WITS
UNIVERSITY

COMS3008A: Parallel Computing

Hairong Wang

May 17, 2024, Johannesburg

Contents

List of Figures . . . iii
List of Tables . . . v
List of Algorithms . . . vi
List of Acronyms . . . vii

1 Introduction **1**
1.1 Parallel computers . . . 1
1.1.1 Parallel Computing — What is it? . . . 1
1.1.2 Why parallelism? . . . 3
1.1.3 A brief history of processor performance . . . 3
1.1.4 A brief history of parallel computing . . . 5
1.2 Supercomputers . . . 7
1.3 Classification of Parallel Computers . . . 9
1.3.1 Control Structure Based Classification - Flynn's Taxonomy . . . 9
1.4 A Quantitative Look at Parallel Computation . . . 11
1.4.1 A simple performance modelling . . . 12
1.4.2 Amdahl's Law . . . 12
1.4.3 Gustafson's Law . . . 13
1.4.4 Efficiency . . . 14
1.4.5 Scalability . . . 14
1.5 Limitations of memory system performance . . . 15
1.6 Exercises . . . 16

Bibliography **18**

2 Overview of Parallel Systems **19**
2.1 Modelling Parallel Computation . . . 19
2.1.1 Random Access Machine . . . 19
2.2 Interconnection Networks . . . 21
2.2.1 Important Factors of Interconnection Networks . . . 23
2.2.2 Some Basic Properties of Interconnection Networks . . . 23
2.2.3 The Classification of Interconnection Networks . . . 23
2.2.4 Popular Interconnection Network Topologies . . . 24
2.2.5 Evaluating the Interconnect Network . . . 27
2.3 Exercises . . . 27

Bibliography **28**

3 Parallel Algorithm Design **30**
3.1 Shared Memory vs Distributed memory 30
3.1.1 Shared Memory Programming 31
3.1.2 Distributed Memory Programming 32
3.2 Parallel Algorithm Design 32
3.2.1 Parallel Algorithm Design — Simple Examples 33
3.3 Summary 36
3.4 Exercises 36

Bibliography **37**

4 Shared Memory Programming using OpenMP **39**
4.1 OpenMP Overview 39
4.1.1 What is OpenMP? 39
4.1.2 OpenMP Programming Model 39
4.1.3 OpenMP API Overview 40
4.1.4 Compiling OpenMP Programs 41
4.2 OpenMP Core Features 42
4.2.1 Parallel Region Construct 42
4.2.2 Example - Computing the π Using Integration in Parallel 43
4.2.3 Synchronization 47
4.2.4 Worksharing in OpenMP 48
4.2.5 Loop Construct 49
4.2.5.1 Examples of More Clauses 55
4.2.5.2 Combined Parallel Worksharing Constructs 56
4.2.5.3 Single Worksharing Construct 56
4.2.5.4 Copyprivate Clause 57
4.2.5.5 Master Construct 57
4.2.6 Sections/Section Construct 58
4.2.7 Task Worksharing Construct 60
4.3 Summary 65
4.4 Exercises 65

Bibliography **70**

5 Distributed Memory Programming using MPI **71**
5.1 Message Passing Model and MPI 71
5.1.1 Message Passing Model 71
5.1.2 MPI 71
5.1.3 The Minimal Set of MPI Functions 72
5.1.4 Compiling and running MPI Programs 74
5.1.5 Installing MPI Standard Implementation 74
5.2 Data Communication 74
5.2.1 Data Communication 74
5.2.2 MPI Blocking Send and Receive 74
5.2.3 MPI Datatypes 76

5.2.4 Benchmarking the Performance 76
5.3 Examples 77
5.4 Collective Communication and Computation Operations 79
5.5 Running MPI Program With MPICH 81
5.5.1 What is MPICH 81
5.5.2 Summary 81
5.6 Point to Point Communication 82
5.6.1 Blocking vs. Non-blocking 82
5.6.2 MPI Message Passing Function Arguments 82
5.6.3 Avoiding Deadlocks 82
5.6.4 Sending and Receiving Messages Simultaneously 84
5.6.5 Overlapping Communication with Computation 85
5.7 Collective Communications 87
5.7.1 MPI Collective Communication Illustrations 88
5.8 Examples 98
5.8.1 Parallel Quicksort Algorithm 98
5.8.2 Hyperquicksort 98
5.8.3 The Odd-Even Transposition Sort 100
5.8.4 Parallel Bitonic Sort 101
5.9 MPI Derived Datatypes 104
5.9.1 Typemap 105
5.9.2 Creating and Using a New Datatype 105
5.9.3 Contiguous Type 105
5.9.4 Vector Type 107
5.9.5 Indexed Type 109
5.9.6 Struct Type 109
5.10 Summary 112
5.11 Exercises 112

Bibliography **114**

List of Figures

1.1 Serial computing 1
1.2 Parallel computing 2
1.3 Parallelization of a vector addition 2
1.4 Intel's shift in chip design to multi-core structure. 3
1.5 Intel single processor performance over time 4
1.6 Moore's Law – transistor count 1971 - 2016 4
1.7 Processor-memory performance gap 5

1.8 Illustration of the die area for integer unit and cache logic for Intel Itanium II. . . 6
1.9 Intel VP Patrick Gelsinger (ISSCC 2001) 6
1.10 Growth of transistors, frequency, cores, and power consumption 7
1.11 Fugaku, a supercomputer from RIKEN and Fujitsu Limited. 158,976 nodes fit into 432 racks, that means 368 nodes in each rack, 48 cores each node. 8
1.12 The SISD architecture 9
1.13 The SIMD architecture 10
1.14 The MIMD architecture 10
1.15 The SMP architecture 10
1.16 The NUMA architecture 11
1.17 The distributed memory architecture 11
1.18 A cluster 12

2.1 RAM model of computation: memory M - contains program instructions and data; processing unit P - execute instructions on data. 19
2.2 (a) PRAM model for parallel computation; (b) multiple PEs try to access the same memory location simultaneously. 20
2.3 (a) LMM; (b) MMM. 22
2.4 (a) Fully connected network; (b) A fully connected crossbar switch. It is non-blocking in the sense that a connection of a PE to a memory does not block another similar connection. It is scalable in terms of performance, but not scalable in terms of cost. 22
2.5 (a) Fully connected network; (b) A fully connected crossbar network. 24
2.6 Bus topology. 24
2.7 (a) Linear array without wraparound link; (b) linear array with wraparound link (also called ring, see next slide). 25
2.8 Ring topology. Each node represents a processing element with local memory. . . 25
2.9 (a) 2D mesh topology; (b) 2D torus topology. Each node represents a processing element with local memory. 25
2.10 (a) 3D mesh, (b) Hypercube topology Each node represents a processing element with local memory. 26
2.11 Hypercubes of 1D, 2D, 3D, and 4D. 26
2.12 Multistage network topology. A 4-stage interconnection network capable of connecting 16 PEs to 16 memory modules. Each switch can establish a connection between a pair of input and output channels. 27
2.13 Fat tree topology. A fattree interconnect network of 16 processing nodes. Each switch can establish a connection between arbitrary pair of incidents. Edges closer to the root are thicker. The idea is to increase the number of communication links and switching nodes closer to the root. 27

3.1 Bus based parallel computers. 30
3.2 (a) Fully connected crossbar switch based system; (b) 2D mesh. 31
3.3 Steps in parallelization, relationship between PEs, tasks and processors. 33
3.4 2D grid points. 34
3.5 Impact of limited concurrency. 35
3.6 Tasks and their communications. 36

3.7 Tasks and communications. 36
3.8 Each row elements depends on the element to the left; each column depends on the previous column. 38
3.9 Update all the red cells first, then update all the black cells. The process is repeated until convergence. 38

4.1 The shared memory model of parallel computation. 39
4.2 Fork-join model 40
4.3 OpenMP parallel construct execution model 41
4.4 Approximating the π using numerical integration 44
4.5 False sharing 45
4.6 Example - cache coherence cont. 46

5.1 The trapezoidal rule: (a) area to be estimated, (b) estimate area using trapezoids . 86
5.2 MPI_Bcast 88
5.3 MPI_Reduce 89
5.4 MPI_Allreduce 90
5.5 (a) A global sum followed by a broadcasting; (2) A butterfly structured global sum. 91
5.6 MPI_Scatter 92
5.7 MPI_Gather 92
5.8 MPI_Allgather 93
5.9 MPI_Alltoall 94
5.10 MPI_Scan 95
5.11 `MPI_Scatter` (top) and `MPI_Scatterv` (bottom) example 96
5.12 `MPI_Alltoallv` example 97
5.13 `MPI_Alltoallv` example 97
5.14 Parallel quicksort 98
5.15 Illustration of Hyperquicksort algorithm 99
5.16 A schematic representation of a network that converts an input sequence into a bitonic sequence. In this example, $\oplus$BM[k] and $\ominus$BM[k] denote bitonic merging networks of input size k that use $\oplus$ and $\ominus$ comparators, respectively. The last merging network ($\oplus$BM[16]) sorts the input. In this example, $n = 16$. 103
5.17 A bitonic merging network for $n = 16$. The input wires are numbered $0, 1 \ldots, n-1$, and the binary representation of these numbers is shown. Each column of comparators is drawn separately; the entire figure represents a $\oplus$BM[16] bitonic merging network. The network takes a bitonic sequence and outputs it in sorted order. 103
5.18 The comparator network that transforms an input sequence of 16 unordered numbers into a bitonic sequence. 104
5.19 MPI_Type_vector 108
5.20 MPI_Type_indexed 110

List of Tables

1.1 No 1 Supercomputer in 11/2022 & 11/2023 — Frontier 7
1.2 No 1 Supercomputer in 11/2020 & 11/2021 — Fugaku 8

2.1 Quantitative characteristics of various interconnect networks 28

4.1 Example - cache coherence 46
4.2 Associative operands that can be used with reduction (for C/C++) and their common initial values 53
4.3 Loops in canonical form 56
4.4 Table for benchmarking the performance of parallel *quicksorts* 69

5.1 Integers from 2 to 60 79
5.2 Sieve = 2 79
5.3 Sieve = 3 79
5.4 Sieve = 5 80
5.5 Sieve = 7 80
5.6 Predefined reduction operators 80
5.7 MPI point-to-point communication functions 82
5.8 Point-to-point communication example 83
5.9 Point-to-point communication example 83
5.10 Understanding MPI_Reduce 90
5.11 Merging a 16-element bitonic sequence through a series of $\log 16$ bitonic splits. . 102

List of Algorithms

Chapter 1

Introduction

Some contents in this chapter are based on [Grama et al. 2003] and [Trobec et al. 2018].

1.1 Parallel computers

1.1.1 Parallel Computing — What is it?

- Parallel Computer: A parallel computer is a computer system that uses multiple processing elements simultaneously in a cooperative manner to solve a computational problem.
- Parallel computing (or processing): Parallel processing includes techniques and technologies that make it possible to compute in parallel.
  - Hardware, networks, operating systems, parallel libraries, languages, compilers, algorithms, tools etc.
- Parallel computing is an evolution of serial computing.
  - Parallelism is natural.
  - Computing problems differ in level or type of parallelism.

Serial computing

- A problem is broken into a discrete series of instructions;
- Instructions are executed sequentially one after another;
- Executed on a single processor;
- Only one instruction may execute at any moment in time.
- An illustrative example is given in Figure 1.1.

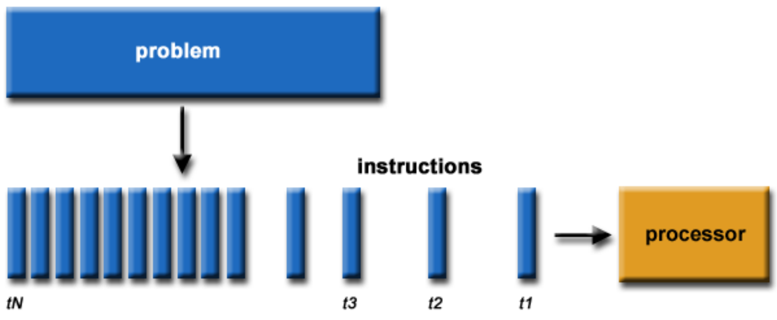


Figure 1.1: Serial computing

Parallel computing

- A problem is broken into discrete parts that can be solved concurrently;
- Each part is further broken down to a series of instructions;
- Instructions from each part execute simultaneously on different processors;
- An overall control/coordination mechanism is employed.
- See an illustrative example in Figure 1.2.

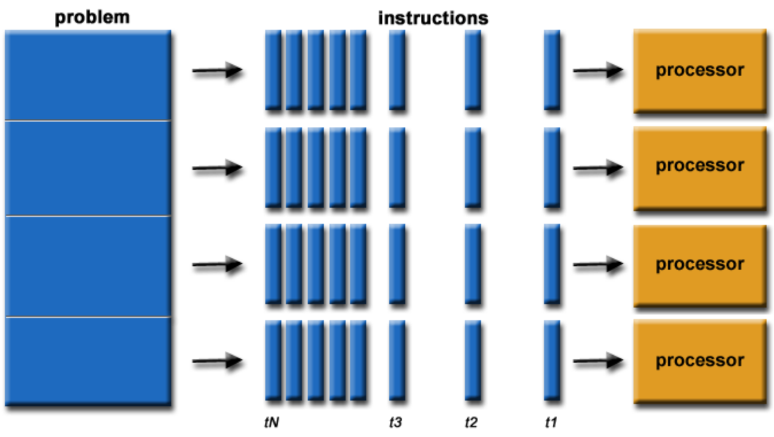


Figure 1.2: Parallel computing

An example of parallelizing an addition of two vectors is shown in Figure 1.3. The computation in this example is element-wise summation of two vectors. Note that the additions of two corresponding vector elements are independent of each other, so they can be executed in parallel. We often call this kind of easily parallelizable problem embarrassingly parallel problem. One of the

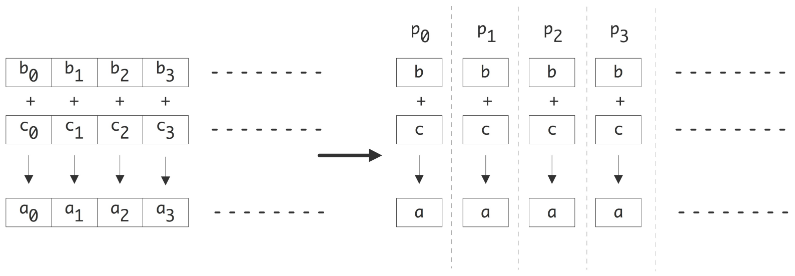


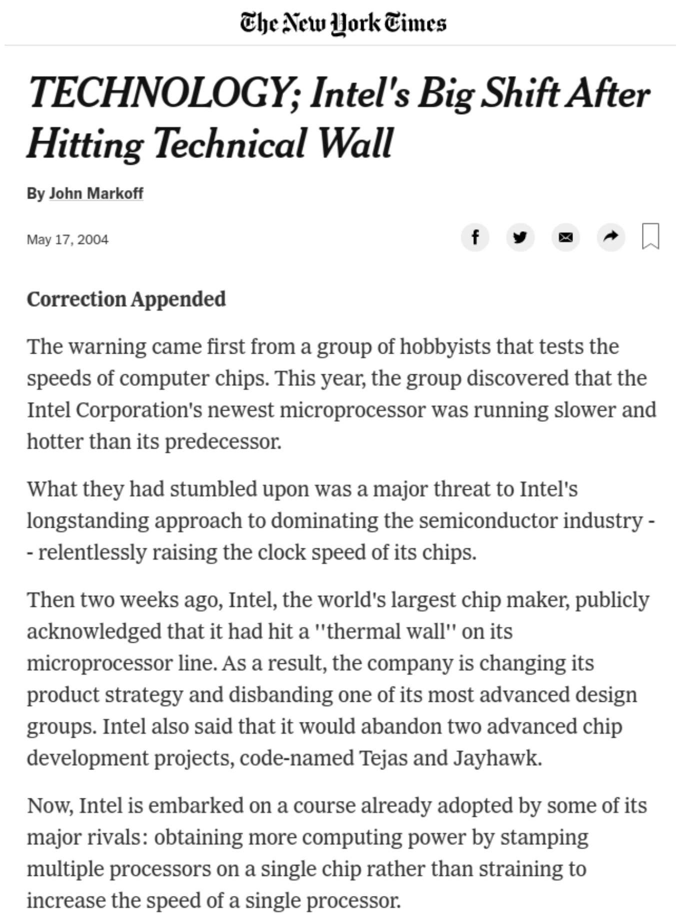
Figure 1.3: Parallelization of a vector addition

main goals of parallel computing is speedup – time on 1 processor / time on p processors. Others include high efficiency (cost, area, power) or working on bigger problems that can't fit on a single machine.

Question: How do you understand concurrency and parallelism?

1.1.2 Why parallelism?

- In 2004, Intel changed its course from traditional chip design approach (single core processor) to embrace "dual core" processor structure [1] (see Figure 1.4).



The New York Times

TECHNOLOGY; Intel's Big Shift After Hitting Technical Wall

By John Markoff

May 17, 2004

Correction Appended

The warning came first from a group of hobbyists that tests the speeds of computer chips. This year, the group discovered that the Intel Corporation's newest microprocessor was running slower and hotter than its predecessor.

What they had stumbled upon was a major threat to Intel's longstanding approach to dominating the semiconductor industry -- relentlessly raising the clock speed of its chips.

Then two weeks ago, Intel, the world's largest chip maker, publicly acknowledged that it had hit a ''thermal wall'' on its microprocessor line. As a result, the company is changing its product strategy and disbanding one of its most advanced design groups. Intel also said that it would abandon two advanced chip development projects, code-named Tejas and Jayhawk.

Now, Intel is embarked on a course already adopted by some of its major rivals: obtaining more computing power by stamping multiple processors on a single chip rather than straining to increase the speed of a single processor.

Figure 1.4: Intel's shift in chip design to multi-core structure.

1.1.3 A brief history of processor performance

- What had been happening before multi-processors? – Single-processor machines: they had been getting exponentially faster (see Figure 1.5).
- Wider data paths
  - 4 bit → 8 bit → 16 bit → 32 bit → 64 bit
- More efficient pipelining
  - For example, from 3.5 cycles per instruction (CPI) → 1.1 CPI
- Exploiting instruction level parallelism (ILP)

[1] https://www.nytimes.com/2004/05/17/business/technology-intel-s-big-shift-after-hitting-technical-wall.html

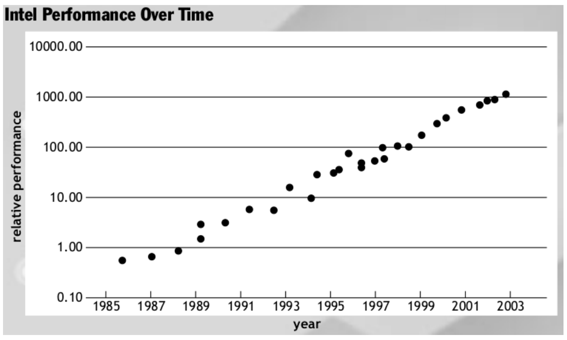


Figure 1.5: Intel single processor performance over time

- “ Superscale” processing: e.g., issues up to 4 instructions/cycle

- Faster clock rates
    - e.g., 10MHz $\rightarrow$ 100MHz $\rightarrow$ 1GHz $\rightarrow$ 3GHz

During this time, that is 80s and 90s in last century, computers had large exponential performance gains — such performance gain is often measured by the Moore’s Law.

Question: What is Moore’s Law?

- “The number of transistors on an integrated circuit doubles every two years.” — Gordon E. Moore

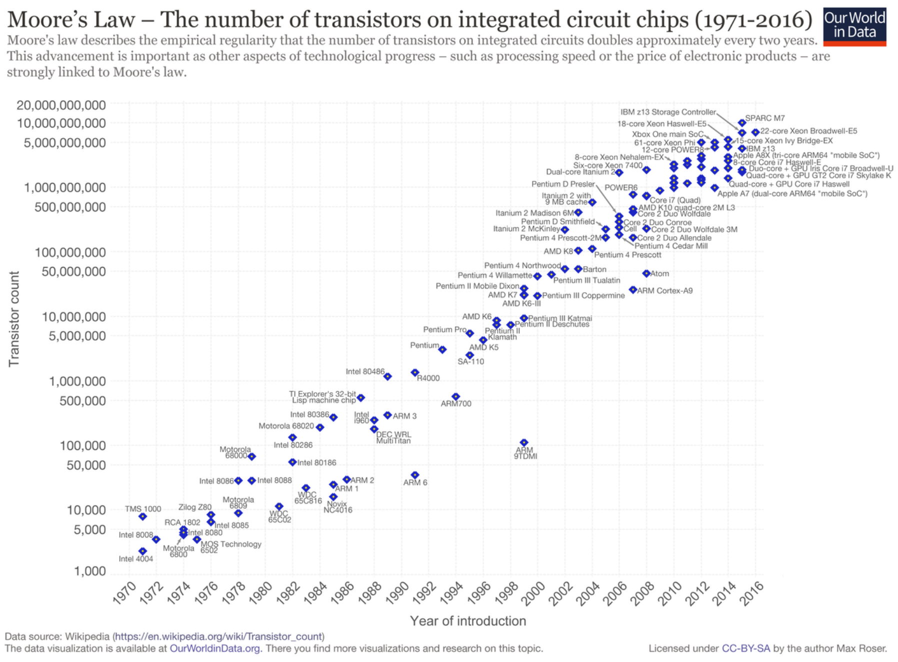


Figure 1.6: Moore’s Law – transistor count 1971 - 2016

Intel launched the first commercial microprocessor chip, the Intel 4004, with 2300 transistors, each the size of a red blood cell. Since then, chips have improved in line with the prediction of Gordon Moore, Intel's co-founder (see Figure 1.6).

Question: How fast is this growth? How do we relate it to the physical world?

An issue comes with this is that when the performance of the processor increases exponentially, the memory speed hasn't increased at the same rate, instead it has grown very slowly compared to the CPU performance (see Figure 1.7). This caused the growing performance gap between the processor and memory. This means that the memory is just not fast enough to meet the demand of the processor. So the actual performance of solving a problem could be limited by the memory performance instead of the faster processor performance. One way to address this

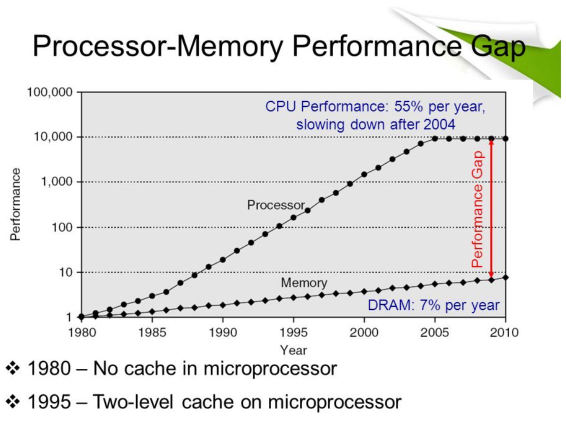


Figure 1.7: Processor-memory performance gap

performance gap is to add faster memory on the processor chip so that the data and the processor are physically close, that means faster data access. we know such on chip memory is cache. The problem with this is that the cache size is usually very small, because the chip area is limited.

- For example, Intel Itanium II (see Figure 1.8)
  - 6-way integer unit $<$ 2% die area;
  - Cache logic $>$ 50% die area.
- Most of chip there to keep these 6 integer units at 'peak' rate.
- Main issue is external DRAM latency (50ns) to internal clock rate (0.25ns) ratio is 200:1.

1.1.4 A brief history of parallel computing

The growing performance gap between processor and memory shows that it is not a good strategy to follow the path of increasing the clock speed of a microprocessor further on. Besides the performance gap, another serious issue is that, in the past, processing chip manufacturers increased processor performance by increasing CPU clock frequency, until the chips got too hot! This mass of billions of tiny transistors flipping on and off generate immense heat. See Figure 1.9.

- Greater clock frequency, greater electrical power
- Intel VP Patrick Gelsinger (ISSCC 2001): "If scaling continues at present pace, by 2005, high speed processors would have power density of nuclear reactor, by 2010, a rocket noz-

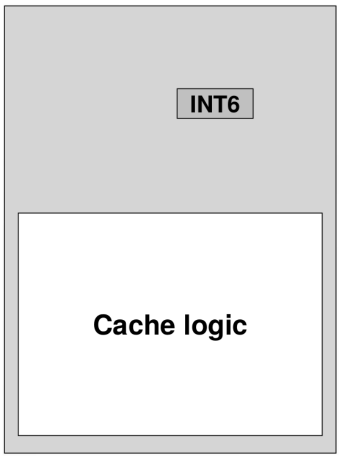


Figure 1.8: Illustration of the die area for integer unit and cache logic for Intel Itanium II.

zle, and by 2015, surface of sun." – Figure 1.9

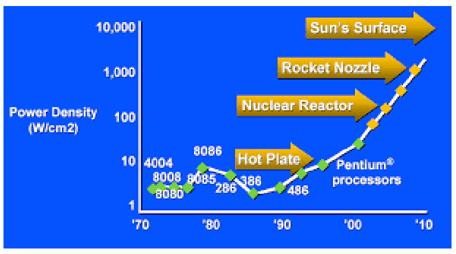


Figure 1.9: Intel VP Patrick Gelsinger (ISSCC 2001)

- What can we do? We can still pack more and more transistors onto a die, but we cannot make the individual transistor faster.
- We have to make better use of those more and more transistors to increase the performance instead of making only the clock speed faster. One solution is to use parallelism.
- Add multiple cores to add performance, keep clock frequency the same or reduced.

Why parallelism – summary

- Serial machines have inherent limitations:
  - Processor speed, memory bottlenecks, ...
- Parallelism has become the future of computing.
- Two primary benefits of parallel computing:
  - Solve fixed size problem in shorter time
  - Solve larger problems in a fixed time
- Other factors motivate parallel processing:
  - Effective use of machine resources;
  - Cost efficiencies;

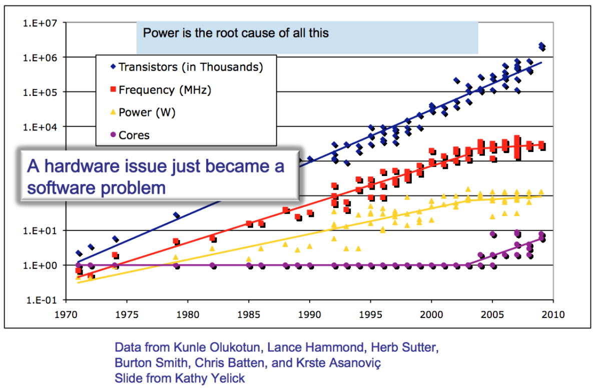


Figure 1.10: Growth of transistors, frequency, cores, and power consumption

  - Overcoming memory constraints.
- Performance is still the driving concern.
- Technology push
- Application pull
  - Application performance demands hardware advances;
  - Hardware advances generate new applications;
  - New applications have greater performance demands.

1.2 Supercomputers

See Table 1.1 for Frontier, the No 1 Supercomputer in 11/2022 & 11/2023. Table 1.2 and Figure 1.11 show Fugaku, the No 1 Supercomputer in 11/2020 & 11/2021. Supercomputers are used to run applications in areas such as drug discovery; personalized and preventive medicine; simulations of natural disasters; weather and climate forecasting; energy creation, storage, and use; development of clean energy; new material development; new design and production process; and-as a purely scientific endeavour-elucidation of the fundamental laws and evolution of the universe. Some performance measures used:

Table 1.1: No 1 Supercomputer in 11/2022 & 11/2023 — Frontier

Site	DOE/SC/Oak Ridge National Laboratory
Manufacturer	HPE (Frontier; Frontier URL)
Cores[2]	8,730,112
Processor	AMD Optimized 3rd Generation EPYC 64C 2GHz
Interconnect	Slingshot-11
Linpack Performance (Rmax)	1,102.00 PFlop/s
Theoretical Peak (Rpeak)	1,685.65 PFlop/s
Nmax	24,440,832
Power	21,100.00 kW (Submitted)
Operating System	HPE[3] Cray OS

Table 1.2: No 1 Supercomputer in 11/2020 & 11/2021 — Fugaku

Site	RIKEN Center for Computational Science
Manufacturer	Fujitsu (Fugaku; Fugaku virtual tour)
Cores	7,630,848
Linpack Performance (Rmax)	442,010 TFlop/s
Theoretical Peak (Rpeak)	537,212 TFlop/s
HPCG[4]	16,004.5 TFlop/s
Nmax	21,288,960
Power	29,899.23 kW (Submitted)
Memory	5,087,232 GB
Processor	A64FX 48C 2.2GHz
Interconnect	Tofu interconnect D
Operating System	RHEL
Compiler	Fujitsu software technical computing suite v4.0
Math library	Fujitsu software technical computing suite v4.0
MPI	Fujitsu software technical computing suite v4.0

Rmax Maximal LINPACK performance achieved. The benchmark used in the LINPACK Benchmark is to solve a dense system of linear equations. The LINPACK Benchmark is a measure of a computer's floating-point rate of execution.

Rpeak Theoretical peak performance. The theoretical peak is based not on an actual performance from a benchmark run, but on a paper and pen computation to determine the theoretical peak rate of execution of floating point operations for the machine. For example, an Intel Itanium 2 at 1.5 GHz can complete 4 double-precision floating point operations per cycle (or per clock tick). Then its theoretical peak performance is $4 \times 1.5 \times 10^9 = 6 \times 10^9$Flops or $=$ 6GFlops.

