SWE-594 MULTI-CORE PROGRAMMING

# Introduction

## Von Neumann Architecture

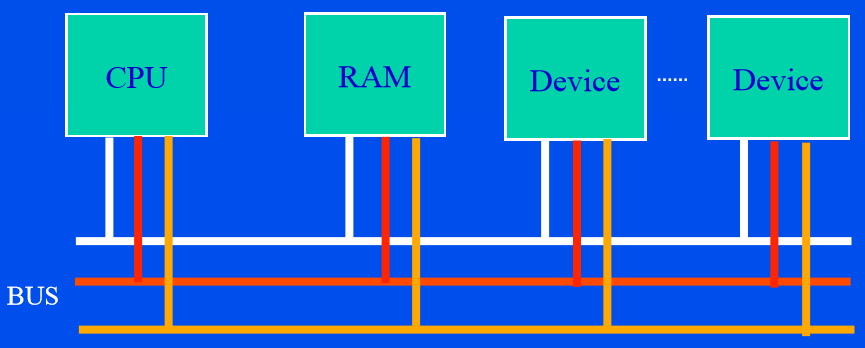
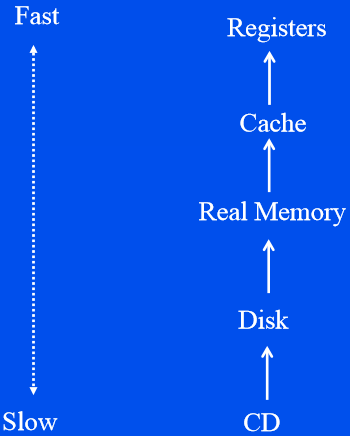


Figure 1 Serial Computer

## Memory Hierarchy

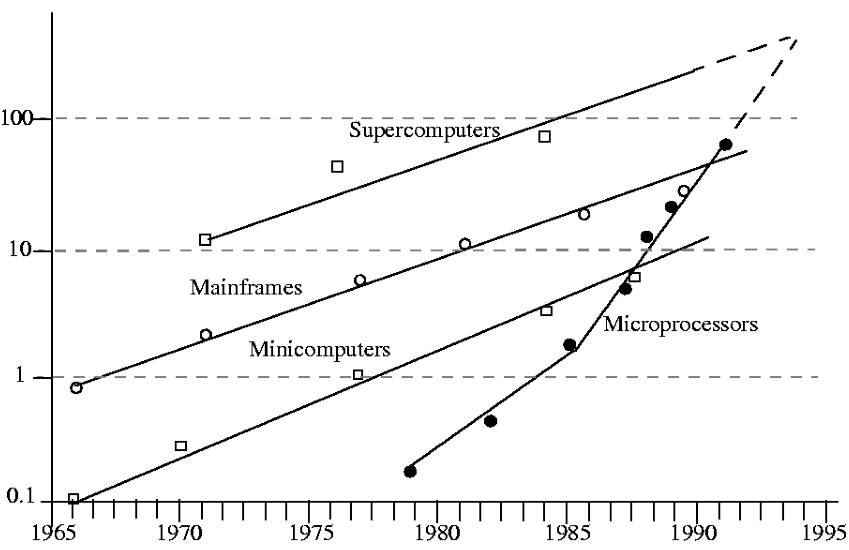


## History of Computer Architecture

Generations (identified by logic technology)

1. Tubes
2. Transistors
3. Integrated Circuits
4. VLSI (very large scale integration)

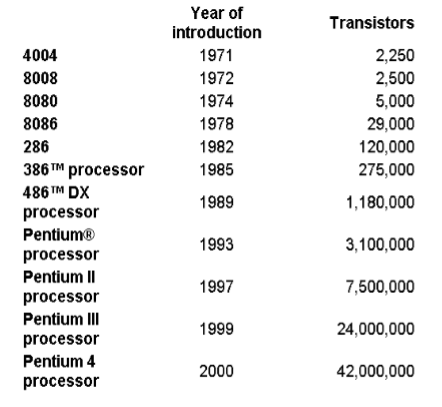
## Performance Trends



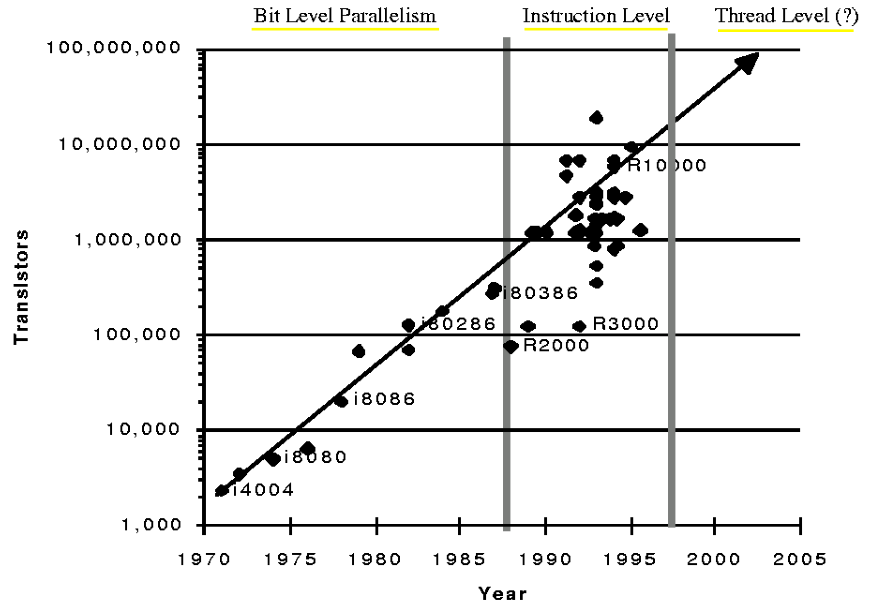
* Traditional mainframe/supercomputer performance 25% increase per year
* But … microprocessor performance 50% increase per year since mid-80.

## Moore’s Law

* “Transistor density doubles every 18 months”
* Moore is co-founder of Intel.
* 60 % increase per year
* Exponential growth
* PC costs decline.
* PCs are building bricks of all future systems.



## VLSI Generation



## Bit Level Parallelism (up to mid-80’s)

* 4 bit microprocessors replaced by 8 bit, 16 bit, 32 bit etc.
* Doubling the width of the data path reduces the number of cycles required to perform a full 32-bit operation
* Mid 80’s reap benefits of this kind of parallelism (full 32bit word operations combined with the use of caches)

## Instruction Level Parallelism (mid-80’s to mid-90’s)

* Basic steps in instruction processing (instruction decode, integer arithmetic, address calculations, could be performed in a single cycle)
* Pipelined instruction processing
* Reduced instruction set (RISC)
* Superscalar execution
* Branch prediction

## Thread/Process Level Parallelism (mid-90’s to present)

* On average control transfers occur roughly once in five instructions, so exploiting instruction level parallelism at a larger scale is not possible
* Use multiple independent “threads” or processes
* Concurrently running threads, processes

## Evolution of the Infrastructure

* Electronic Accounting Machine Era: 1930-1950
* General Purpose Mainframe and Minicomputer Era: 1959Present
* Personal Computer Era: 1981 – Present
* Client/Server Era: 1983 – Present
* Enterprise Internet Computing Era: 1992- Present

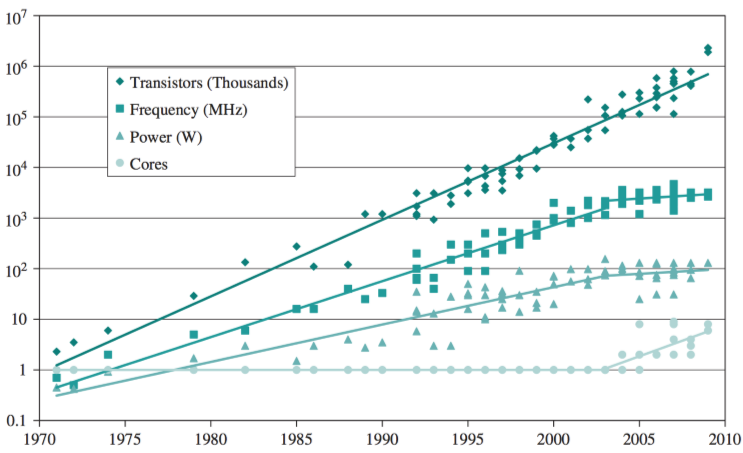
## Sequential vs Parallel Processing

|  |  |
| --- | --- |
| Sequential Processing | Parallel Processing |
| physical limits reached | “raw” power unlimited |
| easy to program | more memory, multiple cache |
| expensive supercomputers | made up of COTS, so cheap |
|  | difficult to program |
|  |  |

## What is Multi-Core Programming?

Answer: It is basically parallel programming on a single computer box (e.g. a desktop, a notebook, a blade)

## Processor Trends



## Another Important Benefit of Multi-Core: Reduced Energy Consumption

|  |  |
| --- | --- |
| Single Core | Multi-Core |
|  |  |

## SPMD Model (Single Program Multiple Data)

* Each processor executes the same program asynchronously
* Synchronization takes place only when processors need to exchange data
* SPMD is extension of SIMD (relax synchronized instruction execution)
* SPMD is restriction of MIMD (use only one source/object)

## Parallel Processing Terminology

* Embarrassingly Parallel:
  + applications which are trivial to parallelize
  + large amounts of independent computation
  + Little communication
* Data Parallelism:
  + model of parallel computing in which a single operation can be applied to all data elements simultaneously
  + amenable to SIMD or SPMD style of computation
* Control Parallelism:
  + many different operations may be executed concurrently
  + require MIMD/SPMD style of computation
* Scalability:
  + If the size of problem is increased, number of processors that can be effectively used can be increased (i.e. there is no limit on parallelism).
  + Cost of scalable algorithm grows slowly as input size and the number of processors are increased.
  + Data parallel algorithms are more scalable than control parallel algorithms
* Granularity:
  + **Fine grain machines:** employ massive number of weak processors each with small memory
  + **Coarse grain machines:** smaller number of powerful processors each with large amounts of memory

## Models of Parallel Computers

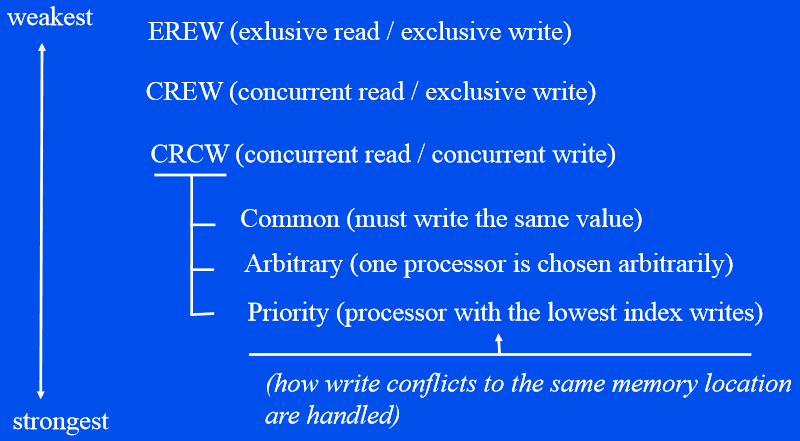
1. Message Passing Model
   * Distributed memory
   * Multicomputer
2. Shared Memory Model
   * Multiprocessor
   * Multi-core
3. Theoretical Model
   * PRAM

New architectures: combination of 1 and 2.

