

Parallel Computation

Hartmut Kaiser (hkaiser@cct.lsu.edu)

Introduction

Overview

3 Classes of parallel/distributed computing

- Capacity computing

- Capability computing

- Cooperative computing

3 Classes of parallel architectures (respectively)

- Loosely coupled clusters and workstation farms

- Tightly coupled vector, SIMD, SMP

- Distributed memory MPPs (and some clusters)

3 Classes of parallel execution models (respectively)

- Workflow, throughput, SPMD (ssh)

- Multithreaded with shared memory semantics (Pthreads)

- Communicating Sequential Processes (sockets)

3 Classes of programming models

- Condor

- OpenMP

- MPI

Scalability

Scalability

Strong scaling limits sustained performance

Fixed size problem to achieve reduced execution time with increased computing resources

Amdahl's Law

Sequential component limits speedup

Overhead imposes limits to granularity

Therefore to parallelism and speedup

Weak scaling allows computation size to grow with data set size

Larger data sets create more concurrent processes

Concurrent processes approximately same size granularity

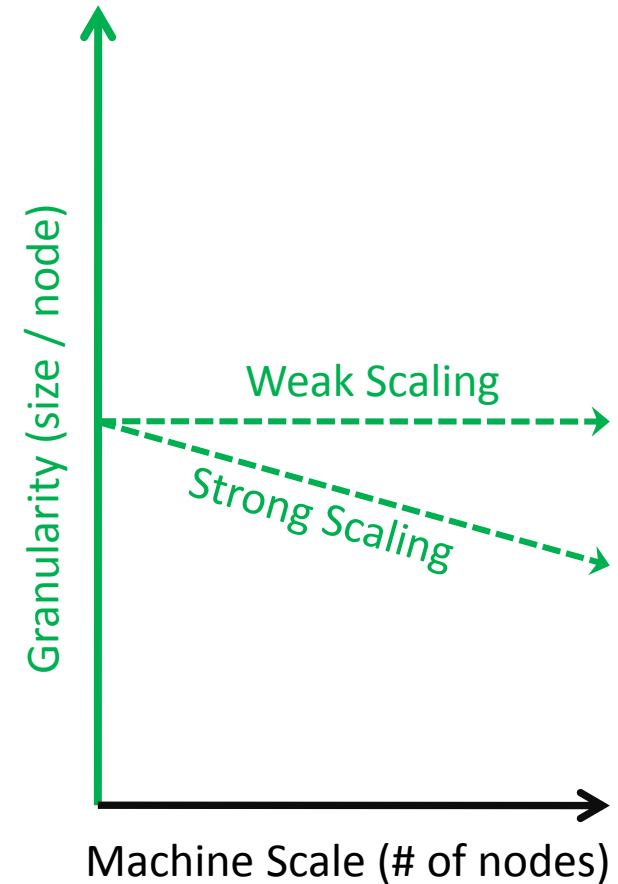
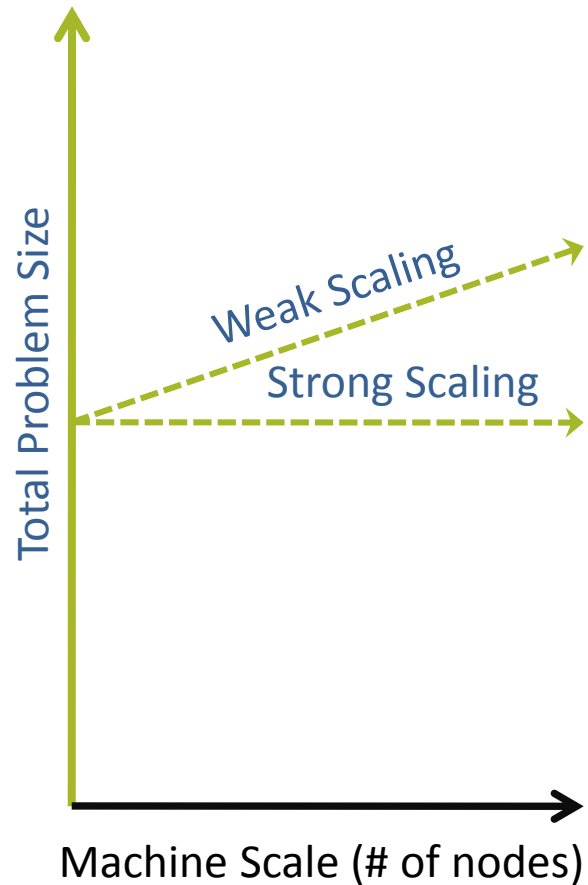
Performance increases with problem set size

Big systems are big memories for big applications

Aggregates memories of many processing nodes

Allows problems far larger than a single processor could manage

Strong Scaling, Weak Scaling



Strong Scaling, Weak Scaling

Capacity Computing

Primary scaling is increase in throughput proportional to increase in resources applied

Decoupled concurrent tasks, increasing in number of instances – scaling proportional to machine.

Capability Computing

Primary scaling is decrease in response time proportional to increase in resources applied

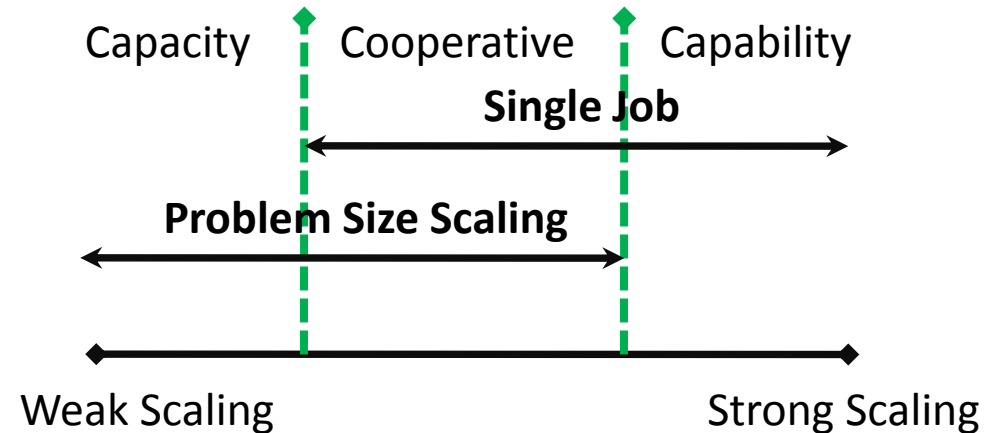
Single job, constant size – scaling proportional to machine size

Cooperative Computing

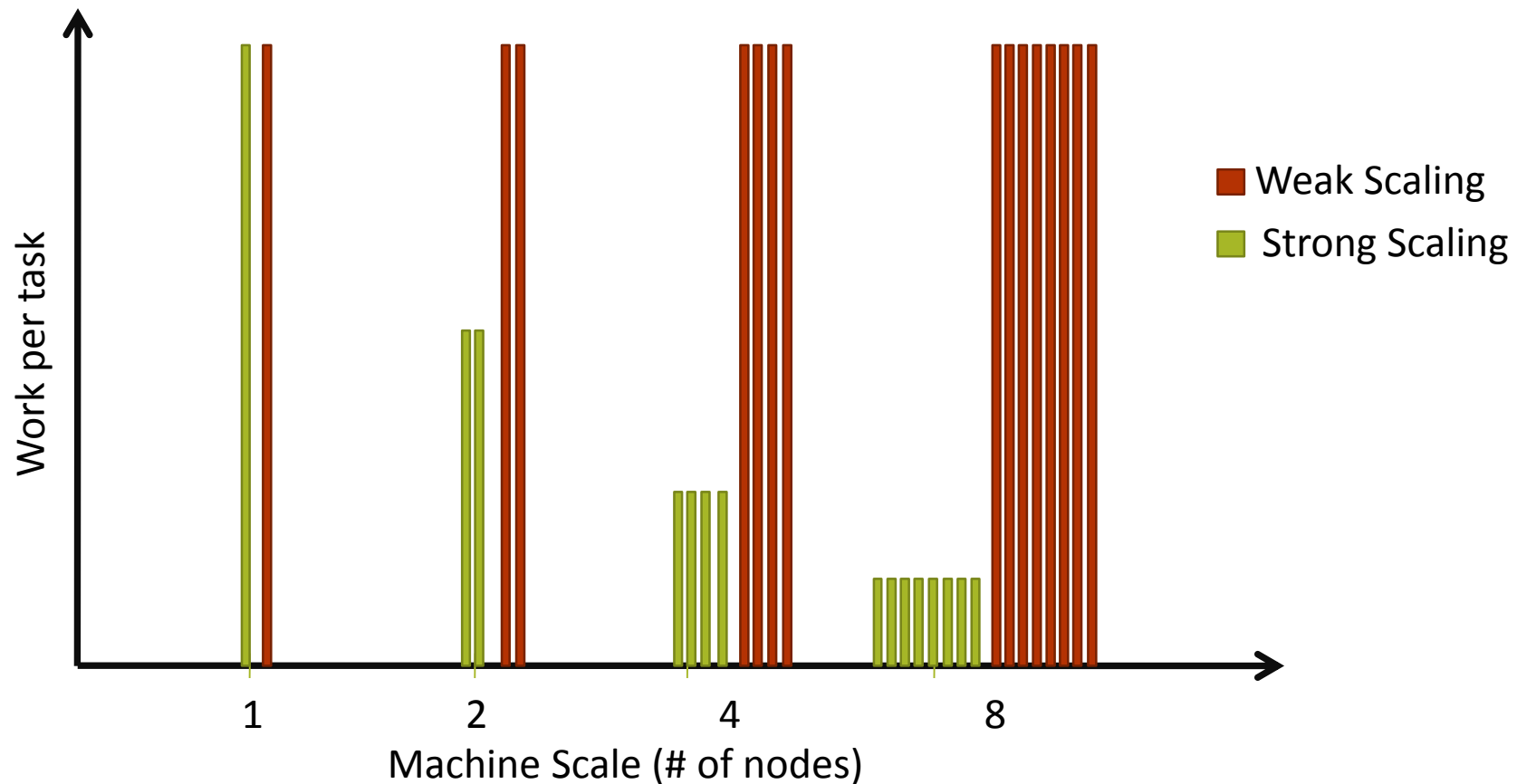
Single job, (different nodes working on different partitions of the same job)