Nmax Problem size for achieving Rmax

N1/2 Problem size for achieving half of Rmax

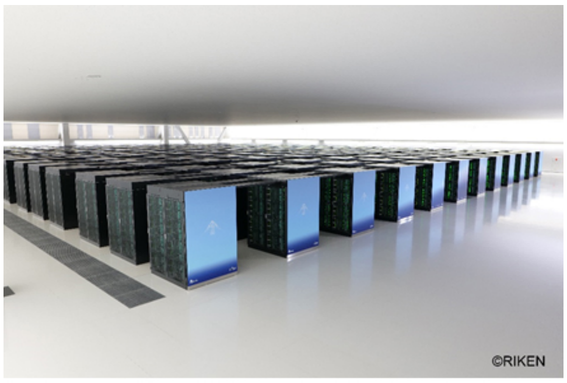


Figure 1.11: Fugaku, a supercomputer from RIKEN and Fujitsu Limited. 158,976 nodes fit into 432 racks, that means 368 nodes in each rack, 48 cores each node.

As of 2023, the clusters (or partitions) at Wits Mathematical Sciences Lab (MSL)

[2]9408 CPUs and 37632 GPUs

[3]Hewlett Packard Enterprise

[4]High performance conjugate gradient

- stampede: For general purpose use or jobs that can leverage InfiniBand (if enabled). It has 40 nodes, each with two Xeon E5-2680 CPUs, two GTX1060 GPUs (6GB per GPU, 12GB per node), and 32GB of system RAM. `MaxTime = 4320` and `MaxNodes = 16`.
- batch: For bigger jobs that can leverage a bigger GPU and additional system RAM. It has 48 nodes each with a single Intel Core i9-10940X CPU (14 cores), NVIDIA RTX3090 GPU (24GB), and 128GB of system RAM. `MaxTime = 4320` and `MaxNodes = 16`. Partition batch has the potential for 10Gb networking (This has not yet been implemented).
- biggpu: For mature code that can meaningfully leverage very large amounts of GPU and system RAM. It has 38 nodes, each node has two Intel Xeon Platinum 8280L CPUs (28 cores per CPU, 56 cores per node) with two NVIDIA Quadro RTX8000 GPUs (48GB per GPU, 96GB per node), and 1TB of system RAM. `MaxTime = 4320` and `MaxNodes = 3`. biggpu has the potential for 10Gb networking (This has not yet been implemented). Whenever possible, limit jobs on biggpu to using only a single node so others can use the partition simultaneously. Please use biggpu responsibly!

1.3 Classification of Parallel Computers

1.3.1 Control Structure Based Classification - Flynn's Taxonomy

Flynn's taxonomy is widely used since 1966 for classification of parallel computers. The classification is based on two independent dimensions of *instruction stream* and *data stream* with two possible states: *single* or *multiple*.

- SISD: Single instruction stream single data stream. This is the traditional CPU architecture: at any one time only a single instruction is executed, operating on a single data item.

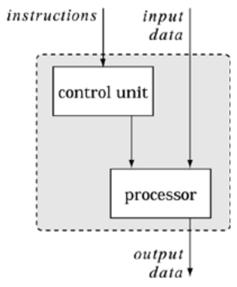


Figure 1.12: The SISD architecture

- SIMD: Single instruction stream multiple data stream. In this computer type there can be multiple processors, each operating on its own data item, but they are all executing the same instruction on that data item. SIMD computers excel at operations on arrays, such as

$$for(i = 0; i < N; i++) \quad a[i] = b[i] + c[i];$$

- MISD: Multiple instruction stream single data stream. Each processing unit executes different instruction streams on a single data stream. Very few computers are in this type.
- MIMD: Multiple instruction stream multiple data stream. Multiple processors operate on

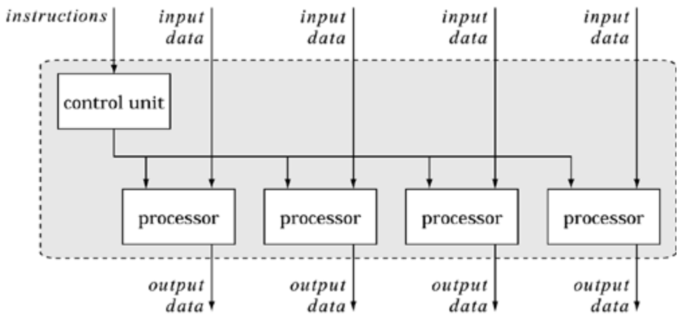


Figure 1.13: The SIMD architecture

multiple data items, each executing independent, possibly different instructions. Most current parallel computers are of this type.

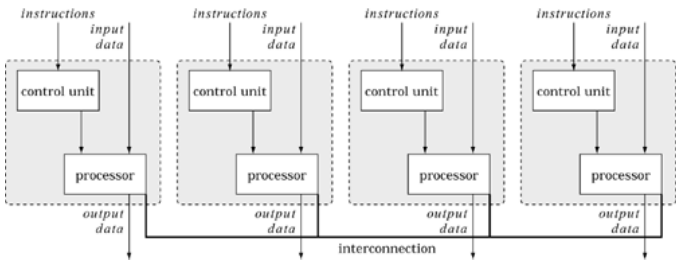


Figure 1.14: The MIMD architecture

Most of MIMD machines operate in *single program multiple data* (SPMD) mode, where the programmers starts up the same executable on the parallel processors. The MIMD category is typically further decomposed according to memory organization: shared memory and distributed memory.

Shared memory: In a shared memory system, all processes share a single address space and communicate with each other by writing and reading shared variables.

- One class of shared-memory systems is called SMPs(symmetric multiprocessors) (see Figure 1.15). All processors share a connection to a common memory and access all memory

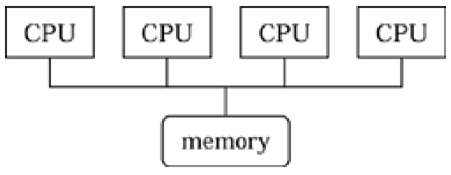


Figure 1.15: The SMP architecture

locations at equal speeds. SMP systems are arguably the easiest parallel systems to program because programmers do not need to distribute data structures among processors.

- The other main class of shared-memory systems is called non-uniform memory access (NUMA) (Figure 1.16). The memory is shared, it is uniformly addressable from all processors, but some blocks of memory may be physically more closely associated with some processors than others.
- To mitigate the effects of non-uniform access, each processor has a cache, along with a protocol to keep cache entries coherent — cache-coherent non-uniform memory access systems (ccNUMA). Programming ccNUMA system is the same as programming an SMP logically, but to obtain the best performance, a programmer needs to be more careful about locality issues and cache effects.

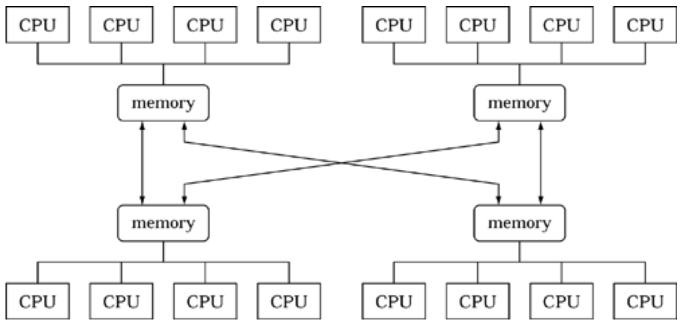


Figure 1.16: The NUMA architecture

Distributed memory systems: Each process has its own address space and communicates with other processes by message passing (sending and receiving messages). Figure 1.17 shows an example. As off-the-shelf networking technology improves, systems of this type are becoming more common and much more powerful.

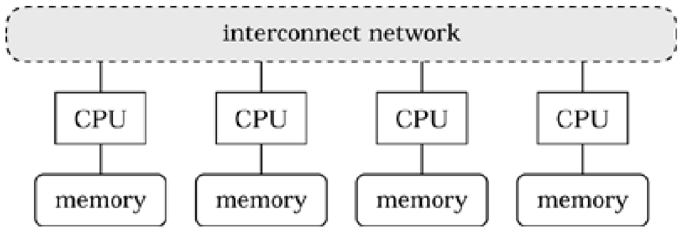


Figure 1.17: The distributed memory architecture

Clusters Clusters are distributed memory systems composed of off-the-shelf computers connected by an off-the-shelf network (Figure 1.18).

1.4 A Quantitative Look at Parallel Computation

Two main reasons for parallel computing are to obtain better performance and to solve larger problems faster. Performance can be both modelled and measured. Let's look at parallel computations by giving some simple analytical models that illustrate some of the factors that influence the performance of a parallel program.

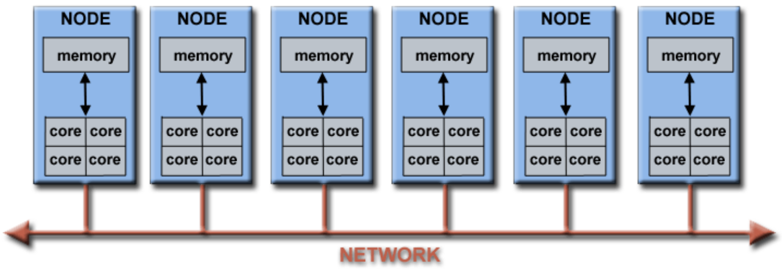


Figure 1.18: A cluster

1.4.1 A simple performance modelling

Consider a computation consisting of three parts: a setup section, a computation section, and a finalization section. The idea that computations have a serial part (for which additional PEs are useless) and a parallelizable part (for which more PEs decrease the running time) is realistic. The total running time of this program on one processing element (PE) is given as:

$$T_{total}(1) = T_{setup} + T_{compute} + T_{finalization} \tag{1.1}$$

What happens when we run this computation on a parallel computer with multiple PEs?

$$T_{total}(P) = T_{setup} + \frac{T_{compute}(1)}{P} + T_{finalization} \tag{1.2}$$

Speedup An important measure of how much additional PEs help is the relative speedup S, which describes how much faster a problem runs:

$$S(P) = \frac{T_{total}(1)}{T_{total}(P)} \tag{1.3}$$

A related measure is the efficiency E, which is the speedup normalized by the number of PEs.

$$E(P) = \frac{S(P)}{P} = \frac{T_{total}(1)}{PT_{total}(P)} \tag{1.4}$$

Ideally, we would want the speedup to be equal to P, the number of PEs. This is sometimes called perfect linear speedup. Unfortunately, perfect linear speedup is rarely the case due to times for setup and finalization are not improved by adding more PEs.

1.4.2 Amdahl's Law

- The terms that cannot be run concurrently are called the serial terms. Their running times represent some fraction of the total, called the serial fraction, denoted γ.

$$\gamma = \frac{T_{setup} + T_{finalization}}{T_{total}(1)} \tag{1.5}$$

- The fraction of time for parallelizable part is then $1 - \gamma$. The total computation time with P number of PEs becomes

$$T_{total}(P) = \gamma T_{total}(1) + \frac{(1-\gamma)T_{total}(1)}{P} \tag{1.6}$$

In the following, let's have a look at some analyses on the limit of parallel processing.

- Then we obtain the well-known Amdahl's law:

$$S(P) = \frac{T_{total}(1)}{T_{total}(P)} = \frac{T_{total}(1)}{(\gamma + \frac{1-\gamma}{P})T_{total}(1)} = \frac{1}{\gamma + \frac{1-\gamma}{P}} \tag{1.7}$$

  Taking the limit as P goes to infinity in Eq. 1.7,

$$S(P) = \frac{1}{\gamma} \tag{1.8}$$

  Eq. 1.8 gives an upper bound on the speedup.

- Amdahl's Law says the speedup you can achieve is limited by the fraction for serial computation in your problem (i.e., the number of PEs does not determine the upper bound of speedup you can achieve).

Example 1.4.1. *Suppose we are able to parallelize* 90% *of a serial program. Further suppose the speedup of this part is* P*, the number of processes we used (which is a perfect linear speedup). If the serial time is* $T_{serial} = 20s$*, then the runtime of the parallelized part is* $(0.9 \times T_{serial})/P = 18/P$*. The runtime of unparallelized part is* $0.1 \times 20 = 2s$*. The overall parallel runtime will be*

$$T_{parallel} = 18/P + 2,$$

and the speedup will be

$$S = \frac{T_{serial}}{T_{parallel}} = \frac{20}{18/p + 2}.$$

As P *gets larger,* $18/P$ *gets close to 0, and the denominator gets close to 2, in turn,* S *gets close to 10. That means* $S \leq 10$*, no matter how many number of processes you use in your program.*

So, what are we doing here? Should we give up? of course no. There is a number of reasons we should not worry about Amdahl's Law too much. First, it does not take into account the problem size. When we increase the problem size, the serial fraction of the problem decreases in size; Second, there are many scientific and engineering applications routinely achieve huge speedups.

1.4.3 Gustafson's Law

- In contrast to Eq. 1.2, we obtain $T_{total}(1)$ from the serial and parallel parts when executed on P PEs.

$$T_{total}(1) = T_{setup} + PT_{compute}(P) + T_{finalization} \tag{1.9}$$

- Now, we define the so-called scaled serial fraction, denoted γ_{scaled}, as

$$\gamma_{scaled} = \frac{T_{setup} + T_{finalization}}{T_{total}(P)}, \tag{1.10}$$

  and then

$$T_{total}(1) = \gamma_{scaled}T_{total}(P) + P(1 - \gamma_{scaled})T_{total}(P). \tag{1.11}$$

The exercise underlying Amdahl's law, namely running exactly the same problem with varying numbers of processors, is artificial in some circumstances. In Eq. 1.2, we obtained the $T_{total}(P)$ from the execution time of serial part and the execution time of parallel part when executed on one PE.

- Using Eq. 1.11, we obtain the scaled speedup, sometimes known as Gustafson's law.

$$\begin{aligned} S(P) &= \frac{T_{total}(1)}{T_{total}(P)} = \frac{\gamma_{scaled}T_{total}(P) + P(1-\gamma_{scaled})T_{total}(P)}{T_{total}(P)} \\ &= P(1-\gamma_{scaled}) + \gamma_{scaled} = P + (1-P)\gamma_{scaled}. \end{aligned} \tag{1.12}$$

- Suppose we take the limit in P while holding $T_{compute}$ and γ_{scaled} constant. That is, we are increasing the size of the problem so that the total running time remains constant when more processors are added. In this case, the speedup is linear in P.

The assumption above implies that the execution time of the serial parts does not change as the problem size grows.

Example 1.4.2. *Now, for the previous example, let's assume the scaled serial fraction of the same problem is* 0.1*, that is* $\gamma_{scaled} = 0.1$*, and* $p = 16$*. Then the scaled speedup is* $S = 16(1-0.1) + 0.1 = 14.5$*, which is greater than the upper bound determined by Amdahl's Law (10).*

1.4.4 Efficiency

- There are another two performance metrics we frequently use in the process of this course: efficiency and scalability.
- Efficiency:

$$E = \frac{S}{P}, \tag{1.13}$$

  where S is the speedup, and P is the number of processes used to achieve the speedup. Note $0 < E \leq 1$. Efficiency is better when E is closer to 1, which simply means you utilized the P processors efficiently.

1.4.5 Scalability

- Scalability: In general, a technology is scalable if it can handle ever-increasing problem size.
- In parallel program performance, scalability refers to the following measure. Suppose we run a parallel program using certain number of processors and with a certain problem size, and obtained an efficiency E. Further suppose that now we want to increase the number of processors. In this case, if we can find a rate at which the problem size increases so that we can still maintain the efficiency E, we say the parallel program scalable.

Example 1.4.3. *Suppose* $T_{serial} = n$*, where* n *is also the problem size. Also suppose* $T_{parallel} = n/p + 1$*. Then*

$$E = \frac{n}{p(n/p+1)} = \frac{n}{n+p}.$$

To see if the problem is scalable, we increase p *by a factor of* k*, then we get*

$$E = \frac{n}{n+p} = \frac{xn}{xn+kp}.$$

If $x = k$, then we have the same efficiency. That is, if we increase the problem size at the same rate that we increase the number of processors, then the efficiency remain constant, hence, the program is scalable.

Strong scalability: When we increase the number of processes, we can keep the efficiency fixed without increasing the problem size, the program is strongly scalable.

Weak scalability: If we can keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes, then the program is said to be weakly scalable.

1.5 Limitations of memory system performance

The effective performance of a program on a computer relies on the speed of the processor but also on the ability of the memory system to feed data to the processor.

1. **Latency:** A memory system, possibly with multiple levels of cache, takes in a request for a memory word and returns a block of data of size b containing the requested word after $l$ns. Here, l is referred to as the latency of the memory.

2. Effect of memory latency on performance

   Example 1.5.1. *Consider a processor operating at 1GHz (1ns clock) connected to a DRAM with a latency of 100ns (no caches). Assume that the processor has two multiply-add units and is capable of executing four instructions in each cycle of 1ns. The peak performance rating is therefore 4GFLOPS. However, since the memory latency is 100ns, and if the block size is only one word (4 bytes), then every time a memory request is made, the processor must wait 100ns (latency) before it can process a word. In the case of each floating point operation requires one data fetch, the peak speed of this computation is limited to one floating point operation every 100ns, or a speed of 10MFLOPS, which is only a small fraction of the peak processor performance.*

3. Improving effective memory latency using caches: Caches are used to address the speed mismatch between the processor and memory. Caches are a smaller and faster memory that is placed between the processor and the DRAM. The data needed by the processor is first fetched into cache. All subsequent accesses to data items residing in the cache are serviced by the cache. Thus, in principle, if a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache.

4. **Cache hit ratio:** The fraction of data references satisfied by the cache is called the cache hit ratio of the computation on the system.

5. **Memory bound computation:** The effective computation rate of many applications is bounded not by the processing rate of the CPU, but by the rate at which data can be pumped into the CPU. Such computations are referred to as being memory bound. The performance of memory bound program is critically impacted by the cache hit ratio.

   Example 1.5.2. *We still consider the 1GHz processor with a 100ns latency DRAM as in Example 1.5.1. In this case, let's consider adding a cache of size 32KB ($1K = 2^{10} \approx 10^3$) with a latency of 1ns. We use this setup to multiply two matrices A and B of dimensions*

32×32 *(for this, the input matrices and output matrix can all fit in the cache). Fetching the two input matrices (2K words) into the cache takes approximately 200*μs*. The total floating point operations in multiplying the two matrix is 64K (multiplying two* $n \times n$ *matrices takes* $2n^3$ *multiply and add operations), which takes approximately 16K cycles, i.e., 16Kns (or 16*μs*) at 4 instructions per cycle. So, the total time for the computation is* $200\mu s + 16\mu s = 216\mu s$*. This corresponds to a peak computation rate of* $64K/216\mu s \approx 303$*MFLOPS. Compared to Example 1.5.1, this is a much better rate (due to cache), but is still a small fraction of the peak processor performance (4GFLOPS).*

6. **Temporal locality:** The notion of repeated reference to a data item in a small time window is called temporal locality of reference. Data reuse is critical for cache performance because if each data item is used only once, it would still have to be fetched once per use from the DRAM, and therefore the DRAM latency would be paid for each operation.
7. **Bandwidth:** The rate at which data can be moved between the processor and the memory. It is determined by the bandwidth of the memory bus as well as the memory units.

   Example 1.5.3. *We continue with Examples 1.5.1 and 1.5.2. Now consider increasing the block size to 4 words (memory bandwidth in this case; also, if a memory request returns a contiguous block of 4 words, the 4-word unit is also considered as a cache line). Now fetching the two input matrices into the cache takes* $200/4 = 50\mu s$*. Consequently the total time for the computation is* $50\mu s + 16\mu s = 66\mu s$*, which corresponds to a peak computation rate of* $64K/66\mu s \approx 993$*MFLOPS.*

8. Increasing the memory bandwidth, or building wide data bus connected to multiple memory banks, is expensive to construct. In a more practical system, consecutive words are sent on the memory bus on subsequent bus cycles after the first word is retrieved. For example, with a 32-bit data bus, the first word is put on the bus after 100ns (the latency) and one word is put on each subsequent bus cycle.Then the 4 words will become available after $100 + 3 \times$ (memory bus cycle)ns. Assuming a data bus operating at 200MHz, this adds 15ns to the cache line access time.
9. **Spatial locality:** Successive computations require contiguous data. If the computation (or data access pattern) in a program does not have spatial locality, then effective bandwidth can be much smaller than the peak bandwidth.

1.6 Exercises

Objectives

The objectives are to understand and be able to apply the concepts and discussions on

- Classification of parallel computers
- Understand the simple metrics used to evaluate the performance of a parallel program
- Understand the similarities and differences between Amdahl's Law and Gustafson's Law

- Apply the basic performance metrics, Amdahl's Law, and Gustafson's Law for simple problems.
- Limitations of memory system performance

Problems

1. In the discussion of parallel hardware, we used Flynn's taxonomy to identify parallel systems: SISD, SIMD, MIMD, SPMD, and MISD. Explain how would each of these parallel computer works, and give some examples of such systems.
2. Assume p processing elements (PEs) share a bus to the memory, each PE accesses k data items, and each data access takes time t_{cycle}.
   (a) What is the maximum execution time for each PE to access the k data items? (**Answer:** $t_{cycle}kp$)
   (b) Assume each PE now has its own cache, and on average, 50% of the data accesses are made to the local memory (or cache in this case). Further, let's assume the access time to local memory is the same as to the global memory, i.e., t_{cycle}. In such case, what is the maximum execution time for each PE to access k data items? (**Answer:** $0.5kt_{cycle} + 0.5kpt_{cycle}$)
3. Suppose that 70% of a program execution can be sped up if the program is parallelized and run on 16 processing units instead of one. By Amdahl's Law, what is the maximum speedup we can achieve for this problem if we increase the number of processing units, respectively, to 32, then to 64, and then to 128? (**Answer:** Use Amdahl's Law.)
4. For a problem size of interest, 6% of the operations of a parallel program are inside I/O functions that are executed on a single processor. What is the minimum number of processors needed in order for the parallel program to exhibit a speedup of 10? (**Answer:** Use Amdahl's Law.)
5. An application executing on 64 processors requires 220 seconds to run. Benchmarking reveals that 5% of the time is spent executing serial portions of the computation on a single processor. What is the scaled speedup of the application? (Answer: 60.85.)
6. A company recently purchased a powerful machine with 128 processors. You are tasked to demonstrate the performance of this machine by solving a problem in parallel. In order to do that, you aim at achieving a speedup of 100 in this application. What is the maximum fraction of the parallel execution time that can be devoted to inherently sequential operations if your application is to achieve this goal? (**Answer:** Use Gustafson's Law.)
7. Both Amdahl's Law and Gustafson's Law are derived from the same general speedup formula. However, when increasing the number of processors p, the maximum speedup predicted by Amdahl's Law converges to a certain limit, while the speedup predicted by Gustafson's Law increases without bound. Explain why this is so.
8. Consider a memory system with a DRAM of 512MB and L1 cache of 32KB with the CPU operating at 1GHz. The $l_{DRAM} = 100$ns and $l_{L1} = 1$ns (l represents the latency). In each memory cycle, the processor fetches 4 words. What is the peak achievable performance of a dot product of two vectors? (**Answer:** Assuming 2 multiply-add units as in Example 1.5.1, the computation performs 8 FLOPS on 2 memory fetches, i.e., 8 FLOPS in 200ns ignoring cache read time and 1 CPU cycle time, thus 40MFLOPS. If cache read time as well 1 CPU cycle are also counted, then it becomes 8 FLOPS in 207ns, which is at 38.64MFLOPS rate.)

9. Consider an SMP with a distributed shared-address-space. Consider a simple cost model in which it takes 1ns to access local cache, 100ns to access local memory, and 400ns to access remote memory. A parallel program is running on this machine. The program is perfectly load balanced with 80% of all accesses going to local cache, 10% to local memory, and 10% to remote memory. What is the effective memory access time of one processor for this computation? If the computation is memory bound, what is the peak computation rate? (**Answer:** 19.7MFLOPS)

Bibliography

Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003). *Introduction to Parallel Computing*. Addison Wesley.

Trobec, R., Slivnik, B., Bulić, P., and Robič, B. (2018). *Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms*. Springer.

Chapter 2

Overview of Parallel Systems

Some contents of this chapter are based on [Trobec et al. 2018] and [Grama et al. 2003].

2.1 Modelling Parallel Computation

2.1.1 Random Access Machine

Random access machine (RAM) is an abstract for serial or sequential computing. It consists of a processing element and a memory. The memory is considered infinite, that is, it has unlimited size. The memory is divided into basic units and can be accessed using the location (or address). The processing unit can access any memory location, that is, it has random access to any memory unit. we can use a graphical notation, shown in Figure 2.1 to represent a random access machine (or RAM) – a processing element connected to a memory. This is an abstract representation of serial computing, using this model we try to abstract most important properties, and ignore the less important ones in the theoretical modelling of serial computing. Important properties in RAM include memory contains program and data; processing element execute instructions on data, a PE has random access to memory.

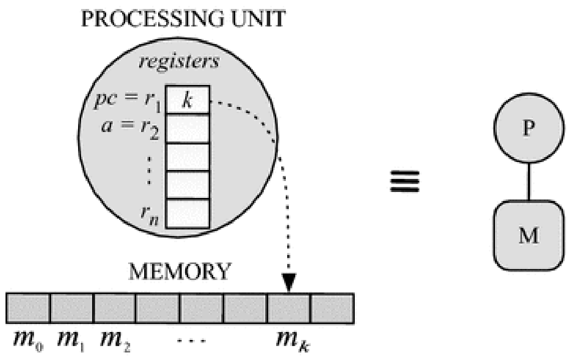


Figure 2.1: RAM model of computation: memory M - contains program instructions and data; processing unit P - execute instructions on data.

A natural extension of RAM to parallel computing can simply consists of multiple processing elements, in contrast to a single PE in serial computing; and we use a global memory of unlimited size that is uniformly accessible to all processing elements. The generalization of RAM to parallel computing can be done in 3 different ways:

- Parallel RAM (PRAM) (Figure 2.2 (a))
- Local memory machine (LMM)
- Modular memory machine (MMM)

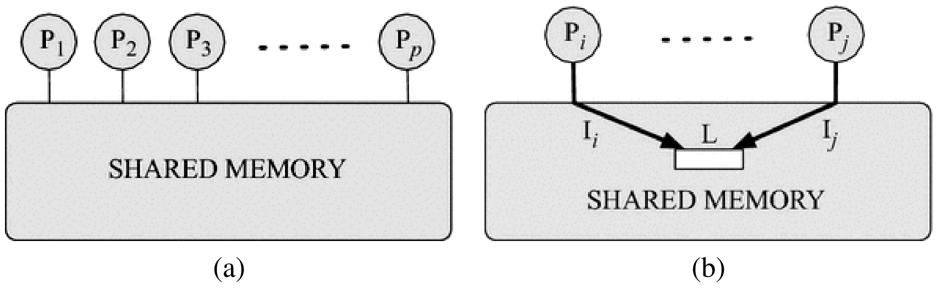


Figure 2.2: (a) PRAM model for parallel computation; (b) multiple PEs try to access the same memory location simultaneously.

First of all, PRAM - parallel random access machine – multiple PEs connected to a global memory, or we can call it shared memory; this is similar to RAM, where each PE has random access to any memory unit, all PEs have uniform, or equal access to the global memory, and we also assume the global memory has unlimited size. Clearly there are some issues with PRAM: what happens if multiple PEs need to access the same memory location (see an example in Figure 2.2 (b), where two PEs are trying to access the same memory unit L)? It could be two or more PEs try to read from the same memory location, multiple PEs try to write to the same memory location, or one or more PEs try to read and some others try to write to the same memory location? Of course, an obvious solution is to serialize such contending accesses, that means at one time, only one PE is allowed to access a particular memory unit. For such solution, we have another issue: which PE should go first, which one go next etc. These issues lead to uncertainties. In summary, simultaneous access to the same memory location could lead to unpredictable data in PEs, as well as in memory locations accessed. Depending on how multiple PEs access the memory simultaneously, a number of variants of PRAM are proposed. These variants differ in the ways for simultaneous access, and the ways for avoiding unpredictability.

- Exclusive read exclusive write PRAM (EREW-PRAM): It does not support simultaneous access to the same memory location - an access to any memory location must be exclusive, that means simultaneous memory accesses will be serialized; hence it affords minimum concurrency in memory access. Note that we are talking about simultaneous accesses to the same memory location; if multiple PEs, each tries to access different memory locations, they can happen in parallel.
- Concurrent read exclusive write PRAM (CREW-PRAM): Allows simultaneous reads from the same memory location, but writing to a memory location must be exclusive, that is. concurrent writes will be serialized.
- Concurrent read concurrent write (CRCW PRAM): Supports simultaneous reads from the same memory location; simultaneous writes to the same memory location, and simultaneous reads and writes to the same memory location. The unpredictability is handled in different ways: so within this type, we can further define a few different varieties:
  - Consistent CRCW-PRAM: PEs may simultaneously write to the same memory loca-

  tion, but they need to write the same value;
  - Abstract CRCW-PRAM: PEs may simultaneously try to write to the same memory location (not necessarily the same value), but only one of them will succeed, and it is unpredictable which one will succeed.
  - Priority CRCW-PRAM: There is a priority order imposed on PEs, e.g. PE0 has higher priority than PE1, then if PE0 and PE1 try to write the same memory location, then PE0 gets to write because it has higher priority than PE1. The process with highest priority gets to write and others fail.
  - Fusion CRCW-PRAM: PEs may simultaneously try to write to L, but it is assumed that a particular operation, such as taking the summation of values from each PE, is first performed on fly, and only the result of such operation will be written to L. Fusion operation could be one of the associative and commutative operators, including sum, product, max, min, and logical AND and logical OR.

- For each variant of CRCW, it will be programmer's job to enforce the specifics such as ensuring the same value to be written in the case of consistent CRCW-PRAM, which one to succeed in abstract CRCW-PRAM, and the priority order of PEs for the priority CRCW-PRAM.

Note that the restriction of simultaneous access is relaxed from EREW-PRAM, to CREW PRAM, and to CRCW-PRAM. This leads to some computational power gain from EREW-PRAM to CRCW-PRAM gradually, but not much, it is only in the order of logarithm.