## Theoretical PRAM Model

* Used by parallel algorithm designers
* Algorithm designers do not want to worry about low level details: They want to concentrate on algorithmic details
* Extends classic RAM model
* Consist of :
  + Control unit (common clock), synchronous
  + Global shared memory
  + Unbounded set of processors, each with its private own memory
* Some characteristics
  + Each processor has a unique identifier, mypid=0, 1, 2…
  + All processors operate synchronously under the control of a common clock
  + In each unit of time, each processor is allowed to execute an instruction or stay idle

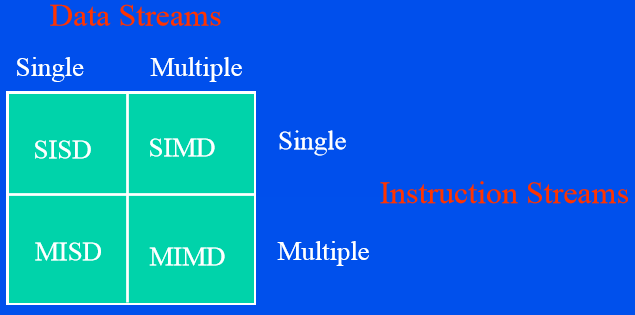
## Various PRAM Models



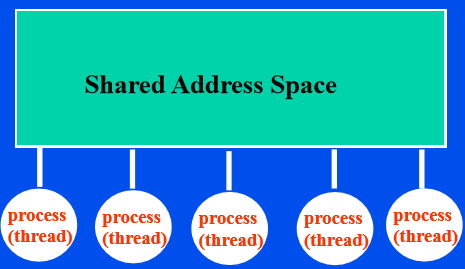
## Flynn’s Taxonomy

Classifies computer architectures according to:

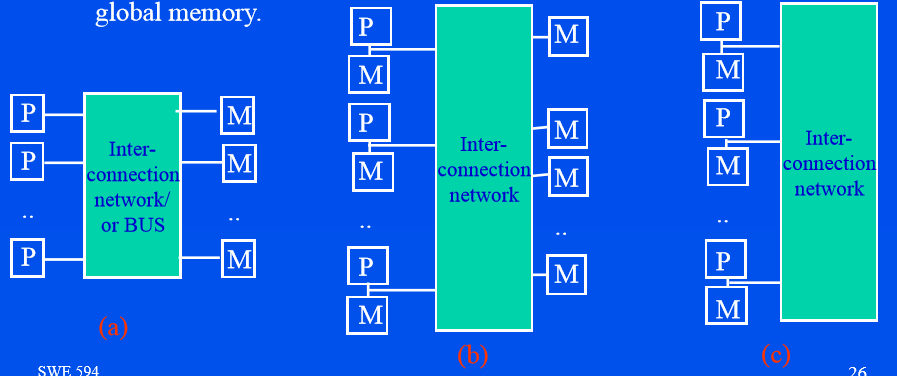
1. Number of instruction streams it can process at a time
2. Number of data elements on which it can operate simultaneously



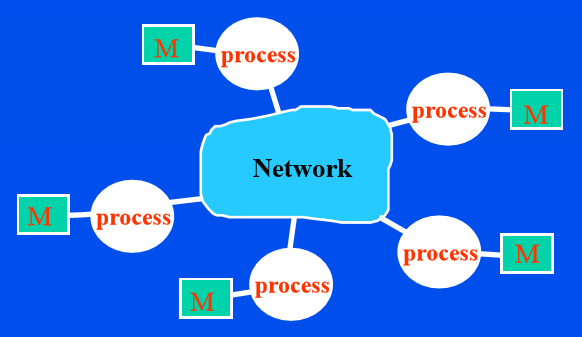
## Shared Memory Machines



* Memory is globally shared, therefore processes (threads) see single address space
* Coordination of accesses to locations done by use of locks provided by thread libraries
* Example Machines: Sequent, Alliant, SUN Ultra, Dual/Quad Board Pentium PC
* Example Thread Libraries: POSIX threads, Linux threads.
* can be classified as:
  + UMA: uniform memory access
  + NUMA: non-uniform memory access based on the amount of time a processor takes to access local and global memory.



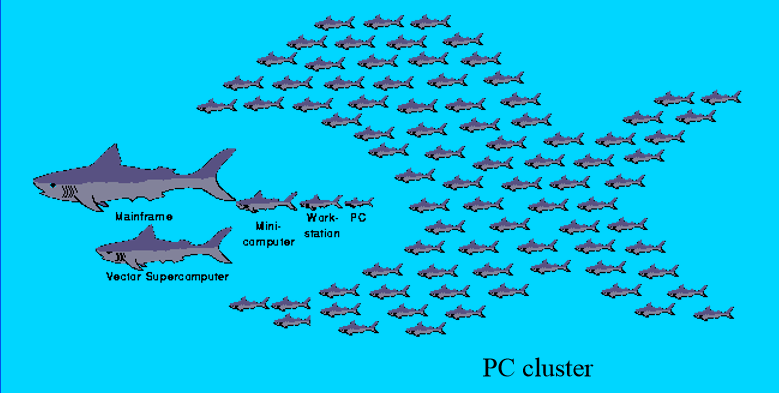
## Distributed Memory Machines



* Each processor has its own local memory (not directly accessible by others)
* Processors communicate by passing messages to each other
* Example Machines: IBM SP2, Intel Paragon, COWs (cluster of workstations)
* Example Message Passing Libraries: PVM, MPI

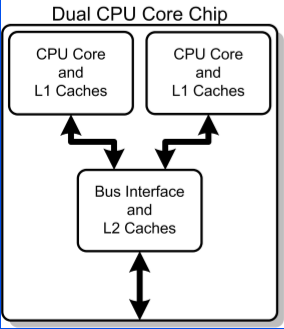
## Beowulf Clusters

* Use COTS, ordinary PCs and networking equipment
* Has the best price/performance ratio



## Multi-Core Computing

* A multi-core microprocessor is one which combines two or more independent processors into a single package, often a single integrated circuit.
* A dual-core device contains only two independent microprocessors.



## Comparison of Different Architectures

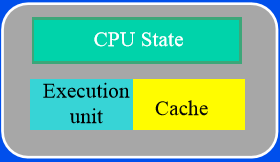


Figure 2 Single Core Architecture

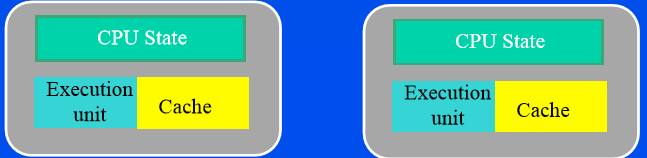


Figure 3 Multi-Processor Architecture

## Hyper-threading

Hyper-threading technology enables two threads to be processed in a single processor core allowing partially parallel execution whereas traditional single threaded processors could only process one thread at a time

Architecturally, a processor with Hyper-Threading Technology consists of two logical processors per core, each of which has its own processor architectural state. Each logical processor can be individually halted, interrupted or directed to execute a specified thread, independently from the other logical processor sharing the same physical core.

The logical processors in a hyper-threaded core share the execution resources. These resources include the execution engine, caches, and system bus interface

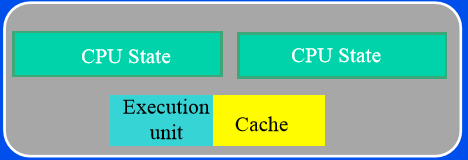


Figure 4 Hyper-threading Architecture

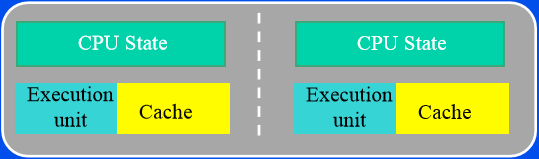


Figure 5 Multi-Core Processor

The dual core processor today's best choice for notebooks which enabled two threads to be fully processed in their own processor cores for truly parallel execution.

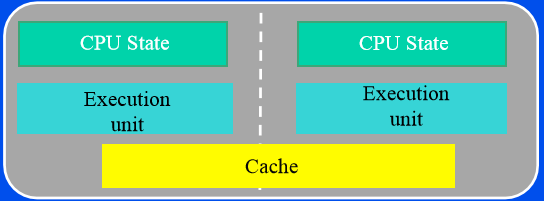


Figure 6 Multi-Core with shared cache

## Multi-Core with Hyper-threading Technology

Multi-core processors with hyper-threading technology within each processor core doubling the number of threads that can be processed at the same time. These advanced processors are the preferred processor for desktop users running multiple demanding applications simultaneously or for running today's multi-threaded digital media applications and gaming titles.

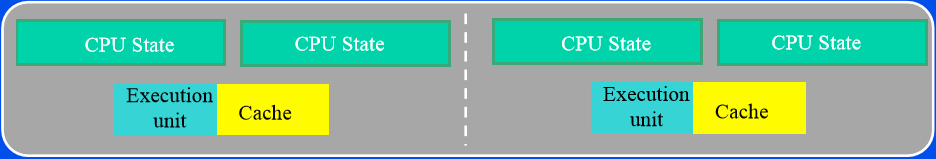
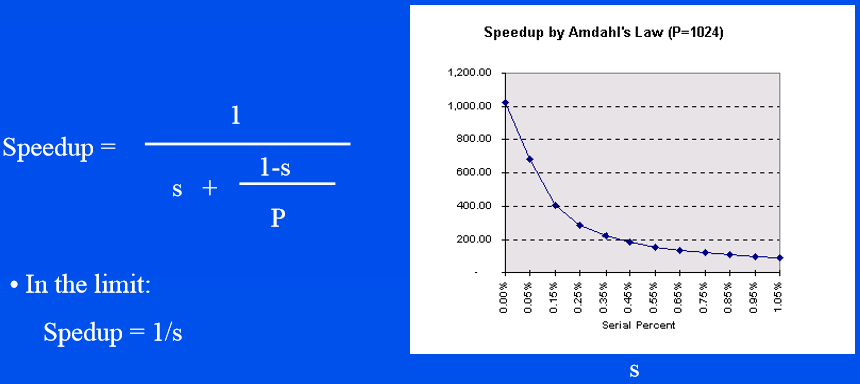


Figure 7 Multi-Core with Hyper-Threading Technology

# PARALLEL PERFORMANCE MODELS and ALGORITHMS

## Amdahl’s Law

* The serial percentage of a program is fixed. So speed-up obtained by employing parallel processing is bounded.
* Lead to pessimism in in the parallel processing community and prevented development of parallel machines for a long time.

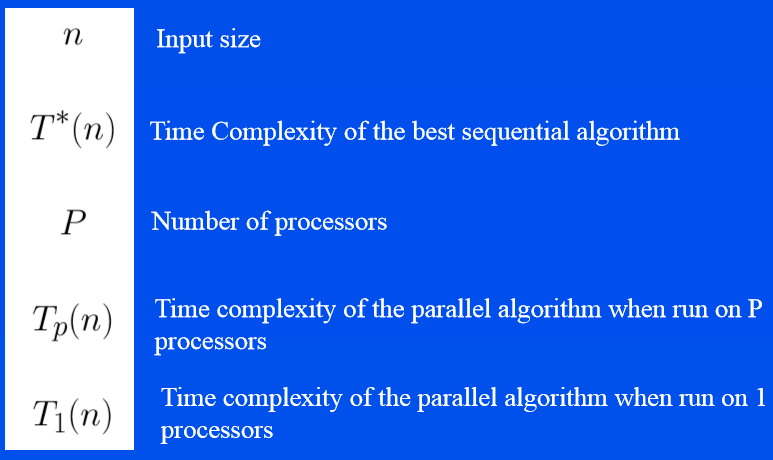


## Gustafson’s Law

* Serial percentage is dependent on the number of processors/input.
* Demonstrated achieving more than 1000 fold speedup using 1024 processors.
* Justified parallel processing

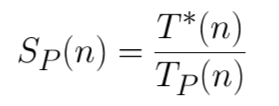
## Algorithmic Performance Parameters

Notation:

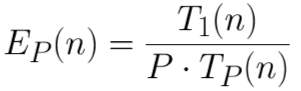


## Algorithmic Performance Parameters

* Speed-Up



* Efficiency

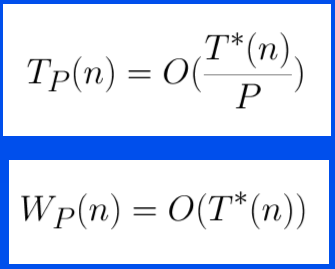


## Algorithmic Performance Parameters

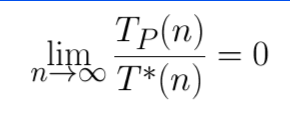
* Work = Processors X Time
  + Informally: How much time a parallel algorithm will take to simulate on a serial machine
  + Formally:



