

IKC-MH.57  
Introduction to High Performance and Parallel  
Computing  
Lecture Notes

Cem Özdoğan

December 19, 2023



# Contents

<b>1 Preliminaries</b>	<b>11</b>
1.1 First Meeting	12
1.1.1 Lecture Information	12
1.1.2 Course Overview	13
1.1.3 Text Book	14
1.1.4 Online Resources	15
1.1.5 Grading Criteria	15
1.1.6 Policies	15
1.2 Installation of Required Tools/Programs	16
1.2.1 Linux System	16
1.2.2 Others	16
<b>2 Introduction</b>	<b>19</b>
2.1 View of the Field	20
2.2 Four Decades of Computing	22
2.3 Flynn's Taxonomy of Computer Architecture	23
2.4 Parallel and Distributed Computers	25
2.5 MPI Hands-On; Performance Analysis	27
2.5.1 Analysis of Parallel Summation with Point-to-Point Communications	27
2.6 SIMD Architecture	28
2.7 MIMD Architecture	29
2.8 Shared Memory Organization	30
2.9 Message Passing Organization	31
2.10 MPI Hands-On - Introduction to MPI	32
2.10.1 Parallel Computing	32
2.10.2 Communicating with other processes	32
2.10.3 What is MPI?	33
2.10.4 MPI Implementations	33
2.10.5 Is MPI Large or Small?	34
2.10.6 Where to use MPI?	35
2.10.7 How To Use MPI? Essential!!	35

2.10.8	Getting started . . . . .	35
<b>3</b>	<b>Performance Metrics, Postulates</b>	<b>39</b>
3.1	Performance Analysis . . . . .	40
3.1.1	Computational Models . . . . .	40
3.1.2	Skeptic Postulates For Parallel Architectures . . . . .	44
3.2	MPI Hands-On - Sending and Receiving Messages I . . . . .	45
3.2.1	Current Message-Passing . . . . .	45
3.2.2	The Buffer . . . . .	47
3.2.3	MPI Basic Send/Receive . . . . .	47
3.2.4	Exercises/Examples . . . . .	48
<b>4</b>	<b>Message-Passing Paradigm</b>	<b>51</b>
4.1	Programming Using the Message-Passing Paradigm . . . . .	52
4.1.1	Principles of Message-Passing Programming . . . . .	52
4.1.2	Structure of Message-Passing Programs . . . . .	53
4.1.3	The Building Blocks: Send and Receive Operations . . . . .	53
4.2	MPI Hands-On; Sending and Receiving Messages II . . . . .	60
4.3	MPI: the Message Passing Interface . . . . .	65
4.3.1	Starting and Terminating the MPI Library . . . . .	65
4.3.2	Communicators . . . . .	66
4.3.3	Getting Information . . . . .	67
4.3.4	Sending and Receiving Messages . . . . .	67
4.3.5	Avoiding Deadlocks . . . . .	71
4.3.6	Sending and Receiving Messages Simultaneously . . . . .	73
4.4	MPI Hands-On; Sending and Receiving Messages III . . . . .	74
4.5	Parallelization Application Example . . . . .	79
4.5.1	Pi Computation . . . . .	79
4.6	Overlapping Communication with Computation . . . . .	83
4.6.1	Non-Blocking Communication Operations . . . . .	83
4.7	Collective Communication and Computation Operations . . . . .	86
4.7.1	Broadcast . . . . .	86
4.7.2	Reduction . . . . .	87
4.7.3	Gather . . . . .	88
4.7.4	Scatter . . . . .	90
4.7.5	All-to-All . . . . .	91
4.8	MPI Hands-On; Collective Communications I . . . . .	93
<b>5</b>	<b>Shared Memory Paradigm</b>	<b>97</b>
5.1	Programming Shared Memory . . . . .	98
5.1.1	What is a Thread? . . . . .	98

5.1.2	Threads Model	98
5.1.3	Why Threads?	100
5.1.4	Thread Basics: Creation and Termination	102
5.2	Hands-on; Shared Memory I; Threads	106
5.3	OpenMP: a Standard for Directive Based Parallel Programming	112
5.3.1	The OpenMP Programming Model	113
5.3.2	The OpenMP Design Concepts	118
5.4	Hands-on; Shared Memory II; OpenMP	120
5.5	Parallelization Application Example-OpenMP	123
5.5.1	Computing $\pi$	123
<b>6</b>	<b>GPU parallelization</b>	<b>127</b>
6.1	Exploring the GPU Architecture	128
6.2	Execution and Programming Models	130
6.3	Hands-on; GPU parallelization	135
<b>7</b>	<b>References:</b>	<b>141</b>



# List of Tables

4.1	The minimal set of MPI routines. . . . .	66
4.2	Correspondence between the datatypes supported by MPI and those supported by C. . . . .	68
4.3	Predefined reduction operations. . . . .	88
5.1	Correct and Wrong outputs of the program. . . . .	116
5.2	Sequential and OpenMP outputs for computing $\pi$ number. . . . .	126





# List of Figures

1.1	Recommended Text Books. . . . .	14
2.1	Abstraction Layers . . . . .	21
2.2	View of the Field . . . . .	21
2.3	SISD Architecture. . . . .	23
2.4	SIMD Architecture. . . . .	24
2.5	MIMD Architecture. . . . .	24
2.6	(a) MIMD Shared Memory, (b) MIMD Distributed Memory. . . . .	26
2.7	(a) SIMD Distributed Computers, (b) Clusters. . . . .	26
2.8	Two SIMD Schemes. . . . .	28
2.9	Two MIMD Categories; Shared Memory and Message Passing MIMD Architectures. . . . .	29
2.10	Cooperative–Communicating with other processes. . . . .	32
2.11	One sided–Communicating with other processes. . . . .	33
3.1	Example program segments. . . . .	42
3.2	MPI messages. . . . .	45
3.3	Data+Envelope. . . . .	45
3.4	MPI basic datatypes for C. . . . .	46
4.1	Handshake for a blocking non-buffered send/receive operation. . . . .	55
4.2	Blocking buffered transfer protocols: <i>Left:</i> in the presence of communication hardware with . . . . .	
4.3	Non-blocking non-buffered send and receive operations <i>Left:</i> in absence of communication hardware . . . . .	
4.4	Midpoint Rule. . . . .	79
4.5	Sequential Code Output. . . . .	80
4.6	Parallel Code Output. . . . .	82
4.7	Diagram for Broadcast. . . . .	87
4.8	Diagram for Reduce. . . . .	88
4.9	Diagram for Gather. . . . .	90
4.10	Diagram for All_Gather. . . . .	90
4.11	Diagram for Scatter. . . . .	91
4.12	Diagram for Alltoall. . . . .	92

5.1	Threads model. . . . .	99
5.2	Thread shared memory model. . . . .	99
5.3	Threads <b>Unsafe!</b> Pointers having the same value point to the same data. . . . .	100
5.4	Creating four threads for "printf" function. . . . .	113
5.5	A sample OpenMP program along with its Pthreads translation that might be performed. . . . .	113
5.6	Fork-Join Model. . . . .	115

# Chapter 1

## Preliminaries

## 1.1 First Meeting

- IKC-MH.57 Introduction to High Performance and Parallel Computing 2023-2024 Fall
- FRIDAY 16:00-18:00 (T) H1-86
- Instructor: Cem Özdoğan, Engineering Sciences Dept.  
Faculty of Engineering and Architecture Building, H1-33
- TA: NA
- WEB page: <http://cemozdogan.net/>
- Announcements: Watch this space for the latest updates.

Wednesday, October 9, 2023 In the first lecture, there will be first meeting. The lecture notes will be published soon, see Course Schedule section.

- All the lecture notes will be accessible via [Tentative Course Schedule & Lecture Notes](#).
- All the example c-files (for lecturing and hands-on sessions) will be accessible via the [link](#).

### 1.1.1 Lecture Information

- Almost all computer systems today are multi-core processors systems. Parallel programming must be used to take benefit of the full performance of such systems.
- Visit <https://top500.org/> & <https://www.truba.gov.tr/>.
- Almost all computer systems today are multi-core processors systems. Parallel programming must be used to take benefit of the full performance of such systems.
- Parallel programming also describes the processes and instructions for dividing a larger problem into smaller steps.
- A practical approach to parallel program design and development will be presented in the course content.
- Awareness of potential design and performance concepts in heterogeneous computer architectures will be gained.

- You will be expected to do significant programming assignments, as well as run programs we supply and analyse the output.
- Since we will program in C on a UNIX environment, some experience using C on UNIX will be important.
- In Hands-on sessions, we will concentrate upon the message-passing method of parallel computing and use the standard parallel computing environment called MPI (Message Passing Interface).
- Each student will complete a project based on parallel computing, (distributed computing, cluster computing) for the midterm/final exam.
- Important announcements will be posted to the Announcements section of the web page, so please check this page frequently.
- You are responsible for all such announcements, as well as announcements made in lecture.

### 1.1.2 Course Overview

- IKC-MH.57 is intended to provide students an introduction to parallel/distributed computing and practical experiences in writing parallel programs by using C.
- MPI (Message Passing Interface) message passing in distributed memory systems and Open MP (Open Multi-Processing) in multi-core systems will be taught for parallel programming.
  - MPI is the industry standardized parallelization paradigm in high-performance computing and enables programs to be written that run on distributed memory machines.
  - OpenMP is a thread-based approach to parallelize a program over a single shared memory machine.
- An introduction to the basic concepts of hybrid and accelerated paradigms as Cuda (, OpenCL) programming will be given.
- The course consists of theoretical topics and hands-on practical exercises on parallel programming.
- Upon completion of this course the students will be able to understand/explain/apply;

- Learn how to work in a scientific computing environment.
- Gain awareness of Parallel and High Performance Computing concepts for systems with shared/distributed memory.
- Can write parallel programs both for systems with shared memory using threading (OpenMP ) and systems with distributed memory using message passing (MPI).
- Gains basic knowledge of Cuda OpenCL hybrid and accelerated paradigms.
- Gains the ability and understanding to develop parallel programs to solve a given big numerical/engineering/ scientific problem.

### 1.1.3 Text Book

- Lecture material will be based on them.
- It is strongly advised that student should read textbooks rather than only content with the lecture material supplied from the lecturer.
- Required: No & Recommended:
  - An Introduction to Parallel Programming by Peter Pacheco and Matthew Malensek, Morgan Kaufmann, 2nd edition, 2021, [Elsevier](#).
  - Paralel Algoritmalar: Modeller ve Yöntemler (Yüksek Başarımli Hesaplama) by Abdulsamet Haşiloğlu, 2020, [Papatya Bilim](#).
  - Parallel Programming: Techniques and Application Using Networked Workstations and Parallel Computers by Barry Wilkinson and Michael Allen, 2nd edition, 2005, [Pearson](#).

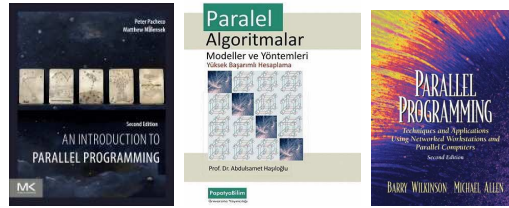


Figure 1.1: Recommended Text Books.

### 1.1.4 Online Resources

The following (some) resources are available online.

- <https://www.cs.purdue.edu/homes/ayg/book/Slides/>
- <https://sites.cs.ucsb.edu/~tyang/class/240a17/>
- <https://hpc-tutorials.llnl.gov/>

### 1.1.5 Grading Criteria

- Midterms & Final Exams: There will be one take-home midterm and one take-home final exam, will count 40% each and 60% of your grade, respectively.
- Homeworks/Assignments (or Term Project): ??

### 1.1.6 Policies

- Attendance is not compulsory (30%), but you are responsible for everything said in class.
- Academic Regulations:  
Derslere devam zorunluluğu ve denetlenmesi  
MADDE 18 - (1) Öğrencilerin derslere, uygulamalara, sınavlara ve diğer çalışmalara devamı zorunludur. Teorik derslerin % 30'undan, uygulamaların % 20'sinden fazlasına devam etmeyen ve uygulamalarda başarılı olamayan öğrenci, o dersin yarıyıl/yılsonu ya da varsa bütünleme sınavına alınmaz. Tekrarlanan derslerde önceki dönemde devam şartı yerine getirilmiş ise derslerde devam şartı aranıp aranmayacağı ilgili birim tarafından hazırlanarak Senato onayına sunulan usul ve esaslar ile belirlenir.
- You can use ideas from the literature (with proper citation).
- The code you submit must be written completely by you. You can use anything from the textbook/notes.
- I encourage you to ask questions in class. You are supposed to ask questions. Don't guess, ask a question!

## 1.2 Installation of Required Tools/Programs

### 1.2.1 Linux System

- Assuming you are using Windows OS.
- Download & Install [VirtualBox-7.0.10-158379-Win.exe](#)
- Download & Install [kubuntu-22.04.3-desktop-amd64.iso](#) under Virtual-Box
  - Post-Installation Steps of Kubuntu
  - ping google.com
  - # Setup "Display Configuration" for resolution
  - cat /proc/cpuinfo
  - sudo apt-get install libopenmpi-dev openmpi-bin libomp-dev
  - # End of Post-Installation Steps of Kubuntu
  - mpicc -o mpi\_helloWorld [mpi\\_helloWorld.c](#)
  - ./mpi\_helloWorld
  - mpirun -np 2 mpi\_helloWorld
  - mpirun -machinefile mf.txt -np 3 mpi\_helloWorld
  - gcc -o omp\_helloWorld -fopenmp [omp\\_helloWorld.c](#)
  - export OMP\_NUM\_THREADS=3
  - ./omp\_helloWorld
  - export OMP\_NUM\_THREADS=8
  - ./omp\_helloWorld
  - [cuda\\_helloWorld.cu](#) later!
  - sudo apt-get update # Regular Updates
  - sudo apt-get upgrade # Regular Upgrades

### 1.2.2 Others

- [See video for Installation of Kubuntu & Parallel Tools](#) under Virtual-Box.
- In take-home exams:



## 1.2. INSTALLATION OF REQUIRED TOOLS/PROGRAMS

17

- Prepare your report/codes.
- Copy your files into a directory named as your ID.
- Upload/send a single file by compressing this directory.
- Check the web page: [IKC-MH.57 2023-2024 Fall](#) frequently.



# Chapter 2

## Introduction

## 2.1 View of the Field

- Data-intensive applications;
  - transaction processing,
  - information retrieval,
  - data mining and analysis,
  - multimedia services,
  - computational physics/chemistry/biology and nanotechnology.
- High performance may come from
  - fast dense circuitry,
  - parallelism.
- Parallel processors are computer systems consisting of
  - multiple *processing units*
  - connected via some *interconnection network*
  - plus the software needed to make the processing units work together.
- *Uniprocessor* – Single processor supercomputers have achieved great speeds and have been pushing hardware technology to the physical limit of chip manufacturing.
  - Physical and architectural bounds (Lithography,  $\mu\text{m}$  size, destructive quantum effects).
  - Proposed solutions are maskless lithography process and nanoimprint lithography for the semiconductor).
  - Uniprocessor systems can achieve to a limited computational power and not capable of delivering solutions to some problems in reasonable time.
- *Multiprocessor* – Multiple processors cooperate to jointly execute a single computational task in order to speed up its execution.
- New issues arise;
  - Multiple threads of control vs. single thread of control

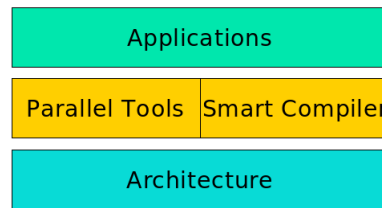


Figure 2.1: Abstraction Layers

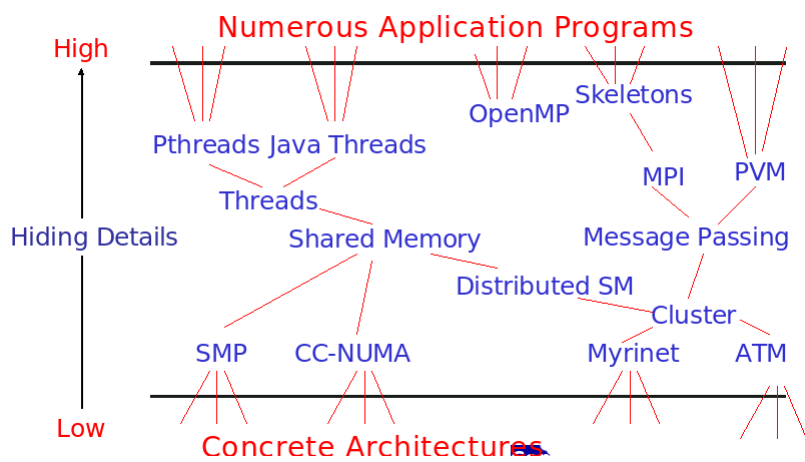


Figure 2.2: View of the Field

- Partitioning for concurrent execution
- Task Scheduling
- Synchronization
- Performance
- Past Trends in Parallel Architecture (inside the box)
  - Completely custom designed components; *processors, memory, interconnects, I/O*.
  - The first three are the major components for the aspects of the parallel computation.
    - \* Longer R&D time (2-3 years).
    - \* Expensive systems.

- \* Quickly becoming outdated.
- In the form of internally linked processors was the main form of parallelism.
- Advances in computer networks  $\Rightarrow$  in the form of networked autonomous computers.
- New Trends in Parallel Architecture (outside the box)
  - Instead of putting everything in a single box and *tightly couple* processors to memory, the Internet achieved a kind of parallelism by *loosely* connecting everything outside of the box.
  - Network of PCs and workstations connected via LAN or WAN forms a Parallel System.
  - Compete favourably (cost/performance).
  - Utilize unused cycles of systems sitting idle.

## 2.2 Four Decades of Computing

Most computer scientists agree that there have been four distinct paradigms or eras of computing. These are: batch, time-sharing, desktop, and network.

1. Batch Era
2. Time-Sharing Era
3. Desktop Era
4. Network Era. They can generally be classified into two main categories:
  - (a) shared memory,
  - (b) distributed memory systems.
    - The number of processors in a single machine ranged from several in a shared memory computer to hundreds of thousands in a massively parallel system.
    - Examples of parallel computers during this era include Sequent Symmetry, Intel iPSC, nCUBE, Intel Paragon, Thinking Machines (CM-2, CM-5), MsPar (MP), Fujitsu (VPP500), and others.
5. Current Trends: Clusters, Grids.

## 2.3 Flynn's Taxonomy of Computer Architecture

- The most popular taxonomy of computer architecture was defined by Flynn in 1966.
- Flynn's classification scheme is based on the notion of a stream of information.
  - Two types of information flow into a processor:
    1. **Instruction.** The instruction stream is defined as the sequence of instructions performed by the processing unit.
    2. **Data.** The data stream is defined as the data traffic exchanged between the memory and the processing unit.
- According to Flynn's classification, either of the instruction or data streams can be **single** or **multiple**.
- Computer architecture can be classified into the following four distinct categories:
  1. single instruction single data streams (SISD)
  2. single instruction multiple data streams (SIMD)
  3. multiple instruction single data streams (MISD)
  4. multiple instruction multiple data streams (MIMD).
- SISD;

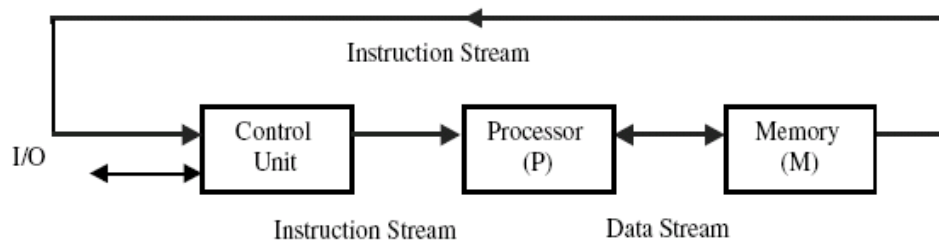


Figure 2.3: SISD Architecture.

- SIMD;

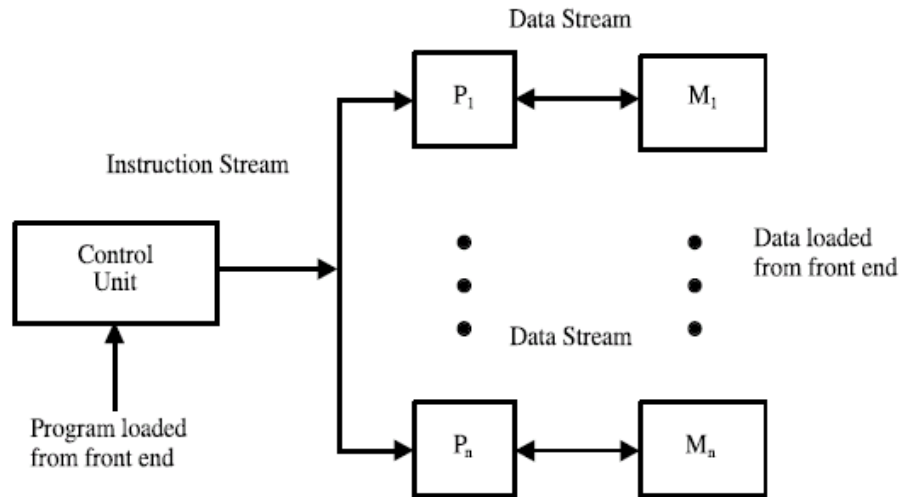


Figure 2.4: SIMD Architecture.

- MIMD;

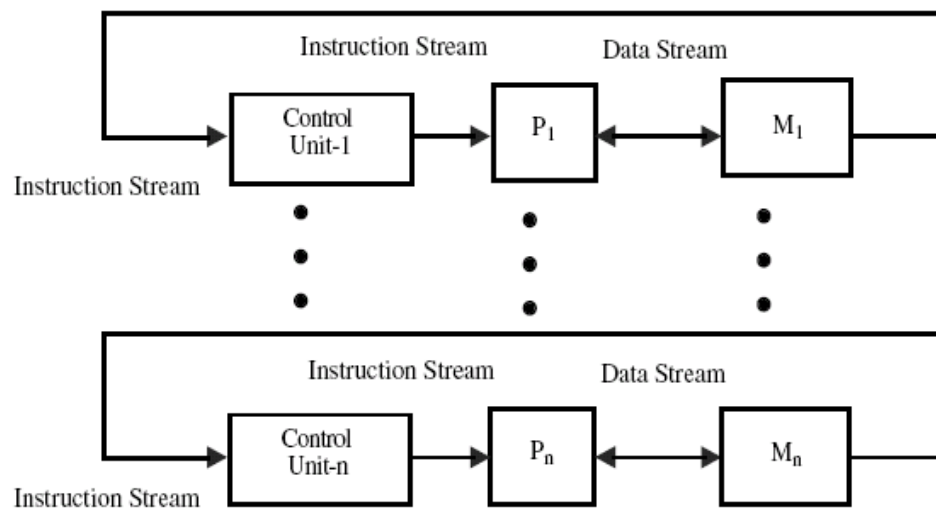


Figure 2.5: MIMD Architecture.

Parallel computers are either SIMD or MIMD.



- When there is only one control unit and all processors execute the same instruction in a synchronized fashion, the parallel machine is classified as SIMD.
- In a MIMD machine, each processor has its own control unit and can execute different instructions on different data.
- In the MISD category, the same stream of data flows through a linear array of processors executing different instruction streams. In practice, there is no viable MISD machine; however, some authors have considered *pipelined machines* as examples for MISD.

