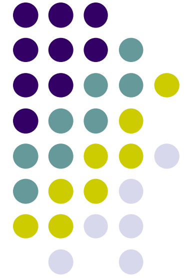
U. de Valparaíso, Escuela Ingeniería Industrial
U. de Chile, Centro de Modelamiento Matemático

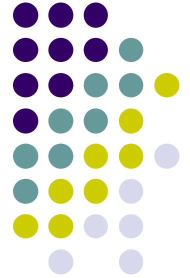


High Volume Manufacturing	2004	2006	2008	2010	2012	2014	2016	2018
Technology Node (nm)	90	65	45	32	22	16	11	8
Integration Capacity (BT)	2	4	8	16	32	64	128	256
Delay = CV/I scaling	0.7	~0.7	>0.7	Delay scaling will slow down				
Energy/Logic Op scaling	>0.35	>0.5	>0.5	Energy scaling will slow down				
Bulk Planar CMOS		High Probability					Low Probability	
Alternate, 3G etc		Low Probability					High Probability	
Variability		Medium		High			Very High	
ILD (K)	~3	<3	Reduce slowly towards 2-2.5					
RC Delay	1	1	1	1	1	1	1	1
Metal Layers	6-7	7-8	8-9	0.5 to 1 layer per generation				

Parallel Programming Models

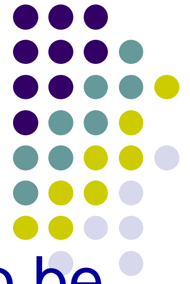
Gonzalo Hernandez

Parallel Programming Models: Agenda



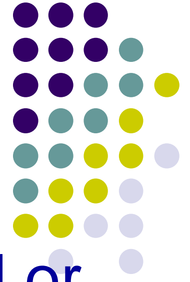
1) Cluster Computing
   - Lefque
   - Distributed & Shared Memory
2) Developing Parallel Programs:
   - Identification of hotspots & bottlenecks
   - Partitioning: domain & functional decomposition
   - Designing communications
   - Performance Evaluation
3) Parallel Programming Models
   - Message passing (distributed memory)
   - Threads (shared memory)
   - Data Parallel
   - Hybrid, SPMD, MPMD

1) Cluster Computing



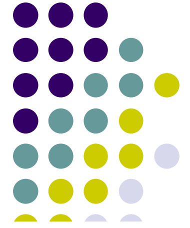
- Traditionally, parallel computing has been considered to be "the high end of computing" and has been motivated by numerical simulations of complex systems and "Grand Challenge Problems" such as:
  - Weather and climate modeling
  - Bioinformatics
  - Geology: Seismic activity simulation, oil exploration
  - Engineering: transportation and telecommunication networks, mining operations, manufacturing processes, pattern recognition and image processing, mechanical devices, electronic circuits, etc.
- These problems are studied mainly in Universities and Research Centers.

1) Cluster Computing



- Today's commercial applications are providing an equal or greater driving force in the development of faster computers. These applications require the processing of large amounts of data in sophisticated ways. For example:
  - Parallel databases
  - Web based business services (business intelligence)
  - Oil exploration
  - Data mining
  - Computer-aided diagnosis in medicine
  - Advanced graphics and virtual reality (entertainment industry)
  - Networked video and multi-media technologies
  - Collaborative work environments

1) Cluster Computing



Parallel Applications

Sequential Applications

Parallel Programming Environments

Cluster Middleware

(Single System Image and Availability Infrastructure)

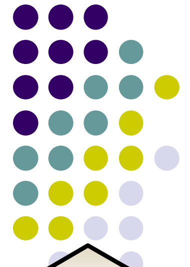
computing node | computing node | computing node | computing node | computing node

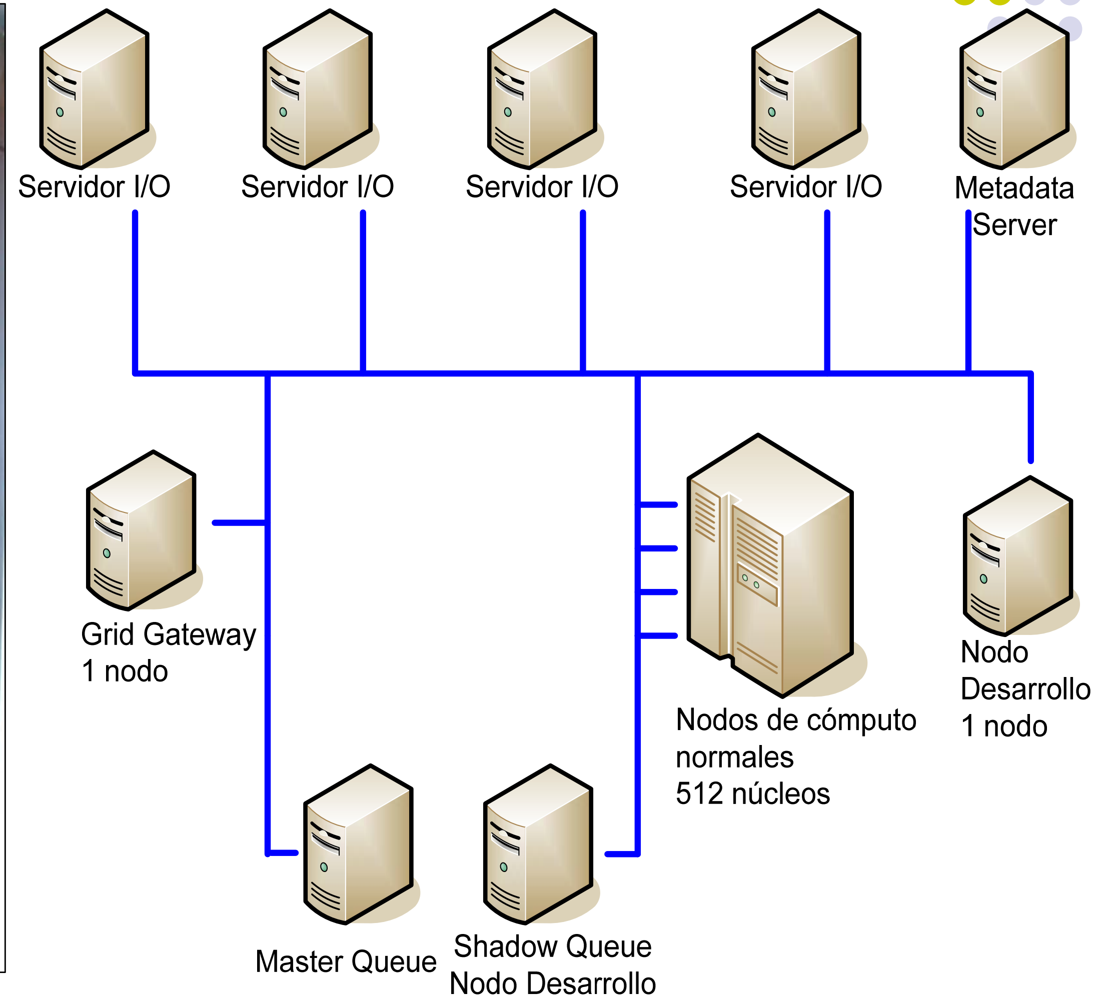
Comm. S/W | Comm. S/W | Comm. S/W | Comm. S/W | Comm. S/W

