C-DAC Four Days Technology Workshop

ON

Hybrid Computing – Co-Processors/Accelerators
Power-aware Computing – Performance of Applications Kernels

hyPACK-2013
(Mode-1:Multi-Core)

Lecture Topic:

Multi-Core Processors : Shared Memory Prog: Pthreads Part-I

Venue : CMSD, UoHYD ; Date : October 15-18, 2013

Shared Memory Programming – Pthreads

Lecture Outline : Following topics will be discussed

- An Overview of Shared Memory Programming
- Shared Memory Programming - PThreads
- System Overview of Threads
- Multi Cores : Threads Parallel Programming

Source : Reference [4],[6], [7], [8]

Explicit Parallelism : Shared Variable Model

- It has a single address space and data resides in single shared address space, thus does not have to be explicitly allocated
- It is multithreading and asynchronous (Similar to message-passing model)
- Workload can be either explicitly or implicitly allocated
- Communication is done implicitly through shared reads and writes of variables. However synchronization is explicit

Defining Threads : What are threads ?

- A thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.
- A thread is a discrete sequence of related instructions that is executed independently of other instructions sequences
- A process can have several threads, each with its own independent flow of control.
- Threads share the resources of the process that created it.

Why Pthreads? : Thread Model

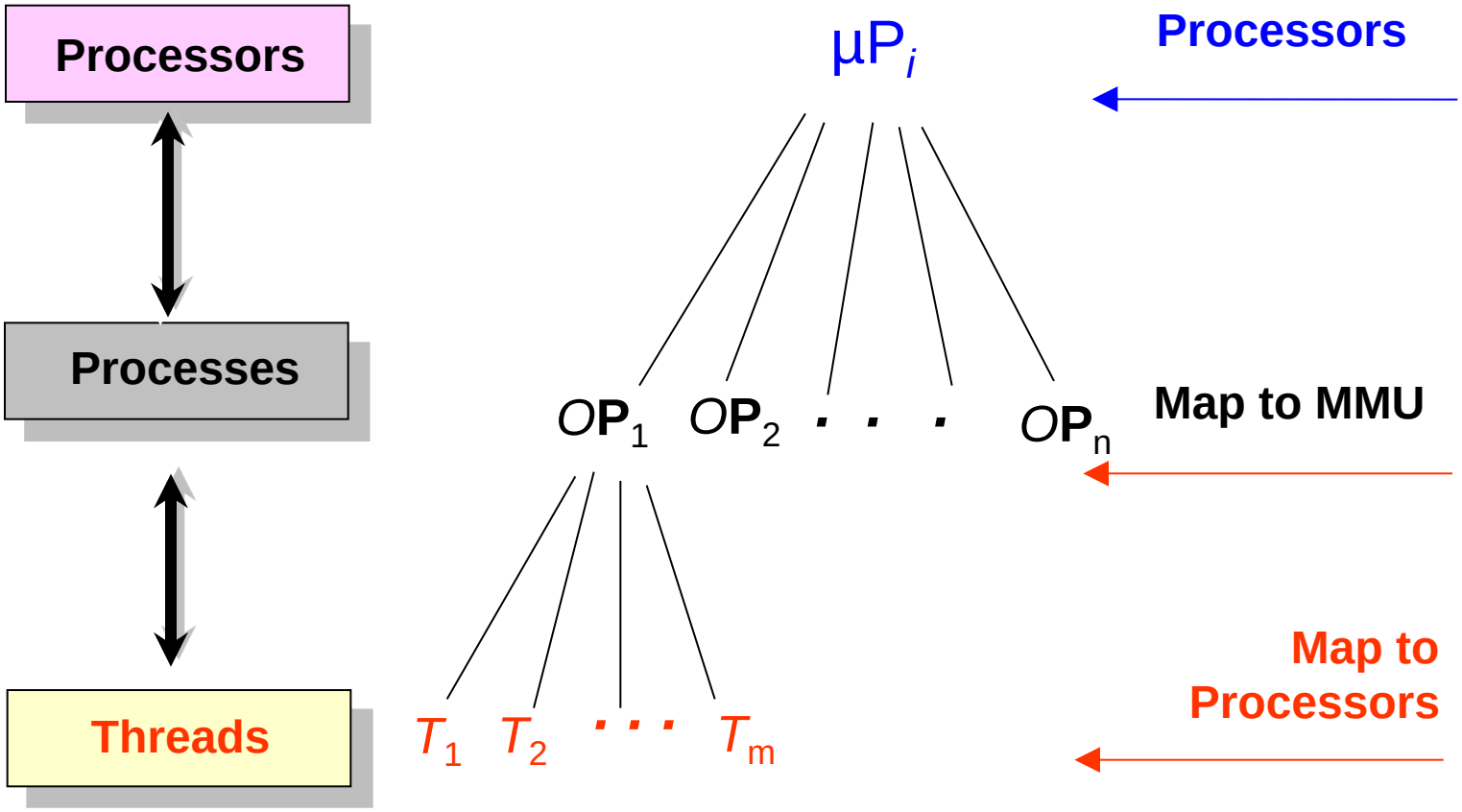
- The thread model takes a process and divide it into two parts
  - One contains resources used across the whole program (the process wide information), such as program instructions and global data. This part is referred to as the *process*.
  - The other contains information related to the execution state, such as program counter, and a stack. This part is referred to as a *thread*.
  - Pthread is a standardized model for dividing a program into subtasks hose execution can be interleaved or run in parallel.
  - The "**P**" in *Pthread* comes form POSIX (Portable Operating System Interface), the family of IEEE operating system interface standards in which *Pthread* is defined. Other thread modes are Mach Threads and NT Threads

Source : Reference [4],[6], [7]

Shared Memory Programming : Threads

- Threads are usually the preferred way to parallelize codes on an SMP
- All threads share a common address space, so communication and synchronization are much faster than possible with either explicit or implicit distributed shared memory (DSM)
- Because all threads are part of the same process, co-coordinating access to resources is very easy and is usually automatic.
- Example
  - All the threads share the resources of the process like files etc. Each thread need not open a separate copy.

