Computação Paralela

Mest. Int. Engenharia Computacional

Ano letivo 2019/2020

Manuel Barroso, Nuno Lau

Parallel Programming Models

Acknowledgement: This lecture is based on slides by Florent Nolot, “Introduction to Parallel Computing”, Université de Reims Champagne-Ardenne

- Overview
- Shared Memory Model
- Threads Model
- Message Passing Model
- Data Parallel Model
- Other Models

Overview

- There are several parallel programming models in common use:
  - Shared Memory
  - Threads
  - Message Passing
  - Data Parallel
  - Hybrid
- Parallel programming models exist as an abstraction above hardware and memory architectures.

Overview

- Although it might not seem apparent, **these models are NOT specific to a particular type of machine or memory architecture**. In fact, any of these models can (theoretically) be implemented on any underlying hardware.
- Shared memory model on a distributed memory machine: Kendall Square Research (KSR) ALLCACHE approach.
  - Machine memory was physically distributed, but appeared to the user as a single shared memory (global address space). Generically, this approach is referred to as "virtual shared memory".
  - Note: although KSR is no longer in business, there is no reason to suggest that a similar implementation will not be made available by another vendor in the future.
  - Message passing model on a shared memory machine: MPI on SGI Origin.
- The SGI Origin employed the CC-NUMA type of shared memory architecture, where every task has direct access to global memory. However, the ability to send and receive messages with MPI, as is commonly done over a network of distributed memory machines, is not only implemented but is very commonly used.

Overview

- Which model to use is often a combination of what is available and personal choice.
- **There is no "best" model**, although there certainly are better implementations of some models over others.
- The following sections describe each of the models mentioned above, and also discuss some of their actual implementations.

Shared Memory Model

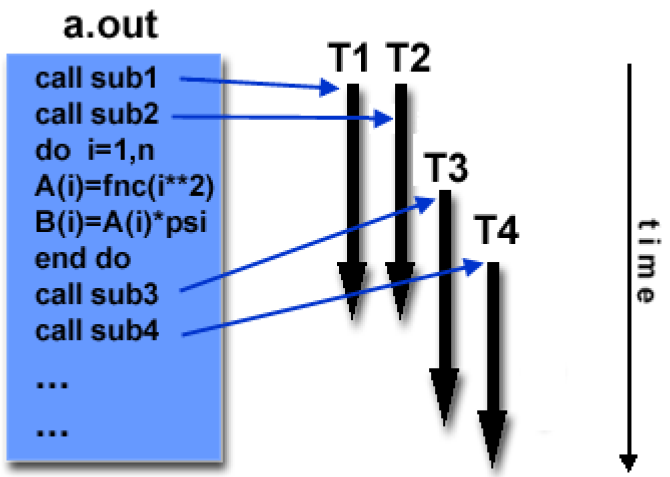
- In the shared-memory programming model, tasks share a common address space, which they read and write asynchronously.
- Various mechanisms such as locks / semaphores may be used to control access to the shared memory.
- An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. Program development can often be simplified.
- An important disadvantage in terms of performance is that it becomes more difficult to understand and manage data locality.

Shared Memory Model: Implementations

- On shared memory platforms, the native compilers translate user program variables into actual memory addresses, which are global.
- No common distributed memory platform implementations currently exist. However, as mentioned previously in the Overview section, the KSR ALLCACHE approach provided a shared memory view of data even though the physical memory of the machine was distributed.