Net. Interface HW | Net. Interface HW | Net. Interface HW | Net. Interface HW | Net. Interface HW

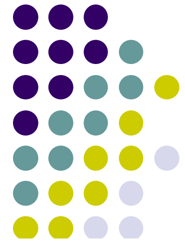
High Speed Network/Switch

1) Cluster Computing: CMM Levque

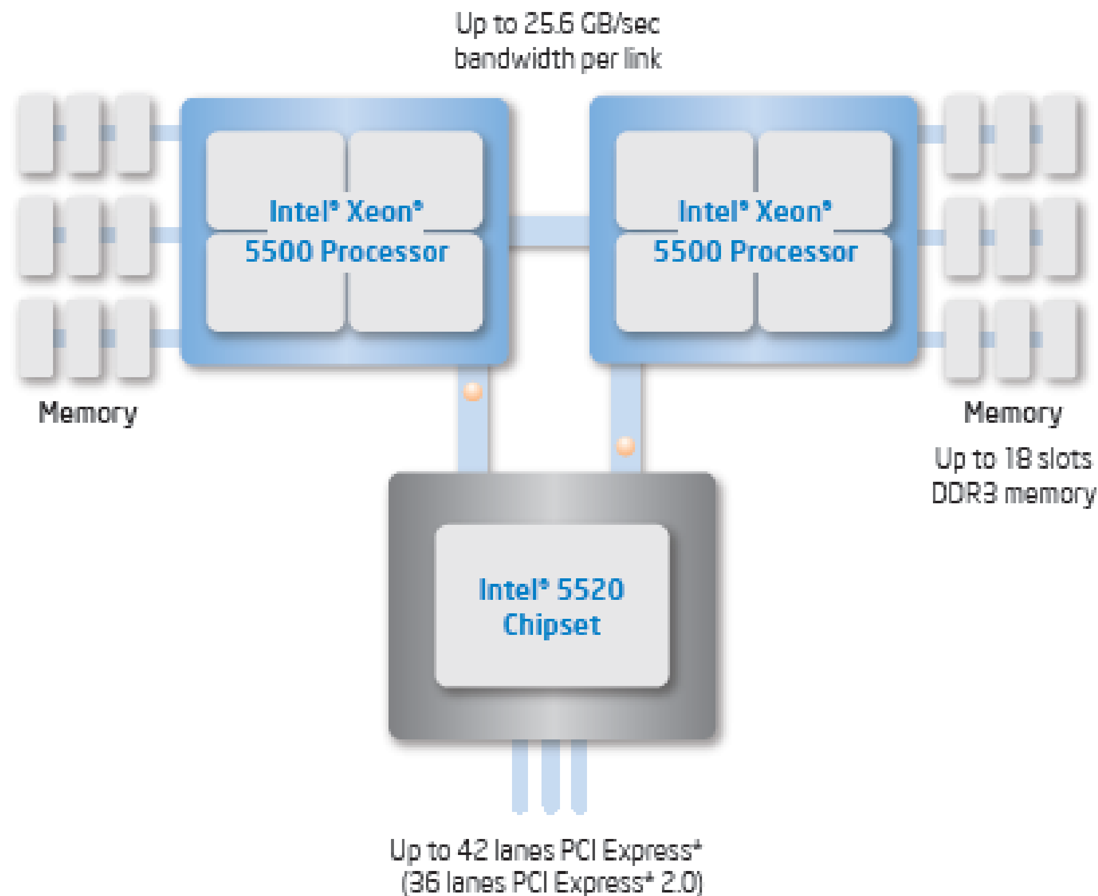




1) Cluster Computing: CMM Levque



- Intel Processors Xeon 5550



Key Benefits

- Up to 3.5x greater bandwidth for data-intensive applications[23]
- Up to 18 slots DIMM with up to 144 GB DDR3 memory
- Up to 42 lanes PCI Express (36 lanes PCI Express 2.0)

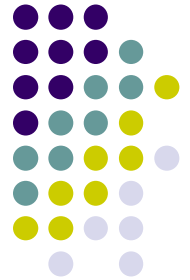
Key Technologies

- Two Intel Xeon processors 5500 series
- Intel Turbo Boost Technology
- Intel Hyper-Threading Technology
- 8 MB shared L3 cache featuring Enhanced Smart Cache
- Intel QuickPath Technology
- Intel Intelligent Power Technology

Key Usage

- Bandwidth-intensive applications
- HPC clusters
- Multi-tasking user environments

1) Cluster Computing: Lefque



NUMA-based Design

Intra-CMP Inter-CMP

Core Core
L2 Cache L2 Cache
Dual Core Chip

Core Core
L2 Cache L2 Cache
Dual Core Chip

Memory Memory

Bus-based Design

Intra-CMP Inter-CMP

Core Core
L2 Cache
Dual Core Chip

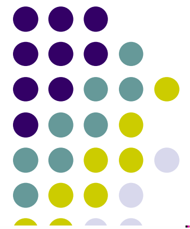
Core Core
L2 Cache
Dual Core Chip

Memory

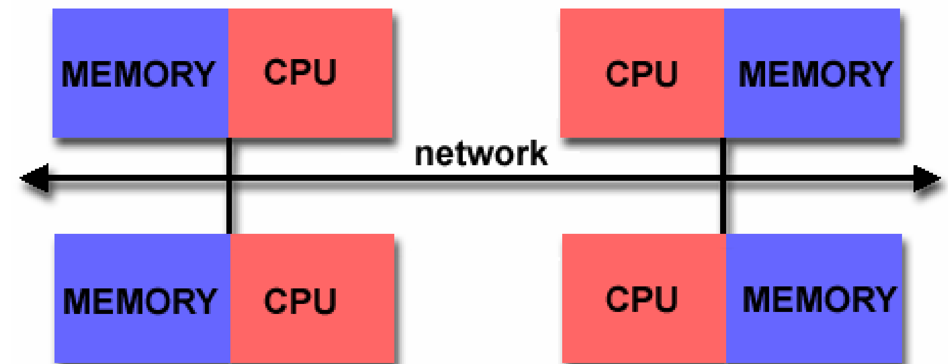
Inter-Node

Network

1) Cluster Computing: Distributed Memory

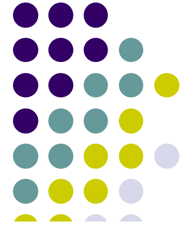


- Distributed memory systems require a communication network to connect inter-processor memory.



