   {
      for(j=i; j<n; j++)
         y[j] = y[j] - L[j][i-1]*y[i-1];
   }
}
```

Now we insert the OpenMP directives:

```
void solve_triangular_system_swapped ( int n, qd_real **L, qd_real *b, qd_real *y )
{
   int i,j;

   for(i=0; i<n; i++) y[i] = b[i];

   for(i=1; i<n; i++)
   {
      #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];
      }
   }
}
```

Running `time /tmp/trisol_qd_omp n p`, for dimension $n = 8,000$ for varying number p of cores gives times as in table Table 5.1.

Table 5.1: Times on solving a triangular system with quad doubles.

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.

5.3.4 Exercises

1. 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.
2. 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.
3. 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.

CHAPTER 6

Synchronized Computations

6.1 Barriers for Synchronizations

For message passing, we distinguish between a linear, a tree, and a butterfly barrier. We end with a simple illustration of barriers with Pthreads.

6.1.1 Synchronizing Computations

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. 6.1 for a linear and a 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. 6.2.

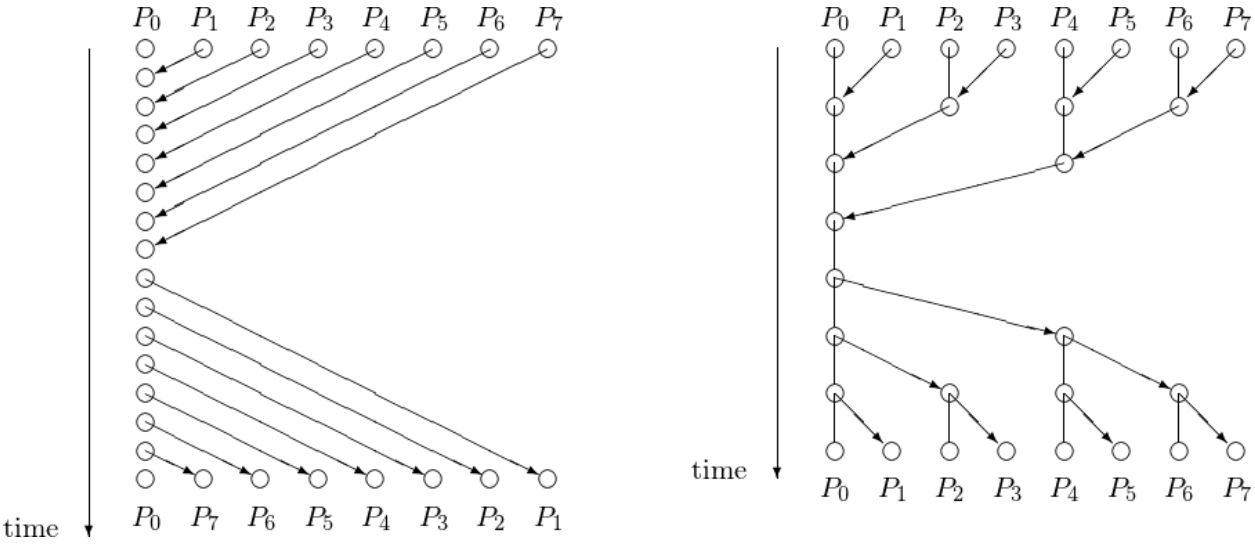


Fig. 6.1: A linear next to a tree barrier to synchronize 8 processes. For 8 processes, the linear barrier takes twice as many time steps as the tree barrier.

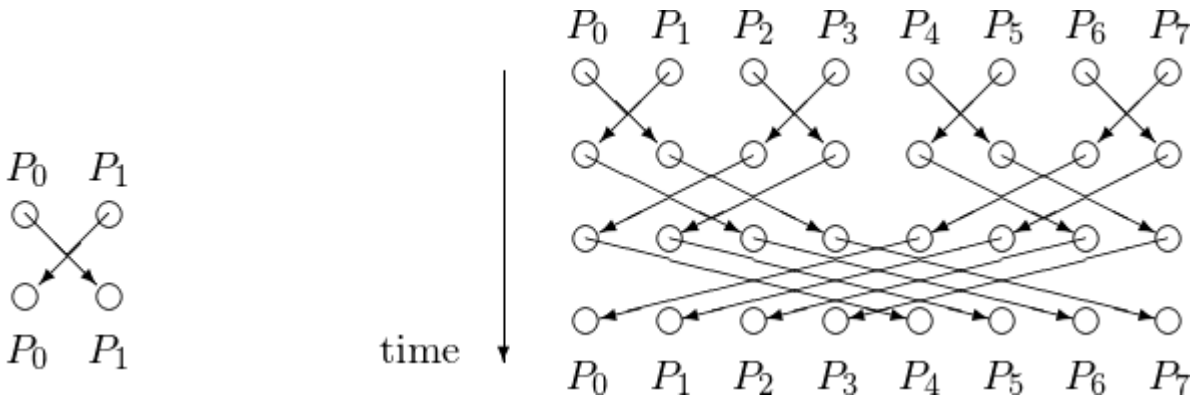


Fig. 6.2: Two processes can synchronize in one step as shown on the left. At the right is a schematic of the time steps for a tree barrier to synchronize 8 processes.

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. 6.3.

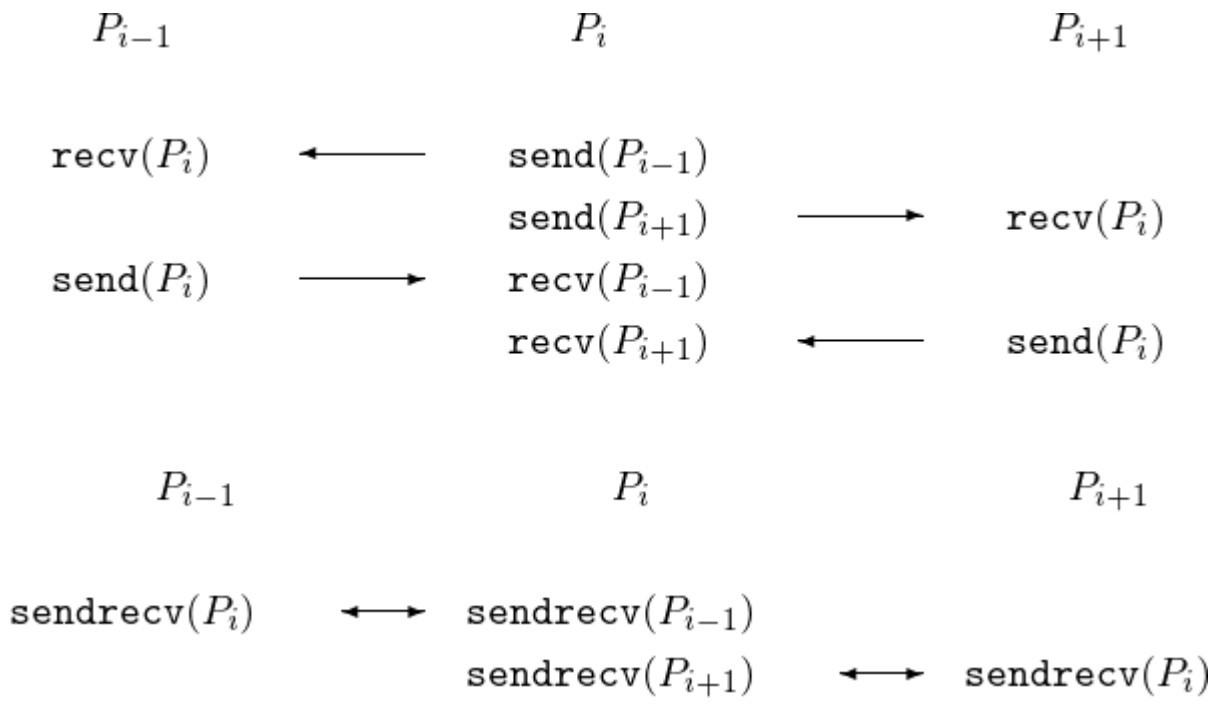


Fig. 6.3: The top picture is equivalent to the bottom picture.

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 6.1.

Table 6.1: 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;
```

6.1.2 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 $\sum_{i=0}^{n-1} a_i$ 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. 6.4.

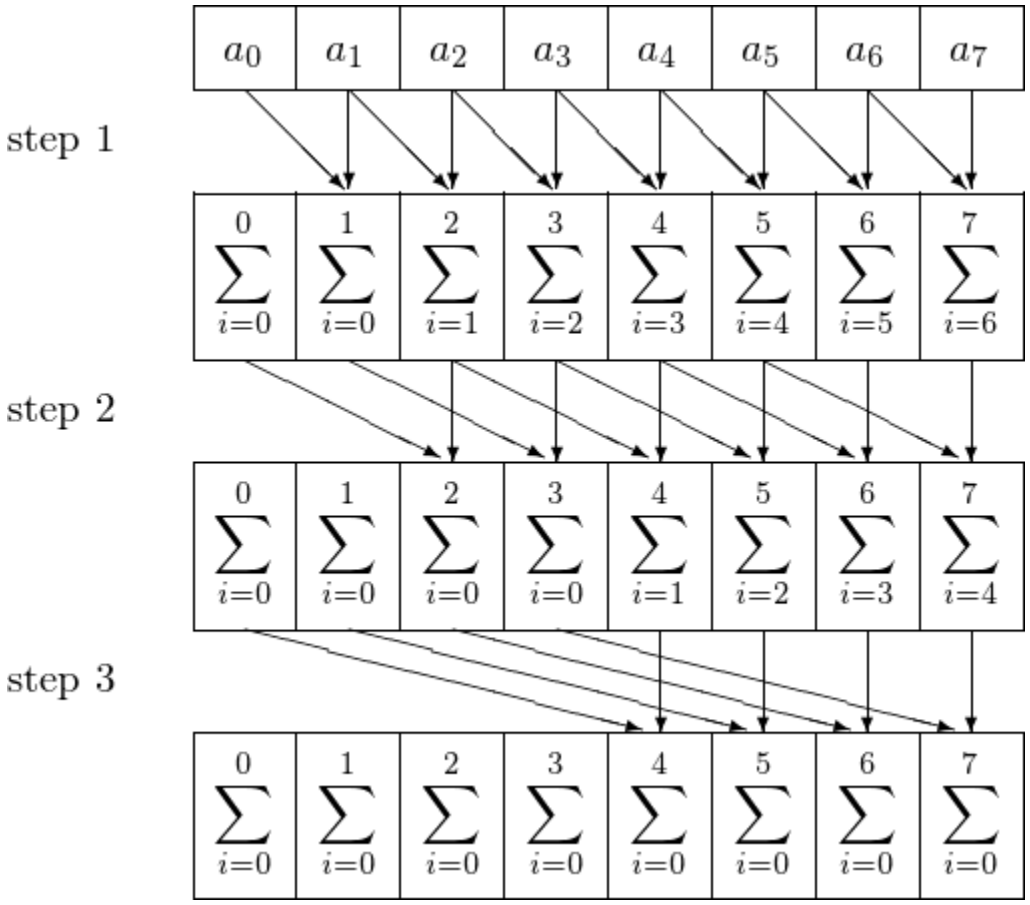


Fig. 6.4: The prefix sum for $n = 8 = p$.

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: $\frac{p}{\log_2(p)}$. 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"
#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.
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.
```

6.1.3 Barriers in Shared Memory Parallel Programming

Recall Pthreads and the work crew model. Often all threads must wait till on each other. We will illustrate the `pthread_barrier_t` with a small illustrative example.

```
int count = 3;
pthread_barrier_t our_barrier;
p_thread_barrier_init(&our_barrier, NULL, count);
```

In the example above, we initialized the barrier that will cause as many threads as the value of `count` to wait. A thread remains trapped waiting as long as fewer than `count` many threads have reached `pthread_barrier_wait(&our_barrier);` and the `pthread_barrier_destroy(&our_barrier)` should only be executed after all threads have finished.

In our illustrative program, each thread generates a random number, rolling a 6-sided die and then sleeps as many seconds as the value of the die (which is an integer number ranging from 1 till 6). The sleeping times are recorded in a shared array that is declared as a global variable. So the shared data is the time each thread sleeps. Each threads prints only after each thread has written its sleeping time in the shared data array. A screen shot of the program running with 5 threads is below.

```
$ /tmp/pthread_barrier_example
Give the number of threads : 5
Created 5 threads ...
Thread 0 has slept 2 seconds ...
Thread 2 has slept 2 seconds ...
Thread 1 has slept 4 seconds ...
Thread 3 has slept 5 seconds ...
Thread 4 has slept 6 seconds ...
Thread 4 has data : 24256
Thread 3 has data : 24256
Thread 2 has data : 24256
Thread 1 has data : 24256
Thread 0 has data : 24256
$
```

The code should be compiled with the `-lpthread` option, if for example, the file `pthread_barrier_example.c` contains the C code, then the compilation command could be

```
$ gcc -lpthread pthread_barrier_example.c -o /tmp/pthread_barrier_example
```

The global variables `size`, `data`, and `our_barrier` will be initialized in the main program. The user is prompted to enter `size`, the number of threads. The array `data` is allocated with `size` elements; The barrier `our_barrier` is initialized. Code for the complete program is below:

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

int size;   /* size equals the number of threads */
int *data;  /* shared data, as many ints as size */
pthread_barrier_t our_barrier; /* to synchronize */

void *fun ( void *args )
{
   int *id = (int*) args;
   int r = 1 + (rand() % 6);
   int k;
   char strd[size+1];

   sleep(r);
   printf("Thread %d has slept %d seconds ...\n", *id, r);
   data[*id] = r;

   pthread_barrier_wait(&our_barrier);

   for(k=0; k<size; k++) strd[k] = '0' + ((char) data[k]);
   strd[size] = '\0';

   printf("Thread %d has data : %s\n", *id, strd);
}

int main ( int argc, char* argv[] )
{
   printf("Give the number of threads : "); scanf("%d", &size);
   data = (int*) calloc(size, sizeof(int));
   {
      pthread_t t[size];
      pthread_attr_t a;
      int id[size], i;

      pthread_barrier_init(&our_barrier, NULL, size);

      for(i=0; i<size; i++)
      {
         id[i] = i;
         pthread_attr_init(&a);
         if(pthread_create(&t[i], &a, fun, (void*)&id[i]) != 0)
            printf("Unable to create thread %d!\n", i);
      }
      printf("Created %d threads ...\n", size);
      for(i=0; i<size; i++) pthread_join(t[i], NULL);

      pthread_barrier_destroy(&our_barrier);
   }
   return 0;
}
```

6.1.4 Bibliography

1. W. Daniel Hillis and Guy L. Steele. **Data Parallel Algorithms**. *Communications of the ACM*, vol. 29, no. 12, pages 1170-1183, 1986.
2. B. Wilkinson and M. Allen. *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 2nd edition, 2005.

6.1.5 Exercises

1. Write code using `MPI_sendrecv` for a butterfly barrier. Show that your code works for $p = 8$.
2. Rewrite `prefix_sum.c` using `MPI_sendrecv`.
3. 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?

6.2 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.

6.2.1 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

$$\begin{aligned} A\mathbf{x} = \mathbf{b} &\Leftrightarrow (L + D + U)\mathbf{x} = \mathbf{b} \\ &\Leftrightarrow D\mathbf{x} = \mathbf{b} - L\mathbf{x} - U\mathbf{x} \\ &\Leftrightarrow D\mathbf{x} = D\mathbf{x} + \mathbf{b} - L\mathbf{x} - U\mathbf{x} - D\mathbf{x} \\ &\Leftrightarrow D\mathbf{x} = D\mathbf{x} + \mathbf{b} - A\mathbf{x} \\ &\Leftrightarrow \mathbf{x} = \mathbf{x} + D^{-1}(\mathbf{b} - A\mathbf{x}). \end{aligned}$$

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.: $p = 4$):

$$D \star \begin{bmatrix} \Delta\mathbf{x}^{[0]} \\ \hline \Delta\mathbf{x}^{[1]} \\ \hline \Delta\mathbf{x}^{[2]} \\ \hline \Delta\mathbf{x}^{[3]} \end{bmatrix} = \begin{bmatrix} \mathbf{b}^{[0]} \\ \hline \mathbf{b}^{[1]} \\ \hline \mathbf{b}^{[2]} \\ \hline \mathbf{b}^{[3]} \end{bmatrix} - \begin{bmatrix} A^{[0,0]} & A^{[0,1]} & A^{[0,2]} & A^{[0,3]} \\ \hline A^{[1,0]} & A^{[1,1]} & A^{[1,2]} & A^{[1,3]} \\ \hline A^{[2,0]} & A^{[2,1]} & A^{[2,2]} & A^{[2,3]} \\ \hline A^{[3,0]} & A^{[3,1]} & A^{[3,2]} & A^{[3,3]} \end{bmatrix} \star \begin{bmatrix} \mathbf{x}^{[0],(k)} \\ \hline \mathbf{x}^{[1],(k)} \\ \hline \mathbf{x}^{[2],(k)} \\ \hline \mathbf{x}^{[3],(k)} \end{bmatrix}$$

- 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. 6.5.

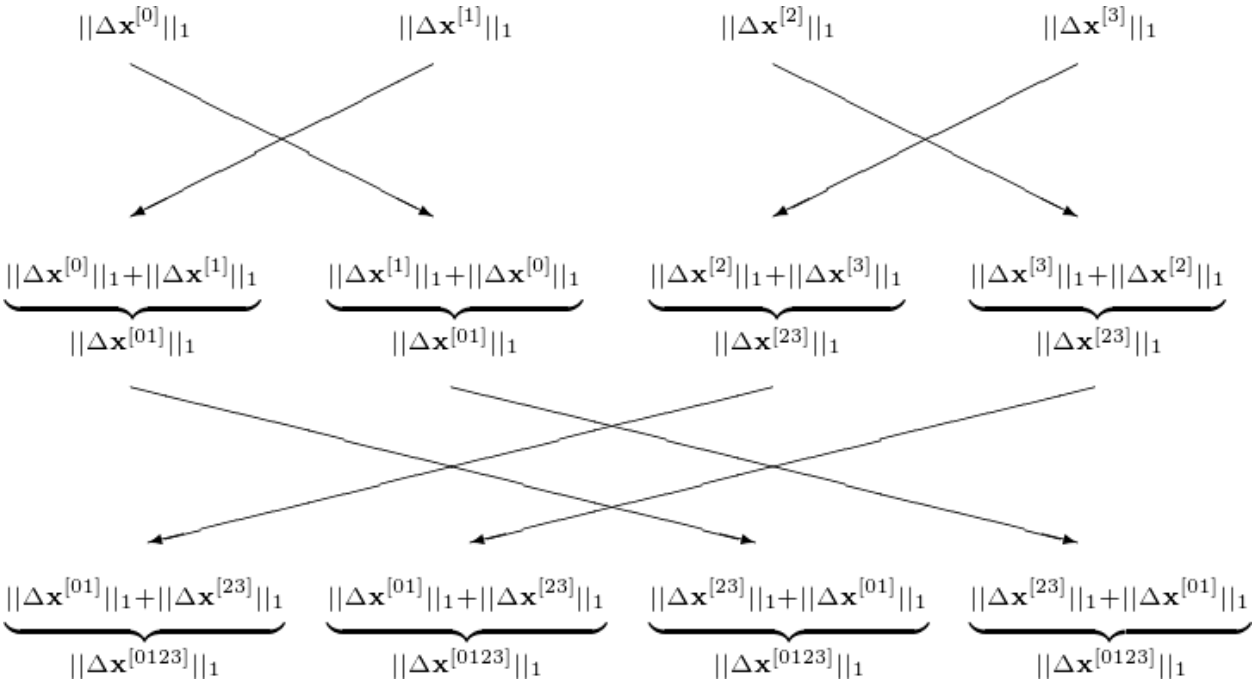


Fig. 6.5: Butterfly synchronization of a parallel Jacobi iteration with 4 processors.

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.

6.2.2 A Parallel Implementation with MPI

For dimension n, we consider the diagonally dominant system:

$$\begin{bmatrix} n+1 & 1 & \cdots & 1 \\ 1 & n+1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & n+1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 2n \\ 2n \\ \vdots \\ 2n \end{bmatrix}.$$

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 )
{
   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);
}
```

6.2.3 Gather-to-All with MPI_Allgather

Gathering the four elements of a vector to four processors is schematically depicted in Fig. 6.6.

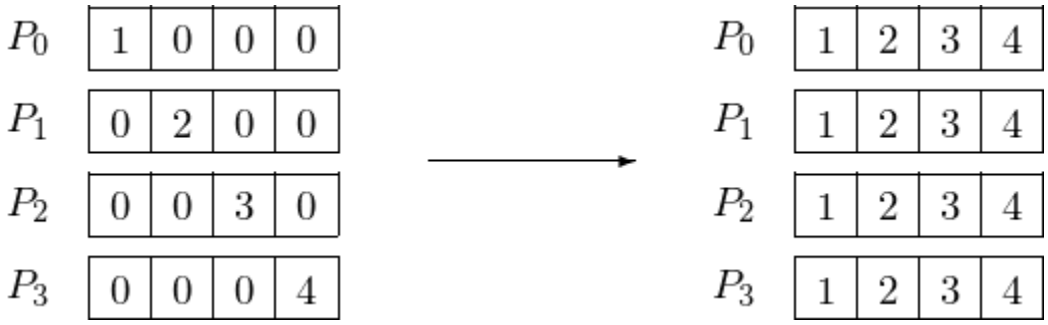


Fig. 6.6: Gathering 4 elements to 4 processors.

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 6.2.

Table 6.2: 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. 6.6 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

$$t_{\text{comp}} = \frac{n(2n+3)}{p}.$$

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

$$t_{\text{comm}} = p\left(t_{\text{startup}} + \frac{n}{p}t_{\text{data}}\right).$$

In the examples, the time unit is the cost of one arithmetical operation. Then the costs t_{startup} and t_{data} are multiples of this unit.

Finding the p with the minimum total cost is illustrated in Fig. 6.7 and Fig. 6.8.

In Fig. 6.7, the communication, computation, and total cost is shown for p ranging from 2 to 32, for one iteration, with $n = 1,000$, $t_{\text{startup}} = 10,000$, and $t_{\text{data}} = 50$. We see that the total cost starts to increase once p becomes larger than 16. For a larger dimension, after a ten-fold increase, $n = 10,000$, $t_{\text{startup}} = 10,000$, and $t_{\text{data}} = 50$, the scalability improves, as in Fig. 6.8, p ranges from 16 to 256.

6.2.4 Exercises

1. Use OpenMP to write a parallel version of the Jacobi method. Do you observe a better speedup than with MPI?
2. The power method to compute the largest eigenvalue of a matrix A uses the formulas $\mathbf{y} := A\mathbf{x}^{(k)}$; $\mathbf{x}^{(k+1)} := \mathbf{y}/||\mathbf{y}||$. Describe a parallel implementation of the power method.
3. 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?

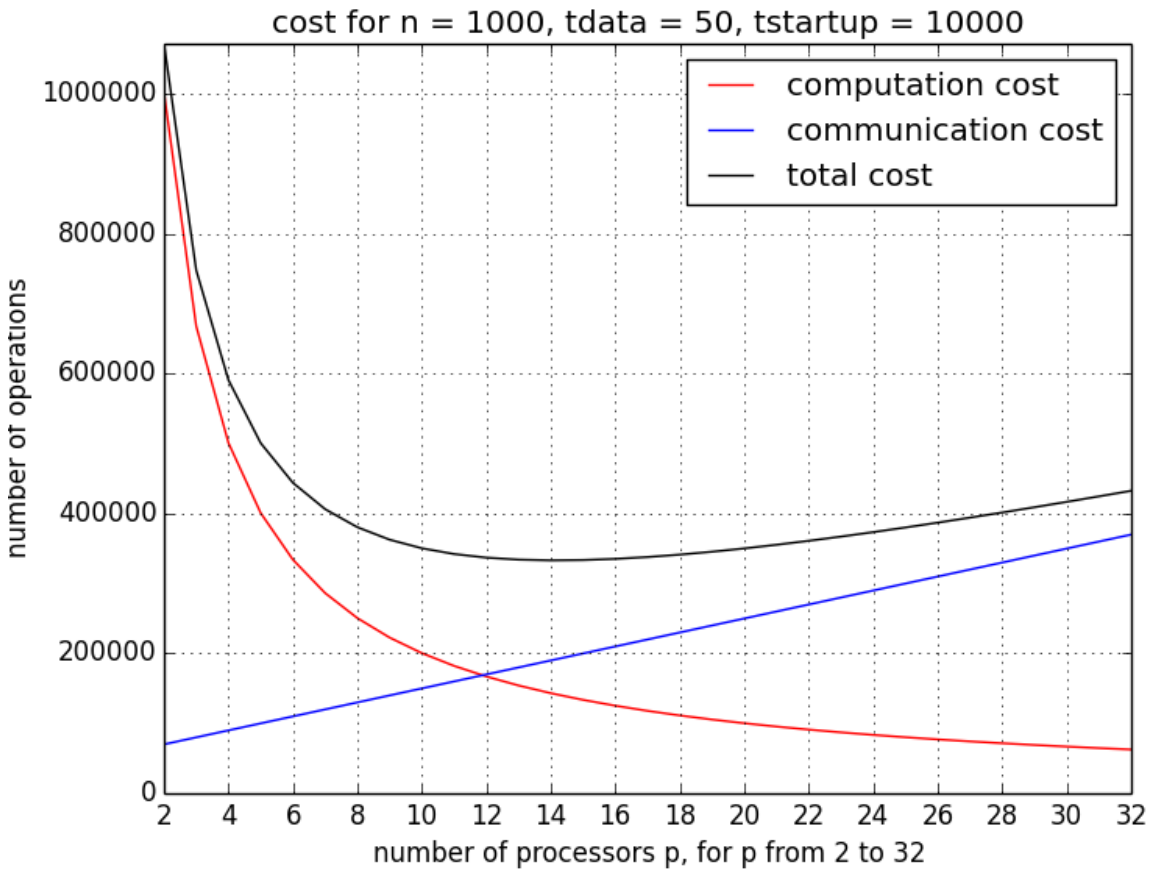


Fig. 6.7: With increasing p, the (red) computation cost decreases, while the (blue) communication cost increases. The minimum of the (black) total cost is the optimal value for p.

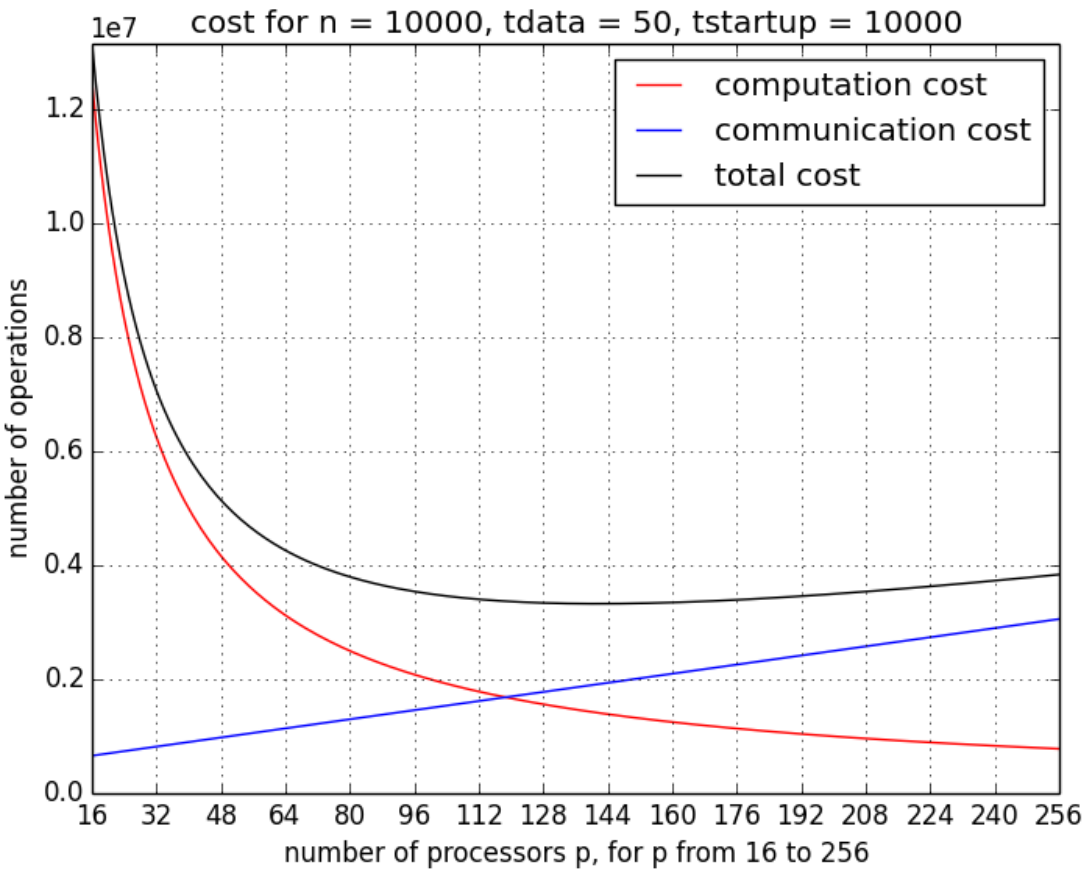


Fig. 6.8: With increasing p, the (red) computation cost decreases, while the (blue) communication cost increases. The minimum of the (black) total cost is the optimal value for p.

6.3 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.

6.3.1 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

$$x_i^{(k+1)} := x_i^{(k)} + \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(k+1)} - \sum_{j=i}^{n} a_{i,j} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n.$$

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
   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
```

6.3.2 Parallel Gauss-Seidel with OpenMP

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;

   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;
         }
      }
   }
}
```

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` are in Table 6.3.

Table 6.3: 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

6.3.3 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

$$\left.\frac{\partial u}{\partial x}\right|_{(x_0,y_0,t_0)} = \lim_{h \to 0} \underbrace{\frac{u(x_0 + h, y_0, t_0) - u(x_0, y_0, h)}{h}}_{u_x(x_0,y_0,t_0)}$$

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

$$\begin{aligned} u(x_0,y_0,t_0+h) &= u(x_0,y_0,t_0) \\ &+ h\ [\ u(x_0+h,y_0,t_0)+u(x_0-h,y_0,t_0) \\ &+ \quad u(x_0,y_0+h,t_0)+u(x_0,y_0-h,t_0)-4u(x_0,y_0,t_0)\ ] \end{aligned}$$

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

$$u_{i,j}^{(k+1)} = u_{i,j}^{(k)} + h\ \left[\ u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)} - 4u_{i,j}^{(k)}\ \right].$$

Fig. 6.9 shows the labeling of the grid points.

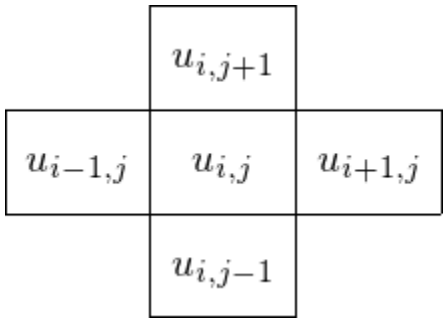


Fig. 6.9: In every step, we update $u_{i,j}$ based on $u_{i-1,j}$, $u_{i+1,j}$, $u_{i,j-1}$, and $u_{i,j+1}$.

We divide the grid in red and black points, as in Fig. 6.10.

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.

domain decomposition

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. 6.11.

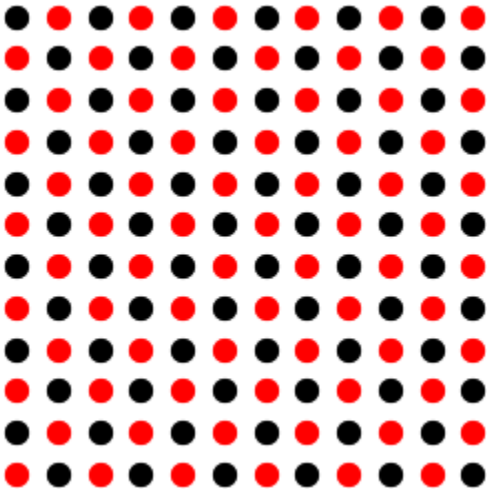


Fig. 6.10: Organization of the grid $u_{i,j}$ in red and black points.

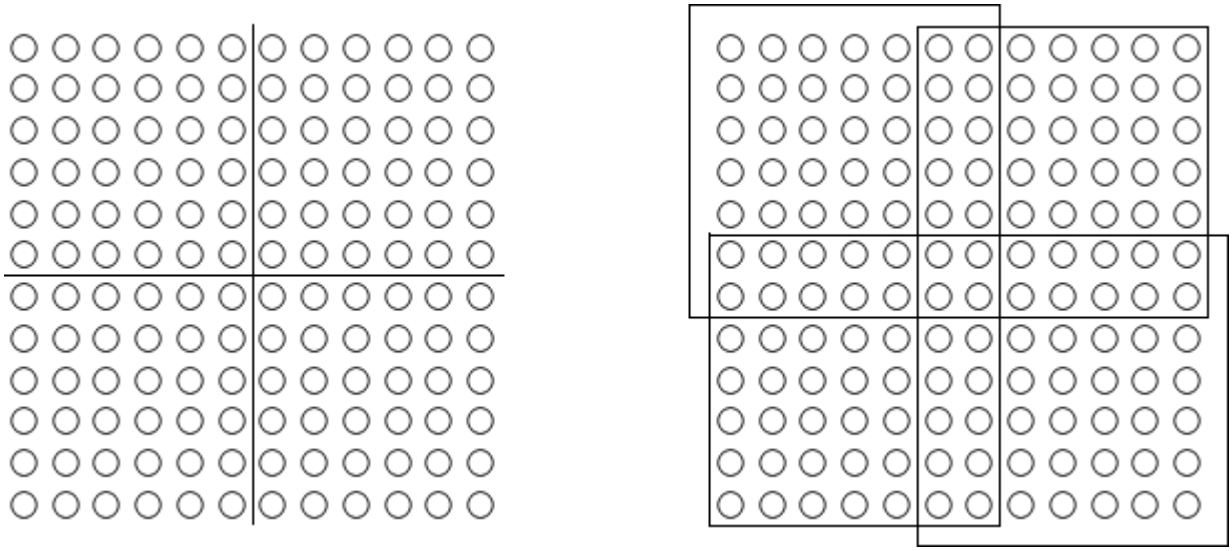


Fig. 6.11: Partitioning of the grid in squares.

Then the boundary elements are proportional to $n/\sqrt{p}$.

In Fig. 6.11, 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. 6.11, 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

$$t_{\text{comm}}^{\text{strip}} = 4\left(t_{\text{startup}} + nt_{\text{data}}\right)$$

to the communication cost for a square partitioning (for $p \geq 9$):

$$t_{\text{comm}}^{\text{square}} = 8\left(t_{\text{startup}} + \frac{n}{\sqrt{p}}t_{\text{data}}\right).$$

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.

6.3.4 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 <http://www.mcs.anl.gov/petsc/>.

PETSc is installed on `kepler` in the directory `/home/jan/Downloads/petsc-3.7.4`. The source code contains the directory `ts` on time stepping methods. We will run `ex3` from `/home/jan/Downloads/petsc-3.7.4/src/ts/examples/tutorials` After making the code, a sequential run goes like

```
$ /tmp/ex3 -draw_pause 1
```

The entry in a `makefile` to compile `ex3`:

```
PETSC_DIR=/usr/local/petsc-3.7.4
PETSC_ARCH=arch-linux2-c-debug

ex3:
        $(PETSC_DIR)/$(PETSC_ARCH)/bin/mpicc ex3.c \
        -I$(PETSC_DIR)/include \
        $(PETSC_DIR)/$(PETSC_ARCH)/lib/libpetsc.so \
        $(PETSC_DIR)/$(PETSC_ARCH)/lib/libopa.so \
        $(PETSC_DIR)/$(PETSC_ARCH)/lib/libfblas.a \
        $(PETSC_DIR)/$(PETSC_ARCH)/lib/libflapack.a \
        /usr/lib64/libblas.so.3 \
        -L/usr/X11R6/lib -lX11 -lgfortran -o /tmp/ex3
```

On a Mac OS X laptop, the `PETSC_ARCH` will be `PETSC_ARCH=arch-darwin-c-debug`.

6.3.5 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.

6.3.6 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.

6.4 Parallel Gaussian Elimination

6.4.1 LU and Cholesky Factorization

Consider as given a square n-by-n matrix A and corresponding right hand side vector $\mathbf{b}$ of length n. To solve an n-dimensional linear system $A\mathbf{x} = \mathbf{b}$ we factor A as a product of two triangular matrices, $A = LU$:

1. L is lower triangular, $L = [\ell_{i,j}]$, $\ell_{i,j} = 0$ if $j > i$ and $\ell_{i,i} = 1$.
2. U is upper triangular $U = [u_{i,j}]$, $u_{i,j} = 0$ if $i > j$.

Solving $A\mathbf{x} = \mathbf{b}$ is then equivalent to solving $L(U\mathbf{x}) = \mathbf{b}$ in two stages:

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 $O(n^2)$. For numerical stability, we apply partial pivoting and compute $PA = LU$, where P is a permutation matrix.

The steps in the LU factorization of the matrix A are shown in Fig. 6.12.

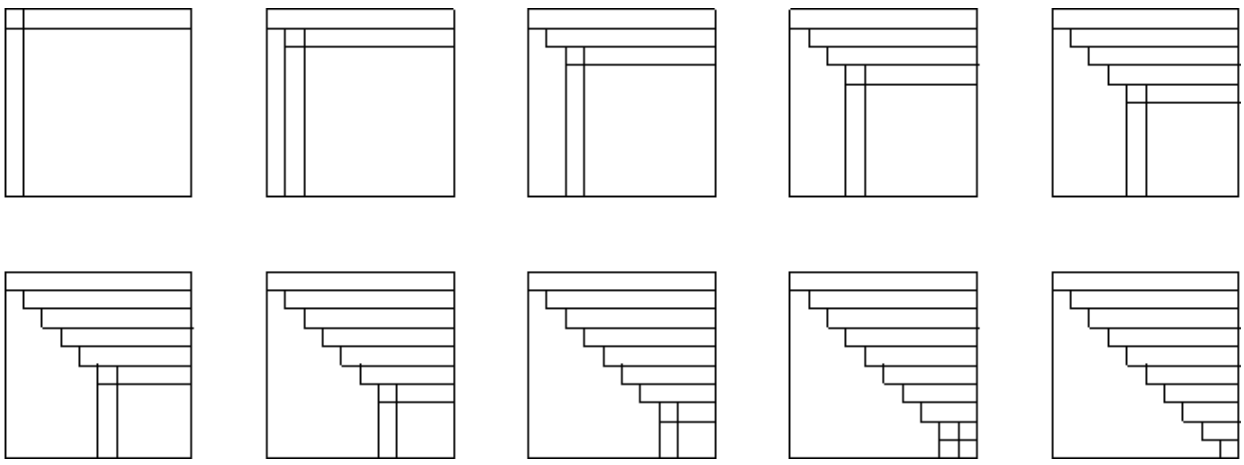


Fig. 6.12: Steps in the LU factorization.

For column $j = 1, 2, \ldots, n-1$ in A do

1. find the largest element $a_{i,j}$ in column j (for $i \geq j$);
2. if $i \neq j$, then swap rows i and j;
3. for $i = j+1, \ldots n$, for $k = j+1, \ldots, n$ do $a_{i,k} := a_{i,k} - \left(\frac{a_{i,j}}{a_{j,j}}\right) a_{j,k}$.

If A is symmetric, $A^T = A$, and positive semidefinite: $\forall \mathbf{x} : \mathbf{x}^T A \mathbf{x} \geq 0$, then we better compute a Cholesky factorization: $A = LL^T$, where L is a lower triangular matrix. Because A is positive semidefinite, no pivoting is needed, and we need about half as many operations as LU.

Pseudo code is listed below:

```
for j from 1 to n do
    for k from 1 to j-1 do
        a[j][j] := a[j][j] - a[j][k]**2
    a[j][j] := sqrt(a[j][j])
    for i from j+1 to n do
        for k from 1 to j do
            a[i][j] := a[i][j] - a[i][k]*a[j][k]
        a[i][j] := a[i][j]/a[j][j]
```

Let *A* be a symmetric, positive definite *n*-by-*n* matrix. In preparation for parallel implementation, we consider tiled matrices. For tile size *b*, let $n = p \times b$ and consider

$$A = \begin{bmatrix} A_{1,1} & A_{2,1} & \cdots & A_{p,1} \\ A_{2,1} & A_{2,2} & \cdots & A_{p,2} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p,1} & A_{p,2} & \cdots & A_{p,p} \end{bmatrix},$$

where $A_{i,j}$ is an *b*-by-*b* matrix.

A crude classification of memory hierarchies distinguishes between registers (small), cache (medium), and main memory (large). To reduce data movements, we want to keep data in registers and cache as much as possible.

Pseudo code for a tiled Cholesky factorization is listed below.