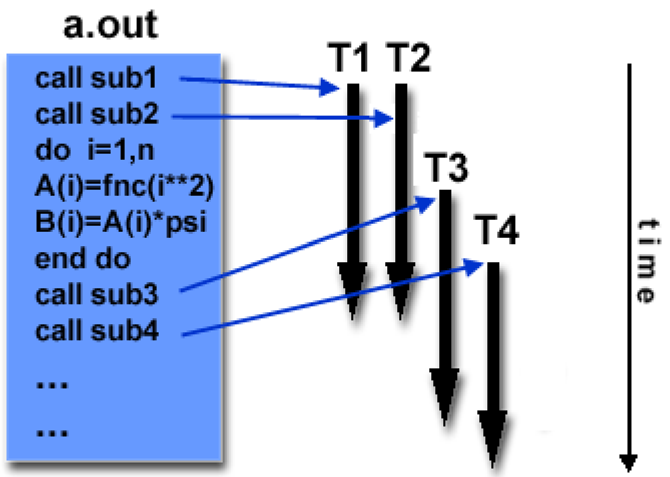
Threads Model

- In the threads model of parallel programming, **a single process can have multiple, concurrent execution paths.**
- Perhaps the most simple analogy that can be used to describe threads is the concept of a single program that includes a number of parallel subroutines:
  - The main program **a.out** is scheduled to run by the native operating system. **a.out** loads and acquires all of the necessary system and user resources to run.
  - **a.out** performs some serial work, and then **creates a number of tasks (threads)** that can be scheduled and run by the operating system concurrently.
  - **Each thread has local data**, but also, **shares the entire resources of a.out**. This saves the overhead associated with replicating a program's resources for each thread. Each thread also benefits from a global memory view because it shares the memory space of **a.out**.



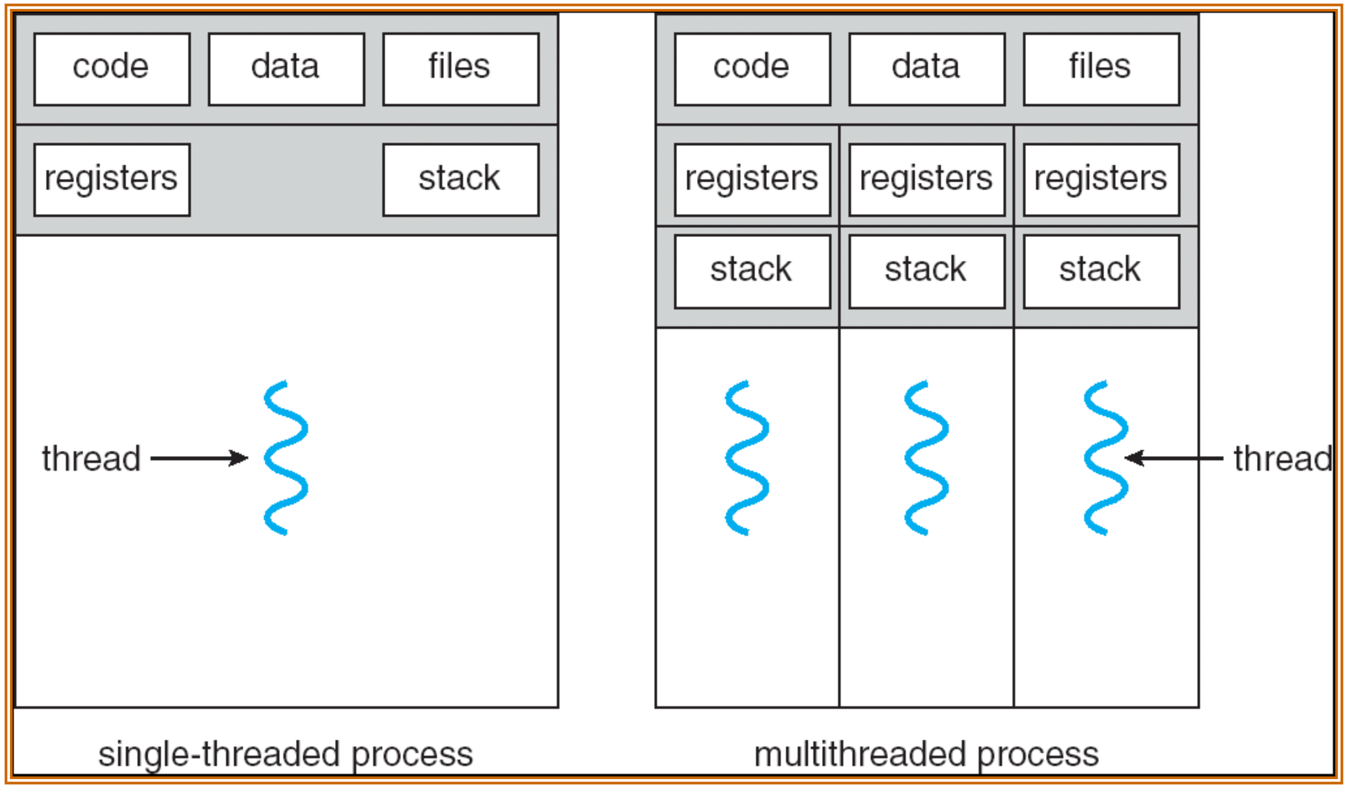
Threads Model

  - A thread's work may best be described as a parallel subroutine within the main program. Any thread can execute any subroutine at the same time as other threads.
  - **Threads communicate** with each other **through global memory** (updating address locations). This requires synchronization constructs to insure that more than one thread is not updating the same global address at any time.
  - Threads can come and go, but **a.out** remains present to provide the necessary shared resources until the application has completed.
- Threads are commonly associated with shared memory architectures and operating systems.

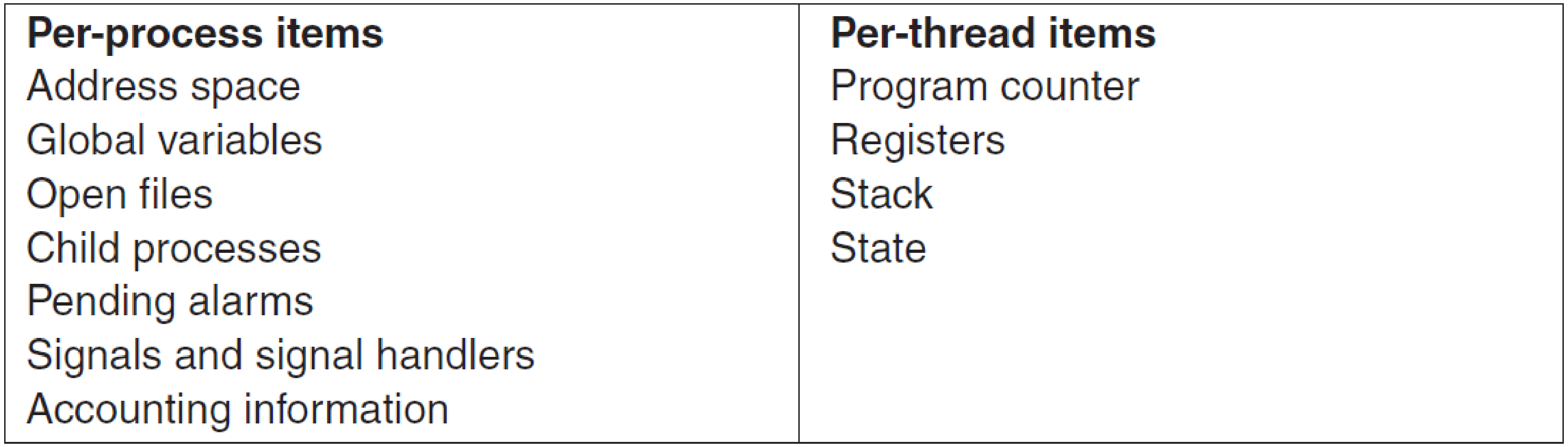


```
call sub1
call sub2
do  i=1,n
A(i)=fnc(i**2)
B(i)=A(i)*psi
end do
call sub3
call sub4
...
...
```

Single and Multi threaded Processes

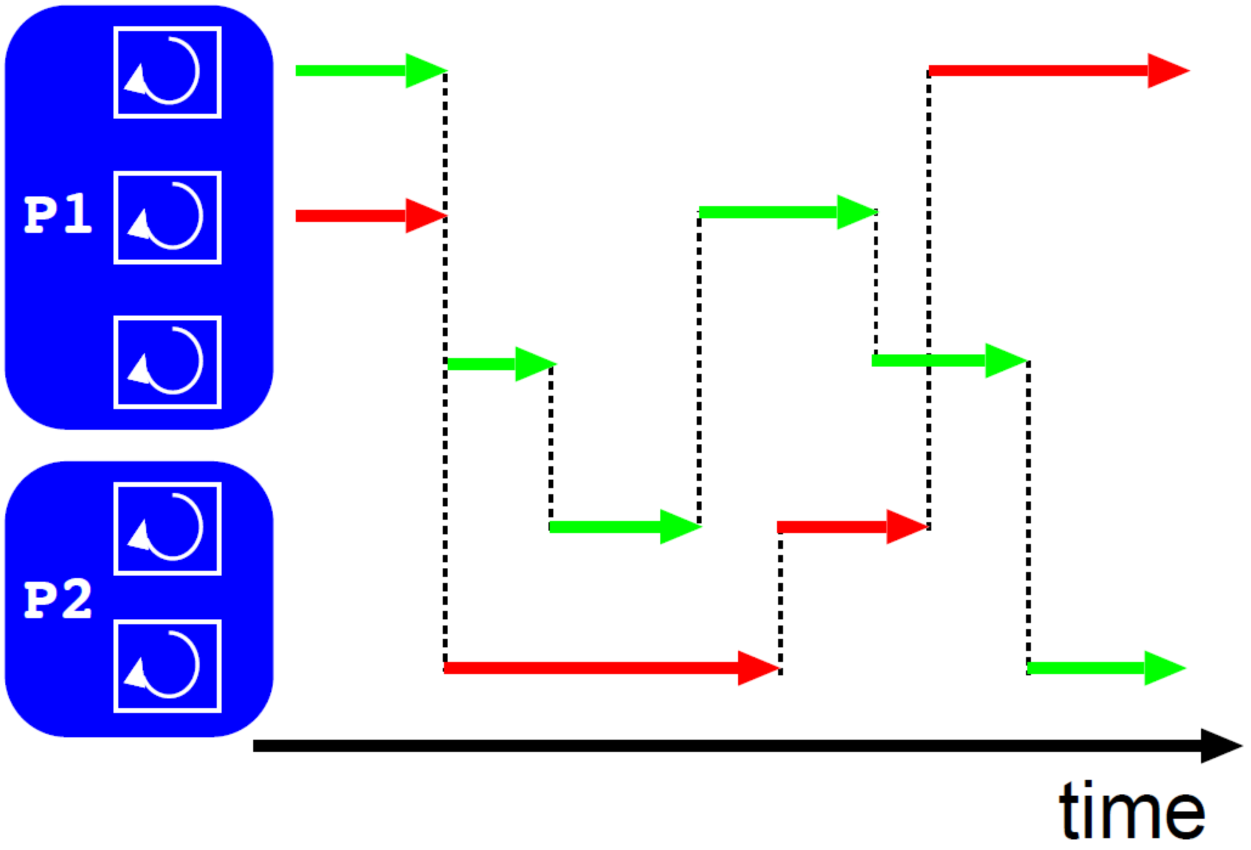


Processes and Threads



Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Processes, Threads and CPUs



from CP@FCUP

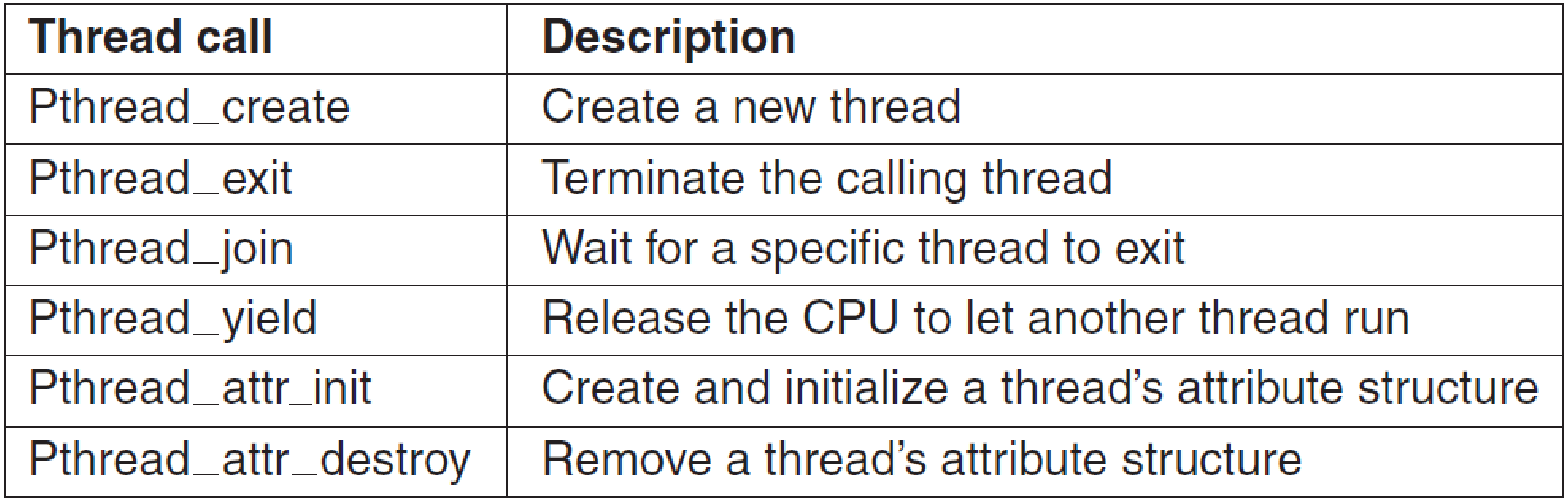
Threads Model Implementations

- From a programming perspective, threads implementations commonly comprise:
  - A **library of subroutines** that are called from within parallel source code
  - A **set of compiler directives** embedded in either serial or parallel source code
- In both cases, the programmer is responsible for determining all parallelism.
- Threaded implementations are not new in computing. Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- Unrelated standardization efforts have resulted in two very different implementations of threads: ***POSIX Threads*** and ***OpenMP***.

Threads Model: POSIX

- **POSIX Threads**
  - Library based; require parallel coding
  - Specified by the IEEE POSIX 1003.1c standard (1995).
  - C Language only
  - Commonly referred to as Pthreads.
  - Most hardware vendors now offer Pthreads in addition to their proprietary threads implementations.
  - Very explicit parallelism; requires significant programmer attention to detail.

POSIX *Threads*



Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Create POSIX Threads

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

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

#define NUM_THREADS 5

void *PrintMsg(void *threadid) {
   long tid;
   tid = (long)threadid;
   printf("Hello World! Thread ID, %d\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];
   int rc;
   int i;

   for( i = 0; i < NUM_THREADS; i++ ) {
      printf( "main() : creating thread, %d\n",i);
      rc = pthread_create(&threads[i], NULL, PrintMsg, (void *)i);