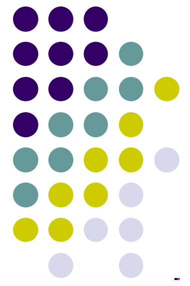
- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.

1) Cluster Computing: Distributed Memory

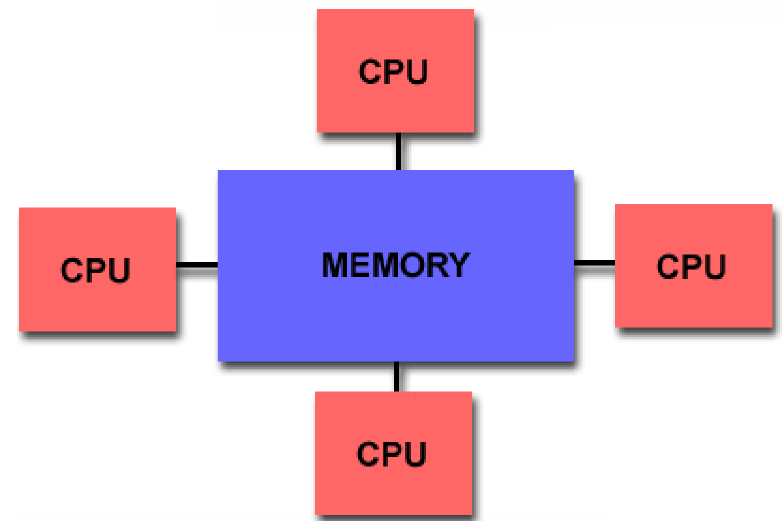


- Advantages:
  - Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
  - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
  - Cost effectiveness: can use commodity, off-the-shelf processors and networking.
- Disadvantages:
  - The programmer is responsible for many of the details associated with data communication between processors.
  - It may be difficult to map existing data structures, based on global memory, to this memory organization.
  - Non-uniform memory access (NUMA) times.

1) Cluster Computing: Shared Memory

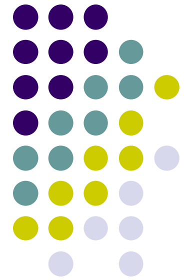


- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.



- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location generated by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: UMA, NUMA.

1) Cluster Computing: Shared Memory

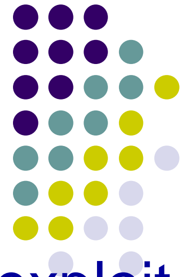


- Advantages Shared Memory:
  - Global address space provides a user-friendly programming perspective to memory.
  - Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs.
- Disadvantages Shared Memory:
  - Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
  - Programmer responsibility for synchronization constructs that insure correct access of global memory.
  - It is difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

1) Cluster Computing: Shared & Distributed Memory

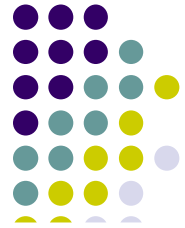
- Advantages:
  - Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
  - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
  - Cost effectiveness: can use commodity, off-the-shelf processors and networking.
- Disadvantages:
  - The programmer is responsible for many of the details associated with data communication between processors.
  - It may be difficult to map existing data structures, based on global memory, to this memory organization.
  - Non-uniform memory access (NUMA) times.

2) Developing Parallel Programs

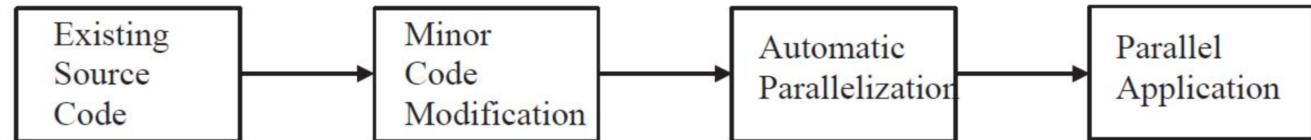


- The goal for developing parallel programs is to efficiently exploit the cluster architecture.
- The programmer is typically responsible for both identifying and actually implementing parallelism (manual process).
- Very often, manually developing parallel codes is a time consuming, complex, error-prone and iterative process.
- For a number of years now, various tools have been available to assist the programmer with converting serial programs into parallel programs.
- The most common type of tool used to automatically parallelize a serial program is a parallelizing compiler or pre-processor.

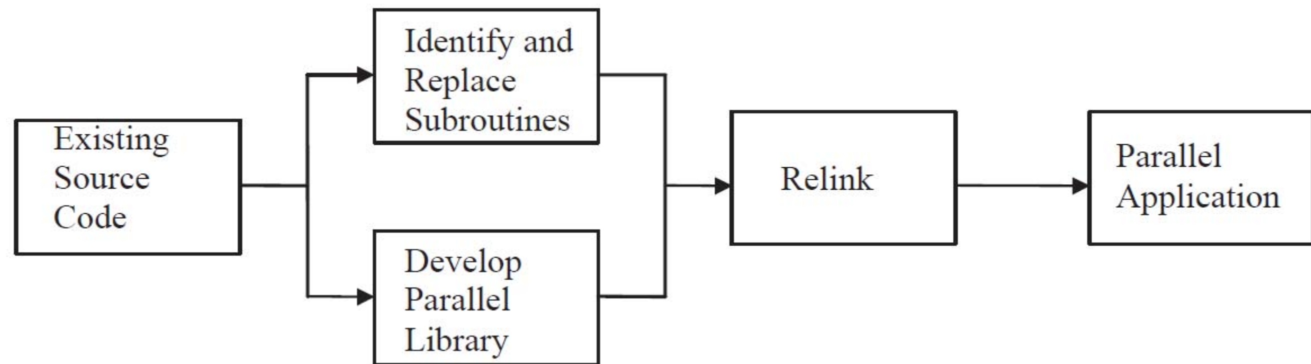
2) Developing Parallel Programs



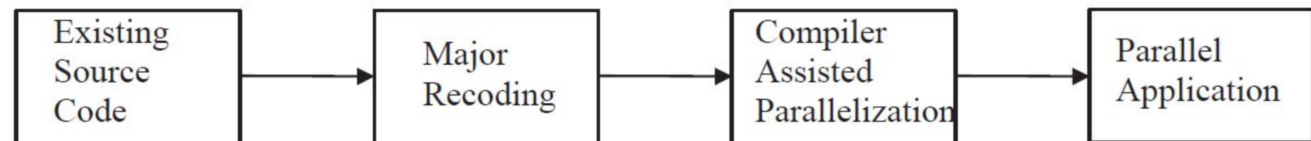
Fine Grain Size
Compiler



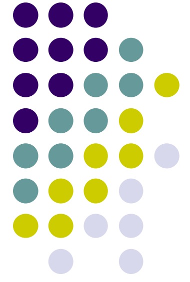
Medium Grain Size
Programmer



Large Grain Size
Programmer

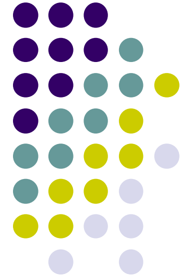


2) Developing Parallel Programs



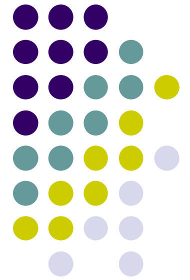
- Understand the Problem and the Program:
  - The first step in developing parallel software is to first understand the problem that you wish to solve in parallel.
  - Determine whether or not the problem is one that can actually be parallelized.
- Identify the program's hotspots:
  - Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places.
  - Profilers and performance analysis tools can help.
  - Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.