* Work Efficient:
  + Informally: a work efficient parallel algorithm does no more work than the best serial algorithm
  + Formally: a work efficient algorithm satisfies:



* Scalability:
  + Informally, scalability implies that if the size of the problem is increased, the number of processors effectively used can be increased (i.e. there is no limit on parallelism)
  + Formally, scalability means:

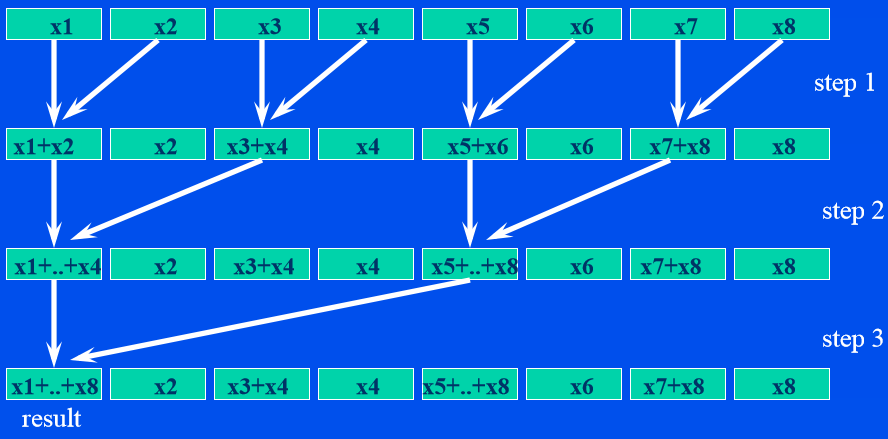


* Some remarks:
  + Cost of scalable algorithm grows slowly as input size and the number of processors are increased
  + Level of ‘control parallelism’ is usually a constant independent of problem size
  + Level of ‘data parallelism’ is an increasing function of problem size
  + Data parallel algorithms are more scalable than control parallel algorithms

## Goals in Designing Parallel Algorithms

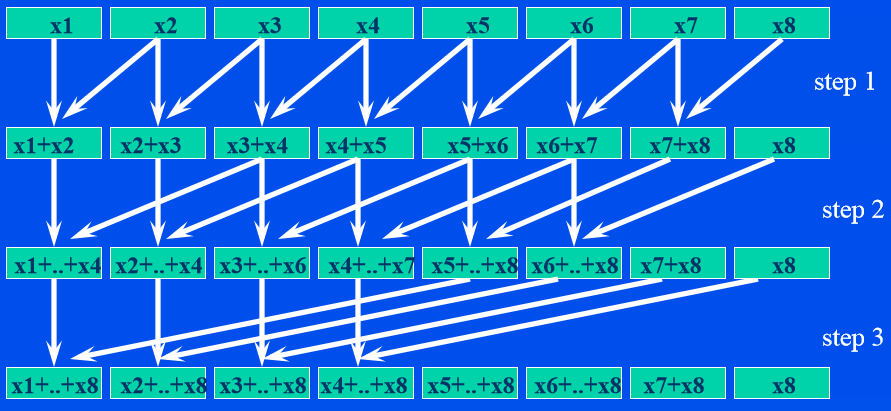
* Scalability:
  + Algorithm cost grows slowly, preferably in a polylogarithmic manner
* Work Efficient:
  + We do not want to waste CPU cycles
  + May be an important point when we are worried about power consumption or ‘money’ paid for CPU usage

## Summing N numbers in Parallel



* Array of N numbers can be summed in log(N) steps using N/2 processor

## Prefix Summing N numbers in Parallel



* Computing partial sums of an array of N numbers can be done in log (N) steps using N processors.

## Prefix Paradigm for Parallel Algorithm Design

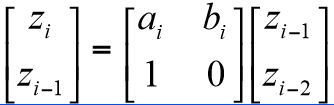
* Prefix computation forms a paradigm for parallel algorithm development, just like other well-known paradigms such as:
  + Divide and conquer, dynamic programming, etc.
* Prefix Paradigm:
  + If possible, transform your problem to prefix type computation
  + Apply the efficient logarithmic prefix computation
* Examples of Problems solved by Prefix Paradigm:
  + Solving linear recurrence equations
  + Tridiagonal Solver
  + Problems on trees
  + Adaptive triangular mesh refinement

## Solving Linear Recurrence Equations

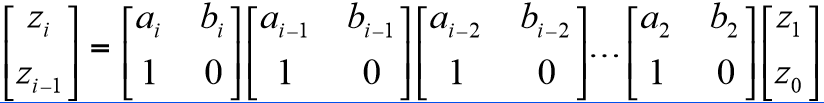
* Given the linear recurrence equation:



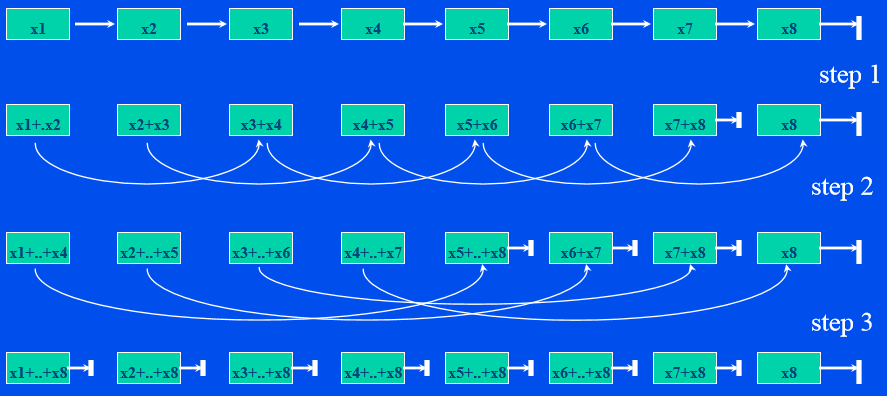
* we can rewrite it as:



* if we expand it, we get the solution in terms of partial products of coefficients and the initial values z1 and z0 :

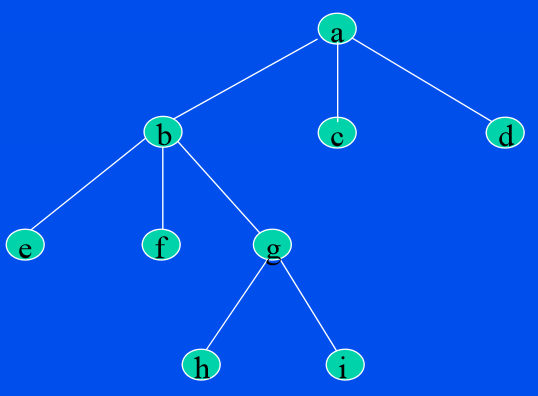


## Pointer Jumping Technique



* A linked list of N numbers can be prefix-summed in log (N) steps using N processors

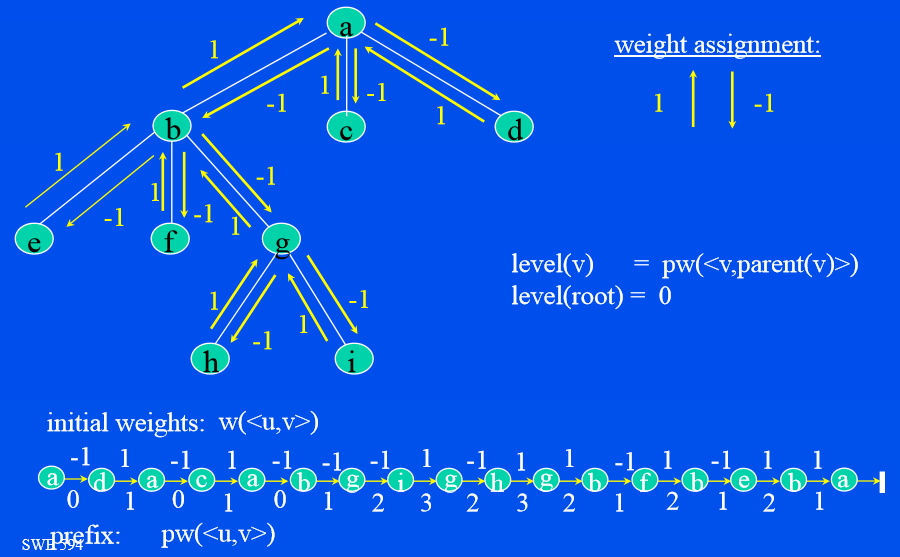
## Euler Tour Technique

**Tree Problems:**

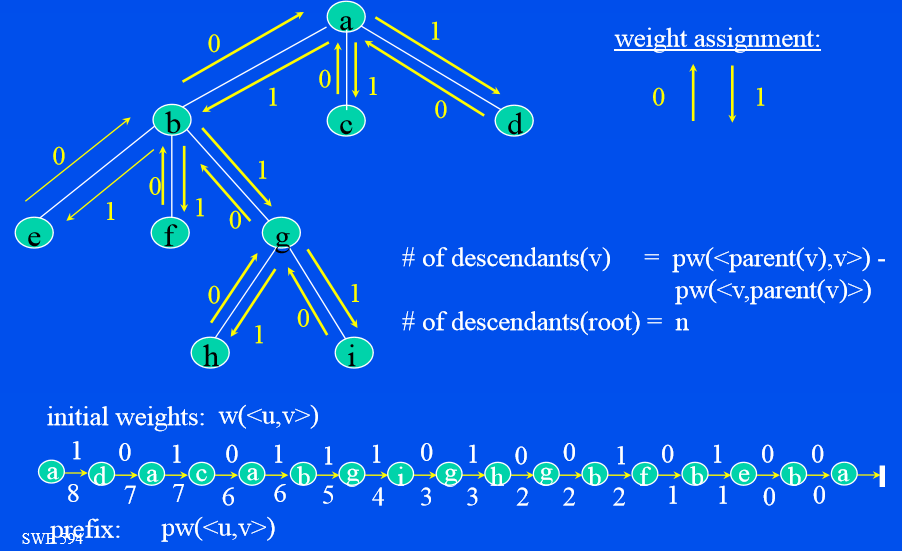
* Preorder numbering
* Post-order numbering
* Number of Descendants
* Level of each node

To solve such problems, first transform the tree by linearizing it into a linked-list and then apply the prefix computation

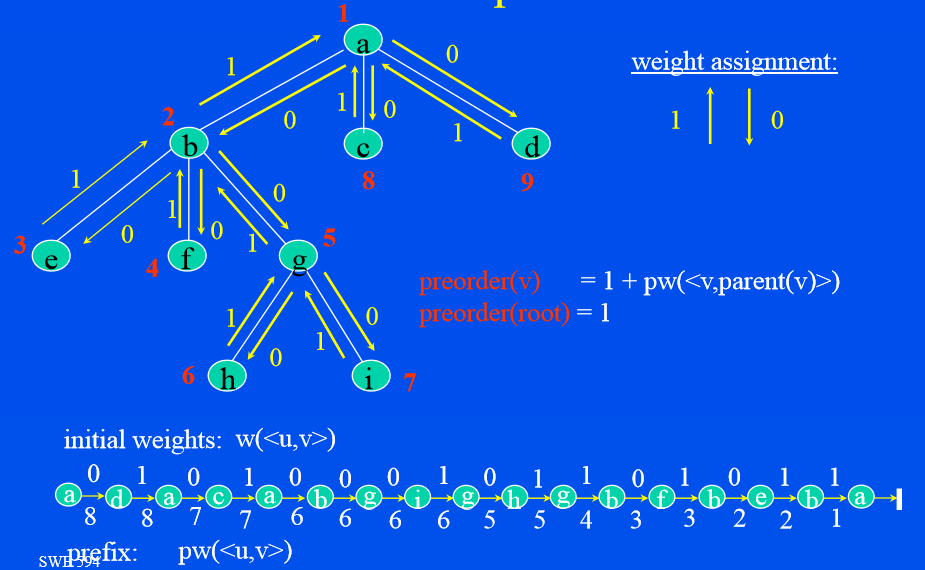
## Computing Level of Each Node by Euler Tour Technique