Job size scales proportional to machine

Granularity per node is fixed



Strong Scaling, Weak Scaling

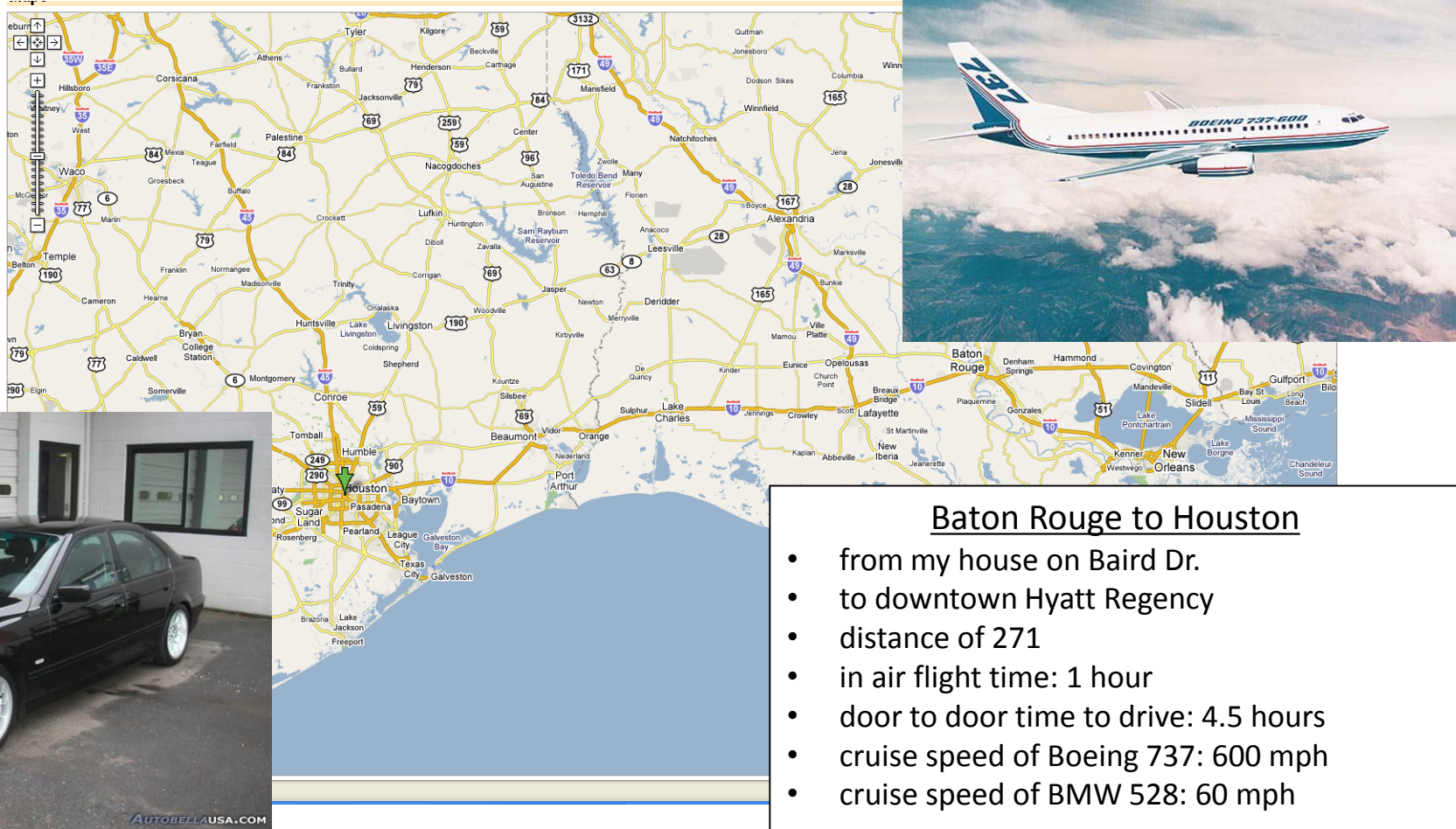


HPC Modalities

Modalities	Degree of Integration	Architectures	Execution Models	Programming Models
Capacity computing	Loosely Coupled	Clusters & Workstation farms	Workflow Throughput	Condor
Capability computing	Tightly Coupled	Vectors, SMP, SIMD	Shared Memory Multithreading	OS-Threads, OpenMP
Cooperative computing	Medium	DM, MPPs & Clusters	CSP	MPI

Amdahl's Law

Performance: Amdahl's Law



Baton Rouge to Houston

- from my house on Baird Dr.
- to downtown Hyatt Regency
- distance of 271
- in air flight time: 1 hour
- door to door time to drive: 4.5 hours
- cruise speed of Boeing 737: 600 mph
- cruise speed of BMW 528: 60 mph

Amdahl's Law: Drive or Fly?

Time door to door

BMW

Google estimates 4 hours 30 minutes

Peak performance gain: 10X

BMW cruise approx. 60 MPH

Boeing 737 cruise approx. 600 MPH

Time door to door

Boeing 737

Time to drive to BTR from my house = 15 minutes

Walking to BTR = 5 minutes

Airline estimates BTR to IAH 1 hour

Taxi time at IAH = 15 minutes (assuming gate available)

Time to get bags at IAH = 25 minutes

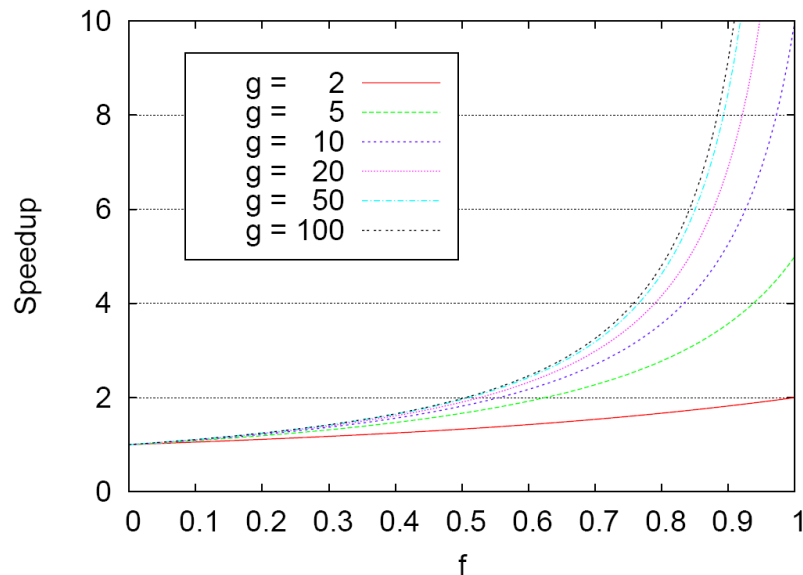
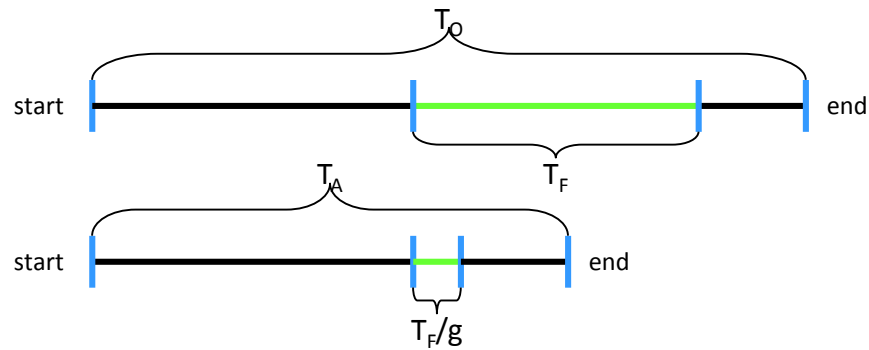
Time to get rental car = 15 minutes