```
for k from 1 to p do
    DPOTF2(A[k][k], L[k][k])                  # L[k][k] := Cholesky(A[k][k])
    for i from k+1 to p do
        DTRSM(L[k][k], A[i][k], L[i][k])      # L[i][k] := A[i][k]*L[k][k]**(-T)
    end for
    for i from k+1 to p do
        for j from k+1 to p do
            DGSMM(L[i][k], L[j][k], A[i][j]) # A[i][j] := A[i][j] - L[i][k]*L[j][k]}
↪**T
        end for
end for
```

6.4.2 Blocked LU Factorization

In deriving blocked formulations of LU, consider a 3-by-3 blocked matrix. The optimal size of the blocks is machine dependent.

$$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} = \begin{bmatrix} L_{1,1} & & \\ L_{2,1} & L_{2,2} & \\ L_{3,1} & L_{3,2} & L_{3,3} \end{bmatrix} \begin{bmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ & U_{2,2} & U_{2,3} \\ & & U_{3,3} \end{bmatrix}$$

Expanding the right hand side and equating to the matrix at the left gives formulations for the LU factorization.

$$\begin{array}{lll} A_{1,1} = L_{1,1}U_{1,1} & A_{1,2} = L_{1,1}U_{1,2} & A_{1,3} = L_{1,1}U_{1,3} \\ A_{2,1} = L_{2,1}U_{1,1} & A_{2,2} = L_{2,1}U_{1,2} + L_{2,2}U_{2,2} & A_{2,3} = L_{2,1}U_{1,3} + L_{2,2}U_{2,3} \\ A_{3,1} = L_{3,1}U_{1,1} & A_{3,2} = L_{3,1}U_{1,2} + L_{3,2}U_{2,2} & A_{3,3} = L_{3,1}U_{1,3} + L_{3,2}U_{2,3} + L_{3,3}U_{3,3} \end{array}$$

The formulas we are deriving gives rise to the right looking LU. We store the $L_{i,j}$'s and $U_{i,j}$'s in the original matrix:

$$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} = \begin{bmatrix} L_{1,1} & & \\ L_{2,1} & I & \\ L_{3,1} & & I \end{bmatrix} \begin{bmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ & B_{2,2} & B_{2,3} \\ & B_{3,2} & B_{3,3} \end{bmatrix}$$

The matrices $B_{i,j}$'s are obtained after a first block LU step. To find $L_{2,2}$, $L_{3,2}$, and $U_{2,2}$ we use

$$\begin{cases} A_{2,2} = L_{2,1}U_{1,2} + L_{2,2}U_{2,2} \\ A_{3,2} = L_{3,1}U_{1,2} + L_{3,2}U_{2,2} \end{cases} \quad \text{and} \quad \begin{cases} A_{2,2} = L_{2,1}U_{1,2} + B_{2,2} \\ A_{3,2} = L_{3,1}U_{1,2} + B_{3,2} \end{cases}$$

Eliminating $A_{2,2} - L_{2,1}U_{1,2}$ and $A_{3,2} - L_{3,1}U_{1,2}$ gives

$$\begin{cases} B_{2,2} = L_{2,2}U_{2,2} \\ B_{3,2} = L_{3,2}U_{2,2} \end{cases}$$

Via LU on $B_{2,2}$ we obtain $L_{2,2}$ and $U_{2,2}$. Then: $L_{3,2} := B_{3,2}U_{2,2}^{-1}$.

The formulas we derived are similar to the scalar case and are called *right looking*.

But we may organize the LU factorization differently, as shown in Fig. 6.13.

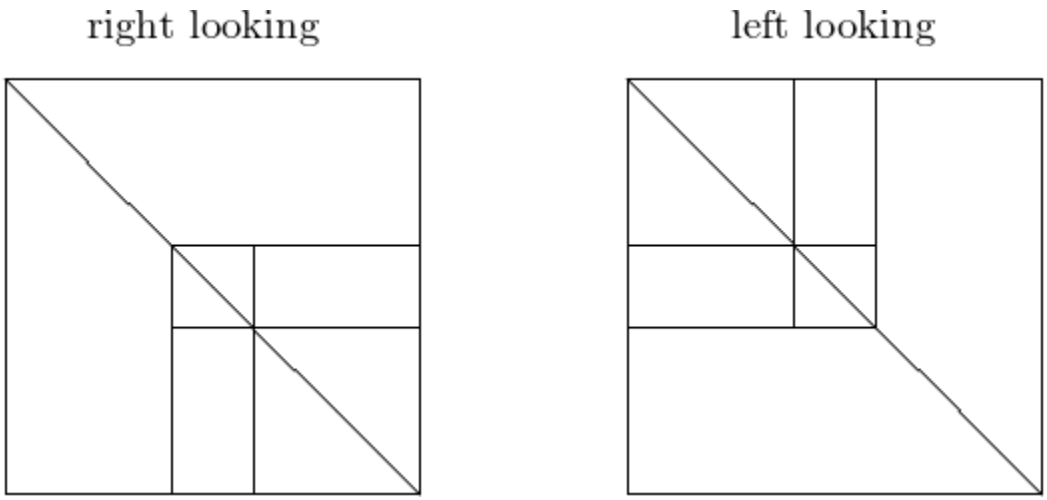


Fig. 6.13: Right and left looking organization of tiled LU factorization.

What is good looking? Left is best for data access. We derive left looking formulas. Going from P_1A to P_2P_1A:

$$\begin{bmatrix} L_{1,1} & & \\ L_{2,1} & I & \\ L_{3,1} & & I \end{bmatrix} \begin{bmatrix} U_{1,1} & A_{1,2} & A_{1,3} \\ & A_{2,2} & A_{2,3} \\ & A_{3,2} & A_{3,3} \end{bmatrix} \rightarrow \begin{bmatrix} L_{1,1} & & \\ L_{2,1} & L_{2,2} & \\ L_{3,1} & L_{3,2} & I \end{bmatrix} \begin{bmatrix} U_{1,1} & U_{1,2} & A_{1,3} \\ & U_{2,2} & A_{2,3} \\ & & A_{3,3} \end{bmatrix}$$

We keep the original $A_{i,j}$'s and postpone updating to the right.

1. We get $U_{1,2}$ via $A_{1,2} = L_{1,1}U_{1,2}$ and compute $U_{1,2} = L_{1,1}^{-1}A_{1,2}$.
2. To compute $L_{2,2}$ and $L_{3,2}$ do $\begin{bmatrix} B_{2,2} \\ B_{3,2} \end{bmatrix} = \begin{bmatrix} A_{2,2} \\ A_{3,2} \end{bmatrix} - \begin{bmatrix} L_{2,1} \\ L_{3,1} \end{bmatrix} U_{1,2}$;

and factor $P_2 \begin{bmatrix} B_{2,2} \\ B_{3,2} \end{bmatrix} = \begin{bmatrix} L_{2,2} \\ L_{3,2} \end{bmatrix} U_{2,2}$ as before.

Replace $\begin{bmatrix} A_{2,3} \\ A_{3,3} \end{bmatrix} := P_2 \begin{bmatrix} A_{2,3} \\ A_{3,3} \end{bmatrix}$ and $\begin{bmatrix} L_{2,1} \\ L_{3,1} \end{bmatrix} := P_2 \begin{bmatrix} L_{2,1} \\ L_{3,1} \end{bmatrix}$.

Pseudo code for the tiled algorithm for LU factorization is below.

```
for k from 1 to p do
    DGETF(A[k][k], L[k][k], U[k][k], P[k][k])
    for j from k+1 to p do
        DGESSM(A[k][j], L[k][k], P[k][k], U[k][j])
    for i from k+1 to p do
        DTSTRF(U[k][k], A[i][k], P[i][k])
        for j from k+1 to p do
            DSSSM(U[k][j], A[i][j], L[i][k], P[i][k])
```

where the function calls correspond to the following formulas:

- `DGETF(A[k][k],L[k][k],U[k][k],P[k][k])` corresponds to $L_{k,k}, U_{k,k}, P_{k,k} := \text{LU}(A_{k,k})$.
- `DGESSM(A[k][j],L[k][k],P[k][k],U[k][j])` corresponds to $U_{k,j} := L_{k,k}^{-1} P_{k,k} A_{k,j}$.
- `DTSTRF(U[k][k],A[i][k],P[i][k])` corresponds to $U_{k,k}, L_{i,k}, P_{i,k} := LU\left(\begin{bmatrix} U_{k,k} \\ A_{i,k} \end{bmatrix}\right)$
- `DSSSM(U[k][j],A[i][j],L[i][k],P[i][k])` corresponds to $\begin{bmatrix} U_{k,j} \\ A_{i,j} \end{bmatrix} := L_{i,k}^{-1} P_{i,k} \begin{bmatrix} U_{k,j} \\ A_{i,j} \end{bmatrix}$.

6.4.3 The PLASMA Software Library

PLASMA stands for Parallel Linear Algebra Software for Multicore Architectures. The software uses FORTRAN and C. It is designed for efficiency on homogeneous multicore processors and multi-socket systems of multicore processors, Built using a small set of sequential routines as building blocks, referred to as *core BLAS*, and free to download from <http://icl.cs.utk.edu/plasma>.

Capabilities and limitations: Can solve dense linear systems and least squares problems. Unlike LAPACK, PLASMA currently does not solve eigenvalue or singular value problems and provide no support for band matrices.

Basic Linear Algebra Subprograms: BLAS

1. Level-1 BLAS: vector-vector operations, $O(n)$ cost. inner products, norms, $\mathbf{x} \pm \mathbf{y}$, $\alpha\mathbf{x} + \mathbf{y}$.
2. Level-2 BLAS: matrix-vector operations, $O(mn)$ cost.
   - $\mathbf{y} = \alpha A\mathbf{x} + \beta\mathbf{y}$
   - $A = A + \alpha\mathbf{x}\mathbf{y}^T$, rank one update
   - :math:{bf x} = T^{-1} {bf b}‘, for T a triangular matrix
3. Level-3 BLAS: matrix-matrix operations, $O(kmn)$ cost.
   - $C = \alpha AB + \beta C$
   - $C = \alpha AA^T + \beta C$, rank *k* update of symmetric matrix
   - $B = \alpha TB$, for T a triangular matrix
   - $B = \alpha T^{-1}B$, solve linear system with many right hand sides

The execution is asynchronous and graph driven. We view a blocked algorithm as a Directed Acyclic Graph (DAG): nodes are computational tasks performed in kernel subroutines; edges represent the dependencies among the tasks. Given a DAG, tasks are scheduled asynchronously and independently, considering the dependencies imposed by the edges in the DAG. A critical path in the DAG connects those nodes that have the highest number of outgoing edges. The scheduling policy assigns higher priority to those tasks that lie on the critical path.

The directory `examples` in `/usr/local/plasma-installer_2.6.0/build/plasma_2.6.0` contains `example_cposv`, an example for a Cholesky factorization of a symmetric positive definite matrix. Running `make` as defined in the `examples` directory:

```
[root@kepler examples]# make example_cposv
gcc -O2 -DADD_ -I../include -I../quark \
-I/usr/local/plasma-installer_2.6.0/install/include -c \
example_cposv.c -o example_cposv.o
gfortran  example_cposv.o -o example_cposv -L../lib -lplasma \
-lcoreblasqw -lcoreblas -lplasma -L../quark -lquark \
-L/usr/local/plasma-installer_2.6.0/install/lib -lcblas \
-L/usr/local/plasma-installer_2.6.0/install/lib -llapacke \
-L/usr/local/plasma-installer_2.6.0/install/lib -ltmg -llapack \
-L/usr/local/plasma-installer_2.6.0/install/lib -lrefblas \
-lpthread -lm
[root@kepler examples]#
```

Cholesky factorization with the dimension equal to 10,000.

```
[root@kepler examples]# time ./example_dposv
-- PLASMA is initialized to run on 2 cores.
============
Checking the Residual of the solution
-- ||Ax-B||_oo/((||A||_oo||x||_oo+||B||_oo).N.eps) = 2.710205e-03
-- The solution is CORRECT !
-- Run of DPOSV example successful !

real    1m22.271s
user    2m42.534s
sys     0m1.064s
[root@kepler examples]#
```

The wall clock times for Cholesky on dimension 10,000 are listed in Table 6.4.

Table 6.4: Wall clock times for Cholesky factorization.

#cores	real time	speedup
1	2m 40.913s	
2	1m 22.271s	1.96
4	42.621s	3.78
8	23.480s	6.85
16	12.647s	12.72

6.4.4 Bibliography

1. E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan. *PLASMA Users' Guide. Parallel Linear Algebra Software for Multicore Architectures. Version 2.0.*

2. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing* 35: 38-53, 2009.

6.4.5 Exercises

1. Write your own parallel shared memory version of the Cholesky factorization, using OpenMP, Pthreads, or the Intel TBB.
2. Derive right looking LU factorization formulas with pivoting, i.e.: introducing permutation matrices *P*. Develop first the formulas for a 3-by-3 block matrix and then generalize the formulas into an algorithm for any *p*-by-*p* block matrix.
3. Take an appropriate example from the PLASMA installation to test the speedup of the multicore LU factorization routines.

6.5 Parallel Numerical Linear Algebra

6.5.1 QR Factorization

To solve an overconstrained linear system $A\mathbf{x} = \mathbf{b}$ of *m* equations in *n* variables, where $m > n$, we factor A as a product of two matrices, $A = QR$:

1. Q is an orthogonal matrix: $Q^HQ = I$, the inverse of Q is its Hermitian transpose Q^H;
2. R is upper triangular: $R = [r_{i,j}]$, $r_{i,j} = 0$ if $i > j$.

Solving $A\mathbf{x} = \mathbf{b}$ is equivalent to solving $Q(R\mathbf{x}) = \mathbf{b}$. We multiply $\mathbf{b}$ with Q^T and then apply backward substitution: $R\mathbf{x} = Q^T\mathbf{b}$. The solution $\mathbf{x}$ minimizes the square of the residual $||b - A\mathbf{x}||_2^2$ (least squares).

Householder transformation

For $\mathbf{v} \neq 0$: $H = I - \dfrac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}}$ is *a Householder transformation*.

By its definition, $H = H^T = H^{-1}$. Because of their geometrical interpretation, Householder transformations are also called Householder reflectors.

For some *n*-dimensional real vector $\mathbf{x}$, let $\mathbf{e}_1 = (1, 0, \ldots, 0)^T$:

$$\mathbf{v} = \mathbf{x} - ||\mathbf{x}||_2\mathbf{e}_1 \quad \Rightarrow \quad H\mathbf{x} = ||\mathbf{x}||_2\mathbf{e}_1.$$

With H we eliminate, reducing A to an upper triangular matrix. Householder QR: Q is a product of Householder reflectors.

The LAPACK `DGEQRF` on an *m*-by-*n* matrix produces:

$$H_1H_2 \cdots H_n = I - VTV^T$$

where each H_i is a Householder reflector matrix with associated vectors $\mathbf{v}_i$ in the columns of V and T is an upper triangular matrix.

For an m-by-n matrix $A = [a_{i,j}]$ with columns bfa_j, the formulas are formatted as pseudo code for a Householder QR factorization in Fig. 6.14.

$$
\begin{aligned}
&\text{for } k = 1, 2, \ldots, n \text{ do} \\
&\quad \alpha_k := -\text{sign}(a_{k,k}) \sqrt{a_{k,k}^2 + \cdots + a_{m,k}^2} \\
&\quad \mathbf{v}_k = [0 \cdots 0 \; a_{k,k} \cdots a_{m,k}]^T - \alpha_k \mathbf{e}_k \\
&\quad \beta_k := \mathbf{v}_k^T \mathbf{v}_k \\
&\quad \text{if } \beta_k \neq 0 \text{ then} \\
&\quad\quad \text{for } j = k, k+1, \ldots, n \text{ do} \\
&\quad\quad\quad \gamma_j := \mathbf{v}_k^T \mathbf{a}_j \\
&\quad\quad\quad \mathbf{a}_j := \mathbf{a}_j - 2 \frac{\gamma_j}{\beta_k} \mathbf{v}_k
\end{aligned}
$$

Fig. 6.14: Pseudo code for Householder QR factorization.

For a parallel implementation, we consider tiled matrices, dividing the *m*-by-*n* matrix *A* into *b*-by-*b* blocks, $m = b \times p$ and $n = b \times q$. Then we consider *A* as an $(p \times b)$-by-$(q \times b)$ matrix:

$$
A = \begin{bmatrix}
A_{1,1} & A_{1,2} & \cdots & A_{1,q} \\
A_{2,1} & A_{2,2} & \cdots & A_{2,q} \\
\vdots & \vdots & \ddots & \vdots \\
A_{p,1} & A_{p,2} & \cdots & A_{p,q}
\end{bmatrix},
$$

where each $A_{i,j}$ is an *b*-by-*b* matrix. A crude classification of memory hierarchies distinguishes between registers (small), cache (medium), and main memory (large). To reduce data movements, we want to keep data in registers and cache as much as possible.

To introduce QR factorization on a tiled matrix, consider for example

$$
A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} = Q \begin{bmatrix} R_{1,1} & R_{1,2} \\ 0 & R_{2,2} \end{bmatrix}.
$$

We proceed in three steps.

1. Perform a QR factorization:

$$
\begin{array}{l} A_{1,1} = Q_1 B_{1,1} \\ (Q_1, B_{1,1}) := \text{QR}(A_{1,1}) \end{array} \quad \text{and} \quad \begin{array}{l} A_{1,2} = Q_1 B_{1,2} \\ B_{1,2} = Q_1^T A_{1,2} \end{array}
$$

2. Coupling square blocks:

$$
(Q_2, R_{1,1}) := \text{QR}\left(\begin{bmatrix} B_{1,1} \\ A_{2,1} \end{bmatrix} \right) \quad \text{and} \quad \begin{bmatrix} R_{1,2} \\ B_{2,2} \end{bmatrix} := Q_2^T \begin{bmatrix} R_{1,2} \\ A_{2,2} \end{bmatrix}.
$$

3. Perform another QR factorization: $(Q_3, R_{2,2}) := \text{QR}(B_{2,2})$.

Pseudo code for a tiled QR factorization is given below.

```
for k from 1 to min(p,q) do
    DGEQRT(A[k][k], V[k][k], R[k][k], T[k][k])
    for i from k+1 to p do
        DLARFB(A[k][j], V[k][k], T[k][k], R[k][j])
    for i from k+1 to p do
        DTSQRT(R[k][k], A[i][k], V[i][k], T[i][k])
        for j from k+1 to p do
            DSSRFB(R[k][j], A[i][j], V[i][k], T[i][k])
```

where the function calls correspond to the folllowing mathematical formulas:

- `DGEQRT(A[k][k],V[k][k],R[k][k],T[k][k])` corresponds to $V_{k,k}, R_{k,k}, T_{k,k} := \text{QR}(A_{k,k})$.
- `DLARFB(A[k]][j],V[k][k],T[k][k],R[k][j])` corresponds to $R_{k,j} := (I - V_{k,k}T_{k,k}V_{k,k}^T)A_{k,j}$.
- `DTSQRT(R{k][k],A[i][k],V[i][k],T[i][k])` corresponds to

$$(V_{i,k}, T_{i,k}, R_{k,k}) := \text{QR}\left(\begin{bmatrix} R_{k,k} \\ A_{i,k} \end{bmatrix}\right).$$

- `DSSRFB(R[k][j],A[i][j],V[i][k],T[i][k])` corresponds to

$$\begin{bmatrix} R_{k,j} \\ A_{i,j} \end{bmatrix} := (I - V_{i,k}T_{i,k}V_{i,k}^T) \begin{bmatrix} R_{k,j} \\ A_{i,j} \end{bmatrix}.$$

The directory `examples` in `/usr/local/plasma-installer_2.6.0/build/plasma_2.6.0` contains `example_dgeqrs`, an example for solving an overdetermined linear system with a QR factorization.

Compiling on a MacBook Pro in the `examples` directory:

```
$ sudo make example_dgeqrs
gcc -O2 -DADD_ -I../include -I../quark
-I/usr/local/plasma-installer_2.6.0/install/include
-c example_dgeqrs.c -o example_dgeqrs.o
gfortran  example_dgeqrs.o -o example_dgeqrs -L../lib
-lplasma  -lcoreblasqw -lcoreblas -lplasma -L../quark
-lquark  -L/usr/local/plasma-installer_2.6.0/install/lib
-llapacke -L/usr/local/plasma-installer_2.6.0/install/lib
-ltmg  -framework veclib  -lpthread -lm
$
```

On `kepler` we do the same.

To make the program in our own home directory, we define a makefile:

```
PLASMA_DIR=/usr/local/plasma-installer_2.6.0/build/plasma_2.6.0

example_dgeqrs:
        gcc -O2 -DADD_ -I$(PLASMA_DIR)/include -I$(PLASMA_DIR)/quark \
            -I/usr/local/plasma-installer_2.6.0/install/include \
            -c example_dgeqrs.c -o example_dgeqrs.o
        gfortran  example_dgeqrs.o -o example_dgeqrs \
            -L$(PLASMA_DIR)/lib -lplasma -lcoreblasqw \
            -lcoreblas -lplasma -L$(PLASMA_DIR)/quark -lquark \
            -L/usr/local/plasma-installer_2.6.0/install/lib -lcblas \
            -L/usr/local/plasma-installer_2.6.0/install/lib -llapacke \
            -L/usr/local/plasma-installer_2.6.0/install/lib -ltmg -llapack \
            -L/usr/local/plasma-installer_2.6.0/install/lib \
            -lrefblas -lpthread -lm
```

Typing `make example_dgeqrs` will then compile and link. Running `example_dgeqrs` with dimension 4000:

```
$ time ./example_dgeqrs
-- PLASMA is initialized to run on 1 cores.
============
Checking the Residual of the solution
-- ||Ax-B||_oo/((||A||_oo||x||_oo+||B||)_oo.N.eps) = 2.829153e-02
-- The solution is CORRECT !
-- Run of DGEQRS example successful !
```

```
real    0m42.172s
user    0m41.965s
sys     0m0.111s
$
```

Results of `time example_dgeqrs`, for dimension 4000, are listed in Table 6.5.

Table 6.5: Timings on dgeqrs for n = 4000, with p threads.

p	real	user	sys	speedup
1	42.172	41.965	0.111	
2	21.495	41.965	0.154	1.96
4	11.615	43.322	0.662	3.63
8	6.663	46.259	1.244	6.33
16	3.872	46.793	2.389	10.89

6.5.2 Conjugate Gradient and Krylov Subspace Methods

We link linear systems solving to an optimization problem. Let A be a positive definite matrix: $\forall \mathbf{x} : \mathbf{x}^T A\mathbf{x} \geq 0$ and $A^T = A$. The optimum of

$$q(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{x}^T\mathbf{b} \text{ is at } A\mathbf{x} - \mathbf{b} = \mathbf{0}.$$

For the exact solution $\mathbf{x}$: $A\mathbf{x} = \mathbf{b}$ and an approximation $\mathbf{x}_k$, let the error be $\mathbf{e}_k = \mathbf{x}_k - \mathbf{x}$.

$$\begin{aligned} ||\mathbf{e}_k||_A^2 = \mathbf{e}_k^T A\mathbf{e}_k &= (\mathbf{x}_k - \mathbf{x})^T A(\mathbf{x}_k - \mathbf{x}) \\ &= \mathbf{x}_k^T A\mathbf{x}_k - 2\mathbf{x}_k^T A\mathbf{x} + \mathbf{x}^T A\mathbf{x} \\ &= \mathbf{x}_k^T A\mathbf{x}_k - 2\mathbf{x}_k^T\mathbf{b} + c \\ &= 2q(\mathbf{x}_k) + c \end{aligned}$$

$\Rightarrow$ minimizing $q(\mathbf{x})$ is the same as minimizing the error.

The conjugate gradient method is similar to the steepest descent method, formulated in Fig. 6.15.

$$\begin{aligned} &\mathbf{x}_0 = \mathbf{0};\ \mathbf{r}_0 = \mathbf{b};\ \mathbf{p}_0 := \mathbf{r}_0; \\ &\text{for } k = 0, 1, 2 \ldots \text{ do} \\ &\quad \alpha_k := \frac{\mathbf{r}_{k-1}^T \mathbf{r}_{k-1}}{\mathbf{p}_{k-1}^T A\mathbf{p}_{k-1}}; && \textit{step length} \\ &\quad \mathbf{x}_k := \mathbf{x}_{k-1} + \alpha_k \mathbf{p}_{k-1}; && \textit{update solution} \\ &\quad \mathbf{r}_k := \mathbf{r}_{k-1} - \alpha_k A\mathbf{p}_{k-1}; && \textit{residual} \\ &\quad \beta_k := \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{r}_{k-1}^T \mathbf{r}_{k-1}}; && \textit{improvement of step} \\ &\quad \mathbf{p}_k := \mathbf{r}_k + \beta_k \mathbf{p}_{k-1}; && \textit{compute search direction} \end{aligned}$$

Fig. 6.15: Mathematical definition of the conjugate gradient method.

For large sparse matrices, the conjugate gradient method is much faster than Cholesky.

Krylov subspace methods

We consider spaces spanned by matrix-vector products.

- Input: right hand side vector $\mathbf{b}$ of dimension n, and some algorithm to evaluate $A\mathbf{x}$.
- Output: approximation for $A^{-1}\mathbf{b}$.

Example ($n = 4$), consider $\mathbf{y}_1 = \mathbf{b}$, $\mathbf{y}_2 = A\mathbf{y}_1$, $\mathbf{y}_3 = A\mathbf{y}_2$, $\mathbf{y}_4 = A\mathbf{y}_3$, stored in the columns of $K = [\mathbf{y}_1\ \mathbf{y}_2\ \mathbf{y}_3\ \mathbf{y}_4]$.

$$
\begin{aligned}
AK &= [A\mathbf{y}_1\ A\mathbf{y}_2\ A\mathbf{y}_3\ A\mathbf{y}_4] = [\mathbf{y}_2\ \mathbf{y}_3\ \mathbf{y}_4\ A^4\mathbf{y}_1] \\
&= K \begin{bmatrix} 0 & 0 & 0 & -c_1 \\ 1 & 0 & 0 & -c_2 \\ 0 & 1 & 0 & -c_3 \\ 0 & 0 & 1 & -c_4 \end{bmatrix}, \quad \mathbf{c} = -K^{-1}A^4\mathbf{y}_1.
\end{aligned}
$$

$\Rightarrow AK = KC$, where C is a companion matrix.

Set $K = [\mathbf{b}\ \ A\mathbf{b}\ \ A^2\mathbf{b}\ \ \cdots\ \ A^k\mathbf{b}]$. Compute $K = QR$, then the columns of Q span the *Krylov subspace*. Look for an approximate solution of $A\mathbf{x} = \mathbf{b}$ in the span of the columns of Q. We can interpret this as a modified Gram-Schmidt method, stopped after k projections of $\mathbf{b}$.

The conjugate gradient and Krylov subspace methods rely mostly on matrix-vector products. Parallel implementations involve the distribution of the matrix among the processors. For general matrices, tiling scales best. For specific sparse matrices, the patterns of the nonzero elements will determine the optimal distribution.

PETSc (see Lecture 20) provides a directory of examples `ksp` of Krylov subspace methods. We will take `ex3` of `/usr/local/petsc-3.4.3/src/ksp/ksp/examples/tests` To get the makefile, we compile in the examples and then copy the compilation and linking instructions. The makefile on kepler looks as follows:

```
PETSC_DIR=/usr/local/petsc-3.4.3
PETSC_ARCH=arch-linux2-c-debug

ex3:
        $(PETSC_DIR)/$(PETSC_ARCH)/bin/mpicc -o ex3.o -c -fPIC -Wall \
        -Wwrite-strings -Wno-strict-aliasing -Wno-unknown-pragmas -g3 \
        -fno-inline -O0   -I/usr/local/petsc-3.4.3/include \
        -I/usr/local/petsc-3.4.3/arch-linux2-c-debug/include \
        -D__INSDIR__=src/ksp/ksp/examples/tests/ ex3.c
        $(PETSC_DIR)/$(PETSC_ARCH)/bin/mpicc -fPIC -Wall -Wwrite-strings \
        -Wno-strict-aliasing -Wno-unknown-pragmas -g3 -fno-inline -O0 \
        -o ex3 ex3.o \
        -Wl,-rpath,/usr/local/petsc-3.4.3/arch-linux2-c-debug/lib \
        -L/usr/local/petsc-3.4.3/arch-linux2-c-debug/lib  -lpetsc \
        -Wl,-rpath,/usr/local/petsc-3.4.3/arch-linux2-c-debug/lib \
        -lflapack -lfblas -lX11 -lpthread -lm \
        -Wl,-rpath,/usr/gnat/lib/gcc/x86_64-pc-linux-gnu/4.7.4 \
        -L/usr/gnat/lib/gcc/x86_64-pc-linux-gnu/4.7.4 \
        -Wl,-rpath,/usr/gnat/lib64 -L/usr/gnat/lib64 \
        -Wl,-rpath,/usr/gnat/lib -L/usr/gnat/lib -lmpichf90 -lgfortran \
        -lm -Wl,-rpath,/usr/lib/gcc/x86_64-redhat-linux/4.4.7 \
        -L/usr/lib/gcc/x86_64-redhat-linux/4.4.7 -lm -lmpichcxx -lstdc++ \
        -ldl -lmpich -lopa -lmpl -lrt -lpthread -lgcc_s -ldl
        /bin/rm -f ex3.o
```

Running the example produces the following:

```
# time /usr/local/petsc-3.4.3/arch-linux2-c-debug/bin/mpiexec \
-n 16 ./ex3 -m 200
Norm of error 0.000622192 Iterations 162
```

```
real    0m1.855s    user    0m9.326s      sys    0m4.352s
# time /usr/local/petsc-3.4.3/arch-linux2-c-debug/bin/mpiexec \
-n 8 ./ex3 -m 200
Norm of error 0.000387934 Iterations 131
real    0m2.156s    user    0m10.363s     sys    0m1.644s
# time /usr/local/petsc-3.4.3/arch-linux2-c-debug/bin/mpiexec \
-n 4 ./ex3 -m 200
Norm of error 0.00030938 Iterations 105
real    0m4.526s    user    0m15.108s     sys    0m1.123s
# time /usr/local/petsc-3.4.3/arch-linux2-c-debug/bin/mpiexec \
-n 2 ./ex3 -m 200
Norm of error 0.000395339 Iterations 82
real    0m20.606s   user    0m37.036s     sys    0m1.996s
# time /usr/local/petsc-3.4.3/arch-linux2-c-debug/bin/mpiexec \
-n 1 ./ex3 -m 200
Norm of error 0.000338657 Iterations 83
real    1m34.991s   user    1m25.251s     sys    0m9.579s
```

6.5.3 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 3.4*. Mathematics and Computer Science Division, Argonne National Laboratory, 15 October 2013.
2. E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan. *PLASMA Users' Guide. Parallel Linear Algebra Software for Multicore Architectures. Version 2.0*.
3. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing* 35: 38-53, 2009.

6.5.4 Exercises

1. Take a 3-by-3 block matrix, apply the tiled QR algorithm step by step, indicating which block is processed in each step along with the type of operations. Draw a Directed Acyclic Graph linking steps that have to wait on each other. On a 3-by-3 block matrix, what is the number of cores needed to achieve the best speedup?
2. Using the `time` command to measure the speedup of `example_dgeqrs` also takes into account all operations to define the problem and to test the quality of the solution. Run experiments with modifications of the source code to obtain more accurate timings of the parallel QR factorization.

CHAPTER 7

Big Data and Cloud Computing

7.1 Disk Based Parallelism

We covered distributed memory and shared memory parallelism as two separate technologies. Adding a software layer on top of message passing, as the software roomy does, provides the parallel programmer with a vast random access memory.

7.1.1 The Disk is the new RAM

When is random access memory not large enough? For the context of this lecture, we consider big data applications. In applications with huge volumes of data, the bottleneck is not the number of arithmetical operations. *The disk is the new RAM*. `Roomy` is a software library that allows to treat disk as Random Access Memory. An application of `Roomy` concerns the determination of the minimal number of moves to solve Rubik's cube.

In *parallel disk-based computation*, instead of Random Access Memory, we use disks as the main working memory of a computation. This gives much more space for the same price. There are at least two performance issues for which we give solutions.

1. Bandwidth: the bandwidth of a disk is roughly 50 times less than RAM (100 MB/s versus 5 GB/s). The solution is to use many disks in parallel.
2. Latency: even worse, the latency of disk is many orders of magnitude worse than RAM. The solution to avoid latency penalties is to use streaming access.

As an example we take the pancake sorting problem. Given a stack of n numbered pancakes, at the left of Fig. 7.1. A spatula can reverse the order of the top k pancakes for $2 \leq k \leq n$.

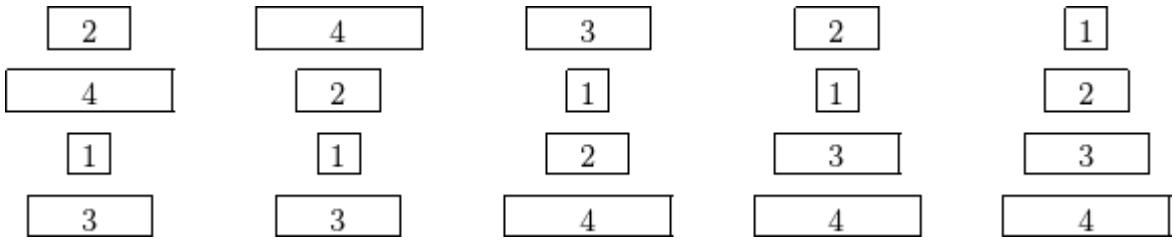


Fig. 7.1: Sorting a pile of pancakes with flips.

The sort happened with 4 flips:

1. k = 2

2. k = 4
3. k = 3
4. k = 2

The question we ask it the following: How many flips (prefix reversals) are sufficient to sort?

To understand this question we introduce the pancake sorting graph. We generate all permutations using the flips, as in Fig. 7.2.

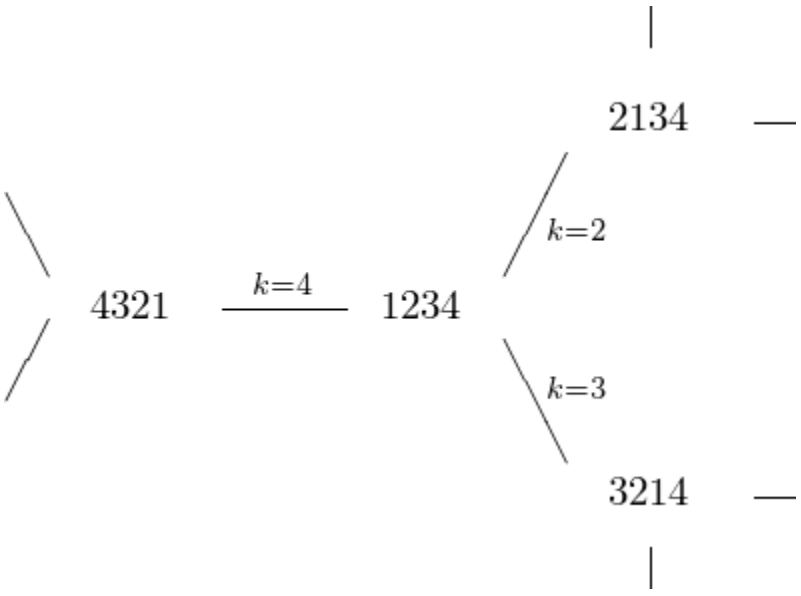


Fig. 7.2: Permutations generated by flips.

There are 3 stacks that require 1 flip. 4! = 24 permutations, $24 = 1 + 3 + 6 + 11 + 3$, at most 4 flips are needed for a stack of 4 pancakes.

A breadth first search, running the program `pancake`, its output is displayed below:

```
$ mpiexec -np 2 ./pancake
Sun Mar  4 13:24:44 2012: BEGINNING 11 PANCAKE BFS
Sun Mar  4 13:24:44 2012: Initial one-bit RoomyArray constructed
Sun Mar  4 13:24:44 2012: Level 0 done: 1 elements
Sun Mar  4 13:24:44 2012: Level 1 done: 10 elements
Sun Mar  4 13:24:44 2012: Level 2 done: 90 elements
Sun Mar  4 13:24:44 2012: Level 3 done: 809 elements
Sun Mar  4 13:24:44 2012: Level 4 done: 6429 elements
Sun Mar  4 13:24:44 2012: Level 5 done: 43891 elements
Sun Mar  4 13:24:45 2012: Level 6 done: 252737 elements
Sun Mar  4 13:24:49 2012: Level 7 done: 1174766 elements
Sun Mar  4 13:25:04 2012: Level 8 done: 4126515 elements
Sun Mar  4 13:25:45 2012: Level 9 done: 9981073 elements
Sun Mar  4 13:26:48 2012: Level 10 done: 14250471 elements
Sun Mar  4 13:27:37 2012: Level 11 done: 9123648 elements
Sun Mar  4 13:27:53 2012: Level 12 done: 956354 elements
Sun Mar  4 13:27:54 2012: Level 13 done: 6 elements
Sun Mar  4 13:27:54 2012: Level 14 done: 0 elements
Sun Mar  4 13:27:54 2012: ONE-BIT PANCAKE BFS DONE
Sun Mar  4 13:27:54 2012: Elapsed time: 3 m, 10 s, 536 ms, 720 us
```

How many flips to sort a stack of 11 pancakes? $11 \times 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 = 39,916,800$.

Table 7.1: Number of flips to sort 11 pancakes.

#moves	#stacks
0	1
1	10
2	90
3	809
4	6429
5	43891
6	252737
7	1174766
8	4126515
9	9981073
10	14250471
11	9123648
12	956354
13	6
total	39916800

So we find that at most 13 moves are needed for a stack of 11 pancakes.

7.1.2 Roomy: A System for Space Limited Computations

`Roomy` is a new programming model that extends a programming language with transparent disk-based computing supports; It an open source C/C++ library implementing this new programming language extension; available at `sourceforge`; written by Daniel Kunkle in the lab of Gene Cooperman.

Roomy has been applied to problems in computational group theory, and large enumerations, e.g.: Rubik's cube can be solved in 26 moves or less.

Example: 50-node cluster with 200GB disk space per computer gives 10TB per computer. Bandwidth of one disk: 100MB/s, bandwidth of 50 disks in parallel: 5GB/s So, the 10TB disk space is considered as RAM.

Some caveats remain. Disk latency remains limiting. Use old-fashioned RAM as cache. If disk is the new RAM, RAM is the new cache. Networks must restructured to emphasize local access over network access.

The Roomy programming programming model

1. provides basic data structures: arrays, lists, and hash tables; transparently distributes data structures across many disks and
2. performs operations on that data in parallel;
3. immediately process streaming access operators;
4. delays processing random operators until they can be performed efficiently in batch.

For example: collecting and sorting updates to an array. We introduce some programming concepts of `Roomy`. The first programming construct we consider is `map`.

```
RoomyArray* ra;
RoomyList* rl;
// Function to map over ra.
void mapFunc ( uint64 i, void* val )
{
```

```
   RoomyListadd(rl,val);
}

int main ( int argc, char **argv )
{
   Roomy_init(&argc,&argv);
   ra = RoomyArray_makeBytes("array",sizeof(uint64),100);
   rl = RoomyList_make("list",sizeof(uint64));
   RoomyArray_map(ra,mapFunc); // execute map
   RoomyList_sync(rl); // synchronize list for delayed ads
   Roomy_finalize();
}
```

The next programming construct is `reduce`. Computing for example the sum of squares of the elements in a `RoomyList`:

```
RoomyList* rl;  // elements of type int

// add square of an element to sum
void mergeElt ( int* sum, int* element )
{
   *sum += (*e) * (*e);
}

// compute sum of two partial answers
void mergeResults ( int * sum1, int* sum2 )
{
    *sum1 += *sum2;
}
int sum = 0;
RoomyList_reduce(rl,&sum,sizeof(int),mergeElt,mergeResults);
```

Another programming construct is the predicate. Predicates count the number of elements in a data structure that satisfy a Boolean function.

```
RoomyList* rl;

// predicate: return 1 if element is greater than 42
uint8 predFunc ( int* val )
{
   return (*val > 42) ? 1 : 0;
}

RoomyList_attachPredicate(rl,predFunc);

uint64 gt42 = RoomyList predicateCount(rl,predFunc);
```

The next programming construct we illustrate is `permutation multiplication`. For permutations X, Y, Z on N elements length N do Z = X*Y as `for i=0 to N-1:  Z[i] = Y[X[i]]`.