## 2.4 Parallel and Distributed Computers

- The processing units can communicate and interact with each other using either
  - shared memory
  - or message passing methods.
- The interconnection network for shared memory systems can be classified as
  - bus-based
  - switch-based.
- SIMD Computers
- MIMD Shared Memory, MIMD Distributed Memory
- Bus based, Switch based
- CC-NUMA
- Clusters, Grid Computing
  - Grids are geographically distributed platforms for computation.
  - They provide dependable, consistent, general, and inexpensive access to high end computational capabilities.

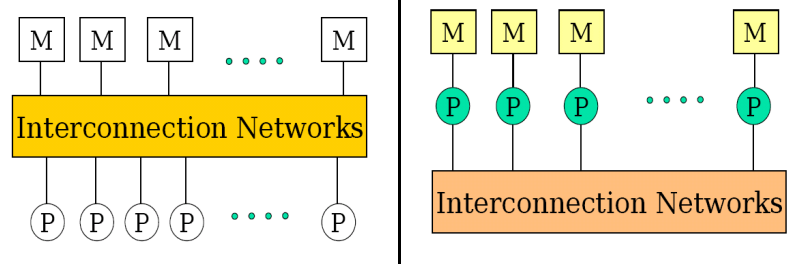


Figure 2.6: (a) MIMD Shared Memory, (b) MIMD Distributed Memory.

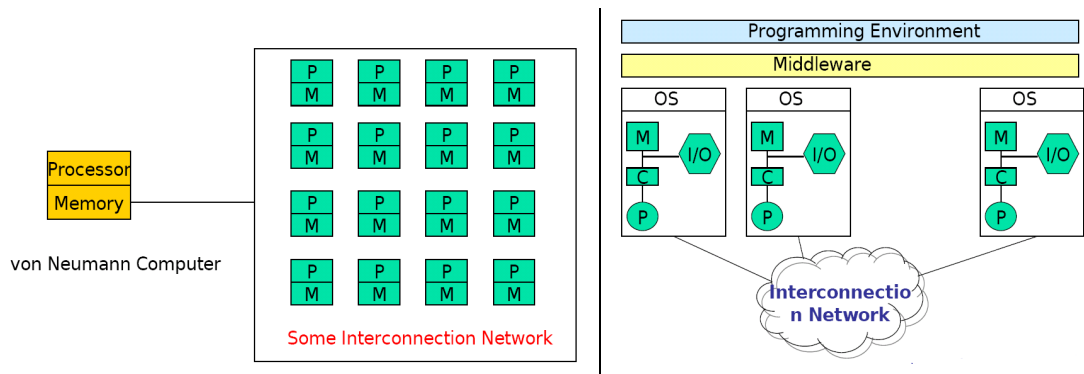


Figure 2.7: (a) SIMD Distributed Computers, (b) Clusters.

## 2.5 MPI Hands-On; Performance Analysis

### 2.5.1 Analysis of Parallel Summation with Point-to-Point Communications

Parallel Summation; the given [program](#) adds  $n$  numbers both in sequential and parallel.

- Take a (quick) look at the source code.
- Download the [binary](#).
- Download the Excel [file](#) to fill the table by increasing the value of  $n$  and the value of  $nproc$ .
- Login to the *kubuntu under VirtualBox* system.
- Make all the test runs by the following command;

```
mpirun -np XXprocXX ./code00 XXNXX  
such as: mpirun -np 2 ./code00 10000
```

- You may erase your output files (#outputfile.dat) after your tests.

```
rm -f *outputfi*
```

- Analyse your plots.

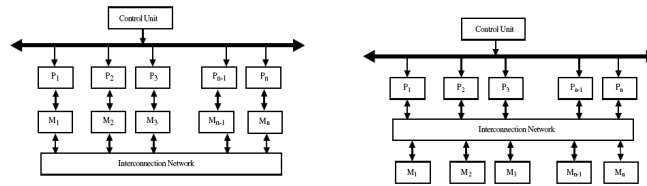


Figure 2.8: Two SIMD Schemes.

## 2.6 SIMD Architecture

- The SIMD model of parallel computing consists of two parts:
  1. a front-end computer of the usual von Neumann style,
  2. a processor array.
- Each processor in the array has a small amount of local memory where the *distributed data resides* while it is being processed in parallel.
- The similarity between serial and data parallel programming is one of the strong points of *data* parallelism.
- Processors either do nothing or exactly the same operations at the same time.
- In SIMD architecture, parallelism is exploited by applying simultaneous operations across large sets of data.
- There are two main configurations that have been used in SIMD machines.
  1. Each processor has its own local memory.
    - Processors can communicate with each other through the interconnection network.
    - If the interconnection network does not provide direct connection between a given pair of processors, then this pair can exchange data via an intermediate processor.
  2. In the second SIMD scheme,
    - Processors and memory modules communicate with each other via the interconnection network.

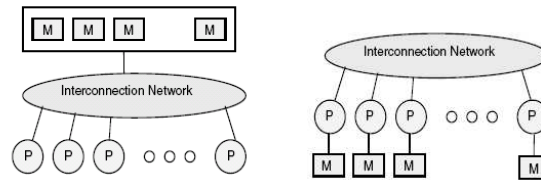


Figure 2.9: Two MIMD Categories; Shared Memory and Message Passing MIMD Architectures.

- Two processors can transfer data between each other via intermediate memory module(s) or possibly via intermediate processor(s).

## 2.7 MIMD Architecture

- It was apparent that distributed memory is the only way efficiently to increase the number of processors managed by a parallel and distributed system.
- If scalability to larger and larger systems (as measured by the number of processors) was to continue, systems had to use distributed memory techniques.
- Two broad categories, see Figure 2.9:
  1. **Shared memory.** Processors exchange information through their **central shared memory**.
    - Because access to shared memory is balanced, these systems are also called SMP (symmetric multiprocessor) systems.
  2. **Message passing.** Also referred to as distributed memory. Processors exchange information through their **interconnection network**.
    - There is no global memory, so it is necessary to *move data from one local memory to another by means of message passing*.
    - This is typically done by a **Send/Receive pair** of commands, which must be written into the application software by a programmer
    - Data copying and dealing with consistency issues.

- Programming in the shared memory model was easier, and designing systems in the message passing model provided scalability.
- The distributed-shared memory (DSM) architecture began to appear in systems. In such systems,
  - memory is physically distributed; for example, the hardware architecture follows the message passing school of design,
  - but the programming model follows the shared memory school of thought.
  - Thus, the DSM machine is a *hybrid* that takes advantage of both design schools.

## 2.8 Shared Memory Organization

- A number of basic issues in the design of shared memory systems have to be taken into consideration.
- These include access control, synchronization, protection/security.
  - **Access control** determines which process accesses are possible to which resources.
  - **Synchronization** constraints limit the time of accesses from sharing processes to shared resources.
  - **Protection** is a system feature that prevents processes from making arbitrary access to resources belonging to other processes.
- The simplest shared memory system consists of one memory module that can be accessed from two processors.
- Requests arrive at the memory module through its two ports.

Depending on the interconnection network, a shared memory system leads to systems can be classified as:

- **Uniform Memory Access (UMA)**. A shared memory is accessible by all processors through an interconnection network in the same way a single processor accesses its memory.
  - Therefore, all processors have equal access time to any memory location.

- **Nonuniform Memory Access (NUMA)**. Each processor has part of the shared memory attached.
  - However, the access time to modules depends on the distance to the processor. This results in a nonuniform memory access time.
- **Cache-Only Memory Architecture (COMA)**. Similar to the NUMA, each processor has part of the shared memory in the COMA.
  - However, in this case the shared memory consists of cache memory.
  - A COMA system requires that data be migrated to the processor requesting it.

## 2.9 Message Passing Organization

- Message passing systems are a class of multiprocessors in which each processor has access to its own local memory.
- Unlike shared memory systems, communications in message passing systems are performed via send and receive operations.
- Nodes are typically able to store messages in buffers (temporary memory locations where messages wait until they can be sent or received), and perform send/receive operations at the same time as processing.
- The processing units of a message passing system may be connected in a variety of ways ranging from architecture-specific interconnection structures to geographically dispersed networks.

Two important design factors must be considered in designing interconnection networks for message passing systems. These are the link bandwidth and the network latency.

1. The *link bandwidth* is defined as the number of bits that can be transmitted per unit time (bits/s).
2. The *network latency* is defined as the time to complete a message transfer.

## 2.10 MPI Hands-On - Introduction to MPI

### 2.10.1 Parallel Computing

- Separate workers or processes.
- Interact by exchanging information.
- Data-Parallel. Same operations on different data. Also called SIMD.
- SPMD. Same program, different data.
- MIMD. Different programs, different data.

### 2.10.2 Communicating with other processes

Data must be exchanged with other workers;

- **Cooperative** — all parties agree to transfer data.
  - Message-passing is an approach that makes the exchange of data cooperative.
  - Data must both be explicitly sent and received.

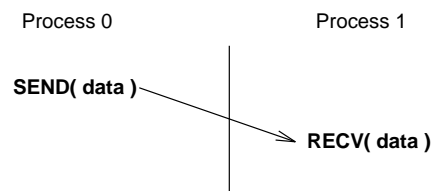


Figure 2.10: Cooperative—Communicating with other processes.

- **One sided** — one worker performs transfer of data.
  - One-sided operations between parallel processes include remote memory reads and writes.
  - An advantage is that data can be accessed without waiting for another process.



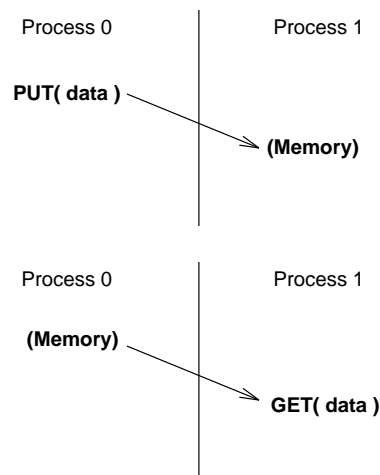


Figure 2.11: One sided-Communicating with other processes.

### 2.10.3 What is MPI?

- *A message-passing library specification*
  - message-passing model.
  - not a compiler specification.
  - not a specific product.
- For parallel computers, clusters, and heterogeneous networks.
- Designed to provide access to advanced parallel hardware for
  - end users.
  - library writers.
  - tool developers.

### 2.10.4 MPI Implementations

- **Open MPI** (a project combining technologies and resources from several other projects (FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI))
- MPICH (Argonne National Laboratory).
- UNIFY (Mississippi State University).
- CHIMP (Edinburgh Parallel Computing Centre).

- LAM (Ohio Supercomputer Center).
- MPI for the Fujitsu AP1000 (Australian National University).
- Cray MPI Product for the T3D (Cray Research and the Edinburgh Parallel Computing Center).
- IBM's MPI for the SP.
- SGI's MPI for 64-bit mips3 and mips4.
- PowerMPI for Parsytec Systems.
- HP's MPI implementation.
- ...

### 2.10.5 Is MPI Large or Small?

- MPI is large (See this [openMPI link](#))
  - MPI's extensive functionality requires many functions.
  - Number of functions not necessarily a measure of complexity.
- MPI is small. Many parallel programs can be written with just 6 basic functions.
  - **MPI\_Init**– Initialise MPI.
  - **MPI\_Comm\_size**– Find out how many processes there are.
  - **MPI\_Comm\_rank**– Find out which process I am.
  - **MPI\_Send**– Send a message.
  - **MPI\_Recv**– Receive a message.
  - **MPI\_Finalize**– Terminate MPI.
- MPI is just right
  - One can access flexibility when it is required.
  - One need not master all parts of MPI to use it.

### 2.10.6 Where to use MPI?

- You need a portable parallel program.
- You are writing a parallel library.
- You have irregular or dynamic data relationships that do not fit a data parallel model.

Where *not* to use MPI:

- You can use HPF or a parallel Fortran 90.
- You don't need parallelism at all.
- You can use libraries (which may be written in MPI).

### 2.10.7 How To Use MPI? Essential!!

1. When possible, start with a debugged serial version.
2. Design parallel algorithm.
3. Write code, making calls to MPI library.
4. Compile and run using implementation specific utilities.
5. Run with a few nodes first, increase number gradually.

### 2.10.8 Getting started

#### Writing MPI programs I

First program with MPI ( [hello.c](#) ). Write the following code and study the response.

```
1 #include "mpi.h"
2 #include <stdio.h>
3
4 int main( argc , argv )
5 int argc ;
6 char **argv ;
7 {
8     MPI_Init( &argc , &argv );
9     printf( "Hello world\n" );
10    MPI_Finalize();
11    return 0;
12 }
```

- `#include "mpi.h"`  
provides basic MPI definitions and types.
- `MPI_Init`  
starts MPI.
- `MPI_Finalize`  
exits MPI.
- Note that all non-MPI routines are local; thus the  
`printf`  
  
run on each process.

```
mpicc -o hello hello.c
mpirun -np 2 hello
```

## Writing MPI programs II

Another Example (Again no message-passing) ( [hello1.c](#)):

```
1 #include <stdio.h>
2 #include <mpi.h>
3
4 int main(argc, argv)
5 int argc;
6 char *argv[];
7 {
8     char name[BUFSIZ];
9     int length;
10    MPI_Init(&argc, &argv);
11    MPI_Get_processor_name(name, &length);
12    printf("%s: hello world\n", name);
13    MPI_Finalize();
14 }
```

## Writing MPI programs III

Another Example (Again hello and again no message-passing) ( [hello2.c](#)):

```
1 #include "mpi.h"
2 #include <stdio.h>
3 #include <unistd.h>
4
5 int main( argc, argv )
6 int argc;
7 char **argv;
8 {
9     int rank, size;
```

```
10 MPI_Init( &argc, &argv );
11 MPI_Comm_rank( MPLCOMM_WORLD, &rank );
12 MPI_Comm_size( MPLCOMM_WORLD, &size );
13 printf( "Hello world! I'm %d of %d\n", rank, size );
14 sleep(10);
15 MPI_Finalize();
16 return 0;
17 }
```

Two of the first questions asked in a parallel program are:

1. How many processes are there? Answered with *MPI\_Comm\_size*
2. Who am I? Answered with *MPI\_Comm\_rank*. The **rank** is a number between zero and **size-1**.

### Exercise - Getting Started

- Designing, compiling, and running a simple MPI program.
  - Write a program that combines all the "Hello world" programs above.
  - Execute several times and/or try different number of nodes. What does the output look like? Why it does differ?



## Chapter 3

# Performance Metrics, Postulates

### 3.1 Performance Analysis

- Analysis of the performance measures of parallel programs.
- Two computational models;
  1. the equal duration processes
  2. parallel computation with serial sections.
- Two measures;
  1. speed-up factor
  2. efficiency.
- The impact of the communication overhead on the overall speed performance of multiprocessors.
- The scalability of parallel systems.

#### 3.1.1 Computational Models

##### Equal Duration Model

Assume that a given computation can be divided into concurrent tasks for execution on the multiprocessor.

- In this model ( $t_s$ : execution time of the whole task using a single processor),
  - a given task can be divided into  $n$  equal subtasks,
  - each of which can be executed by one processor,
  - the time taken by each processor to execute its subtask is

$$t_p = \frac{t_s}{n}$$

- since all processors are executing their subtasks simultaneously, then the time taken to execute the whole task is

$$t_p = \frac{t_s}{n}$$

- The speed-up factor of a parallel system can be defined as



- the ratio between the time taken by a single processor to solve a given problem
- to the time taken by a parallel system consisting of  $n$  processors to solve the same problem.

- Speed Up;

$$S(n) = \frac{t_s}{t_p} = \frac{t_s}{t_s/n} = n \quad (3.1)$$

- This equation indicates that, according to the equal duration model, the speed-up factor resulting from using  $n$  processors is equal to the number of processors used ( $n$ ).
- One important factor has been ignored in the above derivation.
- This factor is the communication overhead,  $t_c$ , which results from the time needed for processors to communicate and possibly exchange data while executing their subtasks.
- Then the actual time taken by each processor to execute its subtask is given by

$$S(n) = \frac{t_s}{t_p} = \frac{t_s}{t_s/n + t_c} = \frac{n}{1 + n * t_c/t_s} \quad (3.2)$$

- This equation indicates that the **relative values of  $t_s$  and  $t_c$  affect the achieved speed-up factor**.
- A number of cases can then be studied:
  1. if  $t_c \ll t_s$  then the potential speed-up factor is approximately  $n$
  2. if  $t_c \gg t_s$  then the potential speed-up factor is  $t_s/t_c \ll 1$
  3. if  $t_c = t_s$  then the potential speed-up factor is  $n/n + 1 \cong 1$ , for  $n \gg 1$ .
- In order to scale the speed-up factor to a value between 0 and 1, we divide it by the number of processors,  $n$ .
- The resulting measure is called the efficiency,  $E$ .
- The efficiency is a measure of the **speed-up achieved per processor**.
- According to the simple equal duration model, the efficiency  $E$  is equal to 1, if the communication overhead is ignored.

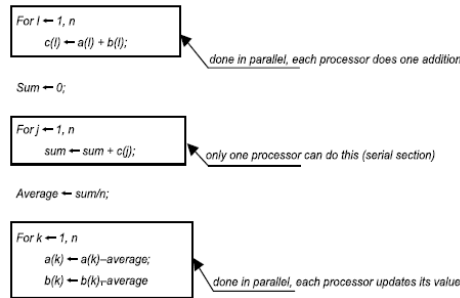


Figure 3.1: Example program segments.

- However if the communication overhead is taken into consideration, the efficiency can be expressed as

$$E = \frac{1}{1 + n * t_c/t_s} \quad (3.3)$$

- Although simple, the equal duration model is however unrealistic.
- This is because it is based on the assumption that a given task can be divided into a number of equal subtasks.
- However, real algorithms contain some (serial) parts that cannot be divided among processors.
- These (serial) parts must be executed on a single processor.
- In Figure program segments, we assume that we start with a value from each of the two arrays (vectors)  $a$  and  $b$  stored in a processor of the available  $n$  processors.
  - The first program block can be done in parallel; that is, each processor can compute an element from the array (vector)  $c$ . The elements of array  $c$  are now distributed among processors, and each processor has an element.
  - The next program segment cannot be executed in parallel. This block will require that the elements of array  $c$  be communicated to one processor and are added up there.
  - The last program segment can be done in parallel. Each processor can update its elements of  $a$  and  $b$ .

### Parallel Computation with Serial Sections Model

- It is assumed (or known) that a **fraction**  $f$  of the given task (computation) is not dividable into concurrent subtasks.
- The remaining part  $(1 - f)$  is assumed to be dividable into concurrent subtasks.
- The time required to execute the task on  $n$  processors is

$$t_p = t_s * f + (1 - f) * (t_s/n)$$

- The speed-up factor is therefore given by

$$S(n) = \frac{t_s}{t_s * f + (1 - f) * (t_s/n)} = \frac{n}{1 + (n - 1) * f} \quad (3.4)$$

- According to this equation, the potential speed-up due to the use of  $n$  processors is determined primarily by the fraction of code that cannot be divided.
- If the task (program) is completely serial, that is,  $f = 1$ , then no speed-up can be achieved regardless of the number of processors used.
- This principle is known as Amdahl's law.
- It is interesting to note that according to this law, the maximum speed-up factor is given by

$$\lim_{n \rightarrow \infty} S(n) = \frac{1}{f}$$

- Therefore, the improvement in performance (speed) of a parallel algorithm over a sequential one is
  - limited not by the number of processors employed
  - but rather by the fraction of the algorithm that cannot be parallelized.
- According to Amdahl's law, researchers were led to believe that a substantial increase in speed-up factor would **not be possible** by using parallel architectures.
- NOT parallelizable;
  - communication overhead,
  - a sequential fraction,  $f$

### 3.1.2 Skeptic Postulates For Parallel Architectures

#### Amdahl's Law

- Amdahl's law made it so pessimistic to build parallel computer systems.
- Due to the intrinsic limit set on the performance improvement (speed) regardless of the number of processors used.
- An interesting observation to make here is that according to Amdahl's law,  $f$  is fixed and does not scale with the problem size,  $n$ .
- However, it has been practically observed that some **real parallel algorithms** have a fraction that is a function of  $n$ .
- Let us assume that  $f$  is a function of  $n$  such that  $\lim_{n \rightarrow \infty} f(n) = 0$

$$\lim_{n \rightarrow \infty} S(n) = \lim_{n \rightarrow \infty} \frac{n}{1 + (n - 1) * f(n)} = n \quad (3.5)$$

- This is clearly in contradiction to Amdahl's law.
- It is therefore **possible to achieve a linear speed-up factor** for large-sized problems, given that

$$\lim_{n \rightarrow \infty} f(n) = 0$$

a condition that has been practically observed.

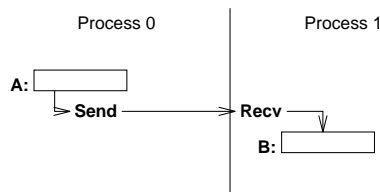


Figure 3.2: MPI messages.

## 3.2 MPI Hands-On - Sending and Receiving Messages I

Questions:

- To whom is data sent?
- What is sent?
- How does the receiver identify it?

### 3.2.1 Current Message-Passing

Message = data + envelope

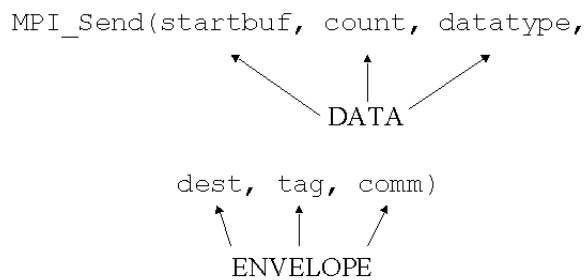


Figure 3.3: Data+Envelope.

- MPI Data; Arguments
  - **startbuf** (starting location of data)
  - **count** (number of elements)

- \* receive count  $\geq$  send count
- **datatype** (basic or derived)
  - \* receiver datatype = send datatype (unless MPI\_PACKED)
  - \* Elementary (all C types). Specifications of elementary datatypes allows heterogeneous communication.
  - \* MPI basic datatypes for C:

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
MPI_BYTE	
MPI_PACKED	

Figure 3.4: MPI basic datatypes for C.

- MPI Envelope; Arguments
  - **destination or source**
    - \* rank in a communicator
    - \* receive = sender or MPI\_ANY\_SOURCE
  - **tag**
    - \* integer chosen by programmer
    - \* receive = sender or MPI\_ANY\_TAG (wild cards allowed)
  - **communicator**
    - \* defines communication "space"
    - \* group + context
    - \* receive = send
  - Collective operations typically operated on all processes.

- All communication (not just collective operations) takes place in groups.
- A context partitions the communication space. A message sent in one context cannot be received in another context. Contexts are managed by the system.
- A group and a context are combined in a communicator.
- Source/destination in send/receive operations refer to rank in group associated with a given communicator.

### 3.2.2 The Buffer

Sending and receiving only a contiguous array of bytes. Specified in MPI by *starting address*, *datatype*, and *count*

- hides the real data structure from hardware which might be able to handle it directly.
- requires pre-packing dispersed data
  - rows of a matrix stored columnwise.
  - general collections of structures.
- prevents communications between machines with different representations (even lengths) for same data type

### 3.2.3 MPI Basic Send/Receive

Thus the basic send (blocking!!) has become:

```
MPI_Send( start, count, datatype, dest, tag, comm )
```

and the receive (blocking!!):

```
MPI_Recv(start, count, datatype, source, tag, comm, status)
```

The source, tag, and count of the message actually received can be retrieved from `status`.

```
MPI_Status status;
MPI_Recv( ..., &status );
... status.MPI_TAG; ... status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &count );
```

*MPI\_Get\_count* may be used to determine how much data of a particular type was received.