Time to drive to Hyatt Regency from IAH = 45 minutes

Total time = 4.0 hours

Sustained performance gain: 1.125X

Amdahl's Law



$T_O \equiv$ time for non - accelerated computation

$T_A \equiv$ time for accelerated computation

$T_F \equiv$ time of portion of computation that can be accelerated

$g \equiv$ peak performance gain for accelerated portion of computation

$f \equiv$ fraction of non - accelerated computation to be accelerated

$f = T_F/T_O$

$S \equiv$ speed up of computation with acceleration applied

$S = T_O/T_A$

$$T_A = (1 - f) \times T_O + \left(\frac{f}{g}\right) \times T_O$$

$$S = \frac{T_O}{(1 - f) \times T_O + \left(\frac{f}{g}\right) \times T_O}$$

$$S = \frac{1}{1 - f + \left(\frac{f}{g}\right)}$$

Amdahl's Law and Parallel Computers

Amdahl's Law (f : original % to be speed up):

$$S = 1 / \left[\left(\frac{f}{g} \right) + (1 - f) \right]$$

A portion is sequential => limits parallel speedup: $S \leq 1 / (1 - f)$

Example:

What fraction sequential to get 80X speedup from 100 processors? Assume either 1 processor or 100 fully used

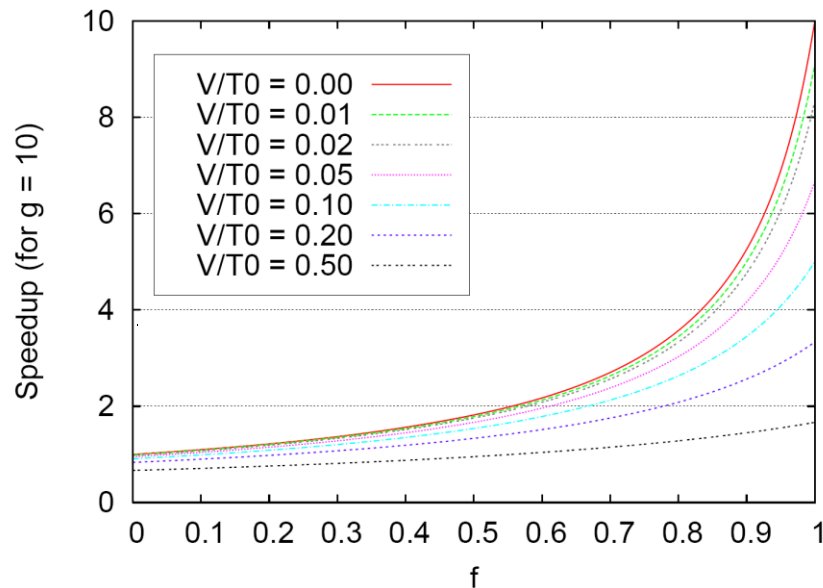
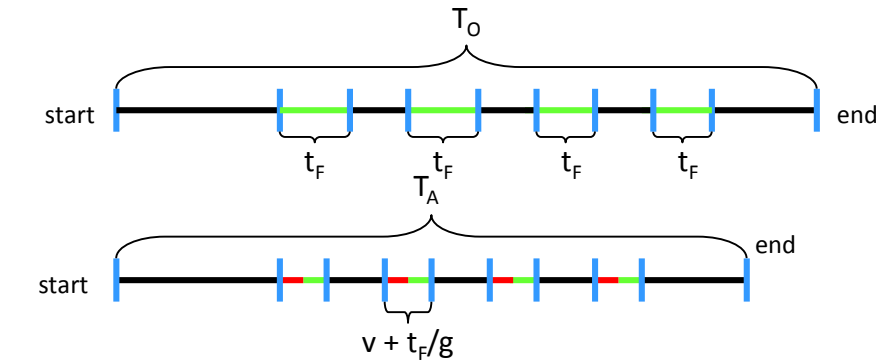
$$80 = \frac{1}{\left[\left(\frac{f}{100} \right) + (1 - f) \right]}$$

$$0.8 * f + 80 * (1 - f) = 80 - 79.2 * f = 1$$

$$f = \frac{80 - 1}{79.2} = 0.9925$$

Only 0.75% sequential!

Amdahl's Law with Overhead



$$T_F = \sum_i^n t_{Fi}$$

$v \equiv$ overhead of accelerated work segment

$$V \equiv \text{total overhead for accelerated work} = \sum_i^n v_i$$

$$T_A = (1-f) \times T_o + \frac{f}{g} \times T_o + n \times v$$

$$S = \frac{T_o}{T_A} = \frac{T_o}{(1-f) \times T_o + \frac{f}{g} \times T_o + n \times v}$$

$$S = \frac{1}{(1-f) + \frac{f}{g} + \frac{n \times v}{T_o}}$$

SMP Systems

SMP Context

A standalone system

Incorporates everything needed for

Processors

Memory

External I/O channels

Local disk storage

User interface

Enterprise server and institutional computing market

Exploits economy of scale to enhance performance to cost

Substantial performance

Target for ISVs (Independent Software Vendors)

Building block for ensemble supercomputers

Commodity clusters

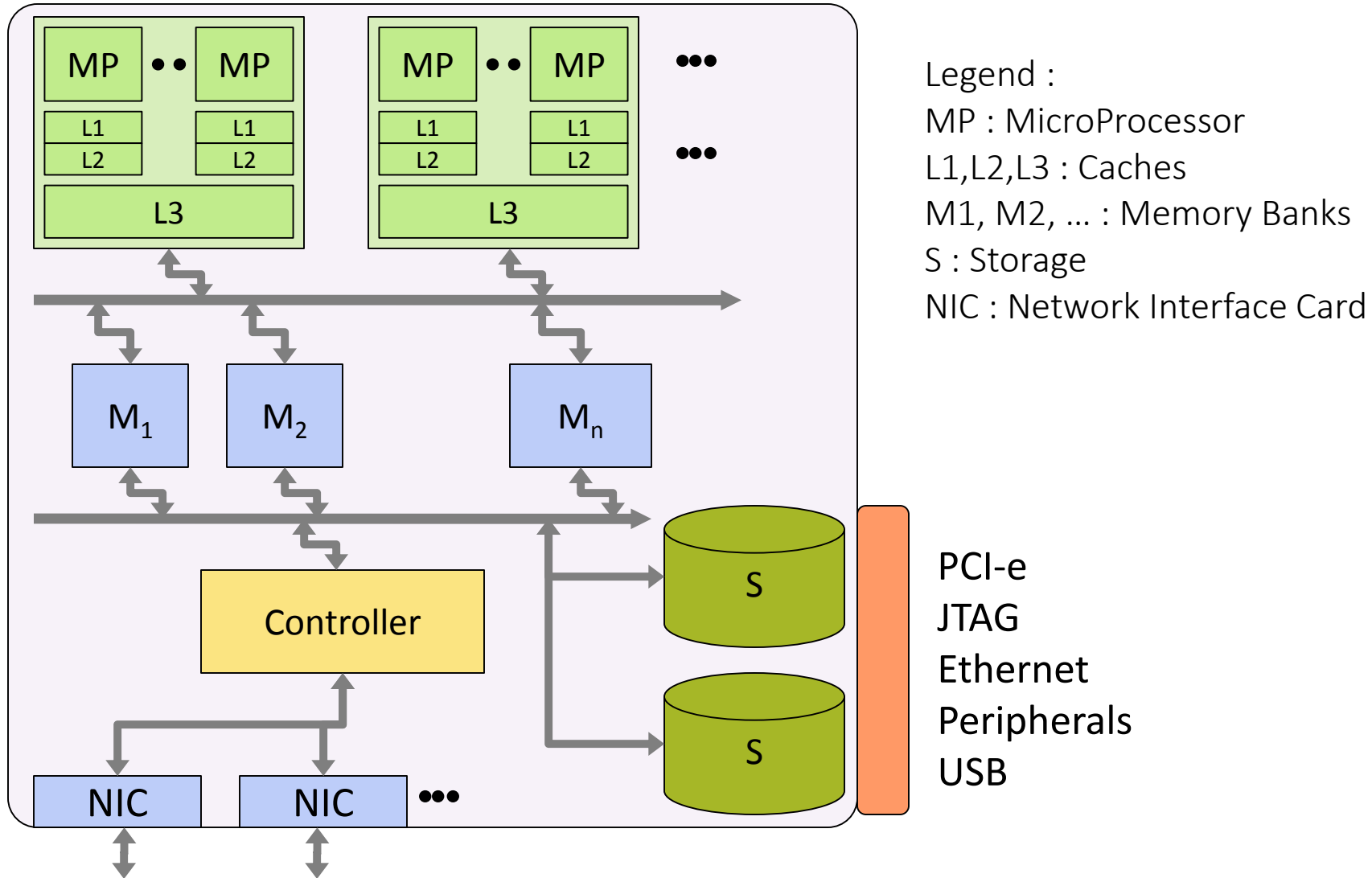
Shared memory multiple thread programming platform

Easier to program than distributed memory machines

Enough parallelism to fully employ system threads (processor cores)