Relationship among Processors, Processes, and Threads



μP_i : Processor OP_1 : Process T_1 : Thread **MMU** : Main Memory Unit

Source : Reference [4],[6], [7]

Shared Memory Programming : Threads

(Contd…)

Parallel programming based on shared-memory model has not progressed as much as message-passing model Why?

- Shared memory programs are written in a platform-specific language for multiprocessors (mostly SMP's) such programs are not portable.
- Platform independent shared memory programming models :
  - Pthreads, and Open MP.
  - The SGI power C uses a small set of structured constructs to extend C to a shared memory parallel language.
  - The X3H5 standard has not gained wide acceptance, but has influenced the design of several shared memory programming languages.

Shared Memory Programming : Threads

(Contd…)

- An example of critical region is in a module where threads try to retrieve messages
  - In general you do not want several threads trying to retrieve a given message at the same time so you would put a critical region around code to retrieve a message :
    - Enter critical region
    - Get message from message queue
    - Update pointers into message queue
    - Leave critical region (allow other threads to enter)

Remark : If two threads simultaneously try to retrieve a message, one will be allowed to retrieve the message and the other will blocked at the point where the critical region is entered

Shared Memory Programming : Threads

(Contd…)

- Threads may communicate to share work, synchronize or do other tasks
- Because all of the threads are in same process, and will therefore have a common address space, it is fast and easy for them to communicate with each other through global variables

Critical Region

- Simplest mechanism for synchronization is called critical region
- A critical region is a block of code which at most one thread can execute at a time
- Critical regions can be created with mutual exclusion locks or MutEx's.

Shared Memory Programming : Threads

(Contd…)

Benefits:

- Major benefit of multi threaded programs over non threaded ones is in their ability to concurrently execute tasks.
  - In providing concurrency, multithreaded programs introduce a certain amount of overhead.
- Threaded programs have more overhead than a non-threaded one.
- Introducing threads in an application that can't use concurrency, you'll add overhead without any performance benefit.
- Algorithms that are inherently concurrent must be parallelised to take advantage of system.

Why Use Threads Over Processes?

- Creation: Creating a new process can be expensive. More resources and required. Threads are lightweight processes. Threads don't incur much overhead.
- Scheduling: Threads collaborate with each other – processes compete.
- Memory Requirement: Each process has its own view of memory and address space private to it. Threads share same address space but have distinct stacks.
- Communication: Process use Inter-process communication through files, pipes as they have private address space and sometimes through an API like Shared memory segments, Message Queues, Semaphores. IPC is costly in terms of OS overheads.

Why Use Threads Over Processes?

Contd..

- Synchronization: Process synchronization is very complicated and expensive. Thread synchronization can be achieved easily through the common address space.
- Context Switching: Thread task switching time is faster as the context to be saved is minimum compared to a process.
- Specific Requirement: Priority scheduling can be done for specific threads.

Conclusion: Threads are faster and easier.

Threads Parallel Programming

- A process on a multithreaded system is more complex than a process on other systems
- Processes from other systems typically own everything associated with the execution:
  - address space
  - file description
  - working directory
  - Priority
  - registers and everything else
- A multithreaded process may have many threads running concurrently
  - It is difficult to have only one copy of certain resources.

Threads Parallel Programming

- A process is divided into two parts
  - The highest level, called the process – contains global information that is unique within the process or must be know by all members of the process.
  - Lower level is threads which include and manage part of the information and resources of which the whole process is composed.

Threads Parallel Programming

- Resources with process level scope include
  - Address space
  - File description
  - Working directory
  - Any resource that is necessarily process wide in space
- Resources with thread scope include
  - Thread priority
  - Signal work
  - Registers, including counter and stack pointer
  - CPU state

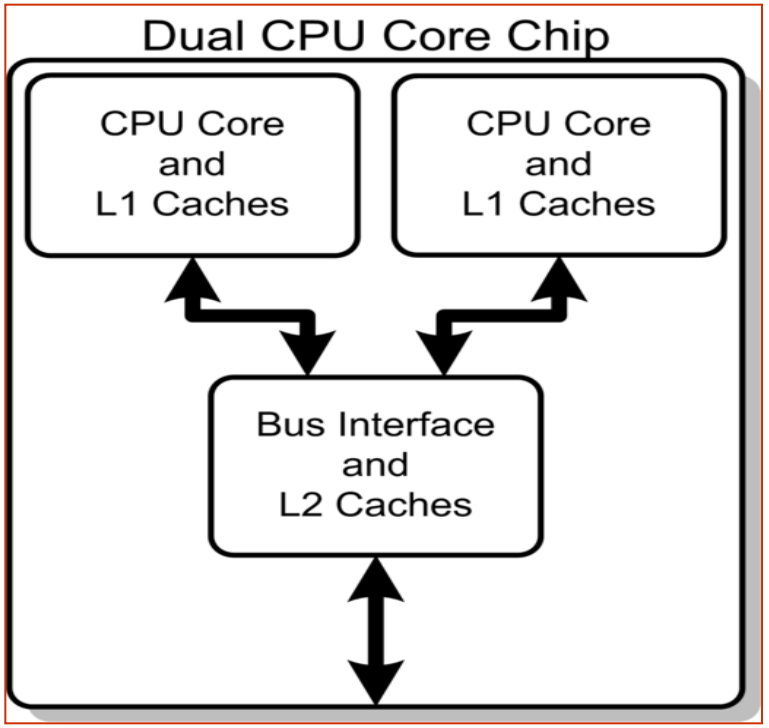
Threads Parallel Programming

(Contd…)

