Parallel Programming

Distributed Computing Group
Polytechnic School
University of Lleida

ESCOLA
POLITÈCNICA SUPERIOR
UNIVERSITAT DE LLEIDA

Francesc Giné
Francesc Giné
Josep Lluis
2022
Lérida

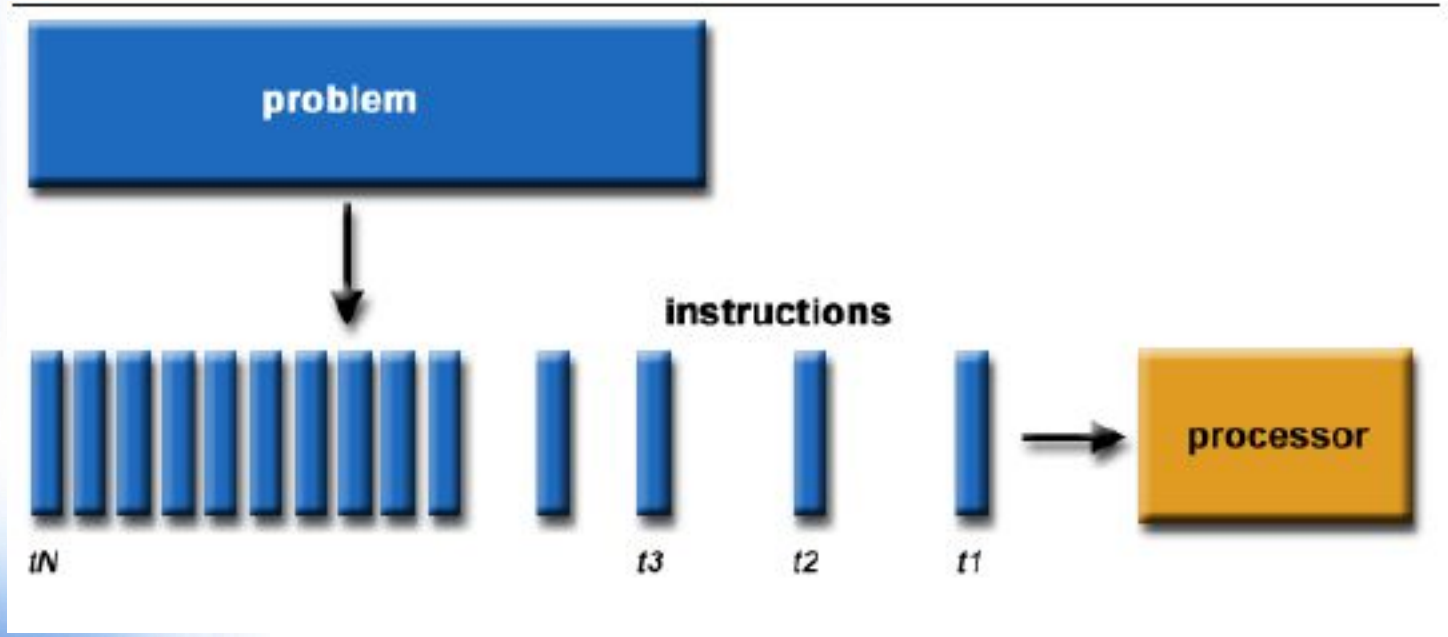
Outline

- Introduction
- Parallel Computer Memory Architectures
- Parallel Programming Models
- Designing Parallel Programs
- Performance Index
- Benchmarking
- Parallel Programming Paradigm
- Parallel Programming Examples

What is parallel computing?

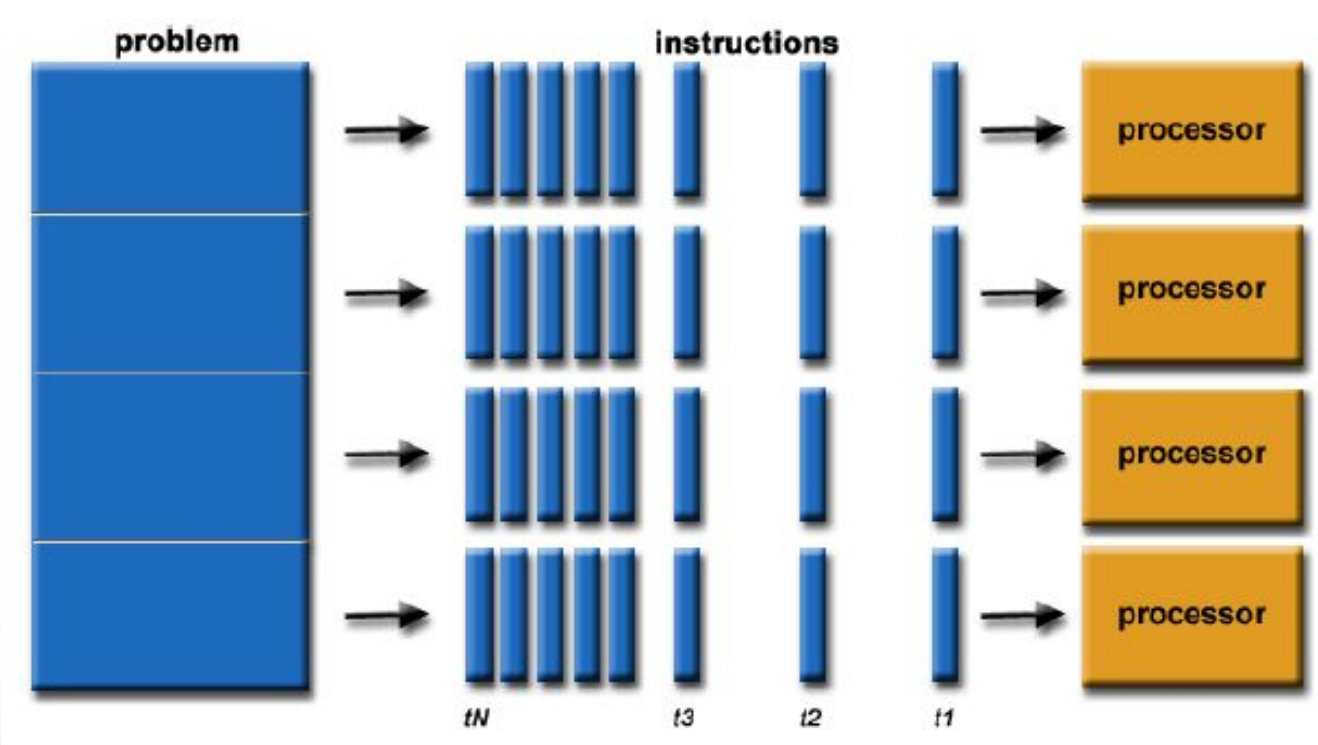
Traditionally, software has been written for ***serial*** computation:

- To be executed by a single computer having a single Central Processing Unit (CPU);
- Problems are solved by a series of instructions, executed one after the other by the CPU. Only one instruction may be executed at any moment in time.



What is parallel computing?

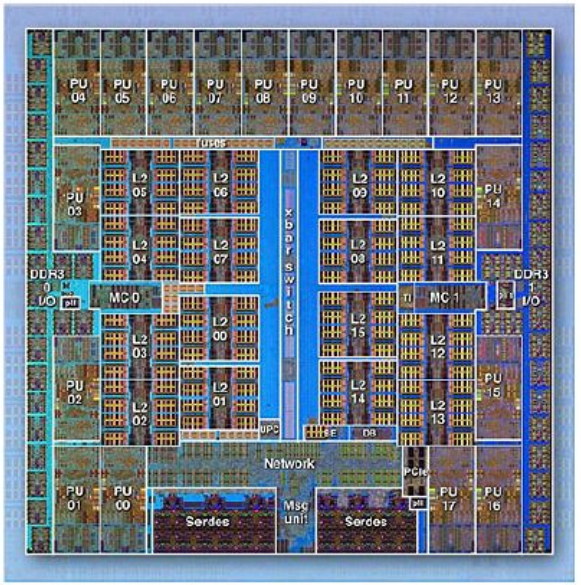
In the simplest sense, ***parallel computing*** is the simultaneous use of multiple computational resources to solve a computational problem.



What is parallel computing?

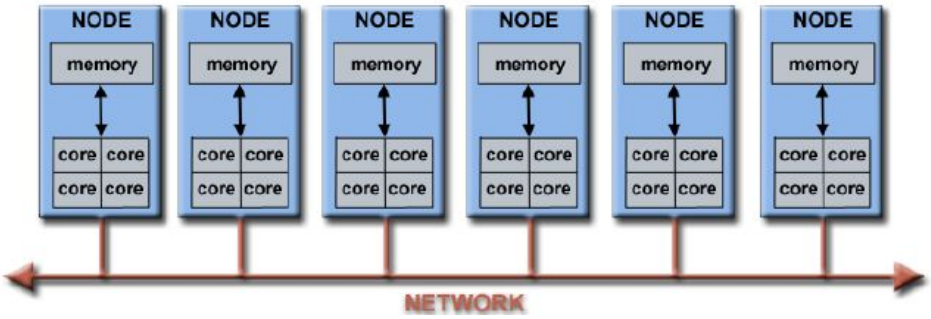
The computational resources can include:

- A single computer with multiple processors (multiprocessor);



IBM BG/Q Compute Chip with 18 cores (PU) and 16 L2 Cache units (L2)

- An arbitrary number of computers connected by a network (cluster, cloud computing, P2P computing, grid computing);



- A combination of both (cluster , cloud computing, P2P computing, grid computing).

Why use parallel computing?

There are two primary reasons for using parallel computing:

- **Save time** - wall clock time: This is the elapsed time between the start of and final execution. It includes the CPU time and waiting time.
- **Solve larger problems**

Other reasons might include:

- **Taking advantage of non-local resources** - using available compute resources on a wide area network, or even the Internet when local compute resources are scarce.
- **Cost savings** - using multiple "cheap" computing resources instead of paying for time on a supercomputer.
- **Overcoming memory constraints** - single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle.