MPPs

SMP Node Diagram



SMP System Examples

Vendor & name	Processor	Number of cores	Cores per proc.	Memory	Chipset	PCI slots
IBM eServer p5 595	IBM Power5 1.9 GHz	64	2	2 TB	Proprietary GX+, RIO-2	≤240 PCI-X (20 standard)
Microway QuadPuter-8	AMD Opteron 2.6 Ghz	16	2	128 GB	Nvidia nForce Pro 2200+2050	6 PCIe
Ion M40	Intel Itanium 2 1.6 GHz	8	2	128 GB	Hitachi CF-3e	4 PCIe 2 PCI-X
Intel Server System SR870BN4	Intel Itanium 2 1.6 GHz	8	2	64 GB	Intel E8870	8 PCI-X
HP Proliant ML570 G3	Intel Xeon 7040 3 GHz	8	2	64 GB	Intel 8500	4 PCIe 6 PCI-X
Dell PowerEdge 2950	Intel Xeon 5300 2.66 GHz	8	4	32 GB	Intel 5000X	3 PCIe

Sample SMP Systems



HP ProLiant



DELL PowerE



Intel Server System

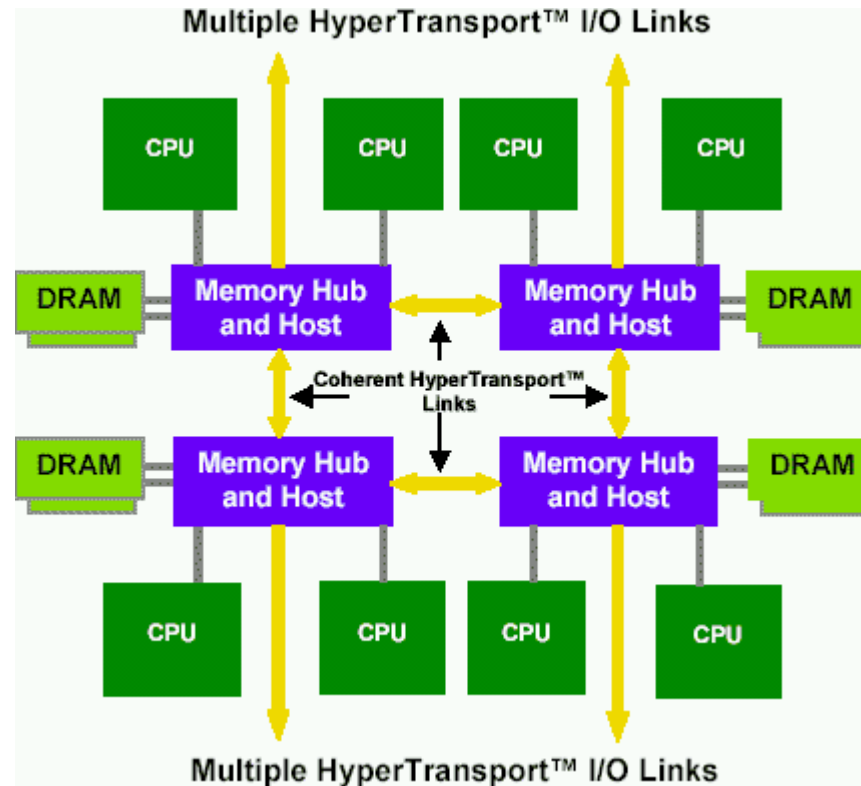


Microway Quadputer



IBM p5 595

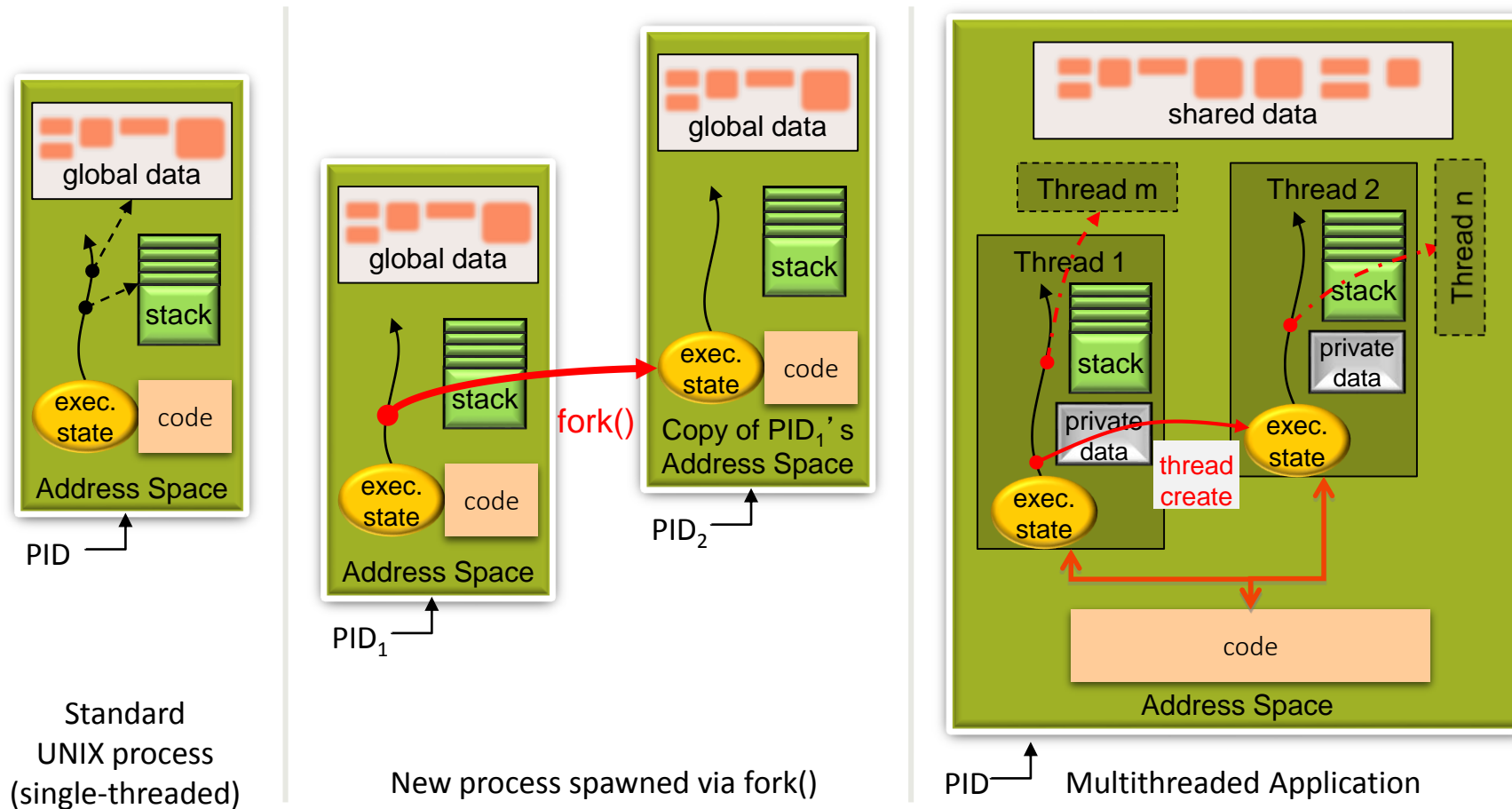
HyperTransport-based SMP System



Source: <http://www.devx.com/amd/Article/17437>

The Threaded Execution Model

UNIX Processes vs. Multithreaded Programs



Anatomy of a Thread

Thread (or, more precisely: thread of execution) is typically described as a lightweight process. There are, however, significant differences in the way standard processes and threads are created, how they interact and access resources. Many aspects of these are implementation dependent.

Private state of a thread includes:

- Execution state (instruction pointer, registers)

- Stack

- Private variables (typically allocated on thread's stack)

Threads share access to global data in application's address space.

Programming with Threads

Race Conditions

Example: consider the following piece of pseudo-code to be executed concurrently by threads T1 and T2 (the initial value of memory location A is x)

Scenario 1:

Step 1) T1: ($A \rightarrow R$) \rightarrow T1: $R=x$
Step 2) T1: ($R++$) \rightarrow T1: $R=x+1$
Step 3) T1: ($A \leftarrow R$) \rightarrow T1: $A=x+1$
Step 4) T2: ($A \rightarrow R$) \rightarrow T2: $R=x+1$
Step 5) T2: ($R++$) \rightarrow T2: $R=x+2$
Step 6) T2: ($A \leftarrow R$) \rightarrow T2: $A=x+2$

$A \rightarrow R$: read memory location A into register R
 $R++$: increment register R
 $A \leftarrow R$: write R into memory location A

Since threads are scheduled arbitrarily by an external entity, the lack of explicit synchronization may cause different outcomes.

Scenario 2:

Step 1) T1: ($A \rightarrow R$) \rightarrow T1: $R=x$
Step 2) T2: ($A \rightarrow R$) \rightarrow T2: $R=x$
Step 3) T1: ($R++$) \rightarrow T1: $R=x+1$
Step 4) T2: ($R++$) \rightarrow T2: $R=x+1$
Step 5) T1: ($A \leftarrow R$) \rightarrow T1: $A=x+1$
Step 6) T2: ($A \leftarrow R$) \rightarrow T2: $A=x+1$

Race condition (or race hazard) is a flaw in system or process whereby the output of the system or process is unexpectedly and critically dependent on the sequence or timing of other events.