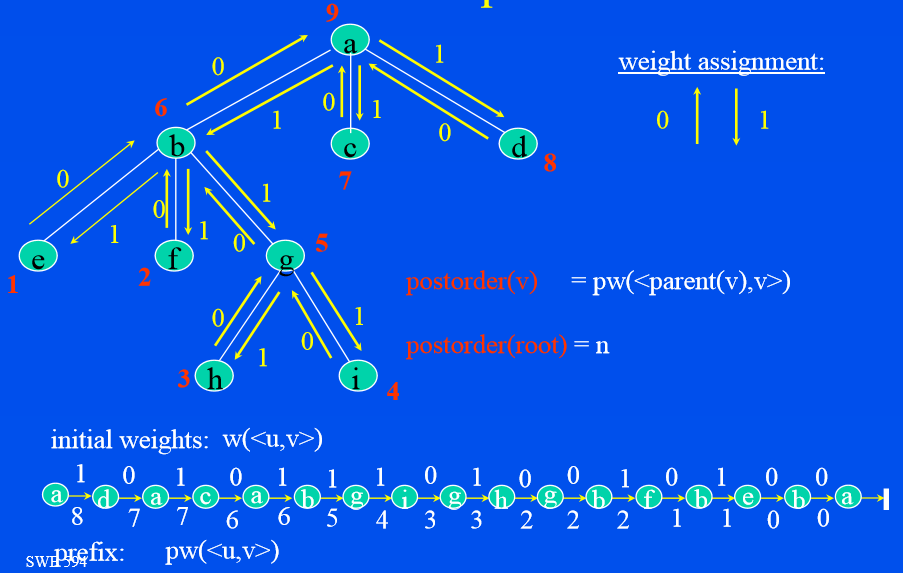
## Computing Number of Descendants by Euler Tour Technique



## Preorder Numbering by Euler Tour Technique



## Post-order Numbering by Euler Tour Technique



## Binary Tree Traversal

* Preorder
* In-order
* Post-order

## Brent’s Theorem

Given a parallel algorithm with computation time D, if parallel algorithm performs W operations then P processors can execute the algorithm in time D + (W-D)/P

For proof: consider DAG representation of computation

## Work Efficiency

* Parallel Summation
* Parallel Prefix Summation

# OpenMP

## OpenMP is …

* An Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism
* Comprised of three primary API components:
  + Compiler Directives
  + Runtime Library Routines
  + Environment Variables
* An abbreviation for:
  + Short version: Open Multi-Processing
  + Long version: Open specifications for Multi-Processing via collaborative work between interested parties from the hardware and software industry, government and academia.

## OpenMP is not …

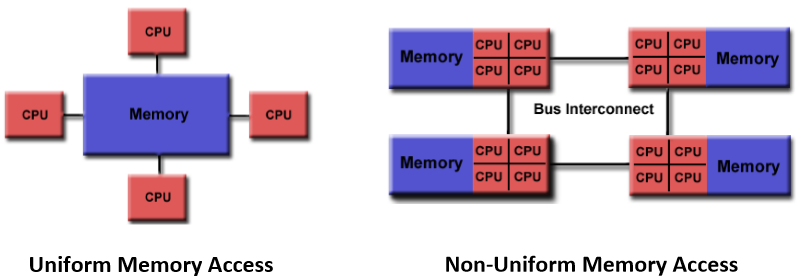
* Meant for distributed memory parallel systems (by itself)
* Necessarily implemented identically by all vendors
* Guaranteed to make the most efficient use of shared memory
* Required to check for data dependencies, data conflicts, race conditions, or deadlocks
* Required to check for code sequences that cause a program to be classified as non-conforming
* Meant to cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization
* Designed to guarantee that input or output to the same file is synchronous when executed in parallel. The programmer is responsible for synchronizing input and output.

## Goals of OpenMP

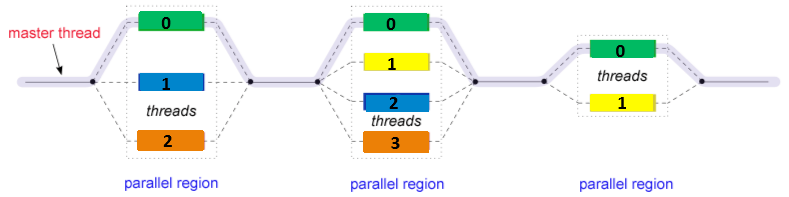
* Standardization:
  + Provide a standard among a variety of shared memory architectures and platforms
  + Jointly defined and endorsed by a group of major computer hardware and software vendors
* Lean and Mean:
  + Establish a simple and limited set of directives for programming shared memory machines.
  + Significant parallelism can be implemented by using just 3 or 4 directives.
  + This goal is becoming less meaningful with each new release, apparently.
* Ease of Use:
  + Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
  + Provide the capability to implement both coarse-grain and fine-grain parallelism
* Portability:
  + The API is specified for C/C++ and Fortran
  + Public forum for API and membership
  + Most major platforms have been implemented including Unix/Linux platforms and Window

## OpenMP Programming Model

* **Shared Memory Model:** 
  + OpenMP is designed for multi-processor/core, shared memory machines.
  + The underlying architecture can be shared memory UMA or NUMA.



* **Thread Based Parallelism:**
  + OpenMP programs accomplish parallelism exclusively through the use of threads.
  + A thread of execution is the smallest unit of processing that can be scheduled by an operating system. The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is.
  + Threads exist within the resources of a single process. Without the process, they cease to exist.
  + Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application.
* **Explicit Parallelism:** 
  + OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
  + Parallelization can be as simple as taking a serial program and inserting compiler directives....
  + Or as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.
* **Fork - Join Model:**



* + All OpenMP programs begin as a single process: the master thread.
  + The master thread executes sequentially until the first parallel region construct is encountered.
  + FORK: the master thread then creates a team of parallel threads. The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
  + JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.
  + The number of parallel regions and the threads that comprise them are arbitrary.
* **Compiler Directive Based:** 
  + Most OpenMP parallelism is specified through the use of compiler directives which are imbedded in C/C++ or FORTRAN source code.
* **Nested Parallelism:** 
  + The API provides for the placement of parallel regions inside other parallel regions.
  + Implementations may or may not support this feature.
* **Dynamic Threads:** 
  + The API provides for the runtime environment to dynamically alter the number of threads used to execute parallel regions. Intended to promote more efficient use of resources, if possible.
  + Implementations may or may not support this feature.
* **I/O:** 
  + OpenMP specifies nothing about parallel I/O. This is particularly important if multiple threads attempt to write/read from the same file.
  + If every thread conducts I/O to a different file, the issues are not as significant.
  + It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.
* **Memory Model: FLUSH Often?** 
  + OpenMP provides a "relaxed-consistency" and "temporary" view of thread memory (in their words). In other words, threads can "cache" their data and are not required to maintain exact consistency with real memory all of the time.
  + When it is critical that all threads view a shared variable identically, the programmer is responsible for insuring that the variable is flushed by all threads as needed.

## OpenMP API Overview

**Three Components:**

* The OpenMP API is comprised of three distinct components.

As of version 3.1:

(1) Compiler Directives (20)

(2) Runtime Library Routines (32)

(3) Environment Variables (9)

* The application developer decides how to employ these components. In the simplest case, only a few of them are needed.
* Implementations differ in their support of all API components. For example, an implementation may state that it supports nested parallelism, but the API makes it clear that may be limited to a single thread - the master thread. Not exactly what the developer might expect?

## Compiler Directives

* Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise – usually by specifying the appropriate compiler flag
* OpenMP compiler directives are used for various purposes:
  + Spawning a parallel region
  + Dividing blocks of code among threads
  + Distributing loop iterations between threads
  + Serializing sections of code
  + Synchronization of work among threads
* Compiler directives have the following syntax
* ***sentinel directive-name [clause, ...]***
* For example:



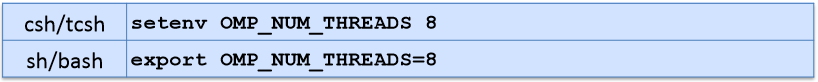
## Run-time Library Routines

* The OpenMP API includes an ever-growing number of run- time library routines.
* These routines are used for a variety of purposes:
  + Setting and querying the number of threads
  + Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  + Setting and querying the dynamic threads feature
  + Querying if in a parallel region, and at what level
  + Setting and querying nested parallelism
  + Setting, initializing and terminating locks and nested locks
  + Querying wall clock time and resolution
* Note that for C/C++, you usually need to include the **<omp.h>** header file.

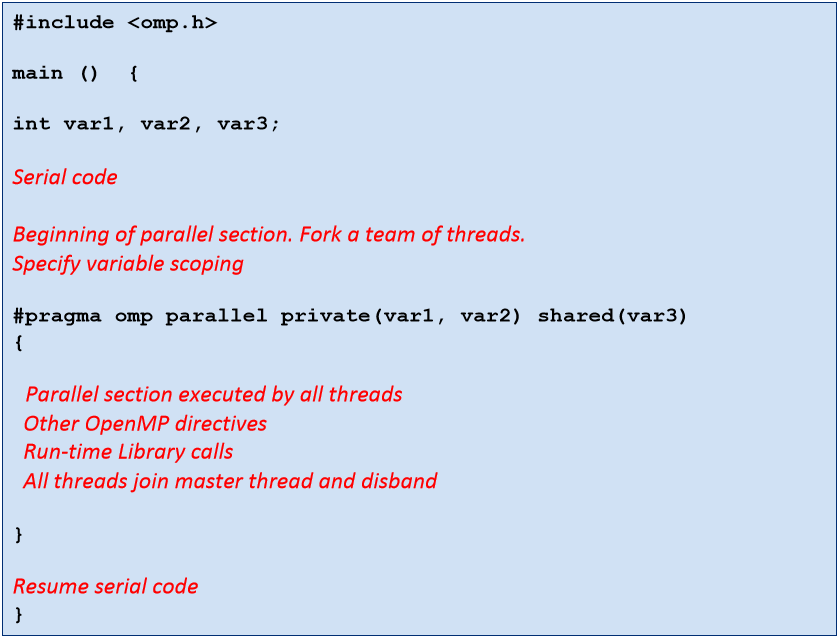


## Environment Variables

* OpenMP provides several environment variables for controlling the execution of parallel code at run-time.
* These environment variables can be used to control such things as:
  + Setting the number of threads
  + Specifying how loop interactions are divided
  + Binding threads to processors
  + Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
  + Enabling/disabling dynamic threads
  + Setting thread stack size ‒ Setting thread wait policy
* Setting OpenMP environment variables is done the same way you set any other environment variables, and depends upon which shell you use. For example:



## C / C++ - General Code Structure



## Compiling OpenMP Programs

|  |  |  |
| --- | --- | --- |
| Compiler | Command | Flag |
| Intel | icc  icpc  ifort | -openmp |
| GNU\* | gcc  g++  g77  gfortran | -fopenmp |

\* If GCC is used to compile C programs, the flag “-march=cpu-type” can be used to optimize compilation to your type of processor and it’s a set of instructions

## Getting CPU Information on Linux

* The “proc” file system is providing most of the important information on the system.
* It interfaces with the kernel to obtain these information and is mounted as “/proc” on most of the UNIX machines.
* Useful Commands:
  + /proc/cpuinfo
  + grep flags /proc/cpuinfo
  + lscpu

## OpenMP Directives Format

**General Rules:**

* Case sensitive
* Directives follow conventions of the C/C++ standards for compiler directives
* Only one directive-name may be specified per directive
* Each directive applies to at most one succeeding statement, which must be a structured block.
* Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.

|  |  |  |  |
| --- | --- | --- | --- |
| #pragma omp | Directive-name | [clause, …] | newline |
| Required for all OpenMP C/C++ directives. | A valid OpenMP directive. Must appear after the pragma and before any clauses. | Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted. | Required. Precedes the structured block which is enclosed by this directive. |

**Example:**

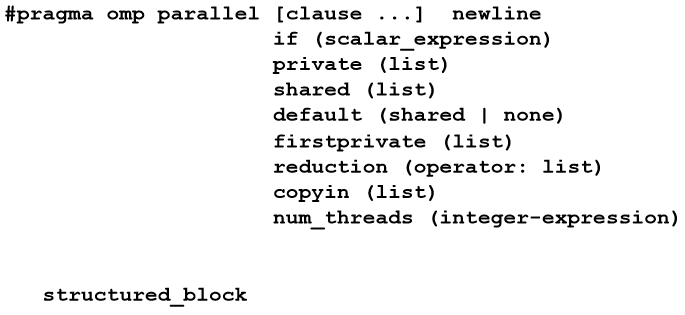


## Directive Scoping

* **Static (Lexical) Extent:** 
  + The code textually enclosed between the beginning and the end of a structured block following a directive.
  + The static extent of a directives does not span multiple routines or code files
* **Orphaned Directive:** 
  + An OpenMP directive that appears independently from another enclosing directive is said to be an orphaned directive.
  + It exists outside of another directive's static (lexical) extent.
  + Will span routines and possibly code files
* **Dynamic Extent:** 
  + The dynamic extent of a directive includes both its static (lexical) extent
  + And the extents of its orphaned directives.