Kind of Parallelisms

- **Implicit Parallelism**

The development of parallel application is complex and requires long time. In order to make easier this task, there are some tools to help to the programmer to translate serial programs into parallel programs.

These tools are compilers, which are able to identify the implicit parallelism of the program, generating the parallel code and making the mapping and scheduling of this code.

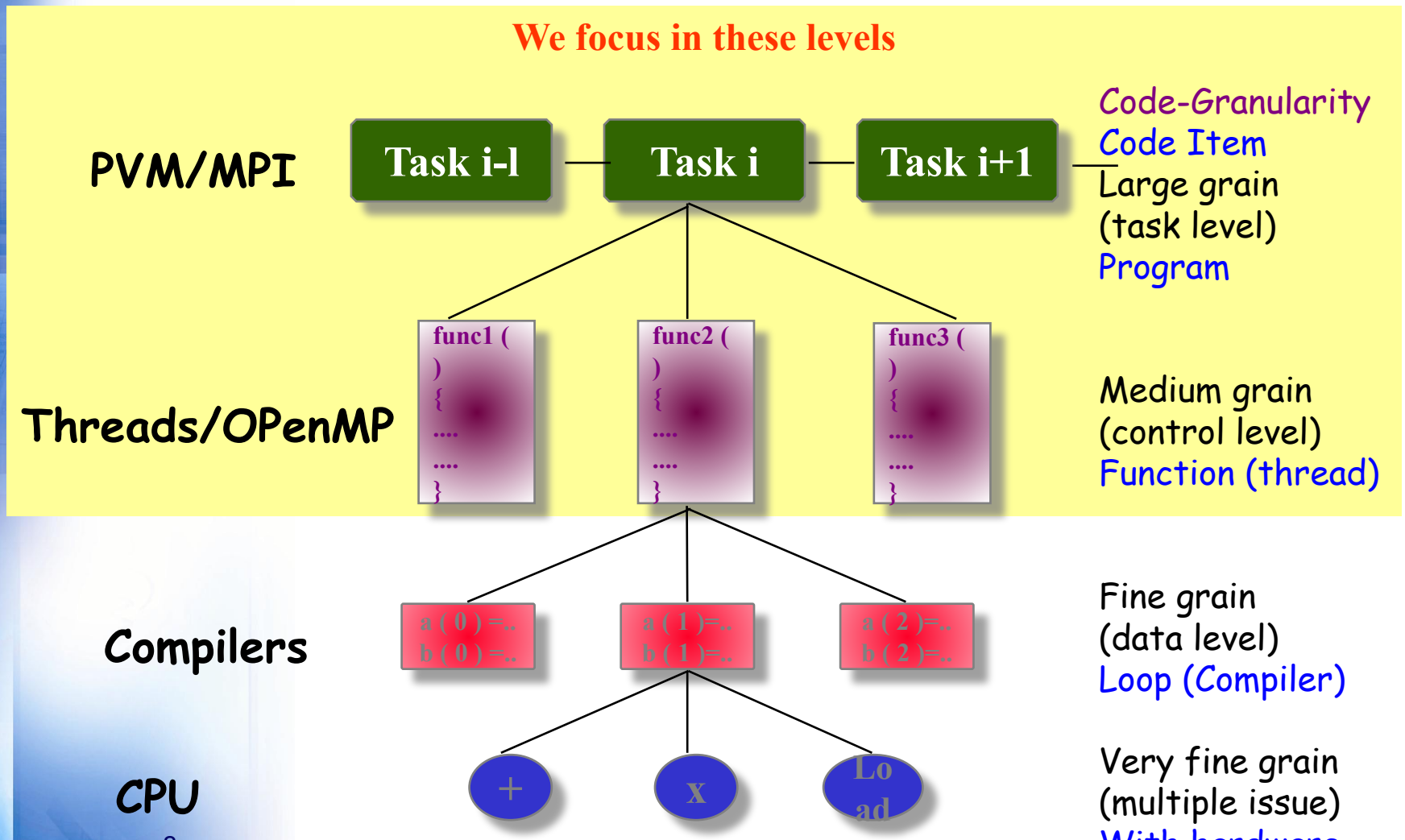
- **Explicit Parallelism**

Aim of the subject

In this approach, the programmer has the responsibility of the parallelization of the application.

This is based on the approach that the user is who knows better his application and as a consequence, he is the best to exploit its parallelism.

Level of Parallelism

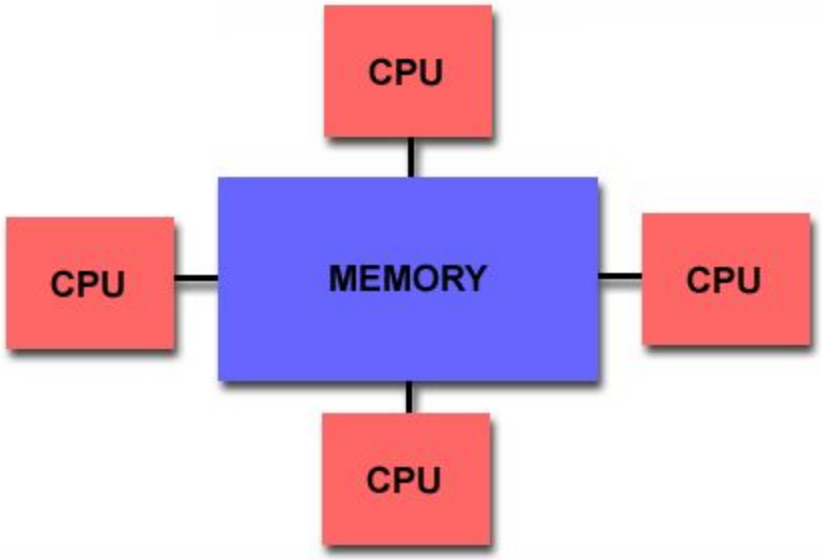


Outline

- Introduction
- Parallel Computer Memory Architectures
- Parallel Programming Models
- Designing Parallel Programs
- Performance Index
- Benchmarking
- Parallel Programming Paradigm
- Parallel Programming Examples

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 effected 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*** and ***NUMA***.

Shared Memory

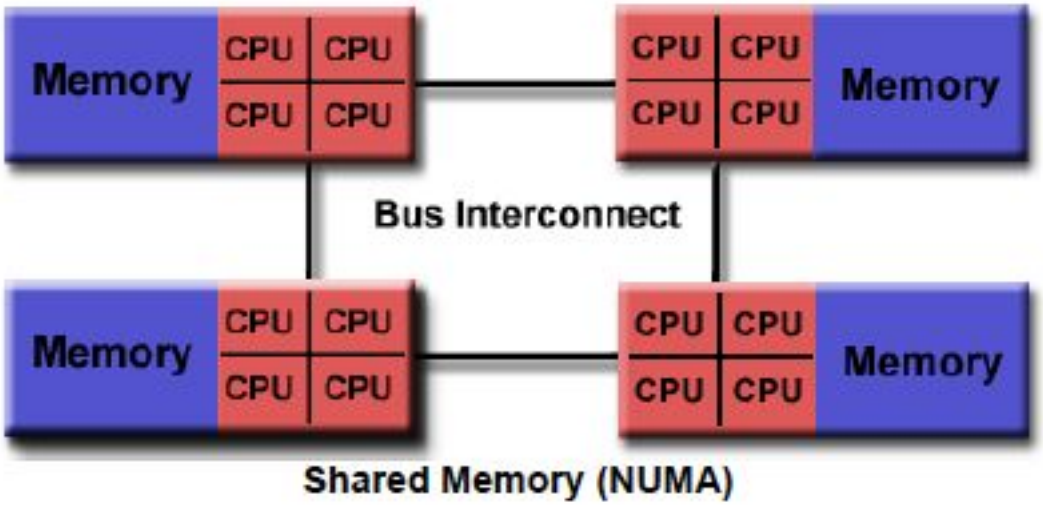
Uniform Memory Access (UMA):

- Most commonly represented today by Symmetric Multiprocessor (SMP) machines
- Identical processors
- Equal access and access times to memory
- Sometimes called CC-UMA (Cache Coherent UMA).
- **Cache coherent** means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.

Shared Memory

Non-Uniform Memory Access (NUMA):

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- **Not all processors have equal access time** to all memories
- If cache coherency is maintained, then may also be called CC-NUMA- Cache Coherent NUMA



Shared Memory

Advantages:

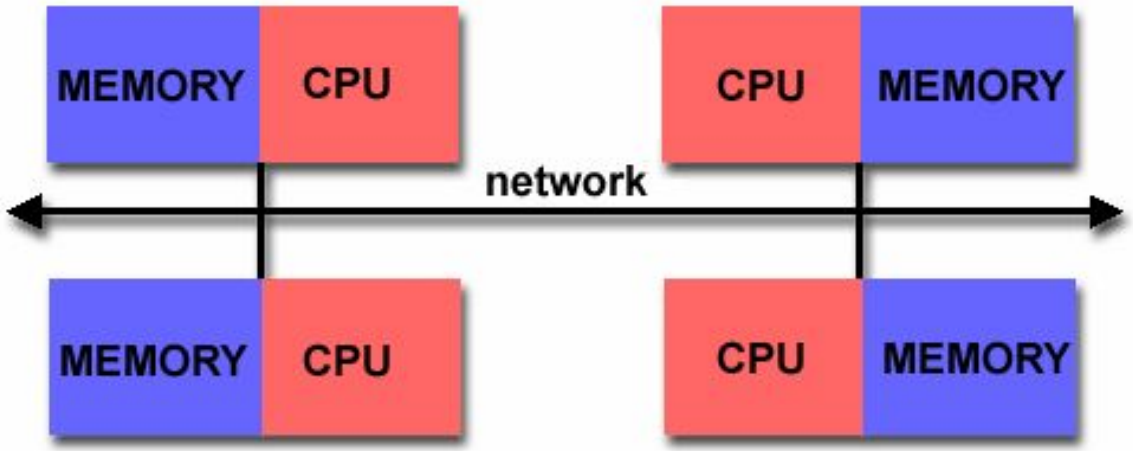
- Global address space provides **a user-friendly programming** perspective to memory
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

Disadvantages:

- Primary disadvantage is the **lack of scalability** between memory and CPUs. 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.
- Expensive: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

Distributed Memory

- Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect inter-processor memory.