The second one is LMM (Figure 2.3 (a)):

- Each PE has its own local memory, hence called local memory machine; accessing such local memory is fast;
- In this figure, each unit is a RAM, and they are connected through an interconnected network;
- A PE can access non-local memory (which is attached to other PEs) via interconnect network; accessing such memory could be slower, dependent on the capability of the interconnection.

The last one is MMM (Figure 2.3 (b)):

- No local memory to PEs;
- A set of PEs is connected to a set of memory modules through interconnection network;
- Accessing any memory module from any PE is considered uniform; by uniform we mean there is no difference in the access speed.

For LMM and MMM, interconnection networks play an important role for the performance of these types of machines.

2.2 Interconnection Networks

- Experiments have shown that the performance of real-world parallel applications are more and more dependent on the communication time rather than calculation time. That means high performance interconnection networks could lead to more efficiency in parallel applications. This is not only in the form of shorter parallel execution time, but also more PEs that can be exploited.

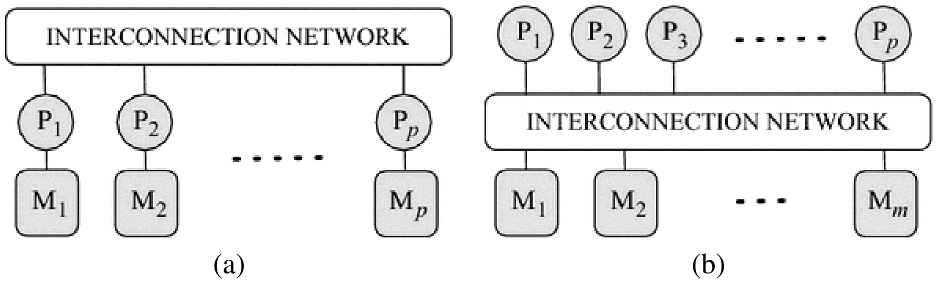


Figure 2.3: (a) LMM; (b) MMM.

- Interconnection networks provide mechanisms for data communication between processing nodes or between processors and memory modules.
- Interconnection networks typically are built using links and switches.
- Figures 2.4 (a) and (b) show examples of a fully connected network and a fully connected crossbar network, respectively. Fully connected network is direct network, where fully connected crossbar network is indirect network.
- For fully connected network, each node is a RAM, it represents a PE with a memory; and each node is connected to every other nodes.
- For the fully connected crossbar network, it is an MMM type, each PE is connected to every memory module; In this figure, each intersection in this grid represents a switch; the bold dots mean a particular state of connections; e.g., P1 is connected to M1; P2 is connected to M3, and so on.
- They are both non-blocking, in the sense that a connection between a pair of input and output nodes does not block connections between any other pair.

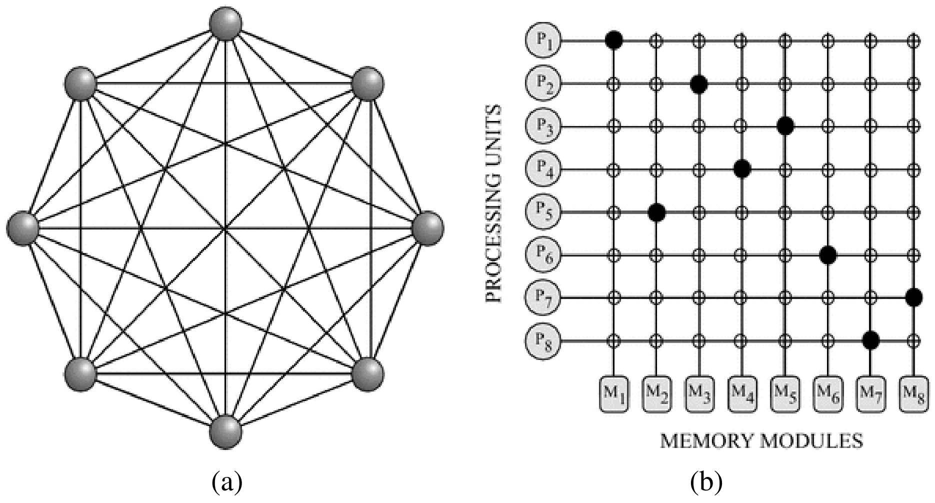


Figure 2.4: (a) Fully connected network; (b) A fully connected crossbar switch. It is non-blocking in the sense that a connection of a PE to a memory does not block another similar connection. It is scalable in terms of performance, but not scalable in terms of cost.

2.2.1 Important Factors of Interconnection Networks

- Routing: the process for choosing a path in an interconnection network traffic;
- Flow control: the process of managing the rate of data transmission between two nodes to avoid scenario where a fast sender could overwhelm a slow receiver;
- Network topology: the arrangement of various elements, such as communication nodes and channels, of an interconnection network.

2.2.2 Some Basic Properties of Interconnection Networks

- An interconnection network can be represented as a graph $G(V, E)$, where V is the set of nodes, and E is the set of links (or edges).
- Topological properties:
  - Node degree: the number of edges connecting a node
    - An interconnection network is regular if all nodes have the same node degree
  - Diameter
    - The number of nodes traversed by a packet from the source to the destination (a path) is called *hop count*
    - If there are multiple paths between a pair of source and destination nodes, the path with the shortest hop count gives the minimum hop count, l
    - Average distance, l_{avg}: the average of all ls taken over all possible pairs of nodes.
    - Diameter: The maximum hop count l taken over all possible pairs of source and destination nodes.
  - Path diversity: Multiple paths between a pair of communication nodes
  - Scalability: i) the capability of a network that handles growing amount of workload; ii) the potential of a network to be enlarged to accommodate growing amount of work.
- Performance properties:
  - Bisection width: the minimum number of communication links that must be removed to partition the network into two equal parts (or almost equal parts)
  - Channel bandwidth: the peak rate at which data can be communicated over a communication link (channel), e.g., if the transfer time of a word is t_w, then the bandwidth is $1/t_w$.
  - Bisection bandwidth: the minimum volume of data communication allowed between any two halves of the network. It is the product of bisection width and channel bandwidth.
  - Cost: One way of defining the cost of a network is in terms of the number of communication links required by the network.

2.2.3 The Classification of Interconnection Networks

- Direct networks (static): Each node is directly connected to its neighbours. It has point-to-point communication links (network interface) between nodes.
  - Fully connected network (Figure 2.5 (a)): If the number of nodes is n, then such a network has $\frac{1}{2}n(n-1)$ connections.

- Indirect networks (dynamic): Connect nodes and memory modules via switches and communication links. A cross point is a switch that can be opened or closed. Uses switches to establish paths among nodes.
  - Fully connected crossbar switch (Figure 2.5 (b)): On one end is nodes, and the other end memory modules. The fully connected crossbar has too large complexity to be used for connecting large numbers of input and output ports. For example, if we have 1000 nodes and 1000 memory modules, then we need one million switches to build the fully connected crossbar switches.

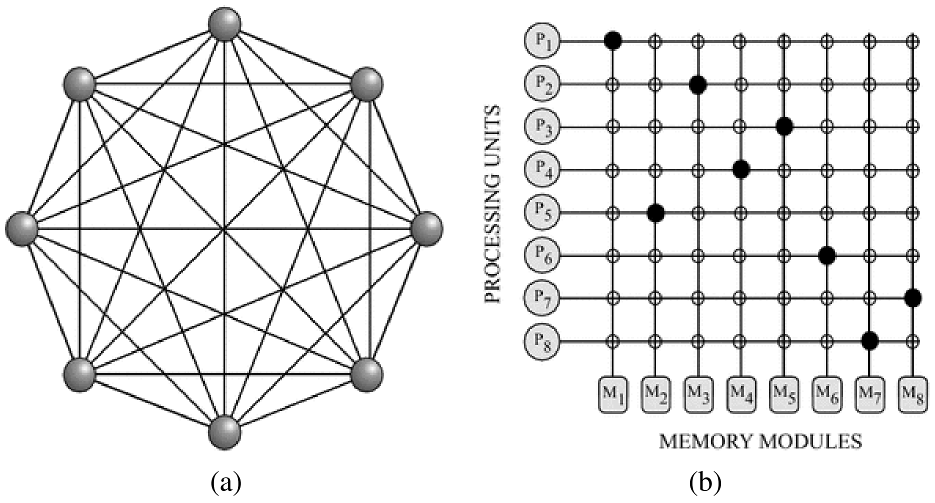


Figure 2.5: (a) Fully connected network; (b) A fully connected crossbar network.

2.2.4 Popular Interconnection Network Topologies

- Bus. Used in both LMMs and MMMs (see Figure 2.6). At one time, only one process is allowed to use the bus for communication.
  - Advantages: simple to build; buses are ideal for broadcasting data among nodes.
  - Disadvantages: unscalable in terms of performance (but scalable in terms of cost)

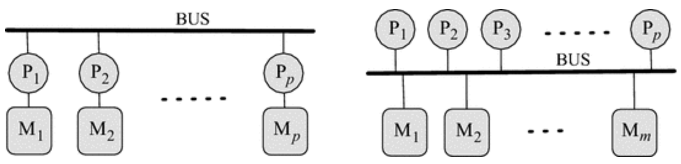


Figure 2.6: Bus topology.

- Linear array. Used in LMMs. Every node (except the two nodes at the ends) is connected to two neighbours, see Figure 2.7. Simple. If a node index is i, its two neighbours can be found using indices $(i+1) \bmod n$ and $(i-1) \bmod n$, where n is the total number of nodes in the linear array.

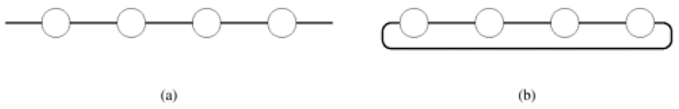


Figure 2.7: (a) Linear array without wraparound link; (b) linear array with wraparound link (also called ring, see next slide).

- Ring. Used in LMMs. Every node is connected to two neighbours, see Figure 2.8. Simple. If a node index is i, its two neighbours can be found using indices $(i+1)$ mod n and $(i-1)$ mod n, where n is the total number of nodes in the ring.

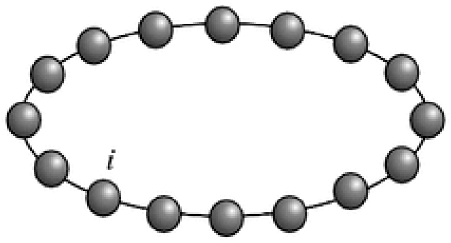


Figure 2.8: Ring topology. Each node represents a processing element with local memory.

- 2D mesh (Figure 2.9 (a)). Can be used in LMMs. Each node is connected to a switch. The number of switches can be determined by the lengths of the two sides. Every switch, except those along the 4 borders, has 4 neighbours.
- 2D torus (Figure 2.9 (b)). Similar to 2D mesh, however, each pair of corresponding border switches is connected. Every switch has 4 neighbours.

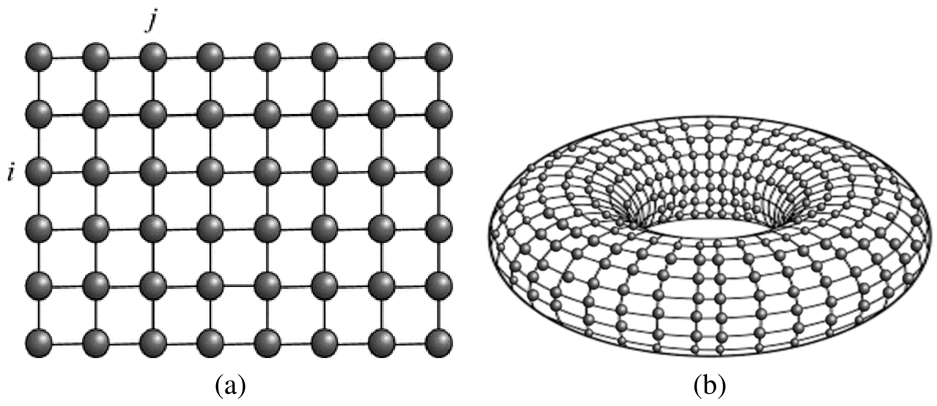


Figure 2.9: (a) 2D mesh topology; (b) 2D torus topology. Each node represents a processing element with local memory.

- 3D mesh. Similar to 2D mesh, however, in 3 dimension. Every switch except the border ones, has 6 neighbours (see Figure 2.10 (a)).

- 3D torus, Every pair of opposite switches are connected in 3D mesh.
- Hypercube: An interconnection network that has $n = 2^d$ nodes, where d is the number of dimensions. Each node has a distinct label consisting of d binary bits. For example, $d = 3$, then $n = 8$. Two nodes are connected via a link if and only if their labels differ in only one bit location (see Figures 2.10 (b) and 2.11). Used in LMMs.

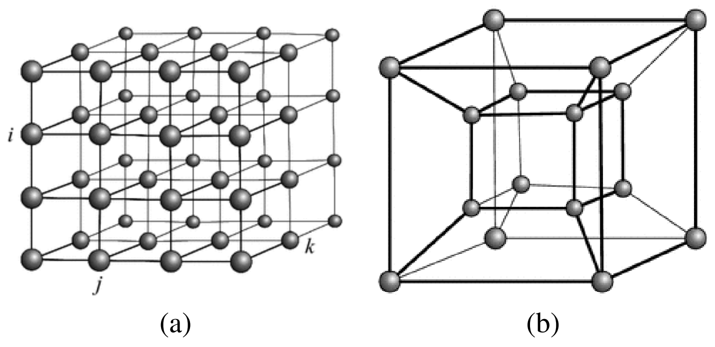


Figure 2.10: (a) 3D mesh, (b) Hypercube topology Each node represents a processing element with local memory.

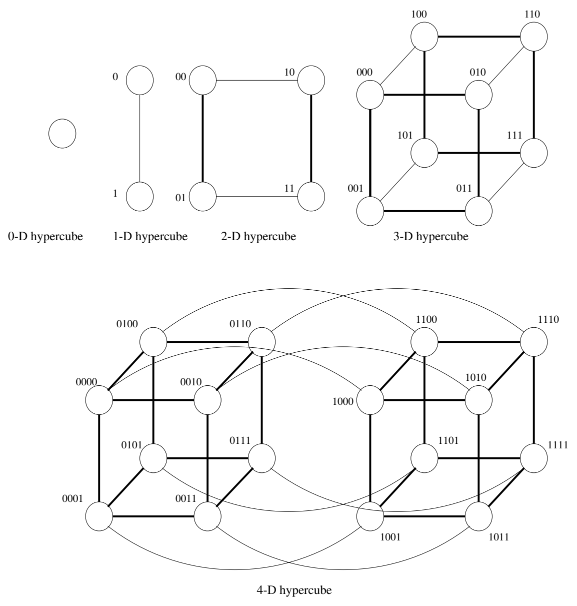


Figure 2.11: Hypercubes of 1D, 2D, 3D, and 4D.

- Multistage network: used in MMM, where input switches are connected to PEs, and output switches are connected to memory modules (see Figure 2.12).
- Fat tree: used in constructing LMM, where PEs with their local memories are attached to the leaves (see Figure 2.13).

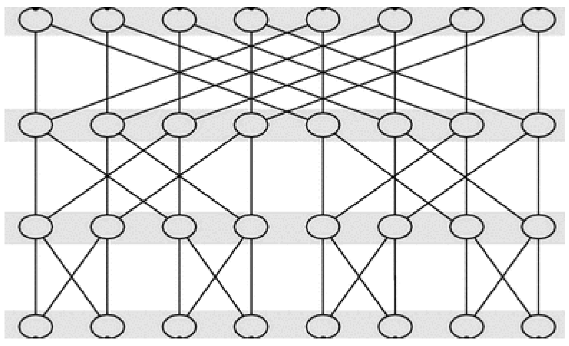


Figure 2.12: Multistage network topology. A 4-stage interconnection network capable of connecting 16 PEs to 16 memory modules. Each switch can establish a connection between a pair of input and output channels.

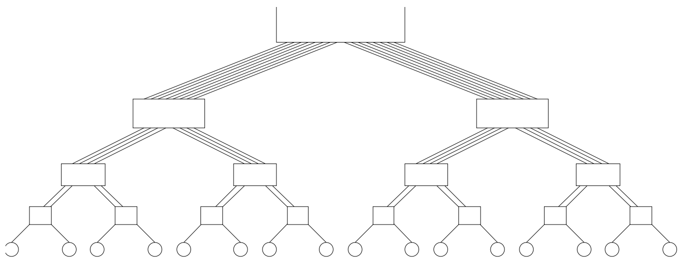


Figure 2.13: Fat tree topology. A fattree interconnect network of 16 processing nodes. Each switch can establish a connection between arbitrary pair of incidents. Edges closer to the root are thicker. The idea is to increase the number of communication links and switching nodes closer to the root.

2.2.5 Evaluating the Interconnect Network

- Diameter
- Bisection width
- Cost

See Table 2.1 for quantitative measures of various interconnect networks.

2.3 Exercises

Objectives

The objectives are to understand and be able to apply the concepts and discussions on

- Theoretical modelling of parallel computation
- The basic properties and topologies of common interconnection networks

Table 2.1: Quantitative characteristics of various interconnect networks

	Network	Diameter	bisection width	Cost
Static	Fully connected	1	$p^2/4$	$p(p-1)/2$
	Linear array	p-1	1	p-1
	2D mesh	$2(\sqrt{p}-1)$	$\sqrt{p}$	$2(p-\sqrt{p})$
	Ring	$\lfloor p/2 \rfloor$	2	p
	2D torus	$2\lfloor \sqrt{p}/2 \rfloor$	$2\sqrt{p}$	2p
	Hypercube	$\log p$	$p/2$	$(p \log p)/2$
Dynamic	Crossbar	1	p	p^2
	Fat tree	$2 \log p$	1 (or the # of links)	p-1 (or the # of switches)

*With p nodes in the above networks

Problems

1. Of the three PRAM models, EREW-PRAM, CREW-PRAM, and CRCW-PRAM, which one has the minimum concurrency, and which one has the maximum power? Why?
2. How do you label the nodes in a d-dimensional hypercube? Construct a 4-dimensional hypercube with labels, where there is a communication link between two nodes when their labels differ in only one binary bit.
3. (a) Partition a d-dimensional hypercube into two equal partitions such that
      i. each partition is still a hypercube;
      ii. neither partition is a hypercube.

   (b) Derive the bisection width for each of the two cases in Item 3a.
4. Derive the diameter, bisection width, and cost for each of the following interconnection networks. Show your derivation. Assume there are p number of nodes where applicable.
   - Fully connected network
   - Bus
   - Ring
   - 2D mesh
   - 2D torus
   - 3D mesh
   - Hypercube
   - Fully connected crossbar network
5. Using the basic interconnection network topologies we studied in the class, design, as many as you can, network topologies that can connect p number of PEs to m number of memory modules, so that each topology should provide a path between any pair of a PE and a memory module.

Bibliography

Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003). *Introduction to Parallel Computing*. Addison Wesley.

Trobec, R., Slivnik, B., Bulić, P., and Robič, B. (2018). *Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms*. Springer.

Chapter 3
Parallel Algorithm Design

The contents of this chapter are mainly based on [Culler et al. 1998] and [Grama et al. 2003].

3.1 Shared Memory vs Distributed memory

- Physical memory in parallel computers is either shared or local.
- Shared memory parallel computer: parallel computers that share memory space
- Distributed memory parallel computer: parallel computers that do not share memory space

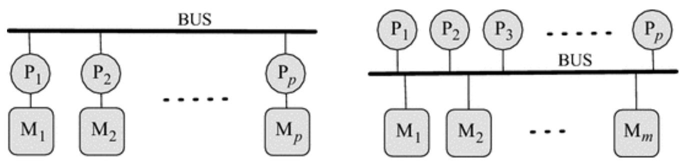


Figure 3.1: Bus based parallel computers.

- From programming point of view: shared memory programming and distributed memory programming
  - Shared memory programming: All PEs have equal access to shared memory space. Data shared among PEs are via shared variables.
  - Distributed memory programming: A PE has access only to its local memory. Data (message) shared among PEs are communicated via the communication channel, i.e., interconnection network.
- In terms of programming, programming shared memory parallel computers is easier than programming distributed memory systems.
- In terms of scalability, shared memory systems and distributed memory systems display different performance and cost characteristics. For example,
  - Bus (Figure 2.6), scalable in terms of cost, but not performance
  - Fully connected crossbar (Figure 3.2 (a)) switch, scalable in terms of performance , but not cost
  - Hypercube, scalable both in performance and cost
  - 2D mesh (Figure 3.2 (b)), scalable both in performance and cost
- In terms of problem size, distributed memory systems are more suitable for problems with vast amount of data and computation

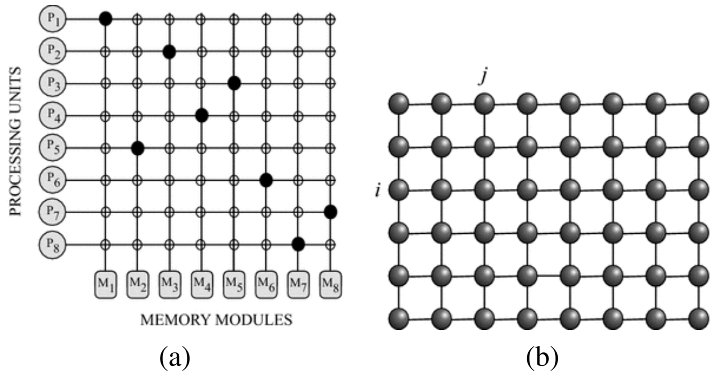


Figure 3.2: (a) Fully connected crossbar switch based system; (b) 2D mesh.

3.1.1 Shared Memory Programming

- In shared memory programming, variables have two types: shared variable and private variable.
- Shared variable can be read and write by any thread; and private variables can usually only be accessed by one thread.
- In shared memory systems, communication among threads happen via shared variables — this means communication is implicit.
- Non-determinism in shared memory programming. Non-determinism: A computation is non-deterministic if a given input can result in different outputs.
- In a shared memory system, non-determinism arises when multiple threads are executing independently, and at different rates. Then these threads could complete at different orders for different runs, and hence could give different outputs from run to run. This is a typical case of non-determinism.
- Non-determinism can be harmless for some problem, and can cause issues for some problems.

Example 3.1.1. *Suppose we have two threads in a parallel program, one with thread ID 0, and the other with thread ID 1. Suppose also that each thread has a private variable* `my_x`*; thread 0 stores a value* 5 *for* `my_x`*, and thread 1 a value* 9 *for its* `my_x`*.*

Thread ID	*Variable*	*Value*
0	`my_x`	*5*
1	`my_x`	*9*

Now, what will be the output like if both threads execute the following line of code?

```
...
printf("Thread %d > my_x = %d\n", my_ID, my_x);
...
```

The output could be

```
Thread 0 > my_x = 5
Thread 1 > my_x = 9
```

or

```
Thread 1 > my_x = 9
Thread 0 > my_x = 5
```

Example 3.1.2. *Suppose we have a shared variable* `min_x` *(with initial value* ∞*) and two threads in our program. Each thread has a private variable* `my_x`*, thread 0 stores a value* 5 *for* `my_x`*, and thread 1 a value* 9 *for its* `my_x`*. What will happen if both threads update* `min_x = my_x` *simultaneously?*

```
/*each thread tries to update variable min_x as follows*/
if (my_x < min_x)
  min_x = my_x;
```

- *This is the situation where two threads more or less try to update a shared variable (*`min_x`*) simultaneously.*
- ***Race condition:*** *When threads attempt to access a resource simultaneously, and the access can result in error, we often say the program has a race condition.*
- *In such case, we can serialize the contending activities by setting a critical section where the section can only be accessed by one thread at a time.*

3.1.2 Distributed Memory Programming

- In distributed memory programming, the processes can only access their own private (or local) memory. Message-passing programming model (or APIs) is most commonly used. For example, MPI.
- In message passing, each process is identified by its rank.
- Processes communicate with each other by explicitly sending and receiving messages.
- Message passing APIs often provide basic send and receive functions, they also provide more powerful communication functions such as broadcast and reduction.
  - Broadcast, a single process transmits the same data to all processes;
  - Reduction, the results computed by individual processes are combined to a single results, e.g., summation.

3.2 Parallel Algorithm Design

The following steps are often taken to solve a problem in parallel [Culler et al. 1998, in Section 2.3.1]:

- Decomposition (or partitioning) of the computation into tasks;
- Assignment of tasks to processes;
- Orchestration of the necessary data access, communication, synchronization among processes;
- Mapping of processes to processors.

Figure 3.3 illustrates these steps.

- Decomposition involves decomposing a problem into finer tasks such that these tasks could be executed in parallel.

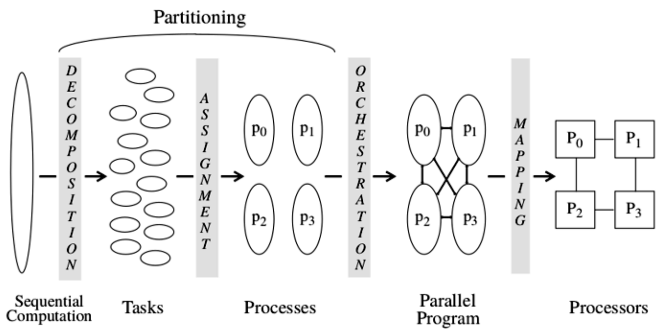


Figure 3.3: Steps in parallelization, relationship between PEs, tasks and processors.

- The major goal of decomposition is to expose enough concurrency to keep processes busy all the time, yet not so much that overhead of managing tasks become substantial.
- Assignment involves assigning fine tasks to available processes, such that each process has approximately similar workload. Load balancing is an challenging issue in parallel programming.
- The primary performance goals of assignment are to achieve balanced workload, reduce the runtime overhead of managing assignment.
- The third step involves necessary orchestration, could involve communication among processes and synchronization.
- The major performance goals in orchestration:
  - reducing the cost of communication and synchronization
  - preserving locality of data reference
  - scheduling tasks
  - reducing the overhead of parallelism management
- Finally, the mapping is to map processes to processors. The number of processes and the number of processors are not necessarily need to be matched. That is, for example, you can have 8 processes on a 4 processor machines. In such a case, a processor may handle more than one processes by techniques such as space sharing and time sharing.
- The mapping process can be taken care of by OS in order to optimize resource allocation and utilization. The program may also control the mapping of course.

3.2.1 Parallel Algorithm Design — Simple Examples

Example 3.2.1. *Consider a simple example program with two phases.*

- *In the first phase, a single operation is performed independently on all points of a 2-dimensional n by n grid (e.g., see Figure 3.4);*
- *in the second phase, the sum of n^2 grid point values is computed.*

If we have p processes,

- *we can assign n^2/p points to each process and complete the first phase in time n^2/p.*
- *In the second phase, each process can add each of its assigned n^2/p values to a global sum variable.*

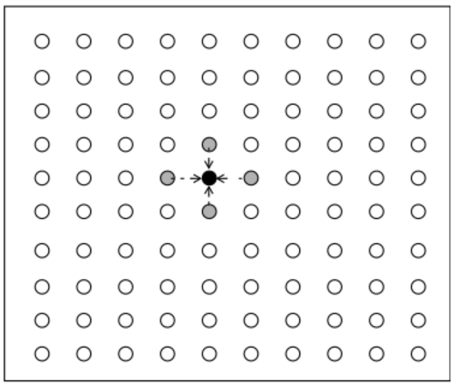


Figure 3.4: 2D grid points.

- *Issue: the second phase is in serial where it takes n^2 time regardless of multiple processes. The total time is $n^2/p + n^2$. Then the speedup, compared to the sequential time $2n^2$, is*

$$s = \frac{2n^2}{\frac{n^2}{p} + n^2} = \frac{2}{\frac{1}{p} + 1},$$

  which is at most 2, even if a large number of processes is used.
- *Can we expose more concurrency?*

We can improve the performance:

- *We can first compute local sums of n^2/p values on all processes simultaneously.*
- *After n^2/p time, we have p number of local sums.*
- *We then add these p local sums into a global sum one at a time, which takes p units of time.*
- *Now the total time is $n^2/p + n^2/p + p$. The speedup is*

$$s = \frac{2n^2}{\frac{2n^2}{p} + p} = p \times \frac{2n^2}{2n^2 + p^2}.$$

  This speedup is almost linear in p, the number of processors used, when n is large compared to p. Figure 3.5 shows the impact of concurrency on the performance for Example 3.2.1.

Example 3.2.2. *Suppose we have an array with large quantities of floating point data stored in it. In order to have a good feeling of the distribution of the data, we can find the histogram of the data. To find the histogram of a set of data, we can simply divide the range of data into equal sized subintervals, or bins, determine the number of values in each bin, and plot a bar graphs showing the sizes of the bins.*

As a very small example, suppose our data are

$$\begin{aligned} A = [&1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3,\\ &4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9] \end{aligned}$$

- *For A, $A_{min} = 0.3$, $A_{max} = 4.9$, $A_{count} = 20$.*

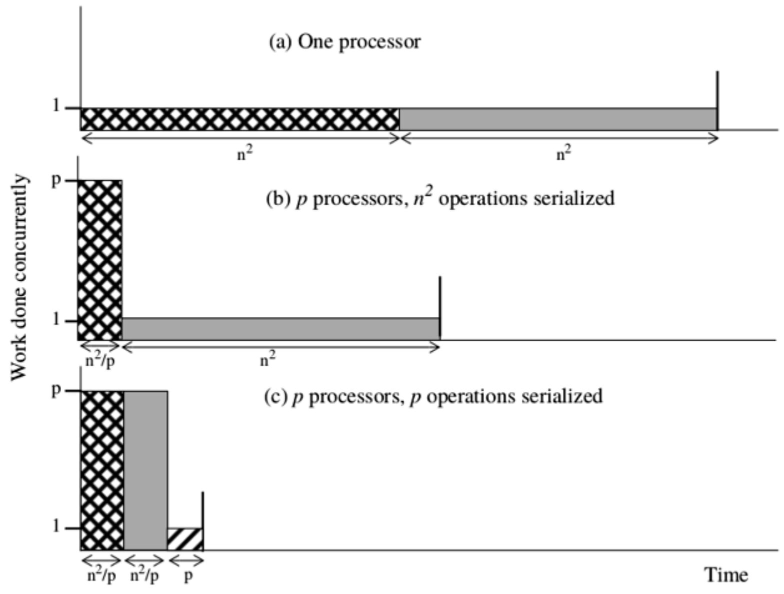


Figure 3.5: Impact of limited concurrency.