2) Developing Parallel Programs



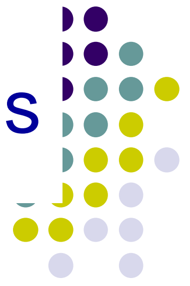
- Identify bottlenecks in the program:
  - Are there areas that are excessively slow, or cause parallelizable work to halt or be deferred? For example: Input/Output.
  - It is possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas ?
  - Identify inhibitors to parallelism. One common class of inhibitor is *data dependence*, as demonstrated by the Fibonacci sequence above.
  - Investigate other algorithms if possible. This may be the single most important consideration when designing a parallel application.

2) Developing Parallel Programs



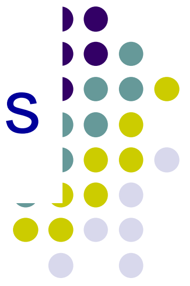
- Partitioning
  - It consist into break the problem into discrete "chunks" of work that can be distributed to multiple tasks.
  - There are two basic ways to partition computational work among parallel tasks: domain and functional decomposition.
- Domain Decomposition:
  - In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of the data.
- Functional Decomposition:
  - In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.

2) Developing Parallel Programs: Communications



1) Cost of communications
   - Inter-task communication generates overhead.
   - Machine cycles and resources that could be used for computation are instead used to package and transmit data.
   - Communications require some type of synchronization between tasks.
   - Communication traffic can saturate the available network bandwidth.
2) Latency v/s Bandwidth:
   - Latency is the time it takes to send a minimal (0 byte) message from point A to point B. It is expressed in microseconds.
   - Bandwidth is the amount of data that can be communicated per unit of time. It is expressed in gigabytes/sec.
   - Sending many small messages cause latency to dominate comm. overheads.
   - To increase the effective communications bandwidth is more efficient to package small messages into a larger message.

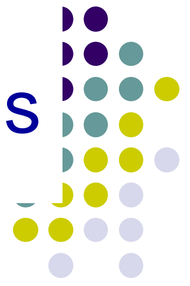
2) Developing Parallel Programs: Communications



3) Synchronous vs. asynchronous communications

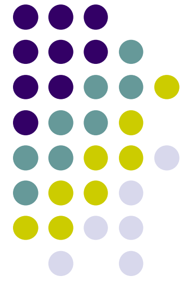
- Synchronous communications require some type of "handshaking" between tasks that are sharing data. This can be explicitly structured in code by the programmer, or it may happen at a lower level.
- Synchronous communications are often referred to as blocking communications since other work must wait until the communications have completed.
- Asynchronous communications are often referred to as non-blocking communications since other work can be done while the communications are taking place.
- Asynchronous communications allow tasks to transfer data independently from one another. For example, task 1 can send a message to task 2, and then immediately begin doing other work.

2) Developing Parallel Programs: Communications



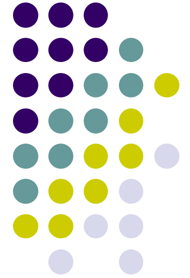
4) Scope of communications
   - Point-to-point: Involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
   - Collective: Involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective. Some common variations (there are more).
5) Efficiency of communications
   - What type of communication operations should be used? Asynchronous communication operations can improve overall program performance.
   - Network media - some platforms may offer more than one network for communications.

2) Developing Parallel Programs: Performance Evaluation



- The operations performed by a parallel algorithm are:
  - Sequential
  - Parallel
  - Overhead:
    - Redundant operations
    - Task start-up and termination times
    - Synchronizations
    - Software overhead imposed by parallel compilers, libraries, tools, operating system, etc; task termination time
- Let $\psi(n, p)$ be the speed-up that is attained with p processors and a problem of size n.

2) Developing Parallel Programs: Performance Evaluation



- Then:

$$\psi(n,p) \le \frac{\sigma(n)+\varphi(n)}{\left(\sigma(n)+\frac{\varphi(n)}{p}+\kappa(n,p)\right)}$$

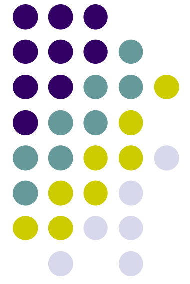
Sequential ops: $\sigma(n)$

Parallel ops: $\varphi(n)$

Overhead: $\kappa(n,p)$

- The maximum theoretical speed-up: $\Psi(\mathrm{n,p}) \le p$
- This speed-up is difficult to obtain because of:
  - Idle cycles of the processors
  - Redundant operations
  - Re-definition of variables
  - Communication time between processors

2) Developing Parallel Programs: Performance Evaluation



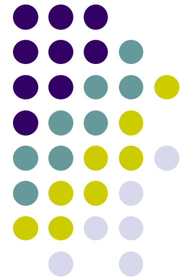
- Efficiency: It is defined as the percentage of time that p processors are used in parallel operations.

$$\varepsilon(n,p) = \frac{t_S}{t_P \times p} \times 100\% \Rightarrow \varepsilon(n,p) = \frac{\psi(n,p)}{p} \times 100\%$$

$$\varepsilon(n,p) \le \frac{\sigma(n) + \varphi(n)}{\left(p\sigma(n) + \varphi(n) + p\kappa(n,p)\right)}$$

- Scalability:
  - Architecture (Hardware): Change of performance as a result of cluster "size" increase (nodes, memory, storage, better network, etc).
  - Algorithmic: Change of performance as a result of problem "size" increase.