Two simple collective operations (just to introduce!):

```
MPI_Bcast(start, count, datatype, root, comm)
MPI_Reduce(start, result, count, datatype,
           operation, root, comm)
```

### 3.2.4 Exercises/Examples

1. An example for communication world `code1.c`.

```
1 #include <stdio.h>
2 #include <mpi.h>
3
4 int main(int argc, char **argv)
5 {
6     int size, my_rank;
7     MPI_Init(&argc,&argv);
8     MPI_Comm_size(MPLCOMM_WORLD,&size);
9     MPI_Comm_rank(MPLCOMM_WORLD,&my_rank);
10    // printf("Executed by all processors: Hello! It is processor %d.\n
11        ", my_rank);
12    if (my_rank == 0)
13    {
14        printf("Hello! It is processor 0. There are %d processors in this
15            comm. world.\n", size);
16        printf("I am process %i out of %i: Hello world!\n",my_rank, size)
17        ;
18    }
19    else
20    {
21        printf("I am process %i out of %i: Hello world!\n", my_rank, size)
22        ;
23    }
24    MPI_Finalize();
25    return 0;
26 }
```



2. Write a program to send/receive and print out your name and age to each processors. [code2.c](#).

```

1 #include <stdio.h>
2 #include <mpi.h>
3 #include <string.h>
4
5 int main(int argc, char **argv)
6 {
7     int my_rank; /* rank of process */
8     int size; /* number of processes */
9     int dest; /* rank of receiver */
10    int my_age = 4; /* storage for my_age */
11    char message[100]; /* storage for message */
12    int recv_my_age = 0; /* storage for received my_age */
13    MPI_Status status; /* return status for receive */
14
15    MPI_Init(&argc, &argv); /* Start up MPI */
16    MPI_Comm_size(MPLCOMM_WORLD, &size); /* Find out number of processes
17    */
18    MPI_Comm_rank(MPLCOMM_WORLD, &my_rank); /* Find out process rank */
19    if (my_rank == 0) /* rank of sender */
20    {
21        sprintf(message, "IKC-MH.57"); /* Create message */
22        for (dest=1; dest<size; dest++)
23        {
24            printf("Sending to worker num:%d\n", dest);
25            MPI_Send(&my_age, 1, MPI_INT, dest, 1, MPLCOMM_WORLD);
26            MPI_Send(message, strlen(message)+1, MPLCHAR, dest, 2,
27            MPLCOMM_WORLD);
28        }
29    }
30    else
31    {
32        MPI_Recv(&recv_my_age, 1, MPI_INT, 0, 1, MPLCOMM_WORLD, &status);
33        MPI_Recv(message, sizeof(message), MPLCHAR, 0, 2, MPLCOMM_WORLD,
34        &status);
35        printf("=====\n");
36        printf("I am node: %d\n", my_rank);
37        printf("My age: %d\n", recv_my_age);
38        printf("My name: %s\n", message);
39        printf("=====\n");
40    }
41    MPI_Finalize(); /* Shut down MPI */
42    return 0;
43 }

```



## Chapter 4

# Message-Passing Paradigm

## 4.1 Programming Using the Message-Passing Paradigm

### 4.1.1 Principles of Message-Passing Programming

Set of Primitives: Allows processes to communicate with each other.

- A message passing architecture uses a set of primitives that allows processes to communicate with each other.
- i.e., *send*, *receive*, *broadcast*, and *barrier*.

There are two key attributes that characterize the message -passing programming paradigm.

1. the first is that it assumes a partitioned address space,
  2. the second is that it supports only explicit parallelization.
- Each data element must **belong to one of the partitions** of the space;
    - hence, data must be explicitly partitioned and placed.
    - **Adds complexity, encourages data locality.**
  - All interactions (read-only or read/write) require **cooperation of two processes**:
    1. the process that has the data,
    2. the process that wants to access the data.
  - Primary advantage of explicit two-way interactions is that the programmer is fully aware of all the costs of non-local interactions
  - The programmer is responsible for analyzing the underlying serial algorithm/application.
  - As a result, programming using the message-passing paradigm tends to be hard and intellectually demanding.
  - However, on the other hand, **properly written** message-passing programs can often *achieve very high performance* and *scale to a very large number of processes*.

### 4.1.2 Structure of Message-Passing Programs

- Message-passing programs are often written using the asynchronous or loosely synchronous paradigms.
- In the *asynchronous* paradigm, all concurrent tasks execute asynchronously.
  - However, such programs can be harder and can have non-deterministic behavior due to race conditions.
- *Loosely synchronous* programs are a good compromise between two extremes.
  - In such programs, tasks or subsets of tasks synchronize to perform interactions.
  - However, between these interactions, tasks execute completely asynchronously.
- Most message-passing programs are written using the single program multiple data (*SPMD*).
- SPMD programs can be loosely synchronous or completely asynchronous.

### 4.1.3 The Building Blocks: Send and Receive Operations

- Since interactions are accomplished by *sending* and *receiving* messages, the basic operations in the message-passing programming paradigm are **send** and **receive**.
- In their simplest form, the prototypes of these operations are defined as follows:

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

- *sendbuf* points to a buffer that stores the data to be sent,
- *recvbuf* points to a buffer that stores the data to be received,
- *nelems* is the number of data units to be sent and received,.
- *dest* is the identifier of the process that receives the data,.
- *source* is the identifier of the process that sends the data.

```

1  P0                                P1
2
3  a = 100;                          receive(&a, 1, 0)
4  send(&a, 1, 1);                    printf("%d\n", a);
5  a=0;

```

- Process  $P_0$  sends a message to process  $P_1$  which receives and prints the message.
- The important thing to note is that process  $P_0$  changes the value of  $a$  to 0 immediately following the send.
- The semantics of the send operation require that the value received by process  $P_1$  must be 100 (not 0).
- That is, the value of  $a$  at the time of the send operation must be the value that is received by process  $P_1$ .
- It may seem that it is quite straightforward to ensure the semantics of the send and receive operations.
- *However, based on how the send and receive operations are implemented this may not be the case.*

### Blocking Message Passing Operations

- As a result, *if the send operation programs the communication hardware and returns before the communication operation has been accomplished, process  $P_1$  might receive the value 0 in a instead of 100!*
- A simple solution to the problem presented in the code fragment above is for the send operation to return only when it is semantically safe to do so.
- Note that this is not the same as saying that the send operation returns only after the receiver has received the data.
- It simply means that the sending operation blocks until it can guarantee that the semantics will not be violated on return irrespective of what happens in the program subsequently.
- There are two mechanisms by which this can be achieved.
  1. Blocking Non-Buffered Send/Receive
  2. Blocking Buffered Send/Receive

1 Blocking Non-Buffered Send/Receive

#### 4.1. PROGRAMMING USING THE MESSAGE-PASSING PARADIGM 55

- The send operation does not return until the matching receive has been encountered at the receiving process.
- When this happens, the message is sent and the send operation returns upon completion of the communication operation.
- Typically, this process involves a *handshake* between the sending and receiving processes (see Figure 4.1).

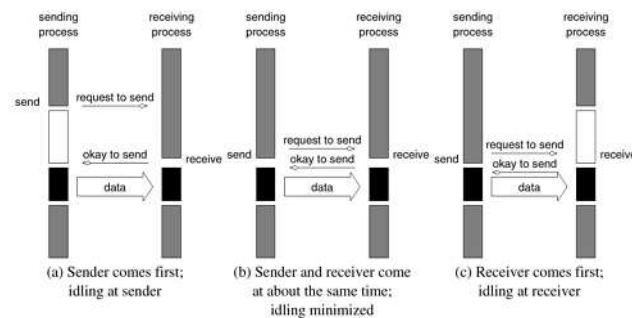


Figure 4.1: Handshake for a blocking non-buffered send/receive operation.

- The sending process sends a request to communicate to the receiving process.
  - When the receiving process encounters the target receive, it responds to the request.
  - The sending process upon receiving this response initiates a transfer operation.
  - Since there are no buffers used at either sending or receiving ends, this is also referred to as a **non-buffered blocking** operation.
- *Idling Overheads in Blocking Non-Buffered Operations:* It is clear from the figure that a blocking non-buffered protocol is suitable when the send and receive are posted at roughly the same time (see Figure(b)).
  - However, in an asynchronous environment, this may be impossible to predict.
  - This idling overhead is one of the major drawbacks of this protocol.

- *Deadlocks in Blocking Non-Buffered Operations:* Consider the following simple exchange of messages that can lead to a deadlock:

```

1   P0                               P1
2
3   send(&a, 1, 1);                   send(&a, 1, 0);
4   receive(&b, 1, 1);                receive(&b, 1, 0);

```

- The code fragment makes the values of  $a$  available to both processes  $P_0$  and  $P_1$ .
- However, if the send and receive operations are implemented using a blocking non-buffered protocol,
  - the send at  $P_0$  waits for the matching receive at  $P_1$
  - whereas the send at process  $P_1$  waits for the corresponding receive at  $P_0$ ,
  - resulting in an infinite wait.

- Deadlocks are very easy in blocking protocols and care must be taken to break cyclic waits.

## 2 Blocking Buffered Send/Receive

- A simple solution to the *idling* and *deadlocking* problems outlined above is to rely on **buffers** at the sending and receiving ends.
- On a send operation, the sender simply *copies the data into* the designated buffer and *returns after the copy operation has been completed*.
- The sender process can now continue with the program knowing that any changes to the data will not impact program semantics.
- Note that at the receiving end, the data cannot be stored directly at the target location since this would violate program semantics.
- Instead, the data is copied into a buffer at the receiver as well.
- When the receiving process encounters a receive operation, it checks to see if the message is available in its receive buffer. If so, the data is copied into the target location.



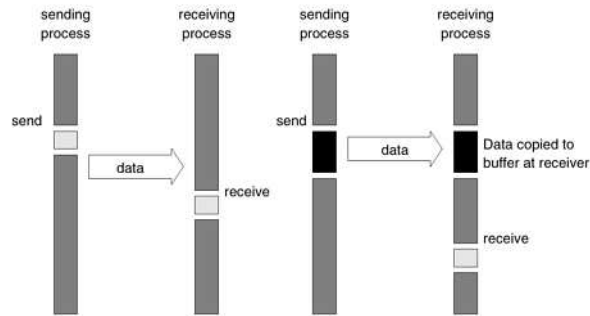


Figure 4.2: Blocking buffered transfer protocols: *Left*: in the presence of communication hardware with buffers at send and receive ends; and *Right*: in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

- In general, if the parallel program is highly synchronous, non-buffered sends may perform better than buffered sends.
- However, generally, this is not the case and buffered sends are desirable unless buffer capacity becomes an issue.
- *Deadlocks in Buffered Send and Receive Operations:*
- While buffering relieves many of the deadlock situations, it is still possible to write code that deadlocks.
- This is due to the fact that as in the non-buffered case, receive calls are always blocking (*to ensure semantic consistency*).
- Thus, a simple code fragment such as the following deadlocks since both processes wait to receive data but nobody sends it.

```

1      P0                                P1
2
3  receive(&a, 1, 1);                      receive(&a, 1, 0);
4  send(&b, 1, 1);                          send(&b, 1, 0);

```

- Once again, such circular waits have to be broken.
- However, deadlocks are caused only by waits on receive operations in this case.

### Non-Blocking Message Passing Operations

- In blocking protocols, the *overhead of guaranteeing semantic correctness* was paid in the form of idling (non-buffered) or buffer management (buffered).
- It is possible to require the programmer
  - to ensure semantic correctness,
  - to provide a fast send/receive operation that incurs little overhead.
- This class of **non-blocking protocols** returns from the send or receive operation before it is semantically safe to do so.
- Consequently, the user must be careful not to alter data that may be potentially participating in communication.
- Non-blocking operations are generally accompanied by a check-status operation,
- which indicates whether the semantics of a previously initiated transfer may be violated or not.
- Upon return from a non-blocking operation, the process is free to perform any computation that does not depend upon the completion of the operation.
- Later in the program, the process can check whether or not the non-blocking operation has completed,
- and, if necessary, wait for its completion.
- Non-blocking operations can be buffered or non-buffered.
- In the non-buffered case, a process wishing to send data to another simply posts a pending message and returns to the user program.
- The program can then do other useful work.
- At some point in the future, *when the corresponding receive is posted*, the communication operation is initiated.
- When this operation is completed, the *check-status operation indicates* that it is safe to touch this data.
- This transfer is indicated in Figure 4.3Left.

#### 4.1. PROGRAMMING USING THE MESSAGE-PASSING PARADIGM 59

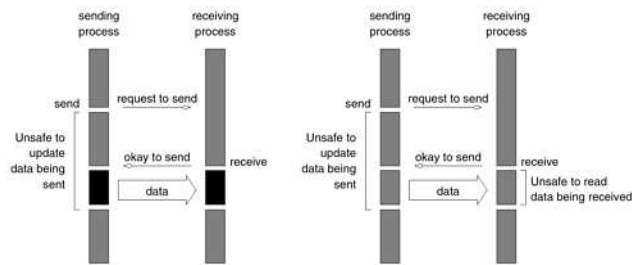


Figure 4.3: Non-blocking non-buffered send and receive operations *Left*: in absence of communication hardware; *Right*: in presence of communication hardware.

- Comparing Figures 4.3Left and 4.1a, it is easy to see that the idling time when the process is waiting for the corresponding receive in a blocking operation can now be utilized for computation.
- This *removes the major bottleneck* associated with the former at the expense of some program restructuring.
- Blocking operations facilitate safe and easier programming.
- Non-blocking operations are useful for performance optimization by masking communication overhead.
- One must, however, be careful using non-blocking protocols since errors can result from unsafe access to data that is in the process of being communicated.

## 4.2 MPI Hands-On; Sending and Receiving Messages II

- The `code3.c` consists of one receiver process and N-1 sender processes.
  - The sender processes send a message consisting of their process identifier (id) and the total number of processes (ntasks) to the receiver.
  - The receiver process prints out the values it receives in the messages from the senders.

```

1  /* A simple SPMD example program using MPI */
2
3  /* The program consists of on receiver process and N-1 sender */
4  /* processes. The sender processes send a message consisting */
5  /* of their process identifier (id) and the total number of */
6  /* processes (ntasks) to the receiver. The receiver process */
7  /* prints out the values it receives in the messages from the */
8  /* senders. */
9
10 /* Compile the program with 'mpicc code3.c -o code3' */
11 /* To run the program, using four of the computers specified in */
12 /* your hostfile, do 'mpirun -machinefile mf.txt -np 4 code3' */
13 /* An example mf.txt is just containing the following lines */
14 /* lecture.ikcu.edu.tr */
15 /* lecture.ikcu.edu.tr */
16 /* lecture.ikcu.edu.tr */
17 /* lecture.ikcu.edu.tr */
18
19 #include <stdio.h>
20 #include <mpi.h>
21 #include <stdlib.h>
22 int main(int argc, char *argv[])
23 {
24     const int tag = 42; /* Message tag */
25     int id, ntasks, source_id, dest_id, err, i;
26     MPI_Status status;
27     int msg[2]; /* Message array */
28
29     err = MPI_Init(&argc, &argv); /* Initialize MPI */
30     if (err != MPI_SUCCESS) {
31         printf("MPI initialization failed!\n");
32         exit(1);
33     }
34     err = MPI_Comm_size(MPLCOMM_WORLD, &ntasks); /* Get nr of tasks */
35     err = MPI_Comm_rank(MPLCOMM_WORLD, &id); /* Get id of this process */
36     if (ntasks < 2) {
37         printf("You have to use at least 2 processors to run this program\n");
38         MPI_Finalize(); /* Quit if there is only one processor */
39         exit(0);
40     }
41
42     if (id == 0) { /* Process 0 (the receiver) does this */
43         for (i=1; i<ntasks; i++) {

```

```

44     err = MPI_Recv(msg, 2, MPI_INT, MPLANY_SOURCE, tag,
45     MPLCOMM_WORLD, &status); /* Receive a message */
46     source_id = status.MPLSOURCE; /* Get id of sender */
47     printf("Received message %d of %d from process %d\n", msg[0],
48     msg[1], source_id);
49 }
50 else { /* Processes 1 to N-1 (the senders) do this */
51     msg[0] = id; /* Put own identifier in the message */
52     msg[1] = ntasks; /* and total number of processes */
53     dest_id = 0; /* Destination address */
54     err = MPI_Send(msg, 2, MPI_INT, dest_id, tag, MPLCOMM_WORLD);
55 }
56 err = MPI_Finalize(); /* Terminate MPI */
57 if (id==0) printf("Ready\n");
58 exit(0);
59 }

```

2. **Sending in a ring.** A [code4.c](#) that takes data from process zero and sends it to all of the other processes by sending it in a ring.

- That is, process  $i$  should receive the data and send it to process  $i+1$ , until the last process is reached.
- Assume that the data consists of a single integer. Process zero reads the data from the user.

```

1 #include <stdio.h>
2 #include "mpi.h"
3
4 int main(int argc, char **argv)
5 {
6     int rank, value, size;
7     MPI_Status status;
8
9     MPI_Init( &argc, &argv );
10
11     MPI_Comm_rank( MPLCOMM_WORLD, &rank );
12     MPI_Comm_size( MPLCOMM_WORLD, &size );
13     do {
14         if (rank == 0) {
15             scanf( "%d", &value );
16             MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPLCOMM_WORLD );
17         }
18         else {
19             MPI_Recv( &value, 1, MPI_INT, rank - 1, 0, MPLCOMM_WORLD, &
20             status );
21             if (rank < size - 1)
22                 MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPLCOMM_WORLD );
23         }
24     } while (value >= 0);
25
26     MPI_Finalize( );
27     return 0;
28 }

```

3. Analyse the example [code5.c](#) for sending/receiving.

```

1  /*
   * *****
2  * FILE: mpl.ex1.c
3  * DESCRIPTION:
4  *   In this simple example, the master task initiates numtasks-1
   *   number of
5  *   worker tasks. It then distributes an equal portion of an array
   *   to each
6  *   worker task. Each worker task receives its portion of the array,
   *   and
7  *   performs a simple value assignment to each of its elements. The
   *   value
8  *   assigned to each element is simply that element's index in the
   *   array+1.
9  *   Each worker task then sends its portion of the array back to the
10 *   master
   *   task. As the master receives back each portion of the array,
   *   selected
11 *   elements are displayed.
12 * AUTHOR: Blaise Barney
13 * LAST REVISED: 09/14/93 for latest API changes   Blaise Barney
14 * LAST REVISED: 01/10/94 changed API to MPL      Stacy Pendell
15 * CONVERTED TO MPI: 11/12/94 by                   Xianneng Shen
16 * *****
   */
17
18 #include <stdio.h>
19 #include "mpi.h"
20 #define ARRAYSIZE      60000
21 #define MASTER        0      /* taskid of first process */
22
23 MPI_Status status;
24 main(int argc, char **argv)
25 {
26     int    numtasks,          /* total number of MPI process in
   *   partiitiion */
27     numworkers,             /* number of worker tasks */
28     taskid,                 /* task identifier */
29     dest,                   /* destination task id to send message
   *   */
30     index,                  /* index into the array */
31     i,                      /* loop variable */
32     arraymsg = 1,          /* setting a message type */
33     indexmsg = 2,          /* setting a message type */
34     source,                 /* origin task id of message */
35     chunksize;             /* for partitioning the array */
36     float data[ARRAYSIZE], /* the intial array */
37     result[ARRAYSIZE];     /* for holding results of array operations */
38
39     /***** initializations
   * *****
40     * Find out how many tasks are in this partition and what my task id
   *   is. Then
41     * define the number of worker tasks and the array partition size as
   *   chunksize.
42     * Note: For this example, the MP_PROCS environment variable should
   *   be set
43     * to an odd number...to insure even distribution of the array to

```

```

44     numtasks-1
45     * worker tasks .
46
47     /*
48     *****
49     */
50     MPI_Init(&argc , &argv);
51     MPI_Comm_rank(MPLCOMM_WORLD, &taskid);
52     MPI_Comm_size(MPLCOMM_WORLD, &numtasks);
53     numworkers = numtasks-1;
54     chunksize = (ARRAYSIZE / numworkers);
55
56     /****** master task
57     *****/
58     if (taskid == MASTER) {
59         printf("\n***** Starting MPI Example 1 *****\n");
60         printf("MASTER: number of worker tasks will be= %d\n", numworkers);
61         fflush(stdout);
62
63         /* Initialize the array */
64         for(i=0; i<ARRAYSIZE; i++)
65             data[i] = 0.0;
66         index = 0;
67
68         /* Send each worker task its portion of the array */
69         for (dest=1; dest<= numworkers; dest++) {
70             printf("Sending to worker task= %d\n", dest);
71             fflush(stdout);
72             MPI_Send(&index, 1, MPL_INT, dest, 0, MPLCOMM_WORLD);
73             MPI_Send(&data[index], chunksize, MPL_FLOAT, dest, 0,
74             MPLCOMM_WORLD);
75             index = index + chunksize;
76         }
77
78         /* Now wait to receive back the results from each worker task and
79         print */
80         /* a few sample values */
81         for (i=1; i<= numworkers; i++) {
82             source = i;
83             MPI_Recv(&index, 1, MPL_INT, source, 1, MPLCOMM_WORLD, &status)
84             ;
85             MPI_Recv(&result[index], chunksize, MPL_FLOAT, source, 1,
86             MPLCOMM_WORLD,
87             &status);
88
89             printf("-----\n");
90             printf("MASTER: Sample results from worker task = %d\n", source);
91             printf("    result [%d]=%f\n", index, result[index]);
92             printf("    result [%d]=%f\n", index+100, result[index+100]);
93             printf("    result [%d]=%f\n\n", index+1000, result[index+1000]);
94             fflush(stdout);
95         }
96
97         printf("MASTER: All Done! \n");
98     }
99
100     /****** worker task
101     *****/
102     if (taskid > MASTER) {
103         /* Receive my portion of array from the master task */
104         source = MASTER;
105         MPI_Recv(&index, 1, MPL_INT, source, 0, MPLCOMM_WORLD, &status);

```

```
97     MPI_Recv(&result[index], chunksize, MPI_FLOAT, source, 0,  
98             MPLCOMM_WORLD, &status);  
99     /* Do a simple value assignment to each of my array elements */  
100    for(i=index; i < index + chunksize; i++)  
101        result[i] = i + 1;  
102  
103    /* Send my results back to the master task */  
104  
105    MPI_Send(&index, 1, MPI_INT, MASTER, 1, MPLCOMM_WORLD);  
106    MPI_Send(&result[index], chunksize, MPI_FLOAT, MASTER, 1,  
107            MPLCOMM_WORLD);  
108    }  
109    MPI_Finalize();  
110 }
```



## 4.3 MPI: the Message Passing Interface

- Many early generation commercial parallel computers were based on the message-passing architecture due to its lower cost relative to shared-address-space architectures.
- Message-passing became the modern-age form of assembly language, in which every hardware vendor provided its own library.
- Performed very well on its own hardware, but was incompatible with the parallel computers offered by other vendors.
- Many of the differences between the various vendor-specific message-passing libraries were only syntactic.
- However, often enough there were some *serious semantic differences* that required significant re-engineering to port a message-passing program from one library to another.
- **The message-passing interface (MPI) was created to essentially solve this problem.**
- MPI defines
  - a standard library for message-passing,
  - can be used to develop **portable** message-passing programs.
- The MPI standard defines both the syntax as well as the semantics of a core set of library routines.
- The MPI library contains many routines, but the number of key concepts is much smaller.
- In fact, it is possible to write fully-functional message-passing programs by using only six routines (see Table 5.5.1).