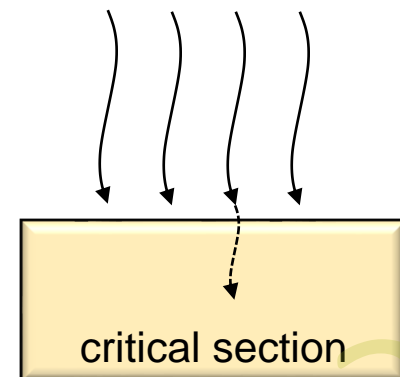
Critical Sections

The implementation of critical section must prevent any change of processor control once the execution enters the critical section.

Code on uniprocessor systems may rely on disabling interrupts and avoiding system calls leading to context switches, restoring the interrupt mask to the previous state upon exit from the critical section

General solutions rely on synchronization mechanisms (hardware-assisted when possible), discussed on the next slides

Critical section is a segment of code accessing a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.



Thread Synchronization Mechanisms

Based on atomic memory operation (require hardware support)

- Spinlocks

- Mutexes (and condition variables)

- Semaphores

- Derived constructs: monitors, rendezvous, mailboxes, etc.

Shared memory based locking

- Dekker's algorithm: http://en.wikipedia.org/wiki/Dekker%27s_algorithm

- Peterson's algorithm: http://en.wikipedia.org/wiki/Peterson%27s_algorithm

- Lamport's algorithm: http://en.wikipedia.org/wiki/Lamport%27s_bakery_algorithm,
<http://research.microsoft.com/users/lamport/pubs/bakery.pdf>

Spinlocks

Spinlock is the simplest kind of lock, where a thread waiting for the lock to become available repeatedly checks lock's status

Since the thread remains active, but doesn't perform a useful computation, such a lock is essentially *busy-waiting*, and hence generally wasteful

Spinlocks are desirable in some scenarios:

- If the waiting time is short, spinlocks save the overhead and cost of context switches, required if other threads have to be scheduled instead

- In real-time system applications, spinlocks offer good and predictable response time

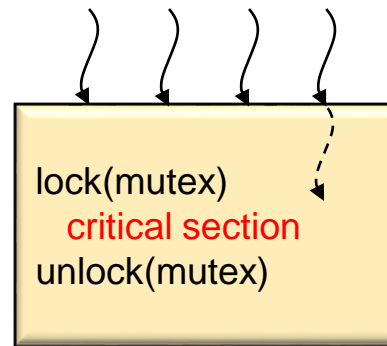
Typically use fair scheduling of threads to work correctly

Spinlock implementations require atomic hardware primitives, such as *test-and-set*, *fetch-and-add*, *compare-and-swap*, etc.

Mutexes

Mutex (abbreviation for *mutual exclusion*) is an algorithm used to prevent concurrent accesses to a common resource. The name also applies to the program object which negotiates access to that resource.

Mutex works by atomically setting an internal flag when a thread (mutex owner) enters a critical section of the code. As long as the flag is set, no other threads are permitted to enter the section. When the mutex owner completes operations within the critical section, the flag is (atomically) cleared.



Suggested reading: <http://en.wikipedia.org/wiki/Mutex>

Condition Variables

Condition variables are frequently used in association with mutexes to increase the efficiency of execution in multithreaded environments

Typical use involves a thread or threads waiting for a certain condition (based on the values of variables inside the critical section) to occur. Note that:

- The thread cannot wait inside the critical section, since no other thread would be permitted to enter and modify the variables

- The thread could monitor the values by repeatedly accessing the critical section through its mutex; such a solution is typically very wasteful

Condition variable permits the waiting thread to temporarily release the mutex it owns, and provide the means for other threads to communicate the state change within the critical section to the waiting thread (if such a change occurred)

```
/* waiting thread code: */
lock(mutex);
/* check if you can progress */
while (condition not true)
    wait(cond_var, mutex, timeout);
/* now you can; do your work */
...
unlock(mutex);
```

```
/* modifying thread code: */
lock(mutex);
/* update critical section variables */
...

/* announce state change */
signal(cond_var);
unlock(mutex);
```

Disadvantages of Locks

Blocking mechanism (forces threads to wait)

Conservative (lock has to be acquired when there's only a possibility of access conflict)

Vulnerable to faults and failures (what if the owner of the lock dies?)

Programming is difficult and error prone (deadlocks, starvation)

Does not scale with problem size and complexity

Require balancing the granularity of locked data against the cost of fine-grain locks

Not composable

Suffer from priority inversion and convoying

Difficult to debug

Pitfalls of Multithreaded Programming

Dining Philosophers Problem

Description:

- N philosophers ($N > 3$) spend their time eating and thinking at the round table
- There are N plates and N forks (or chopsticks, in some versions) between the plates
- Eating requires two forks, which may be picked one at a time, at each side of the plate
- When any of the philosophers is done eating, he starts thinking
- When a philosopher becomes hungry, he attempts to start eating
- They do it in complete silence as to not disturb each other (hence no communication to synchronize their actions is possible)

Problem:

How must they acquire/release forks to ensure that each of them maintains a healthy balance between meditation and eating?



A variation on Edsger Dijkstra's five computers competing for access to five shared tape drives problem (introduced in 1971), retold by Tony Hoare.

What Can Go Wrong at the Philosophers Table?

Deadlock

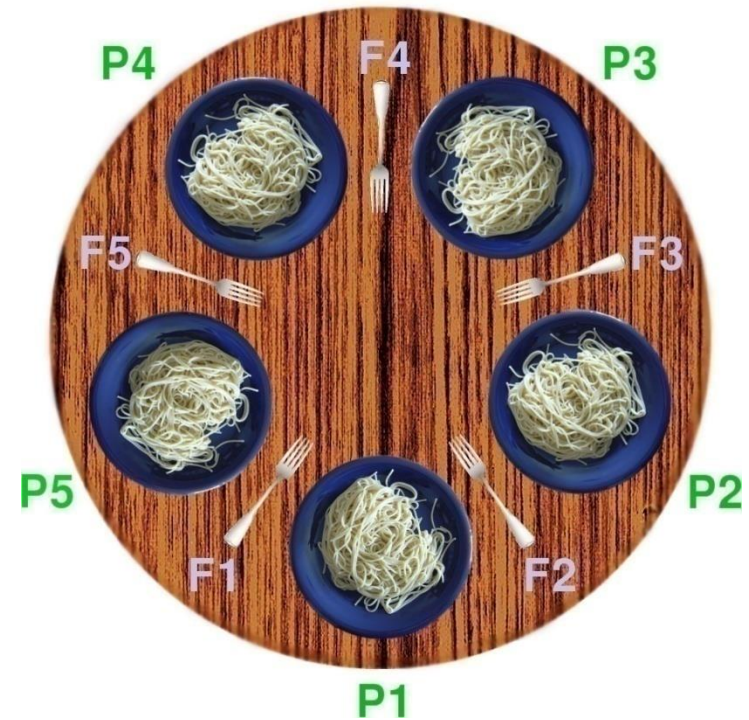
If all philosophers decide to eat at the same time and pick forks at the same side of their plates, they are stuck forever waiting for the second fork.

Livelock

Livelock frequently occurs as a consequence of a poorly thought out deadlock prevention strategy. Assume that all philosophers: (a) wait some length of time to put down the fork they hold after noticing that they are unable to acquire the second fork, and then (b) wait some amount of time to reacquire the forks. If they happen to get hungry at the same time and pick one fork using scenario leading to a deadlock and all (a) and (b) timeouts are set to the same value, they won't be able to progress (even though there is no actual resource shortage).

Starvation

There may be at least one philosopher unable to acquire both forks due to timing issues. For example, his neighbors may alternately keep picking one of the forks just ahead of him and take advantage of the fact that he is forced to put down the only fork he was able to get hold of due to deadlock avoidance mechanism.



Priority Inversion

How it happens:

- A low priority thread locks the mutex for some shared resource

- A high priority thread requires access to the same resource (waits for the mutex)

- In the meantime, a medium priority thread (not depending on the common resource) gets scheduled, preempting the low priority thread and thus preventing it from releasing the mutex

A classic occurrence of this phenomenon lead to system reset and subsequent loss of data in Mars Pathfinder mission in 1997: http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html

Priority inversion is the scenario where a low priority thread holds a shared resource that is required by a high priority thread.



Thread Safety

A code is thread-safe if it functions correctly during simultaneous execution by multiple threads.