```
RoomyArray *X, *Y, *Z;
// access X[i]
void accessX ( unit64 i, uint64* x_i)
{
   RoomyArray_access(Y, *x_i,&i, accessY);
}
// access Y[X[i]]
void accessY ( uint64 x_i, uint64* y_x_i, uint64* i )
```

```
{
   RoomyArray_update(Z,*i,y_x_i,setZ);
}
// set Z[i] = Y[X[i]]
void setZ ( uint64, i, uint64* z_i, uint64* y_x_i, uint64* z_i_NEW )
{
   *z_i_NEW = *y_x_i;
}
RoomyArray_map(X,accessX); // access X[i]
RoomyArray_sync(Y);        // access Y[X[i]]
RoomyArray_sync(Z);        // set Z[i] = Y[X[i]]
```

Roomy supports set operations as programming construct. Convert list to set:

```
RoomyList *A; // can contain duplicates
RoomyList_removeDupes(A); // is a set
```

Union of two sets $A \cup B$:

```
RoomyList *A, *B;
RoomyList_addAll(A,B);
RoomyList_removeDupes(A);
```

Difference of two sets $A \setminus B$:

```
RoomyList *A, *B;
RoomyList_removeAll(A,B);
```

The intersection $A \cap B$ is implemented as $(A \cup B) \setminus (A \setminus B) \setminus (B \setminus A)$.

The last programming construct we consider is `breadth-first search`. Initializing the search:

```
// Lists of all elements, current, and next level
RoomyList* all = RoomyList_make("allLev",eltSize);
RoomyList* cur = RoomyList_make("lev0",eltSize);
RoomyList* next = Roomy_list_Make("lev1",eltSize);

// Function to produce next level from current
void genNext ( T elt )
{
   // user defined code to compute neighbors ...
   for nbr in neighbors
      RoomyList_add(next,nbr);
}
// add start element
RoomyList_add(all,startElt);
RoomyList_add(cur,startElt);
```

Performing the search is shown below:

```
// generate levels until no new states are found
while(RoomyList_size(cur))
{  // generate next level from current
   RoomyList_map(cur,genNext);
   RoomyList_sync(next);
   // detect duplicates within next level
   RoomyList_removeDupes(next);
   // detect duplicates from previous levels
```

```
   RoomyList_removeAll(next,all);
   // record new elements
   RoomyList_addAll(all,next);
   // rotate levels
   RoomyList_destroy(cur);
   cur = next;
   newt = RoomyList_make(levName,eltSize);
}
```

7.1.3 Hadoop and the Map/Reduce model

To process big data with parallel and distributed computing, we could use MySQL which scales well. Hadoop has become the de facto standard for processing big data.

Hadoop is used by Facebook to store photos, by LinkedIn to generate recommendations, and by Amazon to generate search indices. Hadoop works by connecting many different computers, hiding the complexity from the user: work with one giant computer. Hadoop uses a model called Map/Reduce.

The goal of the Map/Reduce model is to help with writing efficient parallel programs. Common data processing tasks: filtering, merging, aggregating data and many (not all) machine learning algorithms fit into Map/Reduce. There are two steps:

- Map:

  Tasks read in input data as a set of records, process the records, and send groups of similar records to reducers. The mapper extracts a key from each input record. Hadoop will then route all records with the same key to the same reducer.

- Reduce:

  Tasks read in a set of related records, process the records, and write the results. The reducer iterates through all results for the same key, processing the data and writing out the results.

The strength of this model is that the map and reduce steps run well in parallel.

Consider for example the prediction of user behavior. The question we ask is: How likely is a user to purchase an item from a website?

Suppose we have already computed (maybe using Map/Reduce) a set of variables describing each user: most common locations, the number of pages viewed, and the number of purchases made in the past. One method to calculate a forecast is to use random forests. Random forests work by calculating a set of regression trees and then averaging them together to create a single model. It can be time consuming to fit the random trees to the data, but each new tree can be calculated independently. One way to tackle this problem is to use a set of map tasks to generate random trees, and then send the models to a single reducer task to average the results and produce the model.

7.1.4 Bibliography

1. Daniel Kunkle. Roomy: A C/C++ library for parallel disk-based computation, 2010. <http://roomy.sourceforge.net/>.
2. Daniel Kunkle and Gene Cooperman. Harnessing parallel disks to solve Rubik's cube. *Journal of Symbolic Computation* 44:872-890, 2009.
3. Daniel Kunkle and Gene Cooperman. Solving Rubik's Cube: Disk is the new RAM. *Communications of the ACM* 51(4):31-33, 2008.
4. Garry Turkington. Hadoop Beginner's Guide. <www.it-ebooks.info>.

7.1.5 Exercises

1. Watch the YouTube google tech talk of Gene Cooperman on the application of disk parallelism to Rubik's cube.
2. Read the paper of Daniel Kunkle and Gene Cooperman that was published in the Journal of Symbolic Computation, see the bibliography.

7.2 Introduction to Hadoop

7.2.1 What is Hadoop?

We are living in an era where large volumes of data are available and the problem is to extract meaning from the data avalanche. The goal of the software tools is to apply complex analytics to large data sets. A complementary technology is the use of cloud computing, in particular: Amazon Web Services. Assumptions for writing MapReduce applications is that one is comfortable writing Java programs; and familiar with the Unix command-line interface.

What is the value of data? Some questions are relevant only for large data sets. For example: movie preferences are inaccurate when based on just another person, but patterns can be extracted from the viewing history of millions. Big data tools enable processing on larger scale at lower cost. Additional hardware is needed to make up for latency. The notion of what is a database should be revisited. It is important to realize that one no longer needs to be among the largest corporations or government agencies to extract value from data.

The are two ways to process large data sets:

1. *scale-up*: large and expensive computer (supercomputer). We move the same software onto larger and larger servers.
2. *scale-out*: spread processing onto more and more machines (commodity cluster).

These two ways are subject to limiting factors. One has to deal with the complexity of concurrency in multiple CPUs; and CPUs are much faster than memory and hard disk speeds. There is a third way: *Cloud computing* In this third way, the provider deals with scaling problems.

The principles of Hadoop are listed below:

1. *All roads lead to scale-out*, scale-up architectures are rarely used and scale-out is the standard in big data processing.
2. *Share nothing*: communication and dependencies are bottlenecks, individual components should be as independent as possible to allow to proceed regardless of whether others fail.
3. *Expect failure*: components will fail at inconvenient times. See our previous exercise on multi-component expected life span; resilient scale-up architectures require much effort.
4. *Smart software, dumb hardware*: push smarts in the software, responsible for allocating generic hardware.
5. *Move processing, not data*: perform processing locally on data. What gets moved through the network are program binaries and status reports, which are dwarfed in size by the actual data set.
6. *Build applications, not infrastructure.* Instead of placing focus on data movement and processing, work on job scheduling, error handling, and coordination.

Who to thank for Hadoop? A brief history of Hadoop: Google released two academic papers describing their technology: in 2003: the google file system; and in 2004: MapReduce. The papers are available for download from <google.research.com.> Doug Cutting started implementing Google systems, as a project within the Apache open source foundation. Yahoo hired Doug Cutting in 2006 and supported Hadoop project.

From Tom White: *Hadoop: The Definitive Guide*, published by O'Reilly, on the name Hadoop:

The name my kid gave a stuffed yellow elephant. Short, relatively easy to spell and pronounce, meaningless, and not used elsewhere: those are my naming criteria. Kids are good at generating such.

Two main components of Hadoop are the Hadoop Distributed File System (HDFS) and MapReduce. HDFS stores very large data sets across a cluster of hosts, it is optimized for throughput instead of latency, achieving high availability through replication instead of redundancy. MapReduce is a data processing paradigm that takes a specification of input (map) and output (reduce) and applies this to the data. MapReduce integrates tightly with HDFS, running directly on HDFS nodes.

Common building blocks are

1. run on commodity clusters, scale-out: can add more servers;
2. have mechanisms for identifying and working around failures;
3. provides services transparently, user can concentrate on problem; and
4. software cluster sitting on physical server controls execution.

The Hadoop Distributed File System (HDFS) spreads the storage across nodes. Its features are thatfiles stored in blocks of 64MB (typical file system: 4-32 KB); it is optimized for throughput over latency, efficient at streaming, but poor at seek requests for many smale ones; optimized for workloads that are read-many and write-once; storage node runs process DataNode that manages blocks, coordinated by master NameNode process running on separate host; and replicates blocks on multiple nodes to handle disk failures.

MapReduce is a new technology built on fundamental concepts: functional programming languages offer map and reduce; divide and conquer breaks problem into multiple subtasks. The input data goes to a series of transformations: the developer defines the data transformations, Hadoop's MapReduce job manages parallel execution.

Unlike relational databases the required structure data, the data is provided as a series of key-value pairs and the output of the map is another set of key-value pairs. The most important point we should not forget is: *the Hadoop platform takes responsibility for every aspect of executing the processing across the data.* The user is unaware of the actual size of data and cluster, the platform determines how best to utilize all the hosts.

The challenge to the user is to break the problem into the best combination of chains of map and reduce functions, where the output of one is the input of the next; and em where each chain could be applied independently to each data element, allowing for parallelism.

The common architecture of HDFS and MapReduce are software clusters: cluster of worker nodes managed by a coordinator node; master (NameNode for HDFS and JobTracker for MapReduce) monitors clusters and handles failures; processes on server (DataNode for HDFS and TaskTracker for MapReduce) perform work on physical host, receiving instructions for master and reporting status. A multi-node Hadoop cluster in an image copied from wikipedia is shown below:

Strengths and weaknesses: Hadoop is flexible and scalable data processing platform; but batch processing not for real time, e.g.: serving web queries. Application by Google, in three stages.

1. A web crawler retrieves updated webpage data.
2. MapReduces processes the huge data set.
3. The web index is then used by a fleet of MySQL servers for search requests.

Cloud computing with Amazon Web Services is another technology. Two main aspects are that it is new architecture option; and it gives a different approach to cost. Instead of running your own clusters, all you need is a credit card. This is the third way, recall the three ways:

1. scale-up: supercomputer;
2. scale-out: commodity cluster;
3. cloud computing: the provider deals with the scaling problem.

How does cloud computing work? There are wo steps:

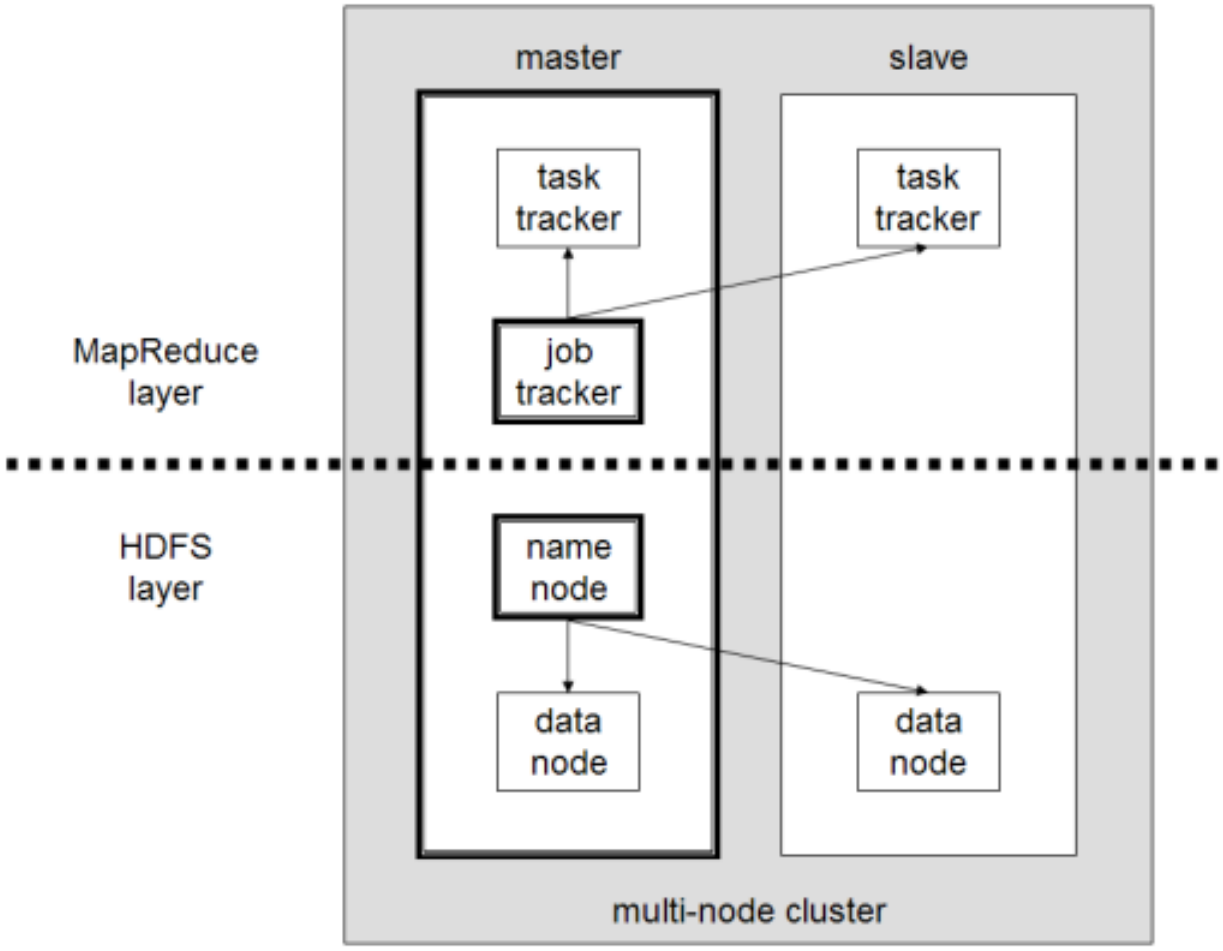


Fig. 7.3: A multi-node Hadoop cluster.

1. The user develops software according to published guidelines or interface; and
2. then deploys onto the cloud platform and allow the service to scale on demand.

The service-on-demand aspect allows to start an application on small hardware and scale up, off loading infrastructure costs to cloud provider. The major benefit is a great reduction of cost of entry for organization to launch an online service.

Amazon Web Services (AWS) is an infrastructure on demand A set of cloud computing services: EC2, S3, and EMR. Elastic Compute Cloud (EC2): a server on demand. Simple Storage Service (S3): programmatic interface to store objects. Elastic MapReduce (EMR): Hadoop in the cloud. In most impressive mode: EMR pulls data from S3, process on EC2.

7.2.2 Understanding MapReduce

We can view MapReduce as a series of key/value transformations:

```
         map                  reduce
{K1, V1} --> {K2, List<V2>} -----> {K3, V3}
```

Two steps, three sets of key-value pairs:

1. The input to the map is a series of key-value pairs, called `K1` and `V1`.
2. The output of the map (and the input to the reduce) is a series of keys and an associated list of values that are called `K2` and `V2`.
3. The output of the reduce is another series of key-value pairs, called `K3` and `V3`.

The word count problem is the equivalent to hello world in Hadoop. On input is a text file. The output lists the frequency of words. Project Gutenberg (at <www.gutenberg.org>) offers many free books (mostly classics) in plain text file format. Doing a word count is a very basic data analytic, same authors will use the same patterns of words.

In solving the word count problem with MapReduce, we determine the keys and the corresponding values. Every word on the text file is a key. Its value is the count, its number of occurrences. Keys must be unique, but values need not be. Each value must be associate with a key, but a key could have no values One must be careful when defining what is a key, e.g. for the word problem do we distinguish between lower and upper case words?

A complete MapReduce Job for the word count problem:

1. Input to the map: `K1/V1` pairs are in the form `< line number,text on the line >`. The mapper takes the line and breaks it up into words.
2. Output of the map, input of reduce: `K2/V2` pairs are in the form `< word,1 >`}.
3. Output of reduce: `K3/V3` pairs are in the form `< word,count >`}.

Pseudo code for the word count problem is listed below:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
    for each v in values:
```

```
    result += ParseInt(v);
  Emit(AsString(result));
```

some more examples

- Distributed Grep

  The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

- Count of URL Access Frequency

  The map function processes logs of web page requests and outputs `(URL,1)`. The reduce function adds together all values for the same URL and emits a `(URL,total count)` pair.

- Reverse Web-Link Graph

  The map function outputs (`target`, `source`) pairs for each link to a target URL found in a page named `source`. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: (`target`, *list* (`source`)).

- Term-Vector per Host

  A term vector summarizes the most important words that occur in a document or a set of documents as a list of (word, frequency) pairs. The map function emits a (`hostname`, `term vector`) pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final (`hostname`, `term vector`) pair.

- Inverted Index

  The map function parses each document, and emits a sequence of (`word`, `document ID`) pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a (`word`, *list* (`document ID`)) pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

- Distributed Sort

  The map function extracts the key from each record, and emits a `key`, `record`) pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning facilities: hashing keys; and and the ordering properties: within a partition key-value pairs are processed in increasing key order.

7.2.3 Bibliography

1. Luiz Barroso, Jeffrey Dean, and Urs Hoelzle. Web Search for a Planet: The Google Cluster Architecture. <research.google.com/archive/googlecluster.html>.
2. Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. At <research.google.com/archive/mapreduce.html>. Appeared in OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December 2004.
3. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. At <research.google.com/archive/gfs.html>. Appeared in 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003.
4. Garry Turkington. Hadoop Beginner's Guide. <www.it-ebooks.info>.

7.2.4 Exercises

1. Install Hadoop on your computer and run the word count problem.
2. Consider the pancake problem we have seen in Lecture 23 and formulate a solution within the MapReduce programming model.

CHAPTER 8

Architecture and Programming Models for the GPU

8.1 A Massively Parallel Processor: the GPU

8.1.1 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; second edition, 2013.

The site <http://gpgpu.org> is a good start for many tutorials.

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 equipment: hardware and software

Our Microway workstation has an NVIDIA GPU with the CUDA software development installed.

- NVDIA Tesla K20c general purpose graphics processing unit
  1. number of CUDA cores: 2,496 (13 $times$ 192)
  2. frequency of CUDA cores: 706MHz
  3. double precision floating point performance: 1.17 Tflops (peak)
  4. single precision floating point performance: 3.52 Tflops (peak)
  5. total global memory: 4800 MBytes
- CUDA programming model with `nvcc` compiler.

To compare the theoretical peak performance of the K20C, consider the theoretical peak performance of the two Intel E5-2670 (2.6GHz 8 cores) CPUs in the workstation:

- 2.60 GHz $\times$ 8 flops/cycle = 20.8 GFlops/core;
- 16 core $\times$ 20.8 GFlops/core = 332.8 GFlops.

$\Rightarrow$ 1170/332.8 = 3.5. One K20c is as strong as 3.5 $\times$ 16 = 56.25 cores.

CUDA stands for Compute Unified Device Architecture, is a general purpose parallel computing architecture introduced by NVDIA.

8.1.2 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. 8.1 and the memory bandwidth in Fig. 8.2.

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, the memory bandwidth of the CPUs is 10.66GB/s. For the NVDIA Tesla K20c the memory bandwidth is 143GB/s. Straightforward parallel implementations on GPGPUs often achieve directly a speedup of 10, saturating the memory bandwidth.

CPU and GPU design

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. 8.3.

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.

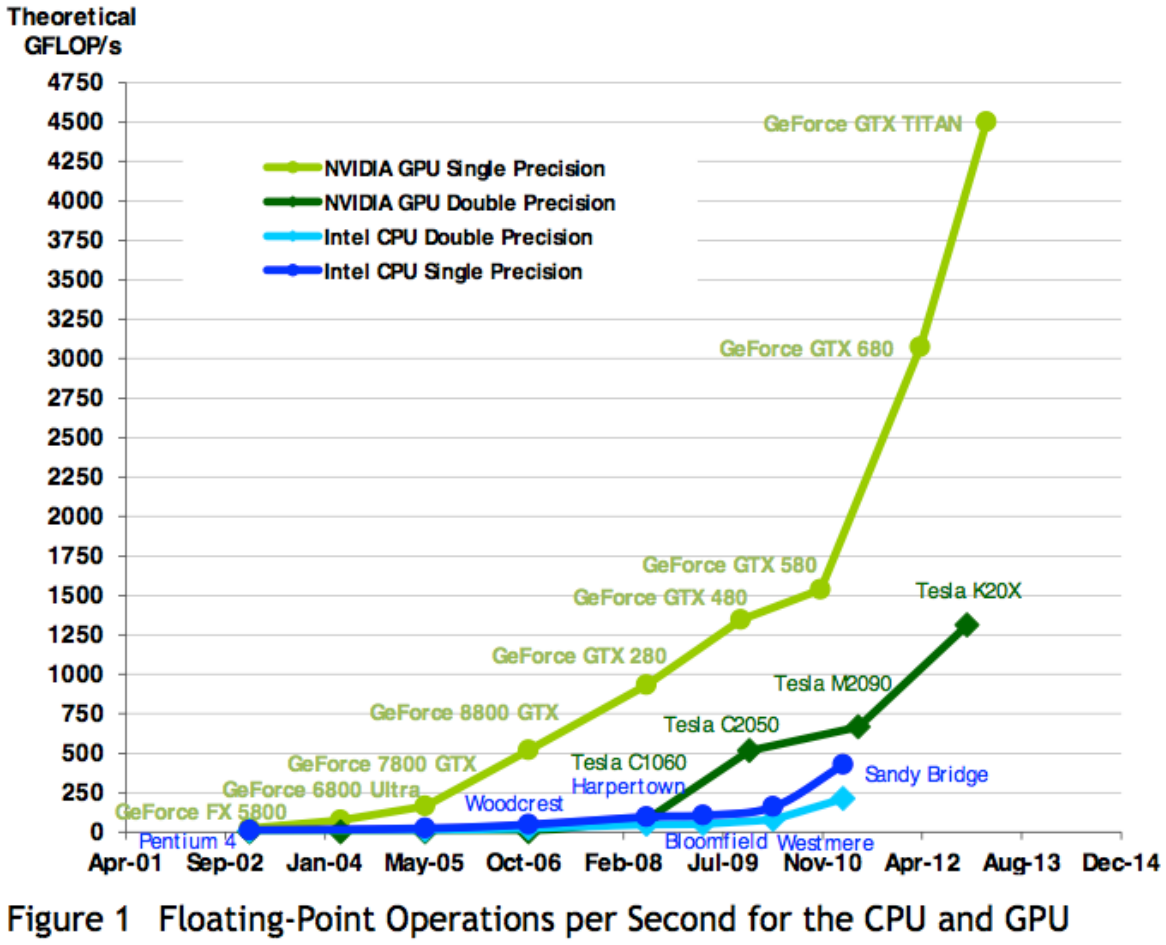


Figure 1 Floating-Point Operations per Second for the CPU and GPU

Fig. 8.1: Flops comparison taken from the NVIDIA programming guide.

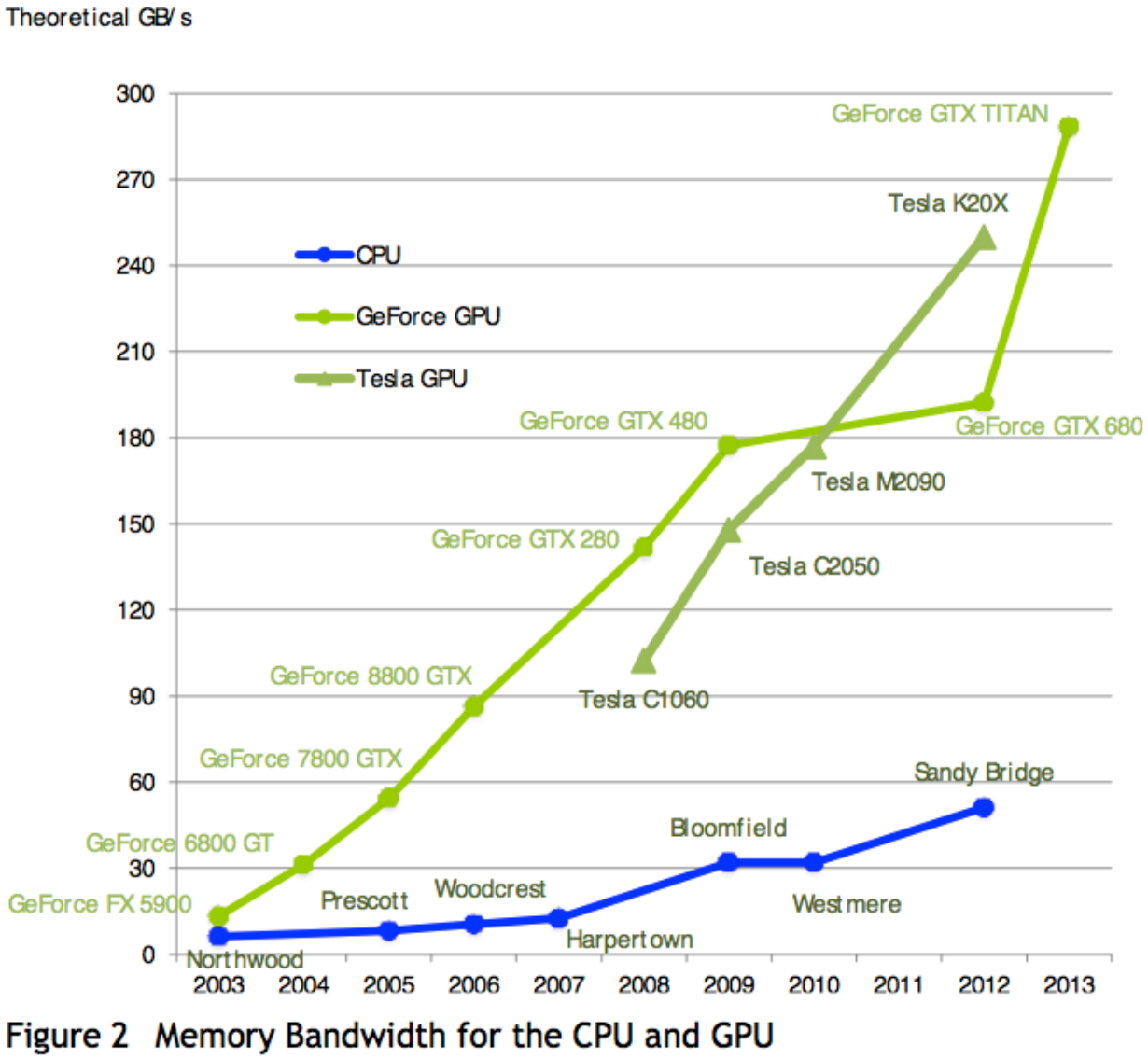


Fig. 8.2: Bandwidth comparision taken from the NVIDIA programming guide.

Fig. 8.3: Distinction between the design of a CPU and a GPU.

- 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. 8.4.

begin{frame}{our NVIDIA Tesla K20C GPU}

Our K20C Graphics card has

- 13 streaming multiprocessors (SM),
- each SM has 192 streaming processors (SP),
- $13 \times 192 = 2496$ cores.

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.

programming models and data parallelism

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

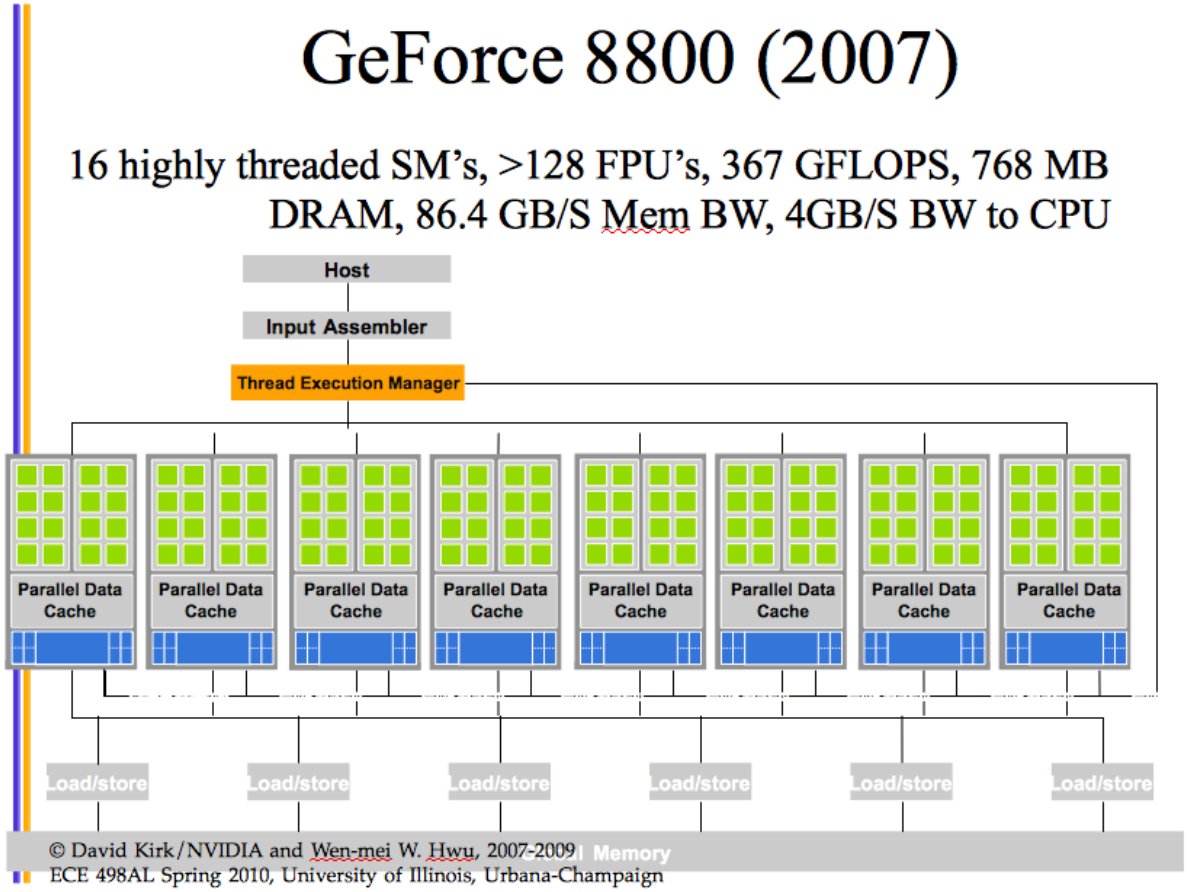


Fig. 8.4: A concrete example of the GPU architecture.

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;
- 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).

8.1.3 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.

8.1.4 Exercises

0. Visit <http://gpgpu.org>.

8.2 Evolution of Graphics Pipelines

8.2.1 Understanding the Graphics Heritage

Graphics processing units (GPUs) are massively parallel numeric computing processors, programmed in C with extensions. Understanding the graphics heritage illuminates the strengths and weaknesses of GPUs with respect to major computational patterns. The history clarifies the rationale behind major architectural design decisions of modern programmable GPUs:

- massive multithreading,
- relatively small cache memories compared to caches of CPUs,
- bandwidth-centric memory interface design.

Insights in the history provide the context for the future evolution. Three dimensional (3D) graphics pipeline hardware evolved from large expensive systems of the early 1980s to small workstations and then PC accelerators in the mid to late 1990s. During this period, the performance increased:

- from 50 millions pixes to 1 billion pixels per second,
- from 100,000 vertices to 10 million vertices per second.

This advancement was driven by market demand for high quality, real time graphics in computer applications. The architecture evolved from a simple pipeline for drawing wire frame diagrams to a parallel design of several deep parallel pipelines capable of rendering the complex interactive imagery of 3D scenes. In the mean time, graphics processors became programmable.

the stages to render triangles

In displaying images, the parts in the GPU are shown in Fig. 8.5.

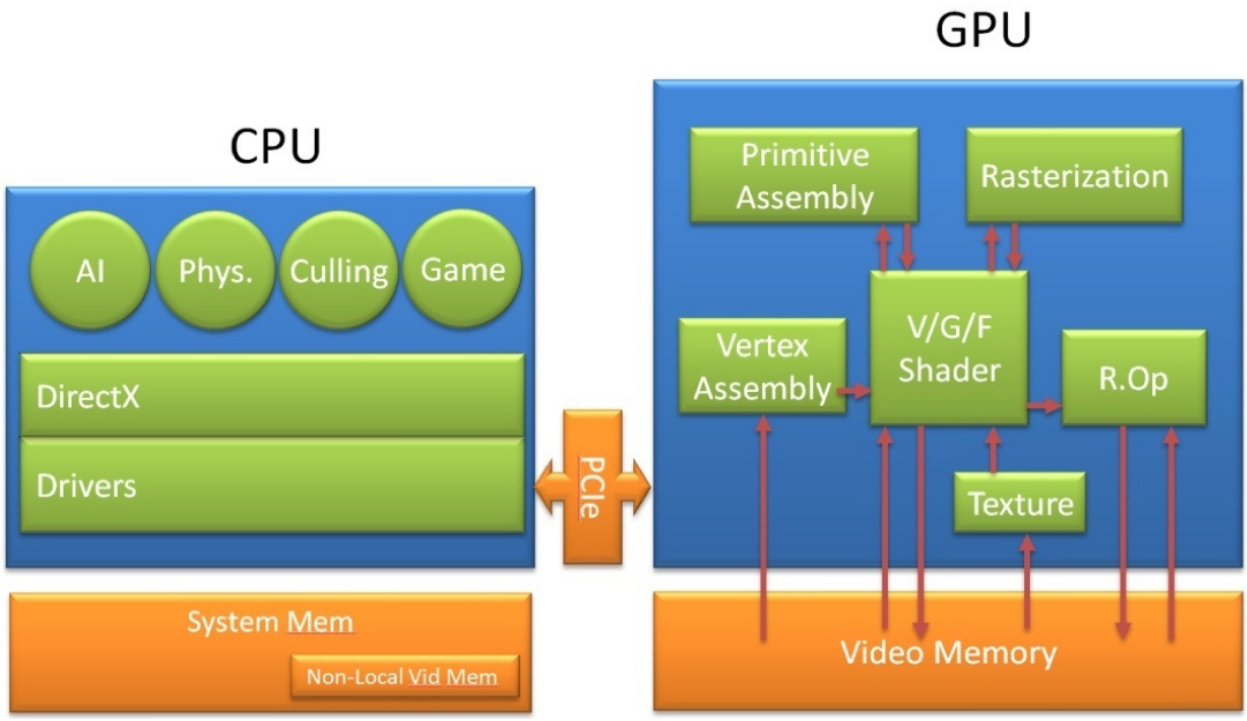


Fig. 8.5: From the GeForce 8 and 9 Series GPU Programming Guide (NVIDIA).

The surface of an object is drawn as a collection of triangles. The Application Programming Interface (API) is a standardized layer of software that allows an application (e.g.: a game) to send commands to a graphics processing unit to draw objects on a display. Examples of such APIs are DirectX and OpenGL. The host interface (the interface to the GPU) receives graphics commands and data from the CPU, communicates back the status and result data of the execution.

The two parts of a fixed function pipeline are shown in Fig. 8.6 and in Fig. 8.7.

The stages in the first part of the pipeline are as follows:

1. vertex control

   This stage receives parametrized triangle data from the CPU. The data gets converted and placed into the vertex cache.

2. VS/T & L (vertex shading, transform, and lighting)

   The VS/T & L stage transforms vertices and assigns per-vertex values, e.g.: colors, normals, texture coordinates, tangents. The vertex shader can assign a color to each vertex, but color is not applied to triangle pixels until later.

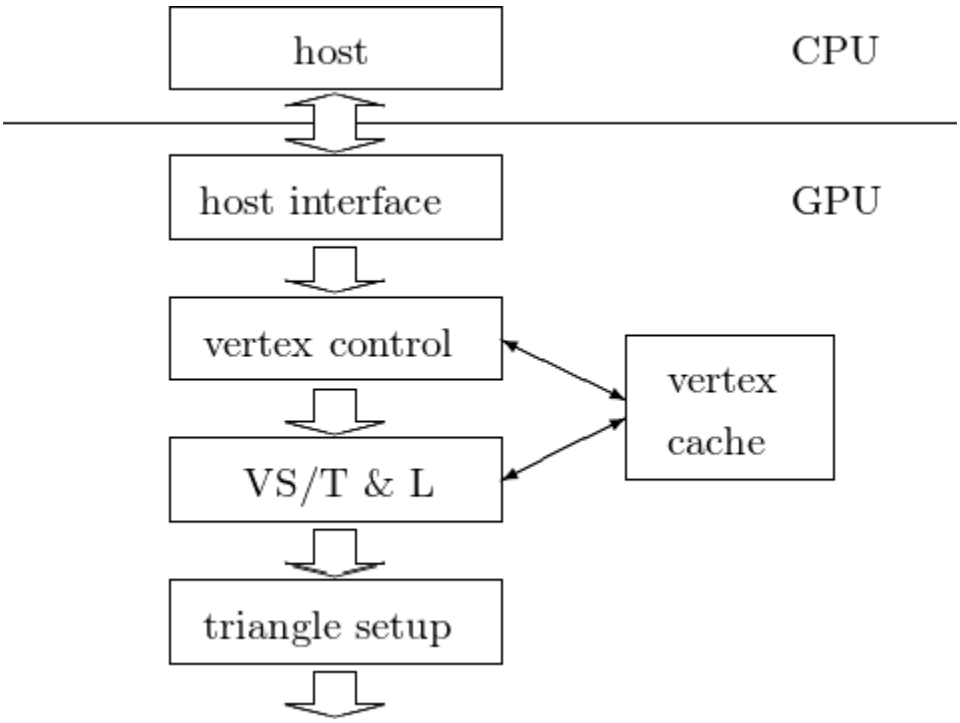


Fig. 8.6: Part one of a fixed function pipeline. VS/T & L = vertex shading, transform, and lighting.

3. triangle setup

   Edge equations are used to interpolate colors and other per-vertex data across the pixels touched by the triangle.

The stages in the second part of the pipeline are as follows:

4. raster

   The raster determines which pixels are contained in each triangle. Per-vertex values necessary for shading are interpolated.

5. shader

   The shader determines the final color of each pixel as a combined effect of interpolation of vertex colors, texture mapping, per-pixel lighting, reflections, etc.

6. ROP (Raster Operation)

   The final raster operations blend the color of overlapping/adjacent objects for transparency and antialiasing effects. For a given viewpoint, visible objects are determined and occluded pixels (blocked from view by other objects) are discarded.

7. FBI (Frame Buffer Interface)

   The FBI stages manages memory reads from and writes to the display frame buffer memory.

   For high-resolution displays, there is a very high bandwidth requirement in accessing the frame buffer. High bandwidth is achieved by two strategies: using special memory designs; and managing simultaneously multiple memory channels that connect to multiple memory banks.

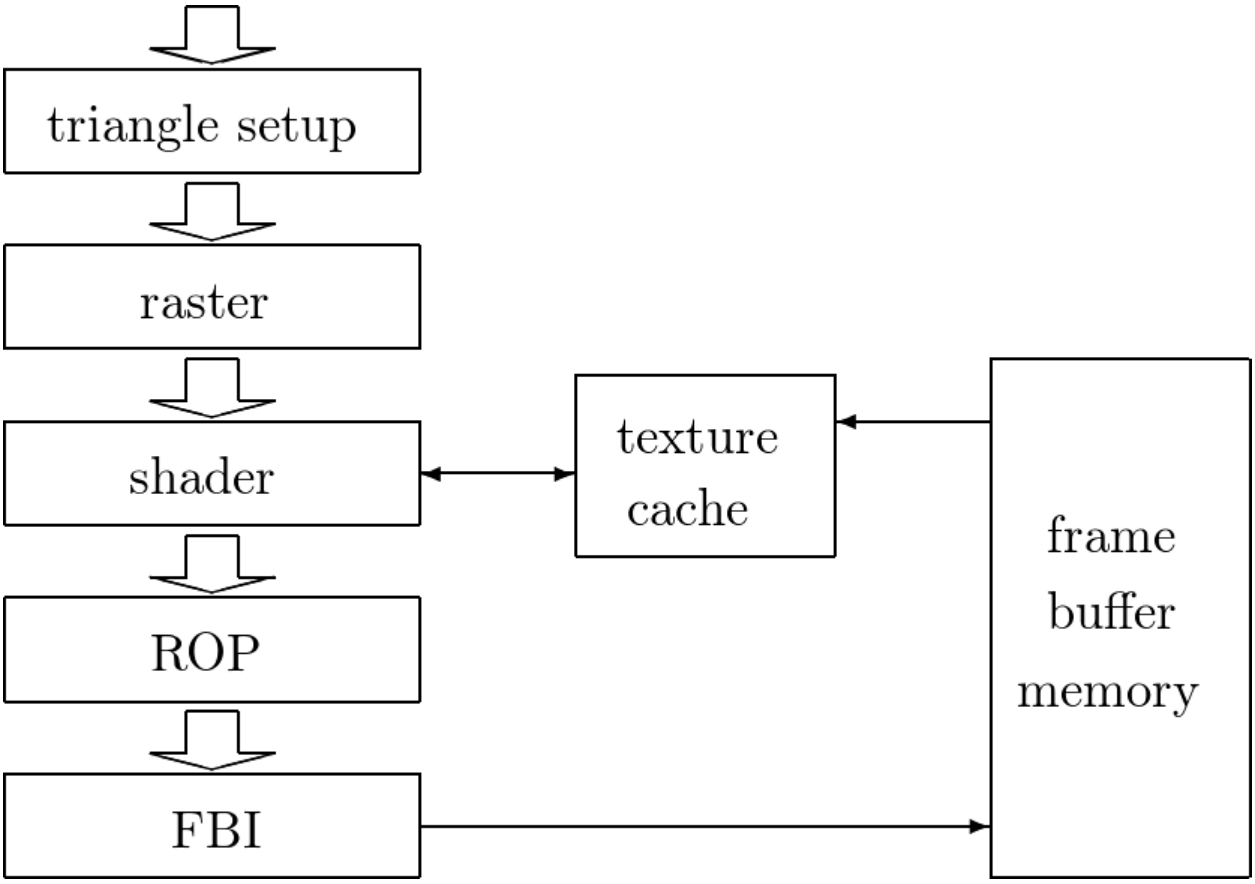


Fig. 8.7: Part two of a fixed-function pipeline. ROP = Raster Operation, FBI = Frame Buffer Interface

8.2.2 Programmable Real-Time Graphics

Stages in graphics pipelines do many floating-point operations on completely independent data, e.g.: transforming the positions of triangle vertices, and generating pixel colors. This *data independence* as the dominating characteristic is the key difference between the design assumption for GPUs and CPUs. A single frame, rendered in 1/60-th of a second, might have a million triangles and 6 million pixels.

Vertex shader programs map the positions of triangle vertices onto the screen, altering their position, color, or orientation. A vertex shader thread reads a vertex position (x, y, z, w) and computes its position on screen. Geometry shader programs operate on primitives defined by multiple vertices, changing them or generating additional primitives. Vertex shader programs and geometry shader programs execute on the vertex shader (VS/T & L) stage of the graphics pipeline.

A shader program calculates the floating-point red, green, blue, alpha (RGBA) color contribution to the rendered image at its pixel sample image position. The programmable vertex processor executes programs designated to the vertex shader stage. The programmable fragment processor executes programs designated to the (pixel) shader stage. For all graphics shader programs, instances can be run in parallel, because each works on independent data, produces independent results, and has no side effects. This property has motivated the design of the programmable pipeline stages into massively parallel processors.

An example of a programmable pipeline is illustrated by a schematic of a vertex processor in a pipeline, in Fig. 8.8 and by a schematic of a fragment processor in a pipeline, in Fig. 8.9.

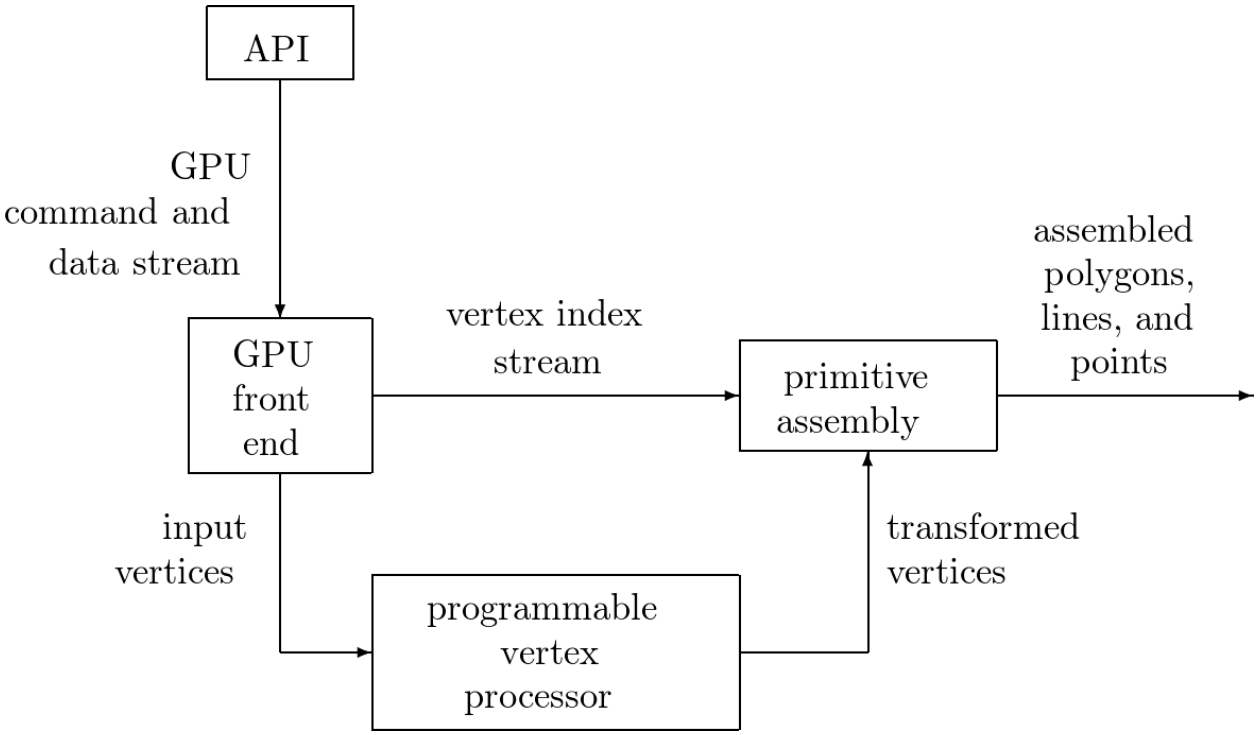


Fig. 8.8: A vertex processor in a pipeline.

Between the programmable graphics pipeline stages are dozens of fixed-function stages that perform well-defined tasks far more efficiently than a programmable processor could and which would benefit far less from programmability. For example, between the vertex processing stage and the pixel (fragment) processing stage is a *rasterizer*. The rasterizer — it does rasterization and interpolation — is a complex state machine that determines exactly which pixels (and portions thereof) lie within each geometric primitive's boundaries. The mix of programmable and fixed-function stages is engineered to balance performance with user control over the rendering algorithm.