### 4.3.1 Starting and Terminating the MPI Library

- **MPI\_Init** is called prior to any calls to other MPI routines.
  - Its purpose is to initialize the mpi environment.
  - Calling **MPI\_Init** more than once during the execution of a program will lead to an error.

Table 4.1: The minimal set of MPI routines.

<code>MPI_Init</code>	Initializes MPI
<code>MPI_Finalize</code>	Terminates MPI
<code>MPI_Comm_size</code>	Determines the number of processes
<code>MPI_Comm_rank</code>	Determines the label of the calling process
<code>MPI_Send</code>	Sends a message
<code>MPI_Recv</code>	Receives a message

- **MPI\_Finalize** is called at the end of the computation.
  - It performs various clean-up tasks to terminate the MPI environment.
  - No MPI calls may be performed after **MPI\_Finalize** has been called, not even **MPI\_Init**.
- Upon successful execution, **MPI\_Init** and **MPI\_Finalize** return `MPI_SUCCESS`; otherwise they return an implementation-defined error code.

### 4.3.2 Communicators

- A key concept used throughout MPI is that of the communication domain.
- A communication domain is a set of processes that are allowed to communicate with each other.
- Information about communication domains is stored in variables of type `MPI_Comm`, that are called communicators.
- These communicators are used as arguments to all message transfer MPI routines.
- They uniquely identify the processes participating in the message transfer operation.
- **In general, all the processes may need to communicate with each other.**
- For this reason, MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes involved.

### 4.3.3 Getting Information

- `MPI_Comm_size` function  $\implies$  number of processes
- `MPI_Comm_rank` function  $\implies$  label of the calling process
- The calling sequences of these routines are as follows:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- The function `MPI_Comm_size` returns in the variable `size` the number of processes that belong to the communicator *comm*.
- Every process that belongs to a communicator is uniquely identified by its *rank*.
- The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.
- Up on return, the variable *rank* stores the rank of the process.

### 4.3.4 Sending and Receiving Messages

- The basic functions for sending and receiving messages in MPI are the `MPI_Send` and `MPI_Recv`, respectively.
- The calling sequences of these routines are as follows:

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

1 `MPI_Send` sends the data stored in the buffer pointed by *buf*.

- This buffer consists of consecutive entries of the type specified by the parameter `datatype`.

Table 4.2: Correspondence between the datatypes supported by MPI and those supported by C.

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
MPI_BYTE	
MPI_PACKED	

- The number of entries in the buffer is given by the parameter *count*.

Note that for all C datatypes, an equivalent MPI datatype is provided.

- MPI allows two additional datatypes that are not part of the C language.
- These are *MPI\_BYTE* and *MPI\_PACKED*.
  - *MPI\_BYTE* corresponds to a byte (8 bits)
  - *MPI\_PACKED* corresponds to a collection of data items that has been created by packing non-contiguous data.
- Note that the length of the message in **MPI\_Send**, as well as in other MPI routines, is specified *in terms of the number of entries* being sent and *not in terms of the number of bytes*.
- The destination of the message sent by **MPI\_Send** is uniquely specified by
  - *dest* argument. This argument is the *rank* of the destination process in the communication domain specified by the communicator *comm*.

- comm argument.
  - Each message has an integer-valued tag associated with it.
  - This is used to **distinguish** different types of messages.
- 2 **MPI\_Recv** receives a message sent by a process whose *rank* is given by the *source* in the communication domain specified by the *comm* argument.
- The *tag* of the sent message must be that specified by the tag argument.
  - If there are many messages with identical tag from the same process, then **any one of** these messages is received.
  - MPI allows specification of wild card 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.
    - Similarly, if tag is set to *MPI\_ANY\_TAG*, then messages with any tag are accepted.
  - The received message is stored in continuous locations in the buffer pointed to by *buf*.
  - The *count* and *datatype* arguments of **MPI\_Recv** are used to specify the length of the supplied buffer.
  - The received message should be of length equal to or less than this length.
  - If the received message is larger than the supplied buffer, then an overflow error will occur, and the routine will return the error *MPI\_ERR\_TRUNCATE*.
  - After a message has been received, the status variable can be used to get information about the **MPI\_Recv** operation.
  - In C, status is stored using the *MPI\_Status* data-structure.
  - This is implemented as a structure with three fields, as follows:

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

- *MPI\_SOURCE* and *MPI\_TAG* store the source and the tag of the received message.
- They are particularly useful when *MPI\_ANY\_SOURCE* and *MPI\_ANY\_TAG* are used for the source and tag arguments.
- *MPI\_ERROR* stores the error-code of the received message.
- The status argument also returns information about the length of the received message.
- This information is not directly accessible from the status variable, but it can be retrieved by calling the **MPI\_Get\_count** function.
- The calling sequence:

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

- **MPI\_Get\_count** takes as arguments the status returned by **MPI\_Recv** and the type of the received data in *datatype*, and returns the number of entries that were actually received in the *count* variable.
- The **MPI\_Recv** returns **only after** the requested message has been **received** and **copied** into the buffer.
- That is, **MPI\_Recv** is a blocking receive operation.
- However, MPI allows two different implementations for **MPI\_Send**.
  - 1 **MPI\_Send** returns only after the corresponding **MPI\_Recv** have been issued and the message has been sent to the receiver.
  - 2 **MPI\_Send** first copies the message into a **buffer** and then returns, without waiting for the corresponding **MPI\_Recv** to be executed.

- MPI programs must be able to run correctly regardless of which of the two methods is used for implementing **MPI\_Send**. Such programs are called safe.
- In writing safe MPI programs, sometimes it is helpful to forget about the alternate implementation of **MPI\_Send** and just think of it as being a blocking send operation.

### 4.3.5 Avoiding Deadlocks

- The semantics of **MPI\_Send** and **MPI\_Recv** place some restrictions on how we can mix and match send and receive operations.
- Consider the following not complete code in which process 0 sends two messages with different tags to process 1, and process 1 receives them in the reverse order.

```

1 int a[10], b[10], myrank;
2 MPI_Status status;
3 ...
4 MPI_Comm_rank(MPLCOMM_WORLD, &myrank);
5 if (myrank == 0) {
6     MPI_Send(a, 10, MPI_INT, 1, 1, MPLCOMM_WORLD);
7     MPI_Send(b, 10, MPI_INT, 1, 2, MPLCOMM_WORLD);
8 }
9 else if (myrank == 1) {
10    MPI_Recv(b, 10, MPI_INT, 0, 2, MPLCOMM_WORLD);
11    MPI_Recv(a, 10, MPI_INT, 0, 1, MPLCOMM_WORLD);
12 }
13 ...

```

- If **MPI\_Send** is implemented using buffering, then this code will run correctly (if sufficient buffer space is available).
- However, if **MPI\_Send** is implemented by blocking until the matching receive has been issued, then neither of the two processes will be able to proceed.
- This code fragment is not safe, as its behavior is implementation dependent.
- The problem in this program can be corrected by matching the order in which the send and receive operations are issued.
- Similar deadlock situations can also occur when a process sends a message to itself.

- Improper use of **MPI\_Send** and **MPI\_Recv** can also lead to deadlocks in situations when each processor needs to send and receive a message in a circular fashion.
- Consider the following not complete code, in which
  - process  $i$  sends a message to process  $i + 1$  (modulo the number of processes),
  - process  $i$  receives a message from process  $i - 1$  (modulo the number of processes).

```

1 int a[10], b[10], npes, myrank;
2 MPI_Status status;
3 ...
4 MPI_Comm_size(MPLCOMM_WORLD, &npes);
5 MPI_Comm_rank(MPLCOMM_WORLD, &myrank);
6 MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPLCOMM_WORLD);
7 MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPLCOMM_WORLD);
8 ...

```

- When **MPI\_Send** is implemented using buffering, the program will work correctly,
  - since every call to **MPI\_Send** will get buffered, allowing the call of the **MPI\_Recv** to be performed, which will transfer the required data.
- However, if **MPI\_Send** blocks until the matching receive has been issued,
  - all processes will enter an infinite wait state, waiting for the neighbouring process to issue a **MPI\_Recv** operation.
- Note that the deadlock still remains even when we have only two processes.
- Thus, when pairs of processes need to exchange data, the above method leads to an unsafe program.
- The above example can be made safe, by rewriting:

```

1 int a[10], b[10], np, myrank;
2 MPI_Status status;
3 ...
4 MPI_Comm_size(MPLCOMM_WORLD, &np);
5 MPI_Comm_rank(MPLCOMM_WORLD, &myrank);
6 if (myrank%2 == 1) {
7     MPI_Send(a, 10, MPI_INT, (myrank+1)%np, 1, MPLCOMM_WORLD);
8     MPI_Recv(b, 10, MPI_INT, (myrank-1+np)%np, 1, MPLCOMM_WORLD);

```



```

9 }
10 else {
11     MPI_Recv(b, 10, MPI_INT, (myrank-1+np)%np, 1, MPLCOMM_WORLD);
12     MPI_Send(a, 10, MPI_INT, (myrank+1)%np, 1, MPLCOMM_WORLD);
13 }
14 ...

```

- This version partitions the processes into two groups.
- One consists of the *odd-numbered* processes and the other of the *even-numbered* processes.

### 4.3.6 Sending and Receiving Messages Simultaneously

- The above communication pattern appears frequently in many message-passing programs,
- For this reason MPI provides the **MPI\_Sendrecv** function that both sends and receives a message.
- **MPI\_Sendrecv** does not suffer from the circular deadlock problems of **MPI\_Send** and **MPI\_Recv**.
- You can think of **MPI\_Sendrecv** as allowing data to travel for both send and receive simultaneously.
- The calling sequence of **MPI\_Sendrecv** is as the following:

```

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)

```

- The arguments of **MPI\_Sendrecv** are essentially the combination of the arguments of **MPI\_Send** and **MPI\_Recv**.
- The safe version of our previous example using **MPI\_Sendrecv** is as the following;

```

1 int a[10], b[10], npes, myrank;
2 MPI_Status status;
3 ...
4 MPI_Comm_size(MPLCOMM_WORLD, &npes);
5 MPI_Comm_rank(MPLCOMM_WORLD, &myrank);
6 MPI_SendRecv(a, 10, MPI_INT, (myrank+1)%npes, 1, b, 10, MPI_INT, (
myrank-1+npes)%npes, 1, MPLCOMM_WORLD, &status);
7 ...

```

## 4.4 MPI Hands-On; Sending and Receiving Messages III

1. **Synchronous sending.** MPI example [code6.c](#) using synchronous send. Modify the code and try to see what may cause to deadlock.

```

1  /* A simple MPI example program using synchronous send          */
2
3  /* The program consists of one sender process and one receiver  */
4  /* The sender process sends a message containing its identifier */
5  /* to the receiver. This receives the message and sends it back */
6  /* Both processes use synchronous send operations (MPI_Ssend)  */
7
8  /* Compile the program with 'mpicc -o code6 code6.c'           */
9  /* Run the program with 'mpirun -np 2 code6'                   */
10
11 #include <stdio.h>
12 #include "mpi.h"
13
14 int main(int argc, char* argv[]) {
15     int x, y, np, me;
16     int tag = 42;
17     MPI_Status status;
18
19     MPI_Init(&argc, &argv);          /* Initialize MPI */
20     MPI_Comm_size(MPLCOMM_WORLD, &np); /* Get number of processes */
21     MPI_Comm_rank(MPLCOMM_WORLD, &me); /* Get own identifier */
22
23     x = me;
24     if (me == 0) { /* Process 0 does this */
25         printf("Sending to process 1\n");
26         MPI_Ssend(&x, 1, MPI_INT, 1, tag, MPLCOMM_WORLD); /* Synchronous
27             send */
28         printf("Receiving from process 1\n");
29         MPI_Recv (&y, 1, MPI_INT, 1, tag, MPLCOMM_WORLD, &status);
30         printf("Process %d received a message containing value %d\n", me,
31             y);
32     }
33     else
34     { /* Process 1 does this */
35         /* Since we use synchronous send, we have to do the receive-
36             operation */
37         /* first, otherwise we will get a deadlock */
38         MPI_Recv (&y, 1, MPI_INT, 0, tag, MPLCOMM_WORLD, &status);
39         MPI_Ssend (&x, 1, MPI_INT, 0, tag, MPLCOMM_WORLD); /*
40             Synchronous send */
41     }
42     MPI_Finalize();
43 }

```

2. **Buffered sending.** MPI example [code7.c](#) using buffered send to pass a message between two processes.

```

1  /* A simple MPI example program using buffered send          */
2  /* The program does exactly the same as code6.c             */
3
4  /* The program consists of one sender process and one receiver */
5  /* The sender process sends a message containing its identifier */

```

#### 4.4. MPI HANDS-ON; SENDING AND RECEIVING MESSAGES III 75

```

6  /* to the receiver. This receives the message and sends it back */
7  /* Both processes use buffered send operations (MPI_Bsend) */
8
9  /* Compile the program with 'mpicc -o code7 code7.c' */
10 /* Run the program with 'mpirun -np 2 code7' */
11
12 #include <stdio.h>
13 #include "mpi.h"
14 #include <stdlib.h>
15
16 #define BUFFSIZE 100 /* Size of the message buffer */
17
18 int main(int argc, char* argv[]) {
19     int x, y, np, me;
20     int buff[BUFFSIZE]; /* Buffer to be used in the communication */
21     int size = BUFFSIZE;
22     int tag = 42;
23     MPI_Status status;
24
25     MPI_Init(&argc, &argv); /* Initialize MPI */
26     MPI_Comm_size(MPI_COMM_WORLD, &np); /* Get number of processes */
27     MPI_Comm_rank(MPI_COMM_WORLD, &me); /* Get own identifier */
28
29     MPI_Buffer_attach(buff, size); /* Create a buffer */
30
31     x = me;
32
33     if (me == 0) { /* Process 0 does this */
34         printf("Sending to process 1\n");
35         MPI_Bsend(&x, 1, MPI_INT, 1, tag, MPI_COMM_WORLD); /* Buffered
36         send */
37         printf("Receiving from process 1\n");
38         MPI_Recv (&y, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
39         printf("Process %d received a message containing value %d\n", me,
40         y);
41     }
42     else
43     { /* Process 1 does this */
44         /* This program would work even though we changed the order of
45         */
46         /* the send and receive calls here, because the messages are
47         */
48         /* buffered and the processes can continue the execution without
49         */
50         /* waiting for the other process to receive the message
51         */
52         MPI_Recv (&y, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
53         MPI_Bsend (&x, 1, MPI_INT, 0, tag, MPI_COMM_WORLD); /* Buffered
54         send */
55     }
56     MPI_Buffer_detach(&buff, &size); /* Detach the buffer */
57     MPI_Finalize();
58     exit(0);
59 }

```

### 3. Non-blocking sending I. MPI example [code8.c](#) using non-blocking send and receive to pass a message between two processes.

```

1  /* A simple MPI example program using non-blocking send
   */

```

```

2  /* The program does exactly the same as code6.c
   */
3
4  /* The program consists of one sender process and one receiver
   */
5  /* The sender process sends a message containing its identifier
   */
6  /* to the receiver. This receives the message and sends it back
   */
7  /* Both processes use non-blocking send and receive operations
   */
8  /* (MPI_Isend and MPI_Irecv, and MPI_Wait to wait until the message
   */
9  /* has arrived)
   */
10
11 /* Compile the program with 'mpicc -o code8 code8.c'
   */
12 /* Run the program with 'mpirun -np 2 code8
   */
13
14 #include <stdio.h>
15 #include "mpi.h"
16 #include <stdlib.h>
17
18 int main(int argc, char* argv[]) {
19     int x, y, np, me;
20     int tag = 42;
21     MPI_Status status;
22     MPI_Request send_req, recv_req; /* Request object for send and
   receive */
23
24     MPI_Init(&argc, &argv); /* Initialize MPI */
25     MPI_Comm_size(MPLCOMM_WORLD, &np); /* Get number of processes */
26     MPI_Comm_rank(MPLCOMM_WORLD, &me); /* Get own identifier */
27
28     x = me;
29     if (me == 0) { /* Process 0 does this */
30         printf("Process %d sending\n", me);
31         MPI_Isend(&x, 1, MPI_INT, 1, tag, MPLCOMM_WORLD, &send_req);
32         printf("Process %d receiving\n", me);
33         MPI_Irecv (&y, 1, MPI_INT, 1, tag, MPLCOMM_WORLD, &recv_req);
34         /* We could do computations here while we are waiting for
   communication */
35         MPI_Wait(&send_req, &status);
36         MPI_Wait(&recv_req, &status);
37         printf("Process %d received a message containing value %d\n", me,
   y);
38     }
39     else
40     {
41         MPI_Irecv (&y, 1, MPI_INT, 0, tag, MPLCOMM_WORLD, &recv_req);
42         MPI_Isend (&x, 1, MPI_INT, 0, tag, MPLCOMM_WORLD, &send_req);
43         /* We could do computations here while we are waiting for
   communication */
44         MPI_Wait(&recv_req, &status);
45         MPI_Wait(&send_req, &status);
46     }
47     MPI_Finalize();
48     exit(0);
49 }

```

4. **Non-blocking sending II.** A simple MPI example [code9.c](#) using non-blocking send and receive. The sender process sends a message to all other processes. They receive the message and send an answer back.

```

1  /* processes. The sender process sends a message containing its
   */
2  /* identifier to all the other processes. These receive the message
   */
3  /* and replies with a message containing their own identifier
   */
4  /* Both processes use non-blocking send and receive operations
   */
5  /* (MPI_Isend and MPI_Irecv, and MPI_Waitall)
   */
6
7  /* Compile the program with 'mpicc -o code9 code9.c'
   */
8  /* Run the program with 'mpirun -np 4 code9
   */
9
10 #include <stdio.h>
11 #include "mpi.h"
12 #include <stdlib.h>
13
14 #define MAXPROC 8    /* Max number of processes */
15
16 int main(int argc, char* argv[]) {
17     int i, x, np, me;
18     int tag = 42;
19
20     MPI_Status status[MAXPROC];
21     /* Request objects for non-blocking send and receive */
22     MPI_Request send_req[MAXPROC], recv_req[MAXPROC];
23     int y[MAXPROC]; /* Array to receive values in */
24
25     MPI_Init(&argc, &argv); /* Initialize */
26     MPI_Comm_size(MPLCOMMLWORLD, &np); /* Get nr of processes */
27     MPI_Comm_rank(MPLCOMMLWORLD, &me); /* Get own identifier */
28
29     x = me; /* This is the value we send, the process id */
30     if (me == 0) { /* Process 0 does this */
31         /* First check that we have at least 2 and at most MAXPROC
           processes */
32         if (np < 2 || np > MAXPROC) {
33             printf("You have to use at least 2 and at most %d processes\n",
34                   MAXPROC);
35             MPI_Finalize();
36             exit(0);
37         }
38         printf("Process %d sending to all other processes\n", me);
39         /* Send a message containing the process id to all other processes
           */
40         for (i = 1; i < np; i++) {
41             MPI_Isend(&x, 1, MPI_INT, i, tag, MPLCOMMLWORLD, &send_req[i]);
42         }
43         /* While the messages are delivered, we could do computations here
           */
44         /* Wait until all messages have been sent */
45         /* Note that we use requests and statuses starting from position 1
           */
46         MPI_Waitall(np - 1, &send_req[1], &status[1]);

```

```
46     printf("Process %d receiving from all other processes\n", me);
47     /* Receive a message from all other processes */
48     for (i=1; i<np; i++) {
49         MPI_Irecv (&y[i], 1, MPI_INT, i, tag, MPLCOMM_WORLD, &recv_req [
50             i]);
51     }
52     /* While the messages are delivered, we could do computations here
53        */
54     /* Wait until all messages have been received */
55     /* Requests and statuses start from position 1 */
56     MPI_Waitall(np-1, &recv_req[1], &status[1]);
57
58     /* Print out one line for each message we received */
59     for (i=1; i<np; i++) {
60         printf("Process %d received message from process %d\n", me, y[i
61             ]);
62     }
63     printf("Process %d ready\n", me);
64 }
65 else
66 { /* all other processes do this */
67
68     /* Check sanity of the user */
69     if (np<2 || np>MAXPROC) {
70         MPI_Finalize();
71         exit(0);
72     }
73     MPI_Irecv (&y, 1, MPI_INT, 0, tag, MPLCOMM_WORLD, &recv_req[0]);
74     MPI_Wait(&recv_req[0], &status[0]);
75     MPI_Isend (&x, 1, MPI_INT, 0, tag, MPLCOMM_WORLD, &send_req[0]);
76     /* Lots of computations here */
77     MPI_Wait(&send_req[0], &status[0]);
78 }
79
80 MPI_Finalize();
81 exit(0);
82 }
```

## 4.5 Parallelization Application Example

### 4.5.1 Pi Computation

- $\pi$  by numerically evaluating the integral

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

- Midpoint Rule for  $\int_a^b f(x) dx \approx (b-a)f(x_m)$

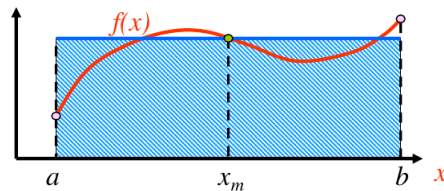


Figure 4.4: Midpoint Rule.

- Midpoint Rule becomes

$$\int_0^1 \frac{1}{1+x^2} dx \approx \sum_{i=1}^n \frac{1}{1 + \left(\frac{i-0.5}{n}\right)^2}$$

#### Sequential Code:

```

1 #include <stdio.h>
2 #include <math.h>
3 int main(int argc, char* argv[])
4 {
5     int done = 0, n, i;
6     double PI25DT = 3.141592653589793238462643;
7     double mypi, h, sum, x;
8     while (!done)
9     {
10        printf("Enter the number of intervals: (0 quits) ");
11        scanf("%d",&n);
12        if (n == 0) break; /* Quit when "0" entered*/
13        /* Integral limits are from 0 to 1 */
14        h = (1.0-0.0)/(double)n; /* Step length*/
15        sum = 0.0; /* Initialize sum variable */
16        /* loop over interval for integration*/
17        for (i = 1; i <= n; i += 1)
18        {
19            x = h * ((double)i - 0.5); /* Middle point at step */
20            sum += 4.0 / (1.0 + x*x); /* Sum up at each step */
21        }
22        printf("i=%d x=%f sum=%f \n",i,x,sum); /* print intermediate steps */
    }

```

```

23     mypi = h * sum; /* Obtain resulting pi number */
24     printf("pi is approximately %.16f, Error is %.16f\n", mypi, \
25           fabs(mypi - PI25DT));
26     }
27 }

```

```

mpicc -o sequential_pi sequential_pi.c
./sequential_pi
Enter the number of intervals: (0 quits) 100
pi is approximately 3.1416009869231254, Error is 0.0000083333333323
Enter the number of intervals: (0 quits) 1000
pi is approximately 3.1415927369231227, Error is 0.0000008333333296
Enter the number of intervals: (0 quits) 10000
pi is approximately 3.1415926544231341, Error is 0.000000008333410
Enter the number of intervals: (0 quits) 0

```

Figure 4.5: Sequential Code Output.