Indicators helpful in determining thread safety

- How the code accesses global variables and heap
- How it allocates and frees resources that have global limits
- How it performs indirect accesses (through pointers or handles)
- Are there any visible side effects

Achieving thread safety

Re-entrancy: property of code, which may be interrupted during execution of one task, reentered to perform another, and then resumed on its original task without undesirable effects

Mutual exclusion: accesses to shared data are serialized to ensure that only one thread performs critical state update.
Acquire locks in an identical order on all threads

Thread-local storage: as much of the accessed data as possible should be placed in thread's private variables

Atomic operations: should be the preferred mechanism of use when operating on shared state

Thread Implementations

Approaches and Issues

Common Approaches to Thread Implementation

Kernel threads

User-space threads

Hybrid implementations

Kernel Threads (OS-Threads)

Also referred to as *Light Weight Processes*

Known to and individually managed by the kernel

Can make system calls independently

Can run in parallel on a multiprocessor (map directly onto available execution hardware)

Typically have wider range of scheduling capabilities

Support preemptive multithreading natively

Require kernel support and resources

Have higher management overhead

User-space Threads

Also known as *fibers*, *coroutines*, or *green-threads*

Operate on top of kernel threads, mapped to them via user-space scheduler

Thread manipulations (“context switches”, etc.) are performed entirely in user space

Usually scheduled cooperatively (i.e., non-preemptively), complicating the application code due to inclusion of explicit processor yield statements

Context switches cost less (on the order of subroutine invocation)

Consume less resources than kernel threads; their number can be consequently much higher without imposing significant overhead

Blocking system calls present a challenge and may lead to inefficient processor usage (user-space scheduler is ignorant of the occurrence of blocking; no notification mechanism exists in kernel either)

MxN Threading

Available on NetBSD , HPUX an Solaris to complement the existing 1x1 (kernel threads only) and Mx1 (multiplexed user threads) libraries

Multiplex M lightweight user-space threads on top of N kernel threads, $M > N$ (sometimes $M \gg N$)

User threads are unbound and scheduled on Virtual Processors (which in turn execute on kernel threads); user thread may effectively move from one kernel thread to another in its lifetime

In some implementations Virtual Processors rely on the concept of Scheduler Activations to deal with the issue of user-space threads blocking during system calls

Pthreads: Concepts and API

POSIX Threads (Pthreads)

POSIX Threads define POSIX standard for multithreaded API (IEEE POSIX 1003.1-1995)

The functions comprising core functionality of Pthreads can be divided into three classes:

- Thread management

- Mutexes

- Condition variables

Pthreads define the interface using C language types, function prototypes and macros

Naming conventions for identifiers:

- pthread_*: Threads themselves and miscellaneous subroutines

- pthread_attr_*: Thread attributes objects

- pthread_mutex_*: Mutexes

- pthread_mutexattr_*: Mutex attributes objects

- pthread_cond_*: Condition variables

- pthread_condattr_*: Condition attributes objects

- pthread_key_*: Thread-specific data keys

Programming with Pthreads

The scope of this short tutorial is:

- General thread management

- Synchronization

 - Mutexes

 - Condition variables

- Miscellaneous functions

Pthreads: Thread Creation

Function: `pthread_create()`

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
                  void *(*routine)(void *), void *arg);
```

Description: Creates a new thread within a process. The created thread starts execution of *routine*, which is passed a pointer argument *arg*. The attributes of the new thread can be specified through *attr*, or left at default values if *attr* is null. Successful call returns 0 and stores the id of the new thread in location pointed to by *thread*, otherwise an error code is returned.

```
#include <pthread.h>  
...  
void *do_work(void *input_data)  
{ /* this is thread's starting routine */  
    ...  
}  
...  
pthread_t id;  
struct { . . . } args = { . . . }; /* struct containing thread arguments */  
int err;  
...  
/* create new thread with default attributes */  
err = pthread_create(&id, NULL, do_work, (void *)&args);  
if (err != 0) { /* handle thread creation failure */  
    ...  
}
```

Pthreads: Thread Join

Function: `pthread_join()`

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Description: Suspends the execution of the calling thread until the target thread terminates (either by returning from its startup routine, or calling `pthread_exit()`), unless the target thread already terminated. If `value_ptr` is not null, the return value from the target thread or argument passed to `pthread_exit()` is made available in location pointed to by `value_ptr`. When `pthread_join()` returns successfully (i.e. with zero return code), the target thread has been terminated.

```
#include <pthread.h>
...
void *do_work(void *args) {/* workload to be executed by thread */}
...
void *result_ptr;
int err;
...
/* create worker thread */
pthread_create(&id, NULL, do_work, (void *)&args);
...
err = pthread_join(id, &result_ptr);
if (err != 0) {/* handle join error */}
else {/* the worker thread is terminated and result_ptr points to its return value */}
...
}
```

Pthreads: Thread Exit

Function: `pthread_exit()`

`void pthread_exit(void *value_ptr);`

Description: Terminates the calling thread and makes the *value_ptr* available to any successful join with the terminating thread. Performs cleanup of local thread environment by calling cancellation handlers and data destructor functions. Thread termination does not release any application visible resources, such as mutexes and file descriptors, nor does it perform any process-level cleanup actions.

```
#include <pthread.h>
...
void *do_work(void *args)
{
    ...
    pthread_exit(&return_value);
    /* the code following pthread_exit is not executed */
}
...
void *result_ptr;
pthread_t id;
pthread_create(&id, NULL, do_work, (void *)&args);
...
pthread_join(id, &result);
/* result_ptr now points to return_value */
```


Pthreads: Thread Termination

Function: `pthread_cancel()`

`void pthread_cancel(pthread_t thread);`

Description: The `pthread_cancel()` requests cancellation of thread `thread`. The ability to cancel a thread is dependent on its state and type.

```
#include <pthread.h>
...
void *do_work(void *args) { /* workload to be executed by thread */
...
pthread_t id;
int err;
pthread_create(&id, NULL, do_work, (void *)&args);
...
err = pthread_cancel(id);
if (err != 0) { /* handle cancelation failure */
...
}
```

Pthreads: Operations on Mutex Objects

Function: `pthread_mutex_lock()`, `pthread_mutex_unlock()`

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Description: The mutex object referenced by *mutex* shall be locked by calling *pthread_mutex_lock()*. If the mutex is already locked, the calling thread blocks until the mutex becomes available. After successful return from the call, the mutex object referenced by *mutex* is in locked state with the calling thread as its owner.

The mutex object referenced by *mutex* is released by calling *pthread_mutex_unlock()*. If there are threads blocked on the mutex, scheduling policy decides which of them shall acquire the released mutex.

```
#include <pthread.h>  
...  
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  
...  
/* lock the mutex before entering critical section */  
pthread_mutex_lock(&mutex);  
/* critical section code */  
...  
/* leave critical section and release the mutex */  
pthread_mutex_unlock(&mutex);  
...
```

Pthreads: Get Thread ID

Function: `pthread_self()`

`pthread_t pthread_self(void);`

Description:

Returns the thread ID of the calling thread.

```
#include <pthread.h>
...
pthread_t id;
id = pthread_self();
...
```

A Small pthread Example

```
#include <pthread.h>
#define NTHREADS 1000
int counter = 0;
pthread_mutex_t mut;
void* incr(void* p)
{
    pthread_mutex_lock(&mut);
    ++counter;
    pthread_mutex_unlock(&mut);
    return NULL;
}
```

```
int main()
{
    int i;
    pthread_t threads[NTHREADS];

    pthread_mutex_init(&mut, 0);
    for(i = 0; i < NTHREADS; ++i)
    {
        pthread_create(&threads[i], 0, incr, 0);
    }