- The network "fabric" used for data transfer varies widely, though it can can be as simple as Ethernet.

Distributed 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.

  Every change that it makes to its local memory have no effect on the memory of other processors.

  Hence, the concept of cache coherency does not apply.
- **When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated.**

  Synchronization between tasks is likewise the programmer's responsibility.

Distributed Memory

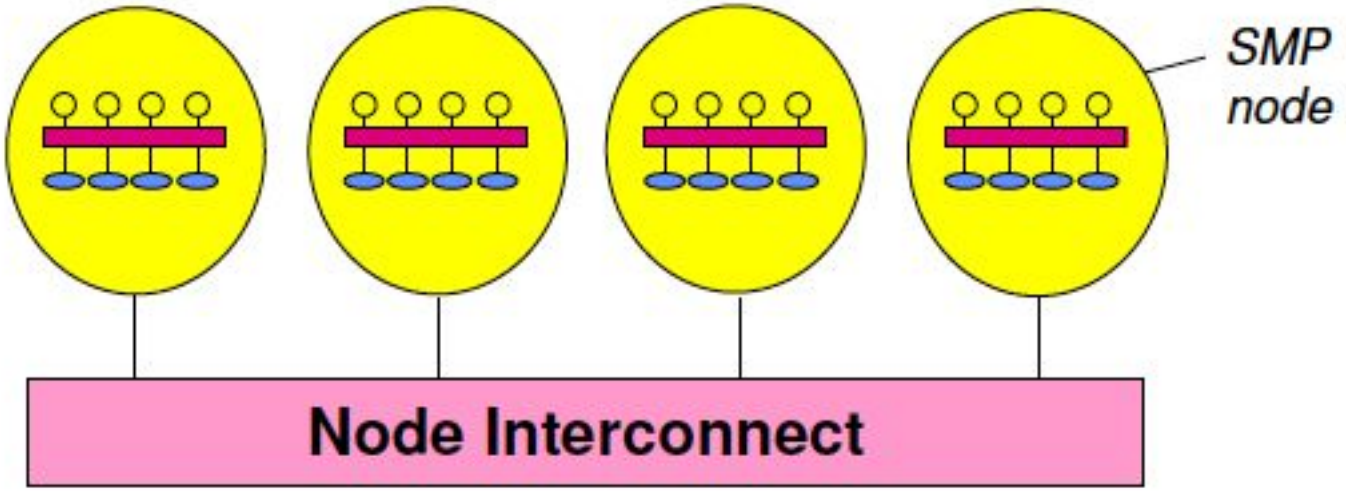
Advantages:

- **Memory is scalable** with number of processors. Increases 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

Hybrid Distributed-Shared Memory



- Most modern high performance computing (HPC) systems are clusters of SMP nodes.
- SMP (symmetric multi-processing) inside of each node.
- DMP (Distributed Memory Parallelization) on the node interconnect.

Outline

- Introduction
- Parallel Computer Memory Architectures
- Parallel Programming Models
- Designing Parallel Programs
- Performance Index
- Benchmarking
- Parallel Programming Paradigm
- Parallel Programming Examples

Parallel Programming Models

- Two major resources of computation:
  - processor
  - memory
- Parallelization means
  - **distributing work** to processors
  - **distributing data** (if memory is distributed)

  and
  - **synchronization** of the distributed work
  - **communication** of remote data to local processor (if memory is distributed)
- Programming models offer a combined method for
  - distribution of work & data, synchronization and communication

Distributing Work & Data

Work decomposition

- based on loop decomposition

do i=1,100
→ i=1,25
i=26,50
i=51,75
i=76,100

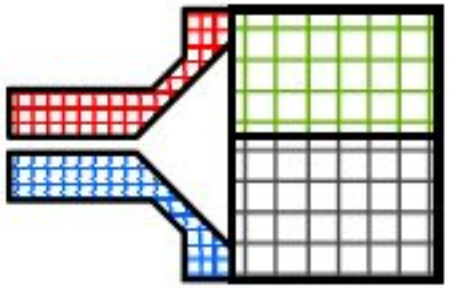
Data decomposition

- all work for a local portion of the data is done by the local processor

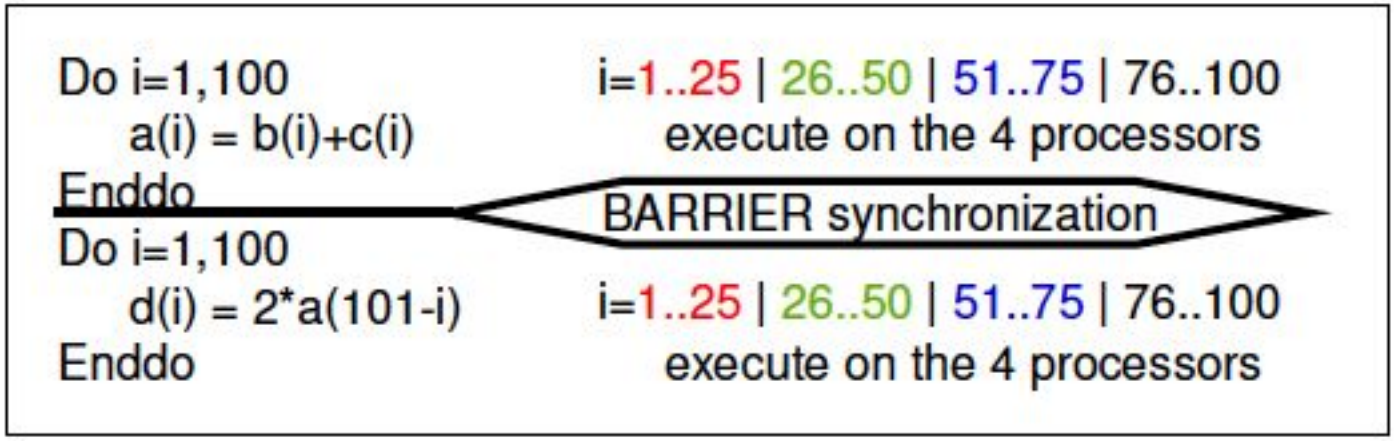
A(1:20, 1: 50)
A(1:20, 51:100)
A(21:40, 1: 50)
A(21:40, 51:100)

Domain decomposition

- decomposition of work and data is done in a higher model, e.g. in the reality



Synchronization



Synchronization

- is necessary
- may cause
  - idle time on some processors
  - overhead to execute the synchronization primitive

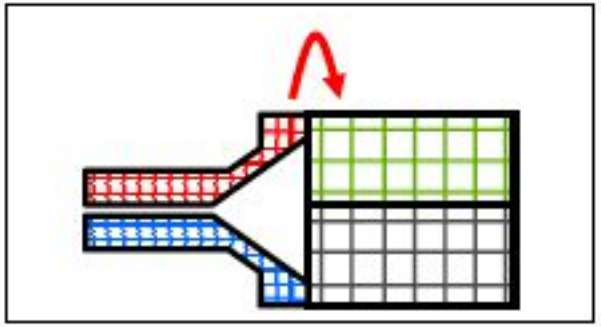
Communication

```
Do i=2,99
    b(i) = a(i) + f*(a(i-1)+a(i+1)-2*a(i))
Enddo
```

- **Communication** is necessary on the boundaries
  - e.g. b(26) = a(26) + f*(a(25)+a(27)-2*a(26))

a(1:25),	b(1:25)
a(26,50),	b(51,50)
a(51,75),	b(51,75)
a(76,100),	b(76,100)

  - e.g. at domain boundaries



Major Programming Models

OpenMP

- Shared Memory Directives
- Define Work Composition
- No Data Decomposition
- Implicit Synchronization (can also be user defined)