- **Parallel Code:**

- The master process reads number of intervals from standard input, this number is then sent to the processes.
- Having received the number of intervals, each process evaluates the total area of  $n/\text{size}$  rectangles under the curve.
- The contributions to the total area under the curve are collected from participating processes by the master process, which then adds them up, and prints the result on standard output.

```

1 #include <stdio.h>
2 #include <math.h>
3 #include "mpi.h"
4
5 int main(int argc, char* argv[])
6 {
7     int done = 0, n, i;
8     double PI25DT = 3.141592653589793238462643;
9     double mypi, h, sum, x;
10    int size, rank, me;
11    int tag=11;
12    MPI_Status status;
13    double mysum;
14    double pi;
15
16    MPI_Init(&argc, &argv); /* Initialize MPI */
17    MPI_Comm_size(MPLCOMM_WORLD, &size); /* Get number of processes */
18    MPI_Comm_rank(MPLCOMM_WORLD, &rank); /* Get own identifier */
19
20    while (!done)
21    {
22        if (rank == 0) { /* Process 0 does this */
23            printf("Enter the number of intervals: (0 quits) ");
24            scanf("%d",&n);

```



```

25  /* Send a message containing number of intervals to all other processes */
26  for (i=1; i<size; i++) {
27      MPI_Send(&n, 1, MPI_INT, i, tag, MPL_COMM_WORLD); /* Blocking send */
28  }
29  if (n == 0) break; /* Quit when "0" entered */
30  /* Computing local pi number for rank 0 process */
31  /* Integral limits are from 0 to 1 */
32  h = (1.0-0.0)/(double)n; /* Step length */
33  mysum = 0.0; /* Initialize sum variable */
34  for (i = rank+1; i <= n; i += size) /* Loop over interval for integration
    */
35  {
36      x = h * ((double)i - 0.5); /* Middle point at step */
37      mysum += 4.0 / (1.0 + x*x); /* Sum up at each step */
38      //printf("i=%d x=%f sum=%f \n", i, x, sum); /* Intermediate steps */
39  }
40  mypi = h * mysum; /* Obtain local resulting pi number */
41  /* Receive a message containing local resulting pi number from all other
    processes */
42  for (i=1; i<size; i++) {
43      MPI_Recv (&pi, 1, MPI_DOUBLE, i, tag, MPL_COMM_WORLD, &status); /*
    Blocking receive */
44      printf("Process 0 : Received local resulting pi number: %.16f from
    process %d \n", pi, i);
45      mypi=mypi+pi; /* Reduce all local values to mypi variable */
46  }
47  printf("pi is approximately %.16f, Error is %.16f\n", mypi, fabs(mypi -
    PI25DT));
48  }
49  else /* Other processes do this */
50  {
51      MPI_Recv (&n, 1, MPI_INT, 0, tag, MPL_COMM_WORLD, &status); /* Blocking
    receive */
52      printf("Process %d : Received number of intervals as %d from process 0 \
    n", rank, n);
53      if (n == 0) break; /* Quit when "0" entered */
54      /* Computing local pi number for other processes */
55      /* Integral limits are from 0 to 1 */
56      h = (1.0-0.0)/(double)n; /* Step length */
57      mysum = 0.0; /* Initialize sum variable */
58      for (i = rank+1; i <= n; i += size) /* Loop over interval for
    integration */
59      {
60          x = h * ((double)i - 0.5); /* Middle point at step */
61          mysum += 4.0 / (1.0 + x*x); /* Sum up at each step */
62          //printf("i=%d x=%f sum=%f \n", i, x, sum); /* Intermediate steps */
63      }
64      mypi = h * mysum; /* Obtain local resulting pi number */
65      /* Send a message containing local resulting pi number to master
    processes */
66      MPI_Send(&mypi, 1, MPI_DOUBLE, 0, tag, MPL_COMM_WORLD); /* Blocking send
    */
67  }
68  }
69  MPI_Finalize();
70 }

```

```

mpicc -o parallel_pi parallel_pi.c
Enter the number of intervals: (0 quits) 100
Process 1 : Received number of intervals as 100 from process 0
Process 2 : Received number of intervals as 100 from process 0
Process 3 : Received number of intervals as 100 from process 0
Process 0 : Received local resulting pi
Process 0 : Received local resulting pi
Process 0 : Received local resulting pi
pi is approximately 3.1416009869231249, Error is 0.00000083333333318
Enter the number of intervals: (0 quits) 1000
Process 2 : Received number of intervals as 1000 from process 0
Process 3 : Received number of intervals as 1000 from process 0
Process 1 : Received number of intervals as 1000 from process 0
Process 0 : Received local resulting pi
Process 0 : Received local resulting pi
Process 0 : Received local resulting pi
pi is approximately 3.1415927369231262, Error is 0.0000000833333331
Enter the number of intervals: (0 quits) 10000
Process 1 : Received number of intervals as 10000 from process 0
Process 2 : Received number of intervals as 10000 from process 0
Process 3 : Received number of intervals as 10000 from process 0
Process 0 : Received local resulting pi
Process 0 : Received local resulting pi
Process 0 : Received local resulting pi
pi is approximately 3.1415926544231239, Error is 0.0000000008333307
Enter the number of intervals: (0 quits) 0
Process 1 : Received number of intervals as 0 from process 0
Process 2 : Received number of intervals as 0 from process 0
Process 3 : Received number of intervals as 0 from process 0

number: 0.7879260283629755 from process 1
number: 0.7829244650957667 from process 2
number: 0.7778741525634219 from process 3

number: 0.7856484350120356 from process 1
number: 0.7851484334495280 from process 2
number: 0.7846479331370270 from process 3

number: 0.7854231661065627 from process 1
number: 0.7853731661050003 from process 2
number: 0.7853231611046871 from process 3

```

Figure 4.6: Parallel Code Output.

## 4.6 Overlapping Communication with Computation

- The MPI programs we developed so far used blocking send and receive operations whenever they needed to perform point-to-point communication.
- Recall that a **blocking send operation** remains blocked until the message has been copied out of the send buffer
  - either into a system buffer at the source process
  - or sent to the destination process.
- Similarly, a **blocking receive operation** returns only after the message has been received and copied into the receive buffer.
- It will be preferable if we can **overlap the transmission of the data with the computation**.

### 4.6.1 Non-Blocking Communication Operations

- In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations.
  - `MPI_Isend`  $\implies$  starts a send operation but **does not complete**, that is, *it returns before the data is copied out of the buffer*.
  - `MPI_Irecv`  $\implies$  starts a receive operation but **returns before the data has been received and copied into the buffer**.

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

- `MPI_Isend` and `MPI_Irecv` functions **allocate a request object** and return a pointer to it in the request variable.
- **At a later point in the program**, a process that has started a non-blocking send or receive operation **must make sure** that this operation has completed before it proceeds with its computations.
- This is because a process that has started a non-blocking send operation may want to

- overwrite the buffer that stores the data that are being sent,
- or a process that has started a non-blocking receive operation may want to use the data.
- To check the completion of non-blocking send and receive operations, MPI provides a pair of functions
  1. `MPI_Test`  $\implies$  tests whether or not a non-blocking operation has finished
  2. `MPI_Wait`  $\implies$  waits (i.e., gets blocked) until a non-blocking operation actually finishes.

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

- The *request* object is used as an argument in the `MPI_Test` and `MPI_Wait` functions to identify the operation whose status we want to query or to wait for its completion.
- `MPI_Test` tests whether or not the non-blocking send or receive operation identified by its *request* has finished.

True It returns `flag = true` (non-zero value in C) if it is completed.

- The *request* object pointed to by *request* is deallocated and *request* is set to `MPI_REQUEST_NULL`.
- Also the *status* object is set to contain information about the operation.

False It returns `flag = false` (a zero value in C) if it is not completed.

- The *request* is not modified and the value of the *status* object is undefined.
- The `MPI_Wait` function blocks until the non-blocking operation identified by *request* completes.
- A non-blocking communication operation can be matched with a corresponding blocking operation.
- For example, a process can send a message using a non-blocking send operation and this message can be received by the other process using a blocking receive operation.

- *Avoiding Deadlocks*; by using non-blocking communication operations we can remove most of the deadlocks associated with their blocking counterparts.
- For example, the following piece of code is not safe.

```

1 int a[10], b[10], myrank;
2 MPI_Status status;
3 ...
4 MPI_Comm_rank(MPLCOMM_WORLD, &myrank);
5 if (myrank == 0) {
6     MPI_Send(a, 10, MPI_INT, 1, 1, MPLCOMM_WORLD);
7     MPI_Send(b, 10, MPI_INT, 1, 2, MPLCOMM_WORLD);
8 }
9 else if (myrank == 1) {
10    MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPLCOMM_WORLD);
11    MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPLCOMM_WORLD);
12 }
13 ...

```

- However, if we replace either the send or receive operations with their non-blocking counterparts, then the code will be safe, and will correctly run on any MPI implementation.
- Safe with non-blocking communication operations;

```

1 int a[10], b[10], myrank;
2 MPI_Status status;
3 MPI_Request requests[2];
4 ...
5 MPI_Comm_rank(MPLCOMM_WORLD, &myrank);
6 if (myrank == 0) {
7     MPI_Send(a, 10, MPI_INT, 1, 1, MPLCOMM_WORLD);
8     MPI_Send(b, 10, MPI_INT, 1, 2, MPLCOMM_WORLD);
9 }
10 else if (myrank == 1) {
11    MPI_Irecv(b, 10, MPI_INT, 0, 2, &requests[0], MPLCOMM_WORLD);
12    MPI_Irecv(a, 10, MPI_INT, 0, 1, &requests[1], MPLCOMM_WORLD);
13 } //Non-Blocking Communication Operations
14 ...

```

- This example also illustrates that the non-blocking operations started by any process can finish in any order depending on the transmission or reception of the corresponding messages.
- For example, the second receive operation will finish before the first does.

## 4.7 Collective Communication and Computation Operations

- MPI provides an extensive set of functions for performing commonly used collective communication operations.
  - All of the collective communication functions provided by MPI take as an argument a communicator that defines the group of processes that participate in the collective operation.
  - All the processes that belong to this communicator **participate** in the operation,
  - and *all of them* must call the collective communication function.
- Even though collective communication operations do not act like **barriers**,
- act like a virtual synchronization step.
- **Barrier**; the barrier synchronization operation is performed in MPI using the `MPI_Barrier` function.
 

```
int MPI_Barrier(MPIComm comm)
```
- The call to `MPI_Barrier` *returns only after all* the processes in the group have called this function.

### 4.7.1 Broadcast

- **Broadcast**; the one-to-all broadcast operation is performed in MPI using the `MPI_Bcast` function.
 

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPIComm comm)
```
- `MPI_Bcast` sends the data stored in the buffer *buf* of process *source* to all the other processes in the group.
- The data that is broadcast consist of *count* entries of type *datatype*.
- The data received by each process is stored in the buffer *buf*.
- Since the operations are virtually synchronous, they do not require tags.

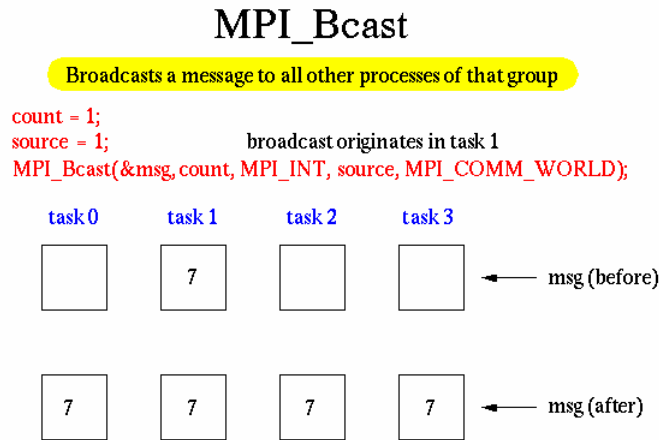


Figure 4.7: Diagram for Broadcast.

#### 4.7.2 Reduction

- **Reduction**; the all-to-one reduction operation is performed in MPI using the `MPI_Reduce` function.

```

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

```

- **combines** the elements stored in the buffer *sendbuf* of each process in the group,
  - using the operation specified in *op*,
  - returns the combined values in the buffer *recvbuf* of the process with rank *target*.
- Both the *sendbuf* and *recvbuf* must have the same number of *count* items of type *datatype*.
  - When *count* is more than one, then the combine operation is applied element-wise on each entry of the sequence.
  - Note that all processes must provide a *recvbuf* array, even if they are not the *target* of the reduction operation.
  - MPI provides a list of **predefined** operations that can be used to combine the elements stored in *sendbuf* (See Table).
  - MPI also allows programmers to define their own operations.

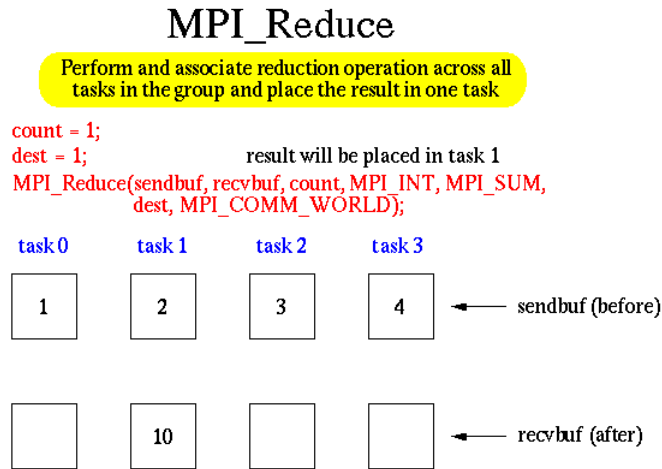


Figure 4.8: Diagram for Reduce.

Table 4.3: Predefined reduction operations.

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-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs

### 4.7.3 Gather

- **Gather**; the all-to-one gather operation is performed in MPI using the `MPI_Gather` function.

```

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

```



#### 4.7. COLLECTIVE COMMUNICATION AND COMPUTATION OPERATIONS 89

- Each process, including the *target* process, sends the data stored in the array *sendbuf* to the *target* process.
- As a result, the *target* process receives a total of  $p$  buffers ( $p$  is the number of processors in the communication *comm*).
- The data is stored in the array *recvbuf* of the target process, in a rank order.
- That is, the data from process with rank  $i$  are stored in the *recvbuf* starting at location  $i * sendcount$  (assuming that the array *recvbuf* is of the same type as *recvdatatype*).
- The data sent by each process must be of the same size and type.
- That is, **MPI\_Gather** must be called with the *sendcount* and *senddatatype* arguments having the same values at each process.
- The information about the receive buffer, its length and type applies only for the target process and is ignored for all the other processes.
- The argument *recvcount* specifies the number of elements received by each process and not the total number of elements it receives.
- So, *recvcount* must be the same as *sendcount* and their datatypes must be matching.
- MPI also provides the **MPI\_Allgather** function in which the data are *gathered to all the processes and not only at the target process*.

```
int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype
    senddatatype, void *recvbuf, int recvcount, MPI_Datatype
    recvdatatype, MPIComm comm)
```

- The meanings of the various parameters are similar to those for **MPI\_Gather**;
- However, each process must now supply a *recvbuf* array that will store the gathered data.

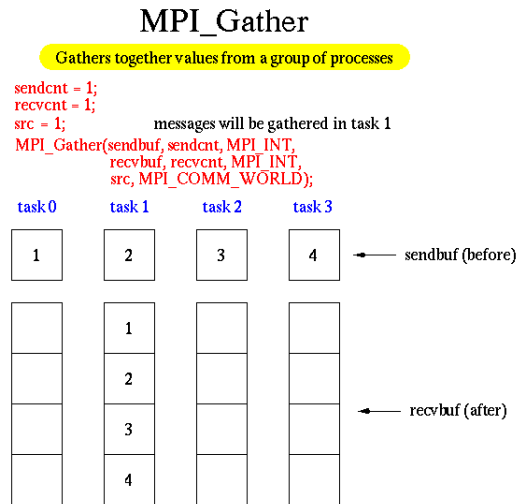


Figure 4.9: Diagram for Gather.

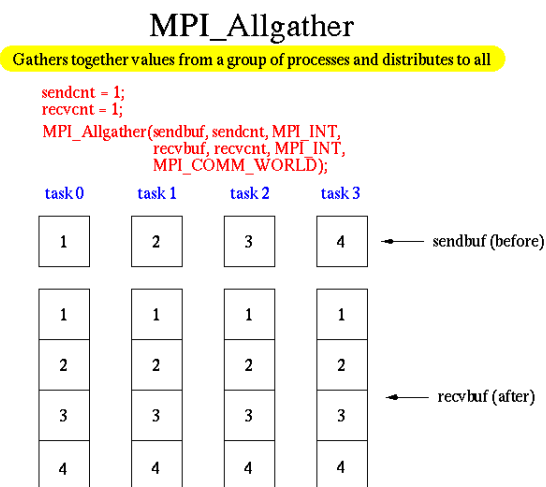


Figure 4.10: Diagram for All\_Gather.

#### 4.7.4 Scatter

- **Scatter**; the one-to-all scatter operation is performed in MPI using the `MPI_Scatter` function.

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

- The *source* process sends a different part of the send buffer *sendbuf* to each processes, including itself.
- The data that are received are stored in *recvbuf*.
- Process *i* receives *sendcount* contiguous elements of type *senddatatype* starting from the  $i * \text{sendcount}$  location of the *sendbuf* of the *source* process (assuming that *sendbuf* is of the same type as *senddatatype*).

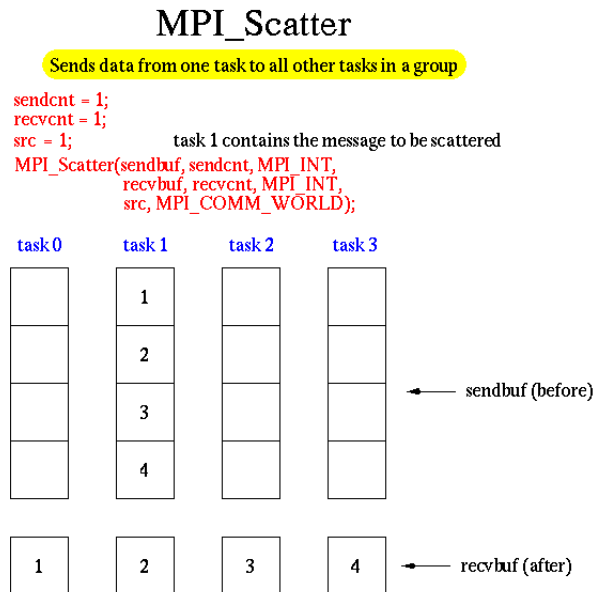


Figure 4.11: Diagram for Scatter.

#### 4.7.5 All-to-All

- **Alltoall**; the all-to-all communication operation is performed in MPI by using the [MPI\\_Alltoall](#) function.

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

- Each process sends a different portion of the *sendbuf* array to each other process, including itself.
- Each process sends to process *i* *sendcount* contiguous elements of type *senddatatype* starting from the  $i * \text{sendcount}$  location of its *sendbuf* array.
- The data that are received are stored in the *recvbuf* array.
- Each process receives from process *i* *recvcount* elements of type *recvdatatype* and stores them in its *recvbuf* array starting at location  $i * \text{recvcount}$ .

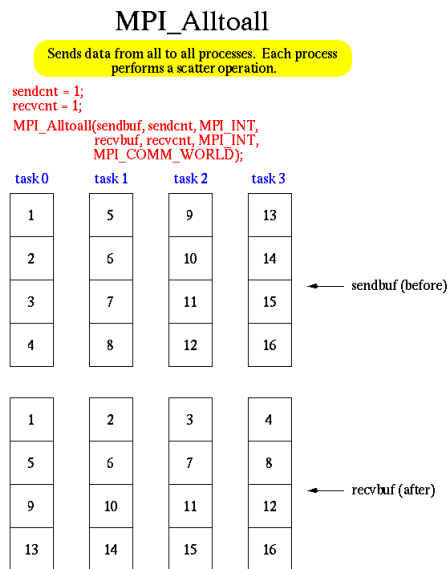


Figure 4.12: Diagram for Alltoall.

## 4.8 MPI Hands-On; Collective Communications I

1. **Broadcasting an integer value to all of the MPI processes**, A program `code10.c` that reads an integer value from the terminal and distributes the value to all of the MPI processes.

- Each process should print out its rank and the value it received. Values should be read until a negative integer is given as input.
- You may find it helpful to include a `fflush( stdout )` to the code; after the `printf` calls in your program. Without this, output may not appear when you expect it.

```
1 #include <stdio.h>
2 #include "mpi.h"
3
4 int main( int argc, char **argv )
5 {
6     int rank, value;
7     MPI_Init( &argc, &argv );
8
9     MPI_Comm_rank( MPLCOMM_WORLD, &rank );
10    do {
11        if (rank == 0)
12            scanf( "%d", &value );
13
14        MPI_Bcast( &value, 1, MPI_INT, 0, MPLCOMM_WORLD );
15
16        printf( "Process %d got %d\n", rank, value );
17        fflush( stdout );
18
19    } while (value >= 0);
20
21    MPI_Finalize( );
22    return 0;
23 }
```

2. **Broadcasting the name of the master process**, A program `code11.c` that first broadcasts the name of the master process then each nodes send hello messages to master node.

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <mpi.h>
4
5 #define TRUE 1
6 #define FALSE 0
7 #define MASTER_RANK 0
8
9 int main( int argc, char *argv[] )
10 {
11     int count, pool_size, my_rank, my_name_length, i_am_the_master =
        FALSE;
12     char my_name[BUFSIZ/2], master_name[BUFSIZ/2], send_buffer[2*BUFSIZ
        ],
13         recv_buffer[2*BUFSIZ];
14     MPI_Status status;
15
16     MPI_Init(&argc, &argv);
17     MPI_Comm_size(MPLCOMM_WORLD, &pool_size);
18     MPI_Comm_rank(MPLCOMM_WORLD, &my_rank);
19     MPI_Get_processor_name(my_name, &my_name_length);
20
21     if (my_rank == MASTER_RANK) {
22         i_am_the_master = TRUE;
23         strcpy (master_name, my_name);
24     }
25
26     MPI_Bcast(master_name, BUFSIZ, MPLCHAR, MASTER_RANK, MPLCOMM_WORLD
        );
27
28     sprintf(send_buffer, "hello %s, greetings from %s, rank = %d",
29             master_name, my_name, my_rank);
30     MPI_Send (send_buffer, strlen(send_buffer) + 1, MPLCHAR,
31              MASTER_RANK, 0, MPLCOMM_WORLD);
32
33     if (i_am_the_master) {
34         for (count = 1; count <= pool_size; count++) {
35             MPI_Recv (recv_buffer, BUFSIZ, MPLCHAR, MPLANY_SOURCE,
36                      MPLANY_TAG,
37                      MPLCOMM_WORLD, &status);
38             printf ("%s\n", recv_buffer);
39         }
40
41     MPI_Finalize ();
42 }

```

### 3. Computation of PI number with collective communications.

This example [code12.c](#) evaluates  $\pi$  by numerically evaluating the integral

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

This code computes PI (with a very simple method) but does not use `MPI_Send` and `MPI_Recv`. Instead, it uses *collective* operations to send data to and from all of the running processes.