## PARALLEL Region Construct

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



* When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team.
* The master is a member of that team and has thread number 0 within that team.
* Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
* There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.
* If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.

**How Many Threads?**

* The number of threads in a parallel region is determined by the following factors, in order of precedence:
  1. Evaluation of the **IF** clause
  2. Setting of the **NUM\_THREADS** clause
  3. Use of the **omp\_set\_num\_threads()** library function
  4. Setting of the **OMP\_NUM\_THREADS** environment variable
  5. Implementation default - usually the number of CPUs on a node, though it could be dynamic (see next bullet).
* Threads are numbered from 0 (master thread) to N-1

**Dynamic Threads:**

* Use the **omp\_get\_dynamic()** library function to determine if dynamic threads are enabled.
* If supported, the two methods available for enabling dynamic threads are:
  1. The **omp\_set\_dynamic()** library routine
  2. Setting of the **OMP\_DYNAMIC** environment variable to TRUE

**Clauses:**

* **IF** clause: If present, it must evaluate to non-zero (C/C++) in order for a team of threads to be created. Otherwise, the region is executed serially by the master thread.

**Restrictions:**

* A parallel region must be a structured block that does not span multiple routines or code files
* It is illegal to branch (goto) into or out of a parallel region
* Only a single IF clause is permitted
* Only a single NUM\_THREADS clause is permitted

## PRIVATE Clause

**Purpose:**

* The PRIVATE clause declares variables in its list to be private to each thread.

**Notes:**

* PRIVATE variables behave as follows:
  + A new object of the same type is declared once for each thread in the team
  + All references to the original object are replaced with references to the new object
  + Variables declared PRIVATE should be assumed to be uninitialized for each thread
* Comparison between PRIVATE and THREADPRIVATE:

|  |  |  |
| --- | --- | --- |
|  | PRIVATE | THREADPRIVATE |
| Data Item | C/C++: variable  Fortran: variable or common block | C/C++: variable  Fortran: common block |
| Where Declared | At start of region or work-sharing group | In declarations of each routine using block or global file scope |
| Persistent? | No | Yes |
| Extent | Lexical only - unless passed as an argument to subroutine | Dynamic |
| Initialized | Use FIRSTPRIVATE | Use COPYIN |

## SHARED Clause

**Purpose:**

* The SHARED clause declares variables in its list to be shared among all threads in the team.

**Notes:**

* A shared variable exists in only one memory location and all threads can read or write to that address
* It is the programmer's responsibility to ensure that multiple threads properly access SHARED variables (such as via CRITICAL sections)

## DEFAULT Clause

**Purpose:**

* The DEFAULT clause allows the user to specify a default scope for all variables in the lexical extent of any parallel region.

**Notes:**

* Specific variables can be exempted from the default using the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses
* The C/C++ OpenMP specification does not include private or firstprivate as a possible default. However, actual implementations may provide this option.
* Using NONE as a default requires that the programmer explicitly scope all variables.

**Restrictions:**

* Only one DEFAULT clause can be specified on a PARALLEL directive

## FIRSTPRIVATE Clause

**Purpose:**

* The FIRSTPRIVATE clause combines the behavior of the PRIVATE clause with automatic initialization of the variables in its list.

**Notes:**

* Listed variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct.

## LASTPRIVATE Clause

**Purpose:**

* The LASTPRIVATE clause combines the behavior of the PRIVATE clause with a copy from the last loop iteration or section to the original variable object.

**Notes:**

* The value copied back into the original variable object is obtained from the last (sequentially) iteration or section of the enclosing construct.

For example, the team member which executes the final iteration for a DO section, or the team member which does the last SECTION of a SECTIONS context performs the copy with its own values

## COPYIN Clause

**Purpose:**

* The COPYIN clause provides a means for assigning the same value to THREADPRIVATE variables for all threads in the team.

**Notes:**

* List contains the names of variables to copy. In FORTRAN, the list can contain both the names of common blocks and named variables.
* The master thread variable is used as the copy source. The team threads are initialized with its value upon entry into the parallel construct.

## COPYPRIVATE Clause

**Purpose:**

* The COPYPRIVATE clause can be used to broadcast values acquired by a single thread directly to all instances of the private variables in the other threads.
* Associated with the SINGLE directive
* See the most recent OpenMP specs document for additional discussion and examples.

## REDUCTION Clause

**Purpose:**

* The REDUCTION clause performs a reduction operation on the variables that appear in its list.
* A private copy for each list variable is created and initialized for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

|  |  |  |  |
| --- | --- | --- | --- |
| Valid Operators and Initialization Values | | | |
| Operation | **Fortran** | **C/C++** | **Initialization** |
| Addition | + | + | 0 |
| Multiplication | \* | \* | 1 |
| Subtraction | - | - | 0 |
| Logical AND | .and. | && | 0 |
| Logical OR | .or. | || | .false. / 0 |
| AND bitwise | iand | & | all bits on / 1 |
| OR bitwise | ior | | | 0 |
| Exclusive OR bitwise | ieor | ^ | 0 |
| Equivalent | .eqv. |  | .true. |
| Not Equivalent | .neqv. |  | .false. |
| Maximum | max | max | Most negative # |
| Minimum | min | min | Largest positive # |

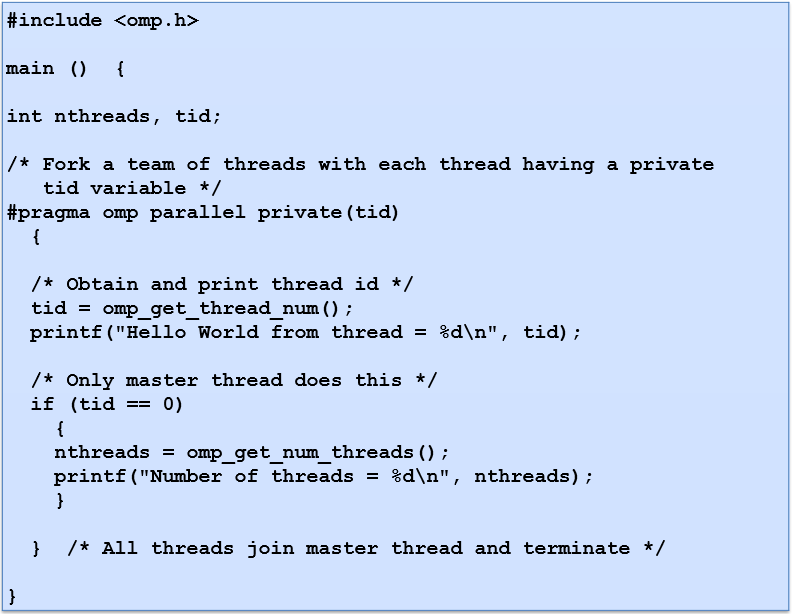
## Clauses / Directives Summary

* The table below summarizes which clauses are accepted by which OpenMP directives.

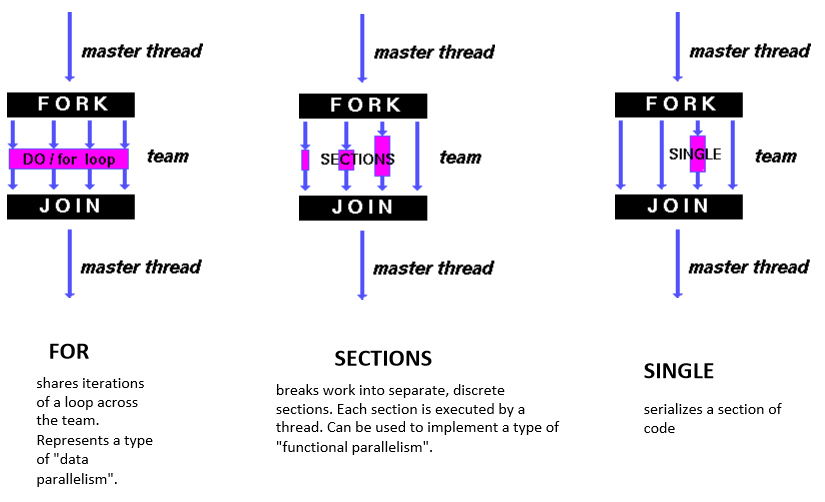
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Clause | Directive | | | | | |
| **PARALLEL** | **DO/for** | **SECTIONS** | **SINGLE** | **PARALLEL DO/for** | **PARALLEL SECTIONS** |
| IF | https://computing.llnl.gov/tutorials/images/ball_red3.png |  |  |  | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png |
| PRIVATE | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png |
| SHARED | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png |  |  | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png |
| DEFAULT | https://computing.llnl.gov/tutorials/images/ball_red3.png |  |  |  | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png |
| FIRSTPRIVATE | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png |
| LASTPRIVATE |  | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png |  | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png |
| REDUCTION | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png |  | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png |
| COPYIN | https://computing.llnl.gov/tutorials/images/ball_red3.png |  |  |  | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png |
| COPYPRIVATE |  |  |  | https://computing.llnl.gov/tutorials/images/ball_red3.png |  |  |
| SCHEDULE |  | https://computing.llnl.gov/tutorials/images/ball_red3.png |  |  | https://computing.llnl.gov/tutorials/images/ball_red3.png |  |
| ORDERED |  | https://computing.llnl.gov/tutorials/images/ball_red3.png |  |  | https://computing.llnl.gov/tutorials/images/ball_red3.png |  |
| NOWAIT |  | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png | https://computing.llnl.gov/tutorials/images/ball_red3.png |  |  |

* The following OpenMP directives do not accept clauses:
  + MASTER
  + CRITICAL
  + BARRIER
  + ATOMIC
  + FLUSH
  + ORDERED
  + THREADPRIVATE
* Implementations may (and do) differ from the standard in which clauses are supported by each directive.

## PARALLEL Region Example

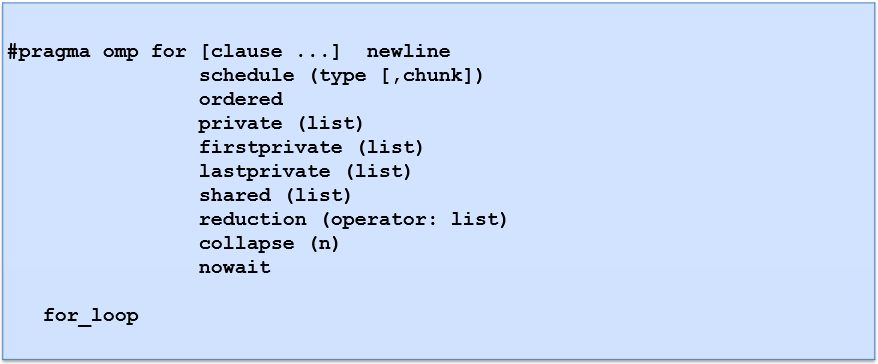


## Work-Sharing Constructs



* A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
* Work-sharing constructs must be encountered by all members of a team or none at all
* Successive work-sharing constructs must be encountered in the same order by all members of a team

## FOR Work Sharing Directive



**Purpose:**

* The DO / for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.