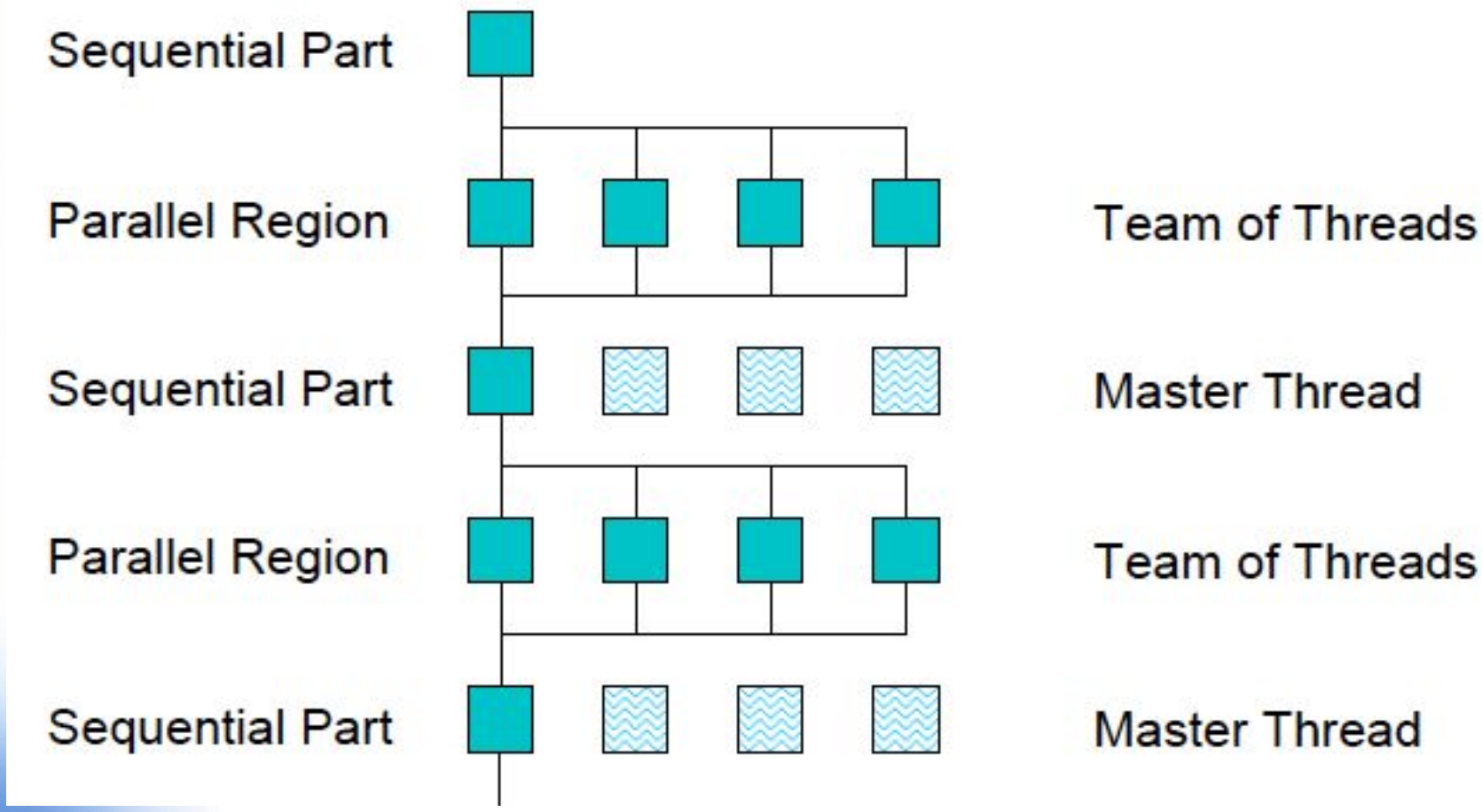
MPI (Message Passing Interface)

- User specifies how work & data is distributed
- User specifies how and when the communication is done
- MPI communication library routines

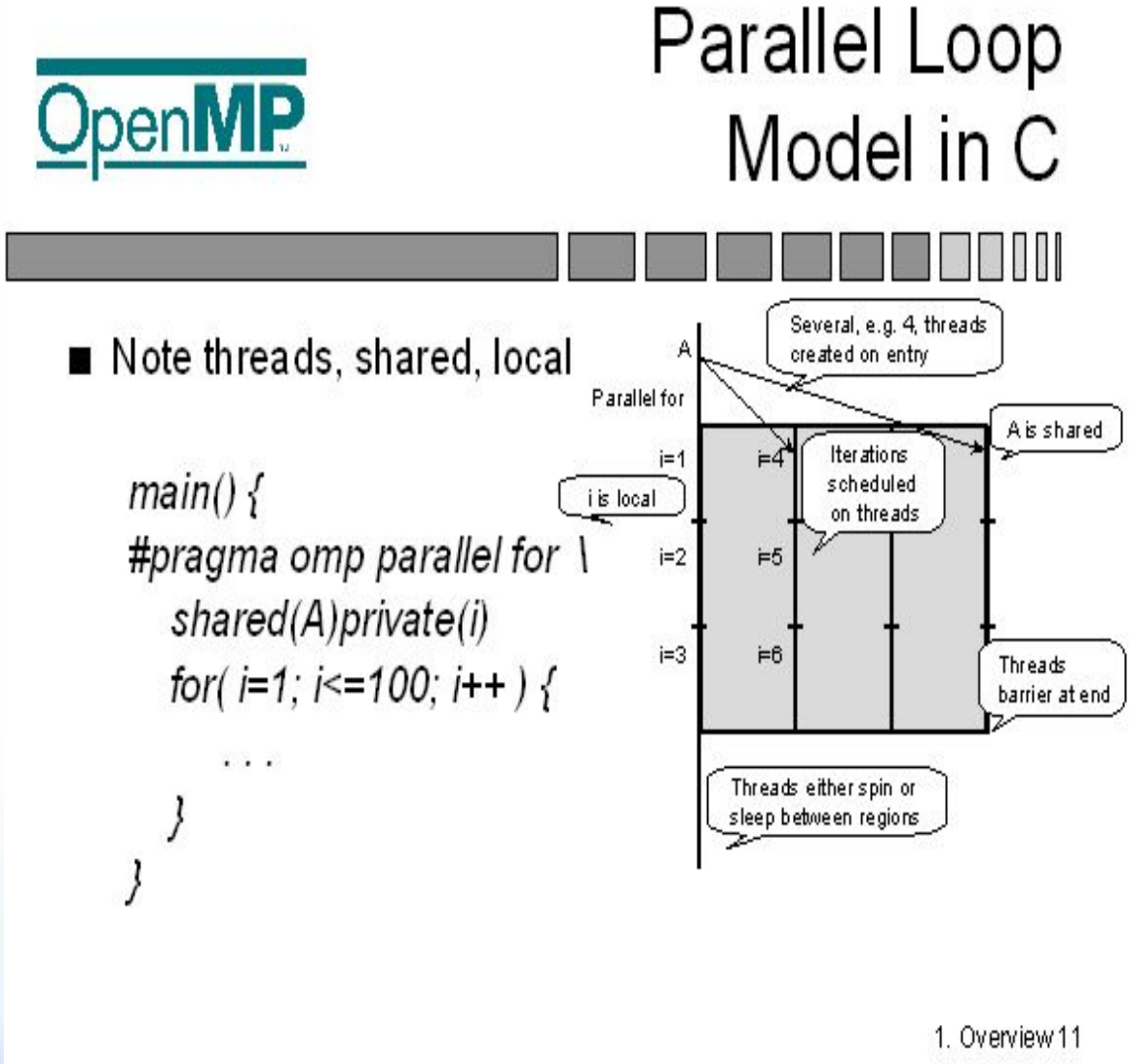
OpenMP based on Work Decomposition

- OpenMP is a **standard programming model** for shared memory parallel programming
- **Threads** based.
- **Portable** across all shared-memory architectures
- It allows incremental parallelization
- Compiler based extensions to existing programming languages
  - mainly by directives
  - a few library routines
- Fortran and C/C++ binding
- OpenMP is a standard
- **Threads communicate by sharing variables.**

Threads Model (OpenMP)



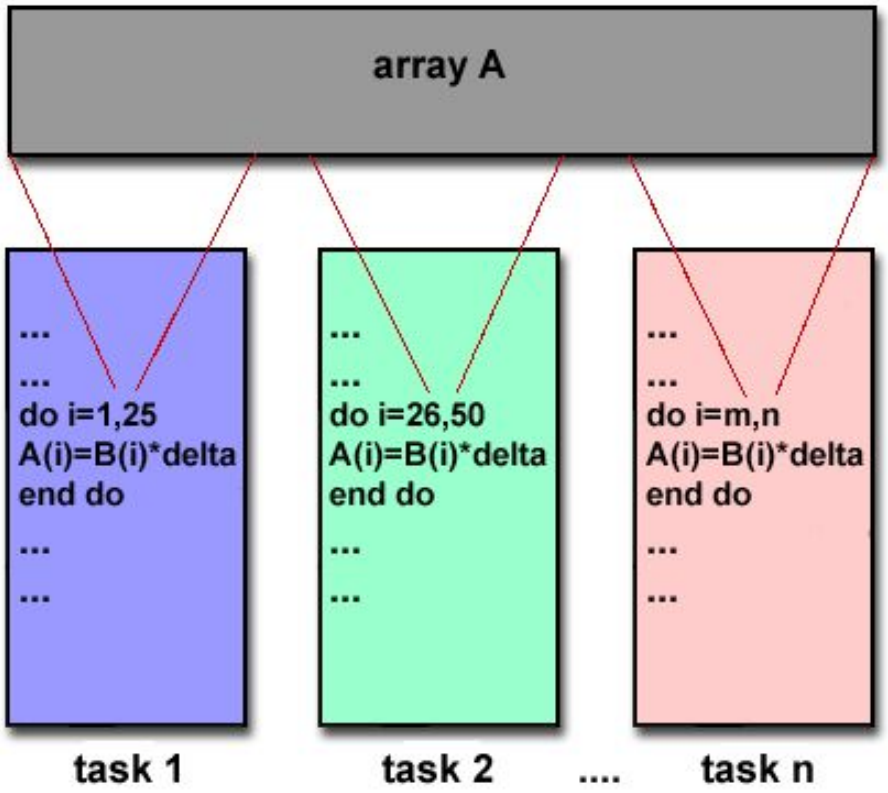
OpenMP Examples



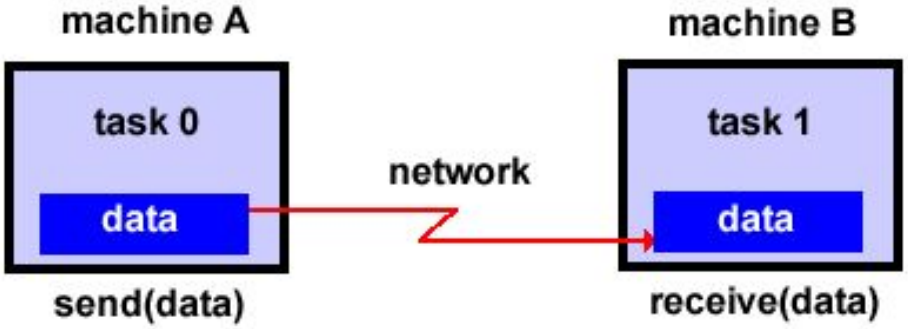
Data Parallel Model

- 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.