- The routine `MPI_Bcast` sends data from one process to all others.
- The routine `MPI_Reduce` combines data from all processes (by adding them in this case), and returning the result to a single process.

```

1 #include <stdio.h>
2 #include "mpi.h"
3 #include <math.h>
4
5 int main( int argc, char *argv[] )
6 {
7     int done = 0, n, myid, numprocs, i, rc;
8     double PI25DT = 3.141592653589793238462643;
9     double mypi, pi, h, sum, x, a;
10
11     MPI_Init(&argc,&argv);
12     MPI_Comm_size(MPLCOMM_WORLD,&numprocs);
13     MPI_Comm_rank(MPLCOMM_WORLD,&myid);
14     while (!done)
15     {
16         if (myid == 0) {
17             printf("Enter the number of intervals: (0 quits) ");
18             scanf("%d",&n);
19         }
20         MPI_Bcast(&n, 1, MPI_INT, 0, MPLCOMM_WORLD);
21         if (n == 0) break;
22
23         h = 1.0 / (double) n;
24         sum = 0.0;
25         for (i = myid + 1; i <= n; i += numprocs) {
26             x = h * ((double) i - 0.5);
27             sum += 4.0 / (1.0 + x*x);
28         }
29         mypi = h * sum;
30
31         MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPLSUM, 0, MPLCOMM_WORLD);
32
33         if (myid == 0)
34             printf("pi is approximately %.16f, Error is %.16f\n",
35                   pi, fabs(pi - PI25DT));
36     }
37     MPI_Finalize();
38 }

```





# Chapter 5

## Shared Memory Paradigm

## 5.1 Programming Shared Memory

### 5.1.1 What is a Thread?

- Technically, a *thread* is defined as an **independent stream of instructions** that can be scheduled to run by the operating system.
  - Suppose that a main program contains a number of procedures (functions, subroutines, ...).
  - Then suppose all of these procedures being able to be scheduled to run simultaneously and/or independently.
  - That would describe a ”**multi-threaded**” program.
- Before understanding a *thread*, one first needs to understand a UNIX *process*.
- Processes contain information about program resources and program execution state.
  - Threads use and exist within these process resources,
  - To be scheduled by the OS,
  - Run as independent entities.
  - A thread has its own independent flow of control as long as its parent process exists (dies if the parent process dies!).
  - A thread duplicates only the essential resources it needs.
- A thread is ”lightweight” because most of the overhead has already been accomplished through the creation of its process.

### 5.1.2 Threads Model

- In shared memory multiprocessor architectures, such as SMPs, *threads can be used to implement parallelism*.
- In the threads model of parallel programming, a single process can have
  - **multiple concurrent**,
  - **execution paths**.
- Most simple analogy for threads is the concept of a single program that includes a number of subroutines:

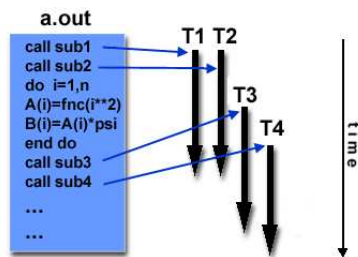


Figure 5.1: Threads model.

- Main program loads and acquires all of the necessary system and user resources to run.
- *Main program* performs some serial work,
- and then creates a number of tasks (threads) that can be scheduled and run by the OS concurrently.

- Each thread has local data, but also, shares the entire resources of *main program*.

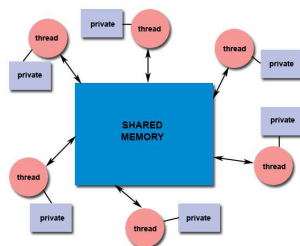


Figure 5.2: Thread shared memory model.

- This saves the overhead associated with replicating a program's resources for each thread.
- Each thread also benefits from a global memory view because it shares the memory space of program.
- Any thread can execute any subroutine at the same time as other threads.
- Threads communicate with each other through global memory (updating address locations).
- Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
- This requires **synchronization constructs** to insure that more than one thread is not updating the same global address at any time.

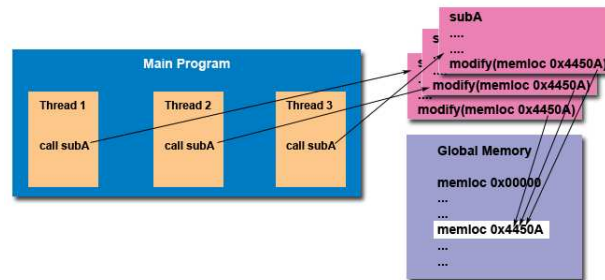


Figure 5.3: Threads **Unsafe!** Pointers having the same value point to the same data.

### 5.1.3 Why Threads?

- The primary motivation for using threads is to realize potential program performance gains.
- When compared to the cost of creating and managing a process, a thread can be created with *much less OS overhead*.
- Managing threads requires fewer system resources than managing processes.
- Threaded programming models offer significant advantages over message-passing programming models along with some disadvantages as well.
- **Software Portability;**
  - Threaded applications can be developed on serial machines and run on parallel machines without any changes.
  - This ability to migrate programs between diverse architectural platforms is a very significant advantage of threaded APIs.
- **Latency Hiding;**
  - One of the major overheads in programs (both serial and parallel) is the access latency for memory access, I/O, and communication.
  - By allowing multiple threads to execute on the same processor, threaded APIs enable this latency to be hidden.
  - In effect, while one thread is waiting for a communication operation, other threads can utilize the CPU, thus *masking associated overhead*.

- **Scheduling and Load Balancing;**
  - While in many *structured* applications the task of allocating equal work to processors is easily accomplished,
  - In *unstructured* and *dynamic* applications (such as game playing and discrete optimization) this task is more difficult.
  - Threaded APIs allow the programmer
    - \* to specify a large number of concurrent tasks
    - \* and support system-level dynamic mapping of tasks to processors with a view to minimizing idling overheads.
- **Ease of Programming, Widespread Use**
  - Due to the mentioned advantages, threaded programs are significantly easier to write (!) than corresponding programs using message passing APIs.
  - With widespread acceptance of the POSIX thread API, development tools for POSIX threads are more widely available and stable.
- **Overlapping CPU work with I/O:** For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.
- **Priority/real-time scheduling:** tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.
- **Asynchronous event handling:** tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.
- A number of vendors provide vendor-specific thread APIs. Standardization efforts have resulted in two very different implementations of threads.
- Microsoft has its own implementation for threads, which is not related to the UNIX POSIX standard or OpenMP.

1. **POSIX Threads.** *Library based; requires parallel coding.*

- C Language only. Very explicit parallelism; requires significant programmer attention to detail.
- Commonly referred to as *Pthreads*.
- POSIX has emerged as the standard threads API, supported by most vendors.

2. **OpenMP**. *Compiler directive based; can use serial code.*

- Jointly defined by a group of major computer hardware and software vendors.
- The OpenMP C/C++ API was released in late 1998.
- Portable / multi-platform, including Unix and Windows platforms
- Can be very easy and simple to use - provides for “incremental parallelism“.
- MPI  $\implies$  on-node communications,
  - MPI libraries usually implement on-node task communication **via shared memory**, which involves at least one memory copy operation (process to process).
- Threads  $\implies$  on-node data transfer.
  - For *Pthreads* there is **no intermediate memory copy** required because threads share the same address space within a single process.
  - There is **no data transfer**.
  - It becomes more of a cache-to-CPU or memory-to-CPU bandwidth (worst case) situation.
  - These speeds are much higher.

### 5.1.4 Thread Basics: Creation and Termination

#### Thread Creation

- The *Pthreads* API subroutines can be informally grouped into **four major groups**:
  1. **Thread management**: Routines that work directly on threads - creating, detaching, joining, set/query thread attributes (joinable, scheduling etc.), etc.

2. **Mutexes:** Routines that deal with synchronization. Mutex functions provide for creating, destroying, locking and unlocking mutexes, setting or modifying attributes associated with mutexes.
3. **Condition variables:** Routines that address communications between threads that share a mutex. Functions to create, destroy, wait and signal based upon specified variable values, set/query condition variable attributes.
4. **Synchronization:** Routines that manage read/write locks and barriers.

### Creating Threads:

- Initially, main program contains a single, default thread.
- **pthread\_create** creates a new thread and makes it executable.

```

1 #include <pthread.h>
2 int
3 pthread_create ( pthread_t *thread_handle ,
4     const pthread_attr_t *attribute ,
5     void * (*thread_function)(void *),
6     void *arg );

```

- Creates a single thread that corresponds to the invocation of the function ***thread\_function*** (and any other functions called by *thread\_function*).
- Once created, threads are peers, and may create other threads.
- On successful creation of a thread, a unique identifier is associated with the thread and assigned to the location pointed to by ***thread\_handle***.
- On successful creation of a thread, **pthread\_create** returns 0; else it returns an error code.
- The thread has the attributes described by the ***attribute*** argument.
- The ***arg*** field specifies a pointer to the argument to function *thread\_function*.
- This argument is typically used to pass the workspace and other *thread-specific data* to a thread.
- There is **no implied hierarchy** or dependency between threads.
- Unless you are using the *Pthreads* scheduling mechanism, it is up to the implementation and/or OS to decide where and when threads will execute.
- Robust programs should not depend upon threads executing in a specific order.

## Thread Termination

### Terminating Threads.

- There are several ways in which a *Pthread* may be terminated:
  - a The thread returns from its starting routine (the main routine for the initial thread).
  - b The thread makes a call to the **pthread\_exit** subroutine.
  - c The thread is cancelled by another thread via the **pthread\_cancel** routine.
  - d The entire process is terminated due to a call to either the *exec* or *exit* subroutines.
    - If *main* finishes before the threads and exits with **pthread\_exit()**, the other threads will continue to execute (join function!).
    - If *main* finishes after the threads and exits, the threads will be automatically terminated.

### Example Code:

- This example code creates 5 threads with the **pthread\_create()** routine.
- Each thread prints a 'Hello World!' message, and then terminates with a call to **pthread\_exit()**.

```

1 #include <pthread.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5
6 #define NUM_THREADS 5
7
8 void *PrintHello(void *threadid)
9 {
10     sleep(10);
11     long tid;
12     tid = (long)threadid;
13     printf("Hello World! It's me, thread #%ld!\n", tid);
14     pthread_exit(NULL);
15 }
16
17 int main(int argc, char *argv[])
18 {
19     pthread_t threads[NUM_THREADS];
20     int rc;
21     long t;

```



```
22     for(t=0;t<NUM_THREADS;t++){
23         printf("In main: creating thread %ld\n", t);
24         rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
25         if (rc){
26             printf("ERROR; return code from pthread_create() is %d\n", rc);
27             exit(-1);
28         }
29     }
30
31     /* Last thing that main() should do */
32     pthread_exit(NULL);
33 }
```

## 5.2 Hands-on; Shared Memory I; Threads

1. **Creation and Termination Threads**, This example [code13.c](#) creates 5 threads with the `pthread_create()` routine. Each thread prints a “Hello World!” message, and then terminates with a call to `pthread_exit()`. Compile as

```
gcc -o code13 code13.c -lpthread /* libpthread as a part of
Unix/Linux operating systems */
./code13
```

```

1  /******
2  * FILE: hello.c
3  * DESCRIPTION:
4  *   A "hello world" Pthreads program. Demonstrates thread creation
5  *   and
6  *   termination.
7  * AUTHOR: Blaise Barney
8  * LAST REVISED: 08/09/11
9  ******
10 #include <pthread.h>
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <unistd.h>
14 #define NUMTHREADS    5
15
16 void *PrintHello(void *threadid)
17 {
18     sleep(10);
19     long tid;
20     tid = (long)threadid;
21     printf("Hello World! It's me, thread #%ld!\n", tid);
22     pthread_exit(NULL);
23 }
24
25 int main(int argc, char *argv[])
26 {
27     pthread_t threads[NUMTHREADS];
28     int rc;
29     long t;
30     for(t=0;t<NUMTHREADS;t++){
31         printf("In main: creating thread %ld\n", t);
32         rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
33         if (rc){
34             printf("ERROR; return code from pthread_create() is %d\n", rc);
35             exit(-1);
36         }
37     }
38
39     /* Last thing that main() should do */
40     pthread_exit(NULL);
41 }
```

2. **Passing Arguments to Threads 1**, This example [code14.c](#) demonstrates how to pass a simple integer to each thread.

```
gcc -o code14 code14.c -lpthread
./code14
```

```

1  /*****
2  * FILE: hello_arg1.c
3  * DESCRIPTION:
4  *   A "hello world" Pthreads program which demonstrates one safe way
5  *   to pass arguments to threads during thread creation.
6  * AUTHOR: Blaise Barney
7  * LAST REVISED: 08/04/15
8  *****/
9  #include <pthread.h>
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <unistd.h>
13
14 #define NUM_THREADS      8
15 char *messages [NUM_THREADS];
16
17 void *PrintHello (void *threadid)
18 {
19     sleep (1);
20     long taskid;
21     taskid = (long) threadid;
22     printf("Thread %ld: %s\n", taskid, messages [taskid]);
23     pthread_exit (NULL);
24 }
25
26 int main (int argc, char *argv [])
27 {
28     pthread_t threads [NUM_THREADS];
29     long taskids [NUM_THREADS];
30     int rc, t;
31
32     messages [0] = "English: Hello World!";
33     messages [1] = "French: Bonjour, le monde!";
34     messages [2] = "Spanish: Hola al mundo";
35     messages [3] = "Klingon: Nuq neH!";
36     messages [4] = "German: Guten Tag, Welt!";
37     messages [5] = "Russian: Zdravstvuyte, mir!";
38     messages [6] = "Japan: Sekai e konnichiwa!";
39     messages [7] = "Latin: Orbis, te saluto!";
40
41     for (t=0;t<NUM_THREADS;t++) {
42         taskids [t] = t;
43         printf("Creating thread %d\n", t);
44         rc = pthread_create(&threads [t], NULL, PrintHello, (void *)
45             taskids [t]);
46         if (rc) {
47             printf("ERROR; return code from pthread_create() is %d\n", rc);
48             exit (-1);
49         }
50     }
51     pthread_exit (NULL);
52 }
```

3. **Passing Arguments to Threads 2**, This example [code15.c](#) shows how to setup/pass multiple arguments via a structure. Each thread receives a unique instance of the structure.

```
gcc -o code15 code15.c -lpthread
./code15
```

```

1  /*****
2  * FILE: hello_arg2.c
3  * DESCRIPTION:
4  * A hello world Pthreads program which demonstrates another safe way
5  * to pass arguments to threads during thread creation. In this case,
6  * a structure is used to pass multiple arguments.
7  * AUTHOR: Blaise Barney
8  * LAST REVISED: 01/29/09
9  *****/
10 #include <pthread.h>
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <unistd.h>
14
15 #define NUM_THREADS      8
16
17 char *messages [NUM_THREADS];
18
19 struct thread_data
20 {
21     int  thread_id;
22     int  sum;
23     char *message;
24 };
25
26 struct thread_data thread_data_array [NUM_THREADS];
27
28 void *PrintHello (void *threadarg)
29 {
30     //    sleep(1);
31     int taskid, sum;
32     char *hello_msg;
33     struct thread_data *my_data;
34
35     sleep(1);
36     my_data = (struct thread_data *) threadarg;
37     taskid = my_data->thread_id;
38     sum = my_data->sum;
39     hello_msg = my_data->message;
40     printf("Thread %d: %s Sum=%d\n", taskid, hello_msg, sum);
41     pthread_exit(NULL);
42 }
43
44 int main(int argc, char *argv[])
45 {
46     pthread_t threads [NUM_THREADS];
47     int *taskids [NUM_THREADS];
48     int rc, t, sum;
49
50     sum=0;
51     messages [0] = "English: Hello World!";

```

```
52 messages[1] = "French: Bonjour, le monde!";
53 messages[2] = "Spanish: Hola al mundo";
54 messages[3] = "Klingon: Nuq neH!";
55 messages[4] = "German: Guten Tag, Welt!";
56 messages[5] = "Russian: Zdravstvyte, mir!";
57 messages[6] = "Japan: Sekai e konnichiwa!";
58 messages[7] = "Latin: Orbis, te saluto!";
59
60 for (t=0;t<NUMTHREADS;t++) {
61     sum = sum + t;
62     thread_data_array[t].thread_id = t;
63     thread_data_array[t].sum = sum;
64     thread_data_array[t].message = messages[t];
65     printf("Creating thread %d\n", t);
66     rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &
        thread_data_array[t]);
67
68     if (rc) {
69         printf("ERROR: return code from pthread_create() is %d\n", rc);
70         exit(-1);
71     }
72 }
73
74 pthread_exit(NULL);
75 }
```

#### 4. Passing Arguments to Threads 3 - Incorrectly, This example `code16.c` performs argument passing incorrectly.

- It passes the address of variable `t`, which is shared memory space and visible to all threads.
- The loop which creates threads modifies the contents of the address passed as an argument, possibly before the created threads can access it.

```
gcc -o code16 code16.c -lpthread
./code16
```

```

1  /*****
2  * FILE: bug3.c
3  * DESCRIPTION:
4  * This "hello world" Pthreads program demonstrates an unsafe (
5  *   incorrect)
6  * way to pass thread arguments at thread creation. Compare with
7  *   hello_arg1.c.
8  * In this case, the argument variable is changed by the main thread
9  *   as it
10 *   creates new threads.
11 * AUTHOR: Blaise Barney
12 * LAST REVISED: 07/16/14
13 *****/
14 #include <pthread.h>
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include <unistd.h>
18
19 #define NUMTHREADS 8
20 void *PrintHello(void *threadid)
21 {
22     // sleep(1);
23     long taskid;
24     taskid = *(long *)threadid;
25     printf("Hello from thread %ld\n", taskid);
26     pthread_exit(NULL);
27 }
28
29 int main(int argc, char *argv[])
30 {
31     pthread_t threads[NUMTHREADS];
32     int rc;
33     long t;
34
35     for(t=0;t<NUMTHREADS;t++) {
36         printf("Creating thread %ld\n", t);
37         rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
38         if (rc) {
39             printf("ERROR; return code from pthread_create() is %d\n", rc);
40             exit(-1);
41         }
42     }
43     pthread_exit(NULL);
44 }

```

5. **Joining Threads**, This example [code17.c](#) demonstrates how to “wait” for thread completions by using the Pthread join routine. Since some implementations of Pthreads may not create threads in a joinable state, the threads in this example are explicitly created in a joinable state so that they can be joined later. Compile as

```
gcc -o code17 code17.c -lpthread -lm
```

```

1  /*****
2  * FILE: join.c
3  * DESCRIPTION:
4  * This example demonstrates how to "wait" for thread completions by
5  * using the Pthread join routine. Threads are explicitly created in
6  * a joinable state for portability reasons. Use of the pthread_exit
7  * status argument is also shown.
8  * AUTHOR: 8/98 Blaise Barney
9  * LAST REVISED: 01/30/09
10 *****/
11 #include <pthread.h>
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <math.h>
15 #include <unistd.h>
16
17 #define NUMTHREADS 4
18 void *BusyWork(void *t)
19 {
20     // sleep(1);
21     int i;
22     long tid;
23     double result=0.0;
24     tid = (long)t;
25     printf("Thread %ld starting...\n",tid);
26     for (i=0; i<1000000; i++)
27     {
28         result = result + sin(i) * tan(i);
29     }
30     printf("Thread %ld done. Result = %e\n",tid, result);
31     pthread_exit((void*) t);
32 }
33
34 int main (int argc, char *argv[])
35 {
36     pthread_t thread[NUMTHREADS];
37     pthread_attr_t attr;
38     int rc;
39     long t;
40     void *status;
41     /* Initialize and set thread detached attribute */
42     pthread_attr_init(&attr);
43     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
44
45     for(t=0; t<NUMTHREADS; t++) {
46         printf("Main: creating thread %ld\n", t);
47         rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
48         if (rc) {
49             printf("ERROR; return code from pthread_create() is %d\n", rc);
50             exit(-1);
51         }
52     }

```

### 5.3 OpenMP: a Standard for Directive Based Parallel Programming

- Although standardization and support for the threaded APIs has a considerable progress, their use is still restricted to **system programmers** as opposed to **application programmers**.
- One of the reasons for this is that APIs such as Pthreads are considered to be **low-level primitives**.
- A large class of applications can be efficiently supported by **higher level constructs (or directives)**
- Which rid the programmer of the mechanics of manipulating threads.
- Such **directive-based languages** have standardization efforts succeeded in the form of OpenMP.
- **OpenMP** is an API that can be used with FORTRAN, C, and C++ for programming shared address space machines.
- Standard API for defining multi-threaded shared-memory programs.
- Allow a programmer to separate a program into serial regions and parallel regions, rather than concurrently-executing threads.
- NOT parallelize automatically and NOT guarantee speedup.
- General structure:

```

1 #include <omp.h>
2 main () {
3   int var1, var2, var3;
4   Serial code
5   Beginning of parallel section. Fork a team of threads
6   Specify variable scoping
7   #pragma omp parallel private(var1, var2) shared(var3)
8   {
9     Parallel section executed by all threads
10    All threads join master thread and disband
11  }
12 Resume serial code
13 }

```



### 5.3.1 The OpenMP Programming Model

- OpenMP *directives* provide support for **concurrency**, **synchronization**, and **data handling** while avoiding the need for explicitly setting up mutexes, condition variables, data scope, and initialization.
- OpenMP directives in C is based on the *#pragma* compiler directives.
- The directive itself consists of a directive name followed by clauses.

```
#pragma omp directive [clause list]
```

- OpenMP programs execute serially until they encounter the parallel directive.
- This directive is responsible for **creating a group of threads**.
- The exact number of threads can be
  - specified in the directive (`num_threads(4)`),
  - set using an environment variable (`export OMP_NUM_THREADS=4` [sh, ksh, bash]),
  - defined at runtime using OpenMP functions (`omp_set_num_threads(4)`).
- The **main** thread that encounters the *parallel* directive becomes the *master* of this group of threads with id 0.
- The *parallel* directive has the following prototype:

```
#pragma omp parallel [clause list]
/* structured block */
```

- Each thread created by this directive executes the structured block specified by the parallel directive (SPMD).

```
1 int main() {
2   omp_set_num_threads(4);
3   // Do this part in parallel
4   #pragma omp parallel
5   {
6     printf( " Hello, World!\n" );
7   }
8   return 0;
```

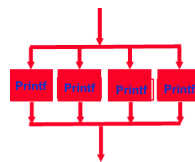


Figure 5.4: Creating four threads for "printf" function.

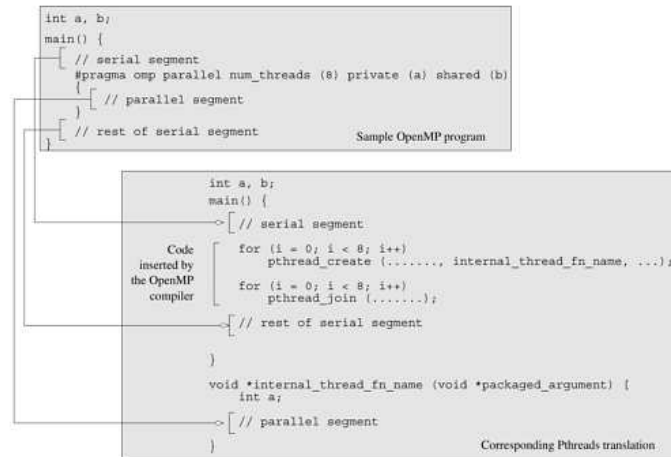


Figure 5.5: A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

- The clause list is used to specify **conditional parallelization**, **number of threads**, and **data handling**.
  - 1 **Conditional Parallelization:** The clause *if (scalar expression)* determines whether the parallel construct results in creation of threads.
  - 2 **Degree of Concurrency:** The clause *num\_threads (integer expression)* specifies the number of threads that are created by the *parallel* directive.
  - 3 **Data Handling:** The clause *private (variable list)* indicates that the set of variables specified is local to each thread.
    - Each thread has its own copy of each variable in the list.
    - The clause *firstprivate (variable list)* is similar to the private clause, except the values of variables on entering the threads are initialized to corresponding values before the parallel directive.
    - The clause *shared (variable list)* indicates that all variables in the list are shared across all the threads,

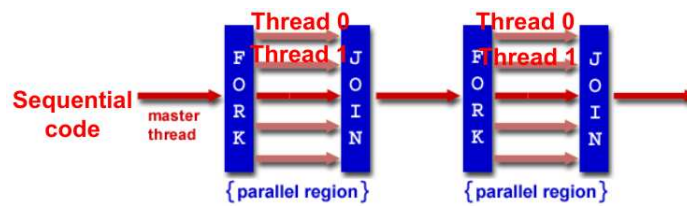


Figure 5.6: Fork-Join Model.