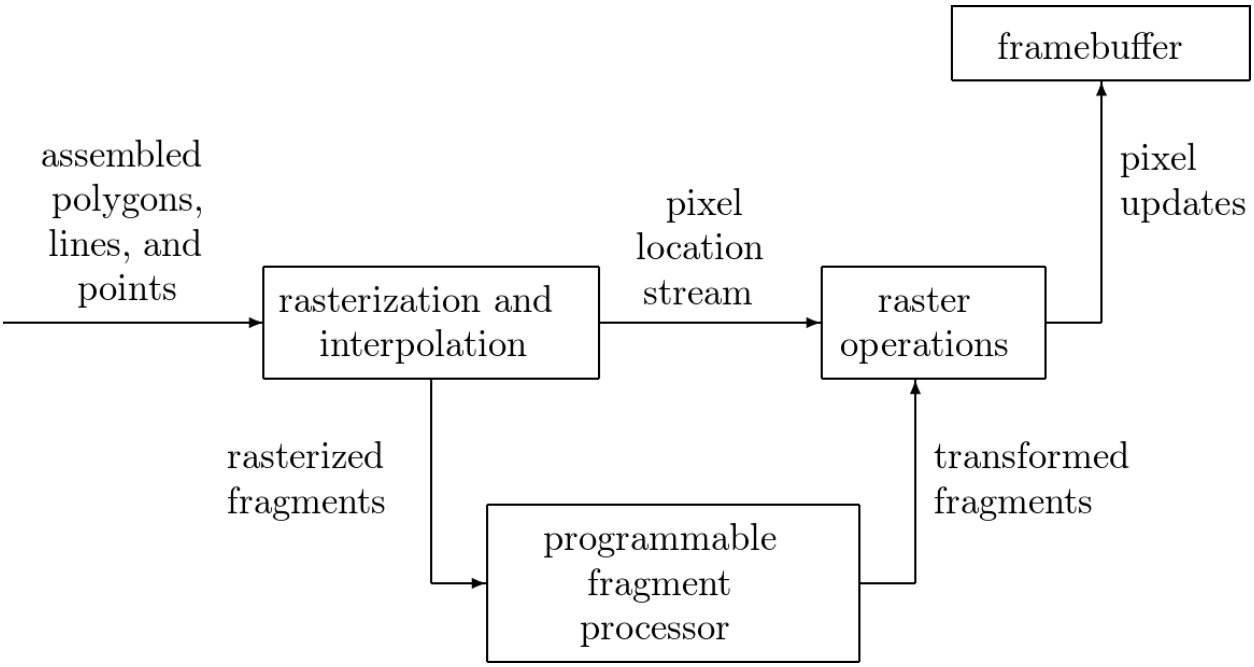


Fig. 8.9: A fragment processor in a pipeline.

Introduced in 2006, the GeForce 8800 GPU mapped the separate programmable graphics stages to an array of unified processors. The graphics pipeline is physically a recirculating path that visits the processors three times, with much fixed-function tasks in between. More sophisticated shading algorithms motivated a sharp increase in the available shader operation rate, in floating-point operations. High-clock-speed design made programmable GPU processor array ready for general numeric computing. Original GPGPU programming used APIs (DirectX or OpenGL): to a GPU everything is a pixel.

The drawbacks of the GPGPU model are many:

1. The programmer must know APIs and GPU architecture well.
2. Programs expressed in terms of vertex coordinates, textures, shader programs, add to the complexity.
3. Random reads and writes to memory are not supported.
4. No double precision is limiting for scientific applications.

Programming GPUs with CUDA (C extension): GPU computing.

Chapter 2 in the textbook ends mentioning the GT200. The next generation is code-named Fermi:

- 32 CUDA cores per streaming multiprocessor,
- 8 $\times$ peak double precision floating point performance over GT200,
- true cache hierarchy, more shared memory,
- faster context switching, faster atomic operations.

We end this section with a number of figures about the GPU architecture, in Fig. 8.12, Fig. 8.13, and Fig. 8.14.

The architecture of the NVIDIA Kepler GPU is summarized and illustrated in Fig. 8.15 and Fig. 8.16.

Summarizing this section, to fully utilize the GPU, one must use thousands of threads, whereas running thousands of threads on multicore CPU will swamp it. Data independence is the key difference between GPU and CPU.

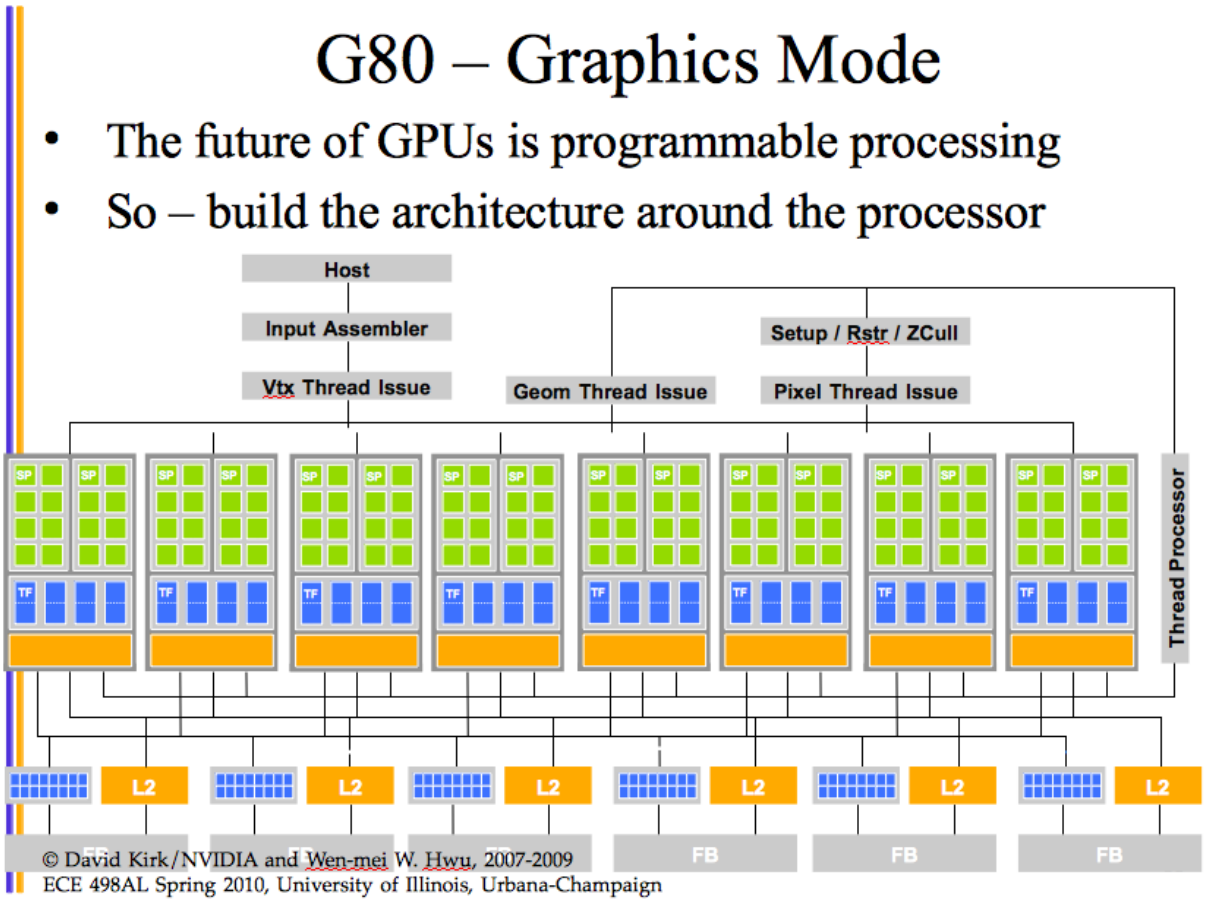


Fig. 8.10: GeForce 8800 GPU for GPGPU programming.

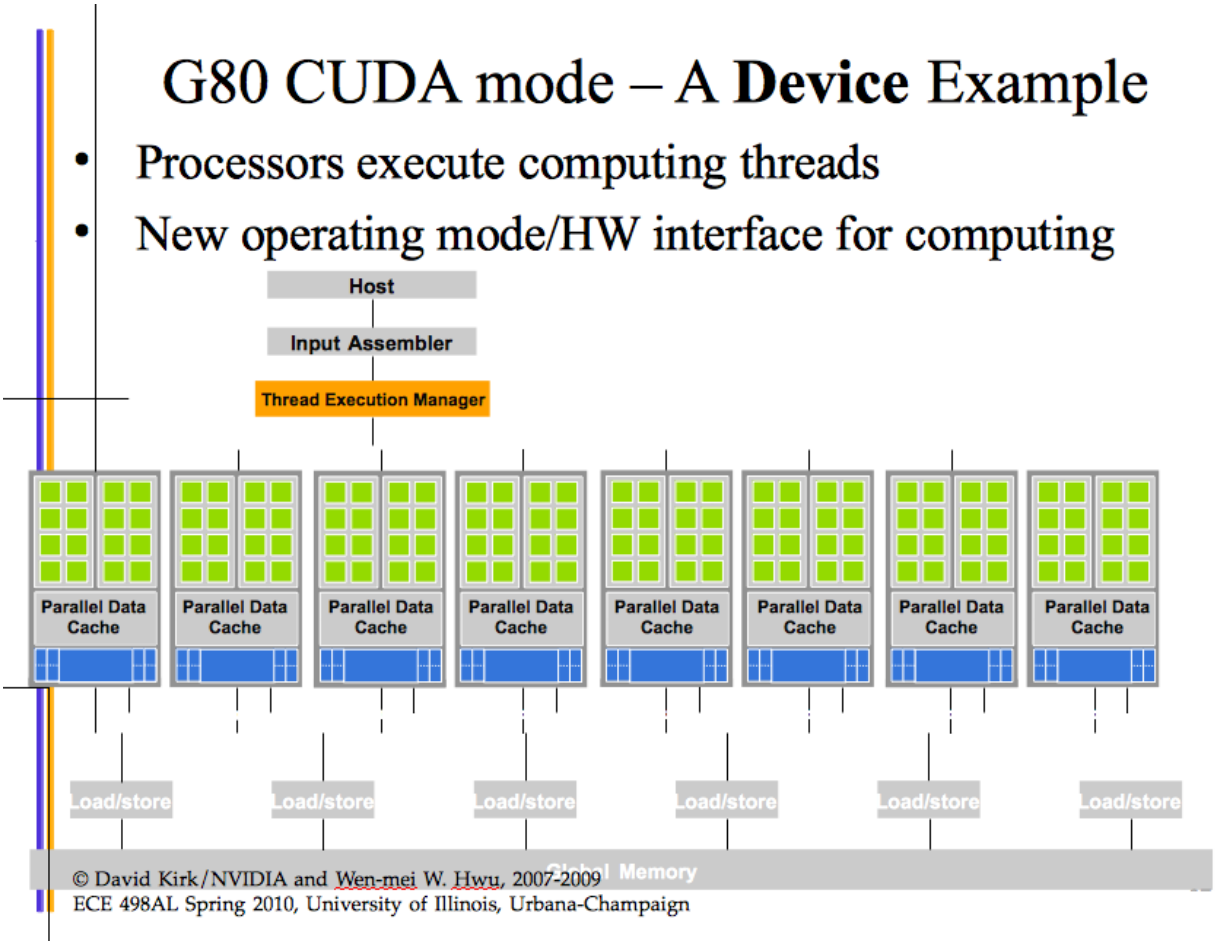


Fig. 8.11: GeForce 8800 GPU with new interface.

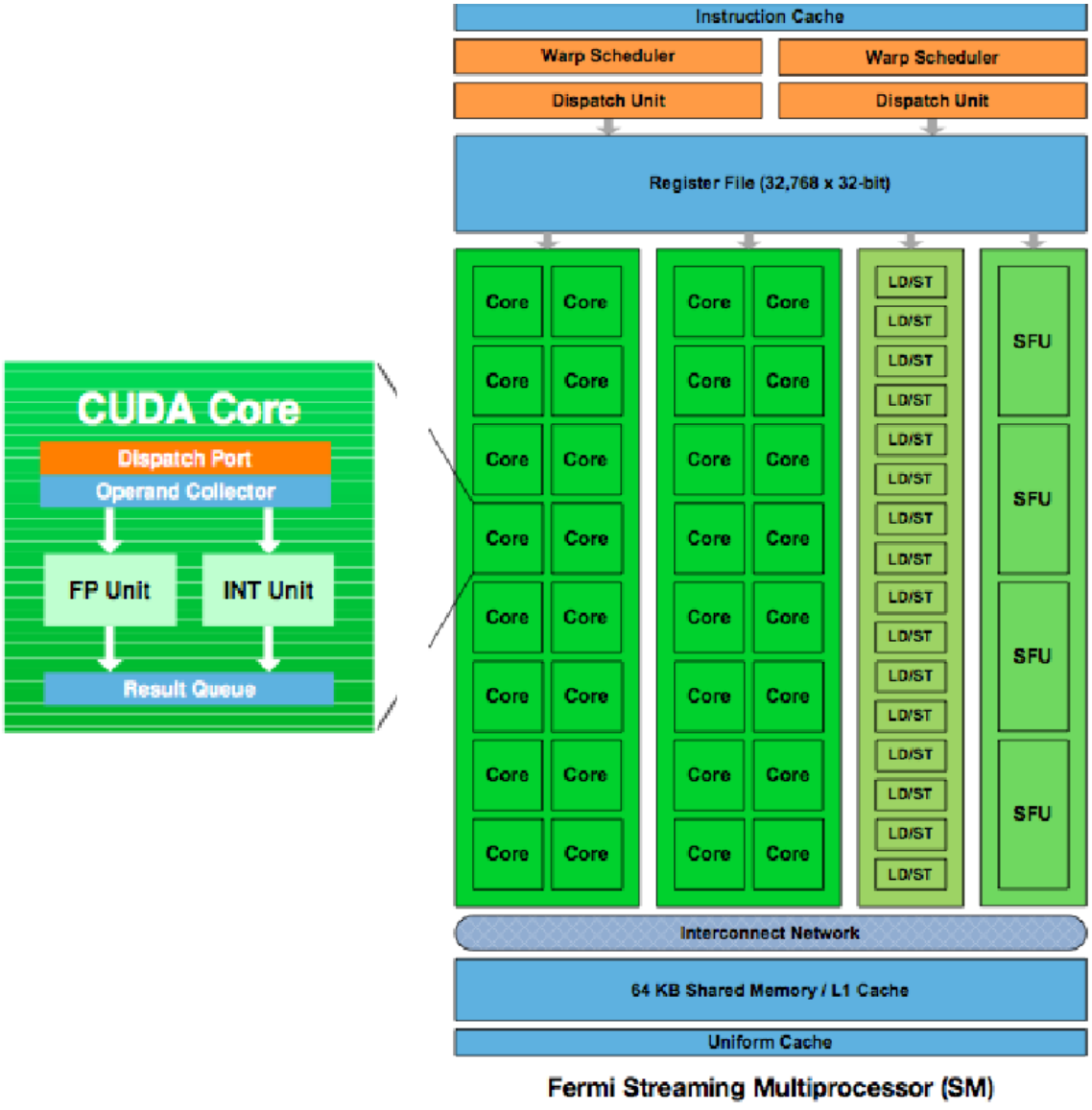


Fig. 8.12: The 3rd generation Fermi Architecture.

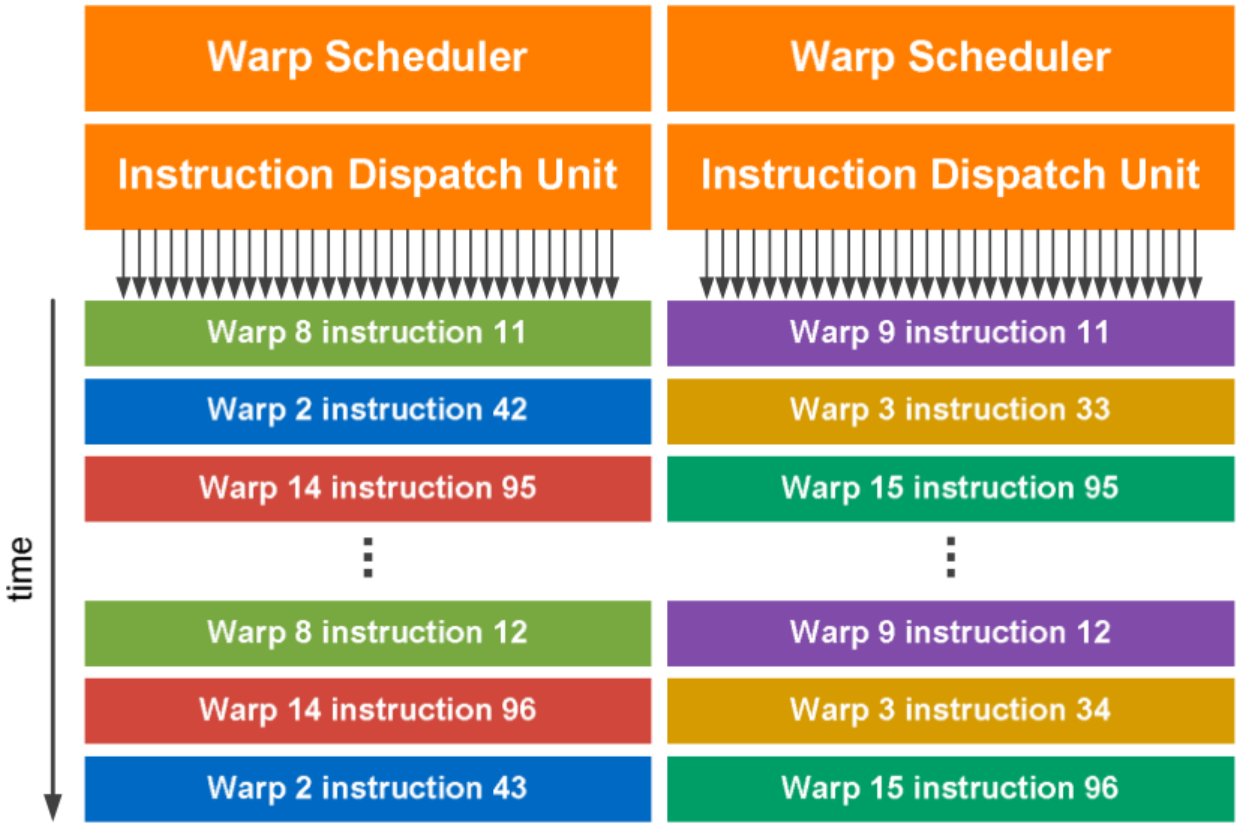


Fig. 8.13: Dual warp scheduler.

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

Fig. 8.14: Summary table of the GPU architecture.

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

Compute Capability of Fermi and Kepler GPUs

Fig. 8.15: The Kepler architecture.

8.2.3 Bibliography

Available at <http://www.nvidia.com>:

1. NVIDIA. **Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi.**
2. NVIDIA. **Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110.**

8.3 Programming GPUs with Python: PyOpenCL and PyCUDA

High level GPU programming can be done in Python, either with PyOpenCL or PyCUDA.

8.3.1 PyOpenCL

OpenCL, the Open Computing Language, is the open standard for parallel programming of heterogeneous system. OpenCL is maintained by the Khronos Group, a not for profit industry consortium creating open standards for the authoring and acceleration of parallel computing, graphics, dynamic media, computer vision and sensor processing on a wide variety of platforms and devices, with home page at <http://www.khronos.org>.

Another related standard is OpenGL <http://www.opengl.org>, the open standard for high performance graphics.

The development of OpenCL was initiated by Apple. Many aspects of OpenCL are familiar to a CUDA programmer because of similarities with data parallelism and complex memory hierarchies. OpenCL offers a more complex platform and device management model to reflect its support for multiplatform and multivendor portability. OpenCL

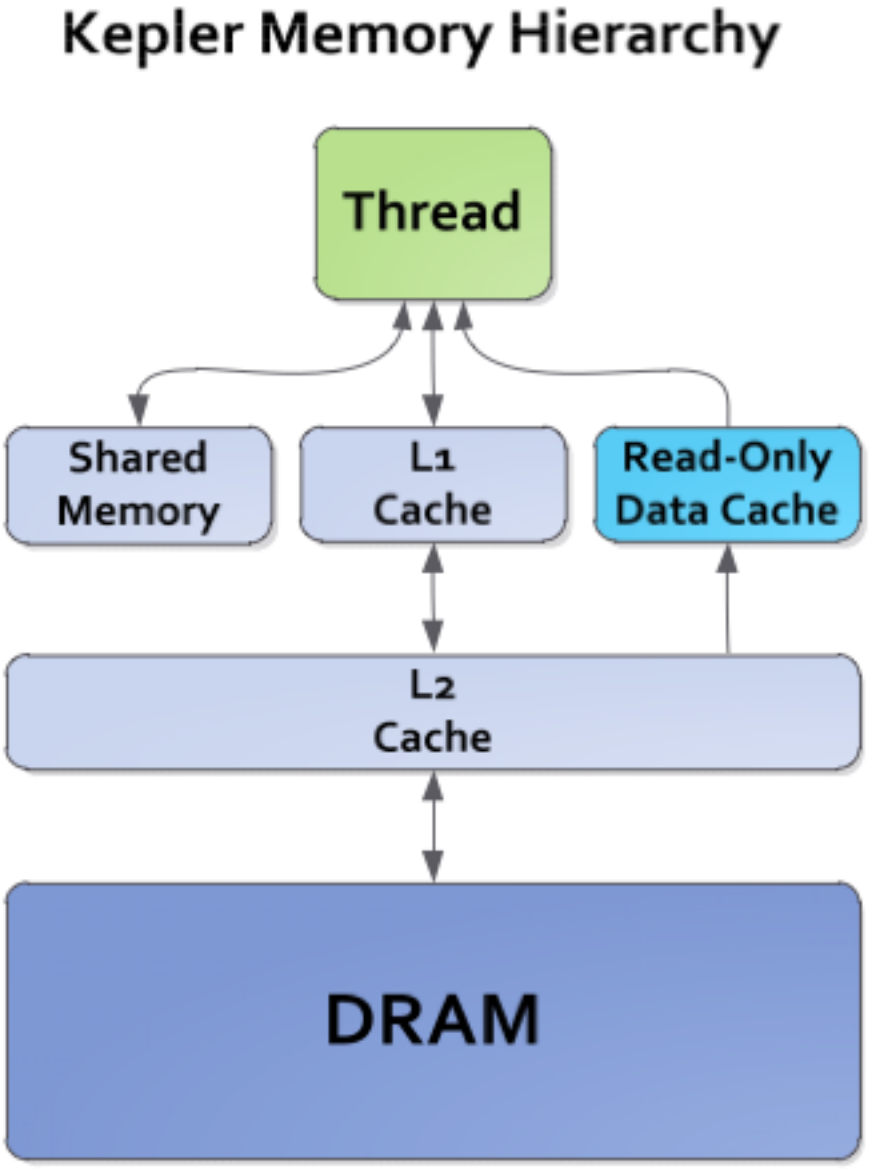


Fig. 8.16: Memory hierarchies of Kepler.

implementations exist for AMD ATI and NVIDIA GPUs as well as x86 CPUs. The code in this lecture runs on an Intel Iris Graphics 6100, the graphics card of a MacBook Pro.

We enjoy the same benefits of PyOpenCL as of PyCUDA:

- takes care of a lot of boiler plate code;
- focus on the kernel, with numpy typing.

Instead of a programming model tied to a single hardware vendor's products, open standards enable portable software frameworks for heterogeneous platforms.

Installing can be done simplity with `pip`, via `sudo pip install pyopencl`. To check the installation in an interactive session:

```
>>> import pyopencl
>>> from pyopencl.tools import get_test_platforms_and_devices
>>> get_test_platforms_and_devices()
[(<pyopencl.Platform 'Apple' at 0x7fff0000>, \
[<pyopencl.Device 'Intel(R) Core(TM) i7-5557U CPU @ 3.10GHz' \
on 'Apple' at 0xffffffff>, \
<pyopencl.Device 'Intel(R) Iris(TM) Graphics 6100'
on 'Apple' at 0x1024500>])]
>>>
```

The multiplication of two matrices is an important application.

```
$ python matmatmulocl.py
matrix A:
[[ 0.  0.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
matrix B:
[[ 1.  1.  0.  1.  1.]
 [ 1.  1.  1.  0.  1.]
 [ 0.  0.  1.  0.  1.]
 [ 1.  0.  1.  0.  1.]]
multiplied A*B:
[[ 1.  0.  2.  0.  2.]
 [ 3.  2.  3.  1.  4.]
 [ 3.  2.  3.  1.  4.]]
$
```

Code for the script `matmatmulocl.py` is below.

```
import pyopencl as cl
import numpy as np

import os
os.environ['PYOPENCL_COMPILER_OUTPUT'] = '1'
os.environ['PYOPENCL_CTX'] = '1'

(n, m, p) = (3, 4, 5)

# a = np.random.randn(n, m).astype(np.float32)
# b = np.random.randn(m, p).astype(np.float32)
a = np.random.randint(2, size=(n*m))
b = np.random.randint(2, size=(m*p))
c = np.zeros((n*p), dtype=np.float32)
```

```
a = a.astype(np.float32)
b = b.astype(np.float32)
```

The setup of the context, queue, and buffers happens by the code below:

```
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_buf = cl.Buffer\
   (ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer\
   (ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
c_buf = cl.Buffer(ctx, mf.WRITE_ONLY, c.nbytes)
```

The kernel is defined next.

```
prg = cl.Program(ctx, """
    __kernel void multiply(ushort n,
    ushort m, ushort p, __global float *a,
    __global float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = 0.0f;
      int rowC = gid/p;
      int colC = gid%p;
      __global float *pA = &a[rowC*m];
      __global float *pB = &b[colC];
      for(int k=0; k<m; k++)
      {
         pB = &b[colC+k*p];
         c[gid] += (*(pA++))*(*pB);
      }
    }
    """).build()
```

Below is the contents of the main function to execute the OpenCL code.

```
prg.multiply(queue, c.shape, None,
             np.uint16(n), np.uint16(m), np.uint16(p),
             a_buf, b_buf, c_buf)

a_mul_b = np.empty_like(c)
cl.enqueue_copy(queue, a_mul_b, c_buf)

print "matrix A:"
print a.reshape(n, m)
print "matrix B:"
print b.reshape(m, p)
print "multiplied A*B:"
print a_mul_b.reshape(n, p)
```

The NVIDIA OpenCL SDK contains also a matrix-matrix multiplication. It execution shows the following.

```
$ python matmatmulsdk.py
GPU push+compute+pull total [s]: 0.0844735622406
GPU push [s]: 0.000111818313599
GPU pull [s]: 0.0014328956604
```

```
GPU compute (host-timed) [s]: 0.0829288482666
GPU compute (event-timed) [s]:  0.08261928

GFlops/s: 24.6958693242

GPU==CPU: True

CPU time (s) 0.0495228767395

GPU speedup (with transfer):  0.586252969875
GPU speedup (without transfer):  0.59717309205
$
```

8.3.2 PyCUDA

Code for the GPU can be generated in Python, see Fig. 8.17, 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.

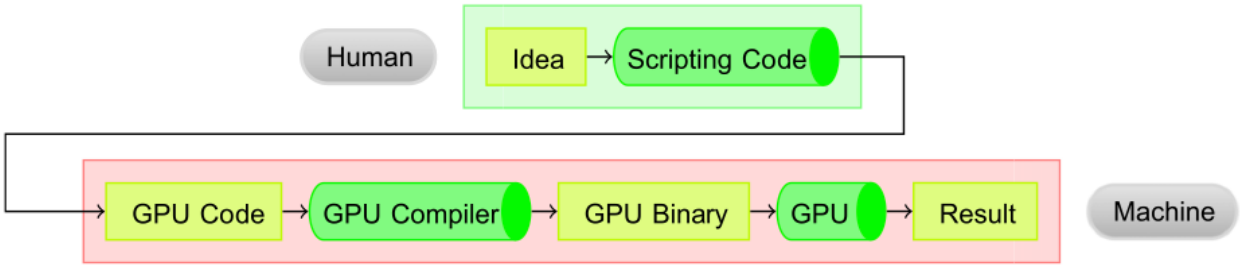


Fig. 8.17: The operating principle of GPU code generation.

To verify whether PyCUDA is correctly installed on our computer, we can run an interactive Python session as follows.

```
>>> import pycuda
>>> import pycuda.autoinit
>>> 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.

```
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.

```
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:

```
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];
    }
    """)
```

The launching of the kernel and printing the result is the last stage.

```
func = mod.get_function("multiply")
func(n, m, p, a_gpu, b_gpu, c_gpu, \
     block=(numpy.int(n), numpy.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
```

8.3.3 Data Parallelism Model

In this section we construct dictionaries between OpenCL and CUDA, summarized in Table 8.1.

After launching a kernel, OpenCL code is executed by *work items*. Work items form *work groups*, which correspond to CUDA blocks. An index space defines how data are mapped to work items.

Table 8.1: Dictionary between OpenCL and CUDA

OpenCL concept	CUDA equivalent
kernel	kernel
host program	host program
NDRange (index space)	grid
work group	block
work item	thread

Like CUDA, OpenCL exposes a hierarchy of memory types. The mapping of OpenCL memory types to CUDA is in Table 8.2.

Table 8.2: Memory types in OpenCL and CUDA

OpenCL memory type	CUDA equivalent
global memory	global memory
constant memory	constant memory
local memory	shared memory
private memory	local memory

Local memory in OpenCL and shared memory in CUDA are accessible respectively to a work group and thread block. Private memory in OpenCL and local memory in CUDA is memory accessible only to individual threads.

Threads in CUDA and work items in OpenCL have their own unique global index values. A dictionary between dimensions and indices is in Table 8.3.

Table 8.3: Dimensions and indices in OpenCL and CUDA

OpenCL API call	CUDA equivalent
get_global_id(0)	blockIdx.x $\times$ blockDim.x + threadIdx.x
get_local_id(0)}	threadIdx.x
get_global_size(0)	gridDim.x $\times$ blockDim.x
get_local_size(0)}	blockDim.x

Replacing `0` in `get_global_id(0)` by `1` and `2` gives the values for the y and z dimensions respectively.

The OpenCL parallel execution model is introduced in Fig. 8.18 and in Fig. 8.19, both copied from *NVIDIA: OpenCL and other interesting stuff* by Chris Lamb.

The structure of an OpenCL program is sketched in Fig. 8.20.

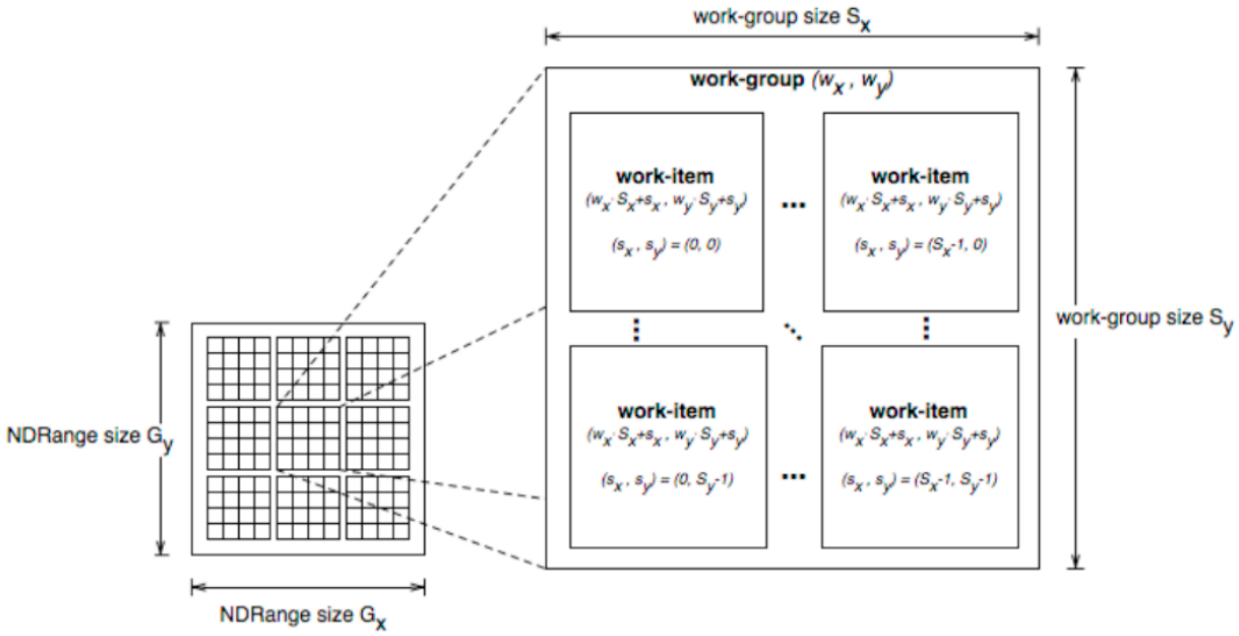


Fig. 8.18: Overview of the OpenCL parallel execution model

8.3.4 Writing OpenCL Programs

A simple program uses OpenCL to compute the square for a buffer of floating point values. Compiling and running goes as follows.

```
gcc -o /tmp/hello hello_opencl.c -framework OpenCL

/tmp/hello
Computed '1024/1024' correct values!
```

The kernel is listed below.

```
const char *KernelSource = "\n" \
"__kernel void square(                       \n" \
"   __global float* input,                   \n" \
"   __global float* output,                  \n" \
"   const unsigned int count)                \n" \
"{                                           \n" \
"   int i = get_global_id(0);                \n" \
"   if(i < count)                            \n" \
"       output[i] = input[i] * input[i];      \n" \
"}                                           \n" \
"\n";
```

The program starts as follows:

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <unistd.h>
```

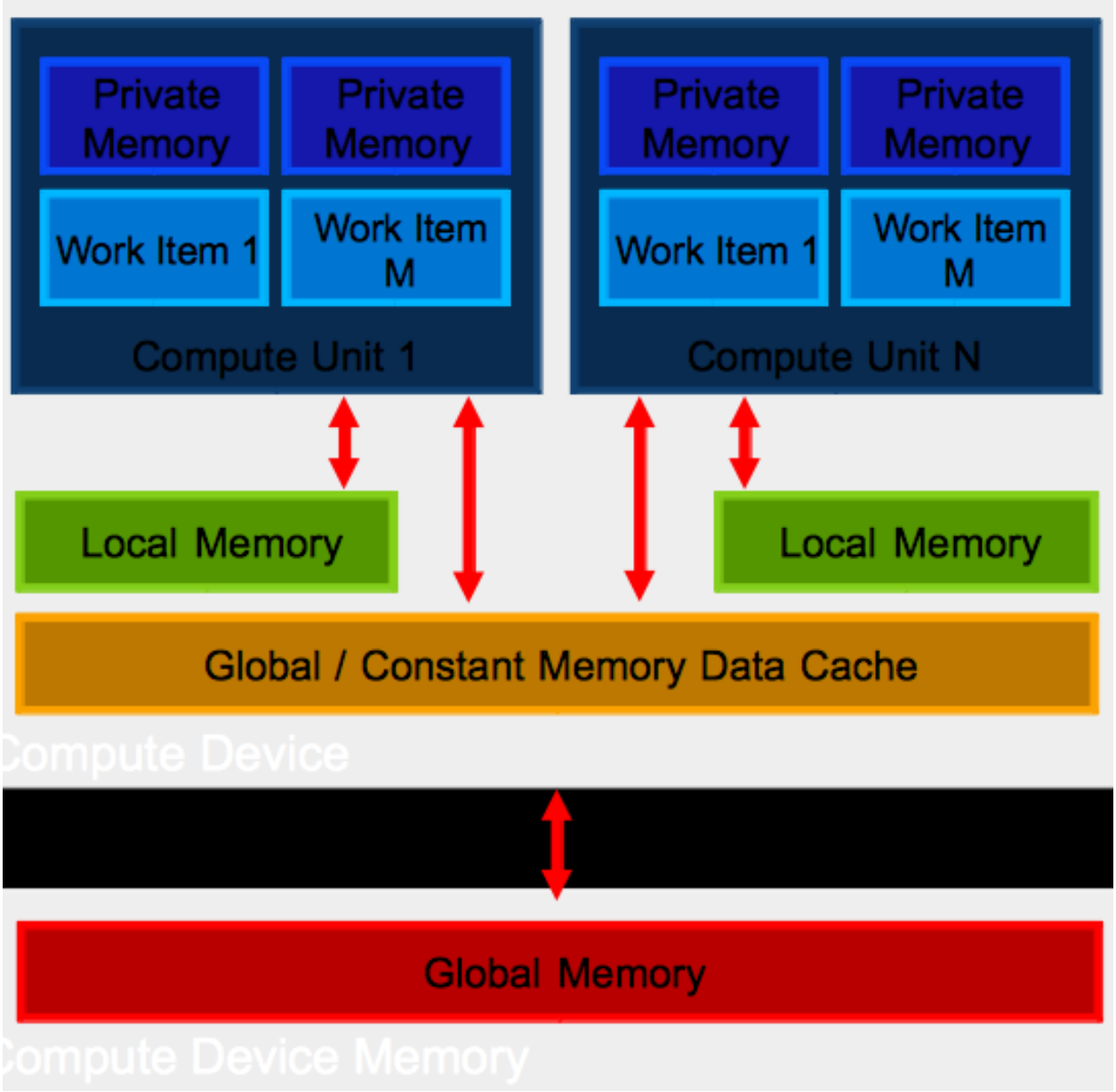


Fig. 8.19: The OpenCL device architecture

• host program + query platform + query compute devices + create contexts	platform layer
+ create memory objects associate the contexts + compile and create kernel program objects + issue commands to command queue + synchronization of commands + clean up OpenCL resources	runtime
• kernels + OpenCL C code	language

Fig. 8.20: Basic OpenCL program structure

```
#include <sys/types.h>
#include <sys/stat.h>
#include <OpenCL/opencl.h>

int main(int argc, char** argv)
{
    int err;           // error code returned from api calls
    float data[DATA_SIZE];  // original data given to device
    float results[DATA_SIZE];// results returned from device
    unsigned int correct;       // number of correct results
    size_t global; // global domain size for our calculation
    size_t local;   // local domain size for our calculation
```

Then we declare the OpenCL types.

```
cl_device_id device_id;      // compute device id
cl_context context;          // compute context
cl_command_queue commands;   // compute command queue
cl_program program;          // compute program
cl_kernel kernel;            // compute kernel
cl_mem input;   // device memory used for the input
cl_mem output;  // device memory used for the output

// Fill our data set with random float values

int i = 0;
unsigned int count = DATA_SIZE;
for(i = 0; i < count; i++)
    data[i] = rand() / (float)RAND_MAX;
```

To connect to the device and create the context, we use the code below.

```
// Connect to a compute device

int gpu = 1;
err = clGetDeviceIDs(NULL, gpu ?
   CL_DEVICE_TYPE_GPU : CL_DEVICE_TYPE_CPU,
```

```
   1, &device_id, NULL);
if(err != CL_SUCCESS)
{
   printf("Error: Failed to create a device group!\n");
   return EXIT_FAILURE;
}

// Create a compute context

context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
if (!context)
{
   printf("Error: Failed to create a compute context!\n");
   return EXIT_FAILURE;
}
```

The creation of a command and the program happens by the code below.

```
// Create a command commands

commands = clCreateCommandQueue(context, device_id, 0, &err);
if (!commands)
{
   printf("Error: Failed to create a command commands!\n");
   return EXIT_FAILURE;
}

// Create the compute program from the source buffer

program = clCreateProgramWithSource(context, 1,
   (const char **) & KernelSource, NULL, &err);
if (!program)
{
   printf("Error: Failed to create compute program!\n");
   return EXIT_FAILURE;
}
```

Then we build the executable.

```
// Build the program executable

err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS)
{
   size_t len;
   char buffer[2048];

   printf("Error: Failed to build program executable!\n");
   clGetProgramBuildInfo(program, device_id,
      CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
   printf("%s\n", buffer);
   exit(1);
}
```

Then we create the kernel and define the input and output data.

```
// Create the compute kernel in the program we wish to run

```

```
kernel = clCreateKernel(program, "square", &err);
if (!kernel || err != CL_SUCCESS)
{
   printf("Error: Failed to create compute kernel!\n");
   exit(1);
}

// Create the input and output arrays in device memory

input = clCreateBuffer(context,  CL_MEM_READ_ONLY,
   sizeof(float) * count, NULL, NULL);
output = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
   sizeof(float) * count, NULL, NULL);
if (!input || !output)
{
   printf("Error: Failed to allocate device memory!\n");
   exit(1);
}
```

The next stage is the writing of the data and kernel arguments.

```
// Write our data set into the input array in device memory

err = clEnqueueWriteBuffer(commands, input, CL_TRUE,
   0, sizeof(float) * count, data, 0, NULL, NULL);
if (err != CL_SUCCESS)
{
   printf("Error: Failed to write to source array!\n");
   exit(1);
}

// Set the arguments to our compute kernel

err = 0;
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
err |= clSetKernelArg(kernel, 2, sizeof(unsigned int), &count);
if (err != CL_SUCCESS)
{
    printf("Error: Failed to set kernel arguments! %d\n", err);
    exit(1);
}
```

We configure the execution with the code below:

```
// Get the maximum work group size for executing the kernel

err = clGetKernelWorkGroupInfo(kernel, device_id,
    CL_KERNEL_WORK_GROUP_SIZE, sizeof(local), &local, NULL);
if (err != CL_SUCCESS)
{
    printf("Error: Failed to retrieve kernel work group info! %d\n", err);
    exit(1);
}

// Execute the kernel over the entire range of
// our 1d input data set // using the maximum number
// of work group items for this device
```

```
global = count;
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL,
    &global, &local, 0, NULL, NULL);
if (err)
{
    printf("Error: Failed to execute kernel!\n");
    return EXIT_FAILURE;
}
```

At last, we finish and read the results.

```
// Wait for the command commands to get serviced
// before reading back results

clFinish(commands);

// Read back the results from the device for verification

err = clEnqueueReadBuffer( commands, output, CL_TRUE,
    0, sizeof(float) * count, results, 0, NULL, NULL );
if (err != CL_SUCCESS)
{
    printf("Error: Failed to read output array! %d\n", err);
    exit(1);
}
```

Before the cleanup, the results are validated.

```
    correct = 0;
    for(i = 0; i < count; i++)
    {
        if(results[i] == data[i] * data[i])
            correct++;
    }

    printf("Computed '%d/%d' correct values!\n",
        correct, count);

    clReleaseMemObject(input);
    clReleaseMemObject(output);
    clReleaseProgram(program);
    clReleaseKernel(kernel);
    clReleaseCommandQueue(commands);
    clReleaseContext(context);

    return 0;
}
```

8.3.5 Bibliography

1. B.R. Gaster, L. Howes, D.R. Kaeli, P. Mistry, D. Schaa. *Heterogeneous Computing with OpenCL*. Revised OpenCL 1.2 Edition. Elsevier 2013.
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. Chapter 14 of the second edition of the book of Kirk and Hwu.