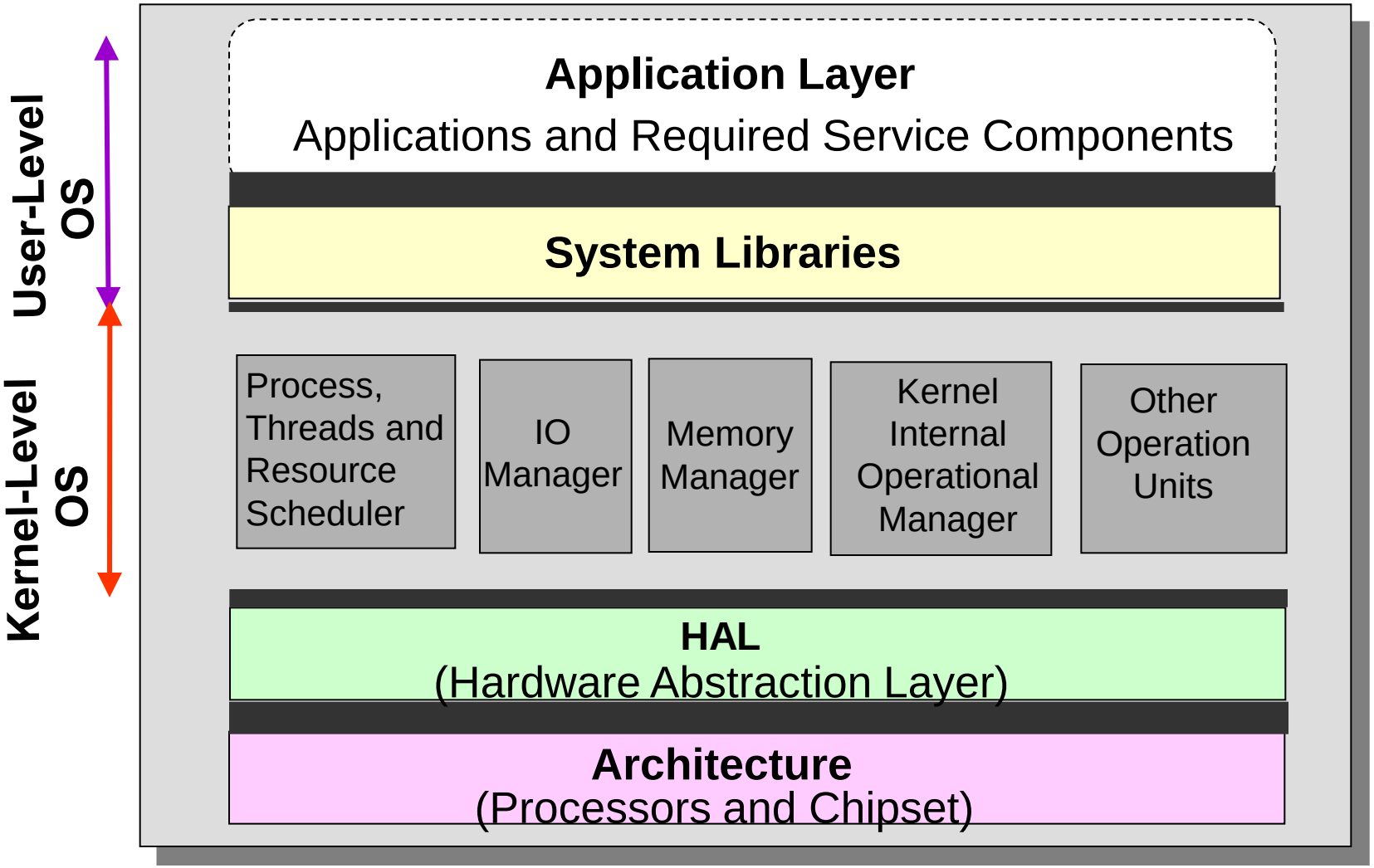
- A thread is a user-level concept that is invisible to the kernel
- Because threads are user-level object, thread operations such as switching from one thread to another are fast because they do not incur a context switch
  - Threads are not visible to the kernel
  - Threads are not scheduled for CPUs
  - Threads have un-describable blocking behavior

System Overview of Threads

- Each Thread maintains its current machine state
- At the hardware level, a thread is an execution path that remains independent of other hardware thread execution paths.
- The operating system maps software threads to hardware execution resources
- Too much threading can hurt Performance



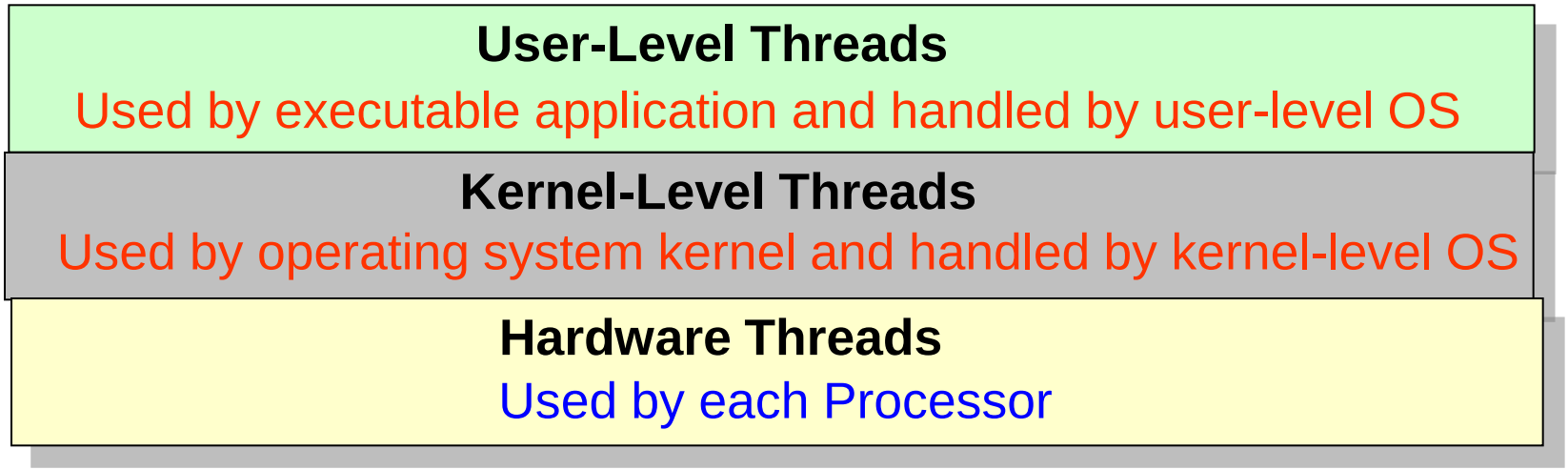
Different Layers of the Operating System /Threads



Source : Reference [4],[6], [7]

System Overview of Threads

- Three levels of threading is commonly used
- Each program thread frequently involves all three levels



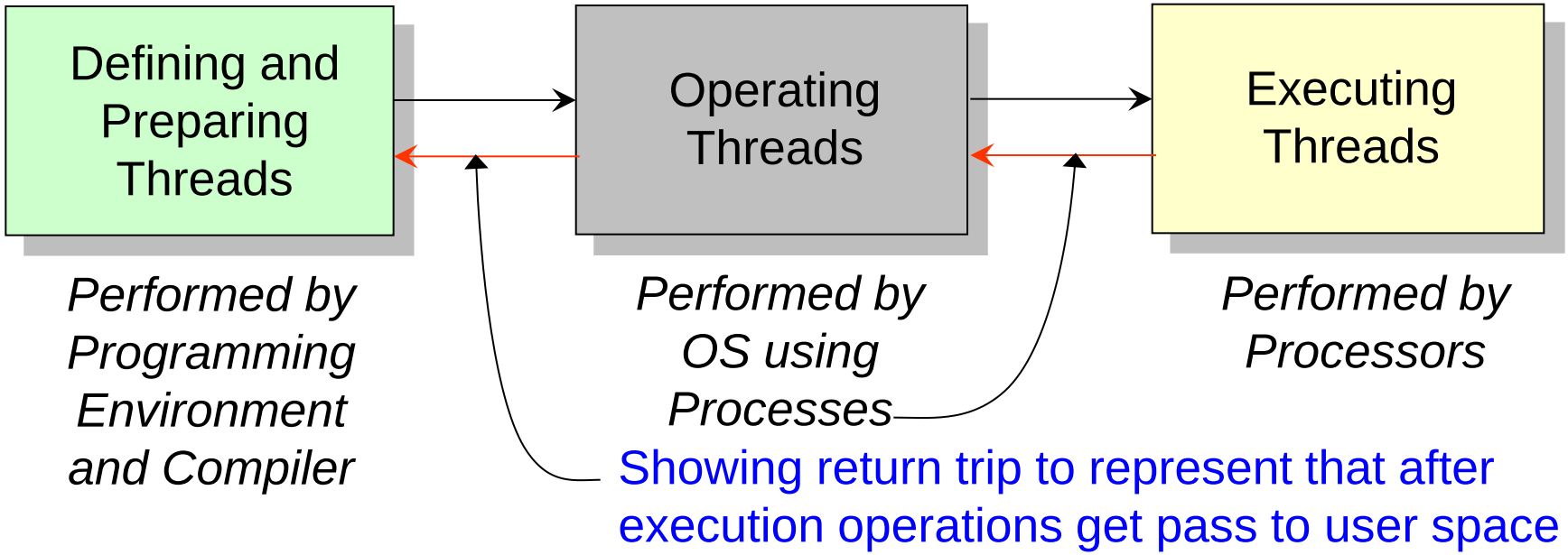
Computation Model of Threading

Source : Reference [4],[6], [7]

System View of Threads

Threads Above the Operating System

- Understand the problems - Face using the threads – Runtime Environment



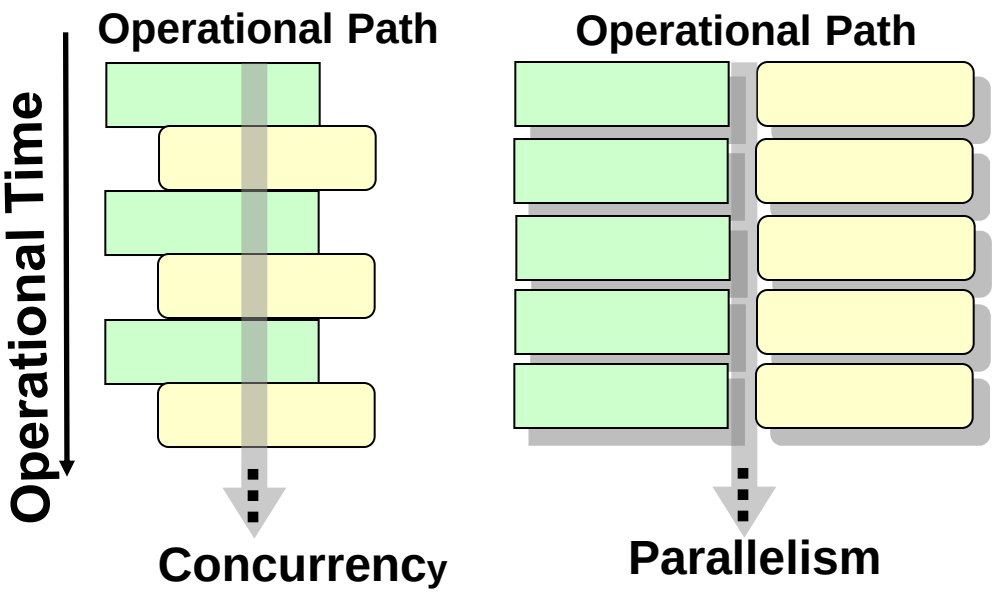
Flow of Threads in an Execution Environment

Source : Reference [4],[6], [7]

System Overview of Threads

Threads Inside the Hardware

- Concurrency versus Parallelism
- Thread stack allocation
- Sharing hardware resources among executing threads – Concurrency



- Hyper-threading Technology
- Chip Multi-threading (CMT)
- Simultaneous Multi-threading (SMT)

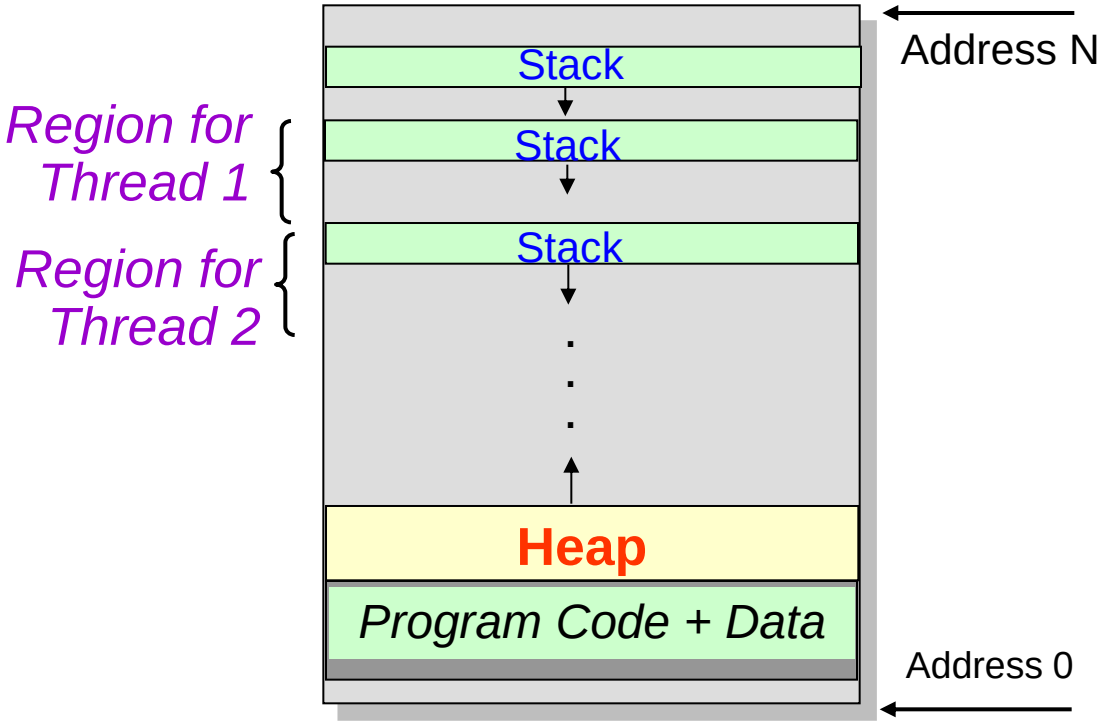
Source : Reference [4],[6], [7]

System Overview of Threads

Stack Layout in a Multi-threaded Process

After thread creation, each thread needs to have its own stack space.

- Thread stack size
- Thread stack allocation
- Know Operating System Limitations
- Default Stack Size may vary from system to system
- Performance may vary from system to system



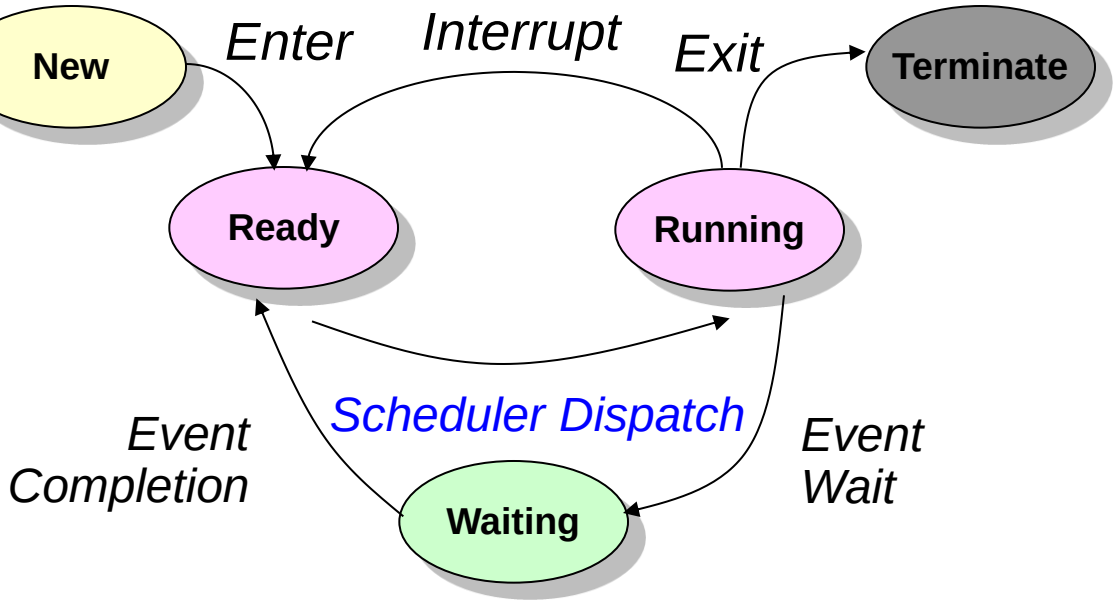
- Bypass the default stack manager and manage stacks on your own as per application demands

Source : Reference [4],[6], [7]

System Overview of Threads

State Diagram for a Thread

- Threads Creation
- Four Stages of Thread Life Cycle
  - Ready,
  - Running,
  - Waiting (blocked),
  - Termination



- Finer Stages in debugging or analyzing a threaded application

Source : Reference [4],[6], [7]

Designing Threaded Programs- Boss/Worker Model

- Identify a task that is suitable for threading by applying to it the following criteria:
  - It is independent of other tasks
  - It can become blocked in potentially long waits
  - It can use a lot of CPU cycles
  - It must respond to asynchronous events
  - Its work has greater or lesser importance than other work in the application

Remark : Several programs - such as those written for database managers, file servers, or print servers - are ideal applications for threading

Source : Reference [4],[6], [7]

Designing Threaded Programs- Boss/Worker Model

- A single thread, the boss, accepts input for the entire program. Based on that input, the *boss* passes off specific tasks to one or more *worker* threads
- The boss creates each worker thread, assigns it tasks, and if necessary, waits for it to finish
- The boss/worker model works well with servers (database servers, file servers, window managers)

Remarks :

- It is important that you minimize the frequency with which the boss and workers communicate.
- One thread is in charge of work assignments for the other threads
- The complexities of dealing with asynchronously arriving requests and communications are encapsulated in the boss.

Source : Reference [4],[6], [7]

Designing Threaded Programs- Peer Models

- All threads work concurrently on their tasks without a specific leader.
- It is also known as work crew model, one thread must create all the other peer threads when the program starts.
- This thread subsequently acts as just another peer thread that processes requests, or suspends itself waiting for the other peers to finish.
- The peer model makes each thread responsible for its own input. A peer knows its own input ahead of time, has its own private way of obtaining its input, or shares a single point of input with other peers.

Remark : The peer model is suitable for applications that have a fixed well-defined set of inputs, such as matrix multipliers, parallel data base search engines, and prime number generators.

Source : Reference [4],[6], [7]

Designing Threaded Programs- Pipeline Models

- The pipeline model assumes
  - A long stream of input
  - A series of sub-operations (known as stages or filters) through which every unit of input must be processed.
  - Each processing stage can handle a different unit of input at a time.

Remark :

  - An automotive assembly line is a classic example of a pipeline.
  - A RISC (reduced instruction set computing) processors also fits the pipeline model. The input to this pipeline is a stream of instructions. Each instruction must pass through the stages of decoding, fetching, operands, computation, and storing results
  - That many instructions may be at various stages of processing at the same time contributes to the exceptionally high performance of RISC processors.

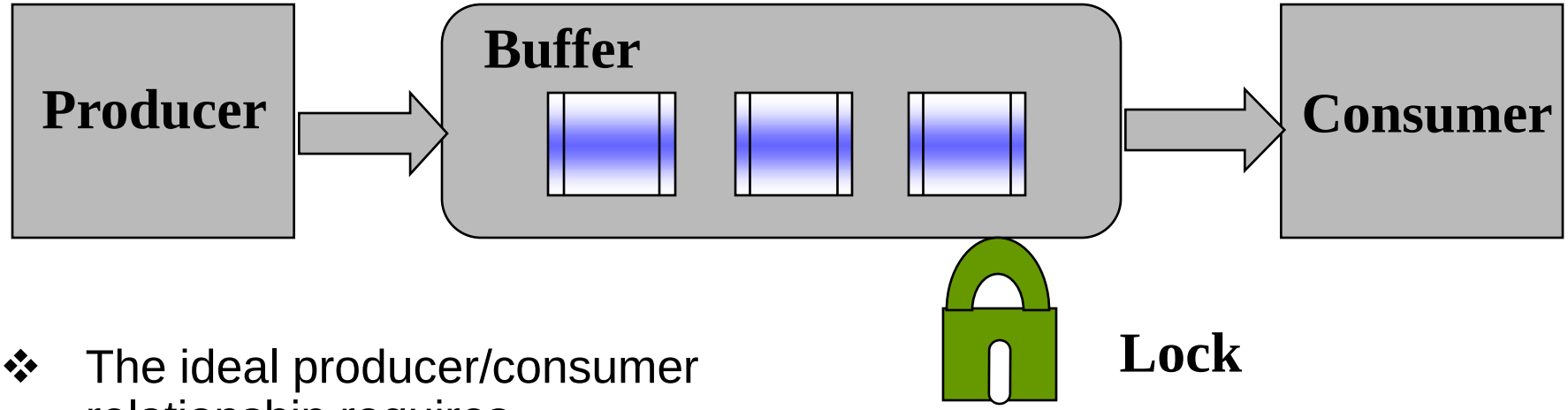
Source : Reference [4],[6], [7]

Designing Threaded Programs-Buffering Data

- Threads transfer data to each other using buffers.
  - In boss/worker model, the boss must transfer requests to the workers.
  - In pipeline model, each thread must pass input to the thread that performs the next stage of processing
  - In peer model, peers may often exchange data
- **Common Problems :** Bugs easily creep into nearly every threaded application. They result from oversights in the way the application manages its shared resources. Managing resources is very difficult
- The basic rule for managing shared resources is simple and twofold
  - Obtain a lock before accessing the resource
  - Release the lock when you are finished with the resource

Designing Threaded Programs-Buffering Data

- A thread assumes either of two roles as it exchanges data in a buffer with another thread. The thread that passes the data to another is known as the *producer*, the one that receives that data is known as *consumer*.



- The ideal producer/consumer relationship requires
  - A buffer
  - A lock
  - A suspend/resume mechanism
  - A state information

Source : Reference [4],[6], [7]

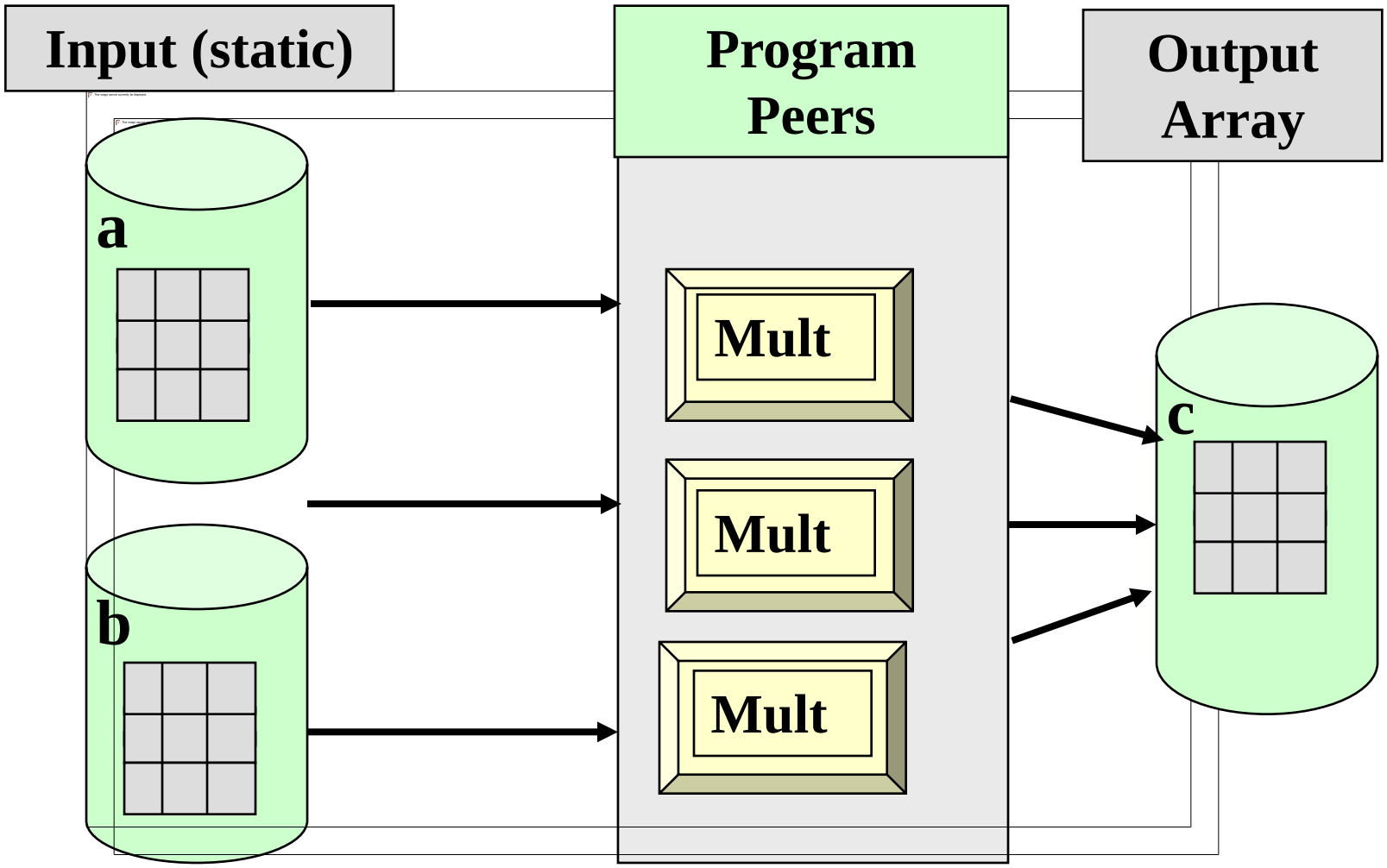
Example for Threaded Program - Matrix Multiplication

- A Matrix multiplication Program : Peer Model

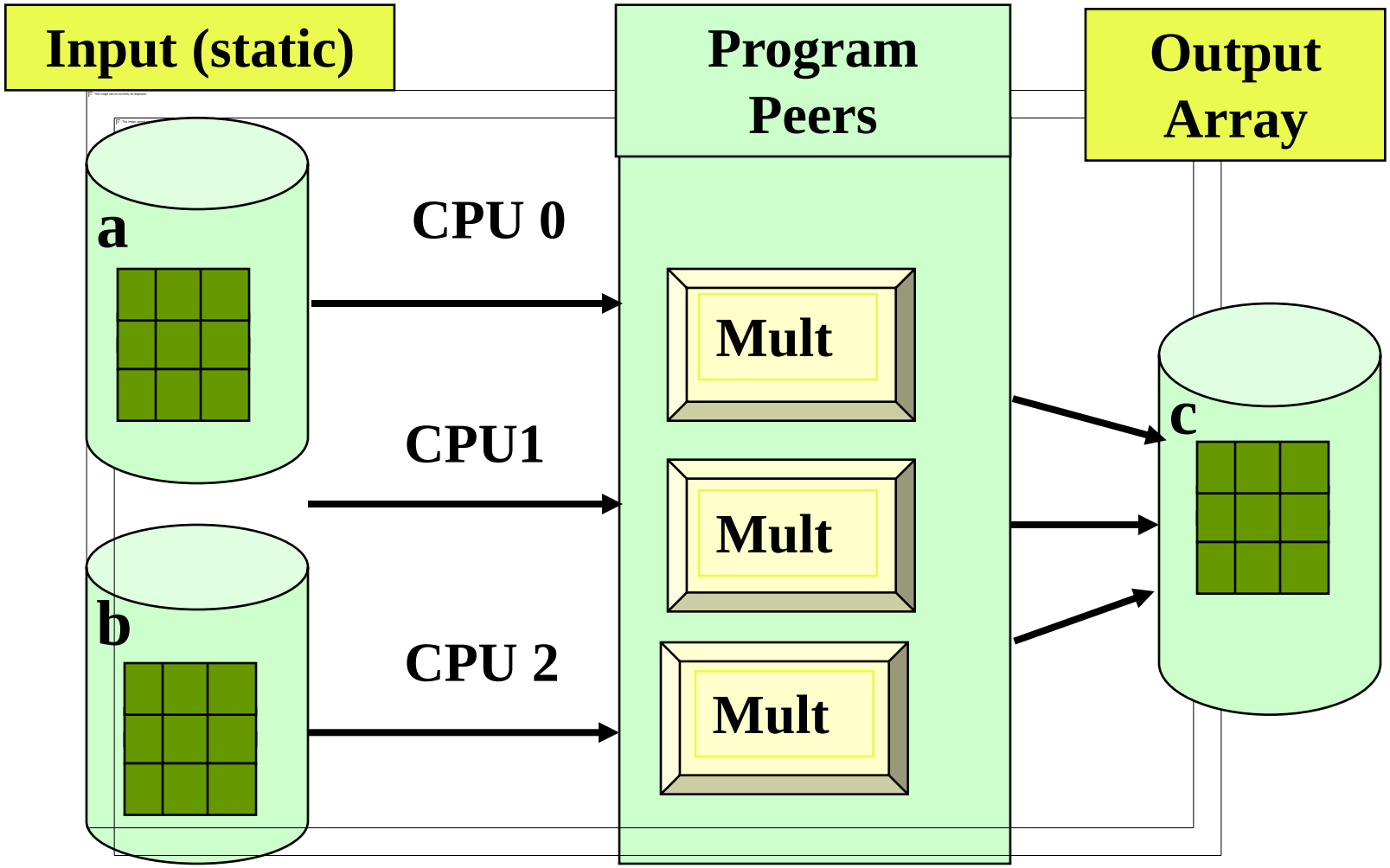
$$C_{row,\ col} = a_{row,1} * b_{1,col} + a_{row,2} * b_{2,col} + \ldots\ldots + a_{row,n} * b_{n,col}$$

  - Assume that the program does not involve I/O operations.
  - Create a peer thread for each individual element in the result array of the matrix C
  - Does not require much unusual synchronization
  - The main thread must wait for the peers to complete
  - No data synchronization is required because the peers never write to any shared locations
  - The computations of each element in the the result array is completely independent of the results for any other element in the result array

Example for Threaded Program - Matrix Multiplication



Example for Threaded Program - Matrix Multiplication



Conclusions

- Important issues in System Overview of Threading
- Defining Threads
- An Overview of Threads – OS /Hardware
- System Overview of Threads
- Different Thread Programming Model

References

1. Andrews, Grogory R. **(2000),** Foundations of Multithreaded, Parallel, and Distributed Programming, Boston, MA : Addison-Wesley

2. Butenhof, David R **(1997),** Programming with POSIX Threads , Boston, MA : Addison Wesley Professional

3. Culler, David E., Jaswinder Pal Singh **(1999),** Parallel Computer Architecture - A Hardware/Software Approach , San Francsico, CA : Morgan Kaufmann

4. Grama Ananth, Anshul Gupts, George Karypis and Vipin Kumar **(2003),** Introduction to Parallel computing, Boston, MA : Addison-Wesley

5. Intel Corporation, **(2003),** Intel Hyper-Threading Technology, Technical User's Guide, Santa Clara CA : Intel Corporation Available at : http://www.intel.com

6. Shameem Akhter, Jason Roberts **(April 2006),** Multi-Core Programming - Increasing Performance through Software Multi-threading , Intel PRESS, Intel Corporation,

7. Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell **(1996),** Pthread Programming O'Reilly and Associates, Newton, MA 02164,

8. James Reinders, Intel Threading Building Blocks – (**2007**) , O'REILLY series

9. Laurence T Yang & Minyi Guo (Editors), (**2006**) *High Performance Computing - Paradigm and Infrastructure* Wiley Series on Parallel and Distributed computing, Albert Y. Zomaya, Series Editor

10. Intel Threading Methodology ; Principles and Practices Version 2.0 copy right **(March 2003),** Intel Corporation

References

11. William Gropp, Ewing Lusk, Rajeev Thakur **(1999),** Using MPI-2, Advanced Features of the Message-Passing Interface, The MIT Press..

12. Pacheco S. Peter, **(1992),** Parallel Programming with MPI, , University of Sanfrancisco, Morgan Kaufman Publishers, Inc., Sanfrancisco, California

13. Kai Hwang, Zhiwei Xu, (**1998**), Scalable Parallel Computing (Technology Architecture Programming), McGraw Hill New York.

14. Michael J. Quinn (**2004**), Parallel Programming in C with MPI and OpenMP McGraw-Hill International Editions, Computer Science Series, McGraw-Hill, Inc. Newyork

15. Andrews, Grogory R. **(2000),** Foundations of Multithreaded, Parallel, and Distributed Progrmaming, Boston, MA : Addison-Wesley

16. SunSoft. Solaris multithreaded programming guide. SunSoft Press, Mountainview, CA, **(1996),** Zomaya, editor. Parallel and Distributed Computing Handbook. McGraw-Hill,

17. Chandra, Rohit, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon, **(2001)**,Parallel Programming in OpenMP San Fracncisco Moraan Kaufmann

18. S.Kieriman, D.Shah, and B.Smaalders **(1995),** Programming with Threads, SunSoft Press, Mountainview, CA. 1995

19. Mattson Tim, **(2002),** Nuts and Bolts of multi-threaded Programming Santa Clara, CA : Intel Corporation, Available at : http://www.intel.com

20. I. Foster **(1995,** Designing and Building Parallel Programs ; Concepts and tools for Parallel Software Engineering, Addison-Wesley (1995)

21. J.Dongarra, I.S. Duff, D. Sorensen, and H.V.Vorst **(1999),** Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools) SIAM, 1999

References

22. OpenMP C and C++ Application Program Interface, Version 1.0". **(1998),** OpenMP Architecture Review Board. October 1998

23. D. A. Lewine. *Posix Programmer's Guide:* **(1991),** Writing Portable Unix Programs with the Posix. 1 Standard. O'Reilly & Associates, 1991

24. Emery D. Berger, Kathryn S McKinley, Robert D Blumofe, Paul R.Wilson, *Hoard : A Scalable Memory Allocator for Multi-threaded Applications* ; The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX). Cambridge, MA, November (**2000)**. Web site URL : http://www.hoard.org/

25. Marc Snir, Steve Otto, Steyen Huss-Lederman, David Walker and Jack Dongarra, (**1998**) *MPI-The Complete Reference: Volume 1, The MPI Core, second edition* [MCMPI-07].

26. William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir (**1998**) *MPI-The Complete Reference: Volume 2, The MPI-2 Extensions*

27. A. Zomaya, editor. Parallel and Distributed Computing Handbook. McGraw-Hill, **(1996)**

28. OpenMP C and C++ Application Program Interface, Version 2.5 (**May 2005**)", From the OpenMP web site, URL **: http://www.openmp.org/**

29. Stokes, Jon 2002 Introduction to Multithreading, Super-threading and Hyper threading *Ars Technica*, October **(2002)**

30. Andrews Gregory R. 2000, Foundations of Multi-threaded, Parallel and Distributed Programming, Boston MA : Addison – Wesley (**2000)**

31. Deborah T. Marr , Frank Binns, David L. Hill, Glenn Hinton, David A Koufaty, J . Alan Miller, Michael Upton, "Hyperthreading, Technology Architecture and Microarchitecture", Intel (**2000-01)**

Thank You

Any questions ?