2) Developing Parallel Programs: Performance Evaluation



- Amdahl Law:

$$\psi(n,p) \le \frac{\sigma(n)+\varphi(n)}{\left(\sigma(n)+\frac{\varphi(n)}{p}+\kappa(n,p)\right)}$$

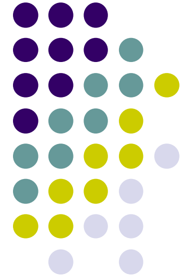
$$\psi_A(n,p) \le \frac{\sigma(n)+\varphi(n)}{\left(\sigma(n)+\frac{\varphi(n)}{p}\right)} \longrightarrow \kappa(n,p)=0$$

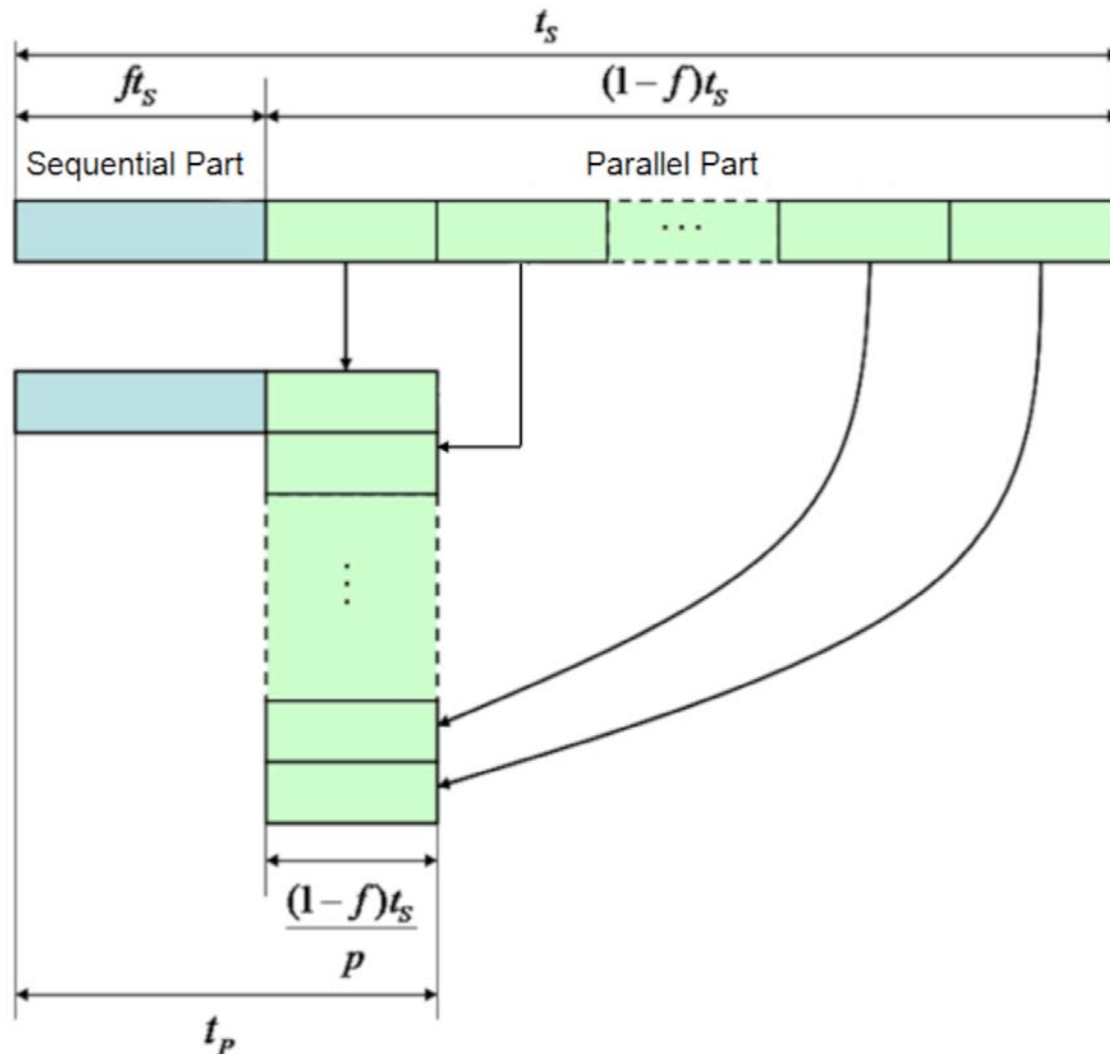
$$\psi_A(n,p) \le \frac{\frac{\sigma(n)}{f}}{\sigma(n)+\frac{\sigma(n)}{p}\left(\frac{1}{f}-1\right)} \quad \text{where } f \doteq \frac{\sigma(n)}{\left(\sigma(n)+\varphi(n)\right)}$$

Sequential fraction of the algorithm

$$\psi_A(n,p)=\psi_A(p,f)=\frac{p}{1+(p-1)f}$$

2) Developing Parallel Programs: Performance Evaluation (Amdahl's Law)