8.3.6 Programming Guides

1. Apple Developer: OpenCL Programming Guide for Mac OS X. 2009-06-10, 55 pages.
2. NVIDIA: OpenCL Programming Guide for the CUDA Architecture. Version 4.1 1/3/2012, 63 pages.
3. AMD Accelerated Parallel Processing OpenCL Programming Guide. December 2011, 232 pages.

CHAPTER 9

Acceleration with CUDA

9.1 Introduction to CUDA

9.1.1 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.

To examine the CUDA Compute Capability, we check the card with `deviceQuery`.

```
$ /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 3 CUDA Capable device(s)

Device 0: "Tesla K20c"
  CUDA Driver Version / Runtime Version          6.0 / 5.5
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 4800 MBytes (5032706048 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Clock rate:                                706 MHz (0.71 GHz)
  Memory Clock rate:                             2600 Mhz
  Memory Bus Width:                              320-bit
  L2 Cache Size:                                 1310720 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536),␣
↪3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 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 Bus ID / PCI location ID:           4 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device␣
↪simultaneously) >
```

Another standard check is the `bandwidthTest`, which runs as below.

```
$ /usr/local/cuda/samples/1_Utilities/bandwidthTest/bandwidthTest

[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: Tesla K20c
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)     Bandwidth(MB/s)
   33554432                  5819.5

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)     Bandwidth(MB/s)
   33554432                  6415.8

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)     Bandwidth(MB/s)
   33554432                  143248.0

Result = PASS
```

9.1.2 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`.

We will now examine every step in greater detail.

In the first step, we add CUDA extensions to functions. We distinguish between three keywords before a function declaration:

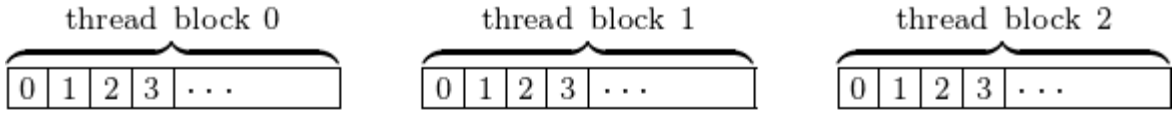
- `__host__`: The function will run on the host (CPU).
- `__device__`: The function will run on the device (GPU).
- `__global__`: The function is called from the host but runs on the device. This function is called a *kernel*.

The CUDA extensions to C function declarations are summarized in Table 9.1.

Table 9.1: 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. 9.1.




```
int threadId = blockIdx.x *
   blockDim.x + threadIdx.x
...
float x = input[threadID]
float y = f(x)
output[threadID] = y
...
```

Fig. 9.1: Defining the data for each thread.

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);
```

```
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;
      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;
}
```

The scalable programming model of the GPU is illustrated in Fig. 9.2.

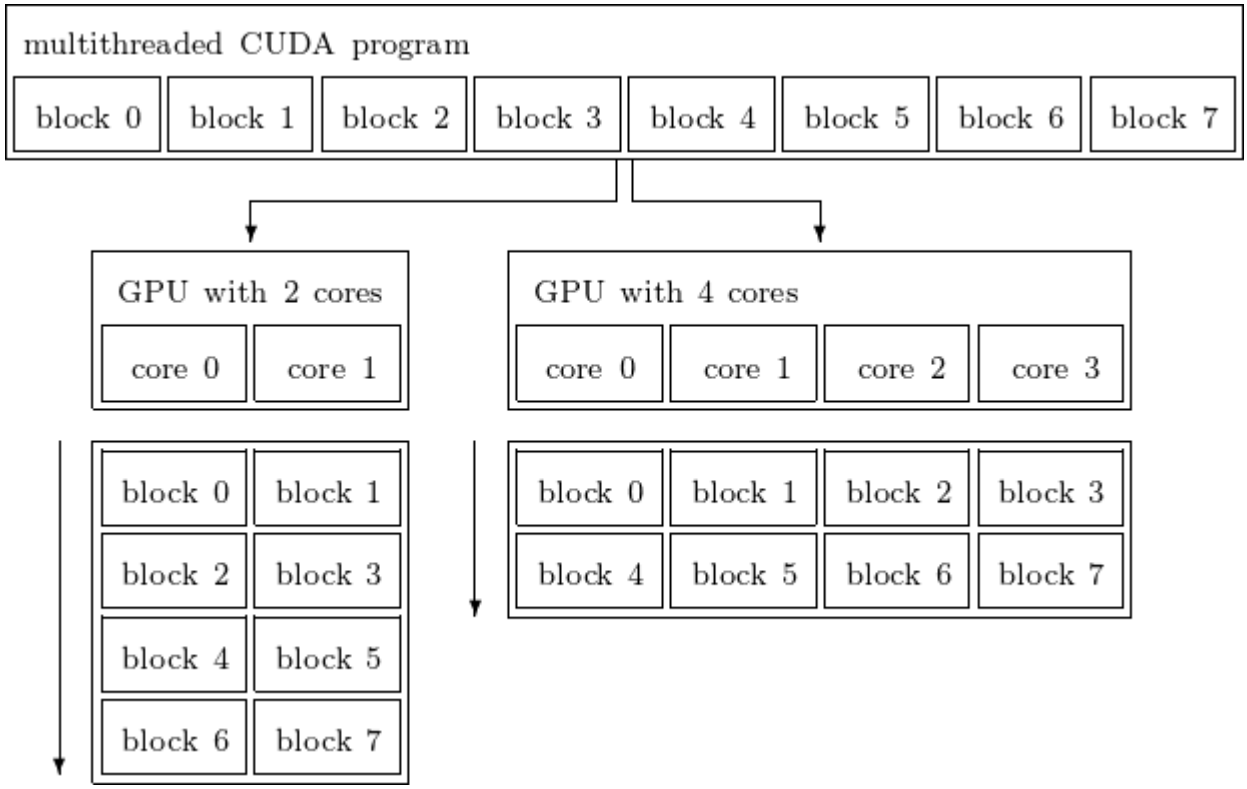


Fig. 9.2: A scalable programming model.

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
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.

```
$ time /tmp/run_cmpsqrt 128000 32 100000 0

real    0m17.345s
user    0m9.829s
sys     0m7.303s

$ time /tmp/run_cmpsqrt 128000 64 100000 0

real    0m10.502s
user    0m5.711s
sys     0m4.497s

$ time /tmp/run_cmpsqrt 128000 128 100000 0

real    0m9.295s
user    0m5.231s
sys     0m3.865s
```

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`.

9.1.3 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?

9.2 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.

9.2.1 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 $n = m = p = 1,000$ we have 1,000,000 inner products. The matrix multiplication is illustrated in Fig. 9.3.

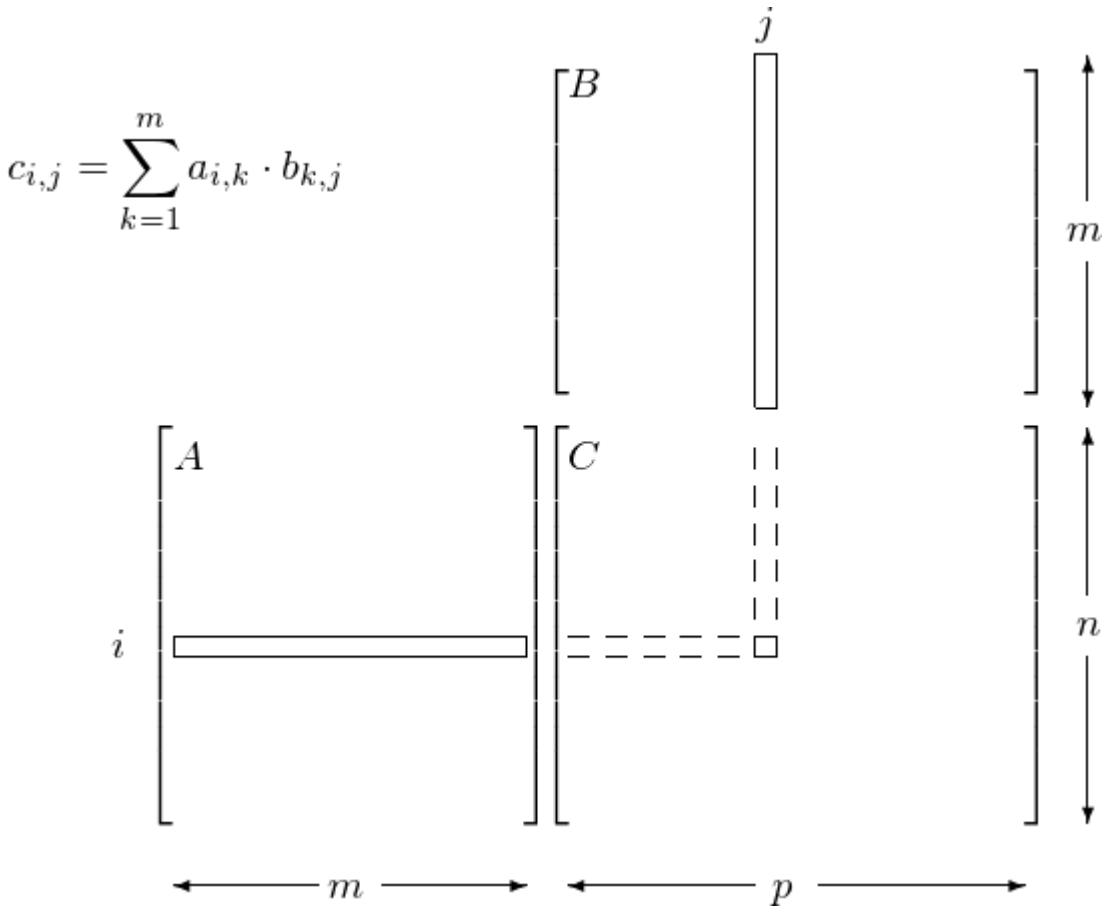


Fig. 9.3: Data parallelism in matrix multiplication.

9.2.2 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. 9.4.

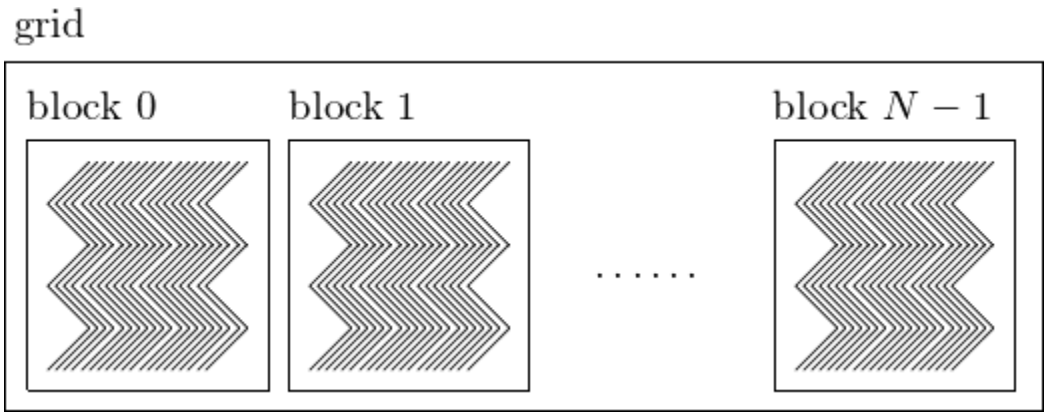


Fig. 9.4: The organization of the threads of execution in a CUDA program.

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. 9.5.

Code to generate a random matrix is below:

```
#include <stdlib.h>

__host__ void randomMatrix ( int n, int m, float *x, int mode )
/*
 * 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;
```

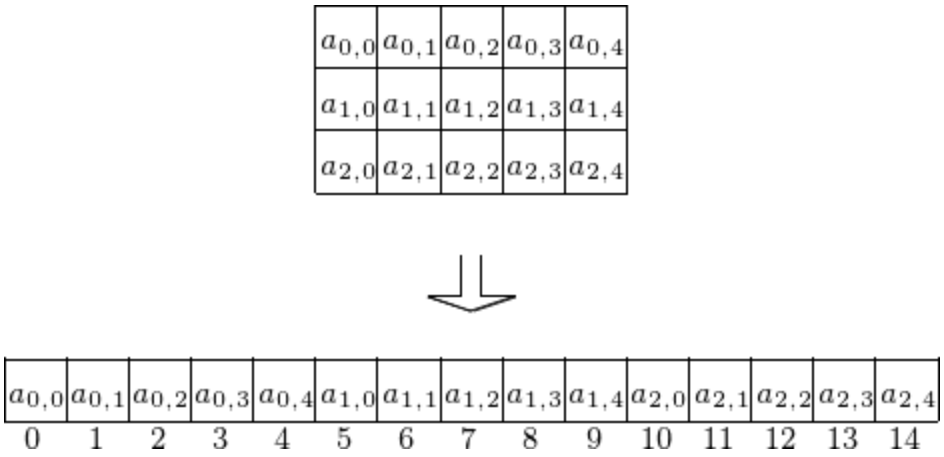


Fig. 9.5: Storing a matrix as a one dimensional array.

```
         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. 9.6.

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. */
{
```

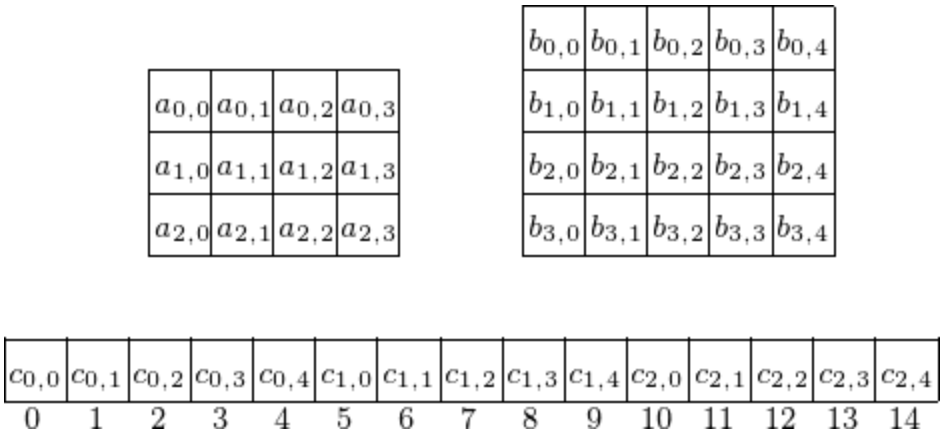


Fig. 9.6: Assigning inner products to threads.

```
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   C[i] = 0.0;
   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);
```

```
}
```

9.2.3 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);
   }
}
```

9.2.4 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 -e`
- with the events following the `-e` flag we count the floating-point operations.

For the Intel Sandy Bridge in `kepler` the codes are

- 530110 : `FP_COMP_OPS_EXE:X87`
- 531010 : `FP_COMP_OPS_EXE:SSE_FP_PACKED_DOUBLE`
- 532010 : `FP_COMP_OPS_EXE:SSE_FP_SCALAR_SINGLE`

- 534010 : `FP_COMP_OPS_EXE:SSE_PACKED_SINGLE`
- 538010 : `FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE`

Executables are compiled with the option `-O2`.

The performance of one CPU core is measured in the session below.

```
$ perf stat -e r530110 -e r531010 -e r532010 -e r534010 \
            -e r538010 ./matmatmul0 745 745 745 0

 Performance counter stats for './matmatmul0 745 745 745 0':

         1,668,710 r530110                                       [79.95%]
                 0 r531010                                       [79.98%]
     2,478,340,803 r532010                                       [80.04%]
                 0 r534010                                       [80.05%]
                 0 r538010                                       [80.01%]

       1.033291591 seconds time elapsed
```

Did 2,480,009,513 operations in 1.033 seconds: $\Rightarrow (2,480,009,513/1.033)/(2^{30}) = 2.23$GFlops.

The output of a session to measure the performance on the GPU, running `matmatmul1` is below:

```
$ perf stat -e r530110 -e r531010 -e r532010 -e r534010 \
            -e r538010 ./matmatmul1 745 745 745 0

 Performance counter stats for './matmatmul1 745 745 745 0':

           160,925 r530110                                       [79.24%]
                 0 r531010                                       [80.62%]
         2,306,222 r532010                                       [80.07%]
                 0 r534010                                       [79.53%]
                 0 r538010                                       [80.71%]

       0.663709965 seconds time elapsed
```

The wall clock time is measured with the `time` command, as below:

```
$ time ./matmatmul1 745 745 745 0

real    0m0.631s
user    0m0.023s
sys     0m0.462s
```

The dimension 745 is too small for the GPU to be able to improve much. Let us thus increase the dimension.

```
$ perf stat -e r530110 -e r531010 -e r532010 -e r534010 \
            -e r538010 ./matmatmul0 4000 4000 4000 0

 Performance counter stats for './matmatmul0 4000 4000 4000 0':

        48,035,278 r530110                                     [80.03%]
                 0 r531010                                     [79.99%]
   267,810,771,301 r532010                                     [79.99%]
                 0 r534010                                     [79.99%]
                 0 r538010                                     [80.00%]

     171.334443720 seconds time elapsed
```

Now we will see if GPU acceleration leads to a speedup... The outcome accelerated performance measured is shown next:

```
$ perf stat -e r530110 -e r531010 -e r532010 -e r534010
            -e r538010 ./matmatmul1 4000 4000 4000 0

 Performance counter stats for './matmatmul1 4000 4000 4000 0':

           207,682 r530110                                       [80.27%]
                 0 r531010                                       [79.96%]
        64,222,441 r532010                                       [79.95%]
                 0 r534010                                       [79.94%]
                 0 r538010                                       [79.98%]

       1.011284551 seconds time elapsed
```

The speedup is 181.334/1.011 = 169, which is significant. Counting flops, f = 267,810,771,301

- $t_{\text{cpu}} = 171.334$ and $f/t_{\text{cpu}}/(2^{30}) = 1.5$ GFlops.
- $t_{\text{gpu}} = 1.011$ and $f/t_{\text{gpu}}/(2^{30}) = 246.7$ GFlops.

9.2.5 Exercises

1. Modify `matmatmul0.c` and `matmatmul1.cu` to work with doubles instead of floats. Examine the performance.
2. Modify `matmatmul2.cu` to use double indexing of matrices, e.g.: `C[i][j] += A[i][k]*B[k][j]`.
3. Compare the performance of `matmatmul1.cu` and `matmatmul2.cu`, taking larger and larger values for n, m, and p. Which version scales best?

9.3 Device Memories and Matrix Multiplication

We take a closer look at different memories on the GPU and how it relates to the problem of matrix multiplication.

9.3.1 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. 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. 9.7.

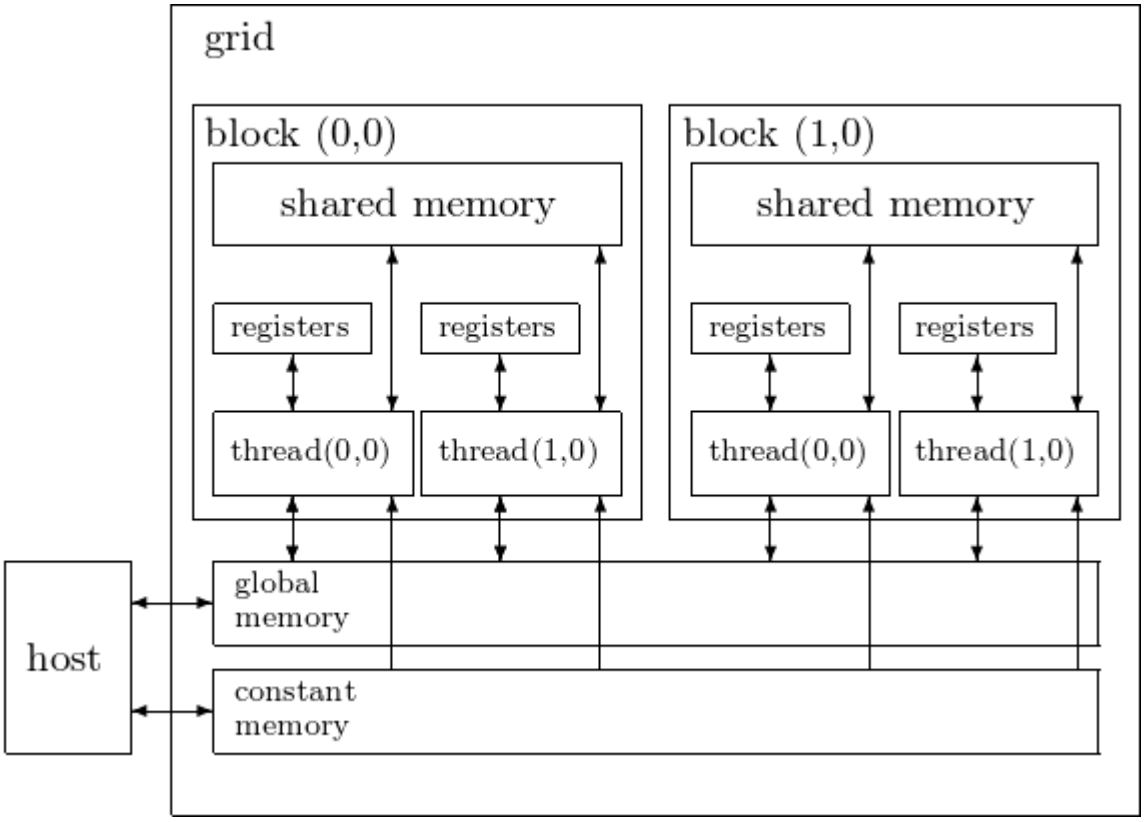


Fig. 9.7: The CUDA device memory types.

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 Tesla K20C and the P100.

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 Tesla K20c and the P100.

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 Tesla K20c has 65,536 bytes of constant memory, the total amount of global memory is 4,800 MBytes with 1,310,720 bytes of L2 cache.
- The Tesla P100 has 65,536 bytes of constant memory, the total amount of global memory is 16,276 MBytes with 4,194,304 bytes of L2 cache.

The relationshiop between thread organization and different types of device memories is shown in Fig. 9.8, 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.

We distinguish between five different variable declarations, based on their memory location, scope, and lifetime, summarized in Table 9.2.

Table 9.2: 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.

9.3.2 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 $w \times w$:

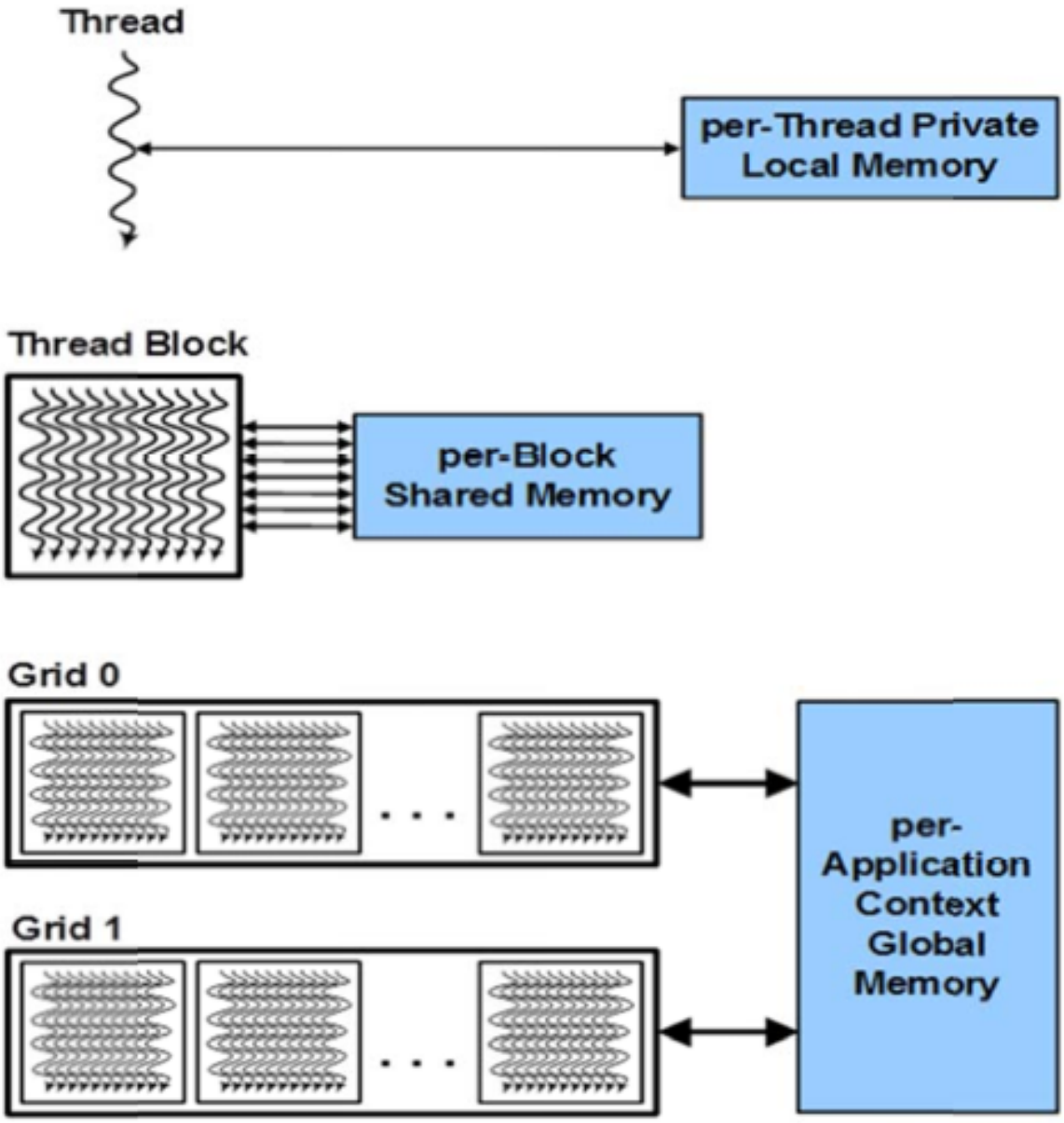


Fig. 9.8: Registers, shared, and global memory per thread, thread block, and grid.

- 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. 9.9.

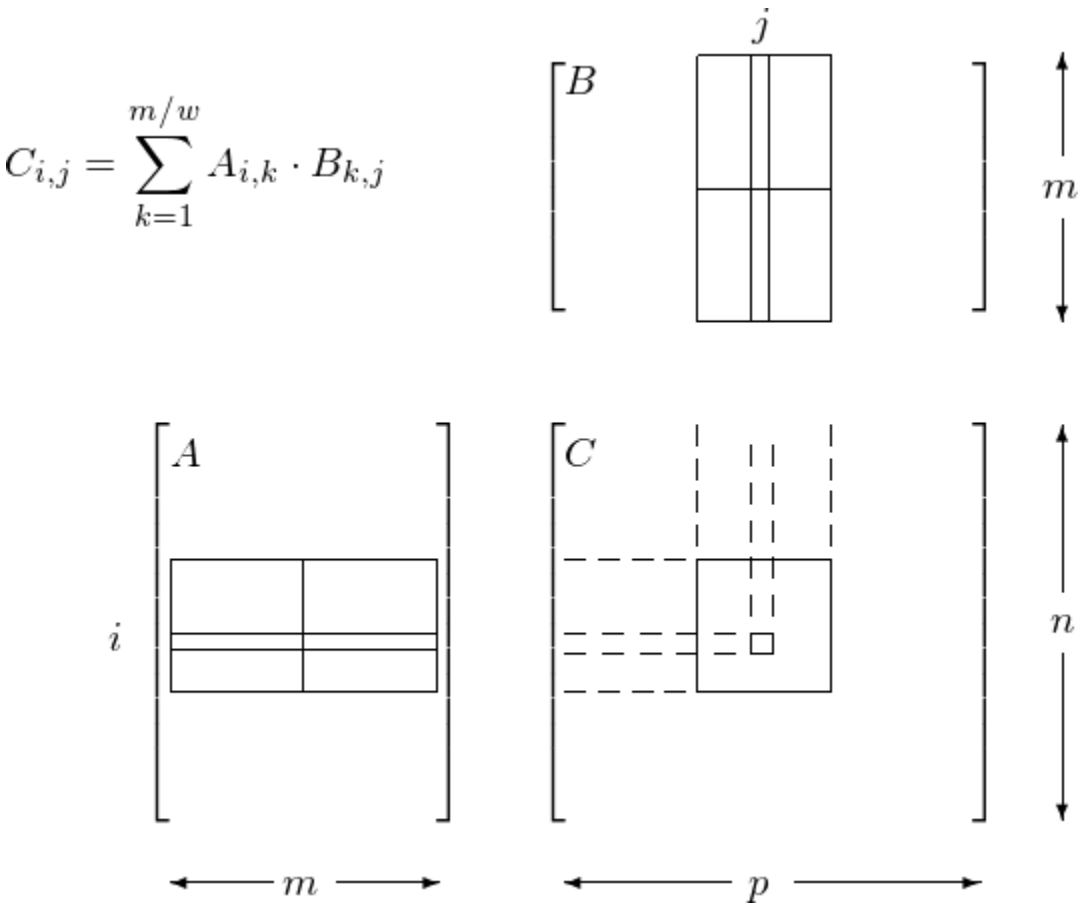


Fig. 9.9: Tiled matrix multiplication and shared memory.

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 Tesla K20C.

A session on the GeForce 9400M is below:

```
/Developer/GPU Computing/C/bin/darwin/release $ ./matrixMul
[matrixMul] starting...

[ matrixMul ]
./matrixMul
     Starting (CUDA and CUBLAS tests)...
```

```
Device 0: "GeForce 9400M" with Compute 1.1 capability

Using Matrix Sizes: A(160 x 320), B(160 x 320), C(160 x 320)

Runing Kernels...

> CUBLAS         7.2791 GFlop/s, Time = 0.00225 s, Size = 16384000 Ops

> CUDA matrixMul 5.4918 GFlop/s, Time = 0.00298 s, Size = 16384000 Ops,\
NumDevsUsed = 1, Workgroup = 256

Comparing GPU results with Host computation...

Comparing CUBLAS & Host results
CUBLAS compares OK

Comparing CUDA matrixMul & Host results
CUDA matrixMul compares OK

[matrixMul] test results...
PASSED
```

A session on the Tesla C2050/C2070 is below:

```
/usr/local/cuda/sdk/C/bin/linux/release jan$ ./matrixMul
[matrixMul] starting...
[ matrixMul ]
./matrixMul Starting (CUDA and CUBLAS tests)...

Device 0: "Tesla C2050 / C2070" with Compute 2.0 capability

Using Matrix Sizes: A(640 x 960), B(640 x 640), C(640 x 960)

Runing Kernels...

> CUBLAS         Throughput = 424.8840 GFlop/s, Time = 0.00185 s, \
Size = 786432000 Ops

> CUDA matrixMul Throughput = 186.7684 GFlop/s, Time = 0.00421 s, \
Size = 786432000 Ops, NumDevsUsed = 1, Workgroup = 1024

Comparing GPU results with Host computation...

Comparing CUBLAS & Host results
CUBLAS compares OK

Comparing CUDA matrixMul & Host results
CUDA matrixMul compares OK

[matrixMul] test results...
PASSED
```

A session on the K20C is below:

```
$ /usr/local/cuda/samples/0_Simple/matrixMul/matrixMul
[Matrix Multiply Using CUDA] - Starting...

GPU Device 0: "Tesla K20c" with compute capability 3.5
```

```
MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 246.13 GFlop/s, Time= 0.533 msec, Size= 131072000 Ops, \
WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS

Note: For peak performance, please refer to the matrixMulCUBLAS \
example.
$
```

The theoretical peak performance of the K20c is 1.17 TFlops double precision, and 3.52 TFlops single precision. The matrices that are multiplied have single float as type.

With CUBLAS we can go for peak performance:

```
$ /usr/local/cuda/samples/0_Simple/matrixMulCUBLAS/matrixMulCUBLAS
[Matrix Multiply CUBLAS] - Starting...
/usr/bin/nvidia-modprobe: unrecognized option: "-u"

GPU Device 0: "Tesla K20c" with compute capability 3.5

MatrixA(320,640), MatrixB(320,640), MatrixC(320,640)
Computing result using CUBLAS...done.
Performance= 1171.83 GFlop/s, Time= 0.112 msec, Size= 131072000 Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS
$
```

The theoretical peak performance of the K20c is 1.17 TFlops double precision, and 3.52 TFlops single precision. The matrices that are multiplied have single float as type. The newer P100 has a theoretical peak performance (with GPU Boost) of 18.7 TFlops in half precision, 9.3 TFlops in single precision, and 4.7 TFlops in double precision. First we run the simple version of the matrix multiplication:

```
$ /usr/local/cuda/samples/0_Simple/matrixMul/matrixMul
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "Tesla P100-PCIE-16GB" with compute capability 6.0

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 1909.26 GFlop/s, Time= 0.069 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.
$
```

Then a run with CUBLAS on the P100:

```
$ /usr/local/cuda/samples/0_Simple/matrixMulCUBLAS/matrixMulCUBLAS
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla P100-PCIE-16GB" with compute capability 6.0

MatrixA(640,480), MatrixB(480,320), MatrixC(640,320)
Computing result using CUBLAS...done.
```

```
Performance= 3089.82 GFlop/s, Time= 0.064 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.
$
```

A second run gave the following:

```
Performance= 3106.43 GFlop/s, Time= 0.063 msec, Size= 196608000 Ops
```

For single floats, the theoretical peak performance is 9.3 TFlops.

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)
{
    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();
    }

    // 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.

9.3.3 Bibliography

- 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.

9.3.4 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.

9.4 Thread Organization and Matrix Multiplication

In this lecture we look at the problem of multiplying two matrices, from the perspective of the thread organization.

9.4.1 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 NVIDIA Tesla C2050 has 14 streaming multiprocessors and threads are executed in groups of 32 (the warp size). This implies: $14 \times 32 = 448$ threads can run simultaneously. For the K20C the numbers are respectively 13, 192, and 2496; and for the P100, we have respectively 56, 64, and 3584. A picture of the scalable programming model was shown in Fig. 9.2.

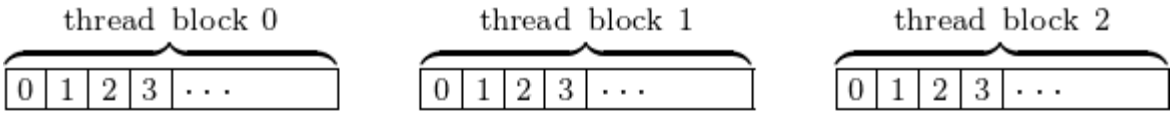
All threads execute the same code, defined by the kernel. The builtin variable `threadIdx`

- 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 `blockIdx.x` $\in \{0, N-1\}$. Within each block, `threadIdx.x` $\in \{0,$ `blockDim.x` $-1\}$. The organization of the data for each thread is in Fig. 9.10.




```
int threadId = blockIdx.x *
   blockDim.x + threadIdx.x
...
float x = input[threadID]
float y = f(x)
output[threadID] = y
...
```

Fig. 9.10: 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 Tesla C2050/C2070 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: $65,535 \times 65,535 \times 65,535$.

  65,535 is the upper limit for the builtin variables `gridDim.x`, `gridDim.y`, and `gridDim.z`.

On the K20C and the P100, the limitations are as follows:

- Maximum number of threads per block: 1,024.
- Maximum dimension size of a thread block: $1,024 \times 1,024 \times 64$.
- Maximum dimension size of a grid size: $2,147,483,647 \times 65,535 \times 65,535$.

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.

9.4.2 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. 9.11.

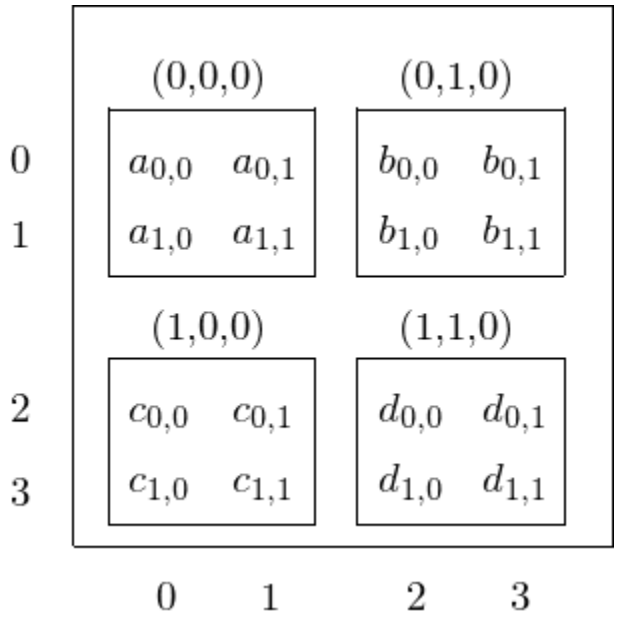


Fig. 9.11: Storing a tiled matrix in a grid.

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. 9.12.

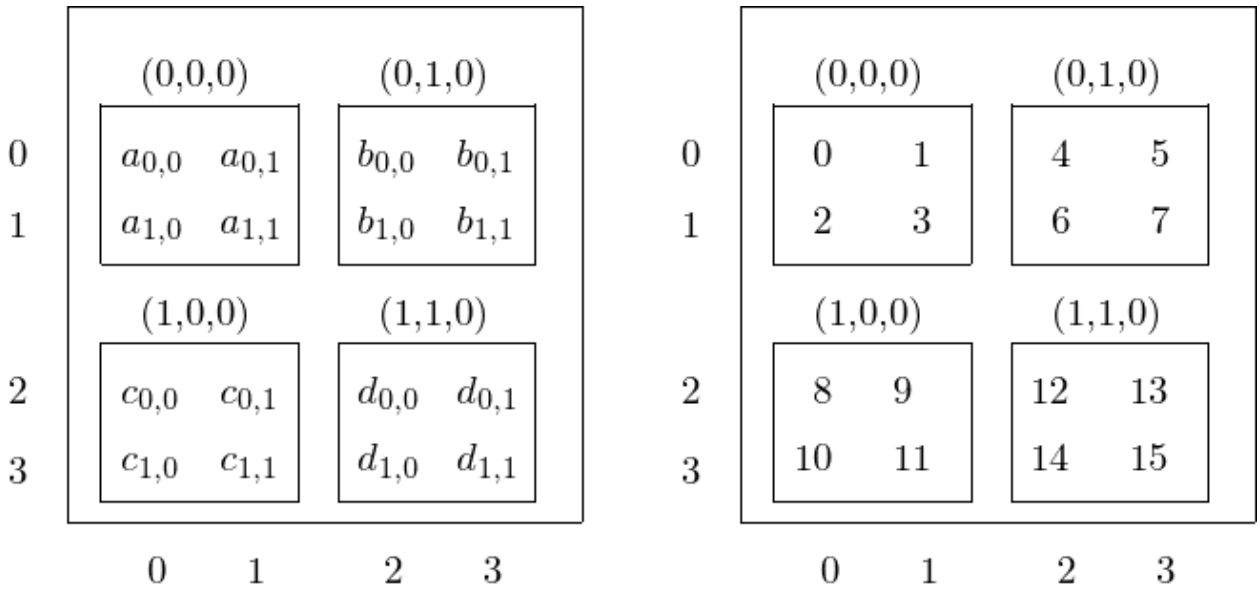


Fig. 9.12: Mapping threads to entries in the matrix.

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. 9.13.

The main function in the CUDA code to organized the threads is listed below.

```
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);
```

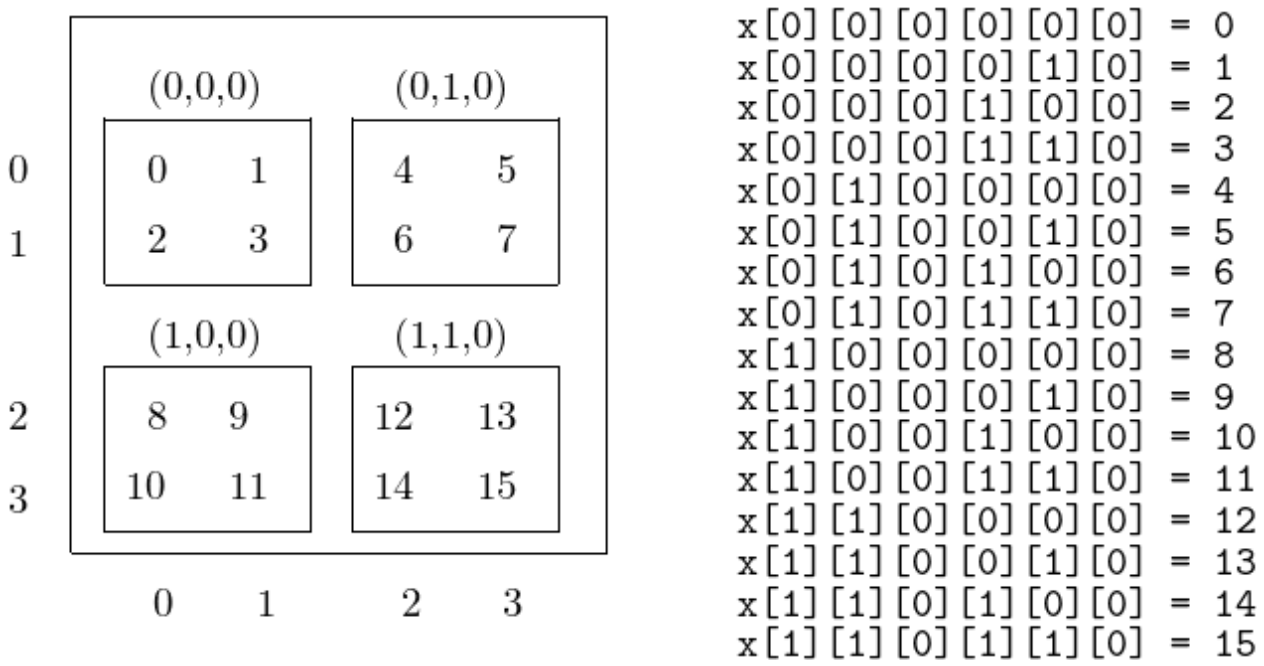


Fig. 9.13: Linear address calculation for threads and submatrices.