- *Let's set the number of bins to be 5, and the bins are* $[0, 1.0), [1.0, 2.0), [2.0, 3.0), [3.0, 4.0), [4.0, 5.0)$, *and* $bin_{width} = (5.0 - 0)/5 = 1.0$.
- *Then* $bin_{count} = [6, 3, 2, 3, 6]$ *is the histogram — the output is an array the number of elements of data that lie in each bin.*

```
for (i = 0; i < data_count; i++){
    bin = Find_bin(data[i], ...);
    bin_count[bin]++;
}
```

`Find_bin` *function returns the bin that* `data[i]` *belongs to.*

- *Now if we want to parallelize this problem, first we can decompose the dataset into subsets. Given the small dataset, we divide it into 4 subsets, so that each subset has 5 elements.*
  - *Identify tasks (decompose): i) finding the bin to which an element of data belongs; ii) increment the corresponding entry in* `bin_count`.
  - *The second task can only be done once the first task has been completed.*
  - *If two processes or threads are assigned elements from the same bin, then both of them will try to update the count of the same bin (Figure 3.6 shows an example). Assuming the* `bin_count` *is shared, this will cause* ***race condition****.*

- *A solution to race condition in this example is to create local copies of* `bin-count`*, each process updates their local copies, and at the end add these local copies into the global* `bin_count`*. Figure 3.7 hows an example of this idea.*

- *In summary, the parallelization approach is to*
  - *Elements of data are assigned to processes/threads so that each process/thread gets roughly the same number of elements;*

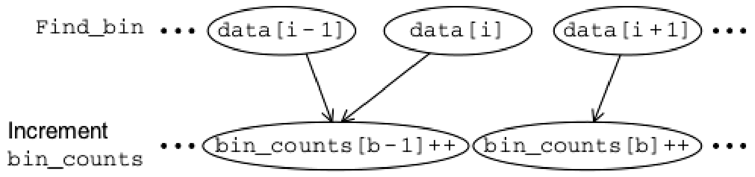


Figure 3.6: Tasks and their communications.

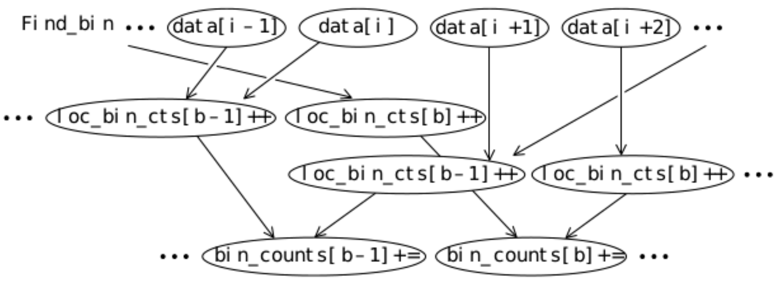


Figure 3.7: Tasks and communications.

- *Each process/thread is responsible for updating its `local_bin_counts` array based on the assigned data elements.*
- *The local `local_bin_counts` are needed to be aggregated into the global `bin_counts`.*
  - *If the number of bins is small, the final aggregation can be done by a single process/thread.*
  - *If the number of bins is large, we can apply parallel addition in this step too.*

3.3 Summary

- Aspects of parallel program design
  - Decomposition to create concurrent tasks
  - Assignment of works to workers
  - Orchestration to coordinate processing of tasks by processes/threads
  - Mapping to hardware

  We will look more into making good decisions in these aspects in the coming lectures.

In this chapter, we gave a general introduction to shared memory programming, distributed memory programming, an approach in parallel algorithm design, and a few examples to demonstrate some of the concepts we discussed.

3.4 Exercises

Objectives

Students should be able to

- Design parallel algorithms for problems with common computation patterns such as reduction and broadcasting.
- Identify the concurrency, task dependency, communication, and synchronization in a parallel solution.

Problems

1. Consider the multiplication of a dense $n \times n$ matrix A with a vector b to yield another vector y. The ith element $y[i]$ of the output vector is the dot-product of the ith row of A and the input vector b, i.e., $y[i] = \sum_{j=1}^{n} A[i,j]b[j]$. For this problem, design a parallel algorithm that involves decomposition, assignment of tasks, analyzing the communication pattern, and mapping.
2. Consider the problem of sorting a sequence A of n elements using quicksort algorithm. Quicksort is a divide and conquer algorithm that starts by selecting a pivot element x and then partitions the sequence A into two subsequences A_0 and A_1 such that all the elements in A_0 are smaller than x and all the elements in A_1 are greater than or equal to x. Each subsequence is then sorted by recursively calling quicksort. How would you parallelize quicksort?
3. Many parallel algorithms for distributed memory systems require a broadcast step in which one task (or process) communicates a value it holds to all the tasks (or processes).
   - How a broadcast would be implemented for a parallel computer with the following interconnect architecture respectively?
     - Bus
     - Ring
     - Hypercube
     - Fat tree
   - What is the time complexity for broadcast with Hypercube interconnect architecture?
4. Figure 3.8 illustrates a task dependency graph of a decomposition for an $N \times N$ grid based iterative solver. Each node represents a task for a grid point, and a red arrows shows the dependency between a pair of tasks, i.e., a red arrow from a grid point P to another grid point Q means the task Q is dependent on task P. Analyse the concurrency, i.e., which tasks can be executed in parallel, based on this task dependency graph.
5. The problem from Question 4 is modified so that it has more concurrency. As shown in Figure 3.9, the solution now takes two phases, where in the first phase, the algorithm updates all the red cells, and in the second phase, it updates all the black cells. The algorithm repeats this process until convergence.
   (a) Discuss the concurrency and task dependencies for this modified approach.
   (b) How would you assign the tasks to P number of processes? Assume $P = 4$.
   (c) What are the communication and synchronization like during the execution of this iterative solver?

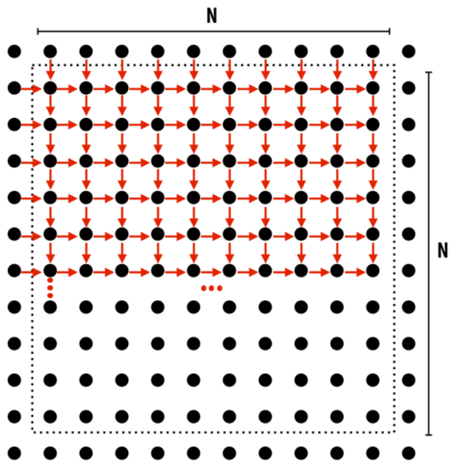


Figure 3.8: Each row elements depends on the element to the left; each column depends on the previous column.

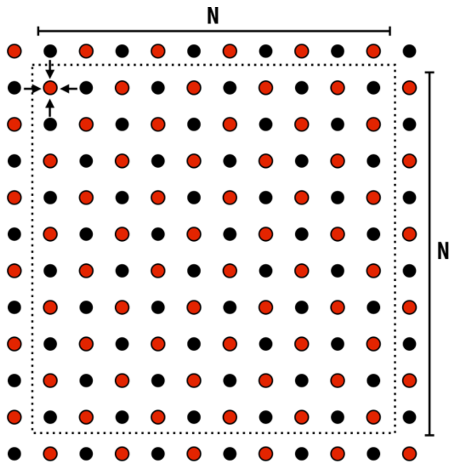


Figure 3.9: Update all the red cells first, then update all the black cells. The process is repeated until convergence.

Bibliography

Culler, D. E., Singh, J. P., and Gupta, A. (1998). *Parallel Computer Architecture: A Hardware/-Software Approach*. Morgan Kaufmann Publishers, Inc.

Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003). *Introduction to Parallel Computing*. Addison Wesley.

Chapter 4

Shared Memory Programming using OpenMP

The contents in this chapter are based on sources [Chapman et al. 2007] and [Trobec et al. 2018].

4.1 OpenMP Overview

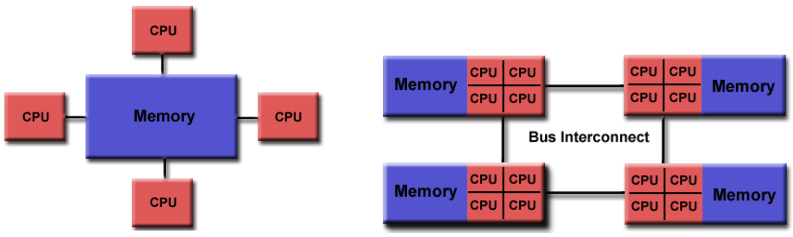
4.1.1 What is OpenMP?

Open specifications for Multi Processing (OpenMP) is

- An Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism.
- Comprised of three primary API components:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- API is specified for C, C++ and Fortran.
- Portable
- Easy to use

4.1.2 OpenMP Programming Model

- OpenMP is designed for multi-processor/core, shared memory machines. For example, Figure 4.1 shows two types of shared memory machines.



(a) UMA (b) NUMA

Figure 4.1: The shared memory model of parallel computation.

- Thread Based Parallelism:

- OpenMP programs accomplish parallelism exclusively through the use of threads.
- A thread of execution is the smallest unit of processing that can be scheduled by an operating system.
- Threads exist within the resources of a single process. Without the process, they cease to exist.
- Typically, the number of threads match the number of processors/cores of a machine, but it can differ.

• Explicit Parallelism:
  - OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.

• Compiler Directive Based: Most OpenMP parallelism is specified through the use of compiler directives which are embedded in C/C++ or Fortran source code.

• Fork - Join Model: OpenMP programs begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
  - FORK:
    * The master thread then creates a team of parallel threads;
    * Becomes the master of this group of threads;
    * Assigned the thread number 0 within the group.
  - JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.
  - Figures 4.2 and 4.3 show illustrations of fork-join model.

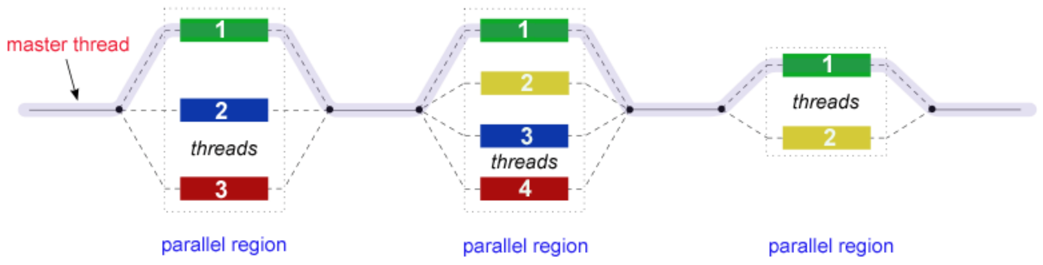


Figure 4.2: Fork-join model

4.1.3 OpenMP API Overview

• Three components: Compiler Directives, Runtime Library Routines, Environment Variables.
  - Compiler Directives: Appear as comments in your source code to guide the compilers.
  - The OpenMP API includes an ever-growing number of run-time library routines.
  - OpenMP provides several environment variables for controlling the execution of parallel code at run-time.

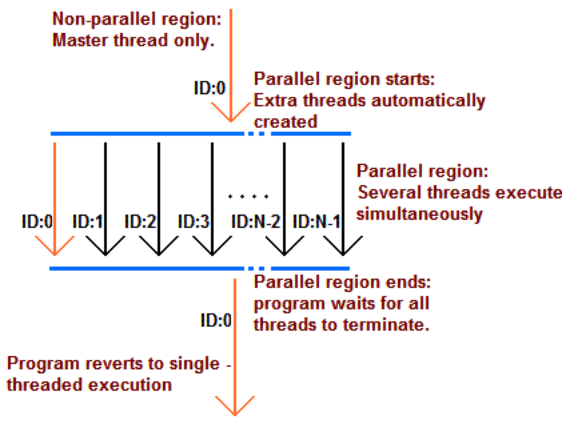

Non-parallel region:
Master thread only.
ID:0
Parallel region starts:
Extra threads automatically created
Parallel region:
Several threads execute simultaneously
ID:0 ID:1 ID:2 ID:3 ID:N-2 ID:N-1
Parallel region ends:
program waits for all threads to terminate.
ID:0
Program reverts to single - threaded execution

Figure 4.3: OpenMP parallel construct execution model

- OpenMP core syntax:
  - Most of the constructs in OpenMP are compiler directives:
    #pragma omp ⟨construct name⟩ [clause [clause]. . .]
  - OpenMP code structure: Structured block. A block of one or more statements with one point of entry and one point of exit at the end.
  - Function prototypes and types in the file:
    #include ⟨omp.h⟩

4.1.4 Compiling OpenMP Programs

Exercise 1: Write a multi-threaded C program that prints “Hello World!” (see Listing 4.1). (`hello_omp.c`)

```
#include <stdio.h>
#include <omp.h>
int main(){
  //Parallel region with default number of threads
  #pragma omp parallel
  //Start of the parallel region
  {
    //Runtime library function to return a thread number
    int ID = omp_get_thread_num();
    printf("Hello World! (Thread %d)\n", ID);
  }//End of the parallel region
}
```

Listing 4.1: The OpenMP code to print "Hello World!"

- To compile, type **gcc -fopenmp hello_omp.c -o hello_omp**.
- To run, type **./hello_omp**

4.2 OpenMP Core Features

4.2.1 Parallel Region Construct

A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.

- Syntax:

```
#pragma omp parallel [clause[[,] clause] ... ]
structured block
```

Typical clauses in `clause` list:

- Degree of concurrency:
  `num_threads(<integer expression>)`
- Data scoping:
  - `private(<variable list>)`
  - `firstprivate(<variable list>)`
  - `shared(<variable list>)`
  - `default(<data scoping specifier>)` — (shared or none)
- Conditional parallelization:
  `if (<scalar expression>)`, determines whether the `parallel` construct creates threads.
- Interpreting an OpenMP parallel directive

```
......
int b, a=10, c=5;
#pragma omp parallel if (is_parallel==1) num_threads(8) shared(b) \
    private(a) firstprivate(c) default(none)
{
    /* structured block */
}
```

Create threads in OpemMP using the parallel construct.

Example 4.2.1. *Create a 4-thread parallel region using* num_threads *clause (see Listing 4.2).*

```
#include <stdio.h>
#include <omp.h>
int main()
{
  //Parallel region with 'num_threads' clause to set number of
     threads
  #pragma omp parallel num_threads(4)
  //Start of the parallel region
  {
    //Runtime library function to return a thread number
    int ID = omp_get_thread_num();
    printf("Hello World! (Thread %d)\n", ID);
    }//End of the parallel region
  }
```

Listing 4.2: The OpenMP code to print "Hello World!"

Example 4.2.2. *Create a 4-thread parallel region using runtime library routine (see Listing 4.3).*

```
#include <stdio.h>
#include <omp.h>
int main()
{
  //Runtime library function to request the number of threads in the
     parallel region
  omp_set_num_threads(4);
  #pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf("Hello World! (Thread %d)\n", ID);
  }
}
```

Listing 4.3: The OpenMP code to print "Hello World!"

Different ways to set the number of threads in a parallel region:

- Using runtime library function `omp_set_num_threads()`.
- Setting clause `num_threads`, e.g., `#pragma omp parallel num_threads (8)`.
- Specify at runtime using environment variable `OMP_NUM_THREADS`, e.g., `export OMP_NUM_THREADS =8`.

4.2.2 Example - Computing the π Using Integration in Parallel

Example 4.2.3. *Compute the number π using numerical integration:* $\int\limits_0^1 \frac{4.0}{1+x^2} = \pi$.

1. *Write a serial program for the problem (see Listing 4.4).*
2. *Parallelize the serial program using OpenMP directive.*
3. *Compare the results from 1 and 2.*

```
  static long num_steps = 1000000;
  double step;
  int main (){
    int i;
    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0;i< num_steps;i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
  }
```

Listing 4.4: Serial program for computing π

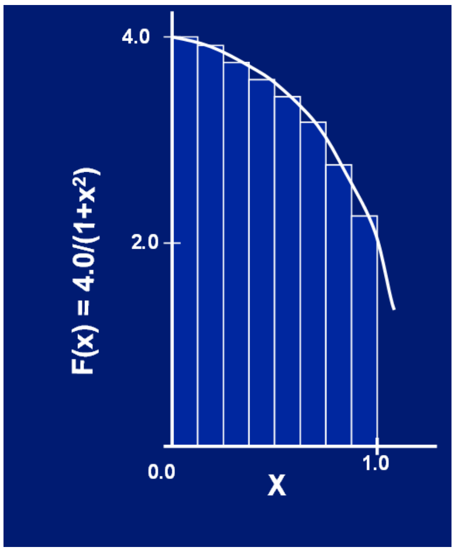


Figure 4.4: Approximating the π using numerical integration

Parallel Program for Computing Π — Method I

See Listing 4.5.

```
static long num_steps = 1000000;
double step;
#define NUM_THREADS 2
int main (){
  int nthreads;
  double pi, sum[NUM_THREADS];
  step = 1.0/(double)num_steps;
  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {
    int i, id, tthreads; double x;
    tthreads = omp_get_num_threads();
    id = omp_get_thread_num();
    if(id==0) nthreads=tthreads;
    for (i=id, sum[id]=0.0;i< num_steps; i=i+tthreads){
      x = (i+0.5)*step;
      sum[id] = sum[id] + 4.0/(1.0+x*x);
    }
  }
  for(i=0, pi=0.0;i<nthreads;i++)
  pi += step * sum[i];
}
```

Listing 4.5: A simple parallel program for computing π

False Sharing

If independent data elements happen to sit on the same cache line, each update will cause the cache lines to "slosh back and forth" between threads...this is called **false sharing** (Figure 4.5 shows an illustration).

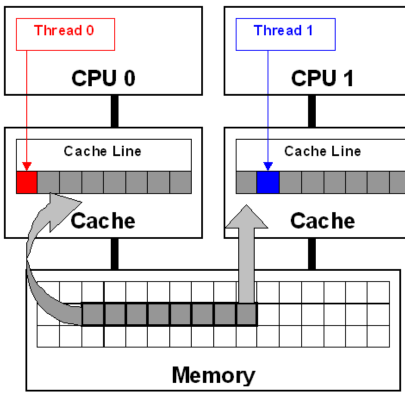


Figure 4.5: False sharing

Cache

- Processors usually execute operations faster than they access data in memory.
- To address this problem, a block of relatively fast memory is added to a processor — cache.
- The design of cache considers the temporal and spatial locality.
- For example, x is a shared variable and $x = 5$, `my_y`, `my_z` are private variables, what will be the value of `my_z`?

```
if myId==0{
    my_y = x;
    x++;
}else if myId==1{
    my_z = x;
}
```

Cache Coherence

Example 4.2.4. *Suppose $x = 2$ is a shared variable, $y0, y1, z1$ are private variables. If the statements in Table 4.1 are executed at the indicated times, then $x =?$, $y0 =?$, $y1 =?$, $z1 =?$ (See Figure 4.6.)*

Cache coherence problem - When the caches of multiple processors store the same variable, an update by one processor to the cached variable is 'seen' by the other processors. That is, the cached value stored by the other processors is also updated.

False Sharing

- Suppose two threads with separate caches access different variables that belong to the same cache line. Further suppose at least one of the threads updates its variable.

Time	Core 0	Core 1
0	$y0 = x;$	$y1 = 3 * x;$
1	$x = 7;$	Statements not involving x
2	Statements not involving x	$z1 = 4 * x;$

Table 4.1: Example - cache coherence

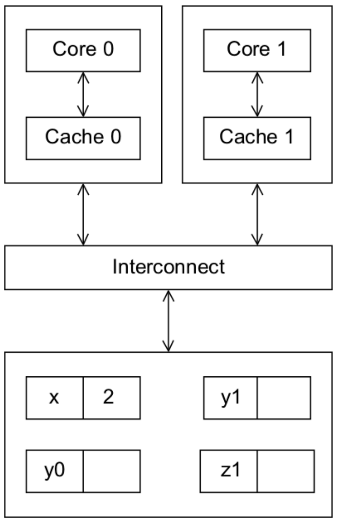


Figure 4.6: Example - cache coherence cont.

- Even though neither thread has written to a shared variable, the cache controller invalidates the entire cache line and forces the other threads to get the values of the variables from main memory.
- The threads aren't sharing anything (except a cache line), but the behaviour of the threads with respect to memory access is the same as if they were sharing a variable. Hence the name false sharing.

Parallel Program for Computing Π — Method II

See Listing 4.6.

```
static long num_steps = 1000000;
int main (){
  int i, tthreads, id;
  double step, pi = 0.0, sum = 0.0, x;
  step = 1.0/(double)num_steps;
  #pragma omp parallel
  {
    tthreads = omp_get_num_threads();
    id = omp_get_thread_num();
    for (i=id; i<num_steps; i+=tthreads){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
```

```
        }
      }
      pi += step * sum;
    }
```

Listing 4.6: A simple parallel program for computing π

Race Condition

- When multiple threads update a shared variable, the computation exhibits non-deterministic behaviour — race condition. That is, two or more threads attempt to access the same resource.
- If a block of code updates a shared resource, it can only be updated by one thread at a time.

4.2.3 Synchronization

- Synchronization: Bringing one or more threads to a well defined and known point in their execution. *Barrier* and *mutual exclusion* are two most often used forms of synchronization.
  - Barrier: Each thread wait at the barrier until all threads arrive.
  - Mutual exclusion: Define a block of code that only one thread at a time can execute.
- Synchronization is used to impose order constraints and to protect access to shared data.
- In Method II of computing π,

  ```
  sum = sum + 4.0/(1.0+x*x);
  ```

  is called a *critical section*.
- Critical section - a block of code executed by multiple threads that updates a shared variable, and the shared variable can only be updated by one thread at a time.

High level synchronizations in OpenMP

- `critical` construct: Mutual exclusion. Only one thread at a time can enter a *critical* section, e.g., Listing 4.7 shows a simple usage of this construct.

  ```
  #pragma omp critical
  ```

```
float result;
......
#pragma omp parallel
{
  float B; int i, id, nthrds;
  id = omp_get_thread_num();
  nthrds = omp_get_num_threads();
  for(i = id; i < nthrds; i+= nthrds) {
    B = big_job(i);
  }
  #pragma omp critical
  result += calc(B);
  ......
}
```

Listing 4.7: Critical

- `atomic` construct: Basic form. Provides mutual exclusion but only applies to the update of a memory location.
- `#pragma omp atomic`. See an example in Listing 4.8.

The statement inside the *atomic* must be one of the following forms:

- $x\ op = expr$, where $op \in (+=, -=, *=, /=, \% =)$
- $x++$
- $++x$
- $x--$
- $--x$

```
#pragma omp parallel
{
  ......
  double tmp, B;
  B = calc();
  tmp = big_calc(B);
  #pragma omp atomic
  X += tmp;
  ......
}
```

Listing 4.8: atomic construct

- `barrier` construct: Each thread waits until all threads arrive. See Listing 4.9.

  `#pragma omp barrier`

```
#pragma omp parallel
{
  int id=omp_get_thread_num();
  A[id]=calc1(id);
  #pragma omp barrier
  B[id]=calc2(id, A);
  ......
}
```

Listing 4.9: Barrier

4.2.4 Worksharing in OpenMP

Worksharing Construct

- A parallel construct by itself creates an SPMD program, i.e., each thread redundantly executes the same code.
- How do you split up pathways through the code between threads within a team? *Worksharing*.
- Worksharing
  - `for` construct or loop construct: Splits up a loop iterations among the threads in a team.

  - `sections/section` constructs
  - `task` construct
  - `single` construct
- The following construct is also discussed as synchronization construct.
  - `master` construct

4.2.5 Loop Construct

Loop construct: `#pragma omp for`

Example 4.2.5. *Given arrays $A[N]$ and $B[N]$. Find $A[i] = A[i] + B[i], 0 \leq i \leq N-1$. Listing 4.10 shows the for loop that does the element-wise vector addition. Listing 4.12 parallelizes the* `for` *loop manually, while Listing 4.11 shows how using* `for` *construct can conveniently parallelize the for loop.*

```
......
//Serial
for(i=0; i< N; i++)
  a[i]=a[i]+b[i];
```

Listing 4.10: Computing A=A+B

```
......
//Using loop construct
#pragma omp parallel
#pragma omp for
{
  for(i=0; i<N; i++)
    a[i] = a[i] + b[i];
}
```

Listing 4.11: Computing A=A+B in parallel using `for` construct

```
//Using parallel construct
#pragma omp parallel
{
  int id, i, nthrds, istart,
      iend;
  id = omp_get_thread_num();
  nthrds = omp_get_num_threads
      ();
  istart = id * N/nthrds;
  iend = (id+1) * N/nthrds;
  if(id==nthrds-1)
  iend = N;
  for(i=istart; i<iend; i++)
    a[i] = a[i] + b[i];
}
```

Listing 4.12: Computing A=A+B in parallel

Clauses Supported by the Loop Construct

- `private`
- `firstprivate`
- `lastprivate`
- `reduction`
- `schedule`
- `ordered`
- `nowait`
- `collapse`

Schedule Clause

- The **schedule** clause specifies how the iterations of the loop are assigned to the threads in a team.
- The syntax is: `#pragma omp for schedule(kind [, chunk])`.
- Schedule kinds:

  - **schedule(static [,chunk])**: Deals out blocks of iterations of size `chunk` to each thread in a round robin fashion.
    * The iterations can be assigned to the threads before the loop is executed.
  - **schedule(dynamic [,chunk])**: Each thread grabs `chunk` size of iterations off a queue until all iterations have been handled. The default is 1.
    * The iterations are assigned while the loop is executing
  - **schedule(guided [,chunk])**: Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size `chunk` as the calculation proceeds. The default is 1.
  - **schedule(runtime)**: Schedule and chunk size taken from the OMP_SCHEDULE environment variable.
    * For example, `export OMP_SCHEDULE="static,1"`
  - **schedule(auto)**: The selection of the schedule is determined by the implementation.
- Most OpenMP implementations use a roughly block partition.
- There is some overhead associated with schedule.
- The overhead for `dynamic` is greater than `static`, and the overhead for `guided` is the greatest.
- If each iteration of a loop requires roughly the same amount of computation, then it is likely that the default distribution will give the best performance.
- If the cost of the iterations decreases linearly as the loop executes, then a static schedule with small chunk size will probably give the best performance.
- If the cost of each iteration can not be determined in advance, then `schedule(runtime)` can be used.

Ordered Construct/Clause

- `ordered` construct: This is a synchronization construct. It allows one to execute a structured block within a parallel loop in sequential order.
- An `ordered` clause has to be added to the parallel region in which the `ordered` construct appears; it informs the compiler that the construct occurs.
- See Listing 4.13 for an example.

```
#pragma omp parallel private(i, TID) shared(a)
#pragma omp for ordered
for(i=0; i<n; i++) {
  TID = omp_get_thread_num();
  printf("Thread %d updates %dth item in the array\n", TID, i);
  a[i] += i;
  #pragma omp ordered
  printf("Thread %d prints %dth value of the array\n", TID, i);
}
```

Listing 4.13: Ordered clause and construct (1)

Basic Approach to Parallelizing a Loop

- Find compute intensive loops
- Make the loop iterations independent, so they can safely execute in any order without loop carried dependencies.
- Place the appropriate OpenMP directives and test.

Loop Carried Dependency

- The computation of one iteration depends on the results of one or more previous iterations.
- The program could compile without errors. However, the result can be incorrect or unpredictable.
- A loop with loop-carried dependency cannot, in general, be correctly parallelized by OpenMP, unless the loop-carried dependency is removed.
- See an example in Listing 4.14.

```
fibo[0] = fibo[1] = 1;
#pragma omp parallel num_threads(thread_count)
#pragma omp for
for(i = 2; i < n; i++)
fibo[i] = fibo[i-1] + fibo[i-2];
```

Listing 4.14: Loop carried dependency

Example 4.2.6. *Removing a loop carried dependency (e.g., see Listing 4.15).*

```
//Loop dependency
int i, j, A[MAX];
j=5;
for(i=0; i<MAX; i++){
  j+=2;
  A[i]=big(j);
}
//Removing loop dependency
int i, A[MAX];
#pragma omp parallel
#pragma omp for
for(i=0; i<MAX; i++){
  int j=5+2*(i+1);
  A[i]=big(j);
  //Loop dependency
  int i, j, A[MAX];
  j=5;
  for(i=0; i<MAX; i++){
    j+=2;
    A[i]=big(j);
  }
  //Removing loop dependency
  int i, A[MAX];
  #pragma omp parallel
  #pragma omp for
```

```
for(i=0; i<MAX; i++){
  int j=5+2*(i+1);
  A[i]=big(j);
}
```

Listing 4.15: Loop carried dependency

- Loop carried dependency: Dependencies between instructions in different iterations of a loop;
- What are the dependencies in the following loop?

```
for(i=0; i<N; i++) {
  B[i]=tmp;
  A[i+1]=B[i+1];
  tmp=A[i];
}
```

It helps to unroll the loop to see the dependencies (see Listing 4.16).

```
i=0:
B[0]=tmp;
A[1]=B[1];
tmp=A[0];

i=1:
B[1]=tmp;
A[2]=B[2];
tmp=A[1];

i=2:
B[2]=tmp;
A[3]=B[3];
tmp=A[2];
......
```

Listing 4.16: Dependency

Reduction

```
......
double ave=0.0, A[MAX];
int i;
for(i=0;i<MAX;i++){
  ave+=A[i];
}
ave = ave/MAX;
......
```

Listing 4.17: Reduction

In Listing 4.17, we are aggregating multiple values into a single value—**reduction**. Reduction operation is supported in most parallel programming environments.