Distributed Memory - Message Passing Model



- 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 Implementation

MPI

MPI (Message Passing Interface)

– standardized **distributed memory parallelism** with message passing

– process-based

– the user has to specify **the work distribution & data distribution & all communication**

– **synchronization implicit/explicit** by completion of communication

– the application processes are calling **MPI library-routines**

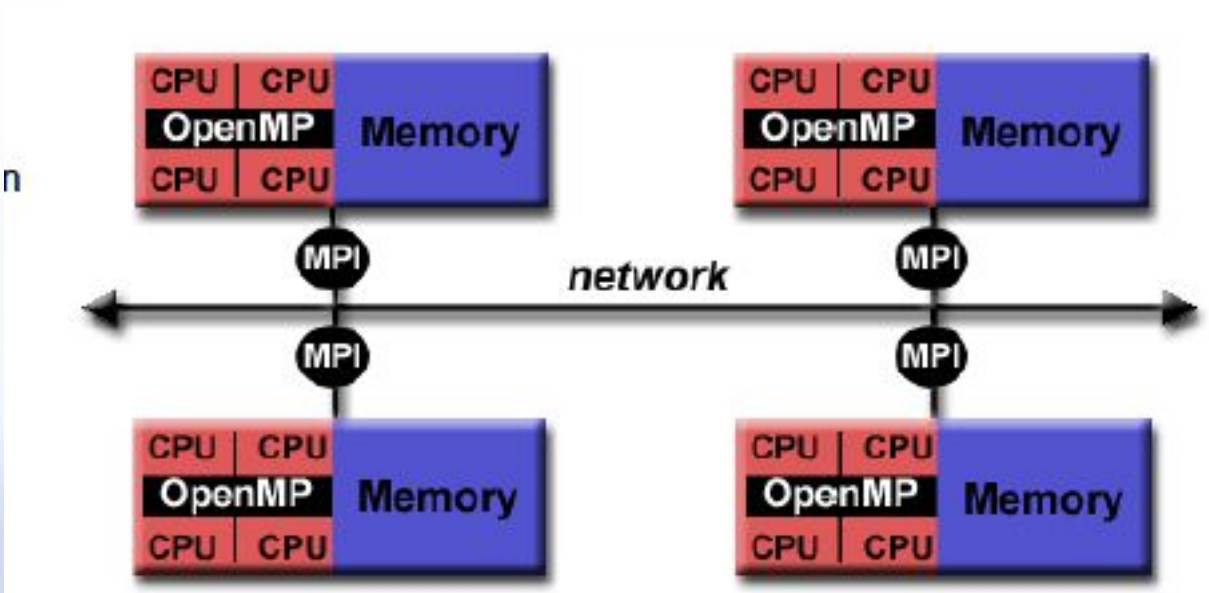
– **standardized**

MPI-1: Version 1.0 (1994), 1.1 (1995), 1.2 (1997)

MPI-2: since 1997

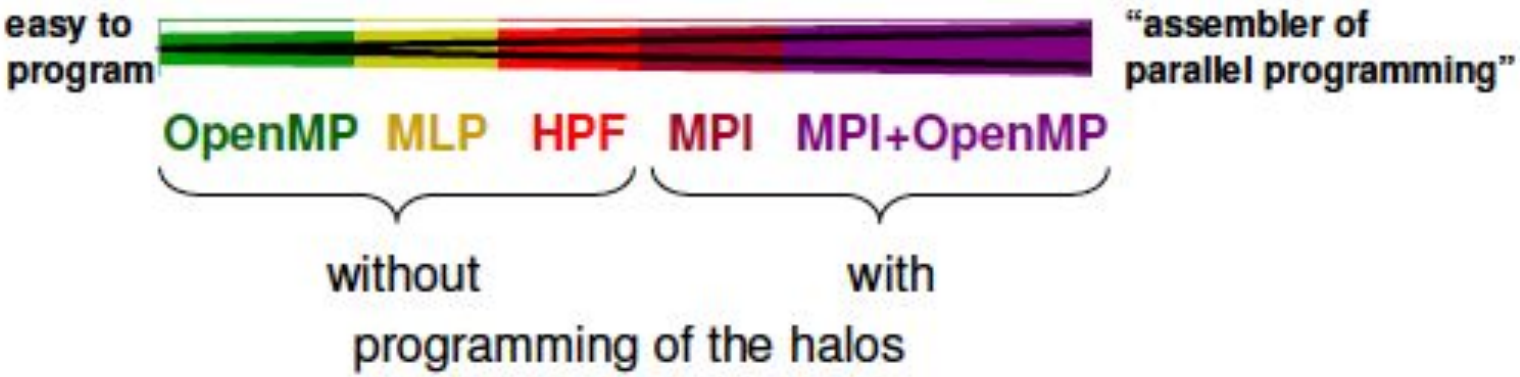
Hybrid Model

- A hybrid model combines more than one of the previous programming models.
- Currently, a typical example of a hybrid model is the combination of OpenMP and MPI:
  - Threads perform computationally intensive kernels using local data
  - Communication between tasks executed on different nodes happens over the network using MPI.



Which Model is the Best for Me?

- Depends on
  - your application
  - your platform
  - which efficiency do you need on your platform
  - how much time do you want to spent on parallelization



Outline

- Introduction
- Parallel Computer Memory Architectures
- Parallel Programming Models
- Designing Parallel Programs
- Performance Index
- Benchmarking
- Parallel Programming Paradigm
- Parallel Programming Examples

Understand the Problem and the Program

Undoubtedly, the first step in developing parallel software is to **understand the problem** that you wish to solve in parallel. If you are starting with a serial program, this necessitates understanding the existing code also. Before spending time in an attempt to develop a parallel solution for a problem,

determine whether or not the problem is one that can actually be parallelized.

Understand the Problem and the Program

- 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 here
  Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.
- Identify ***bottlenecks*** in the program
  Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.
  May be 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.

Partitioning

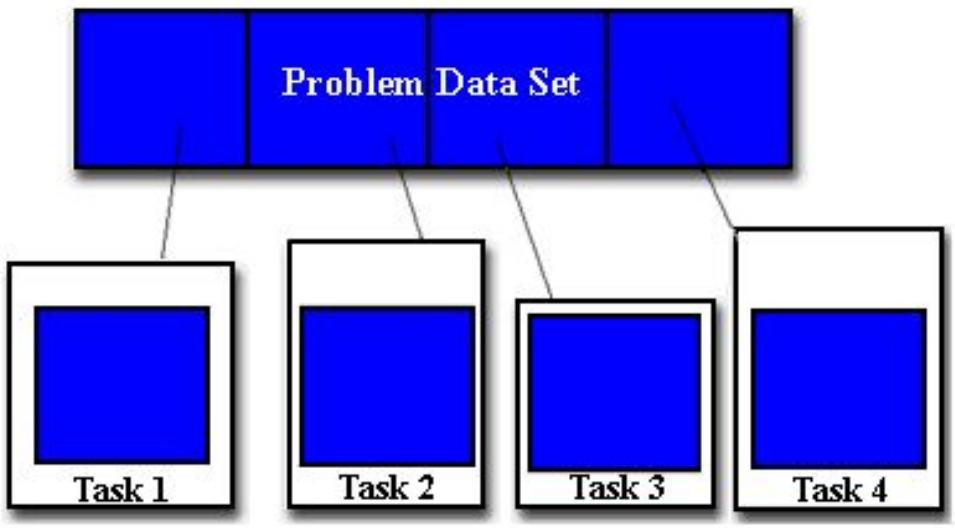
- One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as decomposition or partitioning.

- There are two basic ways to partition computational work among parallel tasks: ***data decomposition*** and ***domain decomposition***.

Partitioning

Data 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.

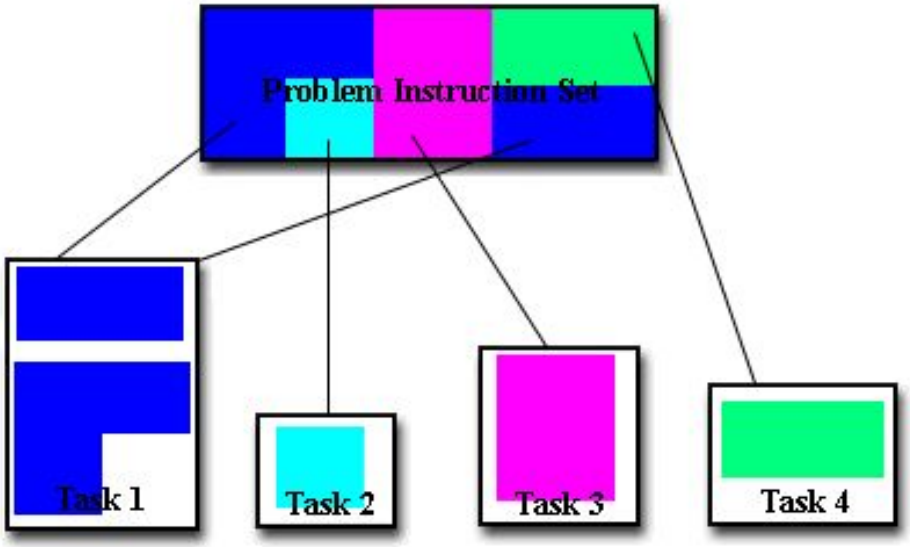


Partitioning

Domain 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.



Communications

Who Needs Communications?

The need for communications between tasks depends upon your problem:

You DON'T need communications

- Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data. For example, imagine an image processing operation where every pixel in a black and white image needs to have its color reversed. The image data can easily be distributed to multiple tasks that then act independently of each other to do their portion of the work.
- These types of problems are often called ***embarrassingly parallel*** because they are so straight-forward. Very little inter-task communication is required.

You DO need communications

- Most parallel applications are not quite so simple, and do require tasks to share data with each other. For example, a 3-D heat diffusion problem requires a task to know the temperatures calculated by the tasks that have neighboring data. Changes to neighboring data has a direct effect on that task's data.