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);

   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;
}
```

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. 9.14.

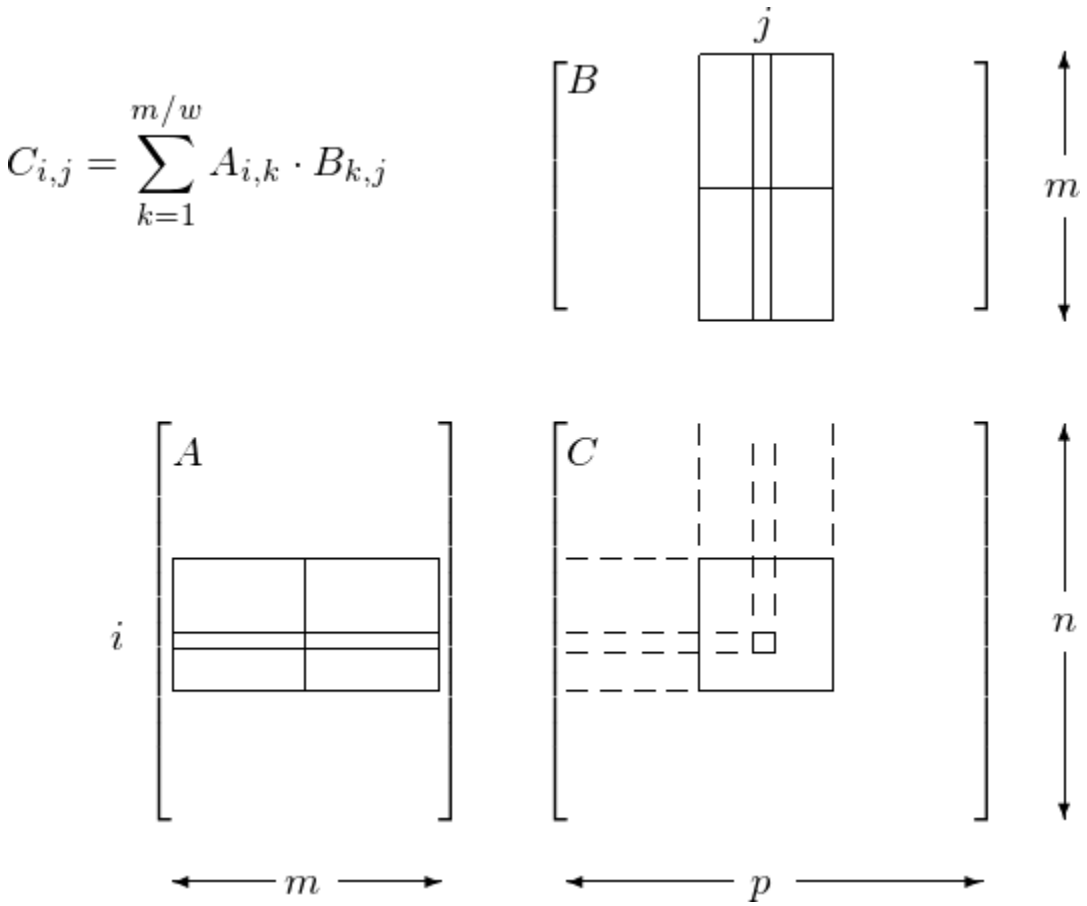


Fig. 9.14: Tiled matrix multiplication with shared memory.

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);

        // 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;
}
```

9.4.3 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.

9.4.4 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.

CHAPTER 10

CUDA Thread Organization

10.1 Warps and Reduction Algorithms

We discuss warp scheduling, latency hiding, SIMT, and thread divergence. To illustrate the concepts we discussed two reduction algorithms.

10.1.1 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*. On the Tesla C2050/C2070, K20C, and P100, each warp consists of 32 threads.

The scheduling of threads is represented schematically in Fig. 10.1.

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. 8.13, 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.

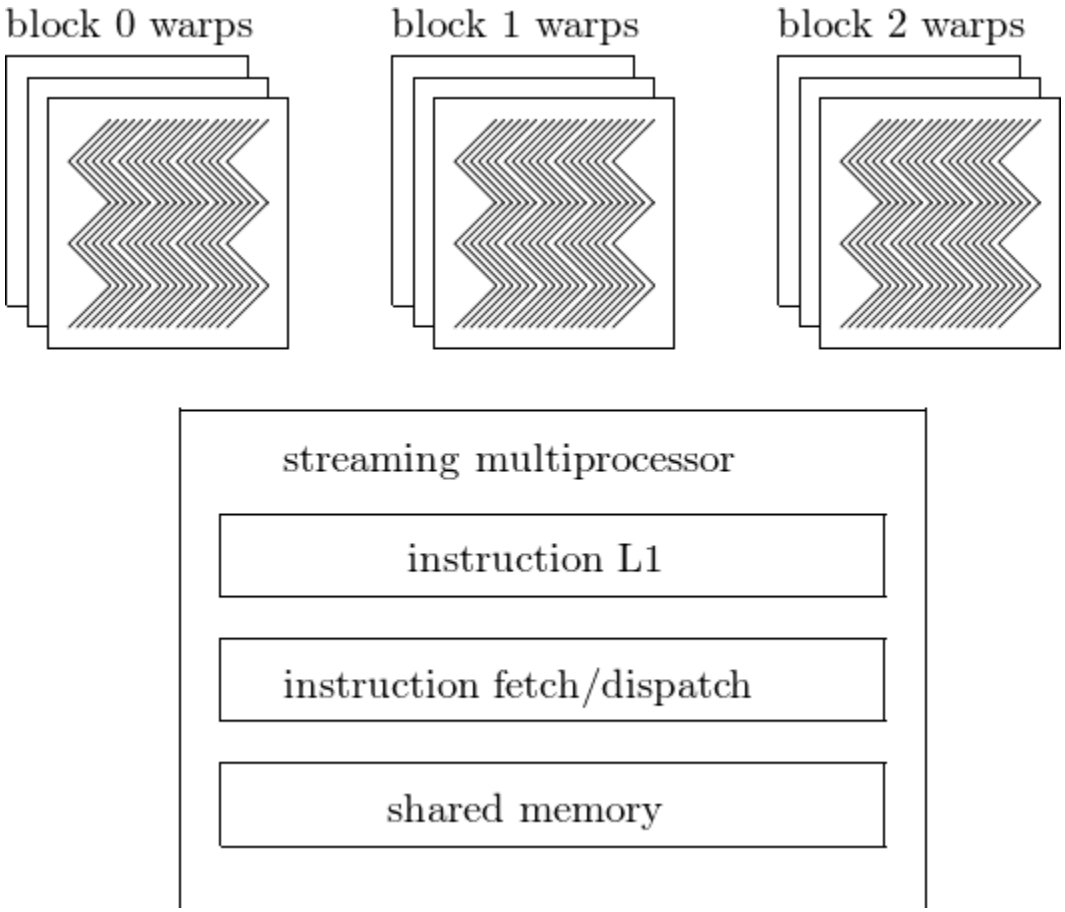


Fig. 10.1: Scheduling of threads by a streaming multiprocessor.

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.

single-instruction, multiple-thread, and divergence

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. 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.

10.1.2 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. 10.2. 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.

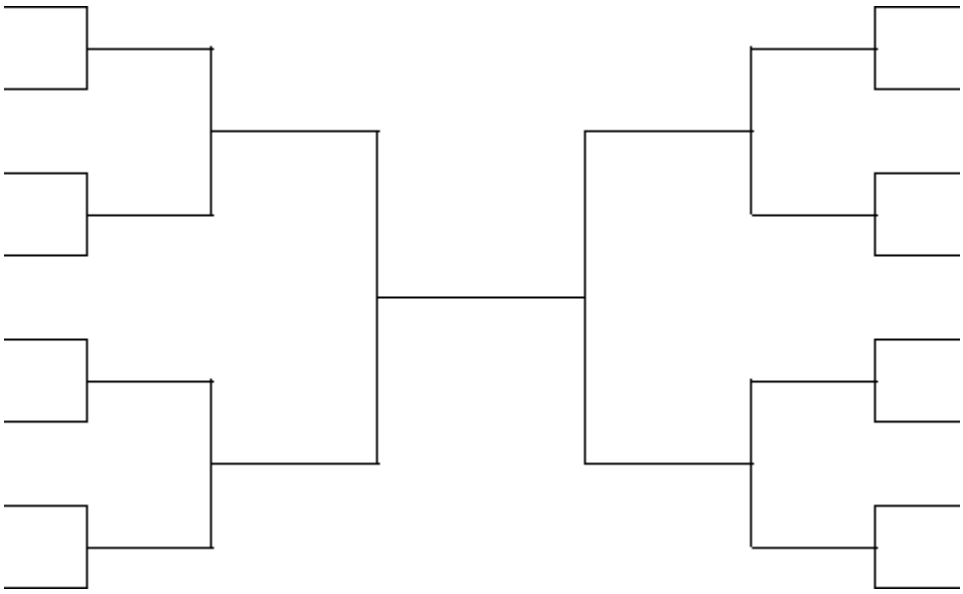


Fig. 10.2: A example of a reduction: a tournament.

As an introduction to a kernel for the parallel sum, consider the summation of 32 numbers, see Fig. 10.3.

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[];

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

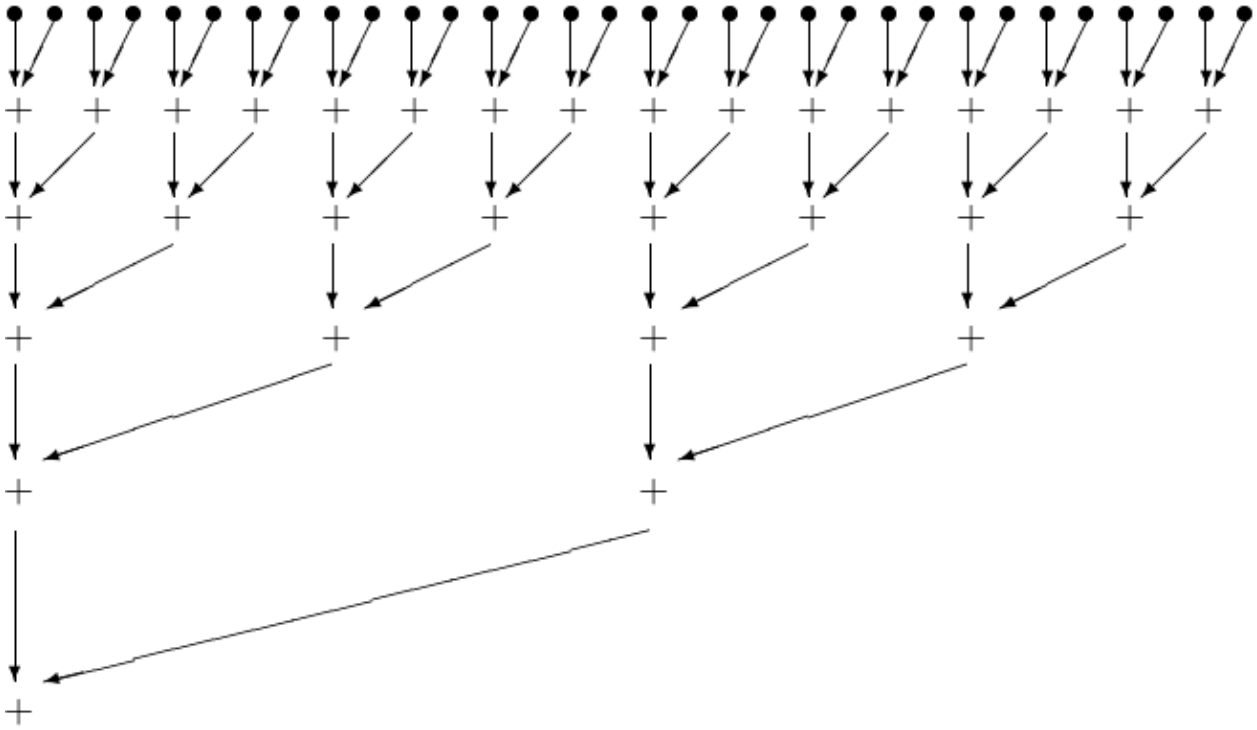


Fig. 10.3: Summing 32 numbers in a parallel reduction.

```
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. 10.4.

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;
    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.

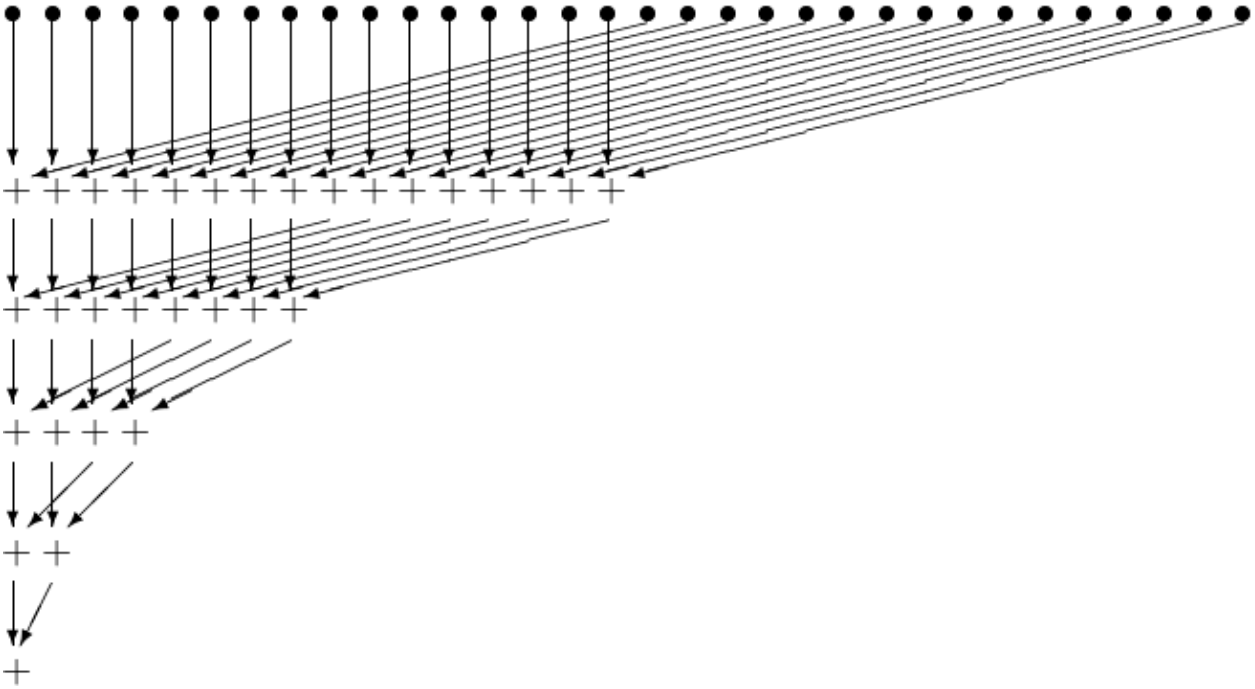


Fig. 10.4: The parallel summation of 32 number revised.

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.

10.1.3 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>.

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

10.2 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.

10.2.1 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.

If 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. 10.5, extracted from Figure G-1 of the NVIDIA Programming Guide.

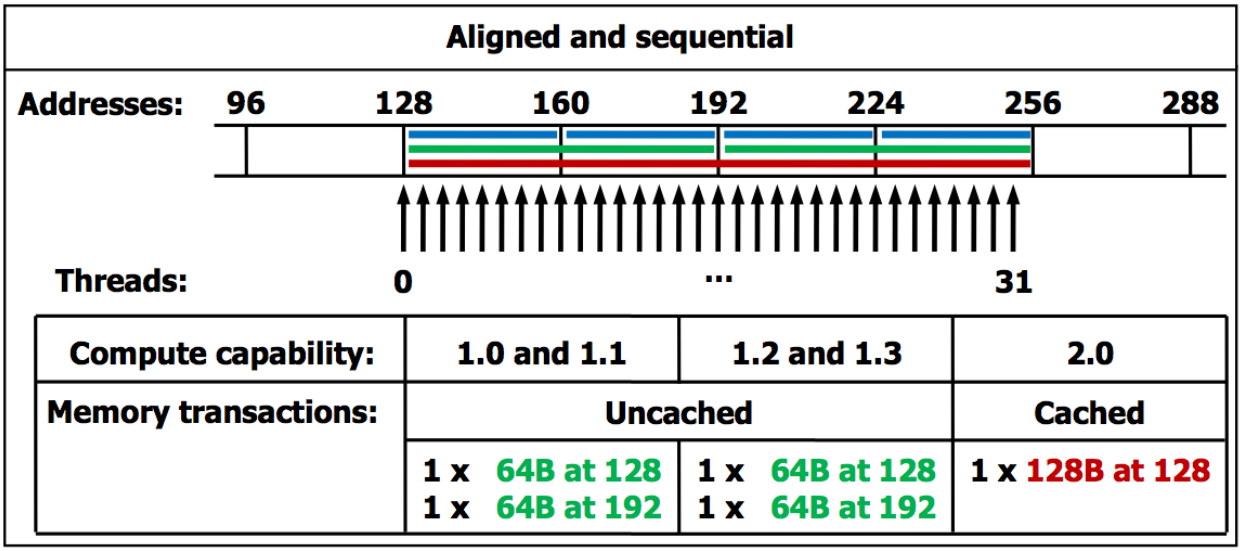


Fig. 10.5: An example of a global memory access by a warp.

More recent examples from the 2016 NVIDIA Programming guide are in Fig. 10.6 and Fig. 10.7.

In `/usr/local/cuda/include/vector_types.h` we find the definition of the type `double2` as

```
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:

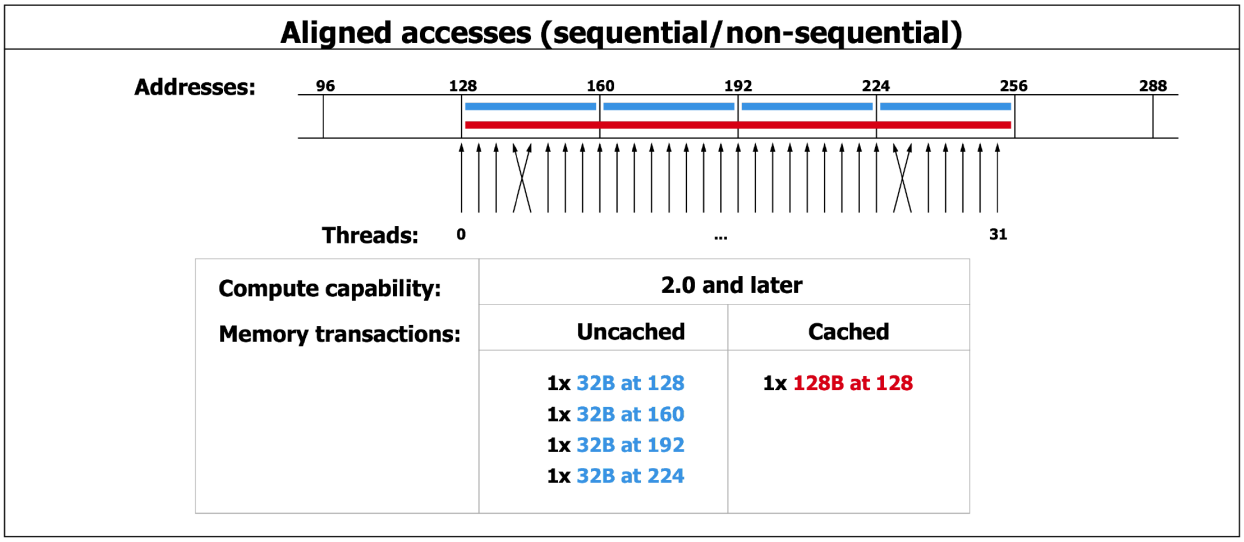


Fig. 10.6: An example of aligned memory access by a warp.

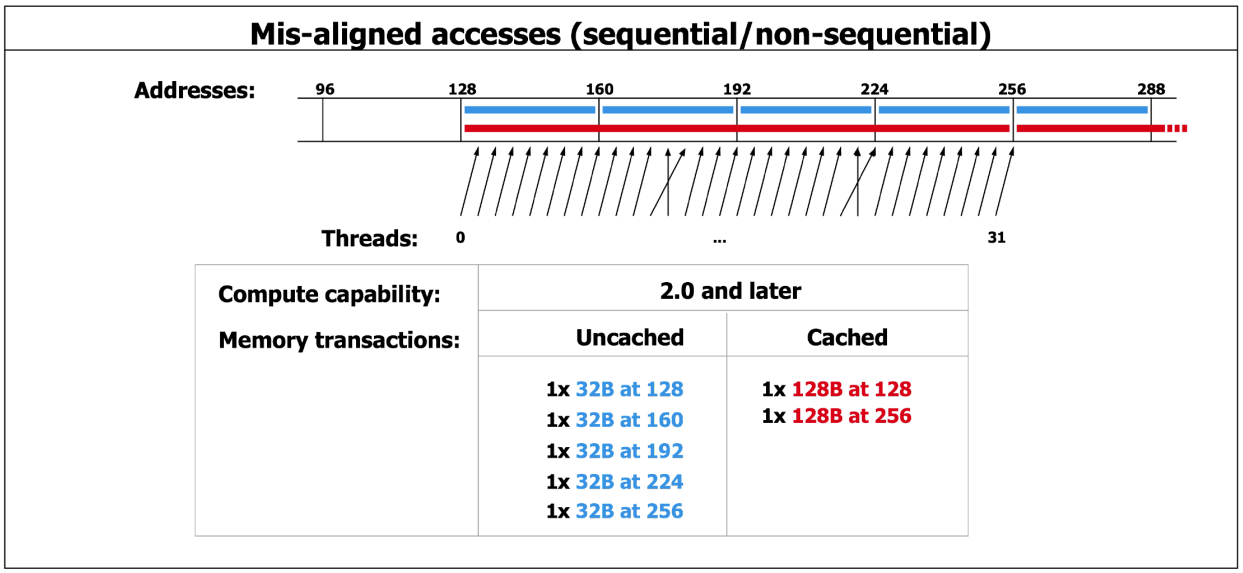


Fig. 10.7: An example of mis-aligned memory access by a warp.

```
__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.

avoiding bank conflicts in shared memory

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.

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. 10.8 and Fig. 10.9.

10.2.2 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. 10.10.

Recall the linear address system to store a matrix. In C, the matrix is stored row wise as a one dimensional array, see Fig. 9.5.

Threads t_0, t_1, t_2, and t_3 access the elements on the first two columns, as shown in Fig. 10.11.

Four threads t_0, t_1, t_2, and t_3 access elements on the first two rows, as shown in Fig. 10.12.

The differences between uncoalesced and coalesced memory accesses are shown in Fig. 10.13.

We can use shared memory for coalescing. Consider Fig. 9.14 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;
```

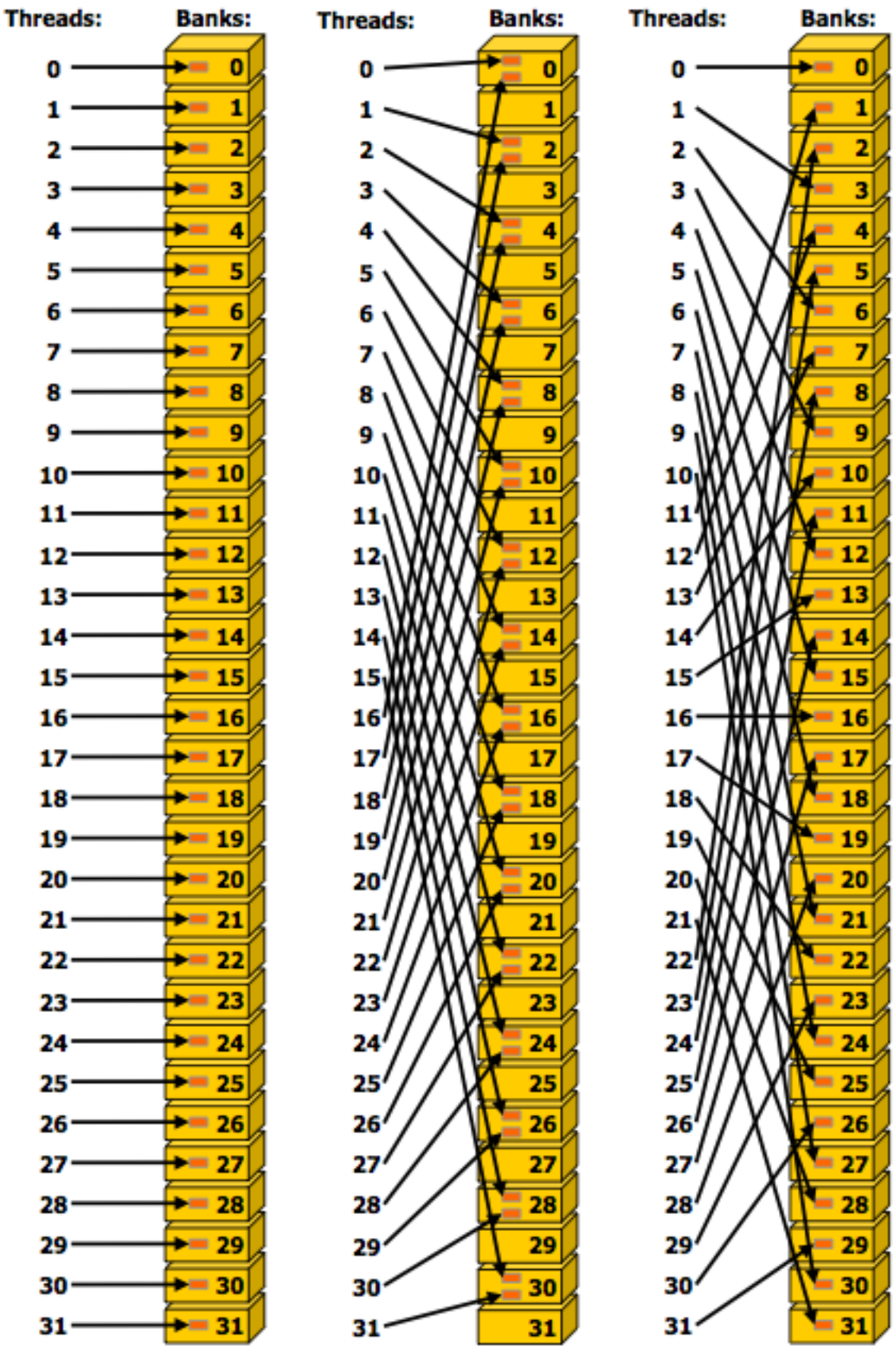


Fig. 10.8: Examples of strided shared memory accesses, copied from Figure G-2 of the NVIDIA Programming Guide.

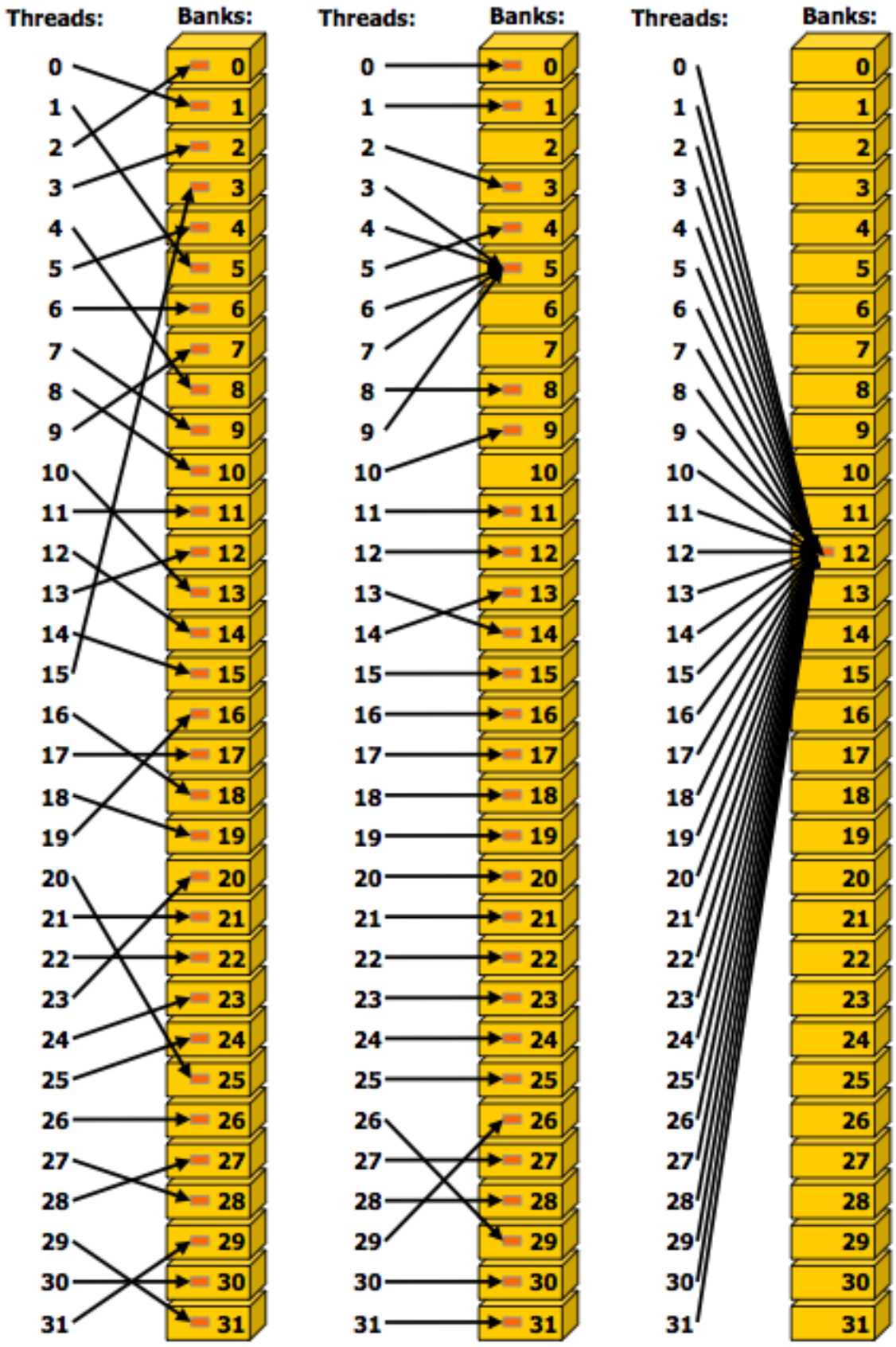


Fig. 10.9: Irregular and colliding shared memory accesses, is Figure G-3 of the NVIDIA Programming Guide.

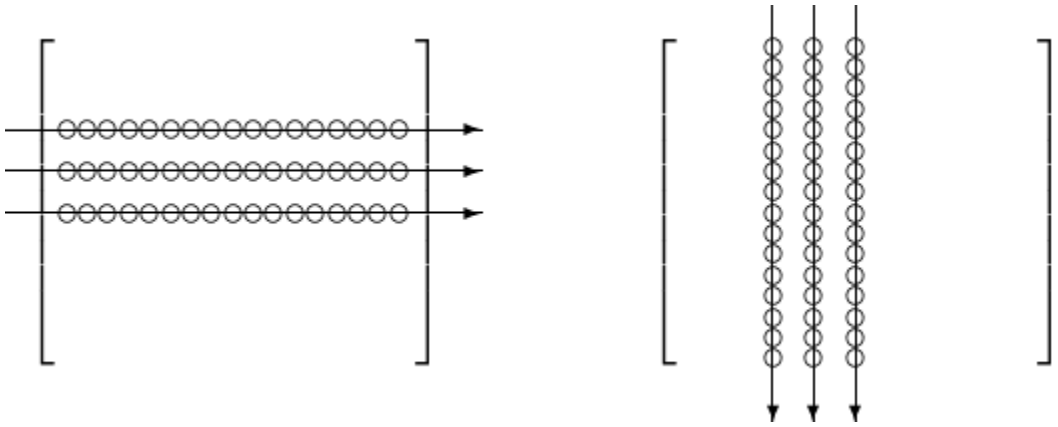


Fig. 10.10: Two ways of accessing elements in a matrix.

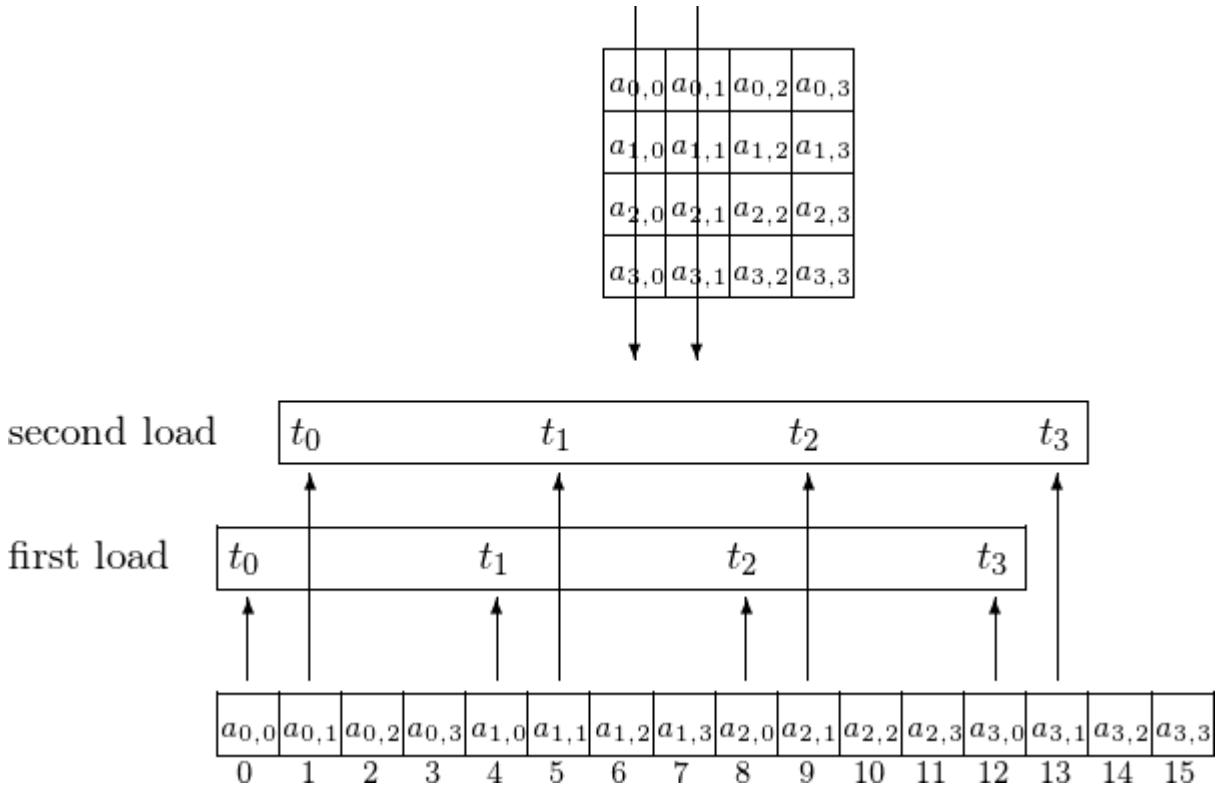


Fig. 10.11: Accessing elements column after column.

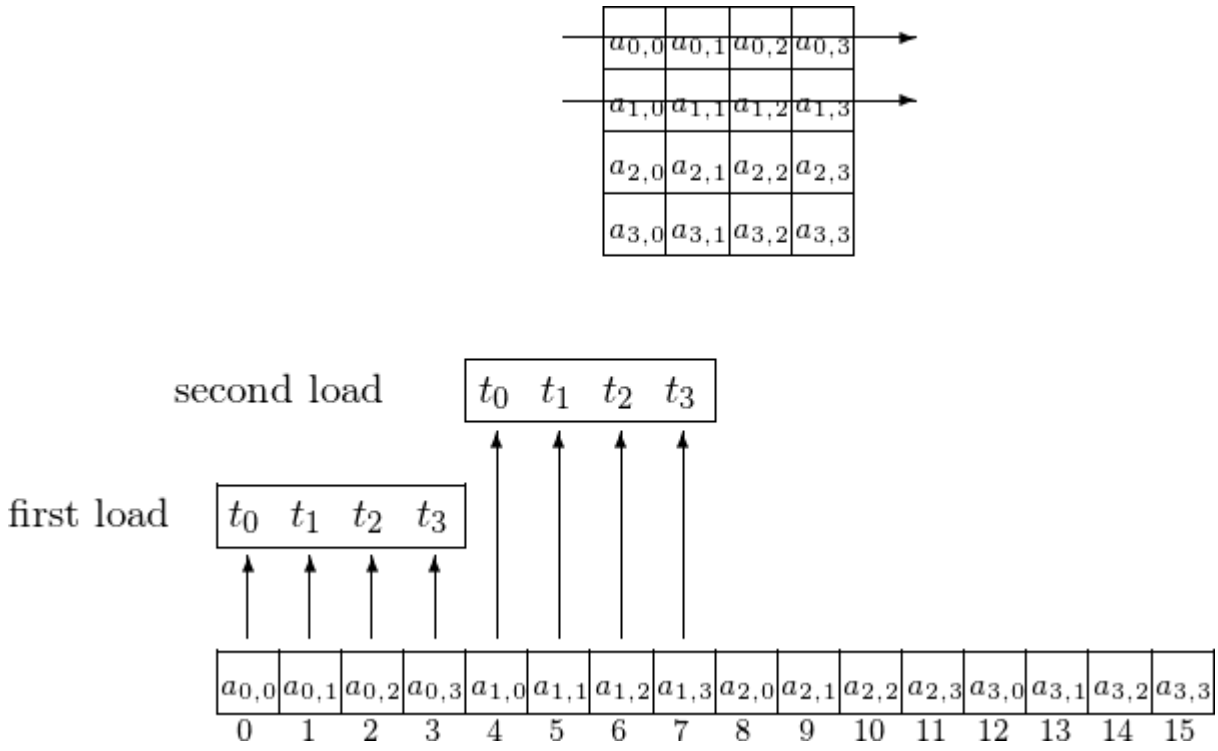


Fig. 10.12: Accesing elements row after row.

```
   int tx = threadIdx.x;        int ty = threadIdx.y;
   int col = bx*w + tx;         int row = by*w + ty;
   float Cv = 0.0;
   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;
   }
}
```

10.2.3 Avoiding Bank Conflicts

Consider the following problem:

On input are $x_0, x_1, x_2, \ldots x_{31}$, all of type float.

The output is

$$\begin{matrix} x_0^2, & x_0^3, & x_0^4, & \ldots, & x_0^{33}, \\ x_1^2, & x_1^3, & x_1^4, & \ldots, & x_1^{33}, \\ x_2^2, & x_2^3, & x_2^4, & \ldots, & x_2^{33}, \\ \vdots & \vdots & \vdots & & \vdots \\ x_{31}^2, & x_{31}^3, & x_{31}^4, & \ldots, & x_{31}^{33}. \end{matrix}$$

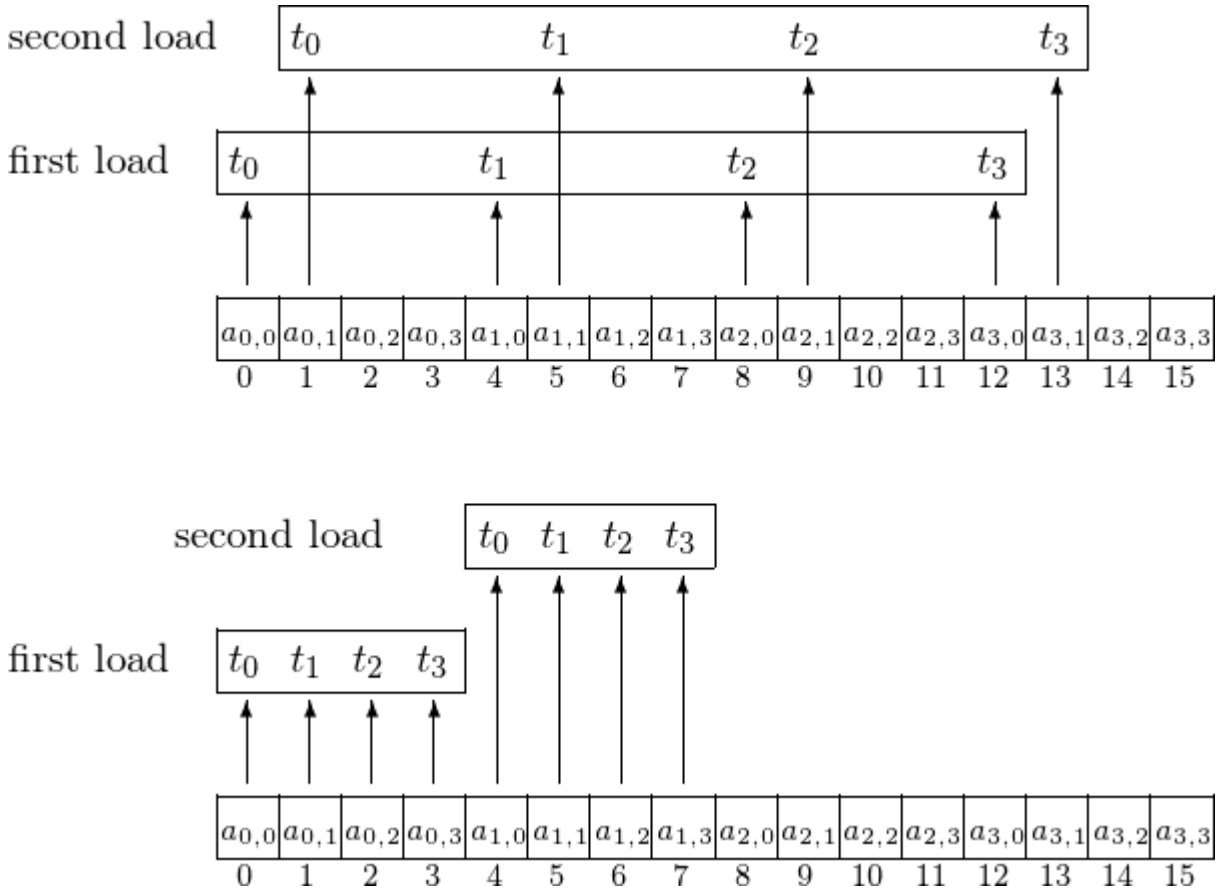


Fig. 10.13: Uncoalesced versus coalesced access.

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.

$$\begin{array}{cccccc} x_0^2, & x_1^2, & x_1^2, & \ldots, & x_{31}^2, \\ x_0^3, & x_1^3, & x_2^3, & \ldots, & x_{31}^3, \\ x_0^4, & x_1^4, & x_2^4, & \ldots, & x_{31}^4, \\ \vdots & \vdots & \vdots & & \vdots \\ x_0^{33}, & x_1^{33}, & x_2^{33}, & \ldots, & x_{31}^{33}. \end{array}$$

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.

10.2.4 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.

10.3 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.

10.3.1 Dynamic Partitioning of Resources

In Table 10.1 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 10.1: 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 10.2, 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 10.2: 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. 10.14.