- OpenMP reduction clause: *reduction(op:list)*.
- Inside a parallel for worksharing construct
  - A local copy of each list variable is made and initialized depending on the operation specified by the operator "op".
  - Each thread updates its own local copy
  - Local copies are aggregated into a single value.

Reduction operation in Listing 4.17 can be parallelized using OpenMP loop construct with reduction clause as shown in Listing 4.18.

```
......
double ave=0.0, A[MAX];
int i;
#pragma omp parallel for reduction(+:ave)
for(i=0;i<MAX;i++){
  ave+=A[i];
}
ave = ave/MAX;
......
```

Listing 4.18: Using reduction clause

- Table 4.2 shows the associative operands that can be used with reduction (for C/C++) and their common initial values.

Table 4.2: Associative operands that can be used with reduction (for C/C++) and their common initial values

Op	Initial value	Op	Initial value
+	0	&	~0
*	1	|	0
-	0	^	0
min	Large number (+)	&&	1
max	Most neg. number	||	0

Collapse Clause

Listing 4.19 gives a simple example of nested loop.

```
void work(int a, int j, int k);
void main() {
  int j, k, a;
  int m = 2, n = 5;
  #pragma omp parallel num_threads(4)
  {
    #pragma omp for private(j,k)
    for (k=0; k<m; k++)
    for (j=0; j<n; j++) {
        printf("%d %d\n", k, j);
        work(a,j,k);
      }
    }
  }
```

Listing 4.19: collapse

- In Listing 4.20, the iterations of the `k` and `j` loops are (manually) collapsed into one loop, and that loop is then divided among the threads in the current team.

```
  void work(int a, int j, int k);
  void main() {
    int t, a;
    int m = 2, n = 5;
    #pragma omp parallel num_threads(4)
    {
      #pragma omp for private(j,k) schedule(static,2)
      for(t=0; t<10; t++) {
        printf("%d %d %d\n", omp_get_thread_num(), (t/n)%m, t%n);
        work(a,t%n,(t/n)%m);
      }
    }
  }
```

Listing 4.20: collapse

Listing 4.21 uses `collapse` clause to turn the nested `for` loop into a single for loop in order to increase the total number of iteration for efficient parallelism.

```
  void work(int a, int j, int k);
  void main() {
    int j, k, a;
    int m = 2, n = 5;
    #pragma omp parallel num_threads(2)
    {
      #pragma omp for collapse(2) ordered private(j,k) schedule(static
          ,2)
      for (k=0; k<m; k++)
      for (j=0; j<n; j++) {
        #pragma omp ordered
        printf("%d %d %d\n", omp_get_thread_num(), k, j);
        /* end ordered */
        work(a,j,k);
      }
    }
  }
```

Listing 4.21: collapse

4.2.5.1 Examples of More Clauses

lastprivate Clause

Listings 4.22 and 4.23 show examples of using `lastprivate` clause.

```
void lastpriv (int n, float *a, float *b) {
  int i, a, n = 5;
  ......
  #pragma omp parallel private(i) lastprivate(a)
  #pragma omp for
  for (i=0; i<n; i++) {
    a = i+1;
    printf("Thread %d has a value of a = %d for i = %d\n",
        omp_get_thread_num(),a,i);
  } /*-- End of parallel for --*/
  printf("Value of 'a' after parallel for: a=%d\n",a);
}
```

Listing 4.22: lastprivate example

```
void sq2(int n, double *lastterm) {
  double x; int i;
  #pragma omp parallel for lastprivate(x)
  for(int i=0; i<1000; i++) {
    x=a[i]*a[i]+b[i]*b[i];
    b[i]=sqrt(x);
  }
  /*x has the value it held for the last sequential iteration, i.e.,
     for i=(1000-1)*/
  *lastterm = x;
}
```

Listing 4.23: lastprivate example

nowait Clause

Listing 4.24 shows an example of using `nowait` clause.

```
#include <math.h>
void nowait_example2(int n, float *a, float *b, float *c, float *y,
    float *z) {
  int i;
  #pragma omp parallel
  {
    #pragma omp for schedule(static)
    for (i=0; i<n; i++) {
      c[i] = (a[i] + b[i]) / 2.0f;
    }/*implicit barrier*/
    #pragma omp for schedule(static) nowait
    for (i=0; i<n; i++) {
      z[i] = sqrtf(c[i]);
    }/*no implicit barrier due to nowait clause*/
```

Table 4.3: Loops in canonical form

			`index++`	
			`++index`	
		`index < end`	`index--`	
		`index <= end`	`--index`	
for(	`index=start;`	`index >= end;`	`index += incr`	)
		`index > end`	`index -= incr`	
			`index=index+incr`	
			`index=incr+index`	
			`index=index-incr`	

```
        #pragma omp for schedule(static) nowait
        for (i=1; i<=n; i++)
        y[i] = z[i-1] + a[i];
        /*no implicit barrier due to nowait clause*/
    }/*implicit barrier at the end of a parallel region, cannot be
      removed*/
}
```

Listing 4.24: nowait example

More on for Construct

- OpenMP parallelizes `for` loops that are in canonical form.
- `for` loop must not contain statements that allow the loop to be exited prematurely, such as `break`, `return`, or `exit` statements. The `continue` statement is allowed.
- Loops in canonical form take one of the forms shown in Table 4.3.

4.2.5.2 Combined Parallel Worksharing Constructs

Combined parallel worksharing constructs are shortcuts that can be used when a parallel region comprises precisely one worksharing construct. Listing 4.26 combines the two compiler directives in Listing 4.25 into one.

```
#pragma omp parallel
#pragma omp for
for-loop
```

Listing 4.25: Full construct

```
//combined for version
#pragma omp parallel for
for-loop
```

Listing 4.26: Combined construct

4.2.5.3 Single Worksharing Construct

- The `single` construct denotes a block of code that is executed by only one thread.
- Syntax: (Listing 4.27)

```
#pragma omp single [clause[[,] clause]...]
```

```
    structured block
```

Listing 4.27: Single syntax

- clauses: `private`, `firstprivate`, `copyprivate`, `nowait`
- A barrier is implied at the end of the *single* block, unless a `nowait` clause is specified.
- This construct is ideally suited for I/O or initialization.
- See an example in Listing 4.28.

```
void work1() {}
void work2() {}
void single_example() {
  #pragma omp parallel
  {
    #pragma omp single
    printf("Beginning work1.\n");
    work1();
    #pragma omp single
    printf("Finishing work1.\n");
    #pragma omp single nowait
    printf("Finished work1 and beginning work2.\n");
    work2();
  }
}
```

Listing 4.28: Single construct

4.2.5.4 Copyprivate Clause

- `copyprivate` clause is used with `single` construct only.
- It provides a mechanism to broadcast the value of a private variable from one thread to the rest of the team. Listing 4.29 shows an example of using `copyprivate` clause.

```
int TID;
float rate=1.2;
omp_set_num_threads(4);
#pragma omp parallel private(rate,TID)
{
  TID = omp_get_thread_num();
  #pragma omp single copyprivate(rate)
  {
    rate = rand()*1.0/RAND_MAX;
  }
  printf("Value for variable rate: %f by thread %d\n",rate, TID);
}
```

Listing 4.29: copyprivate example

4.2.5.5 Master Construct

`master` construct:

- The `master` construct specifies a structured block that is executed by the master thread of the team.
- There is no implied barrier either on entry to, or exit from, the master construct.

The Number of Threads Active

The `omp_set_dynamic` routine enables or disables dynamic adjustment of the number of threads available for the execution of subsequent parallel regions.

- omp_set_dynamic() – A call to this function with nonzero argument allows OpenMP to choose any number of threads between 1 and the set number of threads.

Example 4.2.9. *The following code allows the OpenMP implementation to choose any number of threads between 1 and 8.*

```
omp_set_dynamic(1);
#pragma omp parallel num_threads(8)
```

Example 4.2.10. *The following code only allows the OpenMP implementation to choose 8 threads. The action in this case is implementation dependent.*

```
omp_set_dynamic(0);
#pragma omp parallel num_threads(8)
```

- omp_get_dynamic() – You can determine the default setting by calling this function (returns TRUE if dynamic setting is enabled, and FALSE if disabled).

4.2.6 Sections/Section Construct

- `sections` directive enables specification of **task parallelism**
- The `sections` worksharing construct gives a different structured block to each thread.
- Syntax (Listing 4.30):

```
#pragma omp sections [clause[[,] clause]...]
{
  [#pragma omp section]
  structured block
  [#pragma omp section]
  structured block
  ...
}
```

Listing 4.30: Sections syntax

- clauses: `private`, `firstprivate`, `lastprivate`, `reduction`, `nowait`
- Each section must be a structured block of code that is independent of other sections.
- There is an implicit barrier at the end of a sections construct unless a `nowait` clause is specified.

Examples — firstprivate

See the example in Listing 4.31.

```
#include <omp.h>
#include <stdio.h>
#define NT 4
int main( ) {
  int section_count = 0;
  omp_set_dynamic(0);
  omp_set_num_threads(NT);
  #pragma omp parallel
  #pragma omp sections firstprivate( section_count )
  {
    #pragma omp section
    {
      section_count++;
      printf( "section_count %d\n", section_count );
    }
    #pragma omp section
    {
      section_count++;
      printf( "section_count %d\n", section_count );
    }
  }
  return 0;
}
```

Listing 4.31: firstprivate example

Example – Parallel Quicksort

Example 4.2.11. *Parallelize the sequential* quicksort *program (Listing 4.32 or* `qsort_serial.c`*) using OpenMP* `sections/section` *construct.*

```
q_sort(left, right, data) {
  if(left < right){
    q = partition(left, right, data);
    q_sort(left, q-1, data);
    q_sort(q+1, right, data);
  }
}
partition(left, right, float *data) {
  x = data[right];
  i = left-1;
  for(j=left; j<right; j++) {
    if(data[j] <= x){
      i++;
      swap(data, i, j);
    }
  }
  swap(data, i+1, right);
  return i+1;
}
```

Listing 4.32: Quick sort

4.2.7 Task Worksharing Construct

- Tasks are independent units of work.
- Threads are assigned to perform the work of each task.
  - Tasks may be deferred
  - Tasks may be executed immediately
- The runtime system decides which of the above
- A *task* is composed of:
  - Code to execute
  - A data environment
  - Internal control variables, such as
    - `OMP_NESTED` – `TRUE` or `FALSE`, controls whether nested parallelism is enabled.
    - `omp_set_dynamic()`, `omp_get_dynamic` – controls the dynamic adjustment of threads.
    - `omp_set_num_threads()`, etc.

Task Construct Syntax
#pragma omp task [clause[[,] clause]...]
structured block

where clause can be

- if(*expr*): if *expr*=FALSE, then the task is immediately executed.
- shared
- private
- firstprivate
- default(shared |none)
- tied/untied
- final(expr)

- Variables that are shared in all scopes enclosing the task construct remain shared in the generated task. All other variables are implicitly determined firstprivate.
- `tied/untied`: Upon resuming a suspended task region, a `tied` task must be executed by the same thread again. With an `untied` task, there is no such restriction and any thread in the team can resume execution of the suspended task.
- `if(expr)` clause - If `expr` is evaluated to false, the task is undeferred and executed immediately by the thread that was creating the task.
- `final(expr)` clause - For recursive and nested applications, it stops task generations at a certain depth where we have enough tasks (or parallelism). If `expr` is evaluated to true, the task is considered to be final and no additional tasks are generated.

- Two activities: *packaging* and *execution*
  - Each encountering thread packages a new instance of task

  - Some thread in the team executes the task at some time later or immediately.

- Task barrier: The **taskwait** directive:

Task Construct Example

Example 4.2.12. *Task construct example (Listing 4.33)*

```
......
#pragma omp parallel
{
  #pragma omp single private(p)
  {
    p=list_head;
    while(p){
      #pragma omp task
      processwork(p);
      p=p->next;
    }
  }
}
```

Listing 4.33: Task list

When tasks are guaranteed to be completed?

- At thread or task barriers
- At the directive: `#pragma omp barrier`
- At the directive: `#pragma omp taskwait`

Example 4.2.13. *Task completion*

```
#pragma omp parallel
{
  #pragma omp task
  foo();
  #pragma omp barrier
  #pragma omp single
  {
    #pragma omp task
    bar();
  }
}
```

Listing 4.34: Task completion example

In Listing 4.34, for the first task construct, multiple tasks are created, one for each thread. All foo() tasks are guaranteed to be completed at the omp barrier. For the second task construct, only one bar() task is created. It is guaranteed to be completed at the implicit barrier, or the end of structured block.

Example 4.2.14. *Understanding Task Construct What will be the output of program in Listing 4.35?*

```
int main(int argc, char *argv[]){
  #pragma omp parallel num_threads(2)
  {
    printf("A ");
    printf("soccer ");
    printf("match ");
  }
  printf("\n");
  return 0;
}
```

Listing 4.35: Asimple example

Example 4.2.15. *Understanding Task Construct What will be the output of program in Listing 4.36?*

```
int main(int argc, char *argv[]){
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("A ");
      printf("soccer ");
      printf("match ");
    }
  }
  printf("\n");
  return 0;
}
```

Listing 4.36: Tasking example

Example 4.2.16. *Understanding Task Construct What will be the output of program in Listing 4.37?*

```
int main(int argc, char *argv[]){
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("A ");
      #pragma omp task
      printf("soccer ");
      #pragma omp task
      printf("match ");
    }
  }
  printf("\n");
  return 0;
}
```

Listing 4.37: Tasking example

Example 4.2.17. *Understanding Task Construct What will be the output of program in Listing 4.38?*

```
int main(int argc, char *argv[]){
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("A ");
      #pragma omp task
      printf("soccer ");
      #pragma omp task
      printf("match ");
      printf("is fun to watch ");
    }
  }
  printf("\n");
  return 0;
}
```

Listing 4.38: Tasking example

Example 4.2.18. *Understanding Task Construct What will be the output of program in Listing 4.39?*

```
int main(int argc, char *argv[]){
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("A ");
      #pragma omp task
      printf("soccer ");
      #pragma omp task
      printf("match ");
      #pragma omp taskwait
      printf("is fun to watch ");
    }
  }
  printf("\n");
  return 0;
}
```

Listing 4.39: Tasking example

Example 4.2.19. *Tree traversal using task (Listing 4.40)*

```
void traverse(node *p){
  if(p->left)
    #pragma omp task
    traverse(p->left);
  if(p->right)
    #pragma omp task
    traverse(p->right);
  process(p->data);
}
```

Listing 4.40: Tree traversal using task

Example 4.2.20. *Tree traversal using task (Listing 4.41)*

```
void traverse(node *p){
  if(p->left)
      #pragma omp task
      traverse(p->left)
  if(p->right)
      #pragma omp task
      traverse(p->right)
  #pragma omp taskwait
  process(p->data);
}
```

Listing 4.41: Tree traversal using task

Example 4.2.21. *Write an OpenMP parallel program for computing the nth Fibonacci number. Compare the performance of the parallel implementation to the sequential one.*

Task Switching

`untied` clause example:

```
#define ONEBILLION 1000000000L
#pragma omp parallel
{
  #pragma omp single
  {
    for(i=0; i<ONEBILLION; i++)
      #pragma omp task
      process(item[i]);
  }
  ......
  /* Untied task: any other thread is eligible to resume
  the task generating loop*/
  #pragma omp single
  {
    for(i=0; i<ONEBILLION; i++)
      #pragma omp task untied
      process(item[i]);
  }
}
```

Listing 4.42: Task switching

4.3 Summary

- OpenMP execution model
- Create a parallel region in OpenMP programs
- Worksharing constructs including loop construct, sections/section construct, and task construct are discussed.
- We have studied the following constructs:
  - `parallel`
  - `critical`
  - `atomic`
  - `barrier`
  - `master`
  - `for`
  - `sections/section`
  - `task`
  - `single`
- Clauses that can be used with some of the constructs
- Usage of OpenMP library functions and environment variables
- False sharing and race condition.

4.4 Exercises

Objectives

- Compile and run an OpenMP program written in C
- Use OpenMP `parallel`, `for`, `sections/section` and `task` constructs to parallelize sequential programs.
- Use synchronization constructs including `critical`, `atomic`, and `barrier`, in OpenMP programs where applicable.
- Understand the problems of false sharing and race condition in shared memory programming, and apply proper techniques to eliminate such problems in OpenMP programs.
- Understanding various constructs in OpenMP.

Instructions

1. Baseline codes, as well as `Makefile` (for compilation of the programs) and run script `run.sh` (for running of the programs) are provided for some questions. You may use any IDE for C/C++, such as CodeLite or Code::Blocks, or you may use commands from a terminal to compile and run OpenMP programs.

2. **Note:** In the case of submission of codes for any type of assessments, your code must be able to compile and run from Linux command line (or Linux terminal), as that is the only way we are going to use to mark your codes.

Programming Problems

1. Compile and run `parallel_region.c`. Change the number of threads in the parallel region by setting parallel construct clause, library function, or environment variable, respectively.

2. Write a sequential program for computing the number π using the method discussed in the class (see Section 4.2.2). Parallelize the serial program using OpenMP parallel construct. Implement the following techniques discussed in the class:

   (a) the approach with false sharing (Method 1);

   (b) the approach with race condition (Method 2);

   (c) the approach with race condition eliminated;

   Evaluate the performances of the sequential and parallel codes respectively by measuring their elapsed time, and show the speedups of your parallel implementations.

   Note: A base code is given for this problem in `Lec4_codes` folder. There is a `Makefile` provided in the same folder. Your code must be able to compile using this makefile. For example, open a Linux terminal with the correct path to the program files. From the terminal, type `make clean` to remove all existing executables; then type `make` to compile all `.c` files. Upon successful compilation, run the code using its respective executable file name like `./pi 5` for the number π example, where the argument '5' gives the number of repetitive runs (to get an average runtime).

3. Implement an OpenMP program that computes the histogram of a set of M^2 integers (range from 0 to 255) that are stored in an $M \times M$ two dimensional array. Answer the following questions.

   (a) What is the fine level task in this problem?

   (b) How would you assign the fine level tasks to the threads? Experiment with different assignment strategies in your parallel program.

4. Given the pseudo code of square matrix multiplication in Listing 4.43, implement matrix multiplication, $C = AB$, where $C \in \mathbb{R}^{N\times N}$, $A \in \mathbb{R}^{N\times M}$, and $B \in \mathbb{R}^{M\times N}$, in parallel using OpenMP `for` construct. Consider the following questions for your implementation.

   (a) How many dot products are performed in your serial matrix multiplication?

   (b) Use the `OMP_NUM_THREADS` environment variable to control the number of threads and plot the performance with varying number of threads.

   (c) Experiment with two cases in which (i) only the outermost loop is parallelized; (ii) only the second inner loop is parallelized. What are the observed results from these two cases?

   (d) How does `collapse` clause work with `for` construct? Does it improve the performance of your parallel matrix multiplication?

   (e) Experiment with differing matrix sizes, such as $N \gg M$ or $M \gg N$. How does it impact the performance of your parallel implementation?

(f) Experiment with `static` and `dynamic` scheduling clauses to see the effects on the performance of your parallel program. For each schedule clause, determine how many dot products each thread performs in your program.

```
......
for(int i=0; i<N; i++)
  for(int j=0; j<N; j++){
    C[i][j] = 0.0;
    for(int k=0; k<M; k++)
    C[i][j] = C[i][j]+A[i][k]*B[k][i];
  }
......
```

Listing 4.43: Square matrix multiplication

5. Count sort is a simple serial algorithm that can be implemented as follows (Listing 4.44).

```
void count_sort(int a[], int n) {
  int i, j, count;
  int *temp = malloc(n * sizeof(int));
  for (i = 0; i < n; i++) {
    count = 0;
    for (j = 0; j < n ; j++)
    if (a[j] < a[i])
      count++;
    else if(a[j] == a[i] && j < i)
      count++;
    temp[count] = a[i];
  }
  memcpy (a, temp, n * sizeof(int));
  free (temp);
}
```

Listing 4.44: Count sort

The basic idea is that for each element `a[i]` in the list `a`, we count the number of elements in the list that are less than `a[i]`. Then we insert `a[i]` into a temporary list using the index determined by the count. The algorithm also deals with the issue where there are elements with the same value.

- Write a C program that includes both serial and parallel implementations of count-sort. In your parallel implementation, specify explicitly the data scope of each variable in the parallel region.
- Compare the performance between the serial and parallel versions.

6. In what circumstances do you use the following environment variables? What are the respective equivalent library functions to these environment variables?

(a) `OMP_NUM_THREADS`

(b) `OMP_DYNAMIC`

   (c) `OMP_NESTED`

7. What is the difference between the two code snippets shown in Listing 4.45 in terms of memory access?

```
//Code snippet 1
for(int i=0; i<n; i++)
  for(int j=0; j<n; j++)
    sum += a[i][j];

//Code snippet 2
for(int j=0; j<n; j++)
  for(int i=0; i<n; i++)
    sum += a[i][j];
```

Listing 4.45: Code snippets with nested for loops

8. Given the code snippet in Listing 4.46, collapse the nested for loops into a single for loop manually.

```
int m = 4;
int n = 1000;
......
for(int i=0; i<m; i++)
  for(int j=0; j<n; j++)
    a[i][j] = i+j+1;
......
```

Listing 4.46: Code snippet for a nested for loop

9. Design simple examples to illustrate the functions of various OpenMP synchronization constructs including `barrier`, `critical`, `atomic`, and `ordered`.

10. Suppose OpenMP does not have the reduction clause. Show how to implement an efficient parallel reduction for finding the minimum or maximum value in an array.

11. Implement the problem of computing the *Fibonacci numbers* both in serial and parallel. For your parallel version, use OpenMP `task` and `sections/section` constructs respectively. Compare the performances of these 3 implementations, i.e., serial, parallel using `sections/section` construct, parallel using `task` construct, by running the programs for different problem sizes and thread numbers.

12. Implement *quicksort* algorithm both in serial and parallel. For your parallel version, use OpenMP `task` and `sections/section` constructs respectively. Compare the performances of these 3 implementations, i.e., serial, parallel using `sections/section` construct, parallel using `task` construct, by running the programs for different problem sizes and thread numbers. A base code (`qsort_omp.c`) is given in `Lec6_codes` folder. Put your results in a table like the one shown in Table 4.4. Answer (or consider) the following question:

   (a) What are the differences of `sections/section` and `task` constructs?

(b) Understand the usage of various clauses of `task` and `sections/section` constructs by applying them in your implementations where applicable, and observe the impacts on the performance of your programs.

Table 4.4: Table for benchmarking the performance of parallel *quicksort*s

	Seq	Par(sections/section)		Par(task)			
	1 thread			4 threads		8 threads	
Size	Time(*s*)	Time(*s*)	Speedup	Time(*s*)	Speedup	Time(*s*)	Speedup
10^5							
Threshold[1]							
10^6							
Threshold							
10^7							
Threshold							
10^8							
Threshold							

(Give the machine specs the above results are obtained.)

[1]Put in the value of threshold data size (to exit the parallel sorting) if used. Not valid for sequential sort.

13. Scan operation, or all-prefix-sums operation, is one of the simplest and useful building blocks for parallel algorithms ([Blelloch 1990]). Given a set of elements, $[a_0, a_1, \cdots, a_{n-1}]$, the scan operation associated with addition operator for this input is the output set $[a_0, (a_0 + a_1), \cdots, (a_0 + a_1 + \cdots + a_{n-1})]$. For example, the input set is $[2, 1, 4, 0, 3, 7, 6, 3]$, then the scan with addition operator of this input is $[2, 3, 7, 7, 10, 17, 23, 26]$.

    It is simple to compute scan operation in serial, see Listing 4.47.

```
scan(out[N], in[N]) {
  i=0;
  out[0]=in[0];
  for(i=1;i<N;i++){
    out[i]=in[i]+out[i-1];
  }
}
```

Listing 4.47: Sequential algorithm for computing scan operation with '+' operator

Sometimes it is useful for each element of the output vector to contain the sum of all the previous elements, but not the element itself. Such an operations is called prescan. That is, given the input $[a_0, a_1, \cdots, a_{n-1}]$, the output of prescan operation with addition operator is $[0, a_0, (a_0 + a_1), \cdots, (a_0 + a_1 + \cdots + a_{n-2})]$.

The algorithm for scan operation in Listing 4.47 is inherently sequential, as there is a loop carried dependence in the `for` loop. However, Blelloch [1990] gives an algorithm for calculating the scan operation in parallel (see [Blelloch 1990, Pg. 42]). Based on this algorithm, (i) implement the parallel algorithm for prescan using OpenMP; and (ii) implement an OpenMP parallel program for scan operation based on the prescan parallel algorithm.

Bibliography

Blelloch, G. E. (1990). Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University.

Chapman, B., Jost, G., and van der Pas, R. (2007). *Using OpenMP: Portable Shared Memory Paralle Programming*. The MIT Press, Cambridge.

Trobec, R., Slivnik, B., Bulić, P., and Robič, B. (2018). *Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms*. Springer.

Chapter 5

Distributed Memory Programming using MPI

The contents in this chapter are based on sources [Grama et al. 2003] and [Trobec et al. 2018].

5.1 Message Passing Model and MPI

5.1.1 Message Passing Model

- The usual underlying hardware: a collection of processors, each with its own local memory (or address space).
- A process: a task
  - A process is (traditionally) a program counter and address space.
  - Processes may have multiple threads (program counters and associated stacks) sharing a single address space.
- Every process can communicate with every other processes.

5.1.2 MPI

- MPI (Message Passing Interface) standard is the most popular message passing specification that supports parallel programming.
- MPI is a message passing library specification.
- MPI is for communication among processes, which have separate address spaces.
- Inter-process communication consists of
  - synchronization
  - movement of data from one process' address space to another's.
- MPI is not
  - a language or compiler specification
  - a specific implementation or product
- MPI is for parallel computers, clusters, and heterogeneous networks.
- MPI versions:
  - MPI-1 supports the classical message-passing programming model: basic point-to-point communication, collectives, datatypes, etc. MPI-1 was defined (1994) by a

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of the calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

broadly based group of parallel computer vendors, computer scientists, and applications developers.
- MPI-2 was released in 1997
- MPI-2.1 (2008) and MPI-2.2 (2009) with some corrections to the standard and small features
- MPI-3 (2012) added several new features to MPI.
- MPI-4 (2021) added several new features to MPI.
- The Standard itself: at http://www.mpi-forum.org. All MPI official releases, in both postscript and HTML.

5.1.3 The Minimal Set of MPI Functions

Starting and Terminating MPI Programs

- `int MPI_Init(int *argc, char ***argv)`
  The parameters are from the parameters of the main C program.
  - Initialization: must call this prior to other MPI functions
- `int MPI_Finalize()`
  - Must call at the end of MPI program
- Return codes

```
MPI_SUCCESS
MPI_ERROR
```

Communicators

- `MPI_Comm`: communicator – communication domain
  - Group of processes that can communicate with one another
  - Supplied as an argument to all MPI message passing functions
  - Process can belong to multiple communication domain
- `MPI_COMM_WORLD`: root communicator – includes all the processes

Communicator Inquiry Functions

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  - Determine the number of processes

- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - index of the calling process
  - value: 0 — communicator size - 1

Simple MPI Program — "Hello World"

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
  int num_procs, myrank;

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  printf("From process %d out of %d, Hello World!\n",
  myrank, num_procs);

  MPI_Finalize();
  return 0;
}
```

- Header file for MPI program — `#include <mpi.h>`
- Each active MPI process executes its own copy of the program.
- The first MPI function call made by every MPI process is the call to `MPI_Init`, which allows the system to do any setup needed to handle further calls to the MPI library.
- When MPI has been initialized, every active process becomes a member of a communicator called `MPI_COMM_WORLD`.

- A communicator defines a *communication domain* – a set of processes that are allowed to communicate with each other – a communication context.
- A default communicator: `MPI_COMM_WORLD`, which includes all the processes started by the user.
- Communicators are used as arguments to all message transfer MPI routines.
- A process can belong to many different (possibly overlapping) communication domains.

- A process calls function `MPI_Comm_rank` to determine its rank within a communicator.
- It calls `MPI_Comm_size` to determine the total number of processes in a communicator.