**Clauses:**

* **SCHEDULE:** Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent. For a discussion on how one type of scheduling may be more optimal than others, see <http://openmp.org/forum/viewtopic.php?f=3&t=83>.
  + **STATIC:** Loop iterations are divided into pieces of size chunk and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.
  + **DYNAMIC:** Loop iterations are divided into pieces of size chunk, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.
  + **GUIDED:** Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread. The size of the initial block is proportional to: *number\_of\_iterations / number\_of\_threads*,

Subsequent blocks are proportional to

*Number\_of\_iterations\_remaining / number\_of\_threads*

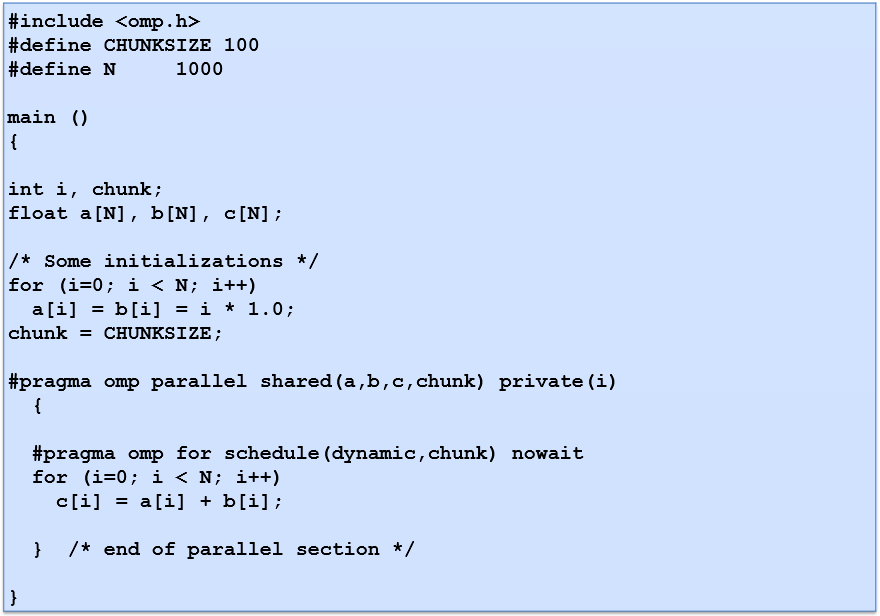
The chunk parameter defines the minimum block size. The default chunk size is 1.

* + **RUNTIME:** The scheduling decision is deferred until runtime by the environment variable OMP\_SCHEDULE. It is illegal to specify a chunk size for this clause.
  + **AUTO:** The scheduling decision is delegated to the compiler and/or runtime system.
* **NO WAIT / nowait:** If specified, then threads do not synchronize at the end of the parallel loop.
* **ORDERED:** Specifies that the iterations of the loop must be executed as they would be in a serial program.
* **COLLAPSE:** Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.
* Other clauses are described in detail later, in the [Data Scope Attribute Clauses](https://computing.llnl.gov/tutorials/openMP/#Clauses) section.

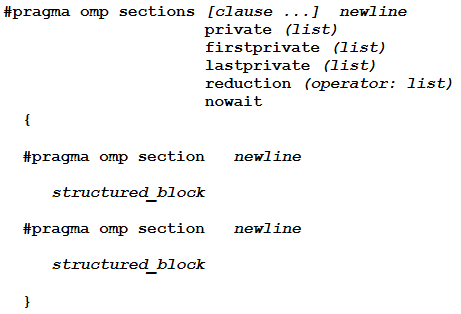
**Restrictions:**

* The DO loop cannot be a DO WHILE loop, or a loop without loop control. Also, the loop iteration variable must be an integer and the loop control parameters must be the same for all threads.
* Program correctness must not depend upon which thread executes a particular iteration.
* It is illegal to branch (go to) out of a loop associated with a DO/for directive.
* The chunk size must be specified as a loop invariant integer expression, as there is no synchronization during its evaluation by different threads.
* ORDERED, COLLAPSE and SCHEDULE clauses may appear once each.
* See the OpenMP specification document for additional restrictions.

**Example:**



## SECTIONS Work Sharing Directive



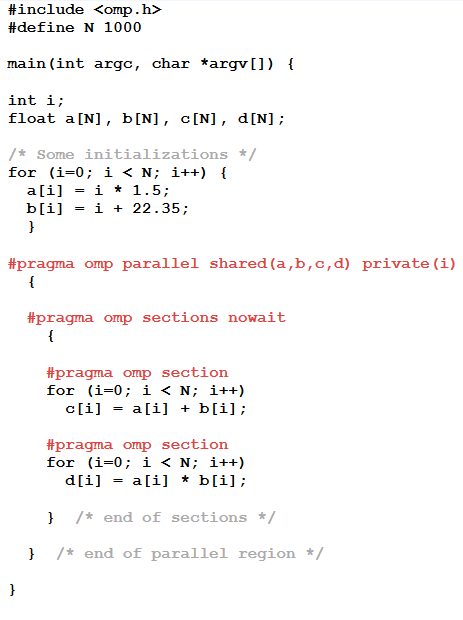
**Purpose:**

* The SECTIONS directive is a non-iterative work-sharing construct. It specifies that the enclosed section(s) of code are to be divided among the threads in the team.
* Independent SECTION directives are nested within a SECTIONS directive. Each SECTION is executed once by a thread in the team. Different sections may be executed by different threads. It is possible for a thread to execute more than one section if it is quick enough and the implementation permits such.

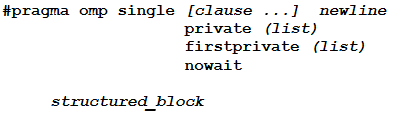
**Clauses:**

* There is an implied barrier at the end of a SECTIONS directive, unless the NOWAIT/nowait clause is used.

**Example:**



## SINGLE Work Sharing Directive



**Purpose:**

* The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team.
* May be useful when dealing with sections of code that are not thread safe (such as I/O)

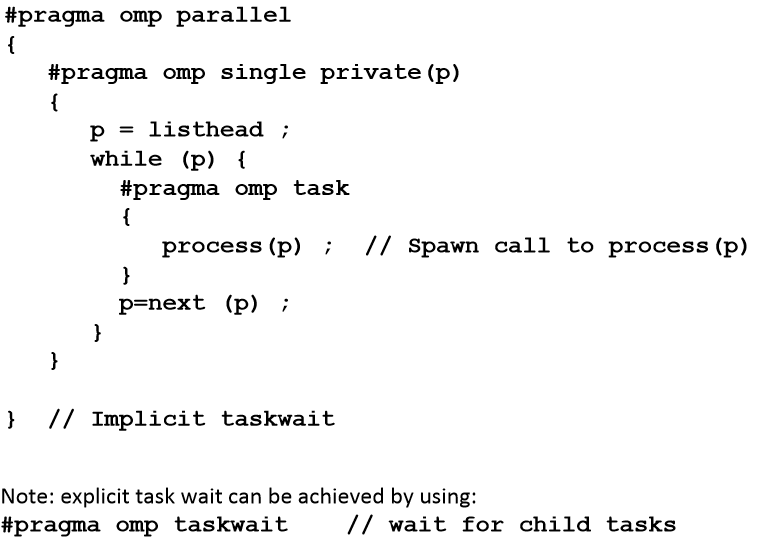
**Clauses:**

* Threads in the team that do not execute the SINGLE directive, wait at the end of the enclosed code block, unless a NOWAIT/nowait clause is specified.
* Clauses are described in detail later, in the [Data Scope Attribute Clauses](https://computing.llnl.gov/tutorials/openMP/#Clauses) section.

**Restrictions:**

* It is illegal to branch into or out of a SINGLE block.

## Tasks in OpenMP 3.0

s

## OMP\_SET\_NUM\_THREADS

**Purpose:**

* Sets the number of threads that will be used in the next parallel region. Must be a postive integer.

**Format:** void omp\_set\_num\_threads(int num\_threads)

**Notes & Restrictions:**

* The dynamic threads mechanism modifies the effect of this routine.
  + Enabled: specifies the maximum number of threads that can be used for any parallel region by the dynamic threads mechanism.
  + Disabled: specifies exact number of threads to use until next call to this routine.
* This routine can only be called from the serial portions of the code
* This call has precedence over the OMP\_NUM\_THREADS environment variable

## OMP\_GET\_NUM\_THREADS

**Purpose:**

* Returns the number of threads that are currently in the team executing the parallel region from which it is called.

Format: int omp\_get\_num\_threads(void)

**Notes & Restrictions:**

* If this call is made from a serial portion of the program, or a nested parallel region that is serialized, it will return 1.
* The default number of threads is implementation dependent.

## OMP\_GET\_MAX\_THREADS

**Purpose:**

* Returns the maximum value that can be returned by a call to the OMP\_GET\_NUM\_THREADS function.

**Format:** int omp\_get\_max\_threads(void)

**Notes & Restrictions:**

* Generally reflects the number of threads as set by the OMP\_NUM\_THREADS environment variable or the OMP\_SET\_NUM\_THREADS() library routine.
* May be called from both serial and parallel regions of code.

## OMP\_GET\_THREAD\_NUM

**Purpose:**

* Returns the thread number of the thread, within the team, making this call. This number will be between 0 and OMP\_GET\_NUM\_THREADS-1. The master thread of the team is thread 0

Format: int omp\_get\_thread\_num(void)

**Notes & Restrictions:**

* If called from a nested parallel region, or a serial region, this function will return 0.

## OMP\_GET\_THREAD\_LIMIT

**Purpose:**

* Returns the maximum number of OpenMP threads available to a program.

Format: int omp\_get\_thread\_limit (void)

**Notes:**

* Also see the **OMP\_THREAD\_LIMIT** environment variable.

## OMP\_GET\_NUM\_PROCS

**Purpose:**

* Returns the number of processors that are available to the program.

Format: int omp\_get\_num\_procs(void)

## OMP\_IN\_PARALLEL

**Purpose:**

* May be called to determine if the section of code which is executing is parallel or not.

Format: int omp\_in\_parallel(void)

**Notes & Restrictions:**

* For FORTRAN, this function returns .TRUE. if it is called from the dynamic extent of a region executing in parallel, and .FALSE. otherwise. For C/C++, it will return a non-zero integer if parallel, and zero otherwise.

## OMP\_SET\_DYNAMIC

**Purpose:**

* Enables or disables dynamic adjustment (by the run time system) of the number of threads available for execution of parallel regions.

**Format:** void omp\_set\_dynamic(int dynamic\_threads)

**Notes & Restrictions:**

* For FORTRAN, if called with .TRUE. then the number of threads available for subsequent parallel regions can be adjusted automatically by the run-time environment. If called with .FALSE., dynamic adjustment is disabled.
* For C/C++, if dynamic\_threads evaluates to non-zero, then the mechanism is enabled, otherwise it is disabled.
* The OMP\_SET\_DYNAMIC subroutine has precedence over the OMP\_DYNAMIC environment variable.
* The default setting is implementation dependent.
* Must be called from a serial section of the program.

## OMP\_GET\_DYNAMIC

**Purpose:**

* Used to determine if dynamic thread adjustment is enabled or not.

**Format:** int omp\_get\_dynamic(void)

**Notes & Restrictions:**

* For Fortran, this function returns .TRUE. if dynamic thread adjustment is enabled, and .FALSE. otherwise.
* For C/C++, non-zero will be returned if dynamic thread adjustment is enabled, and zero otherwise.

## OMP\_SET\_NESTED

**Purpose:**

* Used to enable or disable nested parallelism.

**Format:** void omp\_set\_nested(int nested)

**Notes & Restrictions:**

* For Fortran, calling this function with .FALSE. will disable nested parallelism, and calling with .TRUE. will enable it.
* For C/C++, if nested evaluates to non-zero, nested parallelism is enabled; otherwise it is disabled.
* The default is for nested parallelism to be disabled.
* This call has precedence over the OMP\_NESTED environment variable

## OMP\_GET\_NESTED

**Purpose:**

* Used to determine if nested parallelism is enabled or not.

**Format:** int omp\_get\_nested (void)

**Notes & Restrictions:**

* For Fortran, this function returns .TRUE. if nested parallelism is enabled, and .FALSE. otherwise.
* For C/C++, non-zero will be returned if nested parallelism is enabled, and zero otherwise.

## OMP\_SET\_SCHEDULE

**Purpose:**

* This routine sets the schedule type that is applied when the loop directive specifies a runtime schedule.

**Format:** void omp\_set\_schedule(omp\_sched\_t kind, int modifier)

## OMP\_GET\_SCHEDULE

**Purpose:**

* This routine returns the schedule that is applied when the loop directive specifies a runtime schedule.

**Format:** void omp\_get\_schedule(omp\_sched\_t \* kind, int \* modifier )

## OMP\_SET\_MAX\_ACTIVE\_LEVELS

**Purpose:**

* This routine limits the number of nested active parallel regions.

**Format:** void omp\_set\_max\_active\_levels (int max\_levels)

**Notes & Restrictions:**

* If the number of parallel levels requested exceeds the number of levels of parallelism supported by the implementation, the value will be set to the number of parallel levels supported by the implementation.
* This routine has the described effect only when called from the sequential part of the program. When called from within an explicit parallel region, the effect of this routine is implementation defined.

## OMP\_GET\_MAX\_ACTIVE\_LEVELS

**Purpose:**

* This routine returns the maximum number of nested active parallel regions.

**Format:** int omp\_get\_max\_active\_levels(void)

## OMP\_GET\_LEVEL

**Purpose:**

* This routine returns the number of nested parallel regions enclosing the task that contains the call.

**Format:** int omp\_get\_level(void)

**Notes & Restrictions:**

* The omp\_get\_level routine returns the number of nested parallel regions (whether active or inactive) enclosing the task that contains the call, not including the implicit parallel region. The routine always returns a non-negative integer, and returns 0 if it is called from the sequential part of the program.

## OMP\_GET\_ANCESTOR\_THREAD\_NUM

**Purpose:**

* This routine returns, for a given nested level of the current thread, the thread number of the ancestor or the current thread.

**Format:** int omp\_get\_ancestor\_thread\_num(int level)

**Notes & Restrictions:**

* If the requested nest level is outside the range of 0 and the nest level of the current thread, as returned by the omp\_get\_level routine, the routine returns -1.

## OMP\_GET\_TEAM\_SIZE

**Purpose:**

* This routine returns, for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs.

**Format:** int omp\_get\_team\_size(int level);

**Notes & Restrictions:**

* If the requested nested level is outside the range of 0 and the nested level of the current thread, as returned by the omp\_get\_level routine, the routine returns -1. Inactive parallel regions are regarded like active parallel regions executed with one thread.

## OMP\_GET\_ACTIVE\_LEVEL

**Purpose:**

* The omp\_get\_active\_level routine returns the number of nested, active parallel regions enclosing the task that contains the call.

**Format:** int omp\_get\_active\_level(void);

**Notes & Restrictions:**

* The routine always returns a nonnegative integer, and returns 0 if it is called from the sequential part of the program.

## OMP\_IN\_FINAL

**Purpose:**

* This routine returns true if the routine is executed in a final task region; otherwise, it returns false.

**Format:** int omp\_in\_final(void)

## OMP\_INIT\_LOCK (OMP\_INIT\_NEST\_LOCK)

**Purpose:**

* This subroutine initializes a lock associated with the lock variable.

Format: void omp\_init\_lock(omp\_lock\_t \*lock)

void omp\_init\_nest\_lock(omp\_nest\_lock\_t \*lock)

**Notes & Restrictions:**

* The initial state is unlocked
* For Fortran, var must be an integer large enough to hold an address, such as INTEGER\*8 on 64-bit systems.

## OMP\_DESTROY\_LOCK (OMP\_DESTROY\_NEST\_LOCK)

**Purpose:**

* This subroutine disassociates the given lock variable from any locks.

Format: void omp\_destroy\_lock(omp\_lock\_t \*lock)

void omp\_destroy\_nest\_lock(omp\_nest\_lock\_t \*lock)

**Notes & Restrictions:**

* It is illegal to call this routine with a lock variable that is not initialized.
* For FORTRAN, var must be an integer large enough to hold an address, such as INTEGER\*8 on 64-bit systems.

## OMP\_SET\_LOCK (OMP\_SET\_NEST\_LOCK)

**Purpose:**

* This subroutine forces the executing thread to wait until the specified lock is available. A thread is granted ownership of a lock when it becomes available.

Format: void omp\_set\_lock(omp\_lock\_t \*lock)

void omp\_set\_nest\_\_lock(omp\_nest\_lock\_t \*lock)

**Notes & Restrictions:**

* It is illegal to call this routine with a lock variable that is not initialized.
* For FORTRAN, var must be an integer large enough to hold an address, such as INTEGER\*8 on 64-bit systems.

## OMP\_UNSET\_LOCK (OMP\_UNSET\_NEST\_LOCK)

**Purpose:**

* This subroutine releases the lock from the executing subroutine.

Format: void omp\_unset\_lock(omp\_lock\_t \*lock)

void omp\_unset\_nest\_\_lock(omp\_nest\_lock\_t \*lock)

**Notes & Restrictions:**

* It is illegal to call this routine with a lock variable that is not initialized.
* For Fortran, var must be an integer large enough to hold an address, such as INTEGER\*8 on 64-bit systems.

## OMP\_TEST\_LOCK (OMP\_TEST\_NEST\_LOCK)

**Purpose:**

* This subroutine attempts to set a lock, but does not block if the lock is unavailable.

Format: int omp\_test\_lock(omp\_lock\_t \*lock)

int omp\_test\_nest\_\_lock(omp\_nest\_lock\_t \*lock)

**Notes & Restrictions:**

* For FORTRAN, .TRUE. is returned if the lock was set successfully, otherwise .FALSE. is returned.
* For FORTRAN, var must be an integer large enough to hold an address, such as INTEGER\*8 on 64-bit systems.
* For C/C++, non-zero is returned if the lock was set successfully, otherwise zero is returned.
* It is illegal to call this routine with a lock variable that is not initialized.

## OMP\_GET\_WTIME

**Purpose:**

* Provides a portable wall clock timing routine
* Returns a double-precision floating point value equal to the number of elapsed seconds since some point in the past. Usually used in "pairs" with the value of the first call subtracted from the value of the second call to obtain the elapsed time for a block of code.
* Designed to be "per thread" times, and therefore may not be globally consistent across all threads in a team - depends upon what a thread is doing compared to other threads.

**Format:** double omp\_get\_wtime(void)

## OMP\_GET\_WTICK

**Purpose:**

* Provides a portable wall clock timing routine
* Returns a double-precision floating point value equal to the number of seconds between successive clock ticks.

**Format:** double omp\_get\_wtick(void)

# PTHREADS

## Pthreads API

* The Pthreads API is defined in the ANSI/IEEE POSIX 1003.1 - 1995 standard. The subroutines which comprise the Pthreads API can be informally grouped into three major classes:
* Naming conventions: All identifiers in the threads library begin with **pthread\_**

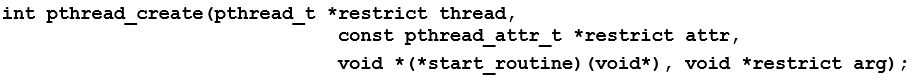
1. **Thread management:** Functions that work directly on threads - creating, detaching, joining, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)
2. **Mutexes:** Functions that deal with synchronization, Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
3. **Condition variables:** The third class of functions address communications between threads that share a mutex. They are based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

## Compiling Threaded Programs

|  |  |  |
| --- | --- | --- |
| Compiler/Platform | Compiler Command | Description |
| Intel  Linux | icc -pthread | C |
| icpc -pthread | C++ |
| GNU  Linux, AIX | gcc -pthread | GNU C |
| g++ -pthread | GNU C++ |

Include file: pthread.h

## Thread Creation



**pthread\_create arguments:**  thread: An opaque, unique identifier for the new thread returned by the subroutine.

**attr:** An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.

**start\_routine:** the C routine that the thread will execute once it is created.

**arg:** A single argument that may be passed to *start\_routine*. NULL may be used if no argument is to be passed.

## Terminating Threads

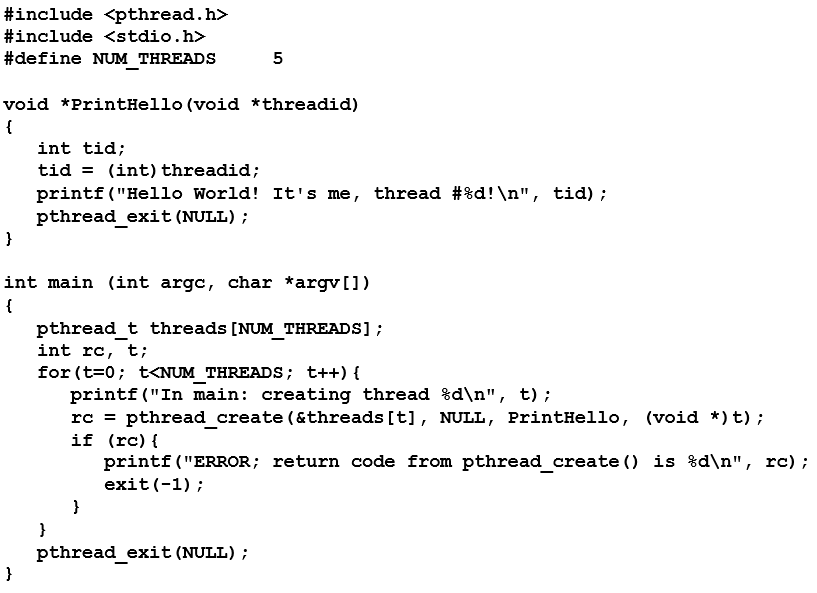
There are several ways in which a Pthread may be terminated:

* The thread returns from its starting routine (the main routine for the initial thread).
* The thread makes a call to the pthread\_exit subroutine.



* The thread is canceled by another thread via the **pthread\_cancel** routine, the entire process is terminated.
* Typically, the **pthread\_exit()** routine is called after a thread has completed its work and is no longer required to exist.
* If **main()** finishes before the threads it has created, and exits with **pthread\_exit()**, the other threads will continue to execute. Otherwise, they will be automatically terminated when **main()** finishes.
* The programmer may optionally specify a termination status, which is stored as a void pointer for any thread that may join the calling thread.
* Cleanup: the **pthread\_exit()** routine does not close files; any files opened inside the thread will remain open after the thread is terminated.

## Example



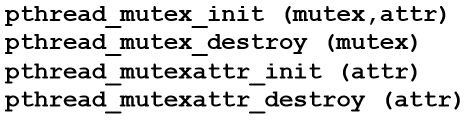
## Joining

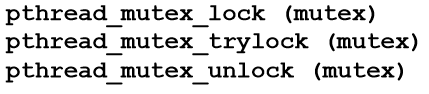


* The **pthread\_join()** subroutine blocks the calling thread until the specified **threadi**d thread terminates.
* The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to **pthread\_exit()**.
* A joining thread can match one **pthread\_join()** call. It is a logical error to attempt multiple joins on the same thread.

## Creating and Destroying Mutexes

* Mutex variables must be declared with type pthread\_mutex\_t, and must be initialized before they can be used. There are two ways to initialize a mutex variable:
  + Statically, when it is declared. For example: pthread\_mutex\_t mymutex = PTHREAD\_MUTEX\_INITIALIZER;
  + Dynamically, with the pthread\_mutex\_init() routine. This method permits setting mutex object attributes, attr.
* The mutex is initially unlocked.
* Routines:

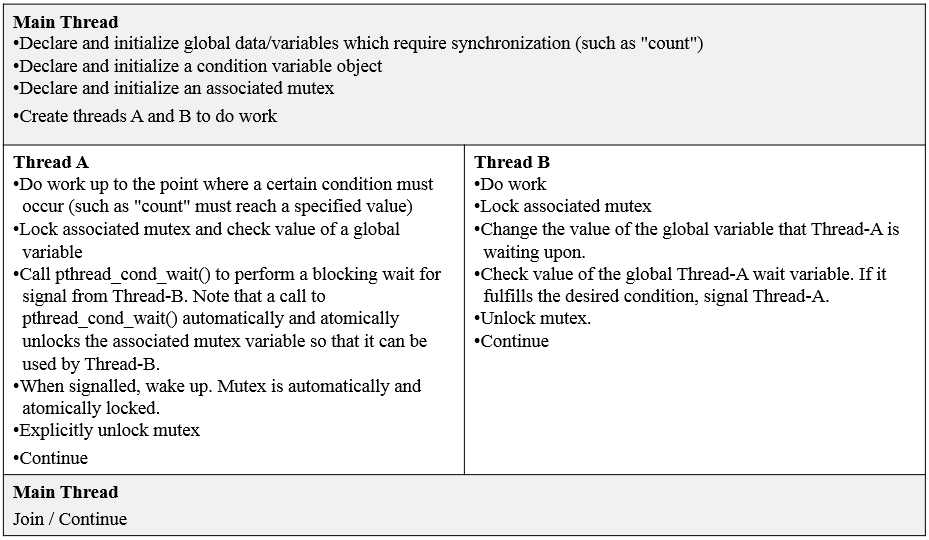




## Condition Variables

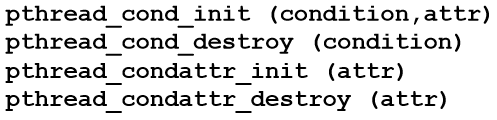
* Condition variables provide yet another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
* Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.
* A condition variable is always used in conjunction with a mutex lock.

## Condition Variables: Example of Usage



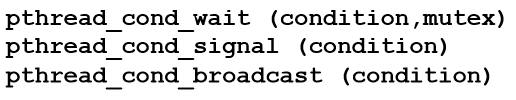
## Creating and Destroying Condition Variables

* Condition variables must be declared with type **pthread\_cond\_t**, and must be initialized before they can be used. There are two ways to initialize a condition variable:
  + Statically, when it is declared. For example: pthread\_cond\_t myconvar = PTHREAD\_COND\_INITIALIZER;
  + Dynamically, with the pthread\_cond\_init() routine. The ID of the created condition variable is returned to the calling thread through the condition parameter. This method permits setting condition variable object attributes, attr.



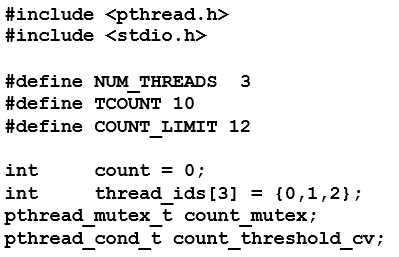
## Waiting and Signaling on Condition Variables

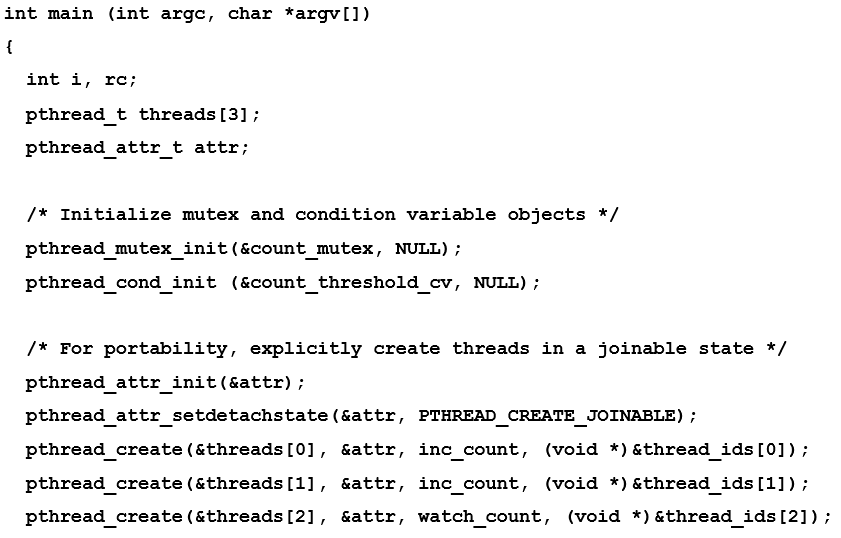
* pthread\_cond\_wait() blocks the calling thread until the specified condition is signalled. This routine should be called while mutex is locked, and it will automatically release the mutex while it waits. After signal is received and thread is awakened, mutex will be automatically locked for use by the thread. The programmer is then responsible for unlocking mutex when the thread is finished with it.
* The pthread\_cond\_signal() routine is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after mutex is locked, and must unlock mutex in order for pthread\_cond\_wait() routine to complete.
* The pthread\_cond\_broadcast() routine should be used instead of pthread\_cond\_signal() if more than one thread is in a blocking wait state.
* It is a logical error to call pthread\_cond\_signal() before calling pthread\_cond\_wait(). (Please note that the prog. does not crash if this happens - it is just that signal is not delivered to the thread, that’s all. So if your requirement is guaranteed delivery – then you have to take extra measures (put extra code) to make sure it is delivered).
* Proper locking and unlocking of the associated mutex variable is essential when using these routines. For example:
  + Failing to lock the mutex before calling pthread\_cond\_wait() may cause it NOT to block.
  + Failing to unlock the mutex after calling pthread\_cond\_signal() may not allow a matching pthread\_cond\_wait() routine to complete (it will remain blocked).

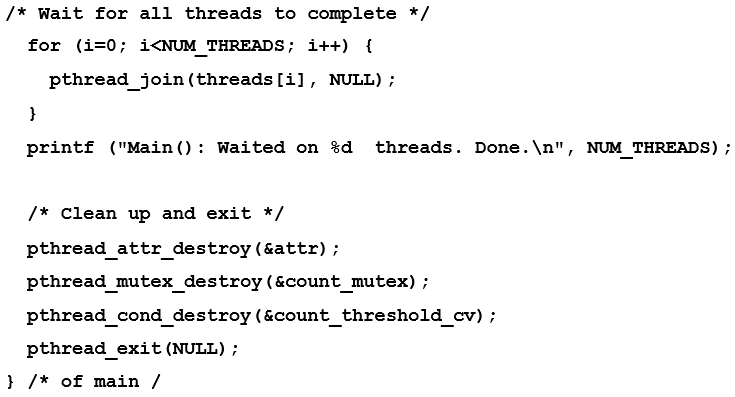


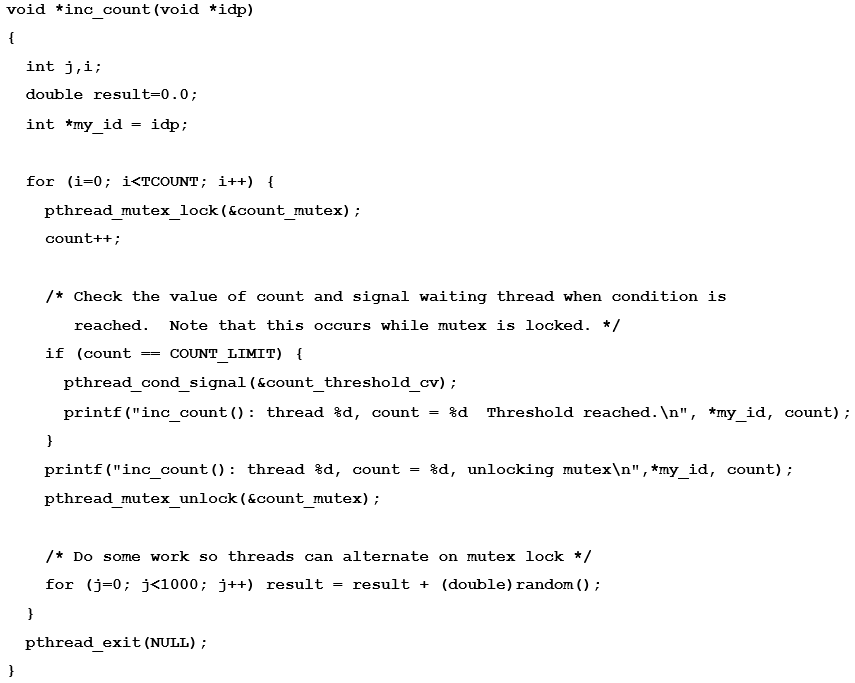
## Example Code - Using Condition Variables

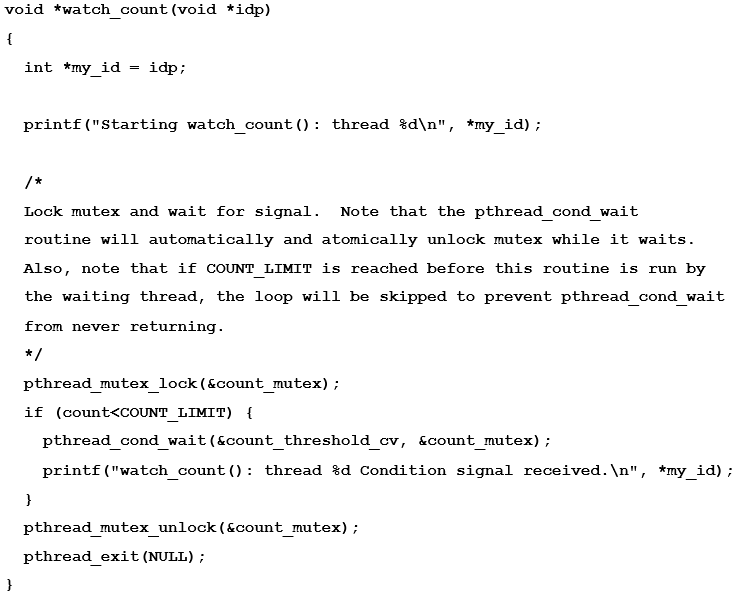
This simple example code demonstrates the use of several **Pthread** condition variable routines. The main routine creates three threads. Two of the threads perform work and update a "count" variable. The third thread waits until the count variable reaches a specified value.

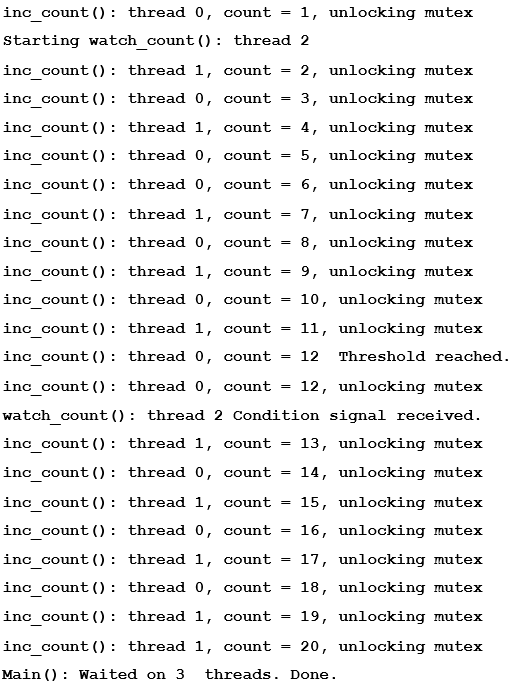












# Sample Codes

## Matrix-vector dot product