**FORK** Master thread then creates a team of parallel threads.

Statements in program that are enclosed by the parallel region construct are executed in parallel among the various threads.

**JOIN** When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.

**Shared Variables.** OpenMP default is shared variables. To make private, need to declare with pragma:

```

1 #include <stdio.h>
2 #include <omp.h>
3 #include <unistd.h>
4 int a,b,x,y,num_threads,thread_num;
5 int main()
6 {
7     printf("I am in sequential part.\n");
8     #pragma omp parallel num_threads(8) private(a) shared(b)
9     {
10        num_threads=omp_get_num_threads();
11        thread_num=omp_get_thread_num();
12        x=thread_num;
13        // sleep(1);
14        y=x+1;
15        printf("I am openMP parellized part and thread %d. \n X and Y values are
16              %d and %d. \n",omp_get_thread_num(),x,y);
17    }
18    printf("I am in sequential part again.\n");
19 }

```

X and y are shared variables. There is a risk of data race.

<pre> I am in sequential part. I am openMP parellized part and thread 2.   X and Y values are 2 and 3. I am openMP parellized part and thread 4.   X and Y values are 4 and 5. I am openMP parellized part and thread 0.   X and Y values are 0 and 1. I am openMP parellized part and thread 1.   X and Y values are 1 and 2. I am openMP parellized part and thread 7.   X and Y values are 7 and 8. I am openMP parellized part and thread 5.   X and Y values are 5 and 6. I am openMP parellized part and thread 3.   X and Y values are 3 and 4. I am openMP parellized part and thread 6.   X and Y values are 6 and 7. I am in sequential part again. </pre>	<pre> I am in sequential part. I am openMP parellized part and thread 4.   X and Y values are 5 and 6. I am openMP parellized part and thread 3.   X and Y values are 5 and 6. I am openMP parellized part and thread 1.   X and Y values are 5 and 6. I am openMP parellized part and thread 6.   X and Y values are 5 and 6. I am openMP parellized part and thread 2.   X and Y values are 5 and 6. I am openMP parellized part and thread 5.   X and Y values are 5 and 6. I am openMP parellized part and thread 7.   X and Y values are 5 and 6. I am openMP parellized part and thread 0.   X and Y values are 5 and 6. I am in sequential part again. </pre>
--	--

Table 5.1: Correct and Wrong outputs of the program.

Using the *parallel directive*;

```

1 pragma omp parallel if (is_parallel == 1) num_threads(8) private
   (a) shared (b) firstprivate(c)
2 {
3   /* structured block */
4 }

```

- Here, if the value of the variable *is\_parallel* equals one, eight threads are created.
- Each of these threads gets private copies of variables *a* and *c*, and shares a single value of variable *b*.
- Furthermore, the value of each copy of *c* is initialized to the value of *c* before the parallel directive.
- The clause *default (shared)* implies that, by default, a variable is shared by all the threads.
- The clause *default (none)* implies that the state of each variable used in a thread must be explicitly specified.
  - This is generally recommended, to guard against errors arising from unintentional concurrent access to shared data.

The *reduction clause* :

- Specifies how **multiple local copies** of a variable at different threads are **combined into a single copy** at the master when threads exit.

- The usage of the *reduction* clause is *reduction (operator: variable list)*.
  - This clause performs a reduction on the scalar variables specified in the list using the *operator*.
  - The variables in the list are implicitly specified as being private to threads.
- The *operator* can be one of

+ \* - & | ^ && ||

- Each of the eight threads gets a copy of the variable *sum*.

```

1 #pragma omp parallel reduction(+: sum) num_threads(8)
2 {
3     /* compute local sums here */
4 }
5 /* sum here contains sum of all local instances of sums */

```

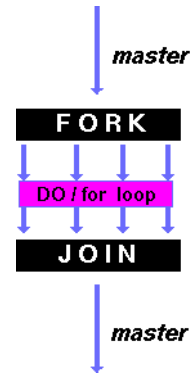
#### Parallel Loop:

- Compiler calculates **loop bounds** for each thread directly from serial source (computation decomposition).
- Compiler also manages data partitioning.
- Synchronization also automatic (barrier).
- Preprocessor calculates loop bounds and divide iterations among parallel threads.

Serial Program:	Parallel Program:
<pre> void main() {     double Res[1000];      for(int i=0;i&lt;1000;i++) {         do_huge_comp(Res[i]);     } } </pre>	<pre> void main() {     double Res[1000];     #pragma omp parallel for     for(int i=0;i&lt;1000;i++) {         do_huge_comp(Res[i]);     } } </pre>

#### Loop Scheduling in Parallel *for* **pragma**

- Master thread creates additional threads, each with a separate execution context.
- All variables declared outside for loop are shared by default, except for loop index which is private per thread.
- Implicit "barrier" synchronization at end of for loop.
- Divide index regions sequentially per thread
  - Thread 0 gets  $0, 1, \dots, (max/n) - 1$
  - Thread 1 gets  $max/n, max/n + 1, \dots, 2 * (max/n) - 1$
  - $\vdots$



Example:

```
#pragma omp parallel for
for (i=0; i<max; i++) zero[i] = 0;
```

- Breaks for loop into chunks, and allocate each to a separate thread.
- if  $max = 1000$  with 2 threads: assign 0-499 to thread 0, and 500-999 to thread 1.

### 5.3.2 The OpenMP Design Concepts

- Load balance, Scheduling overhead, Data locality, Data sharing, Synchronization.
- OpenMP is a compiler-based technique to create concurrent code from (mostly) serial code.
- OpenMP can enable (easy) parallelization of loop-based code with fork-join parallelism.

```
1 pragma omp parallel
2 pragma omp parallel for
3 pragma omp parallel private ( i, x )
4 pragma omp atomic
5 pragma omp critical
6 pragma omp for reduction(+ : sum)
```

### 5.3. OPENMP: A STANDARD FOR DIRECTIVE BASED PARALLEL PROGRAMMING 119

- OpenMP performs comparably to manually-coded threading.

## 5.4 Hands-on; Shared Memory II; OpenMP

### 1. Hello world: [code18.c](#)

- In this example, the master thread forks a parallel region.
- All threads in the team obtain their unique thread number and print it.
- The master thread only prints the total number of threads.
- Two OpenMP library routines are used to obtain the number of threads and each thread's number.

Follow the steps below for executing OpenMP code;

```
export OMP_NUM_THREADS=8
gcc -o code18 code18.c -fopenmp
./code18
```

```

1  /*****
2  * FILE: omp_hello.c
3  * DESCRIPTION:
4  * OpenMP Example – Hello World – C/C++ Version
5  * In this simple example, the master thread forks a parallel region.
6  * All threads in the team obtain their unique thread number and
7  * print it. The master thread only prints the total number of
8  * threads.
9  * Two OpenMP library routines are used to obtain the number of
10 * threads and each thread's number.
11 * AUTHOR: Blaise Barney 5/99
12 * LAST REVISED: 04/06/05
13 *****/
14 #include <omp.h>
15 #include <stdio.h>
16 #include <stdlib.h>
17
18 int main (int argc, char *argv[]) {
19     int nthreads, tid;
20     /* Fork a team of threads giving them their own copies of variables */
21     #pragma omp parallel private(nthreads, tid)
22     {
23         tid = omp_get_thread_num(); /* Obtain thread number */
24         printf("Hello World from thread : %d\n", tid);
25
26         /* Only master thread does this */
27         if (tid == 0)
28         {
29             nthreads = omp_get_num_threads();
30             printf("Number of threads = %d\n", nthreads);
31         }
32     } /* All threads join master thread and disband */
33     return 0;
34 }

```



## 2. Shared Variables: [code19.c](#)

- OpenMP default is shared variables.
- To make private, need to declare with pragma:

Follow the steps below for executing OpenMP code;

```
export OMP_NUM_THREADS=8
gcc -o code19 code19.c -fopenmp
./code19
```

```
1 #include <stdio.h>
2 #include <omp.h>
3 #include <unistd.h>
4
5 int a,b,x,y,num_threads,thread_num;
6 int main()
7 {
8     printf("I am in sequential part.\n");
9     #pragma omp parallel num_threads (8) private (a) shared (b)
10    {
11        num_threads=omp_get_num_threads();
12        thread_num=omp_get_thread_num();
13        x=thread_num;
14        //sleep(1);
15        y=x+1;
16        printf("I am openMP parellized part and thread %d. \n X and Y
17        values are %d and %d. \n",omp_get_thread_num(),x,y);
18    }
19    printf("I am in sequential part again.\n");
20    return 0;
21 }
```

### 3. Loop work-sharing: [code20.c](#)

- The iterations of a loop are scheduled dynamically across the team of threads.
- A thread will perform CHUNK iterations at a time before being scheduled for the next CHUNK of work.

Follow the steps below for executing OpenMP code;

```
gcc -o code20 code20.c -fopenmp
./code20
```

```

1  /*****
2  * FILE: omp_workshare1.c
3  * DESCRIPTION:
4  * OpenMP Example – Loop Work-sharing – C/C++ Version
5  * In this example, the iterations of a loop are scheduled dynamically
6  * across the team of threads. A thread will perform CHUNK iterations
7  * at a time before being scheduled for the next CHUNK of work.
8  * AUTHOR: Blaise Barney 5/99
9  * LAST REVISED: 04/06/05
10 *****/
11 #include <omp.h>
12 #include <stdio.h>
13 #include <stdlib.h>
14 #define CHUNKSIZE 10
15 #define N 100
16
17 int main (int argc, char *argv[]) {
18
19     int nthreads, tid, i, chunk;
20     float a[N], b[N], c[N];
21
22     for (i=0; i < N; i++) /* Some initializations */
23         a[i] = b[i] = i * 1.0;
24     chunk = CHUNKSIZE;
25
26 #pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
27     {
28         tid = omp_get_thread_num();
29         if (tid == 0) {
30             nthreads = omp_get_num_threads();
31             printf("Number of threads = %d\n", nthreads);
32         }
33         printf("Thread %d starting...\n", tid);
34
35         // #pragma omp for schedule(static,chunk)
36 #pragma omp for schedule(dynamic,chunk)
37         for (i=0; i<N; i++) {
38             c[i] = a[i] + b[i];
39             printf("Thread %d: c[%d]= %f\n", tid, i, c[i]);
40         }
41     } /* end of parallel section */
42     return 0;
43 }
```

## 5.5 Parallelization Application Example-OpenMP

### 5.5.1 Computing $\pi$

#### Computing PI using [Sequential Code](#)

An OpenMP version of a threaded program to compute PI number using random numbers.

**Computing PI** using OpenMP directives - [Random Numbers](#). Follow the steps below for executing OpenMP code;

```
gcc -o code21 code21.c -fopenmp
./code21
```

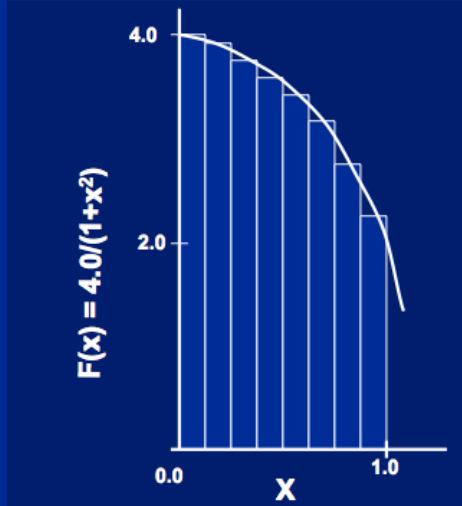
```

1  /*****
2  An OpenMP version of a threaded program to compute PI.
3  *****/
4  #pragma omp parallel default(none) private(rand_no_x, rand_no_y, num_threads
   , sample_points_per_thread) shared(npoints) reduction(+: sum)
   num_threads(8)
5  {
6      num_threads = omp_get_num_threads();
7      sample_points_per_thread = npoints / num_threads;
8
9      sum = 0;
10     for (int i = 0; i < sample_points_per_thread; i++) {
11         rand_no_x = (double)rand() / (double)RAND_MAX;
12         rand_no_y = (double)rand() / (double)RAND_MAX;
13         if (((rand_no_x - 0.5) * (rand_no_x - 0.5) + (rand_no_y - 0.5) *
14             (rand_no_y - 0.5)) < 0.25) {
15             sum = sum + 1;
16         }
17     }

```

- Note that this program is much easier to write in terms of specifying creation and termination of threads compared to the corresponding POSIX threaded program.
- The `omp_get_num_threads()` function returns the number of threads in the parallel region
- The `omp_get_thread_num()` function returns the integer id of each thread (recall that the master thread has an id 0).
- The *parallel directive* specifies that all variables except `npoints`, the total number of random points in two dimensions across all threads, are local.
- Furthermore, the directive specifies that there are eight threads, and the value of `sum` after all threads complete execution is the sum of local values at each thread.

## Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

Where each rectangle has width  $\Delta x$  and height  $F(x_i)$  at the middle of interval  $i$ .

- A for loop generates the required number of random points (in two dimensions) and determines how many of them are within the prescribed circle of unit diameter.

An OpenMP version of a threaded program to compute PI number by numerical integration without reduction clause.

**Computing PI** using OpenMP directives - [Numerical Integration](#). Follow the steps below;

```
export OMP_NUM_THREADS=4
gcc -o code22 code22.c -fopenmp
./code22
```

```
1 #include <stdio.h>
2 #include <math.h>
3 int main(int argc, char* argv[])
4 {
5     int done = 0, n, i;
6     double PI25DT = 3.141592653589793238462643;
7     double mypi, h, sum, x;
8     while (!done)
9     {
10        printf("Enter the number of intervals: (0 quits) ");
11        scanf("%d",&n);
12
13        if (n == 0) break; /* Quit when "0" entered*/
14        /* Integral limits are from 0 to 1 */
15        h = (1.0-0.0)/(double)n; /* Step length*/
16        sum = 0.0; /* Initialize sum variable */
17        for (i = 1; i <= n; i += 1) /* loop over interval for integration*/
```

```

18     {
19         x = h * ((double)i - 0.5); /* Middle point at step */
20         sum += 4.0 / (1.0 + x*x); /* Sum up at each step */
21         //printf(" i=%d x=%f sum=%f \n",i,x,sum); /* print intermediate
steps */
22     }
23     mypi = h * sum; /* Obtain resulting pi number */
24     printf("pi is approximately %.16f, Error is %.16f\n",mypi, fabs(mypi
- PI25DT));
25 }
26 }

```

```

1 #include <stdio.h>
2 #include <math.h>
3 #include <omp.h>
4 #define NUM_THREADS 4
5
6 int main(int argc, char* argv[])
7 {
8     int done = 0, n, i;
9     double PI25DT = 3.141592653589793238462643;
10    double mypi, h, sum[NUM_THREADS], x;
11
12    while (!done)
13    {
14        printf("Enter the number of intervals: (0 quits) ");
15        scanf("%d",&n);
16
17        if (n == 0) break; /* Quit when "0" entered*/
18        /* Integral limits are from 0 to 1 */
19        h = (1.0-0.0)/(double)n; /* Step length*/
20        #pragma omp parallel private ( i, x )
21        {
22            int id = omp_get_thread_num();
23            sum[id]=0.0; /* Initialize sum variable */
24            for (i = id+1; i <= n; i += NUM_THREADS) /* loop over interval for
integration*/
25            {
26                x = h * ((double)i - 0.5); /* Middle point at step */
27                sum[id] += 4.0 / (1.0 + x*x); /* Sum up at each step */
28            }
29        }
30        for(i=1; i<NUM_THREADS; i++)
31            sum[0] += sum[i];
32        mypi = h * sum[0]; /* Obtain resulting pi number */
33        printf("pi is approximately %.16f, Error is %.16f\n",mypi, fabs(mypi - PI25DT));
34    }
35 }

```

Access to [Sequential Code & OpenMP Parallel Code](#) An OpenMP version of a threaded program to compute PI number by numerical integration with reduction clause.

**Computing PI** using OpenMP directives - [Reduction](#). Follow the steps below for executing OpenMP code;

```

export OMP_NUM_THREADS=8
gcc -o code23 code23.c -fopenmp
./code23

```

```

1 #include <stdio.h>
2 #include <math.h>
3 #include <omp.h>
4 int main(int argc, char* argv[])
5 {
6     int done = 0, n, i;
7     double PI25DT = 3.141592653589793238462643;
8     double mypi, h, sum, x;

```

```

gcc -o sequential_pi sequential_pi.c
./sequential_pi
Enter the number of intervals: (0 quits) 100
pi is approximately 3.1416009869231254, Error is 0.0000083333333323
Enter the number of intervals: (0 quits) 1000
pi is approximately 3.1415927369231227, Error is 0.000000833333296
Enter the number of intervals: (0 quits) 10000
pi is approximately 3.1415926544231341, Error is 0.000000083333410
Enter the number of intervals: (0 quits) 0

```

---

```

gcc -o parallel_OpenMP_pi parallel_OpenMP_pi.c -fopenmp
./parallel_OpenMP_pi
Enter the number of intervals: (0 quits) 100
pi is approximately 3.1416009869231249, Error is 0.0000083333333318
Enter the number of intervals: (0 quits) 1000
pi is approximately 3.1415927369231267, Error is 0.000000833333336
Enter the number of intervals: (0 quits) 10000
pi is approximately 3.1415926544231239, Error is 0.000000083333307
Enter the number of intervals: (0 quits) 0

```

Table 5.2: Sequential and OpenMP outputs for computing  $\pi$  number.

```

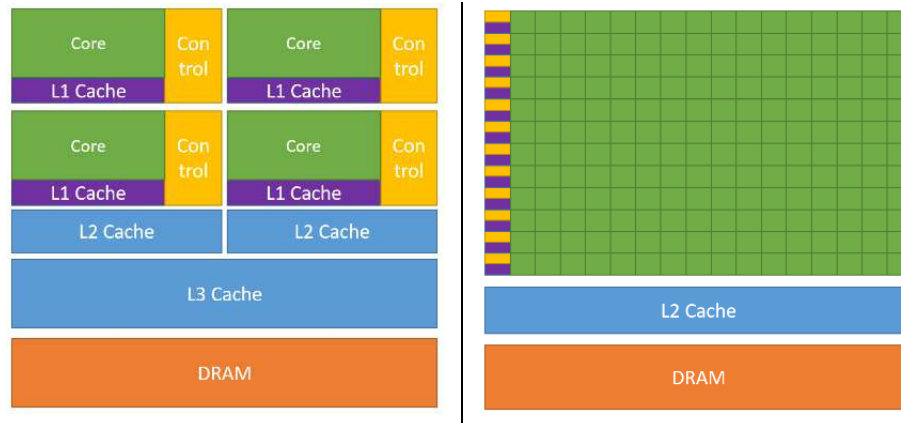
9   while (!done)
10  {
11      printf("Enter the number of intervals: (0 quits) ");
12      scanf("%d",&n);
13      if (n == 0) break; /* Quit when "0" entered*/
14      /* Integral limits are from 0 to 1 */
15      h = (1.0-0.0)/(double)n; /* Step length*/
16      sum = 0.0; /* Initialize sum variable */
17      #pragma omp parallel for private(x) reduction (+:sum)
18      for (i = 1; i <= n; i += 1) /* loop over interval for integration*/
19          {
20              x = h * ((double)i - 0.5); /*Middle point at step*/
21              sum += 4.0 / (1.0 + x*x); /*Sum up at each step*/
22      //printf(" i=%d x=%f sum=%f \n",i,x,sum); /*intermediate steps*/
23          }
24      mypi = h * sum; /* Obtain resulting pi number */
25      printf("pi is approximately %.16f, Error is %.16f\n",mypi, fabs(mypi
26      - PI25DT));
27  }

```

# Chapter 6

## GPU parallelization

## 6.1 Exploring the GPU Architecture



- CPUs are **latency oriented** (minimize execution of serial code).
- GPUs are **throughput oriented** (maximize number of floating point operations).
- If the CPU has  $n$  cores, each core processes  $1/n$  elements.
- GPUs process one element per thread.
- Launching, scheduling threads adds overhead.
- Scheduled by GPU hardware, not by OS.
- A Graphics Processor Unit (GPU) is mostly known for the hardware device used when running applications that weigh heavy on graphics.
- Highly parallel, highly multithreaded multiprocessor optimized for graphic computing and other applications.

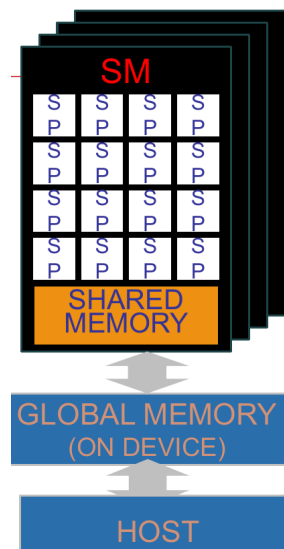
### 1 GPU Programming API: **CUDA (Compute Unified Device Architecture)** : parallel GPU programming API created by NVIDIA

- NVIDIA GPUs can be programmed by CUDA, extension of C language
- API libraries with C/C++/Fortran language
- CUDA C is compiled with `nvcc`
- Numerical libraries: `cuBLAS`, `cuFFT`, `Magma`, ...

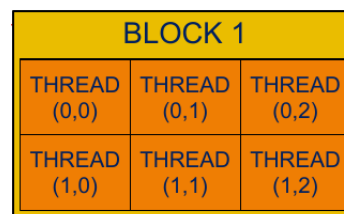


- 2 GPU Programming API: **OpenGL** - an open standard for GPU programming.
  - 3 GPU Programming API: **DirectX** - a series of Microsoft multimedia programming interfaces.
- <https://developer.nvidia.com/> Download: CUDA Toolkit, NVIDIA HPC SDK (Software Development Kit)

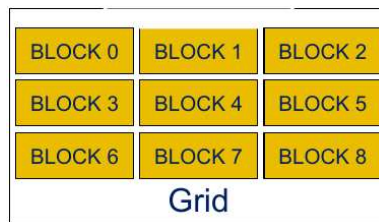
- **SP**: Scalar Processor 'CUDA core'. **Executes one thread.**
- **SM**: Streaming Multiprocessor 32xSP (or 16, 48 or more).
- Fast local 'shared memory' (shared between SPs) 16 KiB (or 64 KiB)
- For example: NVIDIA Maxwell GeForce GTX 750 Ti.
  - 32 SP, 20 SM : 640 CUDA Cores
- **Parallelization**: Decomposition to threads.
- **Memory**: Shared memory, global memory.
- **Thread communication**: Synchronization



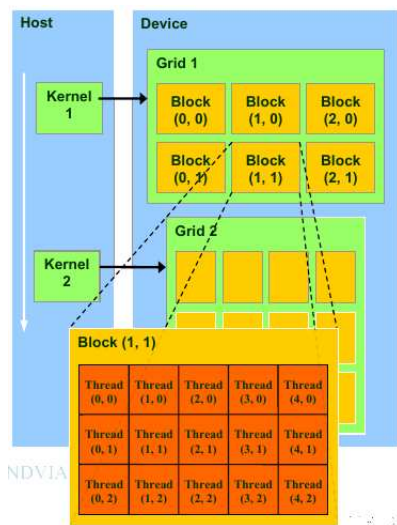
- Threads grouped in thread blocks: 128, 192 or 256 threads in a block
- One thread **block** executes on one **SM**.
  - All threads sharing the 'shared memory'.
  - Each thread block is divided in scheduled units known as a *warp*.