    for(i = 0; i < NTHREADS; ++i)
    {
        pthread_join(threads[i], 0);
    }

    printf("finished: counter=%d\n", counter);
    return 0;
}
```

OpenMP

The OpenMP Programming Model

OpenMP is :

- An API (Application Programming Interface)

- NOT a programming language

- A set of compiler directives that help the application developer to parallelize their workload.

- A collection of the directives, environment variables and the library routines

OpenMP is composed of the following main components :

- Directives

- Environment variables

- Runtime library routines

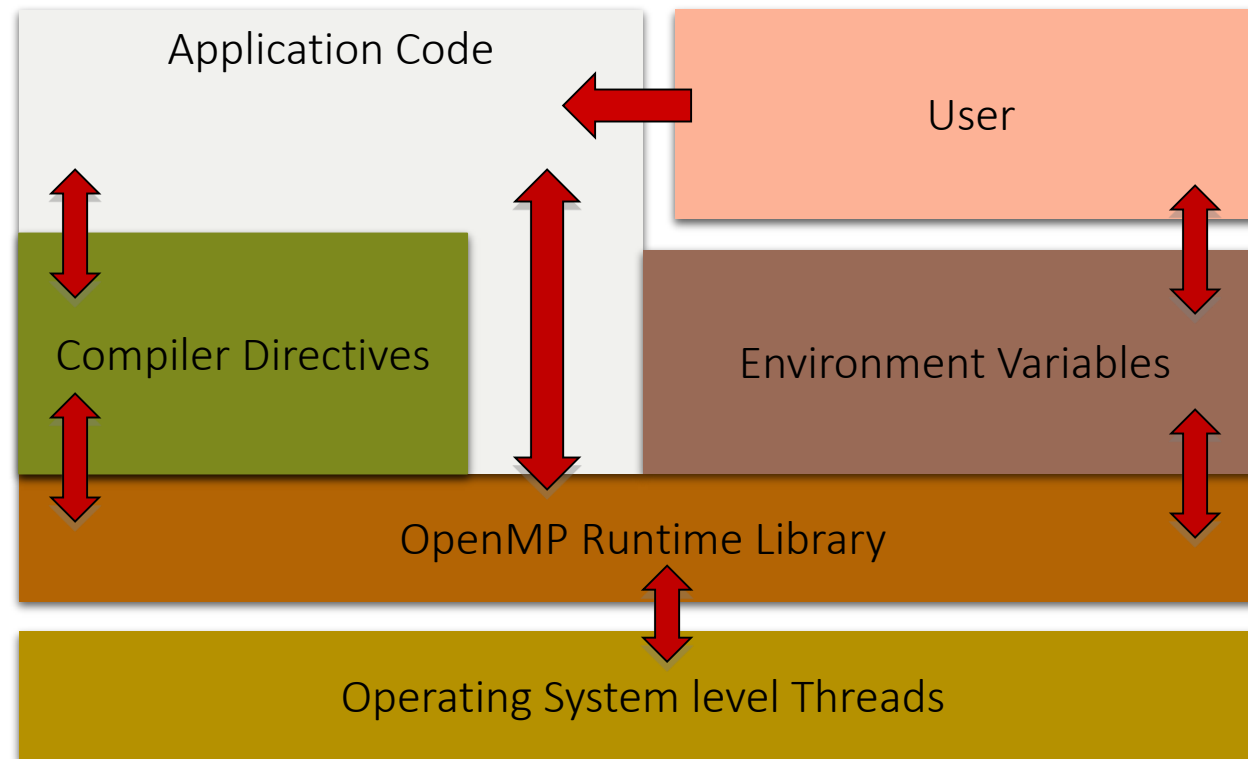
The Components of OpenMP

Directives
Parallel regions
Work sharing
Synchronization
Data scope attributes : <ul style="list-style-type: none">• private• firstprivate• last private• shared• reduction
Orphaning

Environment variables
Number of threads
Scheduling type
Dynamic thread adjustment
Nested Parallelism

Runtime library routines
Number of threads
Thread ID
Dynamic thread adjustment
Nested Parallelism
Timers
API for locking

The Architecture of OpenMP



OpenMP Runtime Library Functions

Runtime library routines help manage parallel programs

Many runtime library routines have corresponding environment variables that can be controlled by the users

Runtime libraries can be accessed by including ***omp.h*** in applications that use OpenMP: `#include <omp.h>`

For example for calls like :

omp_get_num_threads(), (by which an OpenMP program determines the number of threads available for execution) can be controlled using an environment variable set at the command-line of a shell (***\$OMP_NUM_THREADS***)

Some of the activities that the OpenMP libraries help manage are :

- Determining the number of threads/processors

- Scheduling policies to be used

- General purpose locking and portable wall clock timing routines

OpenMP: Runtime Library Functions

Function: `omp_get_num_threads()`

C/ C++ `int omp_get_num_threads(void);`

Fortran `integer function omp_get_num_threads()`

Description:

Returns the total number of threads currently in the group executing the parallel block from where it is called.

Function: `omp_get_thread_num()`

C/ C++ `int omp_get_thread_num(void);`

Fortran `integer function omp_get_thread_num()`

Description:

For the master thread, this function returns zero. For the child threads the call returns an integer between 1 and `omp_get_num_threads()-1` inclusive.

OpenMP Environment Variables

OpenMP provides 4 main environment variables for controlling execution of parallel codes:

- **OMP_NUM_THREADS** – controls the parallelism of the OpenMP application
- **OMP_DYNAMIC** – enables dynamic adjustment of number of threads for execution of parallel regions
- **OMP_SCHEDULE** – controls the load distribution in loops such as *do, for*
- **OMP_NESTED** – Enables nested parallelism in OpenMP applications

OpenMP Environment Variables

Environment Variable:	OMP_NUM_THREADS
-----------------------	------------------------

Usage :	OMP_NUM_THREADS <i>n</i>
bash/sh/ksh:	<i>export</i> OMP_NUM_THREADS=8
csh/tcsh	<i>setenv</i> OMP_NUM_THREADS 8

Description: Sets the number of threads to be used by the OpenMP program during execution.

Environment Variable:	OMP_DYNAMIC
-----------------------	--------------------

Usage :	OMP_DYNAMIC {TRUE FALSE}
bash/sh/ksh:	<i>export</i> OMP_DYNAMIC=TRUE
csh/tcsh	<i>setenv</i> OMP_DYNAMIC "TRUE"

Description: When this environment variable is set to TRUE the maximum number of threads available for use by the OpenMP program is ***n*** (\$OMP_NUM_THREADS).

OpenMP Environment Variables

Environment Variable: **OMP_SCHEDULE**

Usage :	OMP_SCHEDULE “ <i>schedule</i> ,[<i>chunk</i>]”
bash/sh/ksh:	<i>export</i> OMP_SCHEDULE static,N/P
csh/tcsh	<i>setenv</i> OMP_SCHEDULE=“GUIDED,4”

Description: Only applies to ***for*** and ***parallel for*** directives. This environment variable sets the schedule type and chunk size for all such loops. The chunk size can be provided as an integer number, the default being 1.

Environment Variable: **OMP_NESTED**

Usage :	OMP_NESTED {TRUE FALSE}
bash/sh/ksh:	<i>export</i> OMP_NESTED FALSE
csh/tcsh	<i>setenv</i> OMP_NESTED=“FALSE”

Description: Setting this environment variable to **TRUE** enables multi-threaded execution of inner parallel regions in nested parallel regions.

OpenMP: Basic Constructs

OpenMP Execution Model (FORK/JOIN):

Sequential Part (master thread)

Parallel Region (**FORK** : group of threads)

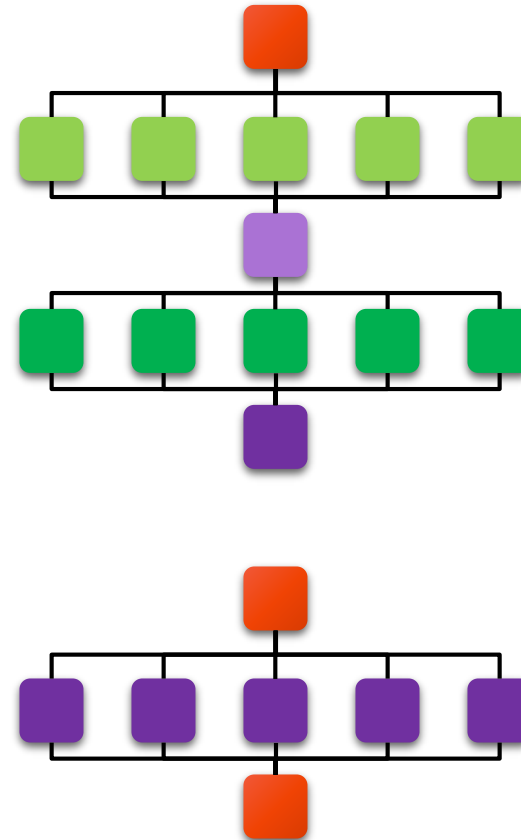
Sequential Part (**JOIN**: master thread)

Parallel Region (**FORK**: group of threads)

Sequential Part (**JOIN** : master thread)

C / C++ :

```
#pragma omp parallel {  
    parallel block  
} /* omp end parallel */
```



HelloWorld in OpenMP

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

int main ()
{
    int nthreads, tid;
    #pragma omp parallel private(nthreads, tid)
    {
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    }
}
```

Non shared copies of
data for each thread

OpenMP directive to
indicate START
segment to be
parallelized

Code segment that
will be executed in
parallel

OpenMP directive to
indicate END
segment to be
parallelized

OpenMP Execution

On encountering the C construct *#pragma omp parallel {*, *n-1* extra threads are created

omp_get_thread_num() returns a unique identifier for each thread that can be utilized. The value returned by this call is between *0* and *(OMP_NUM_THREADS – 1)*

omp_get_num_threads() returns the total number of threads involved in the parallel section of the program

Code after the parallel directive is executed independently on each of the *n* threads.

On encountering the C construct *}* (corresponding to *#pragma omp parallel{*), indicates the end of parallel execution of the code segment, the *n-1* extra threads are deactivated and normal sequential execution begins.

Compiling OpenMP Programs

C/C++:

Case sensitive directives

Syntax :

#pragma omp directive [clause [clause]..]

Compiling OpenMP source code :

(GNU C compiler) : gcc -fopenmp -o exec_name file_name.c

(Intel C compiler) : icc -o exe_file_name -openmp file_name.c

Fortran :

Case insensitive directives

Syntax :

!\$OMP directive [clause[,] clause]... (free format)

!\$OMP / C\$OMP / *\$OMP directive [clause[,] clause]... (free format)

Compiling OpenMP source code :

(GNU Fortran compiler) : gfortran -fopenmp -o exec_name file_name.f95

(Intel Fortran compiler) : ifort -o exe_file_name -openmp file_name.f

OpenMP Parallel Loops

The OpenMP Data Environment

OpenMP program always begins with a single thread of control – *master thread*

Context associated with the master thread is also known as the Data Environment.

Context is comprised of :

- Global variables, Automatic variables, Dynamically allocated variables

Context of the master thread remains valid throughout the execution of the program

The OpenMP parallel construct may be used to either share a single copy of the context with all the threads or provide each of the threads with a private copy of the context.

The sharing of Context can be performed at various levels of granularity

- Select variables from a context can be shared while keeping the context private etc.

The OpenMP Data Environment

OpenMP data scoping clauses allow a programmer to decide a variable's execution context (should a variable be *shared* or *private*.)

3 main data scoping clauses:

Shared:

A variable will have a single storage location in memory for the duration of the parallel construct, i.e. references to a variable by different threads access the same memory location.