Communications

Factors to Consider:

Cost of communications

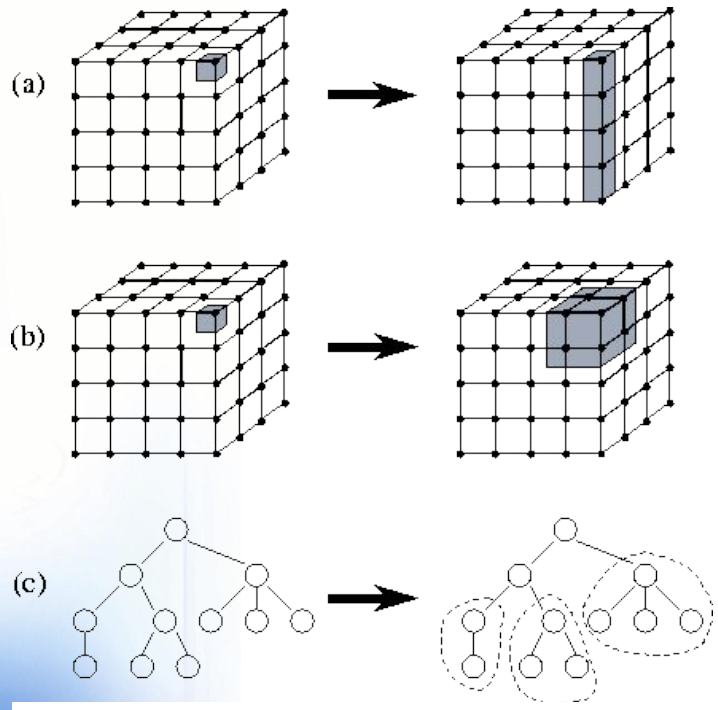
- Inter-task communication virtually always implies overhead.
- Machine cycles and resources that could be used for computation are instead used to package and transmit data.
- Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work.
- Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems.

Latency vs. Bandwidth

- ***latency*** is the time it takes to send a minimal (0 byte) message from point A to point B. Commonly expressed as microseconds.
- ***bandwidth*** is the amount of data that can be communicated per unit of time. Commonly expressed as Megabytes/sec.
- Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.

Gathering Tasks

This level analyzes the need for combining tasks with the aim of reducing the number of tasks, each one with a largest size.



a) The size of the tasks is increased by means of reducing the dimension from three to two.

b) Adjacent tasks are combined with the aim of reducing the granularity.

c) The subtrees are gathered.

Gathering Tasks: Dependencies

Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism. In order to gather tasks, dependencies must be taken into account.

In order to gather tasks efficiently, we must take into account the following aspects:

- Decreasing communication (**Granularity**)
- Create groups of tasks with similar computational cost (**Load balancing**)
- Reduction of **scalability**. The reduction of the number of tasks can provoke a limitation on the scalability.

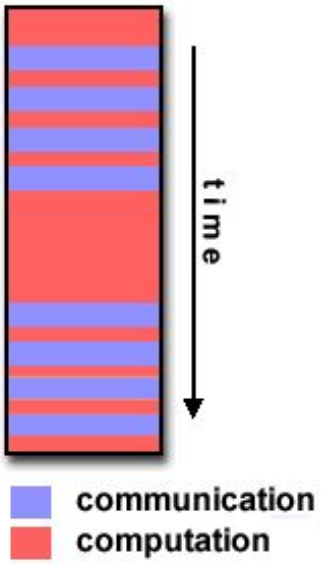
Granularity

Computation / Communication Ratio:

- In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
- Periods of computation are typically separated from periods of communication by synchronization events.

Fine-grain Parallelism:

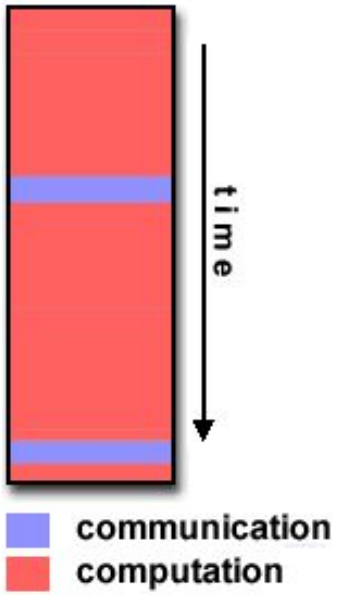
- Relatively small amounts of computational work are done between communication events
- Low computation to communication ratio
- Facilitates load balancing
- Implies high communication overhead and less opportunity for performance enhancement
- If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.



Granularity

Coarse-grain Parallelism:

- Relatively large amounts of computational work are done between communication/synchronization events
- High computation to communication ratio
- Implies more opportunity for performance increase
- Harder to load balance efficiently



Which is Best?

- The most efficient granularity is dependent on the algorithm and the hardware environment in which it runs.
- In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.
- Fine-grain parallelism can help reduce overheads due to load imbalance.

Mapping

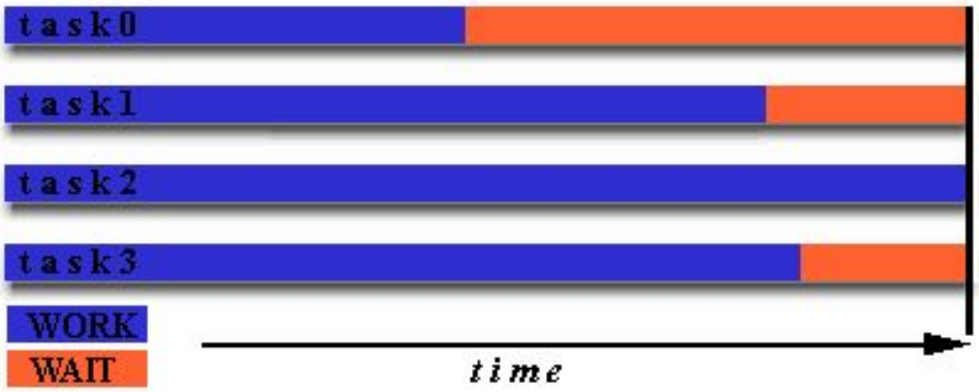
- **Mapping** is the process to assign the tasks to the processors making up the parallel computer.
- The tasks that can be executed concurrently should be assigned to different processors.
- It is important to assign to the same processor those tasks which communicates frequently. Thus the locality is improved.
- The mapping problem is NP.

Mapping - Load Balancing

- **Load balancing** refers to the practice of distributing work among processors so that ***all*** processors are kept busy ***all*** of the time. It can be considered a minimization of task idle time.

- Load balancing is important to parallel programs for performance reasons.

For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.



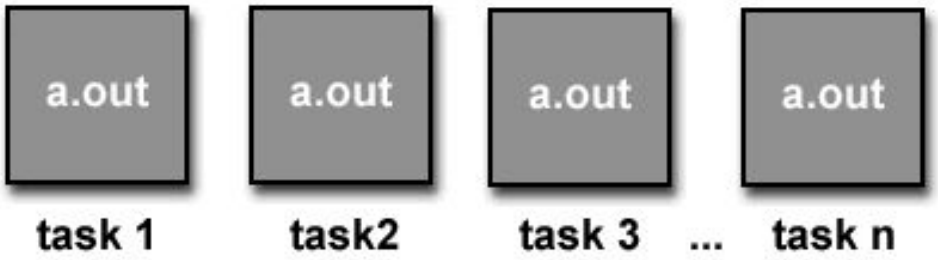
Outline

- Introduction
- Parallel Computer Memory Architectures
- Parallel Programming Models
- Designing Parallel Programs
- Parallel Programming Paradigm
- Parallel Programming Examples
- Performance Index
- Benchmarking

Parallel Programming Paradigm

- Single-Program Multiple-Data (SPMD)
- Multiple-Program Multiple-Data (MPMD)
- Master-Worker Model
- Divide and Conquer

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

Single-Program Multiple-Data (SPMD)

- Each task executes the same code, but over different data.
- Tasks can exchange information each other and they take the following steps:
  - Data Distribution
  - Computation
  - Communication and Synchronization.
  - Collection of results.

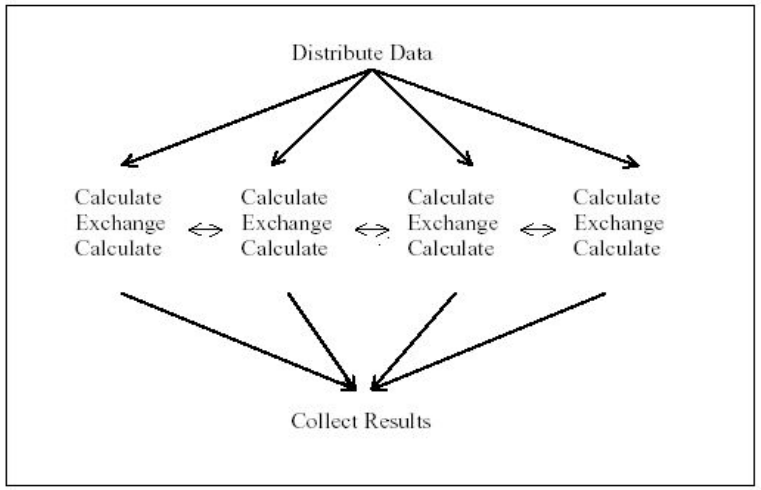
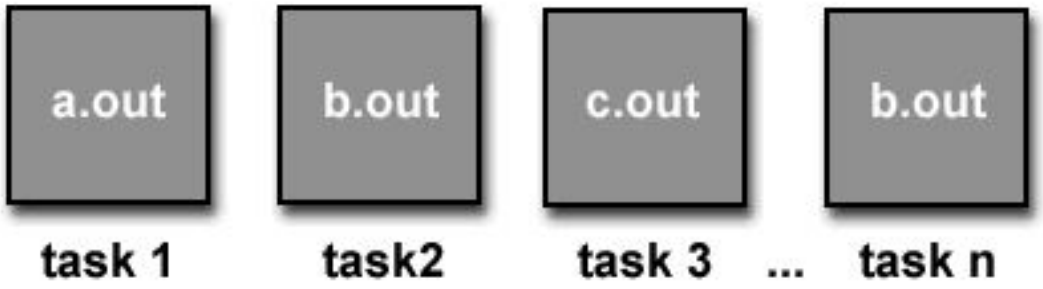


Figure 1.5 Basic structure of a SPMD program.

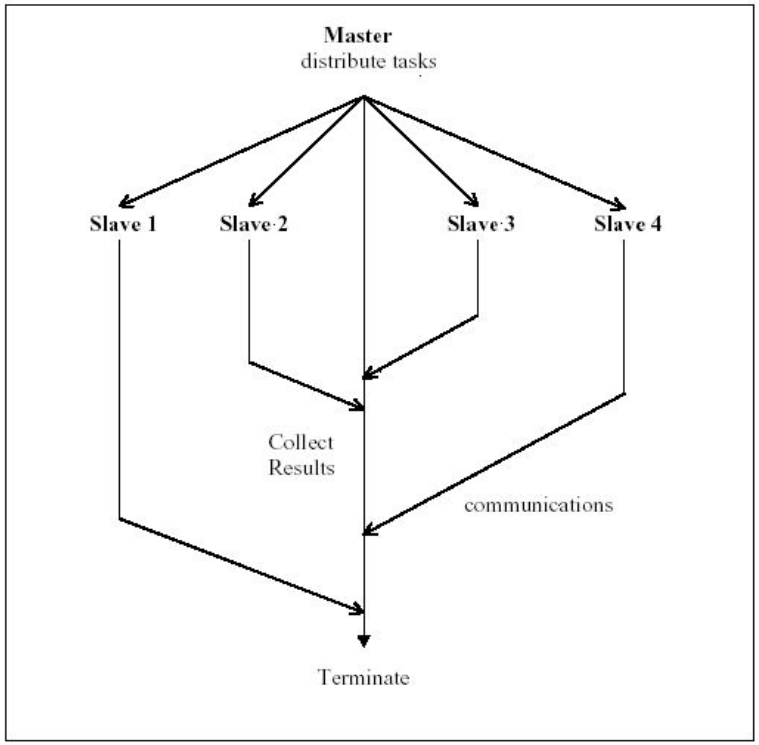
Multiple Program Multiple Data (MPMD)



- 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

Master/Slave Paradigm

- The Master process splits the problem up into small tasks, which are distributed between the workers, and collect the partial results to generate the final result.
- Each worker executes its task independently of the others workers.
- Problems of load balancing.



Divide and Conquer

- The problem is divided recursively between different sub-problems.
- Each sub-problem is solved in an independent way and the results are combined between them.
- Three operations:
  - Split
  - Compute
  - Join
- Problems:
  - Load balancing
  - Granularity

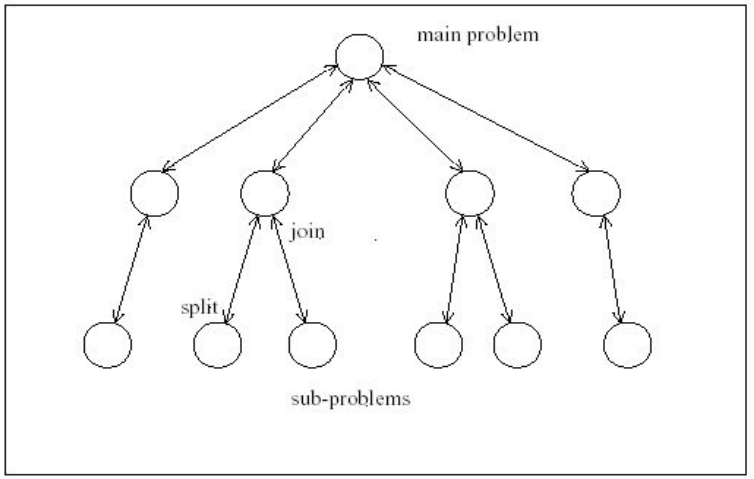


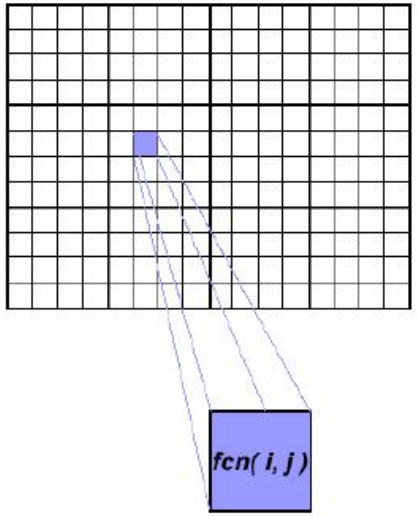
Figure 1.7 Divide and conquer as a virtual tree.

Outline

- Introduction
- Parallel Computer Memory Architectures
- Parallel Programming Models
- Designing Parallel Programs
- Parallel Programming Paradigm
- Parallel Programming Examples
- Performance Index
- Benchmarking

Ex 1. Array Element Evaluation

```
float f(row, col){
  b = do sth;
  return b;
}

main(){
for i in 1...N
  for j in 1...N
      a[i,j] = f(i,j)
}
```



- Is this problem able to be parallelized?
- How would the problem be partitioned?
- Are communications needed?
- Are there any data dependencies?
- Are there synchronization needs?
- Will load balancing be a concern?

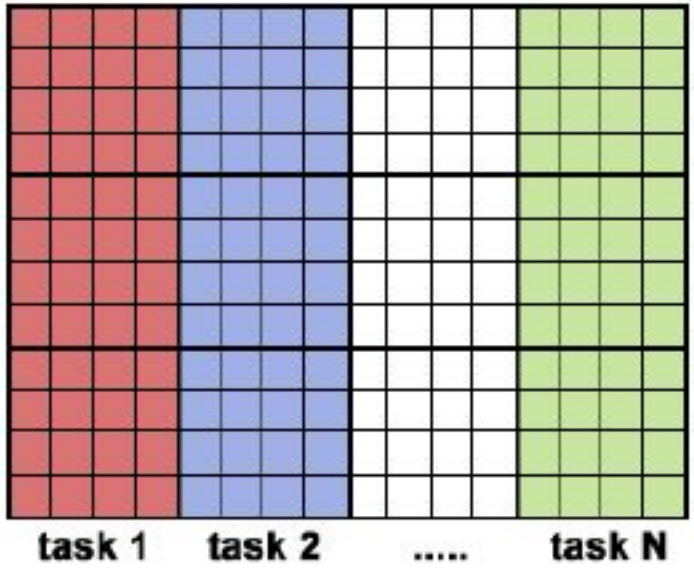
Ex 1. Array Element Evaluation

Solution:

- The calculation of elements is independent of one another – it leads to an embarrassingly parallel solution.
- Arrays element are evenly distributed so that each process own a portion of the array.
- Independent calculation of each array element ensures there is no needed for communication or synchronization between tasks.
- Since he amount of work is evenly distributed across processors, there should no be load balancing concerns.

Ex 1. Array Element Evaluation

Data distribution between tasks:



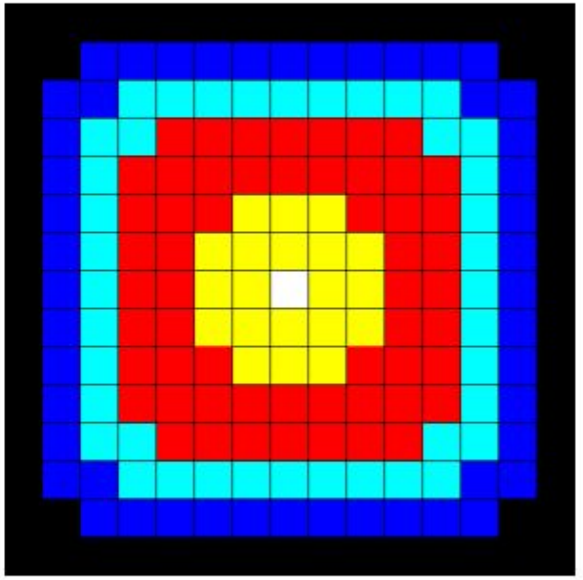

```
find out if I am MASTER or WORKER

if I am MASTER

  initialize the array
  send each WORKER info on part of array it owns
  send each WORKER its portion of initial array

  # calculate my portion of array
  do j = my first column,my last column
    do i = 1,n
      a(i,j) = fcn(i,j)
    end do
  end do

  receive from each WORKER results

else if I am WORKER
  receive from MASTER info on part of array I own
  receive from MASTER my portion of initial array

  # calculate my portion of array
  do j = my first column,my last column
    do i = 1,n
      a(i,j) = fcn(i,j)
    end do
  end do

  send MASTER results

endif
```

https://hpc.llnl.gov/training/tutorials

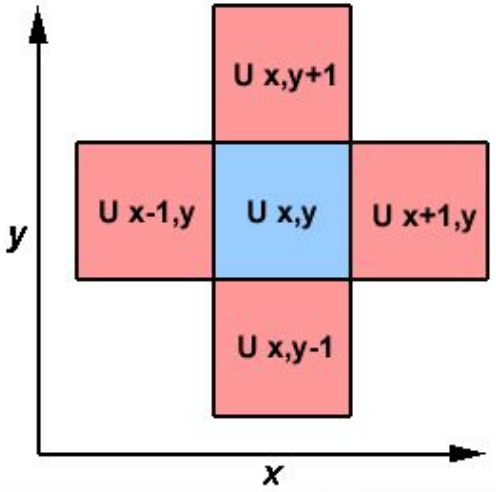
Ex2. Simple Heat Equation

- The 2-D heat equation describes the temperature change over time, given initial temperature distribution and boundary conditions.
- The calculation of an element is dependent upon neighbour element values:

$$U_{x,y} = U_{x,y} + C_x(U_{x+1,y} + U_{x-1,y} - 2U_{x,y}) + C_y(U_{x,y+1} + U_{x,y-1} - 2U_{x,y})$$