- **Blocks** form a GRID.
- **Thread ID**: unique within **block**.
- **Block ID**: unique within grid.



- A kernel is executed as a grid of thread blocks. All threads share data memory space.
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution.
  - Efficiently sharing data through a low latency shared memory.
- Two threads from two different blocks cannot cooperate.



## 6.2 Execution and Programming Models

- Computation partitioning (where to run)
  - Declarations on functions

```
__host__, __global__, __device__
```

```
__global__ void cuda_hello(){
}
```

- Mapping of thread programs to device:

```
compute <<<gs,bs>>>(<args>)
```

```
cuda_hello <<<blocks_per_grid, threads_per_block>>> ();
```

- Data partitioning (where does data reside, who may access it and how?)

- Declarations on data

```
__shared__, __device__, __constant__, ...
```

```
__device__ const char *STR = "HELLO WORLD!";
```

- Data management and orchestration

- Copying to/from host: e.g.,

```
cudaMemcpy(h_obj, d_obj, cudaMemcpyDeviceToHost)
```

```
cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice );
cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );
```

- Concurrency management. e.g..

```
__syncthreads()
```

```
cudaDeviceSynchronize();
```

## Kernel

- a simple C function
- executes on GPU in parallel as many times as there are threads
- The keyword

```
__global__
```

tells the compiler `nvcc` to make a function a kernel (and compile/run it for the GPU, instead of the CPU)

- It's the functions that you may call from the host side using CUDA kernel call semantics (`<<< ... >>>`).

## Setup and data transfer

- `cudaMemcpy` : Transfer data to and from GPU (global memory)

- `cudaMalloc` : Allocate memory on GPU (global memory)

```

1 double *h_a; // Host input vectors
2 double *d_a; // Device input vectors
3 h_a = (double*) malloc(bytes); // Allocate memory for each vector on
  host
4 cudaMalloc(&d_a, bytes); // Allocate memory for each vector on GPU
5 cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice); // Copy data
  from host array h_a to device arrays d_a
6 add_vectors<<<blk_in_grid, thr_per_blk>>>(d_a, d_b, d_c); // Execute
  the kernel
7 cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost ); // Copy data
  from device array d_c to host array h_c

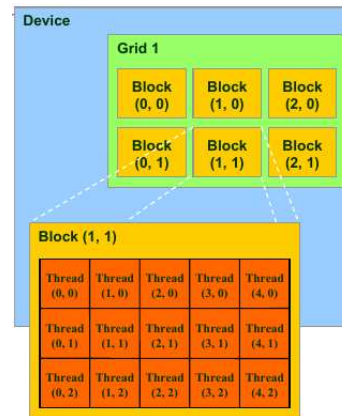
```

- GPU is the 'device', CPU is the 'host'. They do not share memory!
- The HOST launches a kernel that execute on the DEVICE.

- Threads and blocks have IDs

- So each thread can decide what data to work on
- **Block ID**: 1D or 2D (blockIdx.x, blockIdx.y)
- **Thread ID**: 1D, 2D, or 3D (threadIdx.x,y,z)

- Simplifies memory addressing when processing multi-dimensional data.



Courtesy: NDVIA

- Compiler `nvcc` takes as input a `.cu` program and produces
  - C Code for host processor (CPU), compiled by native C compiler
  - Code for device processor (GPU), compiled by `nvcc` compiler

### Cuda Code:

```

1 #include <stdio.h>
2 #include <unistd.h>
3 __device__ const char *STR = "HELLO WORLD!";
4 const int STRLENGTH = 12;
5 __global__ void cuda_hello(){
6 // blockIdx.x: Block index within the grid in x-direction
7 // threadIdx.x: Thread index within the block
8 // blockDim.x: # of threads in a block
9 printf("Hello World from GPU! (%d,%d) : %c ThreadID %d \n", blockIdx.x,
  threadIdx.x, STR[threadIdx.x % STRLENGTH], (threadIdx.x + blockIdx.x *
  blockDim.x));

```

```

10 }
11 int main() {
12     printf("Hello World from CPU!\n");
13     sleep(2);
14     int threads_per_block=12;
15     int blocks_per_grid=2;
16     cuda_hello <<<<blocks_per_grid, threads_per_block>>>> ();
17     cudaDeviceSynchronize(); /* Halt host thread execution on CPU until the
18     device has finished processing all previously requested tasks */
19     return 0;
20 }

```

```

Hello World from CPU!
Hello World from GPU! (1 ,0) : H ThreadID 12
Hello World from GPU! (1 ,1) : E ThreadID 13
Hello World from GPU! (1 ,2) : L ThreadID 14
Hello World from GPU! (1 ,3) : L ThreadID 15
Hello World from GPU! (1 ,4) : O ThreadID 16
Hello World from GPU! (1 ,5) : ThreadID 17
Hello World from GPU! (1 ,6) : W ThreadID 18
Hello World from GPU! (1 ,7) : O ThreadID 19
Hello World from GPU! (1 ,8) : R ThreadID 20
Hello World from GPU! (1 ,9) : L ThreadID 21
Hello World from GPU! (1 ,10) : D ThreadID 22
Hello World from GPU! (1 ,11) : ! ThreadID 23
Hello World from GPU! (0 ,0) : H ThreadID 0
Hello World from GPU! (0 ,1) : E ThreadID 1
Hello World from GPU! (0 ,2) : L ThreadID 2
Hello World from GPU! (0 ,3) : L ThreadID 3
Hello World from GPU! (0 ,4) : O ThreadID 4
Hello World from GPU! (0 ,5) : ThreadID 5
Hello World from GPU! (0 ,6) : W ThreadID 6
Hello World from GPU! (0 ,7) : O ThreadID 7
Hello World from GPU! (0 ,8) : R ThreadID 8
Hello World from GPU! (0 ,9) : L ThreadID 9
Hello World from GPU! (0 ,10) : D ThreadID 10
Hello World from GPU! (0 ,11) : ! ThreadID 11

```

### Cuda Code:

```

1 #include <stdio.h>
2 #include <cuda.h>
3 #include <cuda_runtime.h>
4 #define N 720 // number of computations
5 #define GRID_D1 20 // constants for grid and block sizes
6 #define GRID_D2 3 // constants for grid and block sizes
7 #define BLOCK_D1 12 // constants for grid and block sizes
8 #define BLOCK_D2 1 // constants for grid and block sizes
9 #define BLOCK_D3 1 // constants for grid and block sizes
10
11 --global-- void hello(void) // this is the kernel function called for each thread
12 {
13     // CUDA variables {threadIdx, blockIdx, blockDim, gridDim} to determine a unique thread
14     // ID
15     int myblock = blockIdx.x + blockIdx.y * blockDim.x; // id of the block
16     int blocksize = blockDim.x * blockDim.y * blockDim.z; // size of each block
17     int subthread = threadIdx.z*(blockDim.x * blockDim.y) + threadIdx.y*blockDim.x +
18     threadIdx.x; // id of thread in a given block
19     int idx = myblock * blocksize + subthread; // assign overall id/index of the thread
20     int nthreads=blocksize*gridDim.x*gridDim.y; // Total # of threads
21     int chunk=20; // Vary this value to see the changes at the output
22     if(idx < chunk || idx > nthreads-chunk) { // only print first and last chunks of
23     threads
24         if (idx < N){

```

```

22     printf("Hello world! My block index is (%d,%d) [Grid dims=(%d,%d)], 3D-
thread index within block=(%d,%d,%d) => thread index=%d \n", blockIdx.x, blockIdx.y,
gridDim.x, gridDim.y, threadIdx.x, threadIdx.y, threadIdx.z, idx);
23 }
24 else
25 {
26     printf("Hello world! My block index is (%d,%d) [Grid dims=(%d,%d)], 3D-
thread index within block=(%d,%d,%d) => thread index=%d [### this thread would not
be used for N=%d ###]\n", blockIdx.x, blockIdx.y, gridDim.x, gridDim.y, threadIdx.x,
threadIdx.y, threadIdx.z, idx, N);
27 }
28 }
29 }

30 int main(int argc, char **argv)
31 {
32     // objects containing the block and grid info
33     const dim3 blockSize(BLOCK_D1, BLOCK_D2, BLOCK_D3);
34     const dim3 gridSize(GRID_D1, GRID_D2, 1);
35     int nthreads = BLOCK_D1*BLOCK_D2*BLOCK_D3*GRID_D1*GRID_D2; // Total # of threads
36     if (nthreads < N){
37         printf("\n===== NOT ENOUGH THREADS TO COVER N=%d =====\n\n",N);
38     }
39     else
40     {
41         printf("Launching %d threads (N=%d)\n",nthreads,N);
42     }
43     hello<<<gridSize, blockSize>>>(); // launch the kernel on the specified grid of
thread blocks
44     cudaError_t cudaerr = cudaDeviceSynchronize(); // Need to flush prints, otherwise
none of the prints from within the kernel will show up as program exit does not
flush the print buffer
45     if (cudaerr){
46         printf("kernel launch failed with error \"%s\".\n",
cudaGetErrorString(cudaerr));
47     }
48     else
49     {
50         printf("kernel launch success!\n");
51     }
52     printf("That's all!\n");
53     return 0;
54 }
55 }

```

```

Launching 720 threads (N=720)
Hello world! My block index is (1,0) [Grid dims=(20,3)], 3D-thread index within block=(0,0,0) => thread index=12
Hello world! My block index is (1,0) [Grid dims=(20,3)], 3D-thread index within block=(1,0,0) => thread index=13
Hello world! My block index is (1,0) [Grid dims=(20,3)], 3D-thread index within block=(2,0,0) => thread index=14
Hello world! My block index is (1,0) [Grid dims=(20,3)], 3D-thread index within block=(3,0,0) => thread index=15
Hello world! My block index is (1,0) [Grid dims=(20,3)], 3D-thread index within block=(4,0,0) => thread index=16
Hello world! My block index is (1,0) [Grid dims=(20,3)], 3D-thread index within block=(5,0,0) => thread index=17
Hello world! My block index is (1,0) [Grid dims=(20,3)], 3D-thread index within block=(6,0,0) => thread index=18
Hello world! My block index is (1,0) [Grid dims=(20,3)], 3D-thread index within block=(7,0,0) => thread index=19
Hello world! My block index is (18,2) [Grid dims=(20,3)], 3D-thread index within block=(5,0,0) => thread index=701
Hello world! My block index is (18,2) [Grid dims=(20,3)], 3D-thread index within block=(6,0,0) => thread index=702
Hello world! My block index is (18,2) [Grid dims=(20,3)], 3D-thread index within block=(7,0,0) => thread index=703
Hello world! My block index is (18,2) [Grid dims=(20,3)], 3D-thread index within block=(8,0,0) => thread index=704
Hello world! My block index is (18,2) [Grid dims=(20,3)], 3D-thread index within block=(9,0,0) => thread index=705
Hello world! My block index is (18,2) [Grid dims=(20,3)], 3D-thread index within block=(10,0,0) => thread index=706
Hello world! My block index is (18,2) [Grid dims=(20,3)], 3D-thread index within block=(11,0,0) => thread index=707
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(0,0,0) => thread index=708
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(1,0,0) => thread index=709
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(2,0,0) => thread index=710
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(3,0,0) => thread index=711
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(4,0,0) => thread index=712
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(5,0,0) => thread index=713
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(6,0,0) => thread index=714
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(7,0,0) => thread index=715
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(8,0,0) => thread index=716
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(9,0,0) => thread index=717
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(10,0,0) => thread index=718
Hello world! My block index is (19,2) [Grid dims=(20,3)], 3D-thread index within block=(11,0,0) => thread index=719
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(0,0,0) => thread index=0
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(1,0,0) => thread index=1
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(2,0,0) => thread index=2
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(3,0,0) => thread index=3
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(4,0,0) => thread index=4
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(5,0,0) => thread index=5
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(6,0,0) => thread index=6
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(7,0,0) => thread index=7
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(8,0,0) => thread index=8
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(9,0,0) => thread index=9
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(10,0,0) => thread index=10
Hello world! My block index is (0,0) [Grid dims=(20,3)], 3D-thread index within block=(11,0,0) => thread index=11
kernel launch success!
That's all!

```

## 6.3 Hands-on; GPU parallelization

```
$ lspci | grep -i nvidia
01:00.0 VGA compatible controller: NVIDIA Corporation GP108
[GeForce GT 1030] (rev a1)
01:00.1 Audio device: NVIDIA Corporation GP108 High Definition
Audio Controller (rev a1)
```

Installation Instructions:

```
$ wget https://developer.download.nvidia.com/compute/cuda/repos/
ubuntu2204/x86_64/cuda-keyring_1.0-1_all.deb
$ sudo dpkg -i cuda-keyring_1.0-1_all.deb
$ sudo apt-get update
$ sudo apt-get -y install cuda
$ sudo reboot
$ export PATH=/usr/local/cuda-12.0/bin${PATH:+:${PATH}}
$ export LD_LIBRARY_PATH=/usr/local/cuda-12.0/lib64${LD_LIBRARY_PATH:
+:${LD_LIBRARY_PATH}}
```

1. **Hello world I:** [code24.cu](#) Follow the steps below for executing Cuda code;

```
nvcc -o code24 code24.cu
./code24
```

```
1 #include <stdio.h>
2 #include <unistd.h>
3 __device__ const char *STR = "HELLO WORLD!";
4 const int STRLENGTH = 12;
5 __global__ void cuda_hello(){
6     // blockIdx.x: Block index within the grid in the x direction
7     // threadIdx.x: Thread index within the block
8     // blockDim.x,y,z # of threads in a block
9     printf("Hello World from GPU! (%d,%d) : %c ThreadID %d \n",
10         blockIdx.x, threadIdx.x, STR[threadIdx.x % STRLENGTH], (threadIdx
11         .x +blockIdx.x*blockDim.x));
12 }
13 /*
14 ./deviceQuery Starting...
15
16 CUDA Device Query (Runtime API) version (CUDA static linking)
17
18 Detected 3 CUDA Capable device(s)
19
20 Device 2: "NVIDIA GeForce GT 1030"
21 CUDA Driver Version / Runtime Version          12.0 / 11.8
22 CUDA Capability Major/Minor version number:    6.1
23 Total amount of global memory:                 1998 MBytes
24         (2095185920 bytes)
25 (003) Multiprocessors, (128) CUDA Cores/MP:   384 CUDA Cores
```

```

23 GPU Max Clock rate: 1468 MHz (1.47 GHz)
24 Memory Clock rate: 3004 Mhz
25 Memory Bus Width: 64-bit
26 L2 Cache Size: 524288 bytes
27 Maximum Texture Dimension Size (x,y,z) 1D=(131072), 2D
   =(131072, 65536), 3D=(16384, 16384, 16384)
28 Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048
   layers
29 Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768),
   2048 layers
30 Total amount of constant memory: 65536 bytes
31 Total amount of shared memory per block: 49152 bytes
32 Total shared memory per multiprocessor: 98304 bytes
33 Total number of registers available per block: 65536
34 Warp size: 32
35 Maximum number of threads per multiprocessor: 2048
36 Maximum number of threads per block: 1024
37 Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
38 Max dimension size of a grid size (x,y,z): (2147483647, 65535,
   65535)
39 Maximum memory pitch: 2147483647 bytes
40 Texture alignment: 512 bytes
41 Concurrent copy and kernel execution: Yes with 2 copy
   engine(s)
42 Run time limit on kernels: Yes
43 Integrated GPU sharing Host Memory: No
44 Support host page-locked memory mapping: Yes
45 Alignment requirement for Surfaces: Yes
46 Device has ECC support: Disabled
47 Device supports Unified Addressing (UVA): Yes
48 Device supports Managed Memory: Yes
49 Device supports Compute Preemption: Yes
50 Supports Cooperative Kernel Launch: Yes
51 Supports MultiDevice Co-op Kernel Launch: Yes
52 Device PCI Domain ID / Bus ID / location ID: 0 / 4 / 0
53 Compute Mode:
54 < Default (multiple host threads can use ::cudaSetDevice() with
   device simultaneously) >
55 deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 12.0, CUDA
   Runtime Version = 11.8, NumDevs = 3
56 Result = PASS
57 */
58 int main() {
59     printf("Hello World from CPU!\n");
60     sleep(2);
61     int threads_per_block=12;
62     int blocks_per_grid=2;
63     cuda_hello <<<< blocks_per_grid, threads_per_block >>>> ();
64     cudaDeviceSynchronize(); /* Halt host thread execution on CPU until
   the device has finished processing all previously requested tasks
   */
65     return 0;
66 }

```



2. **Hello world II:** [code25.cu](https://code25.cu) Follow the steps below for executing Cuda code;

```
nvcc -o code25 code25.cu
./code25
```

```

1 #include <stdio.h>
2 #include <cuda.h>
3 #include <cuda_runtime.h>
4
5 #define N 720 // number of computations
6 #define GRID_D1 20 // constants for grid and block sizes
7 #define GRID_D2 3 // constants for grid and block sizes
8 #define BLOCK_D1 12 // constants for grid and block sizes
9 #define BLOCK_D2 1 // constants for grid and block sizes
10 #define BLOCK_D3 1 // constants for grid and block sizes
11
12 __global__ void hello(void) // this is the kernel function called for
    each thread
13 {
14     // we use the CUDA variables {threadIdx, blockIdx, blockDim, gridDim}
    to determine a unique ID for each thread
15     int myblock = blockIdx.x + blockIdx.y * blockDim.x; // id of the
    block
16     int blocksize = blockDim.x * blockDim.y * blockDim.z; // size of
    each block (within grid of blocks)
17     int subthread = threadIdx.z*(blockDim.x * blockDim.y) + threadIdx.y*
    blockDim.x + threadIdx.x; // id of thread in a given block
18     int idx = myblock * blocksize + subthread; // assign overall id/
    index of the thread
19     int nthreads=blocksize*gridDim.x*gridDim.y; // Total # of threads
20     int chunk=20; // Vary this value to see the changes at the output
21     if(idx < chunk || idx > nthreads-chunk) { // print buffer from
    within the kernel is limited so only print for first and last
    chunks of threads
22         if (idx < N){
23             printf("Hello world! My block index is (%d,%d) [Grid dims=(%d,%d)
    ], 3D-thread index within block=(%d,%d,%d) => thread index=%d \n"
    , blockIdx.x, blockIdx.y, blockDim.x, blockDim.y, threadIdx.x,
    threadIdx.y, threadIdx.z, idx);
24         }
25         else
26         {
27             printf("Hello world! My block index is (%d,%d) [Grid dims=(%d,%d)],
    3D-thread index within block=(%d,%d,%d) => thread index=%d [###
    this thread would not be used for N=%d ###]\n", blockIdx.x,
    blockIdx.y, blockDim.x, blockDim.y, threadIdx.x, threadIdx.y,
    threadIdx.z, idx, N);
28         }
29     }
30 }
31
32 int main(int argc, char **argv)
33 {
34     // objects containing the block and grid info
35     const dim3 blockSize(BLOCK_D1, BLOCK_D2, BLOCK_D3);
36     const dim3 gridSize(GRID_D1, GRID_D2, 1);
37     int nthreads = BLOCK_D1*BLOCK_D2*BLOCK_D3*GRID_D1*GRID_D2; // Total
    # of threads

```

```
38  if (nthreads < N){
39      printf("\n===== NOT ENOUGH THREADS TO COVER N=%d\n\n",N);
40  }
41  else
42      {
43      printf("Launching %d threads (N=%d)\n",nthreads ,N);
44      }
45  hello<<<gridSize, blockSize>>>(); // launch the kernel on the
    specified grid of thread blocks
46  // Need to flush prints, otherwise none of the prints from within
    the kernel will show up
47  // as program exit does not flush the print buffer.
48  cudaError_t cudaerr = cudaDeviceSynchronize();
49  if (cudaerr){
50      printf("kernel launch failed with error \"%s\".\n",
51          cudaGetErrorString(cudaerr));
52  }
53  else
54      {
55      printf("kernel launch success!\n");
56      }
57  printf("That's all!\n");
58  return 0;
59 }
```

3. **Vector Addition:** [code26.cu](#) Follow the steps below for executing Cuda code;

```
nvcc -o code26 code26.cu
./code26
```

```

1 // https://www.olcf.ornl.gov/tutorials/cuda-vector-addition/
2 // https://github.com/olcf-tutorials/vector_addition_cuda
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <math.h>
6
7 #define n 100000 // Size of array
8
9 /* CUDA KERNEL Compute the sum of two vectors
10 * Each thread takes care of one element of C
11 * C[i] = A[i] + B[i]
12 */
13 __global__ void add_vectors(double *a, double *b, double *c)
14 {
15     int id = blockIdx.x*blockDim.x+threadIdx.x; // Get our global thread
16     ID
17     if (id < n) // Make sure we do not go out of bounds
18         c[id] = a[id] + b[id]; /* Compute the element of C */
19 }
20 int main( int argc, char* argv[] )
21 {
22     double *h_a; // Host input vectors
23     double *h_b;
24     double *h_c;
25
26     double *d_a; // Device input vectors
27     double *d_b;
28     double *d_c;
29
30     size_t bytes = n*sizeof(double); // Size, in bytes, of each vector
31
32     h_a = (double*) malloc(bytes); // Allocate memory for each vector on
33     host
34     h_b = (double*) malloc(bytes);
35     h_c = (double*) malloc(bytes);
36
37     cudaMalloc(&d_a, bytes); // Allocate memory for each vector on GPU
38     cudaMalloc(&d_b, bytes);
39     cudaMalloc(&d_c, bytes);
40
41     int i;
42     for( i = 0; i < n; i++ ) { // Initialize vectors on host
43         h_a[i] = sin(i)*sin(i);
44         h_b[i] = cos(i)*cos(i);
45     }
46
47     // Copy data from host arrays h_a and h_b to device arrays d_a and
48     d_b
49     cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
50     cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);
51
52     int thr_per_blk = 1024; // blockSize. Number of threads in each
53     thread block

```

```

51  int blk_in_grid = ceil( (float) n/thr_per_blk ); // gridSize. Number
    of thread blocks in grid
52
53  add_vectors<<<<blk_in_grid, thr_per_blk>>>(d_a, d_b, d_c); // Execute
    the kernel
54
55  cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost ); // Copy data
    from device array d_c to host array h_c
56
57  // Sum up vector c and print result divided by n, this should equal
    1 within error
58  double sum = 0;
59  for(i=0; i<n; i++)
60      sum += h_c[i];
61  printf("final result: %f\n", sum/n);
62
63  cudaFree(d_a); // Free device memory
64  cudaFree(d_b);
65  cudaFree(d_c);
66
67  free(h_a); // Free host memory
68  free(h_b);
69  free(h_c);
70
71  printf("-----\n");
72  printf(" _ENDED_ \n");
73  printf("-----\n");
74  printf("N           = %d\n", n);
75  printf("Threads Per Block = %d\n", thr_per_blk);
76  printf("Blocks In Grid   = %d\n", blk_in_grid);
77  printf("-----\n\n");
78
79  return 0;
80 }

```

# Chapter 7

## References:

- <https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial>