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-IV

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

The POSIX Threads (Pthreads) Model

Lecture Outline

Following Topics will be discussed

- Performance issues of Multi-Threaded Programs
- An Overview of Common Errors in Multi-threaded Programs

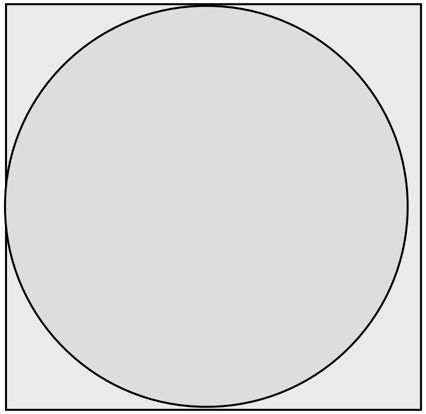
Source : Reference [4],[7]

Pthreads Prog. : Example : π Value

Description : Method is based on generating random numbers in a unit length square and counting the number of points that fall within the largest circle inscribed in the square.

- Area : Circle (πr^2) $=\pi/4$;
- Area : Square = 1 X1

The fraction of random points that fall in the circle should approach to π /4

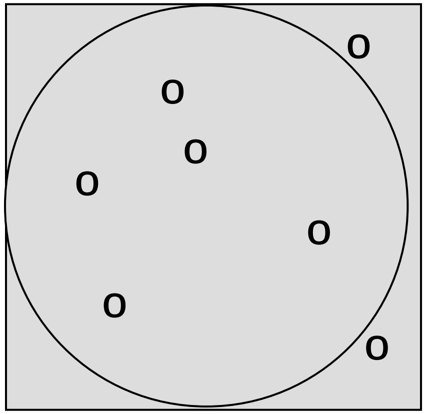


Source : Reference [4],[7]

Pthreads Prog. : Example : π Value

1. Assign fixed number of points to each thread.
2. Each thread generates random points and keeps track of the number of points that land in circle locality.
3. After all threads finish execution, their counts are combined to computer the value of π (by calculating the fraction over all threads and multiplying by 4)

Implementation & Performance Issues



- Use of `pthread_create` function and `pthread_join function`

Pthreads Prog. : Example : π Value

Implementation & Performance Issues

- Read Desired number of threads (**`num_threads`**) and the number of sample points (**`sample_points`**)
- Divide the number of points equally among the threads (Use of **`pthread_create`** function)
- Each thread keeps track of number of **`hits`** (points inside the circle
- Each thread computes the respective hit ratios
- Combine the partial results to determine π (Use of **`pthread_join`** function)

Pthreads Prog. : Example : π Value

Performance Issues

- False Sharing of data items (Two adjoining data items (which likely reside on the same cache line) are being continually written to by threads that might be scheduled on different cores.

- Estimate the cache line size of the cores and use higher dimensional arrays that are proportional to number of cores which share the cache line.

Synchronization Primitives in Pthreads

- Controlling Thread Attributes and Synchronization
  - Attribute Objects for Threads
  - Attribute Objects for Mutexes
- Thread Cancellation
  - Clean-up functions are invoked for reclaiming the thread data structures
- Composite synchronization Primitives
  - Read-Write Locks (Data Structure is read frequently but written infrequently.
  - Issues of Multiple reads /Serial writes
  - Issues of Read Locks; read-write locks etc…

Synchronization Primitives in Pthreads

- Barriers
  - A barrier call is used to hold a thread until all other threads participating in the barrier have reached the barrier
  - Barriers can be implemented using a counter, a mutex, and a condition variable.
  - A single integer is used to keep track of the number rof threads that have reached the Barrier
- Remark :
  - Barrier implementation using mutexes may suffer from the overhead of busy-wait.

Synchronization Primitives in Pthreads

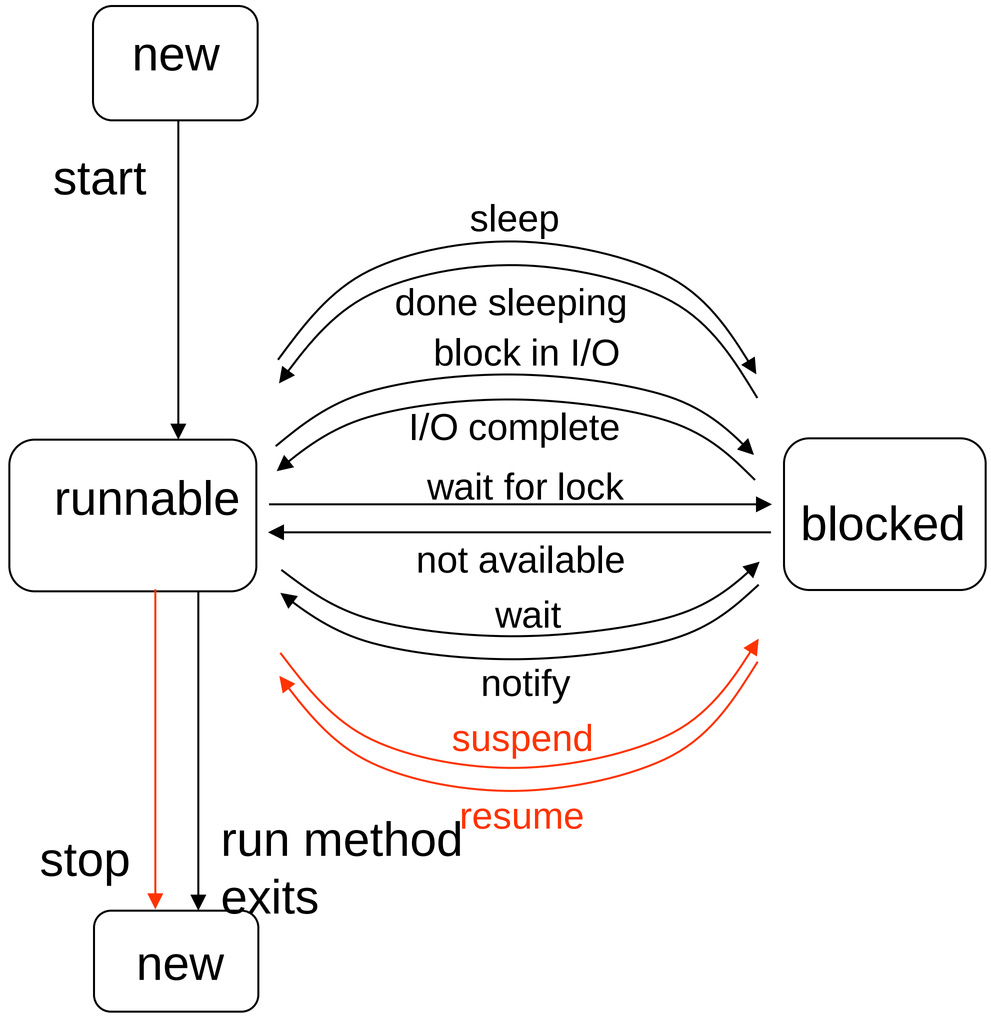
- **Mutual Exclusion for Shared Variables**
  - Thread APIs provide support for implementing critical sections and atomic operations using mutex-locks (mutual exclusion locks)
- **Condition Variables for Synchronization**
  - When thread performs a condition wait, it takes itself off the runnable list – Does not use any CPU cycle

Remark :

- Mutex Lock consumes CPU cycles as it polls for the lock
- Condition wait consumes CPU cycles when it is woken up

Pthreads:Synchronization & Thread States

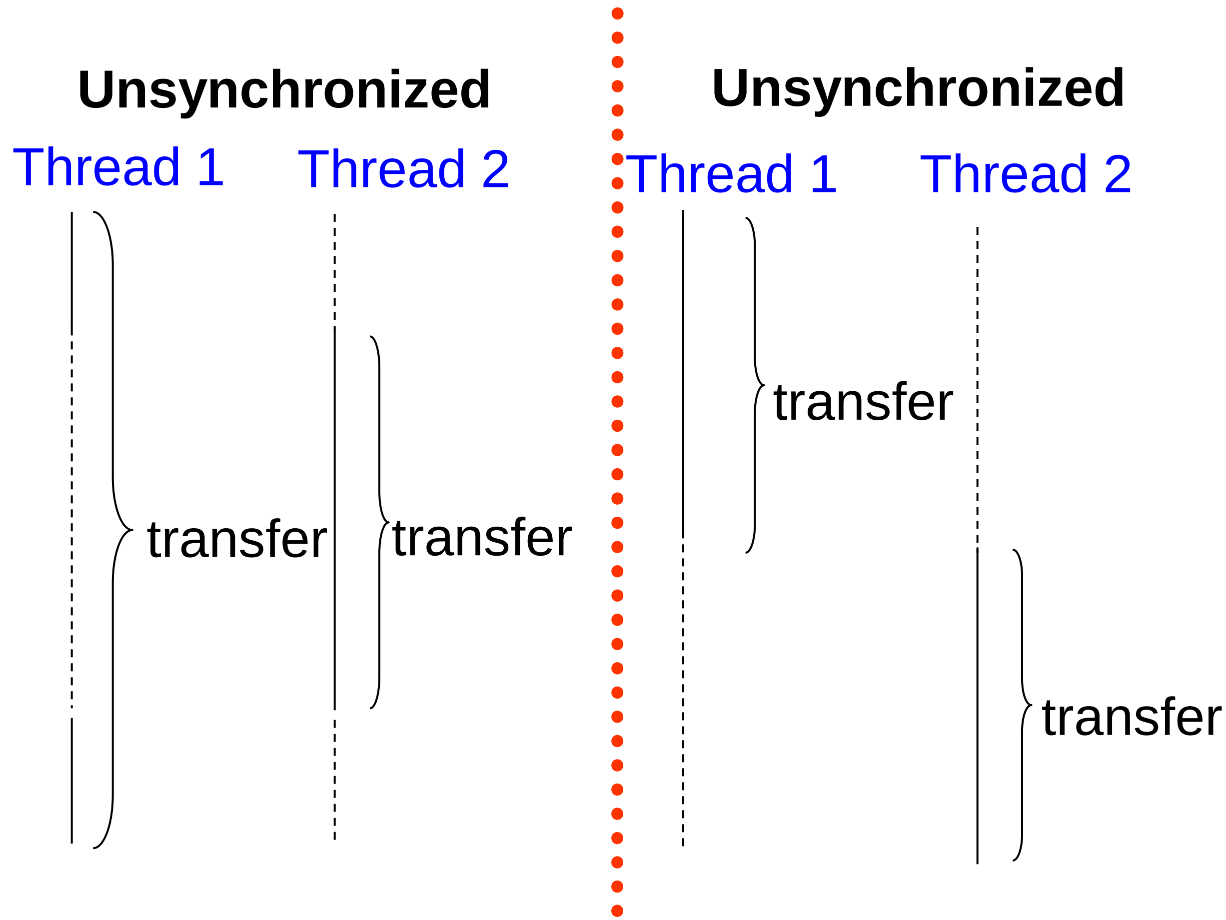
- I/O Requests
- Read-Write Locks
- Available CPU
- Release Locks
- Critical Sections



Comparison of unsynchronized / synchronized threads

❖ Too little / too much synchronization

- ➢ In-Correct Results
- ➢ Performance – Slow done the results



Synchronization Primitives in Pthreads

Example: Two threads on 2 cores are both trying to increment a variable x at the same time (Assume x is initially 0)

THREAD 1 :
Increment (x)
{
x= x+1
}
THREAD 1:
10 LOAD A, (x address)
20 ADD A, 1
30 STORE A, x address)
	THREAD 1 :
	Increment (x)
	{
	x= x+1
	}
	THREAD 1:
	10 LOAD A, (x address)
	20 ADD A, 1
	30 STORE A, x address

Use Threaded APIs mutex-locks (Mutual exclusion locks) to avoid Race Conditions

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

Synchronization Primitives in Pthreads

- Mutual Exclusion for Shared Variables

- Implementation of critical sections and atomic operations using **`mutex-locks`** (mutual exclusion locks)

- Mutex locks have two states (locked and unlocked) Use functions **`pthread_mutex_lock`** & **`pthread_mutex_unlock`** function)

- A function to initialize a mutex-lock to its unlocked state - **`pthread_mutex_init`** function)

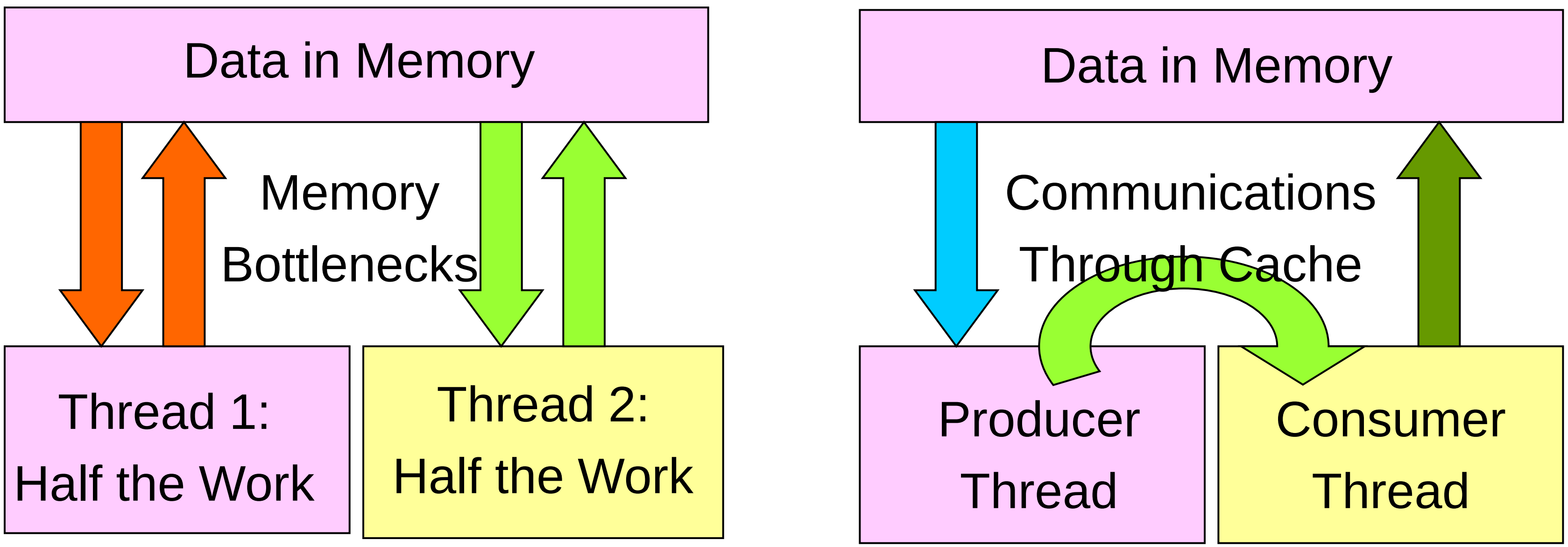
Synchronization Primitives in Pthreads

- **Example : ** Computing the minimum entry in a list of integers
  - The list is partitioned equally among the threads
  - The size of each thread's partition is stored in the variable

- Performance for large number of threads is not scalable (At any point of time, only one thread can hold a lock, only one thread can test updates the variable.)

Producer/Consumer Problem : Synchronizing Issues

- ❖ Producer thread generates tasks and inserts it into a work-queue.
- ❖ The consumer thread extracts tasks from the task-queue and executes them one at a time.



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

Producer & Consumer : Critical Directive

- Producer thread generates tasks and inserts it into a work-queue.
- The consumer thread extracts tasks from the task-queue and executes them one at a time.
  - There is concurrent access to the task-queue, these accesses must be serialized using critical blocks.
  - The tasks of inserting and extracting from the task-queue must be serialized.
  - Define your own "insert_into_queue" and "extract_from_queue" from queue (Note that queue full & queue empty conditions must be explicitly handled)

Producer & Consumer : Critical Directive

- **Possibilities & Implementation Issues on Multi cores**
  - The producer thread must not overwrite the shared buffer when the previous task has not been picked up by a consumer thread
  - The consumer threads must not pick-up tasks until there is something present in the shared data structure.
  - Individual consumer threads should pick-up tasks one at a time.
- Implementation can be done using variable called `task_variable` which handles the wait condition of consumer & producer.

Producer & Consumer : Critical Directive

❖ **Implementation & Performance Issues on Multi cores**

- If **`task_variable`** = 0
  - *Consumer* threads wait but the *producer* thread can insert tasks into the shared data structure.
- If **`task_variable`** = 1
  - *Producer* threads wait to insert the task into the shared data structure but one of the *Consumer* threads can pick up the task available.
- All these operations on the variable **`task_variable`** should be protected by mutex-locks to ensure that only one thread is executing test-update on it.

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

Producer & Consumer : Critical Directive

- **Performance Issues on Multi cores**
  - Consumer thread waits for a task to become available and executes when it is available.
  - Locks represent sterilization points since critical sections must be executed by one after the other.
  - Handle Shared Data Structures and Critical sections to reduce the idling overhead.

- **Alleviating Locking Overheads**
  - To reduce the idling overhead associated with locks using `pthread_mutex_trylock`.

Producer & Consumer : Critical Directive

- Critical Section directive is a direct application of the corresponding mutex function in Pthreads

- Reduce the size of the critical section in Pthreads/OpenMP to get better performance (Remember that critical section represents serialization points in the program)

- Critical section consists simply of an update to a single memory location.

- Safeguard : Define Structured Block I.e. no jumps are permitted into or out of the block. This leads to the threads wait indefinitely.

Synchronization Primitives in Pthreads :Alleviating Locking Overheads

❖ **Example : Finding *k-matches* in a list**

- Finding **k** matches to a query item in a given list. (The list is partitioned equally among the threads. Assume that the list has **n** entries, each of **p** threads is responsible for searching **n/p** entries of the list.
- Implement using **pthread_mutex_lock.**
- Reduce the idling overhead associated with locks using **pthread_mutex_trylock.** (Reduce the Locking overhead can be alleviated)

Source : Reference [4]

Producer & Consumer : Condition Variable for Synchronization

- A Condition Variable is a data object used for synchronization threads. This variable allows a thread to block itself until specified data reaches a predefined state.
- A condition variable always has a *mutex* associated with it.
- Use functions **`pthread_cond_init`** for initializing and **`pthread_cond_destroy`** for destroying condition variables.
- The concept of polling for lock as it consumes CPU cycles can be reduced. Use of condition variables may not use any CPU cycles until it is woken up.

Composite Synchronization Constructs

- The higher level synchronization constructs can be built using basic constructs.
- **Read-Write Constructs**
  - A data structure is read frequently but written infrequently.
  - Multiple reads can proceed without any coherence problems. Write must be serialized.
- A structure can be defined as **`read-write`** lock

Example 1 : Using read-write locks for computing the minimum of a list of integers

Example 2 : Using read-write locks for implementing hash tables.

Source : Reference : [4]

Composite Synchronization Constructs

❖ Read-Write Struct

➢ typedef struct {

```
typedef struct {
     int readers;
    int writer;
    pthread_conf_t   readers_proceed;
    pthread_cond_t  writer_proceed;
    int pending_writers;
    pthread_muex_t read_write_lock;
} mylib_rwlock_t;
```

Source : Reference : [4]

Composite Synchronization Constructs

- **Read-Write Constructs**
  - Offer advantages over normal locks
  - For frequent reads /Writes, overhead is less
  - Using normal **`mutexes`** for writes is advantages when there are a significant number of **`read`** operations
- For performance of database applications (hash tables) on Multi Cores, the **`mutex`** lock version of the progam hashes key into the table requires suitable modification.