```
int id, p;
MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Comm_size(MPI_COMM_WORLD, &p);
```

- After a process has completed all of its MPI library calls, it calls function `MPI_Finalize`, allowing the system to free up resources (such as memory) that have been allocated to MPI, and shuts down MPI environment.

5.1.4 Compiling and running MPI Programs

- MPI is a library. Applications can be written in C, C++ or Fortran and appropriate calls to MPI can be added where required
- Compilation
  - Regular applications: `gcc test.c -o test`
  - MPI applications: `mpicc hello_world.c -o test`
- Execution
  - Regular applications: `./test`
  - MPI applications (running with 16 processes):
    - `mpiexec -n 16 ./test` or
    - `mpirun -np 16 ./test`

5.1.5 Installing MPI Standard Implementation

MPI standard implementation can be already provided as part of the OS, often as MPICH[1]or OpenMPI[2]. If it is not, it can usually be installed through the provided package management, e.g., `apt` in Ubuntu:

- for MPICH,

  ```
  sudo apt-get install libmpich-dev
  ```

- for OpenMPI,

  ```
  sudo apt-get install libopenmpi-dev
  ```

5.2 Data Communication

5.2.1 Data Communication

- Data communication in MPI: One process sends a copy of data to another process (or a group of processes), and the other process receives it.
- Communication requires the following information:
  - Sender: the receiver, data count, data type, a user defined tag for the message
  - Receiver: the sender, data count, data type, the tag for the message

5.2.2 MPI Blocking Send and Receive

MPI Basic (Blocking) Send

```
int MPI_Send(void *buf,int count,MPI_Datatype datatype,
int dest,int tag,MPI_Comm comm)
```

- The message buffer is described by `buf`, `count`, `datatype`.

[1] https://www.mpich.org/
[2] https://www.open-mpi.org/

- The target process is specified by `dest` and `comm`.
  - `dest` is the rank of the target process in the communicator specified by `comm`.
- `tag` is a user-defined type for the message
- When this function returns, the data has been delivered to the communication system and the send buffer (described by `buf, count, datatype`) can be reused. The message may not have been received by the target process.

MPI Basic (Blocking) Receive

```
int MPI_Recv(void *buf,int count,MPI_Datatype datatype,
int source,int tag,MPI_Comm comm,MPI_Status *status)
```

- Waits until a matching (on source, datatype, tag, comm) message is received from the communication system, and the buffer can be used.
- Source is the rank of sender in communicator `comm`, or `MPI_ANY_SOURCE`.
- `status` contains further information: The MPI type `MPI_Status` is a struct with at least three members `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR` .
- `MPI_STATUS_IGNORE` can be used if we don't need any additional information

.

Simple Communication in MPI

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

int main(int argc, char ** argv){
  int rank, data[100];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  //Array data needs to be initialized
  if (rank == 0)
  MPI_Send(data, 100, MPI_INT, 1, 0, MPI_COMM_WORLD);
  else if (rank == 1)
  MPI_Recv(data, 100, MPI_INT, 0, 0, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);

  MPI_Finalize();
  return 0;
}
```

5.2.3 MPI Datatypes

MPI Datatype	C Datatype
`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`

Sending and Receiving Messages

- MPI allows specification of wildcard arguments for both `source` and `tag`.
- If `source` is set to `MPI_ANY_SOURCE`, then any process of the communication domain can be the source of the message.
- If `tag` is set to `MPI_ANY_TAG`, then messages with any tag are accepted.
- Only a receiver can use wildcard arguments. Sender must use a process rank and nonnegative tag.
- On the receive side, the message must be of length equal to or less than the length field specified.

- On the receiving end, the `status` variable can be used to get information about the `MPI_Recv` operation.
- The corresponding data structure contains:

```
typedef struct MPI_Status {
int MPI_SOURCE;
int MPI_TAG;
int MPI_ERROR;
};
```

- The `MPI_Get_count` function returns the source, tag and number of elements of datatype received

```
int MPI_Get_count(MPI_Status *status, /* in  */
MPI_Datatype datatype,        /* in  */
int *count                    /* out */)
```

5.2.4 Benchmarking the Performance

- Running time:
  - MPI provides a function called `MPI_Wtime` that returns a double-precision floating-point number of seconds that have elapsed since some point of time in the past.
  - Function `MPI_Wtick` returns a floating-point number that is the time in seconds between successive ticks of the clock, i.e., the precision of the result returned by `MPI_Wtime`.

-
  ```
  double MPI_Wtime()
  double MPI_Wtick()
  ```

5.3 Examples

Example 5.3.1. ***Rotating a token around processes connected via ring interconnect network*** *Consider a set of n processes arranged in a ring. Process 0 sends a token message, say "Hello!", to process 1; process 1 passes it to process 2; process 2 to process 3, and so on. Process $n-1$ sends back the message to process 0. Write an MPI program that performs this simple token ring.*

Listing 5.1 gives a simple implementation.

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &nproces);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if(myrank == 0)
  prev = nproces - 1;
else
  prev = myrank - 1;
if(myrank == (nproces - 1))
  next = 0;
else
  next = myrank + 1;
if(myrank == 0)
  strcpy(token, "Hello!");
MPI_Send(token, MSG_SZ, MPI_CHAR, next, tag, MPI_COMM_WORLD);
MPI_Recv(token, MSG_SZ, MPI_CHAR, prev, tag, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
printf("Process %d received token %s from process %d.\n", myrank,
    token, prev);
```

Listing 5.1: Token ring implementation one

Running the implementation in Listing 5.1 gives results as shown in Listing **??**. Why the result is not as expected?

```
DATE: Wed Sep 29 16:26:19 SAST 2021
Job is running on nodes  mscluster[30,47,50,55]
-------------------------------------------------------
SLURM: sbatch is running on mscluster0.ms.wits.ac.za
SLURM: job ID is 141789
SLURM: submit directory is /home-mscluster/hwang/PC/mpi_lab1
SLURM: number of nodes allocated is 4
SLURM: number of tasks is 8
SLURM: job name is PC_mpi_com
-------------------------------------------------------
Process 1 received token Hello! from process 0.
Process 7 received token hR???^ from process 6.
Process 0 received token h?p??^ from process 7.
Process 2 received token hRU7?^ from process 1.
Process 3 received token h?^??^ from process 2.
Process 4 received token h?? from process 3.
Process 6 received token h?^P?^ from process 5.
Process 5 received token h?6??^ from process 4.
```

Listing 5.2: A sample run result of Listing 5.1

Listing 5.3 gives an alternative implementation of communication, and Listing **??** shows the result of a sample run of this implementation.

```
if(myrank == 0) {
  strcpy(token, "Hello World!");
  MPI_Send(token, MSG_SZ, MPI_CHAR, next, tag, MPI_COMM_WORLD);
  MPI_Recv(token, MSG_SZ, MPI_CHAR, prev, tag, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
  printf("Process %d received token %s from process %d.\n", myrank,
      token, prev);
}
else {
  MPI_Recv(token, MSG_SZ, MPI_CHAR, prev, tag, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
  MPI_Send(token, MSG_SZ, MPI_CHAR, next, tag, MPI_COMM_WORLD);
  printf("Process %d received token %s from process %d.\n", myrank,
      token, prev);
}
```

Listing 5.3: Token ring implementation two

```
DATE: Wed Sep 29 16:26:19 SAST 2021
Job is running on nodes  mscluster[30,47,50,55]
------------------------------------------------------
SLURM: sbatch is running on mscluster0.ms.wits.ac.za
SLURM: job ID is 141789
SLURM: submit directory is /home-mscluster/hwang/PC/mpi_lab1
SLURM: number of nodes allocated is 4
SLURM: number of tasks is 8
SLURM: job name is PC_mpi_com
------------------------------------------------------
Process 1 received token Hello World! from process 0.
Process 2 received token Hello World! from process 1.
Process 3 received token Hello World! from process 2.
Process 4 received token Hello World! from process 3.
Process 5 received token Hello World! from process 4.
Process 6 received token Hello World! from process 5.
Process 7 received token Hello World! from process 6.
Process 0 received token Hello World! from process 7.
```

Listing 5.4: A sample run result of Listing 5.3

Example 5.3.2. ***The Sieve of Eratosthens – finds primes between 2 and*** n

1. *Create a list of integers* $2, 3, 4, \ldots, n$*, none of which is marked.*
2. *Set* k *to 2, the first unmarked number on the list.*
3. *Repeat until* $k^2 > n$
   (a) *Mark all multiples of k between* k^2 *and* n
   (b) *Find the smallest number greater than* k *that is unmarked. Set* k *to this new value.*
4. *The unmarked numbers are primes*

In Table 5.1, integers in the range of $[2, 60]$ are stored in an array; Table 5.2 shows the result of applying sieve $k = 2$, that is all the multiples of 2 are marked as non-prime numbers; similarly, Tables 5.3, 5.4, and 5.5 show the results after applying $k = 3$, $k = 5$, and $k = 7$, respectively.

Table 5.1: Integers from 2 to 60

	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	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60

Table 5.2: Sieve = 2

	2	3		5		7		9		11		13		15		17		19	
21		23		25		27		29		31		33		35		37		39	
41		43		45		47		49		51		53		55		57		59	

The Sieve of Eratosthens – Parallelization

- The key parallel computation is step 3.(a)
- Communications among the sub problems; Two communications are needed to perform step 3.(b): a reduction and a broadcast.
- To simplify the problem, let's eliminate the reduction by assuming $n/p > \sqrt{n}$, then the first process is responsible for finding the next value of k.
- Decomposing the problem: data decomposition –
  - Block decomposition: the range of array elements for each process i is $\lfloor in/p \rfloor \sim \lfloor (i+1)n/p \rfloor - 1$.
  - Block-cyclic decomposition

5.4 Collective Communication and Computation Operations

- MPI provides an extensive set of functions for performing common collective communication operations.
- Each of these operations is defined over a group corresponding to the communicator.
- All processors in a communicator must call these operations.
- The barrier synchronization operation is performed in MPI using:

  ```
  int MPI_Barrier(MPI_Comm comm)
  ```

  The one-to-all broadcast operation is:

  ```
  int MPI_Bcast(void *buf, int count,MPI_Datatype datatype, int source, MPI_Comm comm)
  ```

- The all-to-one reduction operation is:

  ```
  int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, \
      MPI_Op op, int target, MPI_Comm comm)
  ```

Predefined Reduction Operations

Table 5.6 lists predefined reduction operators commonly used in `MPI_Reduce`.

Table 5.3: Sieve = 3

	2	3		5		7				11		13				17		19	
		23		25				29		31				35		37			
41		43				47		49				53		55				59	

Table 5.4: Sieve = 5

	2	3		5		7				11		13				17		19	
		23						29		31						37			
41		43				47		49				53						59	

Table 5.5: Sieve = 7

	2	3		5		7				11		13				17		19	
		23						29		31						37			
41		43				47						53						59	

A Simple Reduction Program

See Listing 5.5 for a simple example of usgae of `MPI_Reduce`.

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main(int argc, char **argv){
  int n, numprocs, myrank, myval, sum;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  if(myrank == 0){
    printf("Enter the number to be broadcasted: (0 quits.)");
    scanf("%d", &n);
  }
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  //printf("Rank %d received %d\n", myrank + 1, n);
  myval = n * 2;
  MPI_Reduce(&myval, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
      ;
  if(myrank==0)
    printf("Sum = %d \n", sum);
  MPI_Finalize();
  return 0;
```

Table 5.6: Predefined reduction operators

Operation	Meaning	Datatypes
`MPI_MAX`	Maximum	C integers and floating point
`MPI_MIN`	Minimum	C integers and floating point
`MPI_SUM`	Sum	C integers and floating point
`MPI_PROD`	Product	C integers and floating point
`MPI_LAND`	Logical AND	C integers
`MPI_BAND`	Bit-wise AND	C integers and byte
`MPI_LOR`	Logical OR	C integers
`MPI_BOR`	Bit-wise OR	C integers and byte
`MPI_LXOR`	Logical XOR	C integers
`MPI_BXOR`	Bit-wise XOR	C integers and byte
`MPI_MAXLOC`	max-max value-location	Data-pairs
`MPI_MINLOC`	min-min value-location	Data-pairs

```
}
```

Listing 5.5: Example of a simple reduction

Function Prototypes

The list below shows the prototypes of several MPI functions we discussed so far.

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Reduce(void *sendbuf,void *recvbuf,int count,MPI_Datatype
    datatype, \
  MPI_Op op,int root,MPI_Comm comm)
int MPI_Bcast(void *buf,int count,MPI_Datatype datatype,int root,
    MPI_Comm comm)
double MPI_Wtime()
double MPI_Wtick()
```

5.5 Running MPI Program With MPICH

5.5.1 What is MPICH

- MPICH is a high-performance and widely portable open-source implementation of MPI
- It provides all features of MPI that have been defined so far (including MPI-1, MPI-2.0, MPI-2.1, MPI-2.2, and MPI-3.0)
- http://www.mpich.org
- Compiling MPI program using MPICH
  - For C programs: `mpicc test.c -o test`
  - For C++ programs: `mpicxx test.cpp -o test`
  - To link to a math library: `mpicc test.c -o test -lm`

Running MPI Programs with MPICH

- Launch 16 processes on the local node: `mpiexec -n 16 ./test`
- Launch 16 processes on 4 nodes (each has 4 cores) `mpiexec -hosts h1:4,h2:4,h3:4,h4:4 -n 16 ./test` Runs the first four processes on h1, the next four on h2, etc. (`h1, h2, h3, h4` are host names separated by comma.)
  `mpiexec -hosts h1,h2,h3,h4 -n 16 ./test` Runs the first process on h1, the second on h2, etc., and wraps around. So, h1 will have the 1st, 5th, 9th and 13th processes.

5.5.2 Summary

- Basic set of MPI functions to write simple MPI programs
  - Initialize and finalize MPI
  - Inquire the basic MPI execution environment
  - Basic point to point communication: send and receive
  - Basic collective communication: broadcast and reduction

- Developing MPI programs through examples
- Compiling and running MPI programs

5.6 Point to Point Communication

5.6.1 Blocking vs. Non-blocking

- Most of the MPI point-to-point functions can be used in either blocking or non-blocking mode.
- Blocking:
  - A blocking send function will only "return" after it is safe to modify the application buffer (your send data) for reuse.
  - A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send.
  - A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
  - A blocking receive only "returns" after the data has arrived and is ready for use by the program.
- Non-blocking:
  - Non-blocking send and receive functions behave similarly - they will return almost immediately. They do not wait for any communication events to complete.
  - Non-blocking operations simply "request" the MPI library to perform the operation when it is able.
  - It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library.
  - Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.

5.6.2 MPI Message Passing Function Arguments

MPI point-to-point communication functions generally have an argument list that takes one of the formats shown in Table 5.7.

Table 5.7: MPI point-to-point communication functions

Blocking sends	`MPI_Send(buffer,count,type,dest,tag,comm)`
Non-blocking sends	`MPI_Isend(buffer,count,type,dest,tag,comm,request)`
Blocking receive	`MPI_Recv(buffer,count,type,source,tag,comm,status)`
Non-blocking receive	`MPI_Irecv(buffer,count,type,source,tag,comm,request)`

5.6.3 Avoiding Deadlocks

- The semantics of `MPI_Send` and `MPI_Recv` place some restrictions on how we can mix and match send and receive operations.
- Sources of deadlocks:
  - Send a large message from one process to another process
    - ∗ If the receive buffer is not large enough at the destination, the send must wait for the user to provide the memory space (through a receive)

  - Mismatched send and receive – unsafe.

- What happens with the parallel processing in Table 5.8?
- Order the communications as shown in Table 5.9.

Table 5.8: Point-to-point communication example

Process 0	Process 1
Send(1)	Send(0)
Recv(0)	Recv(1)

Table 5.9: Point-to-point communication example

Process 0	Process 1
Send(1)	Recv(1)
Recv(0)	Send(0)

Example 5.6.1. *Process 0 sends two messages with different tags to process 1, and process 1 receives them in reverse order (see Listing 5.6).*

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
  MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
} else if (myrank == 1) {
  MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
...
```

Listing 5.6: Point-to-point communication example

Example 5.6.2. *Consider the following piece of code, in which process i sends a message to process $i+1$ (modulo the number of processes) and receives a message from process $i-1$ (modulo the number of processes) (Listing 5.7). (Note that this example is different from the token ring example in Lec7.)*

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
...
```

Listing 5.7: Point-to-point communication in a ring of processes

Example 5.6.3. *Example 5.6.2 cont. We can break the circular wait to avoid deadlocks as shown in Listing 5.8.*

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 0) {
  MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
  MPI_COMM_WORLD);
  MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else {
  MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
  MPI_COMM_WORLD);
}
...
```

Listing 5.8: Avoiding deadlock in point-to-point communication in a ring of processes

5.6.4 Sending and Receiving Messages Simultaneously

To exchange messages, MPI provides the function in Listing 5.9 that sends and receives a message simultaneously. The arguments in `MPI_Sendrecv` function include those in MPI send and receive functions, respectively.

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
MPI_Datatype senddatatype, int dest, int sendtag,
void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
int source, int recvtag, MPI_Comm comm,
MPI_Status *status)
```

Listing 5.9: MPI_Sendrecv

Using MPI_Sendrecv in Example 5.6.2

Example 5.6.4. *Example 5.6.2 can be made "safe" by using* `MPI_Sendrecv`*:*

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Sendrecv(a, 10, MPI_INT, (myrank+1)%npes, 1, b, 10,
MPI_INT, (myrank-1+npes)%npes, 1,
MPI_COMM_WORLD, &status);
...
```

5.6.5 Overlapping Communication with Computation

- In order to overlap communication with computation, MPI provides the following pair of functions for performing non-blocking send and receive operations.

  ```
  int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,
  int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm
  ```

- These operations return before the communications have been completed. Function `MPI_Test` tests whether or not the non-blocking send or receive operation identified by its `request` has finished.

  ```
  int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
  ```

- `MPI_Wait` blocks until a specified non-blocking send or receive operation has completed. For multiple non-blocking operations, the programmer can specify `all` completions.

  ```
  int MPI_Wait(MPI_Request *request, MPI_Status *status)
  ```

- `MPI_Request` handle is used to determine whether an operations has completed.
  - Non-blocking wait: `MPI_Test`
  - Blocking wait: `MPI_Wait`
- Anywhere you use `MPI_Send` or `MPI_Recv`, you can use the pair of `MPI_Isend/MPI_Wait` or `MPI_Irecv/MPI_Wait`.
- It is sometimes desirable to wait on multiple requests:
  - `MPI_Waitall(int count,`
    `MPI_Request array_of_requests[],`
    `MPI_Status array_of_statuses[])`
  - `MPI_Waitany(count, array_of_requests, &index, &status)`
  - `MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)`
  - There are corresponding versions of TEST for each of these. The corresponding version of `MPI_Test`:

    ```
    int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag, MPI_Status array_
    ```

    `flag`: true if all the requests are completed, otherwise false.

Example 5.6.5.

```
int main(int argc, char *argv[]){
  int myid, numprocs, left, right, flag=0;
  int buffer1[10], buffer2[10];
  MPI_Request request; MPI_Status status;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  /* initialize buffer2 */
  ......
  right = (myid + 1) % numprocs;
  left = myid - 1;
  if (left < 0)
    left = numprocs - 1;
  MPI_Irecv(buffer1, 10, MPI_INT, left, 123, MPI_COMM_WORLD, &request)
      ;
  MPI_Send(buffer2, 10, MPI_INT, right, 123, MPI_COMM_WORLD);
  MPI_Test(&request, &flag, &status);
  while (!flag){
    /* Do some work ... */
```

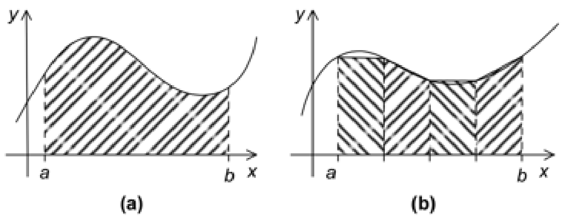


Figure 5.1: The trapezoidal rule: (a) area to be estimated, (b) estimate area using trapezoids

```
        MPI_Test(&request, &flag, &status);
    }
    MPI_Finalize();
}
```

Example 5.6.6.

```
int main(int argc, char *argv[]){
    int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
    MPI_Request reqs[4]; MPI_Status stats[4];
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    prev = rank-1; next = rank+1;
    if (rank == 0)  prev = numtasks - 1;
    if (rank == (numtasks - 1))  next = 0;
    MPI_Irecv(&buf[0],1,MPI_INT,prev,tag1,MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(&buf[1],1,MPI_INT,next,tag2,MPI_COMM_WORLD, &reqs[1]);

    MPI_Isend(&rank,1,MPI_INT,prev,tag2,MPI_COMM_WORLD, &reqs[2]);
    MPI_Isend(&rank,1,MPI_INT,next,tag1,MPI_COMM_WORLD, &reqs[3]);
    MPI_Waitall(4, reqs, stats);
    MPI_Finalize();
}
```

Example 5.6.7. *The Trapezoidal Rule We can use* **the trapezoidal rule** *to approximate the area between the graph of a function, $y = f(x)$, two vertical lines, and the x-axis.*

- If the endpoints of the subinterval are x_i and x_{i+1}, then the length of the subinterval is $h = x_{i+1} - x_i$. Also, if the lengths of the two vertical segments are $f(x_i)$ and $f(x_{i+1})$, then the area of the trapezoid is
$$\text{Area of one trapezoid} = \frac{h}{2}(f(x_i) + f(x_{i+1})).$$
- Since we chose the `N` subintervals, we also know that the bounds of the region are $x = a$ and $x = b$ then
$$h = \frac{b-a}{N}$$
- The pseudo code for a serial program:

```
h = (b-a)/N;
approx = (f(a) + f(b))/2.0;
```

```
for(i=1; i<=n-1; i++){
x_i = a + i * h;
approx += f(x_i);
}
approx = h * approx;
```

Recall we can design a parallel program using four basic steps:

1. Partition the problem solution into tasks.
2. Identify the communication between the tasks.
3. Aggregate the tasks into composite tasks.
4. Map the composite tasks to cores.

Example 5.6.7: Parallel Algorithm for the Trapezoidal Rule Assuming `comm_sz` evenly divides n, the pseudo-code for the parallel program is shown in Listing 5.10.

```
Get a, b, n;
h = (b - a)/n;
local_n = n/comm_sz;
local_a = a + my_rank * local_n * h;
local_b = local_a + local_n * h;
local_integral = Trap(local_a, local_b, local_n, h);
if (my_rank != 0)
  Send local integral to process 0;
else {/* my_rank == 0 */
  total_integral = local_integral;
  for (proc = 1; proc < comm_sz; proc++) {
    Receive local_integral from proc;
    total_integral += local_integral;
  }
}
if (my_rank == 0)
print result;
```

Listing 5.10: Parallel trapezoidal rule

Dealing with I/O

- In most cases, all the processes in `MPI_COMM_WORLD` have access to `stdout` and `stderr`.
- The order in which the processes' output appears is indeterministic.
- For the input, i.e., `stdin`, usually, only process 0 has access to.
- If an MPI program uses `scanf` function, then process 0 reads in the data, and sends it to the other processes.

5.7 Collective Communications

- Communication is coordinated among a group of processes, as specified by the communicator.
- All collective operations are blocking and no message tags are used.
- All processes in the communicator must call the collective operation.
- Three classes of collective operations
  - Data movement
  - Collective computation
  - Synchronization

5.7.1 MPI Collective Communication Illustrations

MPI_Bcast

- A collective communication in which data belonging to a single process is sent to all of the processes in the communicator is called a broadcast — `MPI_Bcast` (see Figure 5.2).

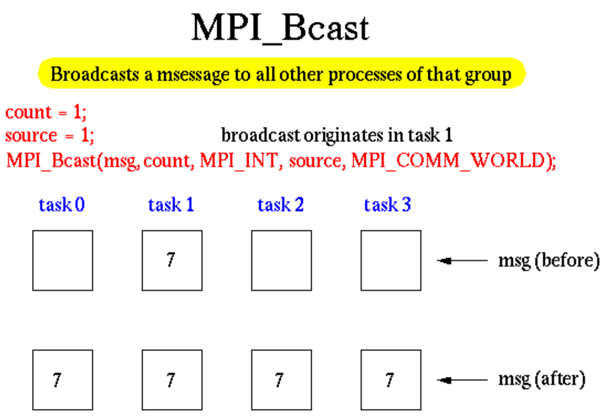

MPI_Bcast
Broadcasts a msessage to all other processes of that group
count = 1;
source = 1;
broadcast originates in task 1
MPI_Bcast(msg, count, MPI_INT, source, MPI_COMM_WORLD);
task 0
task 1
task 2
task 3
7
msg (before)
7
7
7
7
msg (after)

Figure 5.2: MPI_Bcast

- The process with rank `source` sends the contents of the memory referenced by `msg` to all the processes in the communicator `MPI_COMM_WORLD`.

Example 5.6.7 Continued

1. In our example `mpi_trapezoid_1.c`, we are using point-to-point communication function as shown in Listing 5.13.

```
if(my_rank == 0) {
  for(dest = 1; dest < comm_sz; dest++){
    MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
    MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
    MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
  }
} else {/* my rank != 0 */
  MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
  MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
  MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
}
```

Listing 5.11: Point-to-point communications in MPI trapezoid example

2. Instead of using point-to-point communications, you can use collective communications here. *Write another function to implement this part using* — `MPI_Bcast()`.

MPI_Reduce

- `MPI_Reduce` combines data from all processes in the communicator and returns it to one process (see an illustration in Figure 5.3).
- In many numerical algorithms, `Send/Receive` can be replaced by `Bcast/Reduce`, improving both simplicity and efficiency.

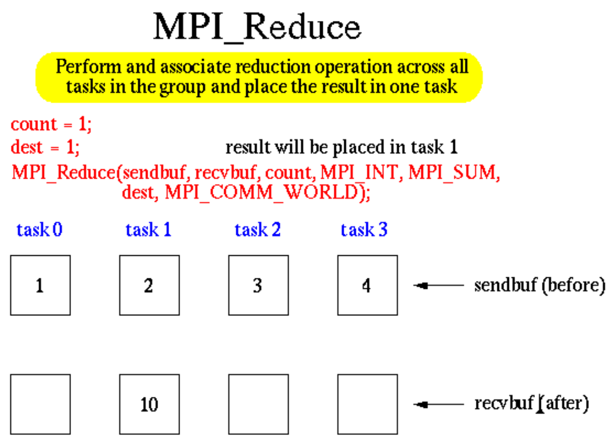

MPI_Reduce
Perform and associate reduction operation across all tasks in the group and place the result in one task
count = 1;
dest = 1;
result will be placed in task 1
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, dest, MPI_COMM_WORLD);
task 0
task 1
task 2
task 3
1
2
3
4
sendbuf (before)
10
recvbuf (after)

Figure 5.3: MPI_Reduce

- When the `count` is greater 1, `MPI_Reduce` operates on arrays instead of scalars.

```
double local_x[N], sum[N];
...
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0, \
MPI_COMM_WORLD);
```

Listing 5.12: MPI_Reduce for arrays

Example 5.6.7 continued

1. In our example `mpi_trapezoid_1.c`, we are using point-to-point communications as shown in Listing 5.13.

```
/* Add up the integrals calculated by each process */
if(my_rank != 0) {
  MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
} else {
  total_int = local_int;
  for(source = 1; source < comm_sz; source++) {
    MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD
        , MPI_STATUS_IGNORE);
    total_int += local_int;
  }
```

Listing 5.13: Point-to-point communication in Example 5.6.7

2. Instead of using point-to-point communications, you can also use collective communications here. *Rewrite this part using appropriate collective communication.*

MPI_Reduce

- Suppose that each process calls `MPI_Reduce` with operator `MPI_SUM`, and *destination process 0*. What happens with the multiple calls of `MPI_Reduce` in Table 5.10? What are the values for `b` and `d` after executing the second `MPI_Reduce`?

Table 5.10: Understanding MPI_Reduce

Time	Process 0	Process 1	Process 2
0	`a=1; c = 2;`	`a=1; c = 2;`	`a=1; c = 2;`
1	`MPI_Reduce(&a,&b,1,...)`	`MPI_Reduce(&c,&d,1,...)`	`MPI_Reduce(&a,&b,1,...)`
2	`MPI_Reduce(&c,&d,1,...)`	`MPI_Reduce(&a,&b,1,...)`	`MPI_Reduce(&c,&d,1,...)`

- The order of the calls will determine the matching.
- What will happen with the following code?

```
MPI_Reduce(&x, &x, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
```

MPI_Allreduce

- If the result of the reduction operation is needed by all processes, MPI provides:

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

- This is equivalent to an `MPI_Reduce` followed by an `MPI_Bcast`, see the illustration in Figure 5.4.
- Figure 5.5 shows two different ways to implement `MPI_Allreduce`.

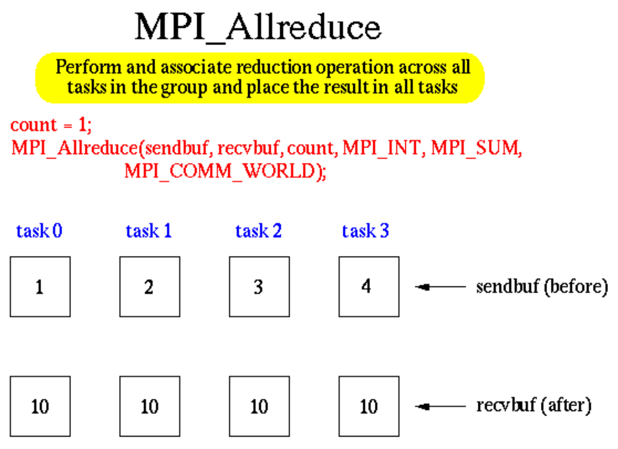
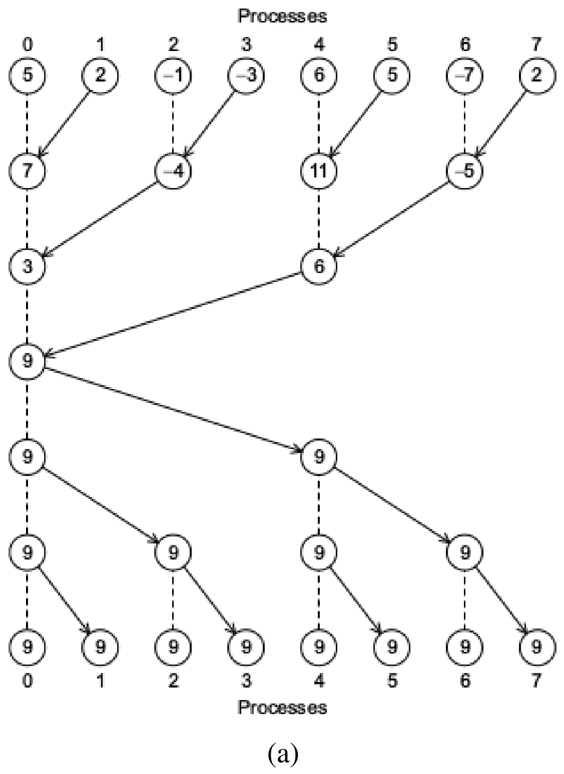

MPI_Allreduce
Perform and associate reduction operation across all tasks in the group and place the result in all tasks
count = 1;
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
task 0
task 1
task 2
task 3
1
2
3
4
sendbuf (before)
10
10
10
10
recvbuf (after)

Figure 5.4: MPI_Allreduce

MPI_Scatter

- The scatter operation is to distribute *distinct messages* from a single source task (or process) to each task in the group.



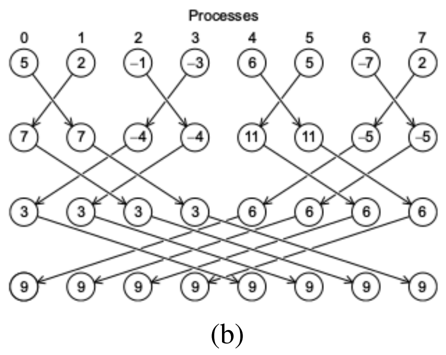


Figure 5.5: (a) A global sum followed by a broadcasting; (2) A butterfly structured global sum.

- Figure 5.6 illustrates a simple scatter operation.

```
int MPI_Scatter(void *sendbuf, int sendcount,
    MPI_Datatype senddatatype, void *recvbuf,
    int recvcount, MPI_Datatype recvdatatype,
    int source, MPI_Comm comm)
```

MPI_Gather

- The gather operation is performed in MPI using `MPI_Gather`.
  - Gathers *distinct messages* from each task in the group to a single destination task.
  - Reverse operation of `MPI_Scatter`.
  - Figure 5.7 illustrates a simple gather operation among 4 MPI processes.

```
int MPI_Gather(void *sendbuf, int sendcount,
MPI_Datatype senddatatype, void *recvbuf,
int recvcount, MPI_Datatype recvdatatype,
int target, MPI_Comm comm)
```

MPI_Allgather

- MPI also provides the `MPI_Allgather` function in which the data are gathered at all the processes.
- See Figure 5.8 for an illustration.

```
int MPI_Allgather(void *sendbuf, int sendcount,
MPI_Datatype senddatatype, void *recvbuf,
```

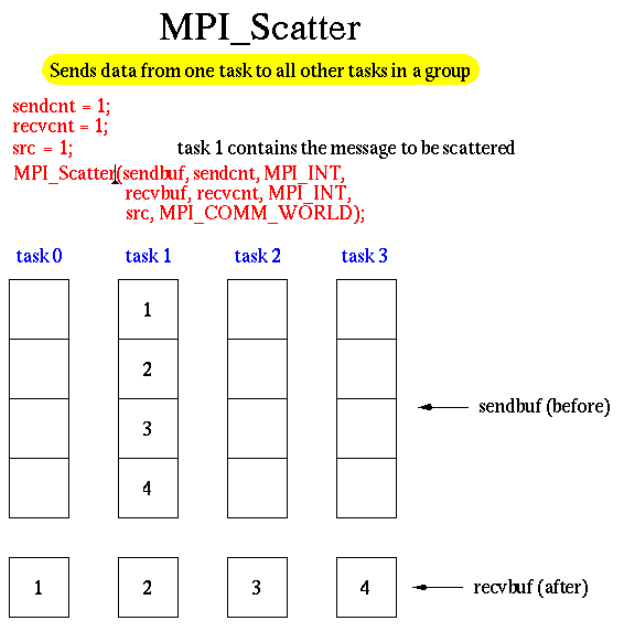

MPI_Scatter
Sends data from one task to all other tasks in a group
sendcnt = 1;
recvcnt = 1;
src = 1;
task 1 contains the message to be scattered
MPI_Scatter(sendbuf, sendcnt, MPI_INT,
recvbuf, recvcnt, MPI_INT,
src, MPI_COMM_WORLD);
task 0
task 1
task 2
task 3
1
2
3
4
sendbuf (before)
1
2
3
4
recvbuf (after)

Figure 5.6: MPI_Scatter

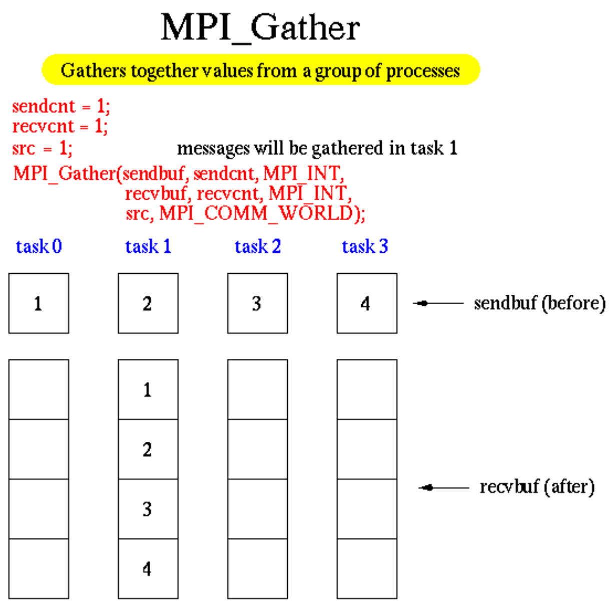

MPI_Gather
Gathers together values from a group of processes
sendcnt = 1;
recvcnt = 1;
src = 1;
messages will be gathered in task 1
MPI_Gather(sendbuf, sendcnt, MPI_INT,
recvbuf, recvcnt, MPI_INT,
src, MPI_COMM_WORLD);
task 0
task 1
task 2
task 3
1
2
3
4
sendbuf (before)
1
2
3
4
recvbuf (after)

Figure 5.7: MPI_Gather

```
int recvcount, MPI_Datatype recvdatatype,
MPI_Comm comm)
```

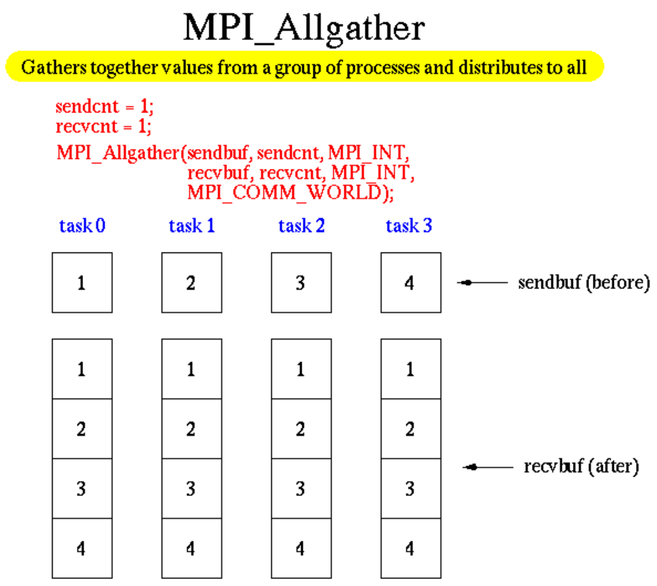

MPI_Allgather
Gathers together values from a group of processes and distributes to all
sendcnt = 1;
recvcnt = 1;
MPI_Allgather(sendbuf, sendcnt, MPI_INT,
recvbuf, recvcnt, MPI_INT,
MPI_COMM_WORLD);
task 0
task 1
task 2
task 3
1
2
3
4
sendbuf (before)
recvbuf (after)

Figure 5.8: MPI_Allgather

MPI_Alltoall

- The all-to-all communication operation is performed by:

  ```
  int MPI_Alltoall(void *sendbuf, int sendcount,
      MPI_Datatype senddatatype, void *recvbuf,
      int recvcount, MPI_Datatype recvdatatype,
      MPI_Comm comm)
  ```

- Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index. See the illustration in Figure 5.9.

Example 5.7.1 (Matrix vector multiplication). *If $A = (a_{ij})$ is an $m \times n$ matrix and $\mathbf{x}$ is a vector with n components, then $\mathbf{y} = A\mathbf{x}$ is a vector with m components. Furthermore,*

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{i,n-1}x_{n-1}.$$

A serial code is given in Listing **??**

```
for (i = 0; i < m; i++) {
  y[i] = 0.0;
  for (j = 0; j < n; j++)
    y[i] += A[i*n+j]*x[j];
}
```

Listing 5.14: Serial matrix-vector multiplicationm label

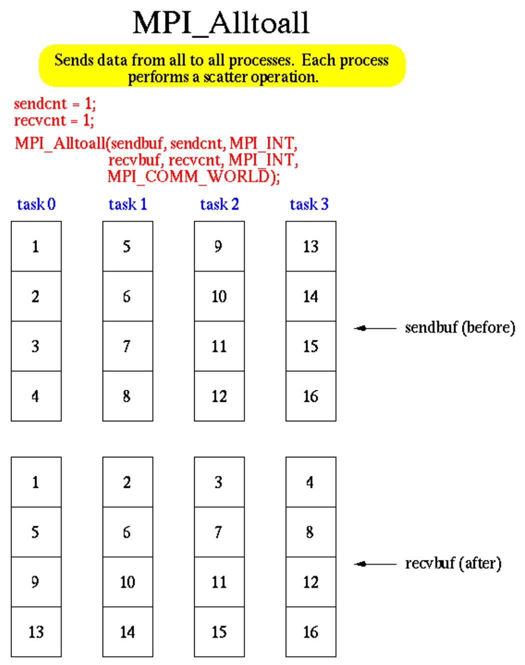

MPI_Alltoall
Sends data from all to all processes. Each process performs a scatter operation.
sendcnt = 1;
recvcnt = 1;
MPI_Alltoall(sendbuf, sendcnt, MPI_INT, recvbuf, recvcnt, MPI_INT, MPI_COMM_WORLD);
task 0
task 1
task 2
task 3
sendbuf (before)
recvbuf (after)

Figure 5.9: MPI_Alltoall

Process 0 reads in the matrix and distributes row blocks to all the processes in communicator comm. Listing 5.15 shows the code snippet for this.

```
if (my_rank == 0) {
  A = malloc(m*n*sizeof(double));
  if (A == NULL) local_ok = 0;
  Check_for_error(local_ok, "Random_matrix", "Can't allocate
      temporary matrix", comm);
  srand(2021);
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      A[i*n+j] = (double)rand( ) / RAND_MAX;
  MPI_Scatter(A, local_m*n, MPI_DOUBLE, local_A, local_m*n,
      MPI_DOUBLE, 0, comm);
  free(A);
} else {
  Check_for_error(local_ok, "Random_matrix", "Can't allocate
      temporary matrix", comm);
  MPI_Scatter(A, local_m*n, MPI_DOUBLE, local_A, local_m*n,
      MPI_DOUBLE, 0, comm);
}
```

Listing 5.15: Distribution of matrix row-blocks using `MPI_Scatter`

Each process gathers the entire vector, then proceeds to compute its share of sub-matrix and vector multiplication (see Listing 5.16).

```
MPI_Allgather(local_x, local_n, MPI_DOUBLE, x, local_n, MPI_DOUBLE,
    comm);
for (local_i = 0; local_i < local_m; local_i++) {
  local_y[local_i] = 0.0;
  for (j = 0; j < n; j++)
    local_y[local_i] += local_A[local_i*n+j]*x[j];
}
```

Listing 5.16: Code snippet for MPI parallel matrix-vector multiplication

MPI_Scan

- To compute prefix-sums, MPI provides:

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Figure 5.10 illustrates a simple scan operation among a group of MPI processes.

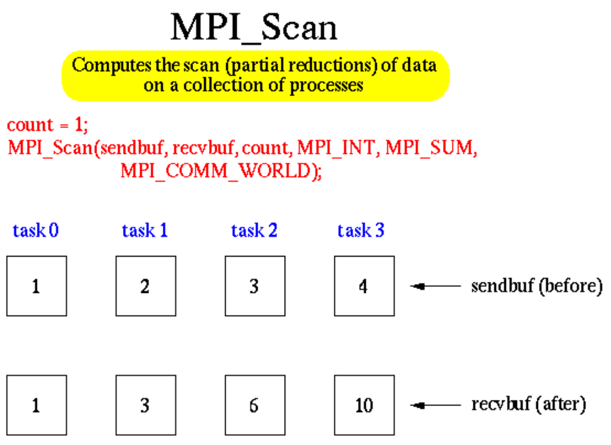

MPI_Scan
Computes the scan (partial reductions) of data on a collection of processes
count = 1;
MPI_Scan(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
task 0
task 1
task 2
task 3
1
2
3
4
sendbuf (before)
1
3
6
10
recvbuf (after)

Figure 5.10: MPI_Scan

- Using this core set of collective operations, MPI communications can be greatly simplified.

MPI_Scatterv Scatters a buffer in parts to all processes in a communicator, which allows different amounts of data to be sent to different processes.

```
int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int source, MPI_Comm comm)
```

- `sendbuf`: address of send buffer (significant only at root)
- `sendcounts`: integer array (of length group size) specifying the number of elements to send to each processor
- `displs`: integer array (of length group size). Entry i specifies the displacement (relative to `sendbuf` from which to take the outgoing data to process i
- `sendtype`: data type of send buffer elements
- `recvcount`: number of elements in receive buffer (integer)
- `recvtype`: data type of receive buffer elements

- `root`: rank of sending process (integer)

Example 5.7.2. *Given an $N \times N$ matrix, A, of integers, write an MPI program that distributes the first M rows of the upper triangle of A to M processes by rows, where each process gets one row of the upper triangle of A (when $M = N$, it means each process gets one row of the upper triangle of A).*

For this example, we can use `MPI_Scatterv` (see Figure 5.11).

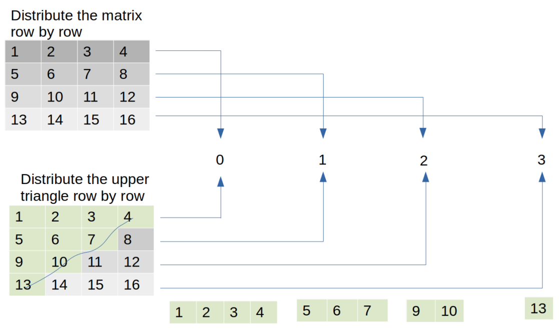


Figure 5.11: `MPI_Scatter` (top) and `MPI_Scatterv` (bottom) example

For Example 5.7.2, assuming the matrix is only 4×4, and we are running the MPI code using 4 processes, then some of the arguments of calling `MPI_Scatterv`:

- `sendcounts[4] = {4, 3, 2, 1};`
- `displs[4] = {0, 4, 8, 12}` which is with reference to `sendbuf`; these values can be expressed as `N * rank`, where `rank` is the rank of a process.
- note also that `recvcount` in `MPI_Scatterv` is a scalar; for process 0, `recvcount = 4 (=4-0)`; for process 1, `recvcount = 3 (=4-1)`; for process 2, `recvcount = 2 (=4-2)`; and for process 3, `recvcount = 1 (=4-3)`; so this value can be obtained as `N - rank` where `N` is the number of rows in the matrix, and `rank` is the rank of a process.

`scatterv_1.c` gives an example code for Example 5.7.2.

MPI_Alltoall Sends data from all to all processes; each process may send a different amount of data and provide displacements for the input and output data.

```
MPI_Alltoallv(void *sendbuf, int *sendcounts, int *sdispls,
MPI_Datatype sendtype, void *recvbuf, int *recvcounts,
int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
```

- `sendbuf`: starting address of send buffer
- `sendcounts`: integer array equal to the group size specifying the number of elements to send to each processor
- `sdispls`: integer array (of length group size). Entry j specifies the displacement (relative to `sendbuf` from which to take the outgoing data destined for process j
- `sendtype`: data type of send buffer elements
- `recvcounts`: integer array equal to the group size specifying the maximum number of elements that can be received from each processor
- `rdispls`: integer array (of length group size). Entry i specifies the displacement (relative to `recvbuf` at which to place the incoming data from process i
- `recvtype`: data type of receive buffer elements

Example 5.7.3. *Given the `MPI_Alltoallv` argument settings shown in Figure 5.12 (the number of processes is 3), what is the content of `recvbuf` for each process (see the results in Figure 5.13)?*

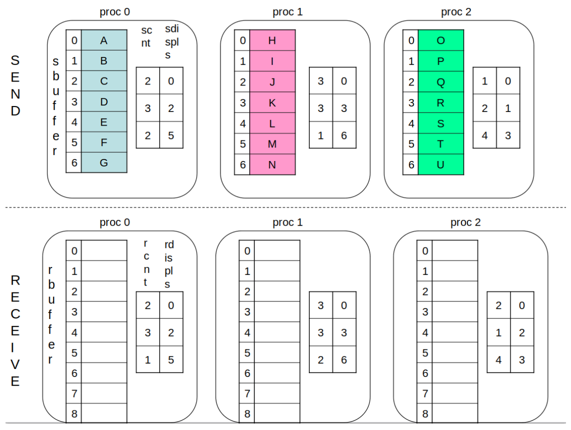


Figure 5.12: `MPI_Alltoallv` example

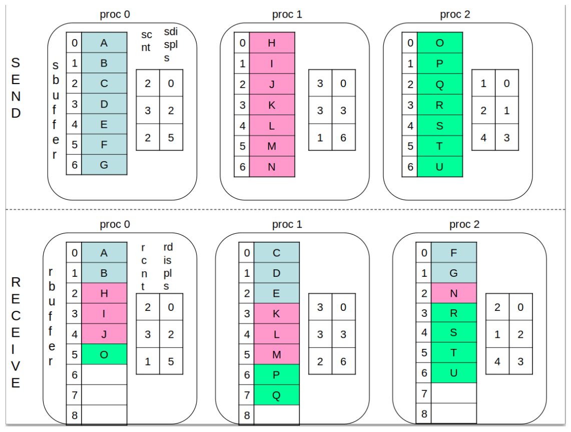


Figure 5.13: `MPI_Alltoallv` example

The following function allows a different number of data elements to be sent by each process by replacing `recvcount` in `MPI_Gather` with an array `recvcounts`.

```
int MPI_Gatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int *recvcounts, int *displs,
    MPI_Datatype recvtype, int target, MPI_Comm comm)
```

- `sendbuf`: pointer, starting address of send buffer (or the data to be sent)
- `sendcount`: the number of elements in the send buffer
- `sendtype`: datatype of send buffer elements
- `recvbuf`: pointer, starting address of receive buffer (significant only at root)
- `recvcounts`: integer array (of length group size) containing the number of elements to be received from each process (significant only at root)
- `displs`: integer array (of length group size). Entry i specifies the displacement relative to `recvbuf` at which to place the incoming data from process i (significant only at root)
- `recvtype`: the datatype of data to be received (significant only at root)
- `target`: rank of receiving process (integer)

Gather data from all processes and deliver the combined data to all processes

```
int MPI_Allgatherv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int *recvcounts, int *displs,
    MPI_Datatype recvtype, MPI_Comm comm)
```

- `sendbuf`: pointer, starting address of send buffer (or the data to be sent)
- `sendcount`: the number of elements in the send buffer
- `sendtype`: datatype of send buffer elements
- `recvbuf`: pointer, starting address of receive buffer (significant only at root)
- `recvcounts`: integer array (of length group size) containing the number of elements to be received from each process (significant only at root)
- `displs`: integer array (of length group size). Entry i specifies the displacement relative to `recvbuf` at which to place the incoming data from process i (significant only at root)
- `recvtype`: the datatype of data to be received (significant only at root)

5.8 Examples

5.8.1 Parallel Quicksort Algorithm

- one process broadcast initial pivot to all processes;
- each process in the upper half swaps with a partner in the lower half
- recurse on each half
- swap among partners in each half
- each process uses quicksort on local elements

Figure 5.14 illustrates the parallel quicksort using four MPI processes.

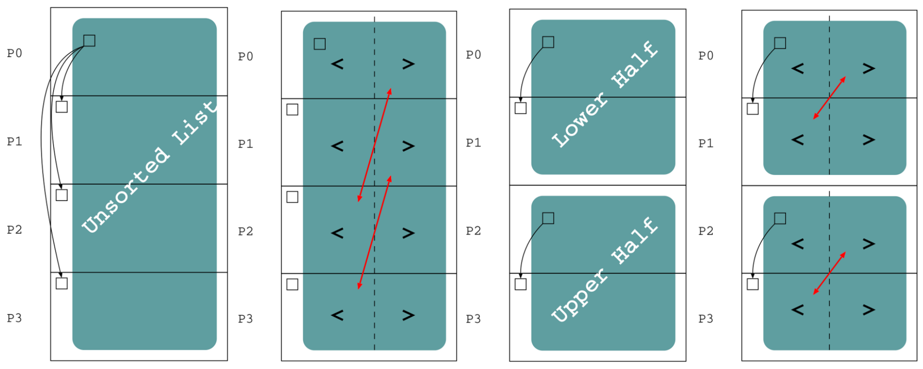


Figure 5.14: Parallel quicksort

5.8.2 Hyperquicksort

Limitation of parallel quicksort: poor balancing of list sizes.
Hyperquicksort: sort elements before broadcasting pivot.

- sort elements in each process
- select median as pivot element and broadcast it
- each process in the upper half swaps with a partner in the lower half
- recurse on each half

Example 5.8.1. *Task 0 pings task 1 and awaits return ping*

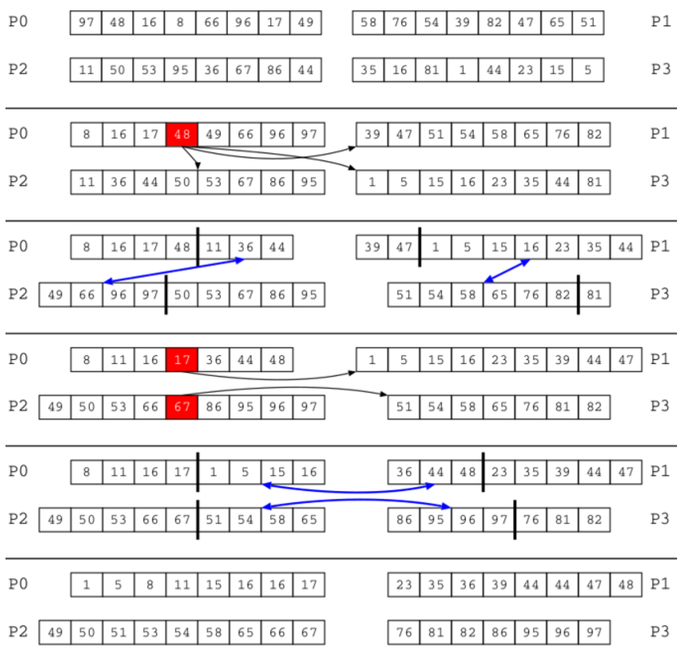


Figure 5.15: Illustration of Hyperquicksort algorithm

```
#include "mpi.h"
#include <stdio.h>
main(int argc, char *argv[]){
  int numtasks, rank, dest, source, rc, count, tag=1;
  char inmsg, outmsg='x';
  MPI_Status Stat;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0){
    dest = 1;
    source = 1;
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,
        &Stat);
  else if (rank == 1) {
    dest = 0;
    source = 0;
    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,
        &Stat);
    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }
  rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
  printf("Task %d: Received %d char(s) from task %d with tag %d \n",
  rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
  MPI_Finalize();
  }
```

Listing 5.17: MPI code for ping-pong point-to-point communication between two MPI processes

Example 5.8.2. *Perform a scatter operation on the rows of an array See Listing 5.18*

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

main(int argc, char *argv[])  {
  int numtasks, rank, sendcount, recvcount, source;
  float sendbuf[SIZE][SIZE] = {
    {1.0, 2.0, 3.0, 4.0},
    {5.0, 6.0, 7.0, 8.0},
    {9.0, 10.0, 11.0, 12.0},
    {13.0, 14.0, 15.0, 16.0}  };
  float recvbuf[SIZE];

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  if (numtasks == SIZE) {
    source = 1;
    sendcount = SIZE;
    recvcount = SIZE;
    MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
        MPI_FLOAT,source,MPI_COMM_WORLD);
    printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
        recvbuf[1],recvbuf[2],recvbuf[3]);
  }
  else
    printf("Must specify %d processors. Terminating.\n",SIZE);
  MPI_Finalize();
```

Listing 5.18: Example of MPI_Scatter

5.8.3 The Odd-Even Transposition Sort

- Sorts n elements in n phases (n is even), each of which requires $n/2$ compare-exchange operations.
- The algorithm alternates between two phases — odd and even phases.
- Let $< a_0, a_1, ..., a_{n-1} >$ be the sequence to be sorted.
  - During the odd phase, elements with odd indices are compared with their right neighbours, and if they are out of sequence they are exchanged; thus, the pairs $(a_1, a_2), (a_3, a_4), \ldots, (a_{n-3}, a_{n-2})$ are compare exchanged.
  - During the even phase, elements with even indices are compared with their right neighbours, and if they are out of sequence they are exchanged; $(a_0, a_1), (a_2, a_3), \ldots, (a_{n-2}, a_{n-1})$.
- After n phases of odd-even exchanges, the sequence is sorted. Each phase requires $n/2$ compare-exchange operations (sequential complexity $O(n^2)$).

Odd-even transposition sort – The serial algorithm Listing 5.19 gives the serial algorithm for odd-even transposition sort.

```
for i = 0 to n-1 do
  if i is even then
    for j = 0 to n/2 - 1 do
      compare-exchange(a(2j), a(2j+1));
  if i is odd then
    for j = 0 to n/2 - 1 do
      compare-exchange(a(2j+1), a(2j+2));
```

Listing 5.19: Odd-even transposition sort

Odd-even transposition sort – The parallel algorithm Listing 5.20 gives the parallel version odd-even transposition sort.

```
void oddevensort(int n)
  id = process's label;
  for i =0 to n-1 do
    if i is odd then
      if id is odd then
        compare-exchange_min(id, id + 1);//increasing comparator
      else
        compare-exchange_max(id, id - 1);//decreasing comparator
    if i is even then
      if id is even then
        compare-exchange_min(id, id + 1);
      else
        compare-exchange_max(id, id - 1);
```

Listing 5.20: Parallel odd-even transposition sort

5.8.4 Parallel Bitonic Sort

The following is a brief introduction of bitonic sorting taken from [Grama et al. 2003, Chap. 9].

- A sequence of keys $(a_0, a_1, \dots, a_{n-1})$ is bitonic if
  1. there exists an index m, $0 \leq m \leq n-1$ such that
  $$a_0 \leq a_1 \leq \cdots \leq a_m \geq a_{m+1} \geq \dots a_{n-1},$$
  2. or there exists a cyclic shift σ of $(0, 1, \dots, n-1)$ such that the sequence $(a_{\sigma(0)}, a_{\sigma(1)}, \dots, a_{\sigma(n-1)})$ satisfies condition 1. A cyclic shift sends each index i to $(i+s)$ mod n, for some integers s.
- A bitonic sequence has two tones – increasing and decreasing, or vice versa. Any cyclic rotation of such networks is also considered bitonic.
- $(1, 2, 4, 7, 6, 0)$ is a bitonic sequence, because it first increases and then decreases. $(8, 9, 2, 1, 0, 4)$ is another bitonic sequence, because it is a cyclic shift of $(0, 4, 8, 9, 2, 1)$. Similarly, the sequence $(1, 5, 6, 9, 8, 7, 3, 0)$ is bitonic, as is the sequence $(6, 9, 8, 7, 3, 0, 1, 5)$, since it can be obtained from the first by a cyclic shift.
- If sequence $A = (a_0, a_1, \dots, a_{n-1})$ is bitonic, then we can form two bitonic sequences from A as
  $$A_{min} = (min(a_0, a_{n/2}), min(a_1, a_{n/2+1}), \dots, min(a_{n/2-1}, a_{n-1})),$$
  and
  $$A_{max} = (max(a_0, a_{n/2}), max(a_1, a_{n/2+1}), \dots, max(a_{n/2-1}, a_{n-1})).$$

A_{min} and A_{max} are bitonic sequences, and each element of A_{min} is less than every element in A_{max}.

- We can apply the procedure recursively on A_{min} and A_{max} to get the sorted sequence.
- For example, $A = (6, 9, 8, 7, 3, 0, 1, 5)$ is a bitonic sequence. We can split it into two bitonic sequence by finding $A_{min} = (min(6, 3), min(9, 0), min(8, 1), min(7, 5))$, which is $A_{min} = (3, 0, 1, 5)$ (first decrease, then increase), and $A_{max} = (max(6, 3), max(9, 0), max(8, 1), max(7, 5))$, which is $A_{max} = (6, 9, 8, 7)$ (first increase, then decrease).
- The kernel of the network is the rearrangement of a bitonic sequence into a sorted sequence.

Original sequence	3	5	8	9	10	12	14	20	95	90	60	40	35	23	18	0
1st Split	3	5	8	9	10	12	14	0	95	90	60	40	35	23	18	20
2nd Split	3	5	8	0	10	12	14	9	35	23	18	20	95	90	60	40
3rd Split	3	0	8	5	10	9	14	12	18	20	35	23	60	40	95	90
4th Split	0	3	5	8	9	10	12	14	18	20	23	35	40	60	90	95

Table 5.11: Merging a 16-element bitonic sequence through a series of $\log 16$ bitonic splits.

- We can easily build a sorting network to implement this bitonic merge algorithm.
- Such a network is called a *bitonic merging network*. See Table 5.11 for an example.
- The network contains $\log n$ columns (see Figure 5.18). Each column contains $n/2$ comparators and performs one step of the bitonic merge.
- We denote a bitonic merging network with n inputs by $\oplus$BM[n].
- Replacing the $\oplus$ comparators by $\ominus$ comparators results in a decreasing output sequence; such a network is denoted by $\ominus$BM[n]. **(Here, a comparator refers to a device with two inputs x and y and two outputs x' and y'. For an increasing comparator, denoted by $\oplus$, $x' = min(x, y)$ and $y' = max(x, y)$, and vice versa for decreasing comparator, denoted by $\ominus$.)**
- The depth of the network is $\Theta(\log^2 n)$. Each stage of the network contains $n/2$ comparators. A serial implementation of the network would have complexity $\Theta(n \log^2 n)$. On the other hand, a parallel bitonic sorting network sorts n elements in $\Theta(\log^2 n)$ time. (The comparators within each stage are independent of one another, i.e., can be done in parallel.)
- How do we sort an unsorted sequence using a bitonic merge?
  - We must first build a single bitonic sequence from the given sequence. See Figure 5.16 for an illustration of building a bitonic sequence from an input sequence.
    - A sequence of length 2 is a bitonic sequence.
    - A bitonic sequence of length 4 can be built by sorting the first two elements using $\oplus$BM[2] and next two, using $\ominus$BM[2].
    - This process can be repeated to generate larger bitonic sequences.
  - Once we have turned our input into a bitonic sequence, we can apply a bitonic merge process to obtain a sorted list. Figure 5.17 shows an example.

To implement bitonic sorting in MPI, the basic idea is that you have much more elements than the number of PEs. For example sorting one million or even more data elements using a small number of MPI processes in our case, say 4, 8, or 16 processes. So, you need to fit the bitonic sorting idea into this kind of framework, not that you are sorting 16 elements only using 16 PEs. In your implementation, the main element from the perspective of distributed programming model is to figure out how to pair up a PE

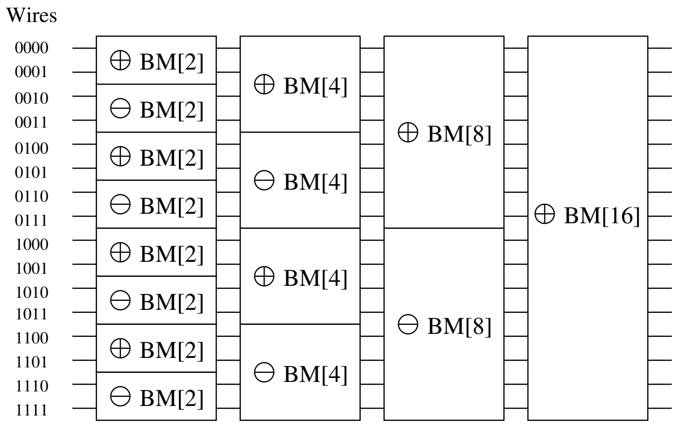


Figure 5.16: A schematic representation of a network that converts an input sequence into a bitonic sequence. In this example, $\oplus$BM[k] and $\ominus$BM[k] denote bitonic merging networks of input size k that use $\oplus$ and $\ominus$ comparators, respectively. The last merging network ($\oplus$BM[16]) sorts the input. In this example, $n = 16$.

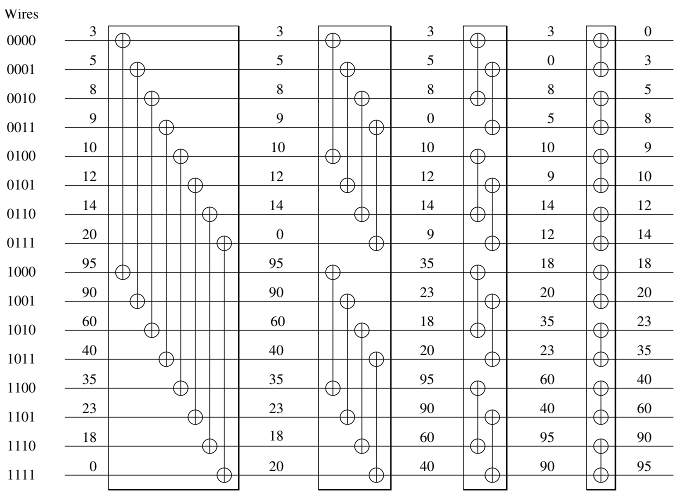


Figure 5.17: A bitonic merging network for $n = 16$. The input wires are numbered $0, 1 \ldots, n-1$, and the binary representation of these numbers is shown. Each column of comparators is drawn separately; the entire figure represents a $\oplus$BM[16] bitonic merging network. The network takes a bitonic sequence and outputs it in sorted order.

with its correct partner at each step. The overall idea is to begin by dividing the data elements among the PEs evenly first, then each PE sorts its share of elements in increasing or decreasing orders depending on its MPI process rank. In this way, the goal is to generate a global bitonic sequence owned by the entire MPI processes. For example, if you have 4 MPI processes, then you can have processes 0 and 1 jointly own a monotonic (say increasing) sequence; and processes 2 and 3 jointly own another monotonic (say decreasing) sequence. Then processes 0, 1, 2, and 3 jointly own a bitonic sequence. The remaining work will be then how to carry out the bitonic split steps. **Lastly, note that you need to assume both the number of elements to be sorted and the number of PEs to be powers of two.**

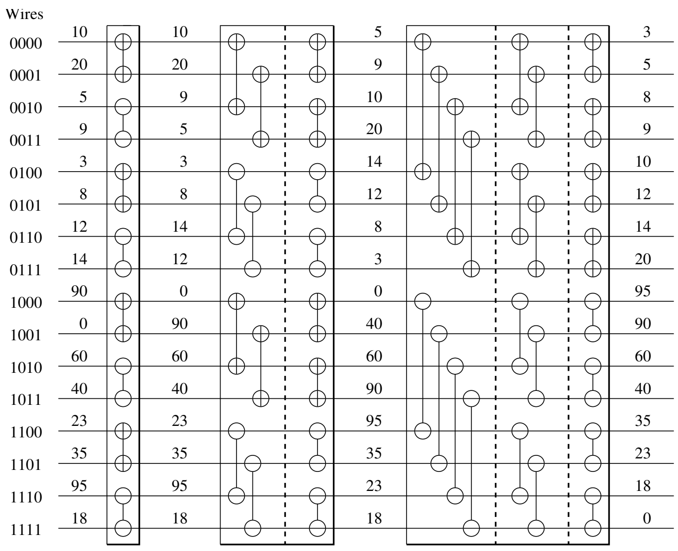


Figure 5.18: The comparator network that transforms an input sequence of 16 unordered numbers into a bitonic sequence.

5.9 MPI Derived Datatypes

Example 5.9.1 (Sending many short messages is not communication efficient)**.** *Listing 5.21 shows an example of sending many small pieces of messages.*

```
double x[1000];
....
for(i=0; i<1000; i++){
  if(my_rank == 0)
    MPI_Send(&x[i], 1, MPI_DOUBLE, 1, 0, comm);
  else
    MPI_Recv(&x[i], 1, MPI_DOUBLE, 0, 0,comm,&status);
}
/*the following is more efficient than using the for loop*/
if(my_rank == 0)
  MPI_Send(&x[0], 1000, MPI_DOUBLE, 1, 0, comm);
else
  MPI_Recv(&x[0], 1000, MPI_DOUBLE, 0, 0, comm, &status);
```

Listing 5.21: An example of communicating multiple short messages

In distributed-memory systems, communication can be much more expensive than local computation. Thus, if we can reduce the number of communications, we are likely to improve the performance of our programs.

MPI Built-in Datatypes

- The MPI standard defines many built in datatypes, mostly mirroring standard C/C++ or FORTRAN datatypes
- These are sufficient when sending single instances of each type
- They are also usually sufficient when sending contiguous blocks of a single type
- Sometimes, however, we want to send non-contiguous data or data that is comprised of multiple

types
- MPI provides a mechanism to create *derived datatypes* that are built from simple datatypes
- In MPI, a *derived datatype* can be used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory.
- Why use derived datatypes?
  - Primitive datatypes are contiguous;
  - Derived datatypes allow you to specify non-contiguous data in a convenient manner and treat it as though it is contiguous;
  - Useful to
    - Make code more readable
    - Reduce number of messages and increase their size (faster since less latency);
    - Make code more efficient if messages of the same size are repeatedly used.

5.9.1 Typemap

Formally, a derived datatype in MPI is described by a *typemap* consists of a sequence of basic MPI datatypes together with a *displacement* for each of the datatypes. That is,

- a sequence of basic datatypes: $\{type_0, ..., type_{n-1}\}$
- a sequence of integer displacements: $\{displ_0, ..., displ_{n-1}\}$.
- Typemap = $\{(type_0, disp_0), \cdots, (type_{n-1}, disp_{n-1})\}$

For example, a typemap might consist of (double,0),(char,8) indicating the type has two elements:

- a double precision floating point value starting at displacement 0,
- and a single character starting at displacement 8.

- Types also have *extent*, which indicates how much space is required for the type
- The extent of a type may be more than the sum of the bytes required for each component
- For example, on a machine that requires double-precision numbers to start on an 8-byte boundary, the type (double,0),(char,8) will have an extent of 16 even though it only requires 9 bytes

5.9.2 Creating and Using a New Datatype

Three steps are necessary to create and use a new datatype in MPI:

- Create the type using one of MPI's type construction functions
- Commit the type using `MPI_Type_commit()`.
- Release the datatype using `MPI_Type_free()` when it is not needed any more.

MPI provides several methods for constructing derived datatypes to handle a wide variety of situations.

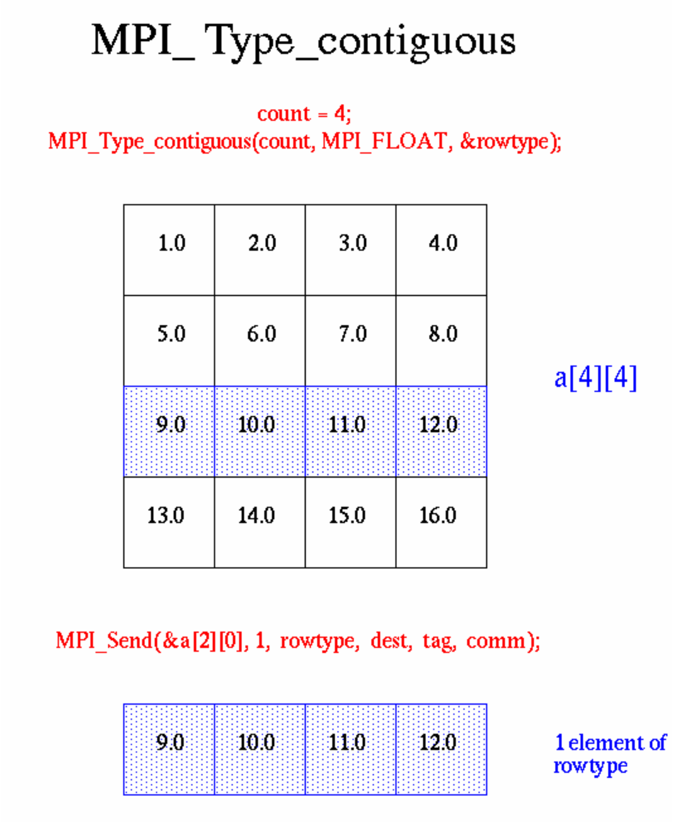
- Contiguous
- Vector
- Indexed
- Struct

5.9.3 Contiguous Type

Contiguous: The contiguous datatype allows for a single type to refer to contiguous multiple elements of an existing datatype.

```
int MPI_Type_contiguous(
           int count,        //count
```

```
                MPI_Datatype oldtype,  //old datatype
                MPI_Datatype *newtype) //new datatype
```



To define the new datatype in this example and release it after finished using it:

```
MPI_Datatype rowtype;
MPI_Type_contiguous(4, MPI_DOUBLE, &rowtype);
MPI_Type_commit(&rowtype);
......
MPI_Type_free(&rowtype);
```

To define a new datatype:

- Declare the new datatype as `MPI_Datatype`.
- Construct the new datatype.
- Before we can use a derived datatype in a communication function, we must first **commit** it with a call to

  ```
  int MPI_Type_commit(MPI_Datatype* datatype);
  ```

  Commits new datatype to the system. Required for all derived datatypes.
- When we finish using the new datatype, we can free any additional storage used with a call to

  ```
  int MPI_Type_free(MPI_Datatype* datatype)
  ```

The new datatype is essentially an array of `count` elements having type `oldtype`. For example, the following two code fragments are equivalent:

```
MPI_Send (a,n,MPI_DOUBLE,dest,tag,MPI_COMM_WORLD);
```

and

```
MPI_Datatype rowtype;
MPI_Type_contiguous(n, MPI_DOUBLE, &rowtype);
MPI_Type_commit(&rowtype);
MPI_Send(a, 1, rowtype, dest, tag, MPI_COMM_WORLD);
```

Example 5.9.2.

```
#define SIZE 4
float a[SIZE][SIZE] =
{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
  9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
float b[SIZE];
MPI_Status stat;
MPI_Datatype rowtype;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);
if(numtasks == SIZE){
  if(rank == 0)
    for (i=0; i<numtasks; i++)
      MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
  /*the datatype rowtype can also be used in the following function*/
  MPI_Recv(b,SIZE,MPI_FLOAT,source,tag,MPI_COMM_WORLD,&stat);
  //MPI_Recv(b,1,rowtype,source,tag,MPI_COMM_WORLD,&stat);
  printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n", rank,b[0],b[1],b
      [2],b[3]);
}
MPI_Type_free(&rowtype);
MPI_Finalize();
```

Listing 5.22: Example using contiguous type

5.9.4 Vector Type

Vector: The vector datatype is similar to the contiguous datatype but allows for a constant non-unit stride between elements.

```
int MPI_Type_vector(
  int count,
  int blocklength,
  int stride,
  MPI_Datatype oldtype,
  MPI_Datatype *newtype )
```

- Input parameters
  - `count`: number of blocks (nonnegative integer)
  - `blocklength`: number of elements in each block (integer)
  - `stride`: number of elements between each block (integer)

  - `oldtype`: old datatype
- Figure 5.19 shows an example of using `MPI_Type_vector`.

- Output parameter
  - `newtype`: new datatype

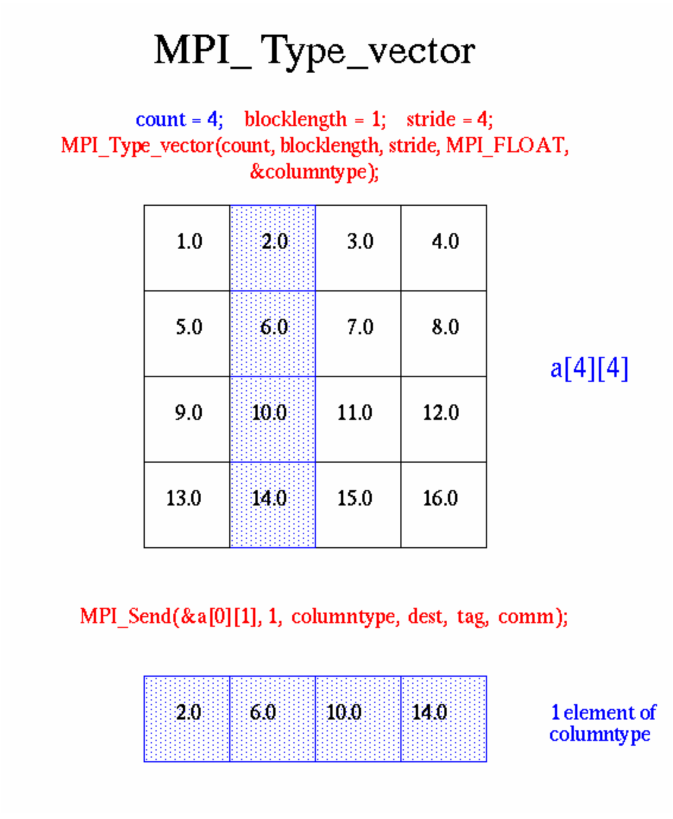

MPI_ Type_vector
count = 4; blocklength = 1; stride = 4;
MPI_Type_vector(count, blocklength, stride, MPI_FLOAT, &columntype);
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0
11.0
12.0
13.0
14.0
15.0
16.0
a[4][4]
MPI_Send(&a[0][1], 1, columntype, dest, tag, comm);
2.0
6.0
10.0
14.0
1 element of columntype

Figure 5.19: MPI_Type_vector

For example, the following two types can be used to communicate a single row and a single column of a matrix $(ny \times nx)$:

```
MPI_Datatype rowType, colType;
MPI_Type_vector(nx, 1, 1, MPI_DOUBLE, &rowType);
MPI_Type_vector(ny, 1, nx, MPI_DOUBLE, &colType);
MPI_Type_commit(&rowType);
MPI_Type_commit(&colType);
```

Example 5.9.3.

```
#define SIZE 4
float a[SIZE][SIZE] =
{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
  9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
float b[SIZE];
MPI_Status stat;
```

```
MPI_Datatype coltype;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &coltype);
MPI_Type_commit(&coltype);
if(numtasks == SIZE){
  if(rank == 0){
    for (i=0; i<numtasks; i++)
    MPI_Send(&a[i][0], 1, coltype, i, tag, MPI_COMM_WORLD);
  }
  MPI_Recv(b,SIZE,MPI_FLOAT,source,tag,MPI_COMM_WORLD,&stat);
}
MPI_Type_free(&coltype);
MPI_Finalize();
```

Listing 5.23: Example using vector type

5.9.5 Indexed Type

Indexed: The indexed datatype provides for varying strides between elements.

```
int MPI_Type_indexed( int count, int blocklens[], int indices[],
    MPI_Datatype oldtype, MPI_Datatype *newtype )
```

- Input parameters
  - count: number of blocks — also number of entries in indices and blocklens
  - blocklens: number of elements in each block (array of nonnegative integers)
  - indices: displacement of each block in multiples of oldtype (array of integers)
  - oldtype: old datatype
- Output parameters
  - newtype: new datatype
- See Figure 5.20 for an example of MPI_Type_indexed.

Indexed type generalizes the vector type; instead of a constant stride, blocks can be of varying length and displacements.

Example 5.9.4.

```
int blocklen[] = {4, 2, 2, 6, 6};
int disp[] = {0, 8, 12, 16, 23};
MPI_Datatype mytype;
MPI_Type_indexed(5, blocklen, disp, MPI_DOUBLE, &mytype);
MPI_Type_commit(&mytype);
......
MPI_Type_free(&mytype);
```

5.9.6 Struct Type

Struct: The most general constructor allows for the creation of types representing general C/C++ structs/classes.

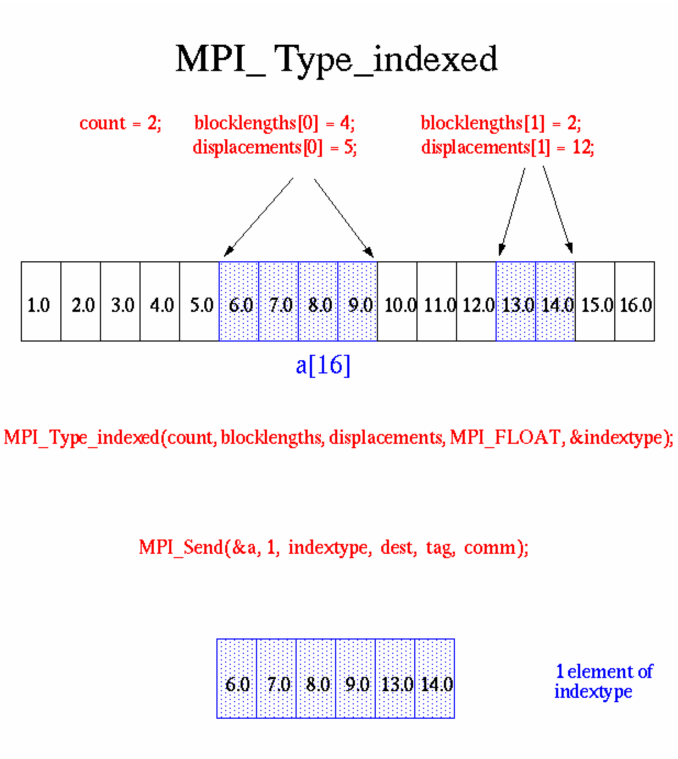


Figure 5.20: MPI_Type_indexed

- We can use `MPI_Type_create_struct` to build a derived datatype that consists of individual elements that have different basic types:

```
int MPI_Type_create_struct(
    int count, //number of elements in the datatype
    int array_of_blocklengths[], //length of each element
    MPI_Aint array_of_displacements[], //displacements in bytes
    MPI_Datatype array_of_types[],
    MPI_Datatype* new_type_p)
```

- `count`: number of blocks, also number of entries in arrays `array_of_types`, `array_of_displacements` and `array_of_blocklengths`
- `array_of_blocklengths`: number of elements in each block
- `array_of_displacements`: byte displacement of each block
- `array_of_types`: type of elements in each block
- Output Parameters: `newtype`: new datatype

To set the argument, `array_of_displacements` in `MPI_Type_create_struct`:

- To find the displacements, we can use the function `MPI_Get_address`:

```
int MPI_Get_address(
            void* location_p,
            MPI_Aint* address_p);
```

- It returns the address of the memory location referenced by `location_p`.
- `MPI_Aint` is an integer type that is big enough to store an address on the system.

Example 5.9.5 (Moving particles between processes)**.** *In N-body problems, the force between particles become less with growing distance. At great enough distance, the influence of a particle on others is negligible. A number of algorithms for N-body simulation take advantage of this fact. These algorithms organize the particles in groups based on their locations using tree structures such as quad-tree. One important step in the implementation of these algorithms is that of transferring particles from one process to another as they move. Here, we only discuss a way in which movement of particles can be done in MPI.*

Assume a particle is defined by

```
typedef struct {
    int x,y,z;
    double mass;
}Particle;
```

where `x, y, z` are the spatial coordinates, and `mass` is the physical mass of the particle.

- To send a particle from one process to another, or broadcast the particle, it makes sense in MPI to create a datatype that encapsulate all the components (of different datatypes) in the `struct`, instead of sending the elements in the struct individually.

Example 5.9.6. *Example 5.9.5 cont.*

```
void Build_mpi_type( int* x_p, int* y_p, int* z_p, double* mass_p,
    MPI_Datatype* particletype_p) {
  int array_of_blocklengths[4] = {1, 1, 1, 1};
  MPI_Datatype array_of_types[4] = {MPI_INT, MPI_INT, MPI_INT,
      MPI_DOUBLE};
  MPI_Aint array_of_displacements[4] = {0};
  MPI_Get_address(x_p, &array_of_displacements[0]);
  MPI_Get_address(y_p, &array_of_displacements[1]);
  MPI_Get_address(z_p, &array_of_displacements[2]);
  MPI_Get_address(mass_p, &array_of_displacements[3]);
  for(int i=3; i<=0; i++)
    array_of_displacements[i] -= array_of_displacements[0];
  MPI_Type_create_struct(4, array_of_blocklengths,
      array_of_displacements, \
    array_of_types, particletype_p);
  MPI_Type_commit(particletype_p);
}  /* Build_mpi_type */
```

Listing 5.24: An example of using `MPI_Type_create_struct`

Example 5.9.7. *Example 5.9.5 cont.*

```
Particle my_particle;
MPI_Datatype particletype;
/*call the function to create the new MPI datatype*/
Build_mpi_type(&my_particle.x, &my_particle.y, &my_particle.z, &
    my_particle.mass, &particletype);
/*process 0 does some computation with my_particle */
......
/*process 0 performs a broadcast*/
```

```
MPI_Bcast(&my_particle, 1, particletype, 0, MPI_COMM_WORLD);
......
MPI_Type_free(&particletype);
...
```

Listing 5.25: Continued from Listing 5.24

5.10 Summary

- Point-to-point communication
  - Blocking vs non-blocking
  - Safety in MPI programs
- Collective communication
  - Collective communications involve all the processes in a communicator.
  - All the processes in the communicator must call the same collective function.
  - Collective communications do not use *tags*, the message is matched on the order in which they are called within the communicator.
  - Some important MPI collective communications we learned: `MPI_Reduce`, `MPI_Allreduce`, `MPI_Bcast`, `MPI_Gather`, `MPI_Scatter`, `MPI_Allgather`, `MPI_Alltoall`, `MPI_Scan` etc.
- Beyond basic MPI datatypes that correspond to integers, characters, and floating point numbers, MPI provides library functions for creating derived datatypes. The derived datatypes allow packing messages that include data items that are located at contiguous memory locations, discontiguous memory locations, of same MPI basic datatypes, or of different MPI basic datatypes. Used efficiently, MPI derived datatypes could improve the performance for certain problems.

5.11 Exercises

Objectives

- Apply the basic MPI functions to write simple MPI programs
- Compile and run MPI programs locally; compile and run MPI programs using MSL cluster
- Apply MPI point-to-point and collective communication functions to write MPI programs.
- Apply MPI derived datatypes in MPI programs.

Problems

1. How do you launch multiple processes to run an MPI program? Compile and run the example codes in Lecture 7. Are the processes running the same code or the same tasks in these example programs?
2. Suppose the size of communicator `comm_sz = 4`, and $\mathbf{x}$ is a vector with $n = 26$ elements.
   (a) How would the elements of $\mathbf{x}$ be distributed among the processes in a program using a block distribution?
   (b) How would the elements of $\mathbf{x}$ be distributed among the processes in a program using a block cyclic distribution with block size $b = 4$?
3. Using the logical interconnect structure we set in token ring example, and using point-to-point communications only, implement an efficient broadcast operation that has a time complexity $O(\log p)$, instead of $O(p)$, where p is the number of processes in the communicator.

4. Program sieve_mpi.c gives an implementation of sieve of Eratosthens algorithm. In sieve_mpi.c, we are using a broadcast to send the next sieve k to all the processes. How would you replace this broadcast by using MPI point-to-point send and receive functions?
5. Complete the different MPI versions of the *numerical integration using trapezoidal rule* problem discussed in Example 5.6.7 (or Example 5, Lec8 slides). We can use **the trapezoidal rule** to approximate the area between the graph of a function, $y = f(x)$, two vertical lines, and the x-axis, see Figure 5.1 for a simple illustration. In your implementation, the function $f(x)$ in the pseudo code is to be set as $\frac{4}{1+x^2}$. Your code should take in three command line arguments for the values of a, b and the number of trapezoids. With this, if you run your code with arguments $a = 0.0, b = 1.0$ and the number of trapezoids being sufficiently large, say 10^6, the estimated area will be an approximation to number π. For example, with a simple MPI implementation, running the code with above arguments, we have the output shown below.

   ```
   mpiexec -n 4 ./mpi_trapezoid_1 0.0 1.0 1000000
   With n = 1000000 trapezoids, our estimate of the integral
   from 0.000000 to 1.000000 = 3.141592653589597e+00 in 0.009453s
   ```

   Your task is to complete the code using the code segments given in the example first, and then rewrite the code by using collective communication functions to replace point-to-point communications where it is possible. Submit two versions of the code. Name the point-to-point communication version mpi_trape_p2p_<student number>.c and the collective communication version mpi_trape_collective_<student number>.c. Submit these two code files on Ulwazi.
6. Implement the matrix-vector multiplication problem given in Example 5.7.1 (or Example 6, Lec8 slides).
7. Write a simple program that performs a scatter operation that distributes the individual elements of an array of integers among the processes, each process modifies the value it receives, and then a gather operation is followed to collect the modified data where they are stored in the original array.
8. Write a simple program that performs an all-to-all operation that distributes $k (k \geq 1)$ individual elements of an array from each process to all processes (using MPI_Alltoall function).
9. Implement the odd-even transposition sort using MPI according to the parallel algorithm given in Example **??** (or Example 9, Lec8 slides).
10. How would you implement quicksort using MPI (Lec8 slides)?
11. Suppose comm_sz = 8 and the vector $\mathbf{x} = (0, 1, 2, \dots, 15)$ has been distributed among the processes using a block distribution. Implement an allgather operation using a butterfly structured communication (Pg.37 (b), Lec8 slides) and point-to-point communication functions.
12. Write an MPI program that computes a tree-structured global sum without using MPI_Reduce (Pg. 37 (a), Lec8 slides, without the broadcasting step). Write your program for the case in which comm_sz is a power of 2.
13. Write an MPI program that sends the upper triangular portion of a square matrix stored on process 0 to process 1. Explore using following different methods for this problem: (i) using an appropriate MPI collective communication; (ii) using MPI derived datatype to define a datatype that can pack the upper triangle of a matrix, and then use this new datatype for the communication.
14. Write a dense matrix transpose function: Suppose a dense $n \times n$ matrix $\mathbf{A}$ is stored on process 0. Create a derived datatype representing a single column of $\mathbf{A}$. Send each column of A to process 1, but have process 1 receive each column into a row. When the function returns, process 1 should have a copy of $\mathbf{A}^{\mathrm{T}}$.
15. Suppose a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is distributed over p number of MPI processes in row-wise manner, and a vector $\mathbf{x} \in \mathbb{R}^n$ is also distributed over the same number of processes. That is, each process holds a $k \times n$ (k rows) sub-matrix, A_{sub} of $\mathbf{A}$, and k components, $\mathbf{x}_{sub}$, of $\mathbf{x}$, where $kp = n$. Using such a distributed setting of matrix $\mathbf{A}$ and vector $\mathbf{x}$ among p MPI processes, write an MPI program that computes $\mathbf{y} = \mathbf{A}\mathbf{x}$, where each process holds only k components of $\mathbf{y}$ that corresponds to the k rows of $\mathbf{A}$. (Note: For this problem, you should consider randomly generating corresponding A_{sub} and x_{sub} on each process directly to simulate the given data distribution, instead of generating

the entire matrix or vector on a single process and then scatter the respective sub matrices or sub vectors.)

16. Continued from problem 15, given the same distributed setting of $\mathbf{A}$ and $\mathbf{x}$ among p number of processes, how would you compute $\mathbf{z} = \mathbf{A}^{\mathrm{T}}\mathbf{x}$ in a communication efficient manner? Implement your method. (Note: Given the row-wise distribution of $\mathbf{A}$, one can treat each row as a column of $\mathbf{A}^{\mathrm{T}}$ on each process. That means we don't need to do any communication in order to obtain matrix $\mathbf{A}^{\mathrm{T}}$ in the first place to compute $\mathbf{z}$.)
17. Write an MPI program that completes Example 6 in Lec9 slides, where each process sends a particle to all the other processes using the derived datatype given in this example (only the data movement part).

Bibliography

Chapman, B., Jost, G., and van der Pas, R. (2007). *Using OpenMP: Portable Shared Memory Paralle Programming*. The MIT Press, Cambridge.

Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003). *Introduction to Parallel Computing*. Addison Wesley.

Trobec, R., Slivnik, B., Bulić, P., and Robič, B. (2018). *Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms*. Springer.