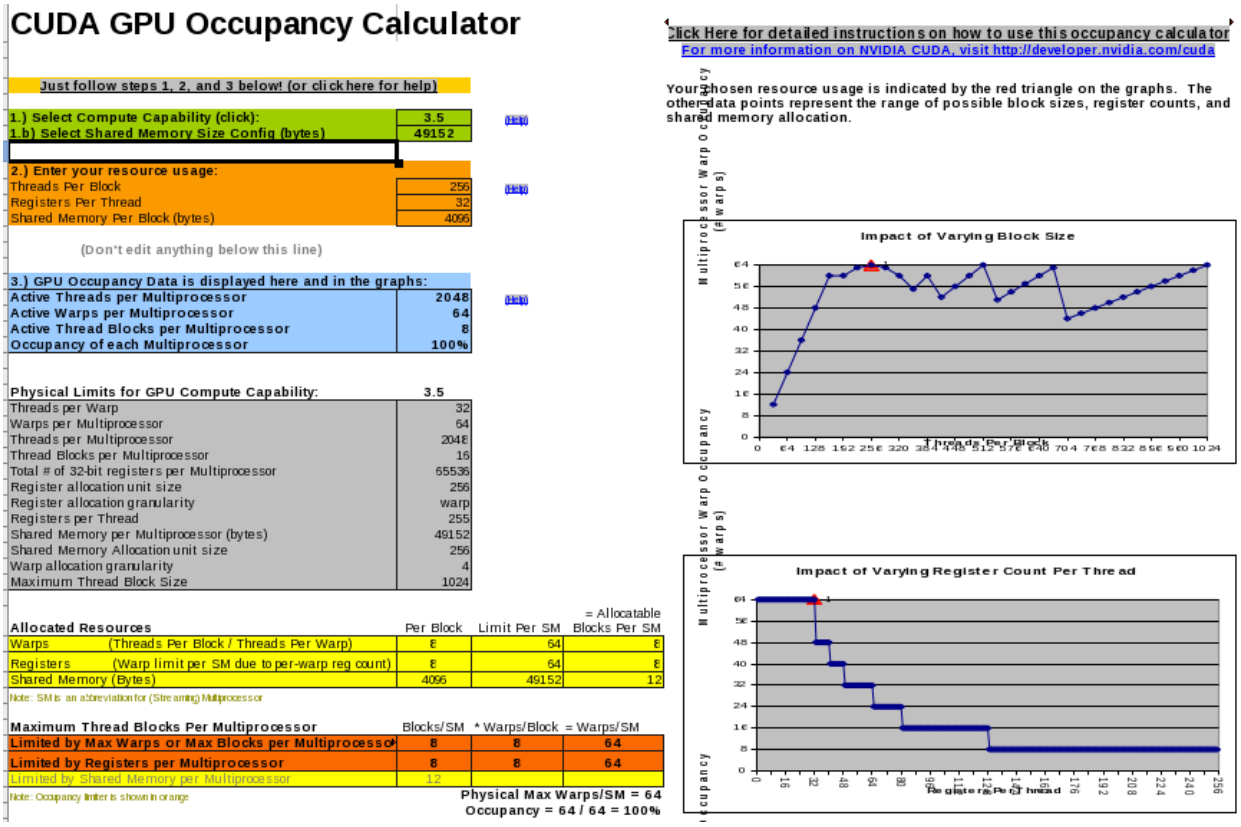


Fig. 10.14: The CUDA occupancy calculator.

10.3.2 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. 10.15, Fig. 10.16, Fig. 10.17, Fig. 10.18, and Fig. 10.19.

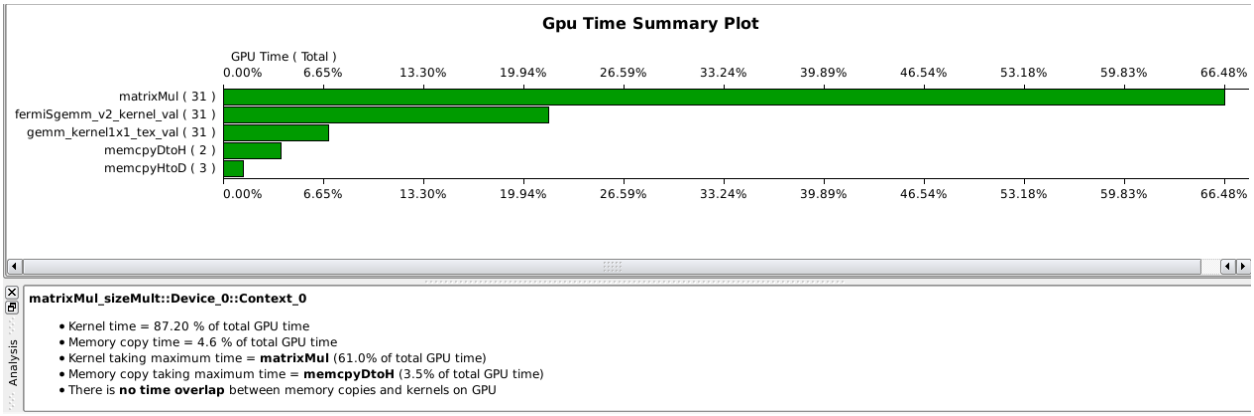


Fig. 10.15: GPU time summary of the matrixMul kernel.

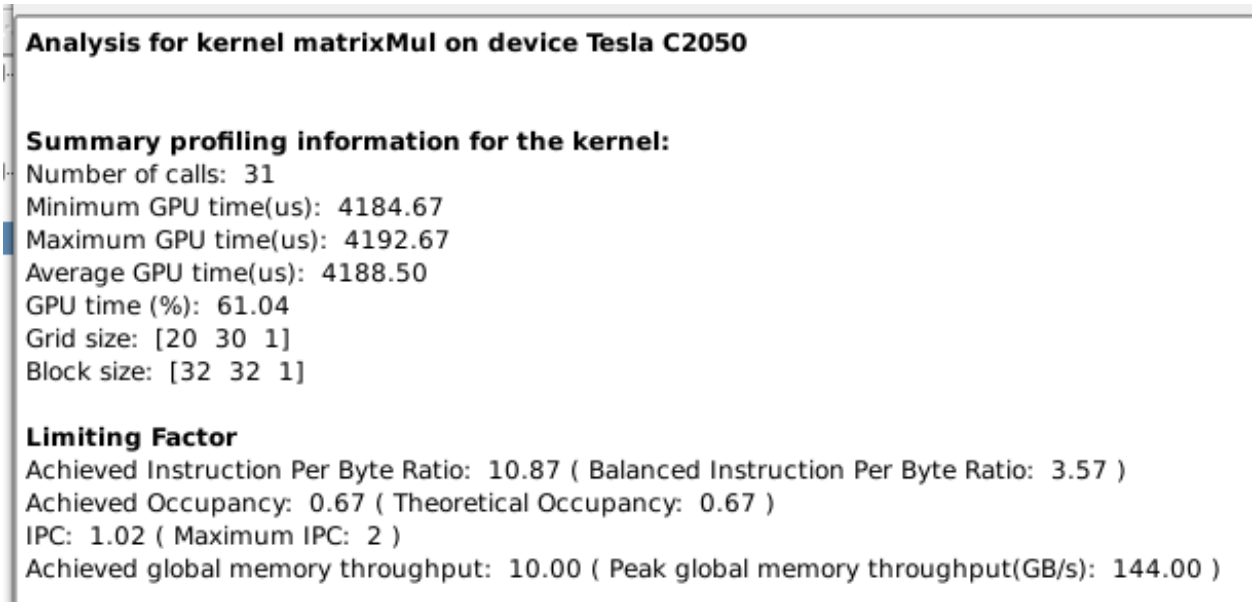


Fig. 10.16: Limiting factor identification of the matrixMul kernel, IPC = Instructions Per Cycle.

10.3.3 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

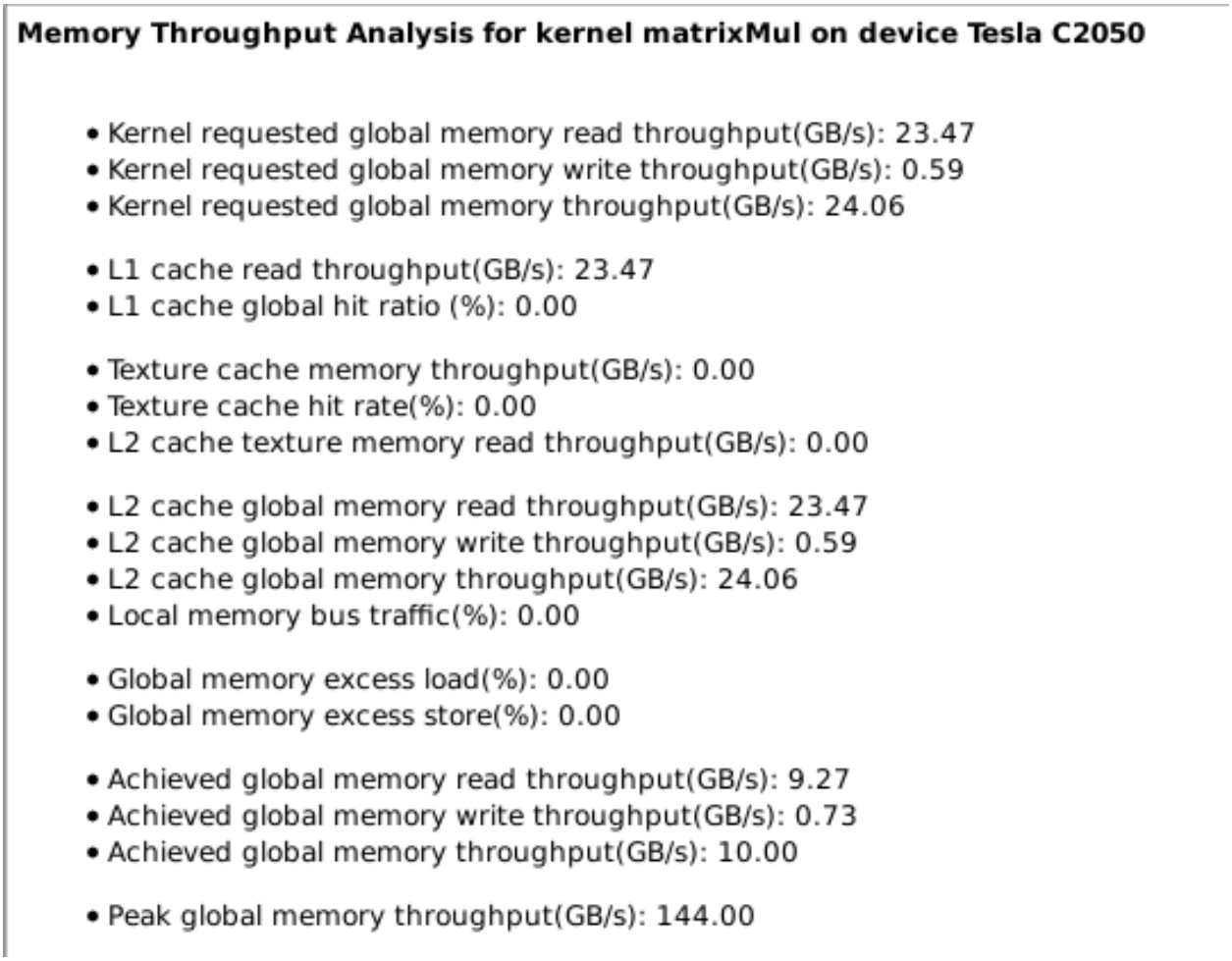


Fig. 10.17: Memory throughput analysis of the matrixMul kernel.

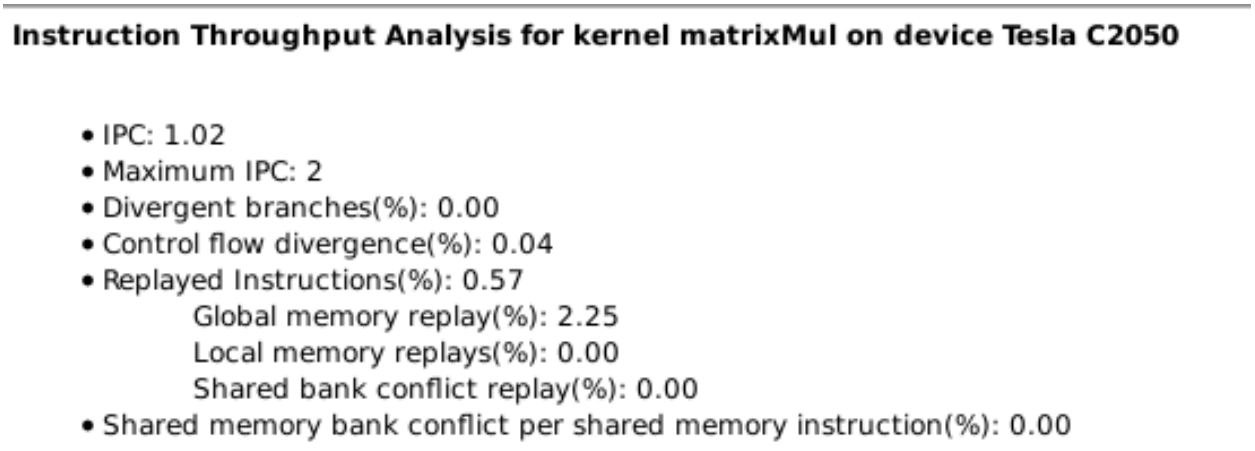


Fig. 10.18: Instruction throughput analysis of the matrixMul kernel, IPC = Instructions Per Cycle.

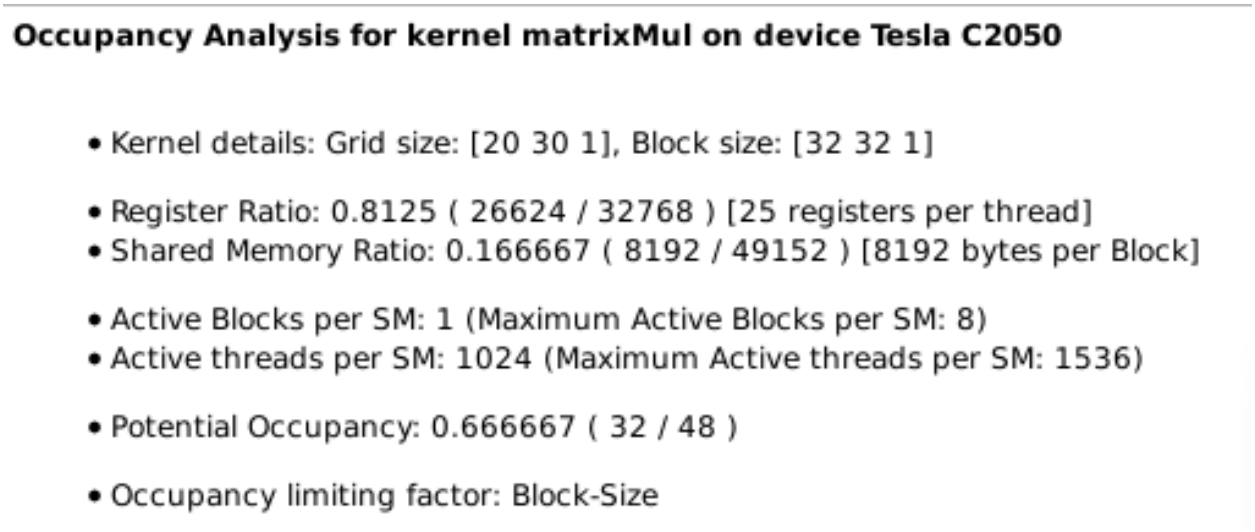


Fig. 10.19: Occupancy analysis of the matrixMul kernel.

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.

Table Table 10.3 is taken from Table 2 of the CUDA C Programming Guide. The ftp in Table 10.3 stands for floating-point and int for integer.

Table 10.3: 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`.

10.3.4 Exercises

1. Examine the occupancy calculator for the graphics card on your laptop or desktop.
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.
4. Redo the second interactions between resource limitations of this lecture using the specifications for compute capability 1.1.

CHAPTER 11

Applications and Case Studies

11.1 Quad Double Arithmetic

11.1.1 Floating-Point Arithmetic

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;

   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`.

The QD library has been ported to the GPU. by Mian Lu, Bingsheng He, and Qiong Luo, described in *Supporting extended precision on graphics processors*, published in the Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010), June 7, 2010, Indianapolis, Indiana, pages 19–26, 2010. The software is available at <http://code.google.com/p/gpuprec/>.

For graphics cards of compute capability < 1.3, one could use the freely available Cg software of Andrew Thrall to achieve double precision using float-float arithmetic.

The `gqd_real` numbers are of type `double4`. The functions below convert a quad double of the QDlib into a GQD quad double and vice versa.

```
#include "gqd_type.h"
#include "vector_types.h"
#include <qd/qd_real.h>

void qd2gqd ( qd_real *a, gqd_real *b )
{
   b->x = a->x[0];
   b->y = a->x[1];
   b->z = a->x[2];
   b->w = a->x[3];
}

void gqd2qd ( gqd_real *a, qd_real *b )
{
   b->x[0] = a->x;
   b->x[1] = a->y;
```

```
   b->x[2] = a->z;
   b->x[3] = a->w;
}
```

A first kernel is listed below.

```
#include "gqd.cu"

__global__ void testdiv2 ( gqd_real *x, gqd_real *y )
{
   *y = *x/2.0;
}

int divide_by_two ( gqd_real *x, gqd_real *y )
{
   gqd_real *xdevice;
   size_t s = sizeof(gqd_real);
   cudaMalloc((void**)&xdevice,s);
   cudaMemcpy(xdevice,x,s,cudaMemcpyHostToDevice);
   gqd_real *ydevice;
   cudaMalloc((void**)&ydevice,s);
   testdiv2<<<1,1>>>(xdevice,ydevice);
   cudaMemcpy(y,ydevice,s,cudaMemcpyDeviceToHost);
   return 0;
}
```

The main program to test our first kernel is below.

```
#include <iostream>
#include <iomanip>
#include "gqd_type.h"
#include "first_gqd_kernel.h"
#include "gqd_qd_util.h"
#include <qd/qd_real.h>
using namespace std;

int main ( int argc, char *argv[] )
{
   qd_real qd_x = qd_real::_pi;
   gqd_real x;
   qd2gqd(&qd_x,&x);
   gqd_real y;

   cout << " x : " << setprecision(64) << qd_x << endl;

   int fail = divide_by_two(&x,&y);

   qd_real qd_y;
   gqd2qd(&y,&qd_y);

   if(fail == 0) cout << " y : " << qd_y << endl;

   cout << "2y : " << 2.0*qd_y << endl;

   return 0;
}
```

The makefile is a bit more complicated because we link two different libraries.

```
QD_ROOT=/usr/local/qd-2.3.13
QD_LIB=/usr/local/lib
GQD_HOME=/usr/local/gqd_1_2
SDK_HOME=/usr/local/cuda/sdk

test_pi2_gqd_kernel:
        @-echo ">>> compiling kernel ..."
        nvcc -I$(GQD_HOME)/inc -I$(SDK_HOME)/C/common/inc \
             -c first_gqd_kernel.cu -arch=sm_13
        @-echo ">>> compiling utilities ..."
        g++ -I/usr/local/cuda/include -I$(GQD_HOME)/inc \
            -I$(QD_ROOT)/include -c gqd_qd_util.cpp
        @-echo ">>> compiling test program ..."
        g++ test_pi2_gqd_kernel.cpp -c \
            -I/usr/local/cuda/include -I$(GQD_HOME)/inc \
            -I$(QD_ROOT)/include
        @-echo ">>> linking ..."
        g++ -I$(GQD_HOME)/inc -I$(QD_ROOT)/include \
             first_gqd_kernel.o test_pi2_gqd_kernel.o gqd_qd_util.o \
             $(QD_LIB)/libqd.a \
             -o /tmp/test_pi2_gqd_kernel \
            -lcuda -lcutil_x86_64 -lcudart \
            -L/usr/local/cuda/lib64 -L$(SDK_HOME)/C/lib
```

Compiling and running goes as follows:

```
$ make test_pi2_gqd_kernel
>>> compiling kernel ...
nvcc -I/usr/local/gqd_1_2/inc -I/usr/local/cuda/sdk/C/common/inc \
             -c first_gqd_kernel.cu -arch=sm_13
>>> compiling utilities ...
g++ -I/usr/local/cuda/include -I/usr/local/gqd_1_2/inc \
            -I/usr/local/qd-2.3.13/include -c gqd_qd_util.cpp
>>> compiling test program ...
g++ test_pi2_gqd_kernel.cpp -c \
            -I/usr/local/cuda/include -I/usr/local/gqd_1_2/inc \
            -I/usr/local/qd-2.3.13/include
>>> linking ...
g++ -I/usr/local/gqd_1_2/inc -I/usr/local/qd-2.3.13/include \
             first_gqd_kernel.o test_pi2_gqd_kernel.o gqd_qd_util.o \
             /usr/local/lib/libqd.a \
             -o /tmp/test_pi2_gqd_kernel \
            -lcuda -lcutil_x86_64 -lcudart \
            -L/usr/local/cuda/lib64 -L/usr/local/cuda/sdk/C/lib
$ /tmp/test_pi2_gqd_kernel
 x : 3.14159265358979323846264338327950288419716939937510582097494459 23e+00
 y : 1.57079632679489661923132169163975144209858469968755291048747229 61e+00
2y : 3.14159265358979323846264338327950288419716939937510582097494459 23e+00
$
```

11.1.2 Quad Double Square Roots

Recall our first CUDA program to take the square root of complex numbers stored in a `double2` array. In using quad doubles on a GPU, we have 3 stages:

1. The kernel in a file with extension `cu` is compiled with `nvcc -c` into an object file.

2. The application code is compiled with `g++ -c`.
3. The linker `g++` takes `.o` files and libraries on input to make an executable file.

Working without a makefile now becomes very tedious. The makefile is listed below.

```
QD_ROOT=/usr/local/qd-2.3.13
QD_LIB=/usr/local/lib
GQD_HOME=/usr/local/gqd_1_2
SDK_HOME=/usr/local/cuda/sdk

sqrt_gqd_kernel:
        @-echo ">>> compiling kernel ..."
        nvcc -I$(GQD_HOME)/inc -I$(SDK_HOME)/C/common/inc \
             -c sqrt_gqd_kernel.cu -arch=sm_13
        @-echo ">>> compiling utilities ..."
        g++ -I/usr/local/cuda/include -I$(GQD_HOME)/inc \
            -I$(QD_ROOT)/include -c gqd_qd_util.cpp
        @-echo ">>> compiling test program ..."
        g++ run_sqrt_gqd_kernel.cpp -c \
            -I/usr/local/cuda/include -I$(GQD_HOME)/inc \
            -I$(QD_ROOT)/include
        @-echo ">>> linking ..."
        g++ -I$(GQD_HOME)/inc -I$(QD_ROOT)/include \
             sqrt_gqd_kernel.o run_sqrt_gqd_kernel.o gqd_qd_util.o \
             $(QD_LIB)/libqd.a \
             -o /tmp/run_sqrt_gqd_kernel \
            -lcuda -lcutil_x86_64 -lcudart \
            -L/usr/local/cuda/lib64 -L$(SDK_HOME)/C/lib
```

The code of a kernel using `gqd_real` in the file `sqrt_gqd_kernel.cu` is listed below.

```
#include "gqd.cu"

__global__ void sqrtNewton ( gqd_real *x, gqd_real *y )
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   gqd_real c = x[i];
   gqd_real r = c;
   for(int j=0; j<10; j++)
      r = r - (r*r - c)/(2.0*r);
   y[i] = r;
}

int sqrt_by_Newton ( int n, gqd_real *x, gqd_real *y )
{
   gqd_real *xdevice;
   size_t s = n*sizeof(gqd_real);
   cudaMalloc((void**)&xdevice,s);
   cudaMemcpy(xdevice,x,s,cudaMemcpyHostToDevice);

   gqd_real *ydevice;
   cudaMalloc((void**)&ydevice,s);

   sqrtNewton<<<n/32,32>>>(xdevice,ydevice);

   cudaMemcpy(y,ydevice,s,cudaMemcpyDeviceToHost);

   return 0;
```

```
}
```

The main program is in `run_sqrt_gqd_kernel.cpp`.

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include "gqd_type.h"
#include "sqrt_gqd_kernel.h"
#include "gqd_qd_util.h"
#include <qd/qd_real.h>
using namespace std;

int main ( int argc, char *argv[] )
{
   const int n = 256;
   gqd_real *x = (gqd_real*)calloc(n,sizeof(gqd_real));
   gqd_real *y = (gqd_real*)calloc(n,sizeof(gqd_real));

   for(int i = 0; i<n; i++)
   {
      x[i].x = (double) (i+2);
      x[i].y = 0.0; x[i].z = 0.0; x[i].w = 0.0;
   }
   int fail = sqrt_by_Newton(n,x,y);
   if(fail == 0)
   {
      const int k = 24;
      qd_real qd_x;
      gqd2qd(&x[k],&qd_x);
      qd_real qd_y;
      gqd2qd(&y[k],&qd_y);
      cout << "     x    : " << setprecision(64) << qd_x << endl;
      cout << "sqrt(x)   : " << setprecision(64) << qd_y << endl;
      cout << "sqrt(x)^2 : " << setprecision(64) << qd_y*qd_y
           << endl;
   }
   return 0;
}
```

We close this chapter with some performance considerations. Consider four quad doubles a, b, c, and d. Ways to store this sequence of four quad doubles are shown in Fig. 11.1 and in Fig. 11.2.

a_0	a_1	a_2	a_3	b_0	b_1	b_2	b_3	c_0	c_1	c_2	c_3	d_0	d_1	d_2	d_3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Fig. 11.1: Four quad doubles stores in a sequential memory layout.

a_0	b_0	c_0	d_0	a_1	b_1	c_1	d_1	a_2	b_2	c_2	d_2	a_3	b_3	c_3	d_3
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Fig. 11.2: Four quad doubles stores in an interval memory layout.

The implementation with an interval memory layout is reported to be three times faster over the sequential memory

layout.

11.1.3 Bibliography

- T.J. Dekker. **A floating-point technique for extending the available precision.** *Numerische Mathematik*, 18(3):224-242, 1971.
- 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>.
- 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 <http://code.google.com/p/gpuprec>.

11.1.4 Exercises

1. Compare the performance of the CUDA program for Newton's method for square root with quad doubles to the code of lecture 29.
2. Extend the code so it works for complex quad double arithmetic.
3. Use quad doubles to implement the second parallel sum algorithm of lecture 33. Could the parallel implementation with quad doubles run as fast as sequential code with doubles?
4. Consider the program to approximate π of lecture 13. Write a version for the GPU and compare the performance with the multicore version of lecture 13.

11.2 Advanced MRI Reconstruction

11.2.1 an Application Case Study

Magnetic Resonance Imaging (MRI) is a safe and noninvasive probe of the structure and function of tissues in the body. MRI consists of two phases:

1. Acquisition or scan: the scanner samples data in the spatial-frequency domain along a predefined trajectory.
2. Reconstruction of the samples into an image.

The limitations of MRI are noise, imaging artifacts, long acquisition times. We have three often conflicting goals:

1. Short scan time to reduce patient discomfort.
2. High resolution and fidelity for early detection.
3. High signal-to-noise ratio (SNR).

Massively parallel computing provides disruptive breakthrough.

Consider the the mathematical problem formulation. The reconstructed image $m(\mathbf{r})$ is

$$\widehat{m}(\mathbf{r}) = \sum_j W(\mathbf{k}_j) s(\mathbf{k}_j) e^{i2\pi \mathbf{k}_j \cdot \mathbf{r}}$$

where

- $W(\mathbf{k})$ is the weighting function to account for nonuniform sampling;

- $s(\mathbf{k})$ is the measured k-space data.

The reconstruction is an inverse fast Fourier Transform on $s(\mathbf{k})$.

To reconstruct images, a Cartesian scan is slow and may result in poor images. A spiral scan of the data as shown schematically in Fig. 11.3 is faster and with gridding leads to images of better quality than the ones reconstructed in rectangular coordinates.

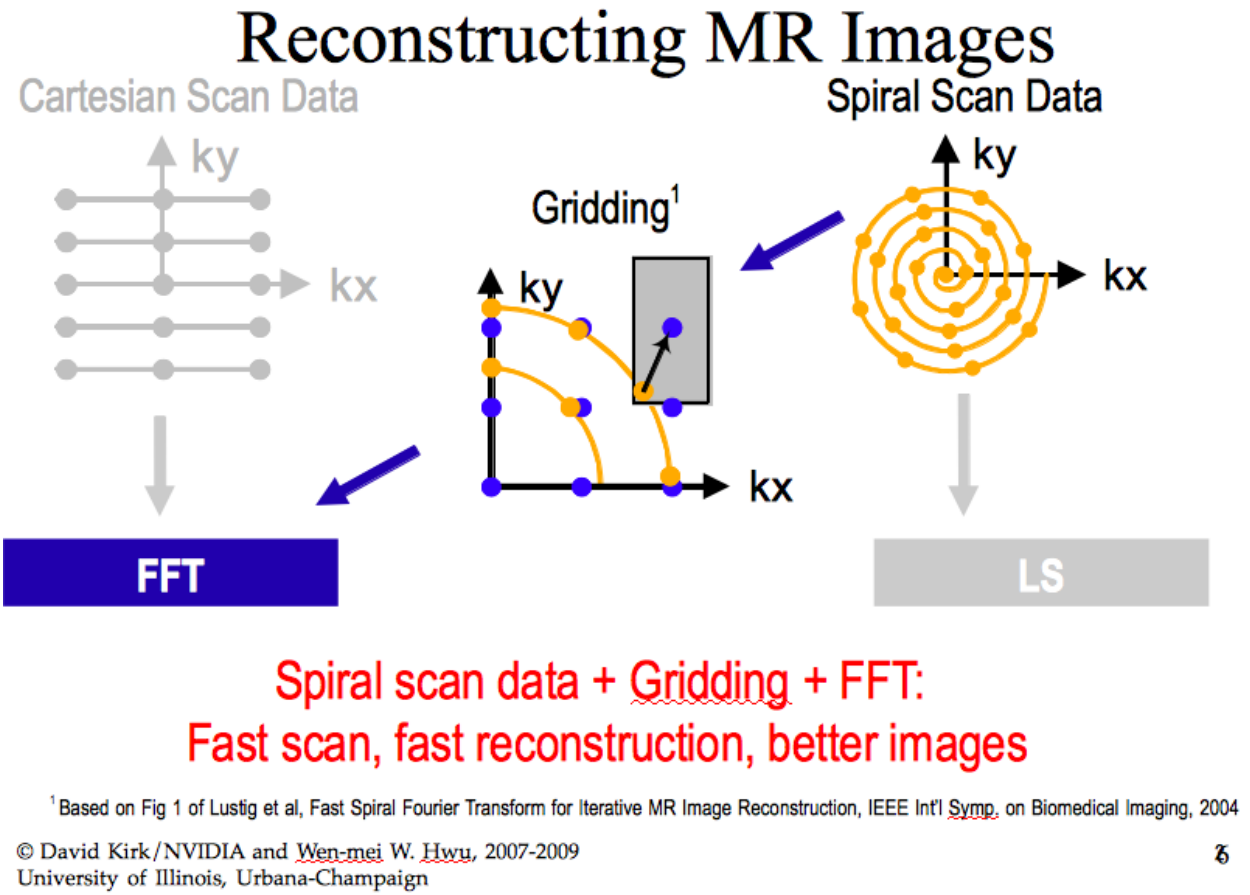


Fig. 11.3: A spiral scan of data to reconstruct images, LS = Least Squares.

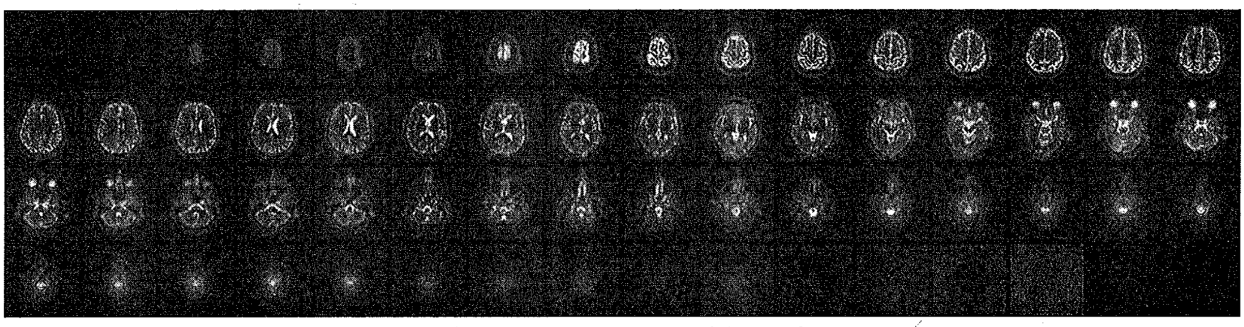
The problem is summarized in Fig. 11.4, copied from the textbook of Hwu and Kirk.

The mathematical problem formulation leads to an iterative reconstructiion method. We start with the formulation of a linear least squares problem, as a quasi-Bayesian estimation problem:

$$\widehat{\rho} = \arg \min_{\rho} \underbrace{||\mathbf{F}\rho - \mathbf{d}||_2^2}_{\text{data fidelity}} + \underbrace{||\mathbf{W}\rho||_2^2}_{\text{prior info}},$$

where

- $\widehat{\rho}$ contains voxel values for reconstructed image,
- the matrix $\mathbf{F}$ models the imaging process,
- $\mathbf{d}$ is a vector of data samples, and
- the matrix $\mathbf{W}$ incorporates prior information, derived from reference images.



Courtesy of Keith Thulborn and Ian Atkinson, Center for MR Research, University of Illinois at Chicago

FIGURE 8.2

The use of non-Cartesian *k*-space sample trajectory and accurate linear-solver-based reconstruction has resulted in new MRI modalities with exciting medical applications. The improved SNR allows reliable collection of *in vivo* concentration data on such chemical substances as sodium in human tissues. The variation or shifting of sodium concentration is an early sign of disease development or tissue death; for example, the sodium map of a human brain can provide an early indication of brain tumor tissue responsiveness to chemotherapy protocols, thus enabling individualized medicine.

Fig. 11.4: Problem statement copied from Hwu and Kirk.

The solution to this linear least squares problem is

$$\widehat{\rho} = \left(\mathbf{F}^H\mathbf{F} + \mathbf{W}^H\mathbf{W}\right)^{-1}\mathbf{F}^H\mathbf{d}.$$

The advanced reconstruction algorithm consists of three primary computations:

1. $Q(\mathbf{x}_n) = \sum_{m=1}^{M} |\phi(\mathbf{k}_m)|^2 e^{i2\pi\mathbf{k}_m\cdot\mathbf{x}_n}$

   where $\phi(\cdot)$ is the Fourier transform of the voxel basis function.

2. $\left[\mathbf{F}^H\mathbf{d}\right]_n = \sum_{m=1}^{M} \phi^*(\mathbf{k}_m)\mathbf{d}(\mathbf{k}_m)e^{i2\pi\mathbf{k}_m\cdot\mathbf{x}_m}$

3. The conjugate gradient solver performs the matrix inversion to solve $\left(\mathbf{F}^H\mathbf{F} + \mathbf{W}^H\mathbf{W}\right)\rho = \mathbf{F}^H\mathbf{d}$.

The calculation for $\mathbf{F}^H\mathbf{d}$ is an excellent candidate for acceleration on the GPU because of its substantial data parallelism.

11.2.2 Acceleration on a GPU

The computation of $\mathbf{F}^H\mathbf{d}$ is defined in the code below.

```
for(m = 0; m < M; m++)
{
   rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
   iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];
   for(n = 0; n < N; n++)
   {
      expFHd = 2*PI*(kx[m]*x[n]
                   + ky[m]*y[n]
                   + kz[m]*z[n]);
      cArg = cos(expFHd);
      sArg = sin(expFHd);
```

```
      rFHd[n] += rMu[m]*cArg - iMu[m]*sArg;
      iFHd[n] += iMu[m]*cArg + rMu[m]*sArg;
   }
}
```

In the development of the kernel we consider the Compute to Global Memory Access (CGMA) ratio. A first version of the kernel follows:

```
__global__ void cmpFHd ( float* rPhi, iPhi, phiMag,
               kx, ky, kz, x, y, z, rMu, iMu, int N)
{
   int m = blockIdx.x*FHd_THREADS_PER_BLOCK + threadIdx.x;

   rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
   iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];

   for(n = 0; n < N; n++)
   {
      expFHd = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);
      carg = cos(expFHd); sArg = sin(expFHd);
      rFHd[n] += rMu[m]*cArg - iMu[m]*sArg;
      iFHd[n] += iMu[m]*cArg + rMu[m]*sArg;
   }
}
```

A first modificiation is the splitting of the outer loop.

```
for(m = 0; m < M; m++)
{
   rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
   iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];
}
for(m = 0; m < M; m++)
{
   for(n = 0; n < N; n++)
   {
      expFHd = 2*PI*(kx[m]*x[n]
                   + ky[m]*y[n]
                   + kz[m]*z[n]);
      cArg = cos(expFHd);
      sArg = sin(expFHd);
      rFHd[n] += rMu[m]*cArg - iMu[m]*sArg;
      iFHd[n] += iMu[m]*cArg + rMu[m]*sArg;
   }
}
```

We convert the first loop into a CUDA kernel:

```
__global__ void cmpMu ( float *rPhi,iPhi,rD,iD,rMu,iMu)
{
   int m = blockIdx * MU_THREADS_PER_BLOCK + threadIdx.x;

   rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
   iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];
}
```

Because M can be very big, we will have many threads. For example, if $M = 65,536$, with 512 threads per block, we have $65,536/512 = 128$ blocks.

Then the second loop is defined by another kernel.

```
__global__ void cmpFHd ( float* rPhi, iPhi, PhiMag,
             kx, ky, kz, x, y, z, rMu, iMu, int N )
{
   int m = blockIdx.x*FHd_THREADS_PER_BLOCK + threadIdx.x;

   for(n = 0; n < N; n++)
   {
      float expFHd = 2*PI*(kx[m]*x[n]+ky[m]*y[n]
                                      +kz[m]*z[n]);
      float cArg = cos(expFHd);
      float sArg = sin(expFHd);

      rFHd[n] += rMu[m]*cArg - iMu[m]*sArg;
      iFHd[n] += iMu[m]*cArg + rMu[m]*sArg;
   }
}
```

To avoid conflicts between threads, we interchange the inner and the outer loops:

```
for(m=0; m<M; m++)
{
   for(n=0; n<N; n++)
   {
      expFHd = 2*PI*(kx[m]*x[n]
                    +ky[m]*y[n]
                    +kz[m]*z[n]);
      cArg = cos(expFHd);
      sArg = sin(expFHd);
      rFHd[n] += rMu[m]*cArg
               - iMu[m]*sArg;
      iFHd[n] += iMu[m]*cArg
               + rMu[m]*sArg;
   }
}
```

```
for(n=0; n<N; n++)
{
   for(m=0; m<M; m++)
   {
      expFHd = 2*PI*(kx[m]*x[n]
                    +ky[m]*y[n]
                    +ky[m]*y[n]);
      cArg = cos(expFHd);
      sArg = sin(expFHd);
      rFHd[n] += rMu[m]*cArg
               - iMu[m]*sArg;
      rFHd[n] += iMu[m]*cArg
               + rMu[m]*sArg;
   }
}
```

In the new kernel, the n-th element will be computed by the n-th thread. The new kernel is listed below.

```
__global__ void cmpFHd ( float* rPhi, iPhi, phiMag,
             kx, ky, kz, x, y, z, rMu, iMu, int M )
{
   int n = blockIdx.x*FHD_THREAD_PER_BLOCK + threadIdx.x;

   for(m = 0; m < M; m++)
   {
      float expFHd = 2*PI*(kx[m]*x[n]+ky[m]*y[n]
                                      +kz[m]*z[n]);
      float cArg = cos(expFHd);
      float sArg = sin(expFHd);
      rFHd[n] += rMu[m]*cArg - iMu[m]*sArg;
      iFHd[n] += iMu[m]*cArg + rMu[m]*sArg;
   }
}
```

For a 128^3 image, there are $(2^7)^3 = 2,097,152$ threads. For higher resolutions, e.g.: 512^3, multiple kernels may be needed.

To reduce memory accesses, the next kernel uses registers:

```
__global__ void cmpFHd ( float* rPhi, iPhi, phiMag,
             kx, ky, kz, x, y, z, rMu, iMu, int M )
{
   int n = blockIdx.x*FHD_THREAD_PER_BLOCK + threadIdx.x;
   float xn = x[n]; float yn = y[n]; float zn = z[n];
   float rFHdn = rFHd[n]; float iFHdn = iFHd[n];
   for(m = 0; m < M; m++)
   {
      float expFHd = 2*PI*(kx[m]*xn+ky[m]*yn+kz[m]*zn);
      float cArg = cos(expFHd);
      float sArg = sin(expFHd);
      rFHdn += rMu[m]*cArg - iMu[m]*sArg;
      iFHdn += iMu[m]*cArg + rMu[m]*sArg;
   }
   rFHd[n] = rFHdn; iFHd[n] = iFHdn;
}
```

The usage of registers improved the Compute to Memory Access (CGMA) ratio.

Using constant memory we use cache more efficiently. The technique is called chunking data. Limited in size to 64KB, we need to invoke the kernel multiple times.