$$S(p) = \frac{t_S}{t_P}$$

$$t_P = ft_S + \frac{(1-f)t_S}{p}$$

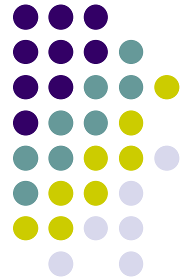
$$\psi_A(p,f) = \frac{p}{1+(p-1)f}$$

$$\lim_{p\to\infty} \psi_A(p,f) = \frac{1}{f}$$

$$\lim_{f\to 0} \psi_A(p,f) = p$$

$$\lim_{f\to 1} \psi_A(p,f) = 1$$

2) Developing Parallel Programs: Performance Evaluation Example

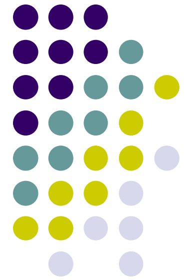


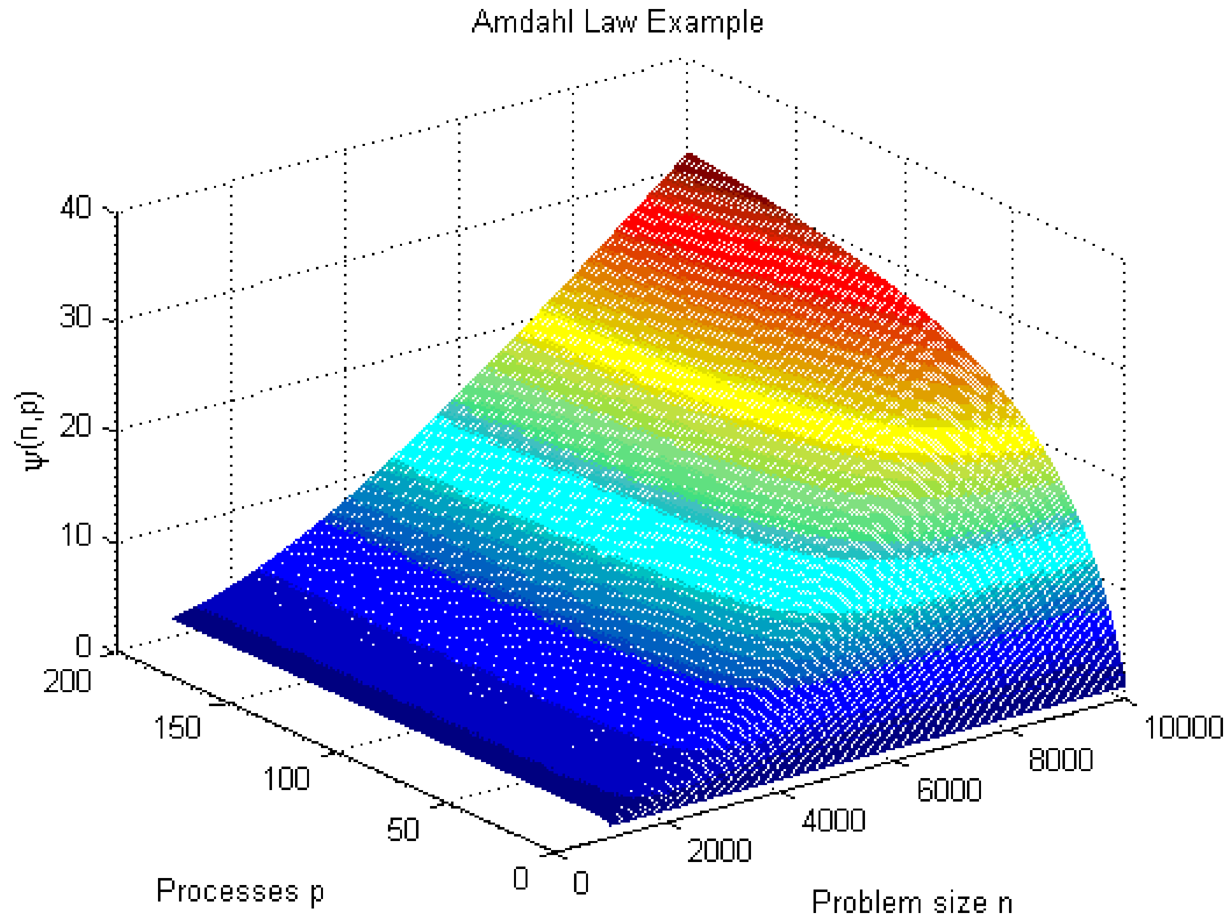
- An example:
- Let us suppose that the parallel fraction of an algorithm has a complexity of $\varphi(n)=n^2/100$ [μs].
- And the sequential fraction (data input, results output) has a complexity $\sigma(n)=(18000+n)$ [μs].
- If n=10000, which is the maximum speed-up according to the Amdahl law ?

$$\psi_A(n,p) \le \frac{\sigma(n)+\varphi(n)}{\left(\sigma(n)+\frac{\varphi(n)}{p}\right)}$$

Amdahl law

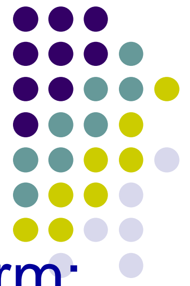
2) Developing Parallel Programs: Performance Evaluation Example





$$\psi_A(n,p) \leq \frac{\left(18000 + n + n^2/100\right)}{\left(18000 + n + n^2/100p\right)}$$

2) Developing Parallel Programs: Performance Evaluation Example



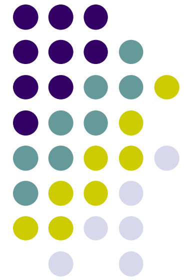
- Let us consider an overhead cost for n=10000 of the form:

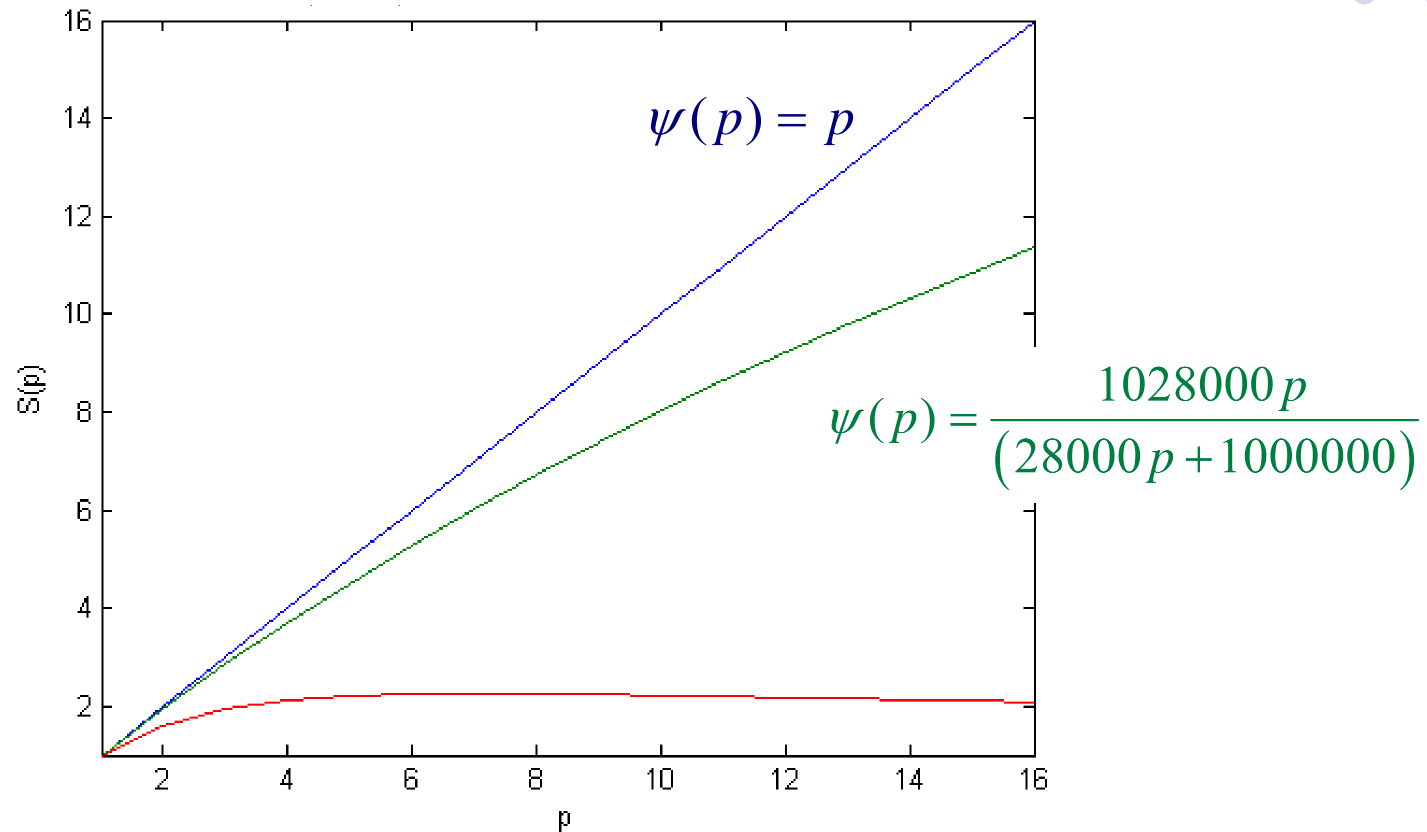
$$\kappa(n=10000,p)=14(10000\lceil \log p \rceil+1000)[\mu s]$$

$$\psi(n=10000,p)\leq \frac{28000+1000000}{28000+\dfrac{1000000}{p}+14\left(10000\lceil \log p \rceil+1000\right)}[\mu s]$$

$$\psi(n=10000,p)\leq \frac{1028000}{42000+\dfrac{1000000}{p}+140000\lceil \log p \rceil}[\mu s]$$