Source : Reference : [4]

Composite Synchronization Constructs

- **Barrier :** A barrier call is used to hold a thread until all other threads participating in the barrier have reached the barrier.
  - Barrier can be implemented using a **`counter`**, a **`mutex`**, and a **`condition`** variable.
  - Overheads will vary for large number of **`threads`**.
  - Performance of programs depends upon the application characteristics such as the number of threads & the number of **`condition`** variable **`mutexes`** pairs for implementation of a barrier for **`n threads`**.

Programming Aspects Examples

Implementation of Streaming Media Player on Multi-Core

- One decomposition of work using Multi-threads
- It consists of
  - A thread Monitoring a network port for arriving data,
  - A decompressor thread for decompressing packets
  - Generating frames in a video sequence
  - A rendering thread that displays frame at programmed intervals

Source : Reference : [4]

Programming Aspects Examples

Implementation of Streaming Media Player on Multi-Core

❖ The thread must communicate via shared buffers –

- an in-buffer between the network and decompressor,
- an out-buffer between the decompressor and renderer

❖ It consists of

➢ Listen to port ……..Gather data from the network

➢ Thread generates frames with random bytes (Random string of specific bytes)

➢ Render threads pick-up frames & from the out-buffer and calls the display function

➢ Implement using the Thread Condition Variables

Explicit Threads *versus* OpenMP Based Prog.

- ❖ OpenMP provides a layer on top of naïve threads to facilities a variety of thread-related tasks.

- ❖ Using Directives provided by OpenMP, a programmer is get rid of the task of initializing attribute objects, setting up arguments to threads, partitioning iteration spaces etc…. (This may be useful when the underlying problem has a static and /or regular task graph.)

- ❖ The overheads associated with automated generation of threaded code from directives have been shown to be minimal in the context of a variety of applications.

Explicit Threads *versus* OpenMP Based Prog.

- An Artifact of Explicit threading is that data exchange is more apparent. This helps in alleviating some of the overheads from data movement, false sharing, and contention.
- Explicit threading also provides a richer API in the form of condition waits.
- Locks of different types, and increased flexibility for building composite synchronization operations

Explicit Threads *versus* OpenMP Based Prog.

- Compiler support on Multi-Cores play an important role
- Issues related to OpenMP performance on Multi cores need to be addressed.
- Inter-operability of OpenMP/Pthreads on Multi-Cores require attention -from performance point of view
- Performance evaluation and use of tools and Mathematical libraries play an important role.

Source : Reference [4]

Common Errors /Solutions : Prog. Paradigms

Key Points

- Match the number of runnable software threads to the available hardware threads
- Synchronization : In correct Answers ; Performance Issues
- Keeps Locks private
- Avoid dead-locks by acquiring locks in a consistent order
- Memory Bandwidth & contention Issues
- Lock contention (Using Multiple distributed locks)
- Design Lockless Algorithms – Advantages & dis-advantages
- Cache lines are – Hardware threads
- Writing synchronized code – Memory Consistency

Common Errors /Solutions : Prog. Paradigms

Key Points

- Set up all the requirements for a thread before actually creating the thread. This includes initializing the data, setting thread attributes, thread priorities, mutex, attributes, etc…
- Buffer management is required in applications such as producer and consumer problems.
- Define synchronizations and data replication wherever it is possible and address stack variables,
- Avoid Race Conditions in designing algorithms and implementation
- Extreme caution is required to avoid parallel overheads associated with synchronization
- Design of asynchronous Programs and use of scheduling techniques require attention.

Pthreads :Conclusions

- ❖ Features and advantages of Pthreads is discussed.
- ❖ Pthreads – Synchronization Constructs are discussed.
- ❖ Performance issues of Multi-threaded Programs using POSIX Thread APIs.
- ❖ Thread Safety issues & Error handling are important for producing correct results

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 ?