That part of the memory is shared among the threads involved, hence modifications to the variable can be made using simple read/write operations

Modifications to the variable by different threads is managed by underlying shared memory mechanisms

Private:

A variable will have a separate storage location in memory for each of the threads involved for the duration of the parallel construct.

All read/write operations by the thread will affect the thread's private copy of the variable .

Reduction:

Exhibit both shared and private storage behavior. Usually used on objects that are the target of arithmetic reduction.

Example : summation of local variables at the end of a parallel construct

OpenMP Work-Sharing Directives

Work sharing constructs divide the execution of the enclosed block of code among the group of threads.

They do not launch new threads.

No implied barrier on entry

Implicit barrier at the end of work-sharing construct

Commonly used Work Sharing constructs :

for directive (C/C++, equivalent *DO* construct available in Fortran): shares iterations of a loop across a group of threads

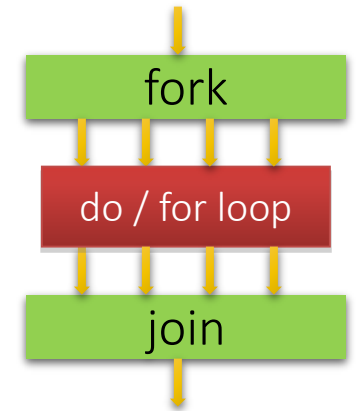
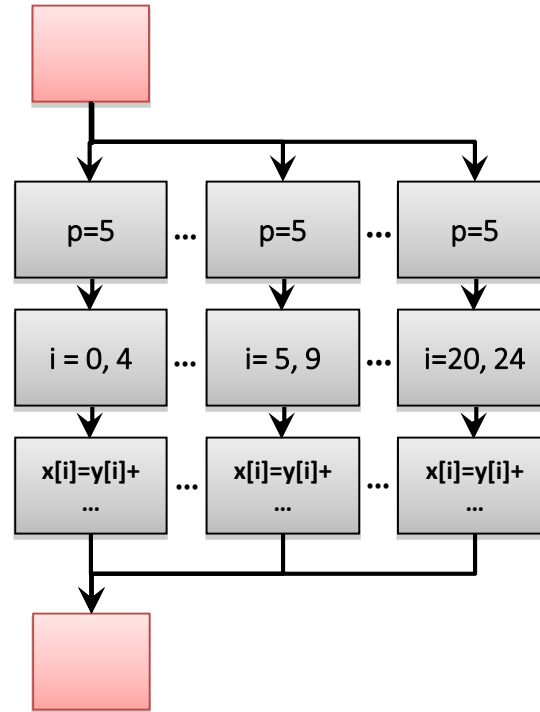
sections directive : breaks work into separate sections between the group of threads; such that each thread independently executes a section of the work.

critical directive: serializes a section of code

OpenMP *for* directive

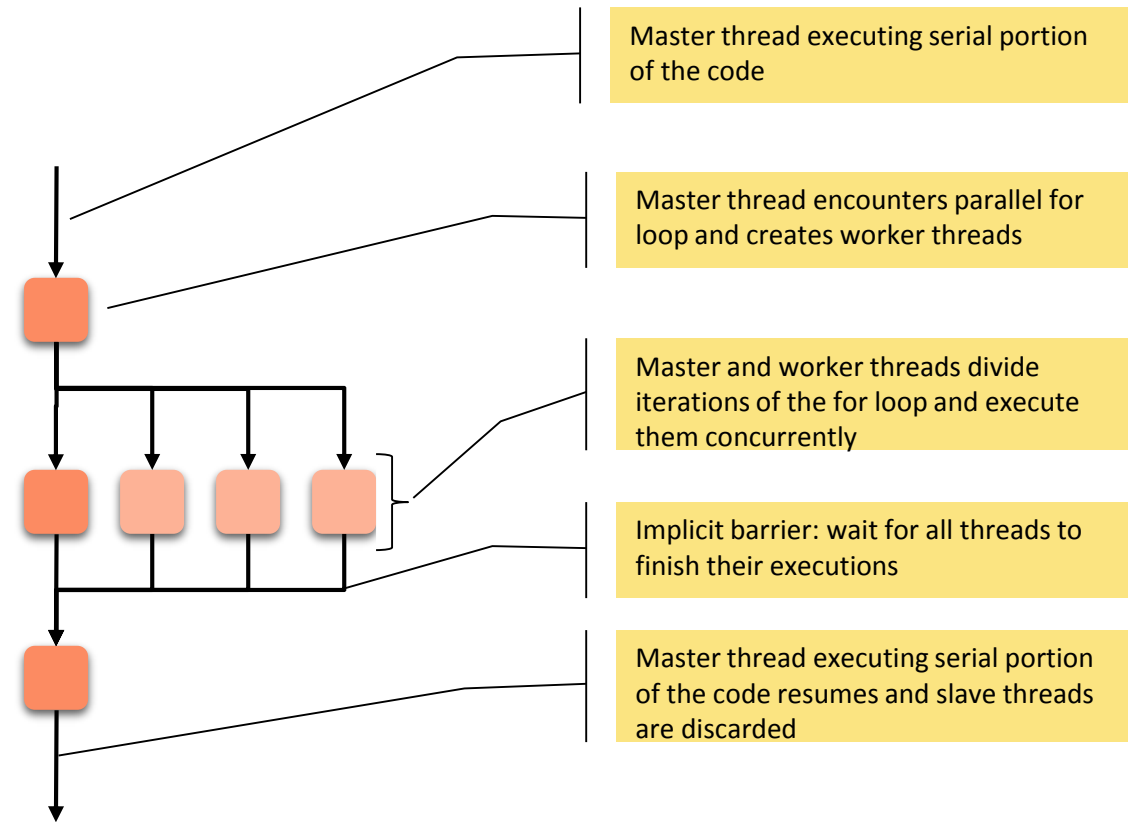
for directive helps share iterations of a loop between a group of threads

```
#pragma omp parallel
{
    p=5;
    #pragma omp for
        for (i=0; i<24; i++)
            x[i]=y[i]+p*(i+3)
    ...
} /* omp end parallel */
```



Simple Loop Parallelization

```
#pragma omp parallel for  
for (i = 0; i < n; ++i)  
    z(i) = a*x[i] + y
```



OpenMP Parallel Loops

Reductions

OpenMP : Reduction

performs reduction on *shared variables* in list based on the *operator* provided.

for C/C++ operator can be any one of :

+, *, -, ^, |, ||, & or &&

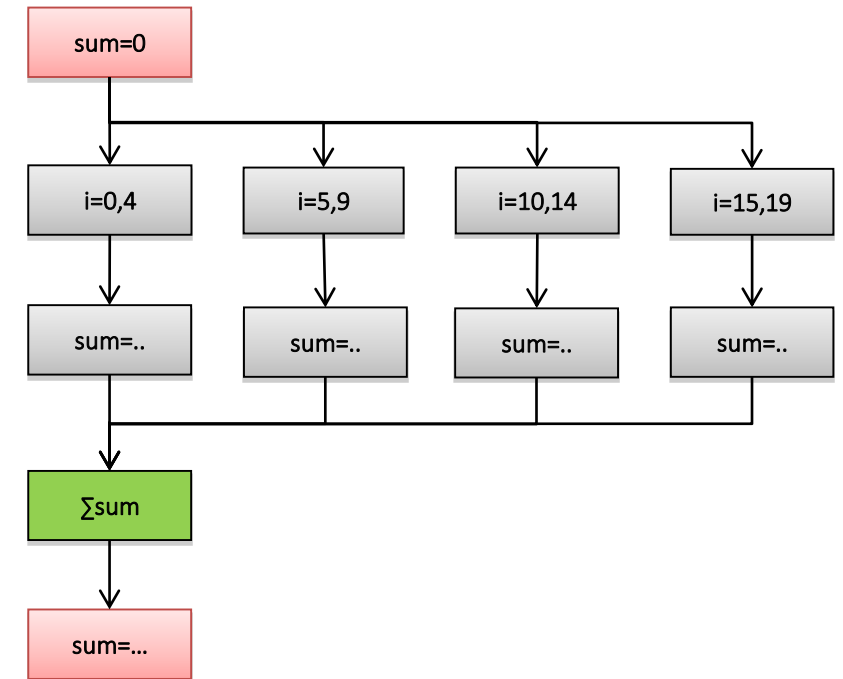
At the end of a reduction, the shared variable contains the result obtained upon combination of the list of variables processed using the operator specified.

```
sum = 0.0
```

```
#pragma omp parallel for reduction(+:sum)
```

```
for (i = 0; i < 20; ++i)
```

```
    sum = sum + (a[i] * b[i]);
```



Example: Reduction

```
#include <omp.h>
int main ()
{
    int i;
    float a[16], b[16];
    int n = 16;
    int chunk = 4;
    float result = 0.0;
    for (i = 0; i < n; ++i)
    {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
    }
}
```

Reduction example with summation where the result of the reduction operation stores the dot-product of two vectors: $\sum a[i] * b[i]$

```
#pragma omp parallel for default(shared) private(i) \
schedule(static, chunk) reduction(+:result)
for (i = 0; i < n; ++i)
    result = result + (a[i] * b[i]);

printf("Final result= %f\n",result);
}
```

SRC : <https://computing.llnl.gov/tutorials/openMP/>