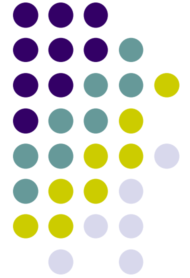
2) Developing Parallel Programs: Performance Evaluation Example





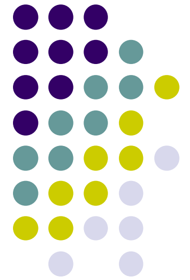
$$\psi(n,p) \le \frac{1028000}{42000 + \frac{1000000}{p} + 140000\lceil \log p \rceil}[\mu s]$$

2) Developing Parallel Programs: Summary



1) Construction of the sequential program
2) Exposing parallelism by means of computation graph
3) Identification of hotspots & bottleneck by profiling
4) Partitioning design: domain and functional decomposition
5) Design and evaluation of communications
6) Performance Evaluation
7) Go Back to 3)

3) Parallel Programming Models: Message Passing - Distributed Memory

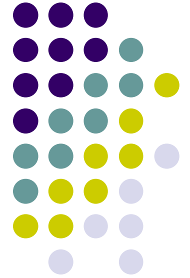


- Advantages:
  - Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
  - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
  - Cost effectiveness: can use commodity, off-the-shelf processors and networking.
- Disadvantages:
  - The programmer is responsible for many of the details associated with data communication between processors.
  - It may be difficult to map existing data structures, based on global memory, to this memory organization.
  - Non-uniform memory access (NUMA) times.

3) Parallel Programming Models: Message Passing

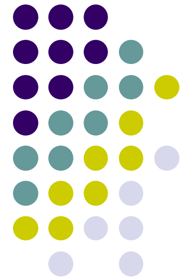
- Characteristics of the message passing model:
  - Assignation of processes to computing nodes: statically (MPI 1.0) or dynamically (MPI 2.0).
  - A set of tasks that use their own local memory during computation.
  - Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines.
  - Tasks exchange data through communications by sending and receiving messages. These communications can be point to point or collective.
  - Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

3) Parallel Programming Models: Message Passing - MPI Hello World



```
#include "mpi.h"
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
int numprocs, myrank, namelen, i;
char processor_name[MPI_MAX_PROCESSOR_NAME], greeting[200];
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
MPI_Get_processor_name(processor_name,&namelen);
sprintf(greeting, "hello, world from process %d of %d running on
    %s",myrank,numprocs,processor_name);
```

3) Parallel Programming Models: Message Passing - MPI Hello World

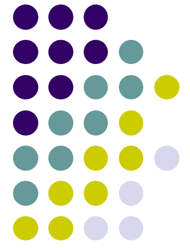


```
if (myrank == 0)
{
  printf("%s\n", greeting);
  for (i=1;i<numprocs;i++)
  {
    MPI_Recv(greeting,sizeof(greeting),MPI_CHAR,i,1,MPI_COMM_WORLD,
    &status);
    printf("%s\n",greeting);
  }
}
else
    MPI_Send(greeting,strlen(greeting)+1,MPI_CHAR,0,1,MPI_COMM_WORLD);
MPI_Finalize();
return 0;
}
```

3) Parallel Programming Models: Message Passing

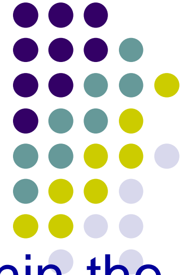
- Implementations:
  - From a programming perspective, message passing implementations comprise a library of subroutines that are imbedded in source code.
  - The programmer is responsible for determining all parallelism.
  - Part 1 of the Message Passing Interface (MPI) was released in 1994. Part 2 (MPI-2) was released in 1996. Both MPI specifications are available on the web at www.mcs.anl.gov/Projects/mpi/standard.html
  - MPI is now the 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.

3) Parallel Programming Models: Threads



- In the threads model of parallel programming, a single process can have multiple, concurrent execution paths.
- An analogy that can be used to describe threads is the concept of a single program that includes a number of subroutines:
  - The main program (e.g. 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 (called 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.

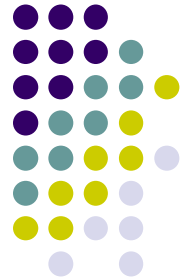
3) Parallel Programming Models: Threads



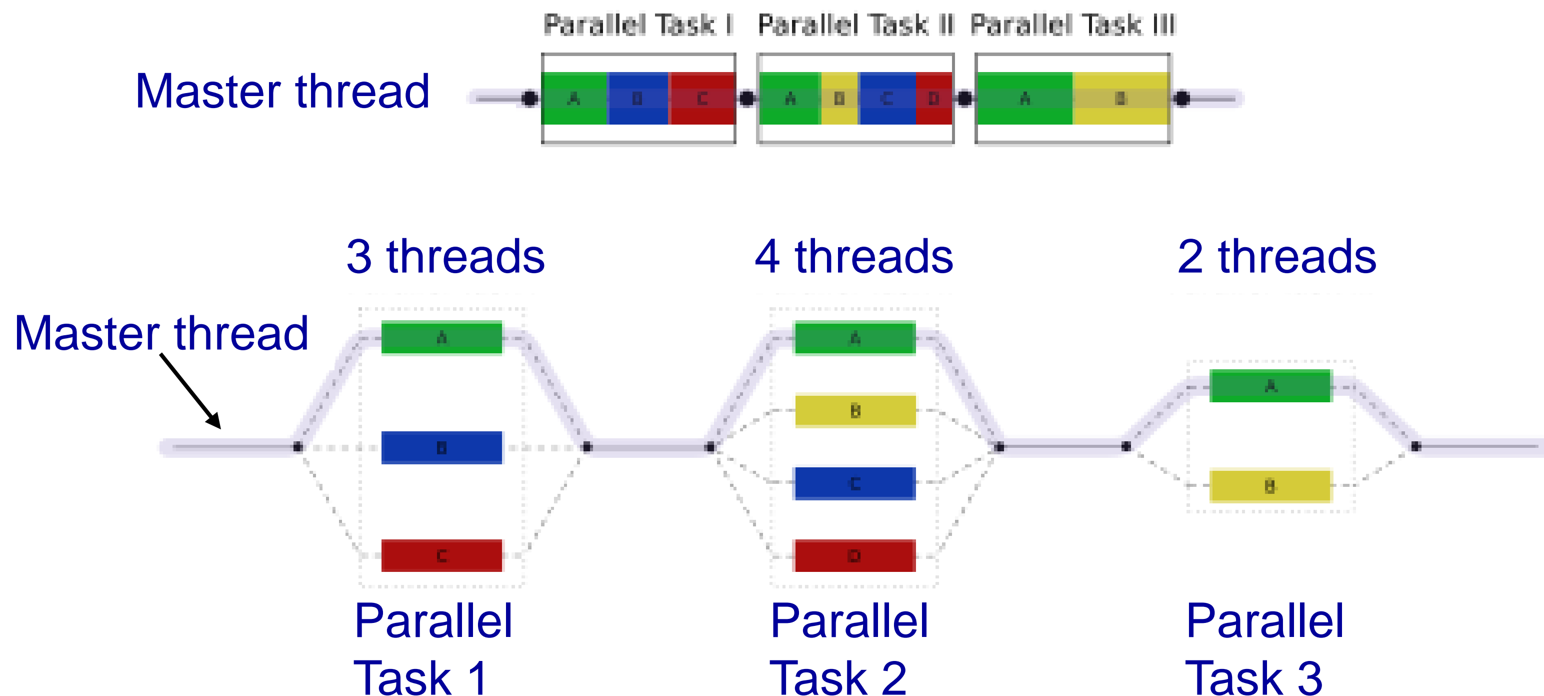
- A thread's work may best be described as a subroutine within the main program. Any thread can execute any subroutine at the same time as other threads.
- Threads communicate with each other through global memory (updating address locations). This requires synchronization constructs to insure that more 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.

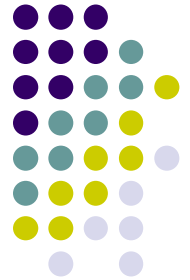
3) Parallel Programming Models: Threads



- Fork-join parallelism:
  - Master thread spawns a team of threads as needed.
  - Parallelism is added incrementally: the sequential program evolves into a parallel program.

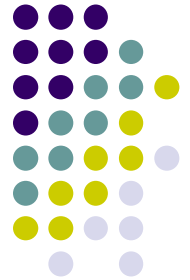


3) Parallel Programming Models: Threads



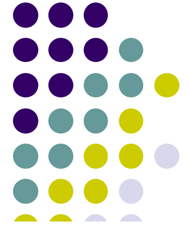
- Implementations:
- From a programming perspective, threads implementations commonly comprise:
  - A library of subroutines that are called from within parallel source code.
  - A set of compiler directives imbedded in either serial or parallel code.
  - A set of environment variables.
- The programmer 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.

3) Parallel Programming Models: Threads



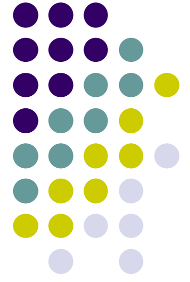
- POSIX Threads
  - Library based; requires parallel coding (IEEE POSIX 1003.1c 1995).
  - Most hardware vendors now offer C Language Pthreads in addition to their proprietary threads implementations.
  - Very explicit parallelism; requires significant attention to detail.
- OpenMP: Open Multi-Processing
  - 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 .
  - Can be very simple to use - provides for "incremental parallelism“.
  - Microsoft has its own implementation for threads, which is not related to the UNIX POSIX or OpenMP.

3) Parallel Programming Models: Threads OpenMP Hello World



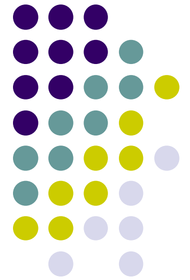
```
#include <stdio.h>
#include <omp.h>
#define Num_Threads 8 // Number of threads
int main()
{
int nprocs, nt, max_nt;
double start_time, elapsed_time;
start_time=omp_get_wtime();
nprocs = omp_get_num_procs();  // Determine number of processors
omp_set_num_threads(Num_Threads); // Setting the number of threads
nt = omp_get_num_threads(); // Determine the number of threads
#pragma omp parallel
{
    printf("Hello World from thread %d of %d\n",omp_get_thread_num(),nt);
}
elapsed_time = omp_get_wtime() - start_time;
printf("Elapsed time is: %f\n",elapsed_time);
}
```

3) Parallel Programming Models: Data Parallel



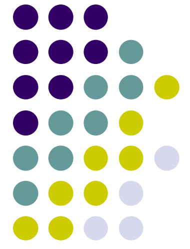
- Characteristics of the data parallel model:
  - Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.
  - A set of tasks work collectively on the same data structure, however, each task works 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.

3) Parallel Programming Models: Data Parallel



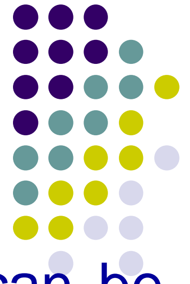
- Implementations:
  - Programming with the data parallel model is accomplished by writing a program with data parallel constructs. The constructs can be calls to a data parallel subroutine library or, compiler directives recognized by a data parallel compiler.
  - Fortran 90 and 95 (F90, F95): ISO/ANSI standard extensions to Fortran 77.
    - New source code format; additions to character set
    - Additions to program structure and commands
    - Variable additions - methods and arguments
    - Pointers and dynamic memory allocation added
    - Array processing (arrays treated as objects) added
    - Recursive and new intrinsic functions added
  - High Performance Fortran (HPF): Extensions to Fortran 90 to support data parallel programming.

3) Parallel Programming Models: Hybrid



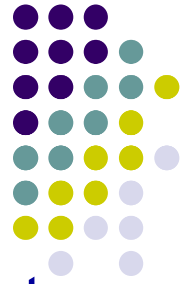
- In this model, any two or more parallel programming models are combined.
- A common example of a hybrid model is the combination of the message passing model (MPI) with the shared memory model (OpenMP). This hybrid model lends itself well to the increasingly common hardware environment of networked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

3) Parallel Programming Models: SPMD Single Program Multiple Data



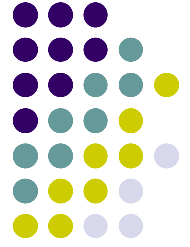
- SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- A single program is executed by all tasks simultaneously.
- At any moment in time, tasks can be executing the same or different instructions within the 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.
- All tasks may use different data.

3) Parallel Programming Models: MPMD Multiple Program Multiple Data



- 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.

References



1) Dongarra, J., I. Foster (Eds.), The Sourcebook of Parallel Computing, Morgan Kaufmann, 2002.
2) Hoffmann, K.H., A. Meyer, Parallel Algorithms and Cluster Computing: Implementations, Algorithms and Applications, Springer, 2006.
3) Pacheco, P., Parallel Programming with MPI, M. Kaufmann, 1997.
4) Quinn, M.J., Parallel Programming in C with MPI and OpenMP, McGraw-Hill, 2004.
5) Tutorial on Parallel Computing and Programming, Lawrence Livermore National Laboratory, 2007.