      if (rc) {
         printf("Error: unable to create thread, %d\n", rc);
         exit(1);
      }
   }
   pthread_exit(NULL);
}
```

Threads Model: OpenMP

- **OpenMP**
  - Compiler directive based; can use serial code
  - Jointly defined and endorsed by a group of major computer hardware and software vendors. The OpenMP Fortran API was released October 28, 1997. The C/C++ API was released in late 1998.
  - Portable / multi-platform, including Unix and Windows NT platforms
  - Available in C/C++ and Fortran implementations
  - Can be very easy and simple to use - provides for "incremental parallelism"
- Microsoft has its own implementation for threads, which is not related to the UNIX POSIX standard or OpenMP.

Message Passing Model

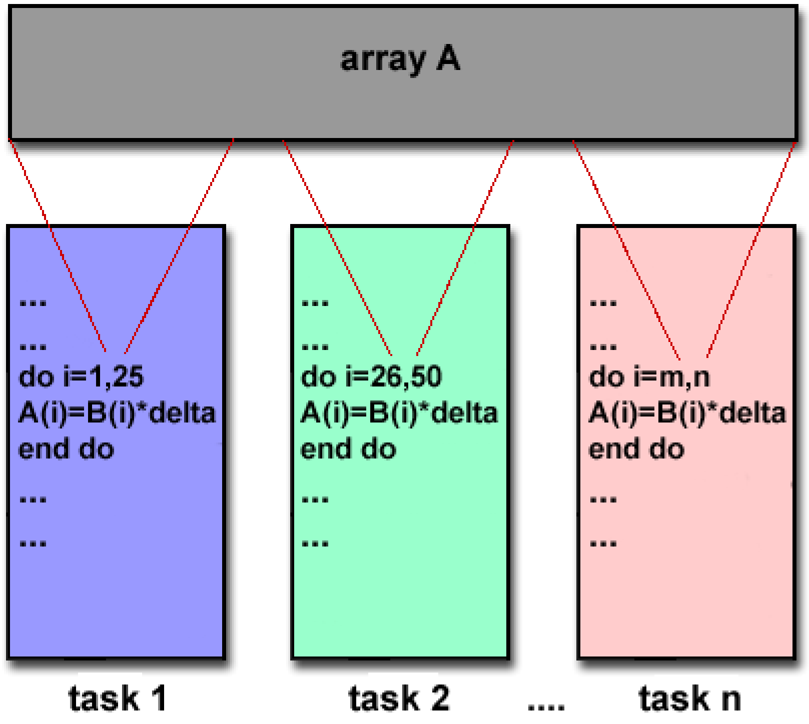
- The message passing model demonstrates the following characteristics:
  - A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines.
  - Tasks exchange data through communications by sending and receiving messages.
  - Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

Message Passing Model Implementations: MPI

- From a programming perspective, message passing implementations commonly comprise a library of subroutines that are imbedded in source code. The programmer is responsible for determining all parallelism.
- Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.
- In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations.
- Part 1 of the **Message Passing Interface (MPI)** was released in 1994. Part 2 (MPI-2) was released in 1996. MPI-3 was approved in 2012. MPI specifications are available on the web at https://www.mpi-forum.org/docs/.

Message Passing Model Implementations: MPI

- MPI is now the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. Most, if not all of the popular parallel computing platforms offer at least one implementation of MPI.
- For shared memory architectures, MPI implementations usually don't use a network for task communications. Instead, they use shared memory (memory copies) for performance reasons.



Data Parallel Model

- The data parallel model demonstrates the following characteristics:
  - Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.
  - A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
  - Tasks perform the same operation on their partition of work, for example, "add 4 to every array element".
- On shared memory architectures, all tasks may have access to the data structure through global memory.
- On distributed memory architectures the data structure is split up and resides as "chunks" in the local memory of each task.

Other Models

- Other parallel programming models besides those previously mentioned certainly exist, and will continue to evolve along with the ever changing world of computer hardware and software.
- Only three of the more common ones are mentioned here.
  - Hybrid
  - Single Program Multiple Data
  - Multiple Program Multiple Data

Hybryd

- In this model, any two or more parallel programming models are combined.
- Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with either the threads model (POSIX threads) or the shared memory model (OpenMP). This hybrid model lends itself well to the increasingly common hardware environment of networked SMP machines.
- Another common example of a hybrid model is combining data parallel with message passing. Data parallel implementations (F90, HPF) on distributed memory architectures actually use message passing to transmit data between tasks, transparently to the programmer.

Single Program Multiple Data (SPMD)

- Single Program Multiple Data (SPMD):
- SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- A single program is executed by all tasks simultaneously.
- At any moment in time, tasks can be executing the same or different instructions within the same program.
- SPMD programs usually have the necessary logic programmed into them to allow different tasks to branch or conditionally execute only those parts of the program they are designed to execute. That is, tasks do not necessarily have to execute the entire program - perhaps only a portion of it.
- All tasks may use different data

Multiple Program Multiple Data (MPMD)

- Multiple Program Multiple Data (MPMD):
- Like SPMD, MPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- MPMD applications typically have multiple executable object files (programs). While the application is being run in parallel, each task can be executing the same or different program as other tasks.
- All tasks may use different data