```
__constant__ float kx[CHUNK_SZ],ky[CHUNK_SZ],kz[CHUNK_SZ];
// code omitted ...
for(i = 0; k < M/CHUNK_SZ; i++)
{
   cudaMemcpy(kx,&kx[i*CHUNK_SZ],4*CHUNK_SZ,
              cudaMemCpyHostToDevice);
   cudaMemcpy(ky,&ky[i*CHUNK_SZ],4*CHUNK_SZ,
              cudaMemCpyHostToDevice);
   cudaMemcpy(kz,&kz[i*CHUNK_SZ],4*CHUNK_SZ,
              cudaMemCpyHostToDevice);
   // code omitted ...
   cmpFHD<<<FHd_THREADS_PER_BLOCK,
            N/FHd_THREADS_PER_BLOCK>>>
     (rPhi,iPhi,phiMag,x,y,z,rMu,iMu,M);
}
```

Due to size limitations of constant memory and cache, we will adjust the memory layout. Instead of storing the components of *k*-space data in three separate arrays, we use an array of structs:

```
struct kdata
{
   float x, float y, float z;
}
__constant struct kdata k[CHUNK_SZ];
```

and then in the kernel we use `k[m].x`, `k[m].y`, and `k[m].z`.

In the next optimization, we use the hardware trigonometric functions. Instead of `cos` and `sin` as implemented in software, the hardware versions `__cos` and `__sin` provide a much higher throughput. The `__cos` and `__sin` are implemented as hardware instructions executed by the special function units.

We need to be careful about a loss of accuracy.

The validation involves a perfect image:

- a reverse process to generate scanned data;
- metrics: mean square error and signal-to-noise ratios.

The last stage is the experimental performance tuning.

11.2.3 Bibliography

- A. Lu, I.C. Atkinson, and K.R. Thulborn. **Sodium Magnetic Resonance Imaging and its Bioscale of Tissue Sodium Concentration**. *Encyclopedia of Magnetic Resonance*, John Wiley and Sons, 2010.
- S.S. Stone, J.P. Haldar, S.C. Tsao, W.-m.W. Hwu, B.P. Sutton, and Z.-P. Liang. **Accelerating advanced MRI reconstructions on GPUs**. *Journal of Parallel and Distributed Computing* 68(10): 1307-1318, 2008.
- The IMPATIENT MRI Toolset, open source software available at <http://impact.crhc.illinois.edu/mri.php>.

11.3 Concurrent Kernels and Multiple GPUs

11.3.1 Page Locked Host Memory

In contrast to regular pageable host memory, the runtime provides functions to allocate (and free) *page locked* memory. Another name for memory that is page locked is *pinned*.

Using page locked memory has several benefits:

- Copies between page locked memory and device memory can be performed concurrently with kernel execution.
- Page locked host memory can be mapped into the address space of the device, eliminating the need to copy, we say *zero copy*.
- Bandwidth between page locked host memory and device may be higher.

Page locked host memory is a scarce resource. The NVIDIA CUDA Best Practices Guide assigns a low priority to zero-copy operations (i.e.: mapping host memory to the device).

To allocate page locked memory, we use `cudaHostAlloc()` and to free the memory, we call `cudaFreeHost()`.

To map host memory on the device:

- The flag `cudaHostAllocMapped` must be given to `cudaHostAlloc()` when allocating host memory.
- A call to `cudaHostGetDevicePointer()` maps the host memory to the device.

If all goes well, then no copies from host to device memory and from device to host memory are needed.

Not all devices support pinned memory, it is recommended practice to check the device properties (see the `deviceQuery` in the SDK).

Next we illustrate how a programmer may use pinned memory with a simple program. A run of this program is below.

```
$ /tmp/pinnedmemoryuse
Tesla K20c supports mapping host memory.

The error code of cudeHostAlloc : 0

Squaring 32 numbers 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 31 32...


The fail code of cudaHostGetDevicePointer : 0

After squaring 32 numbers 1 4 9 16 25 36 49 64 81 \
100 121 144 169 196 225 256 289 324 361 400 441 484 \
```

```
529 576 625 676 729 784 841 900 961 1024...

$
```

The execution of the program is defined by the code in `pinnedmemoryuse.cu`, listed below. First we check whether the devide supports pinned memory.

```
#include <stdio.h>

int checkDeviceProp ( cudaDeviceProp p );
/*
 * Returns 0 if the device does not support mapping
 * host memory, returns 1 otherwise. */

int checkDeviceProp ( cudaDeviceProp p )
{
   int support = p.canMapHostMemory;

   if(support == 0)
      printf("%s does not support mapping host memory.\n",
             p.name);
   else
      printf("%s supports mapping host memory.\n",p.name);

   return support;
}
```

To illustrate pinned memory, we use the following simple kernel.

```
__global__ void Square ( float *x )
/*
 * A kernel where the i-th thread squares x[i]
 * and stores the result in x[i]. */
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   x[i] = x[i]*x[i];
}
```

This kernel is launched in the main program below.

```
void square_with_pinned_memory ( int n );
/*
 * Illustrates the use of pinned memory to square
 * a sequence of n numbers. */

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

   cudaGetDeviceProperties(&dev,0);

   int success = checkDeviceProp(dev);

   if(success != 0)
      square_with_pinned_memory(32);

   return 0;
}
```

The function which allocates the pinned memory is below.

```
void square_with_pinned_memory ( int n )
{
   float *xhost;
   size_t sz = n*sizeof(float);
   int error = cudaHostAlloc((void**)&xhost,
                  sz,cudaHostAllocMapped);
   printf("\nThe error code of cudeHostAlloc : %d\n",
          error);

   for(int i=0; i<n; i++) xhost[i] = (float) (i+1);
   printf("\nSquaring %d numbers",n);
   for(int i=0; i<n; i++) printf(" %d",(int) xhost[i]);
   printf("...\n\n");

   // mapping host memory

   float *xdevice;

   int fail = cudaHostGetDevicePointer
                ((void**)&xdevice,(void*)xhost,0);
   printf("\nThe fail code of cudaHostGetDevicePointer : \
          %d\n",fail);

   Square<<<1,n>>>(xdevice);

   cudaDeviceSynchronize();

   printf("\nAfter squaring %d numbers",n);
   for(int i=0; i<n; i++) printf(" %d",(int) xhost[i]);
   printf("...\n\n");

   cudaFreeHost(xhost);
}
```

11.3.2 Concurrent Kernels

The Fermi architecture supports the simultaneous execution of kernels. The benefits of this concurrency are the following.

- Simultaneous execution of small kernels utilize whole GPU.
- Overlapping kernel execution with device to host memory copy.

A *stream* is a sequence of commands that execute in order. Different streams may execute concurrently. The maximum number of kernel launches that a device can execute concurrently is four.

That the GPU may be fully utilized is illustrated in Fig. 11.5.

The overlapping of execution of kernels with memory copies is illustrated in Fig. 11.6.

To illustrate the use of streams with actual code, we consider a simple kernel to square a sequence of numbers. Its execution happens as shown below.

```
$ /tmp/concurrent
Tesla K20c supports concurrent kernels
  compute capability : 3.5
  number of multiprocessors : 13
```

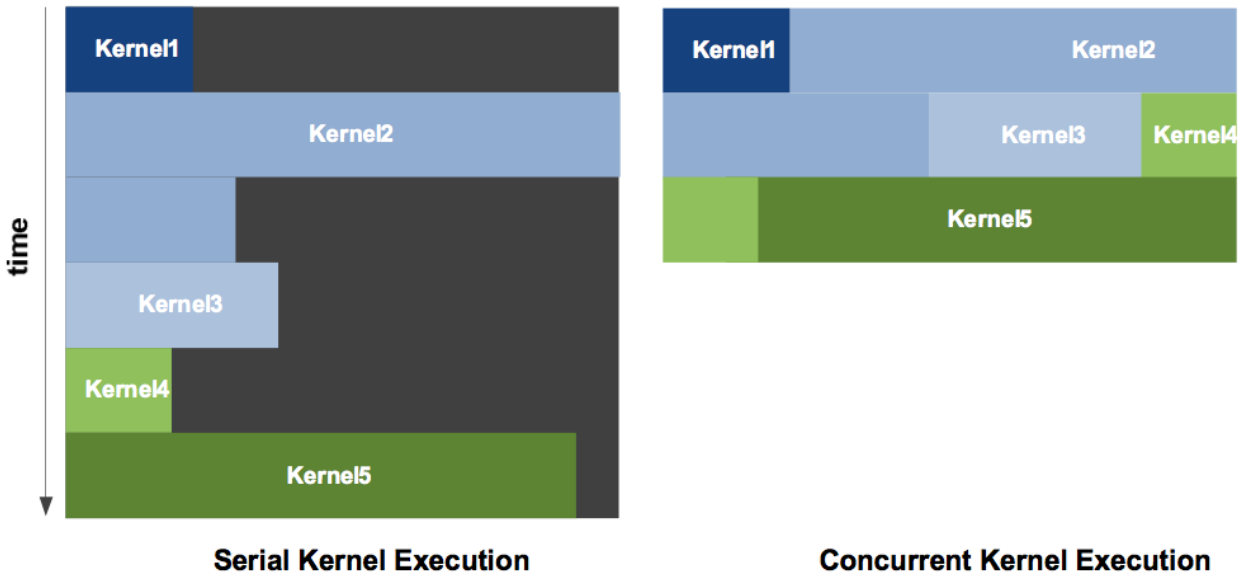


Fig. 11.5: Concurrent kernel execution, taken from the NVIDIA Fermi Compute Architecture Whitepaper.

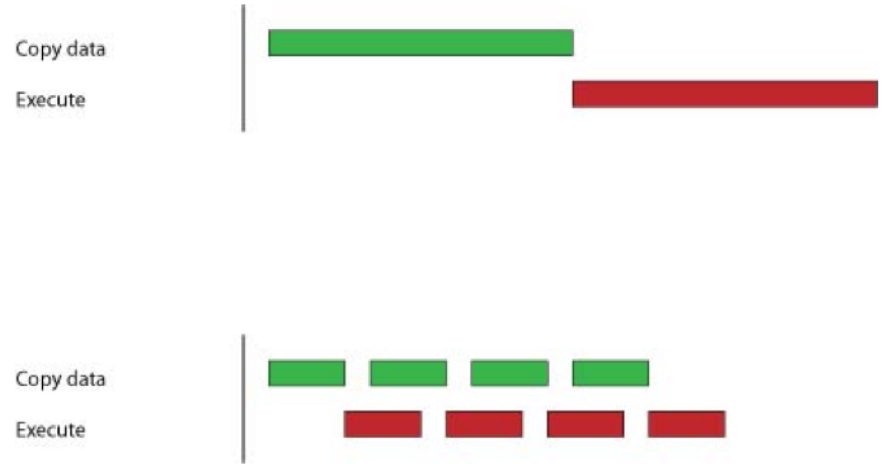


Figure 3.1 Timeline Comparison for Sequential (top) and Concurrent (bottom) Copy and Kernel Execution

Fig. 11.6: Concurrent copy and kernel execution with 4 streams, taken from the NVIDIA CUDA Best Practices Guide.

```
Launching 4 kernels on 16 numbers 1 2 3 4 5 6 7 8 9 10 \
11 12 13 14 15 16...

the 16 squared numbers are 1 4 9 16 25 36 49 64 81 100 \
121 144 169 196 225 256
$
```

The simple kernel is defined by the function `Square`, listed below.

```
__global__ void Square ( float *x, float *y )
/*
 * A kernel where the i-th thread squares x[i]
 * and stores the result in y[i]. */
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   y[i] = x[i]*x[i];
}
```

Before we launch this kernel, we want to check if our GPU supports concurrency. This check is done by the following function.

```
int checkDeviceProp ( cudaDeviceProp p )
{
   int support = p.concurrentKernels;

   if(support == 0)
      printf("%s does not support concurrent kernels\n",
             p.name);
   else
      printf("%s supports concurrent kernels\n",p.name);

   printf("  compute capability : %d.%d \n",
          p.major,p.minor);
   printf("  number of multiprocessors : %d \n",
          p.multiProcessorCount);

   return support;
}
```

Then the main program follows.

```
void launchKernels ( void );
/*
 * Launches concurrent kernels on arrays of floats. */

int main ( int argc, char* argv[] )
{
   cudaDeviceProp dev;
   cudaGetDeviceProperties(&dev,0);

   int success = checkDeviceProp(dev);
   if(success != 0) launchKernels();

   return 0;
}
```

The memory allocation and the asynchronous kernel execution is defined in the code below.

```
void launchKernels ( void )
{
   const int nbstreams = 4;
   const int chunk = 4;
   const int nbdata = chunk*nbstreams;

   float *xhost;
   size_t sz = nbdata*sizeof(float);
   // memory allocation
   cudaMallocHost((void**)&xhost,sz);

   for(int i=0; i<nbdata; i++) xhost[i] = (float) (i+1);
   printf("\nLaunching %d kernels on %d numbers",
          nbstreams,nbdata);
   for(int i=0; i<nbdata; i++)
      printf(" %d",(int) xhost[i]);
   printf("...\n\n");
   float *xdevice; cudaMalloc((void**)&xdevice,sz);
   float *ydevice; cudaMalloc((void**)&ydevice,sz);
   // asynchronous execution
   cudaStream_t s[nbstreams];
   for(int i=0; i<nbstreams; i++) cudaStreamCreate(&s[i]);

   for(int i=0; i<nbstreams; i++)
      cudaMemcpyAsync
         (&xdevice[i*chunk],&xhost[i*chunk],
          sz/nbstreams,cudaMemcpyHostToDevice,s[i]);

   for(int i=0; i<nbstreams; i++)
      Square<<<1,chunk,0,s[i]>>>
         (&xdevice[i*chunk],&ydevice[i*chunk]);

   for(int i=0; i<nbstreams; i++)
      cudaMemcpyAsync
         (&xhost[i*chunk],&ydevice[i*chunk],
          sz/nbstreams,cudaMemcpyDeviceToHost,s[i]);

   cudaDeviceSynchronize();

   printf("the %d squared numbers are",nbdata);
   for(int i=0; i<nbdata; i++)
      printf(" %d",(int) xhost[i]);
   printf("\n");

   for(int i=0; i<nbstreams; i++) cudaStreamDestroy(s[i]);
   cudaFreeHost(xhost);
   cudaFree(xdevice); cudaFree(ydevice);
}
```

11.3.3 Using Multiple GPUs

Before we can use multiple GPUs, it is good to count how many devices are available. A run to enumerate the available devices is below.

```
$ /tmp/count_devices
number of devices : 3
```

```
graphics card 0 :
  name : Tesla K20c
  number of multiprocessors : 13
graphics card 1 :
  name : GeForce GT 620
  number of multiprocessors : 2
graphics card 2 :
  name : Tesla K20c
  number of multiprocessors : 13
$
```

The CUDA program to count the devices is listed next. The instructions are based on the `deviceQuery.cpp` of the GPU Computing SDK.

```
#include <stdio.h>

void printDeviceProp ( cudaDeviceProp p )
/*
 * prints some device properties */
{
   printf("  name : %s \n",p.name);
   printf("  number of multiprocessors : %d \n",
          p.multiProcessorCount);
}

int main ( int argc, char* argv[] )
{
   int deviceCount;
   cudaGetDeviceCount(&deviceCount);
   printf("number of devices : %d\n",deviceCount);

   for(int d = 0; d < deviceCount; d++)
   {
      cudaDeviceProp dev;
      cudaGetDeviceProperties(&dev,d);
      printf("graphics card %d :\n",d);
      printDeviceProp(dev);
   }

   return 0;
}
```

Chapter 8 of the NVIDIA CUDA Best Practices Guide describes multi-GPU programming.

To work with p GPUs concurrently, the CPU can use

- p lightweight threads (Pthreads, OpenMP, etc); or
- p heavyweight threads (or processes) with MPI.

The command to select a GPU is `cudaSetDevice()`.

All inter-GPU communication happens through the host. See the `simpleMultiGPU` of the GPU Computing SDK.

CHAPTER 12

Coprocessor Acceleration

12.1 The Intel Xeon Phi Coprocessor

12.1.1 Many Integrated Core Architecture

The Intel Xeon Phi coprocessor, defined by the Intel Many Integrated Core Architecture (Intel MIC) has the following characteristics:

- 1.01 TFLOPS double precision peak performance on one chip.
- CPU-like versatile programming: *optimize once, run anywhere*. Optimizing for the Intel MIC is similar to optimizing for CPUs.
- Programming Tools include OpenMP, OpenCL, Intel TBB, pthreads, Cilk.

In the High Performance Computing market, the Intel MIC is a direct competitor to the NVIDIA GPUs.

The schematic in Fig. 12.1 sketches the decision path in the adoption of the coprocessor.

In comparing the Phi with CUDA, we first list the similarities between Phi and CUDA. Also the Intel Phi is an accelerator:

- Massively parallel means: many threads run in parallel.
- Without extra effort unlikely to obtain great performance.
- Both Intel Phi and GPU devices accelerate applications in *offload mode* where portions of the application are accelerated by a remote device.
- Directive-based programming is supported:
  - OpenMP on the Intel Phi; and
  - OpenACC on CUDA-enabled devices.
- Programming with libraries is supported on both.

What makes the Phi different from CUDA is that the Intel Phi is not just an accelerator:

- In *coprocessor native execution mode*, the Phi appears as another machine connected to the host, like another node in a cluster.

  In *symmetric execution mode*, application processes run on both the host and the coprocessor, communicating through some sort of message passing.
- MIMD versus SIMD:

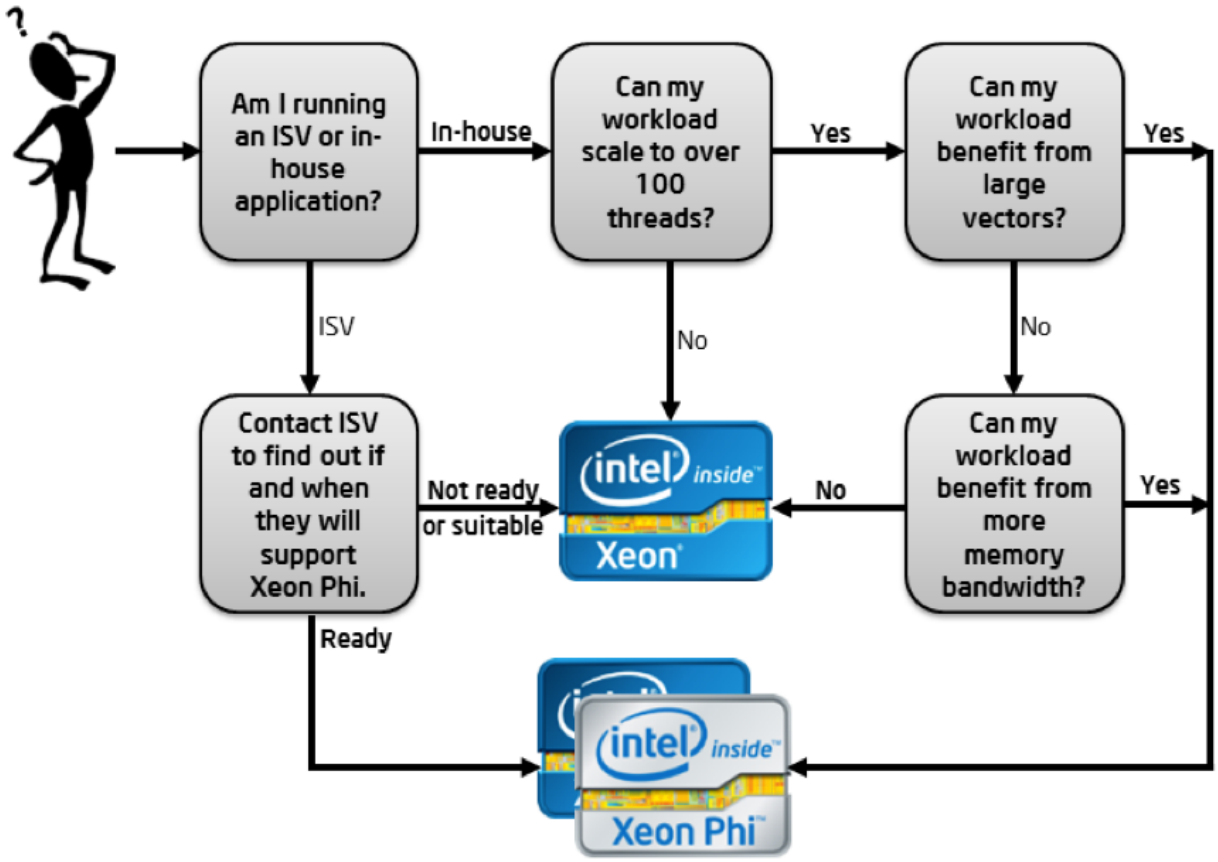


Fig. 12.1: Is the Intel Xeon Phi coprocessor right for me? Copied from the Intel site, from <https://software.intel.com/mic-developer>

- CUDA threads are grouped in blocks (work groups in OpenCL) in a SIMD (Single Instruction Multiple Data) model.
- Phi coprocessors run generic MIMD threads individually. MIMD = Multiple Instruction Multiple Data.

begin{frame}{the Intel Many Integrated Core architecture}

The Intel Xeon Phi coprocessor is connected to an Intel Xeon processor, also known as the host, through a Peripheral Component Interconnect Express (PCIe) bus, as shown in Fig. 12.2.

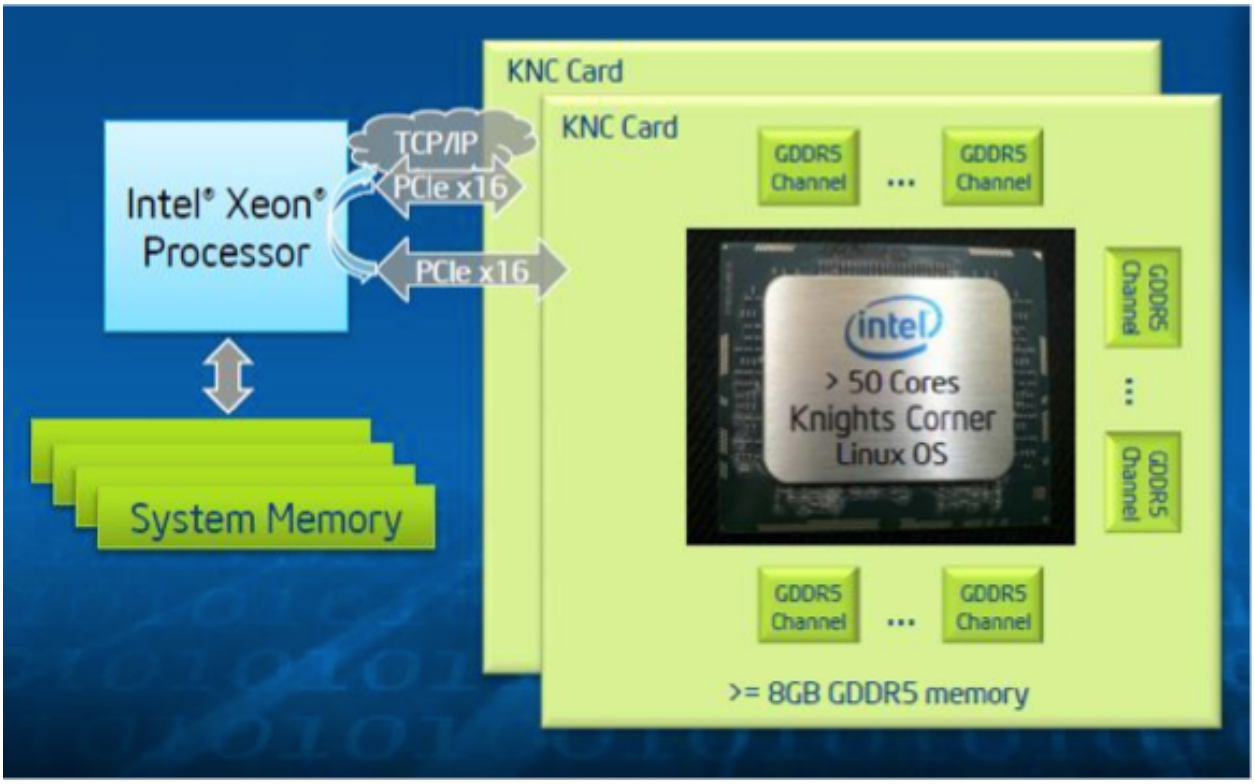


Fig. 12.2: The connection between host and device, taken from <https://software.intel.com/mic-developer>.

Since the Intel Xeon Phi coprocessor runs a Linux operating system, a virtualized TCP/IP stack could be implemented over the PCIe bus, allowing the user to access the coprocessor as a network node. Thus, any user can connect to the coprocessor through a secure shell and directly run individual jobs or submit batchjobs to it. The coprocessor also supports heterogeneous applications wherein a part of the application executes on the host while a part executes on the coprocessor.

Each core in the Intel Xeon Phi coprocessor is designed to be power efficient while providing a high throughput for highly parallel workloads. A closer look reveals that the core uses a short in-order pipeline and is capable of supporting 4 threads in hardware. It is estimated that the cost to support IA architecture legacy is a mere 2 of the area costs of the core and is even less at a full chip or product level. Thus the cost of bringing the Intel Architecture legacy capability to the market is very marginal.

The architecture is shown in greater detail in Fig. 12.3.

The interconnect is implemented as a bidirectional ring, shown in Fig. 12.4.

Each direction is comprised of three independent rings:

1. The first, largest, and most expensive of these is the data block ring. The data block ring is 64 bytes wide to support the high bandwidth requirement due to the large number of cores.

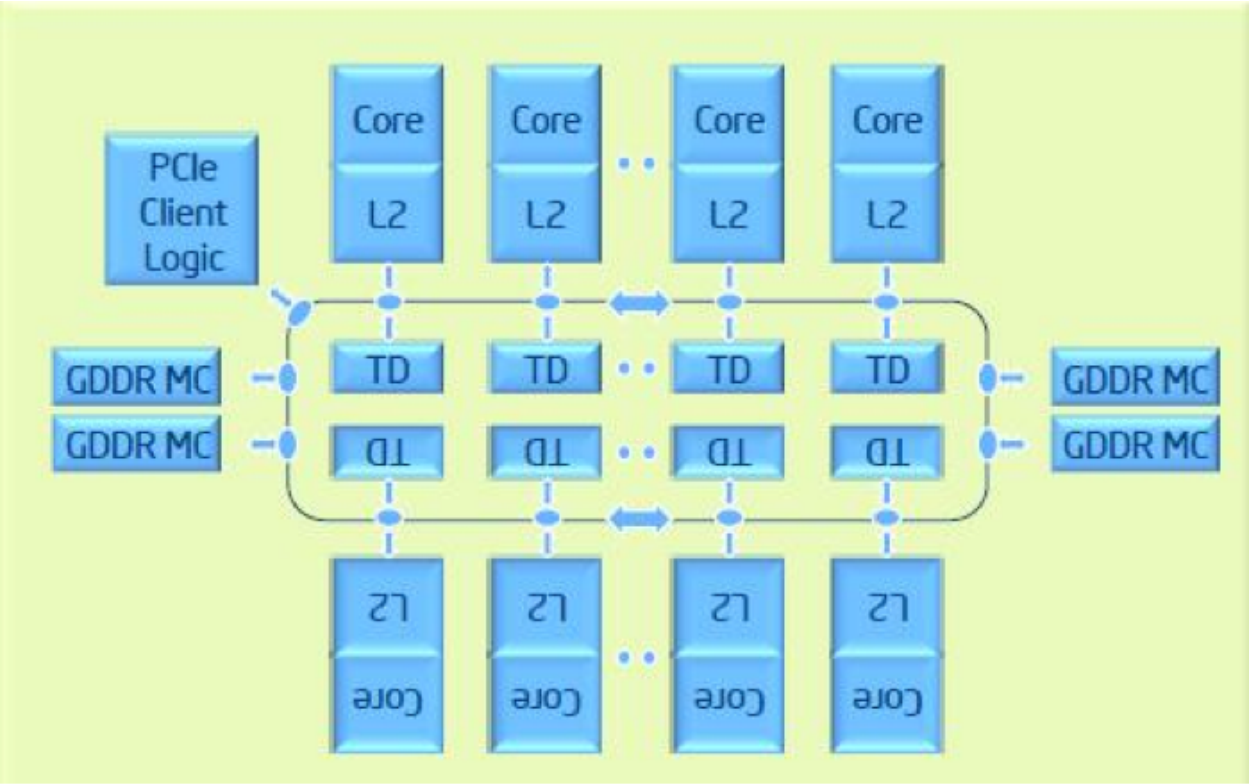


Fig. 12.3: Cores, caches, memory controllers, tag directories, taken from <https://software.intel.com/mic-developer>.

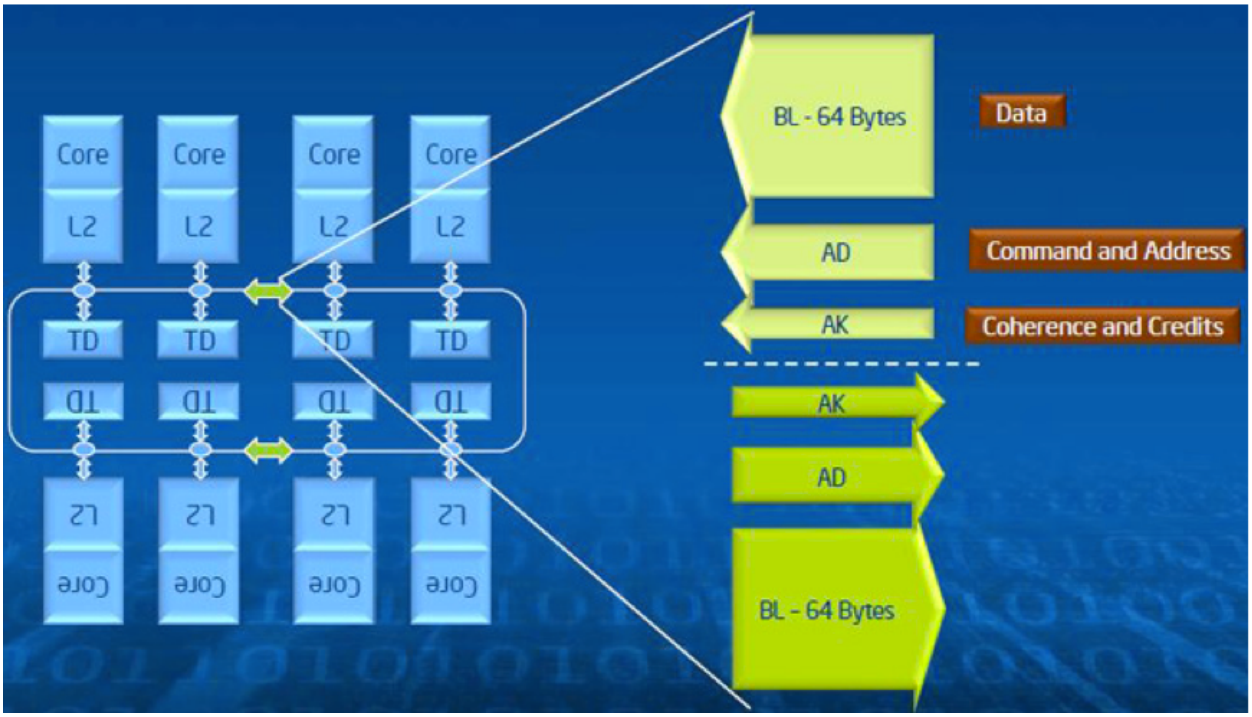


Fig. 12.4: The interconnect, taken from <https://software.intel.com/mic-developer>.

2. The address ring is much smaller and is used to send read/write commands and memory addresses.
3. Finally, the smallest ring and the least expensive ring is the acknowledgement ring, which sends flow control and coherence messages.

One core is shown in Fig. 12.5.

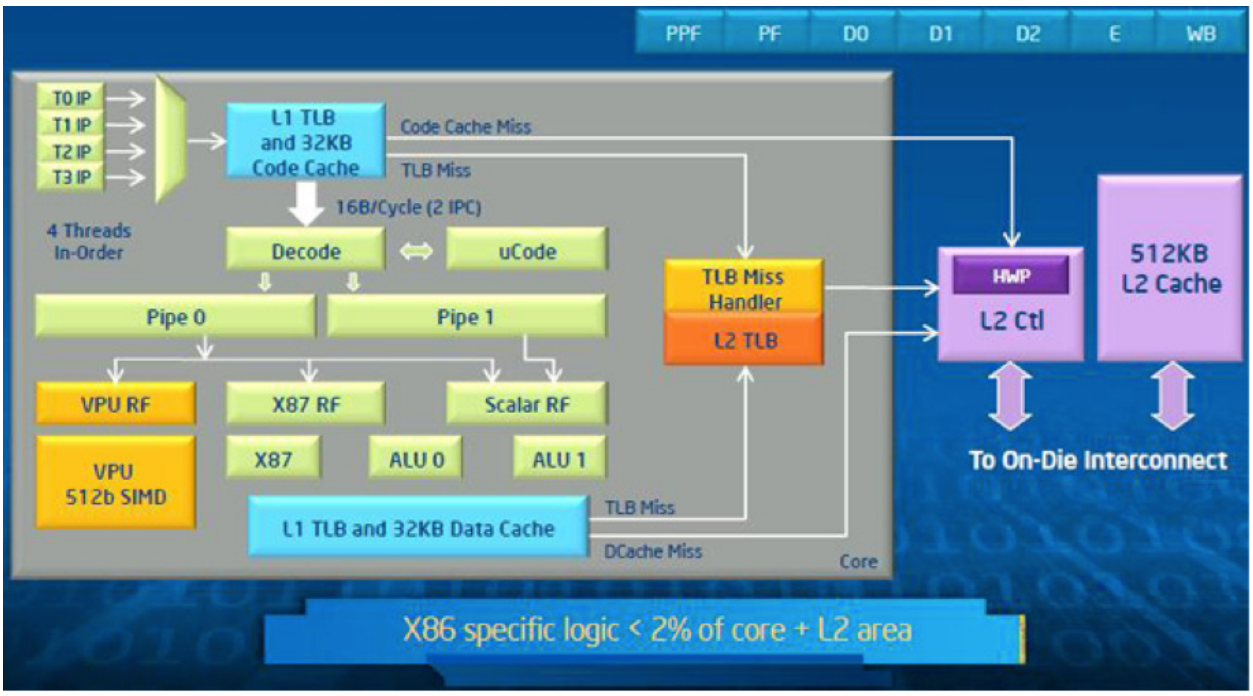


Fig. 12.5: The architecture of one core, taken from <https://software.intel.com/mic-developer>.

An important component of the Intel Xeon Phi coprocessor's core is its Vector Processing Unit (VPU), shown in Fig. 12.6.

The VPU features a novel 512-bit SIMD instruction set, officially known as Intel Initial Many Core Instructions (Intel IMCI). Thus, the VPU can execute 16 single precision (SP) or 8 double precision (DP) operations per cycle. The VPU also supports Fused Multiply Add (FMA) instructions and hence can execute 32 SP or 16 DP floating point operations per cycle. It also provides support for integers.

The theoretical peak performance is 1.01 TFLOPS double precision:

$$1.053 \text{ GHz} \times 60 \text{ cores } \times 16 \text{ DP/cycle} = 1,010.88 \text{ GFLOPS}.$$

Applications that process large vectors may benefit and may achieve great performance on both the Phi and GPU, but notice the fundamental architectural difference between Phi and GPU:

- The Intel Phi has 60 cores each with a vector processing unit that can perform 16 double precision operations per cycle.
- The NVIDIA K20C has 13 streaming multiprocessors with 192 CUDA cores per multiprocessor, and threads are grouped in warps of 32 threads. The P100 has 56 streaming multiprocessors with 64 cores per multiprocessor.

12.1.2 Programming the Intel Xeon Phi Coprocessor with OpenMP

For this class, we have a small allocation on the Texas Advanced Computing Center (TACC) supercomputer: Stampede.

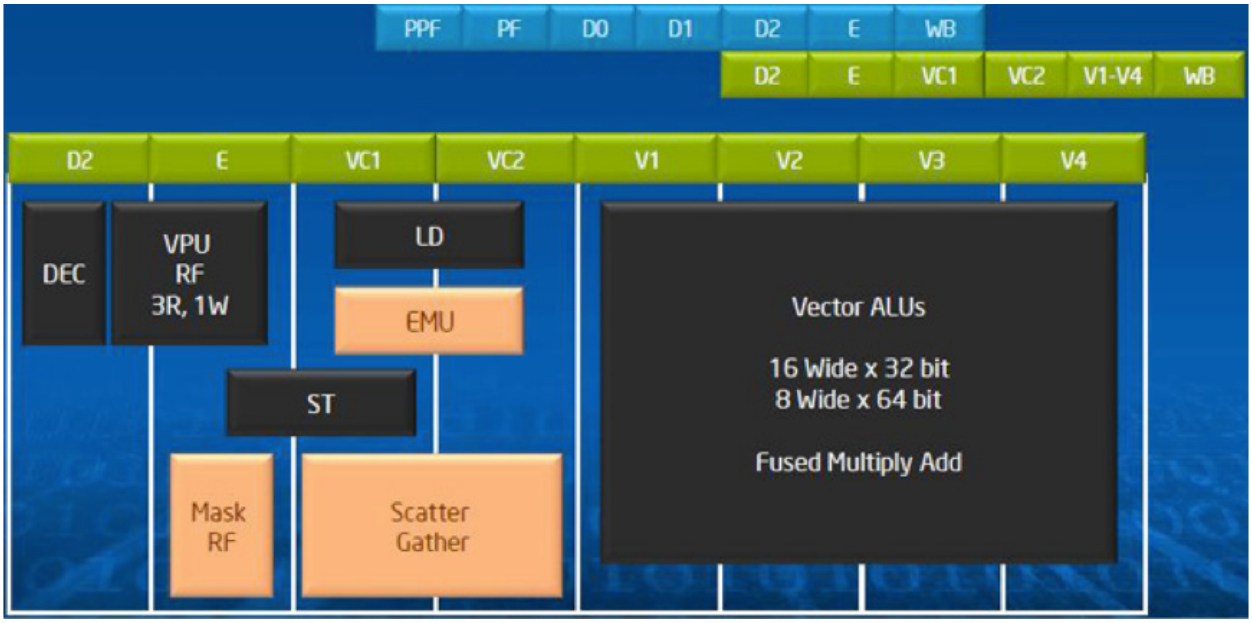


Fig. 12.6: The Vector Processing Unit (VPU), taken from <https://software.intel.com/mic-developer>.

Stampede is a 9.6 PFLOPS (1 petaflops is a quadrillion flops) Dell Linux Cluster based on 6,400+ Dell PowerEdge server nodes. Each node has 2 Intel Xeon E5 (Sandy Bridge) processors and an Intel Xeon Phi Coprocessor. The base cluster gives a 2.2 PF theoretical peak performance, while the co-processors give an additional 7.4 PF.

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

#pragma offload_attribute (push, target (mic))

void hello ( void )
{
   int tid = omp_get_thread_num();
   char *s = "Hello World!";
   printf("%s from thread %d\n", s, tid);
}

#pragma offload_attribute (pop)

int main ( void )
{
   const int nthreads = 1;
   #pragma offload target(mic)
   #pragma omp parallel for
   for(int i=0; i<nthreads; i++) hello();
   return 0;
}
```

In addition to the OpenMP constructs, there are specific pragmas for the Intel MIC:

- To make header files available to an application built for the Intel MIC architecture:

```
#pragma offload_attribute (push, target (mic))

#pragma offload_attribute (pop)
```

- The programmer uses

```
#pragma offload target(mic)
```

  to mark statements (offload constructs) that should execute on the Intel Xeon Phi coprocessor.

An interactive session on Stampede is below.

```
login4$ srun -p development -t 0:10:00 -n 1 --pty /bin/bash -l
------------------------------------------------------------------
              Welcome to the Stampede Supercomputer
------------------------------------------------------------------

--> Verifying valid submit host (login4)...OK
--> Verifying valid jobname...OK
--> Enforcing max jobs per user...OK
--> Verifying availability of your home dir (/home1/02844/janv)...OK
--> Verifying availability of your work dir (/work/02844/janv)...OK
--> Verifying availability of your scratch dir (/scratch/02844/janv)...OK
--> Verifying valid ssh keys...OK
--> Verifying access to desired queue (development)...OK
--> Verifying job request is within current queue limits...OK
--> Checking available allocation (TG-DMS140008)...OK
srun: job 3206316 queued and waiting for resources
srun: job 3206316 has been allocated resources
```

We use the Intel C++ compiler `icc`:

```
c559-402$ icc -mmic -openmp -Wno-unknown-pragmas -std=c99 hello.c
c559-402$ ls -lt
total 9
-rwx------ 1 janv G-815233 11050 Apr 22 16:33 a.out
-rw------- 1 janv G-815233   331 Apr 22 16:21 hello.c
c559-402$ ./a.out
Hello World! from thread 0
c559-402$
```

Because of the `pragma offload_attribute`, the header files are wrapped and we can compile also without the options `-mmic` and `-openmp`:

```
c559-402$ icc -std=c99 hello.c
c559-402$ ./a.out
Hello World! from thread 0
c559-402$ more hello_old.c
c559-402$ exit
logout
login4$
```

Let us consider the multiplication of two matrices.

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

void MatMatMult ( int size,
   float (* restrict A)[size],
   float (* restrict B)[size],
   float (* restrict C)[size] )
{
```

```
   #pragma offload target(mic) \
      in(A:length(size*size)) in(B:length(size*size)) \
      out(C:length(size*size))
   {
      #pragma omp parallel for default(none) shared(C, size)
      for(int i=0; i<size; i++)
         for(int j=0; j<size; j++) C[i][j] = 0.0f;

      #pragma omp parallel for default(none) shared(A, B, C, size)
      for(int i=0; i<size; i++)
         for(int j=0; j<size; j++)
            for(int k=0; k<size; k++)
               C[i][j] += A[i][k] * B[k][j];
   }
}
```

The main function is listed below.

```
int main ( int argc, char *argv[] )
{
   if(argc != 4) {
      fprintf(stderr,"Use: %s size nThreads nIter\n", argv[0]);
      return -1;
   }
   int i, j, k;
   int size = atoi(argv[1]);
   int nThreads = atoi(argv[2]);
   int nIter = atoi(argv[3]);
   omp_set_num_threads(nThreads);
   float (*restrict A)[size] = malloc(sizeof(float)*size*size);
   float (*restrict B)[size] = malloc(sizeof(float)*size*size);
   float (*restrict C)[size] = malloc(sizeof(float)*size*size);

#pragma omp parallel for default(none) shared(A,B,size) private(i,j,k)
   for(i=0; i<size; i++)
      for(j=0; j<size; j++)
      {
         A[i][j] = (float)i + j;
         B[i][j] = (float)i - j;
      }
   double avgMultTime = 0.0;
   MatMatMult(size, A, B, C); // warm up

   double startTime = dsecnd();
   for(int i=0; i < nIter; i++) MatMatMult(size, A, B, C);
   double endTime = dsecnd();

   avgMultTime = (endTime-startTime)/nIter;

#pragma omp parallel
#pragma omp master
   printf("%s nThreads %d matrix %d %d runtime %g GFlop/s %g",
      argv[0], omp_get_num_threads(), size, size,
      avgMultTime, 2e-9*size*size*size/avgMultTime);

#pragma omp barrier
   free(A); free(B); free(C);
   return 0;
```

```
}
```

12.1.3 Bibliography

- George Chrysos: **Intel Xeon Phi Coprocessor - the architecture**. Whitepaper available at <https://software.intel.com/mic-developer>.
- Rob Farber: **CUDA vs. Phi: Phi Programming for CUDA Developers**. *Dr Dobb's Journal.* <http://www.drdobbs.com>
- **Intel Xeon Phi Coprocessor Developer's Quick Start Guide.** Whitepaper available at <https://software.intel.com/mic-developer>.
- Jim Jeffers and James Reinders: **Intel Xeon Phi coprocessor high-performance programming.** Elsevier/MK 2013.
- Rezaur Rahman: **Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers**. Apress 2013.

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

W