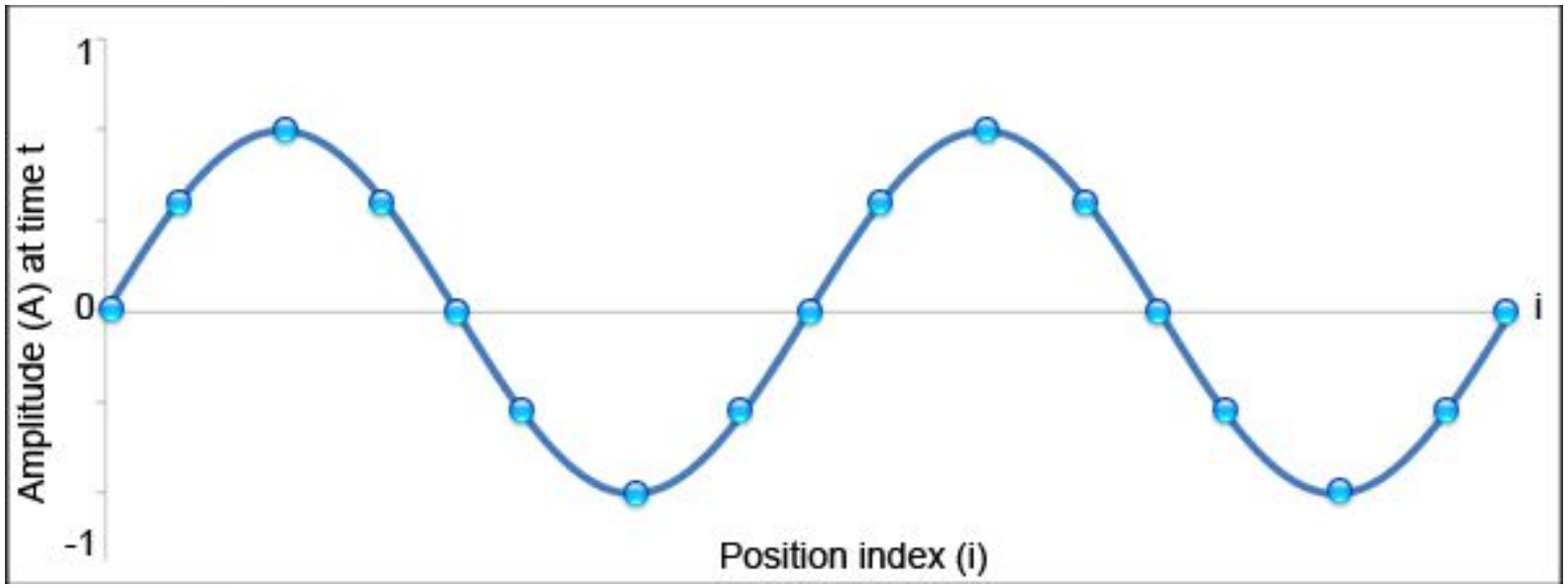


- Is this problem able to be parallelized?
- How would the problem be partitioned?
- Are communications needed?
- Are there any data dependencies?
- Are there synchronization needs?
- Will load balancing be a concern?



https://hpc.llnl.gov/training/tutorials

1-D Wave Equation

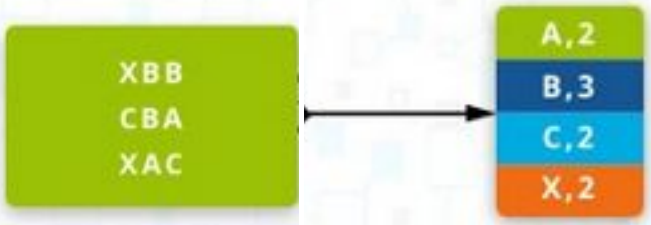


A(i,t+1) =

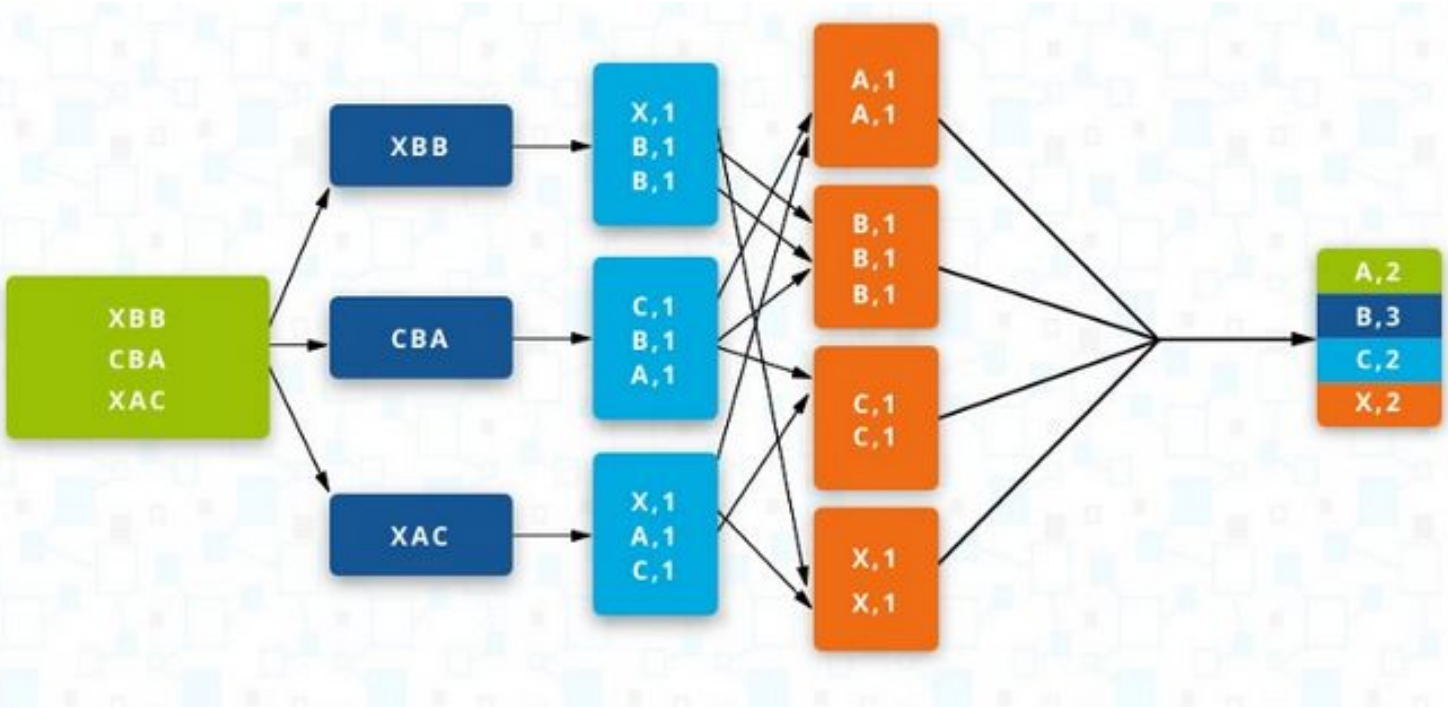
```
(2.0 * A(i,t)) - A(i,t-1) + (c * (A(i-1,t) - (2.0 * A(i,t)) + A(i+1,t)))
```

- Is this problem able to be parallelized?
- How would the problem be partitioned?
- Are communications needed?
- Are there any data dependencies?
- Are there synchronization needs?
- Will load balancing be a concern?

Ex. Count



- Is this problem able to be parallelized?
- How would the problem be partitioned?
- Are communications needed?
- Are there any data dependencies?
- Are there synchronization needs?
- Will load balancing be a concern?


XBB
CBA
XAC
XBB
CBA
XAC
X,1
B,1
B,1
C,1
B,1
A,1
X,1
A,1
C,1
A,1
A,1
B,1
B,1
B,1
C,1
C,1
X,1
X,1
A,2
B,3
C,2
X,2

Outline

- Introduction
- Parallel Computer Memory Architectures
- Parallel Programming Models
- Designing Parallel Programs
- Parallel Programming Paradigm
- Parallel Programming Examples
- Benchmarking
- Performance Index

Benchmarks

Why do you use benchmarks?

- **To assess differences:**
  - Different systems
  - Changes on a given system
- **To forecast the performance:**
  - The benchmarks should represent a broad class of representative programs, so they represent a wide range of common uses of a computer system.
  - Improving the performance of the benchmarks should also help to improve the corresponding programs.
- **Good benchmarks help to improve the development of the systems.**
- **Bad benchmarks damage the progress, they only help to sellers.**

Kind of Benchmarks

The main kinds of benchmarks are:

- **Synthetic benchmarks**: are artificial programs that are constructed to match the characteristics of large set of programs. Whetstone and Dhrystone are the most popular synthetic benchmarks. Whetstone performance is measured in “Whetstone per second” – the number of executions of one iteration of the whetstone benchmark
- **Kernel Benchmarks**: They are a fraction of the main code of real programs. Examples: Livermore, Linpack, …
- **Real Benchmarks**: They are real applications. For example SPEC.

Outline

- Introduction
- Parallel Computer Memory Architectures
- Parallel Programming Models
- Designing Parallel Programs
- Parallel Programming Paradigm
- Parallel Programming Examples
- Benchmarking
- Performance Index

Performance Index

- **Execution Rate:** Measures the machine output per unit of time.
  - *Instructions*: $R = \frac{instructions}{s} \times 10^{-6}$; measured in MIPS (millions of instructions per second)
  - *Operations*: $R = \frac{operations}{s} \times 10^{-6}$; measured in Mops (millions of operations per second)
  - *Floating Point Operations*: $R = \frac{op\_float\_point}{s} \times 10^{-6}$; measured in MFLOPS (millions floating operations per second)

Metrics of Performance

- **Microprocessors:**
  - **MIPS (Millions of Instructions per Second)** Used in the early computers. However, it depends of the set of instruction of each machine. Whenever the complexity of the instruction was bigger, lower MIPS will be obtained; but it does not mean that the performance of these machines was smaller.

$$MIPS = \frac{NI}{T_{CPU}} x10^{-6}$$

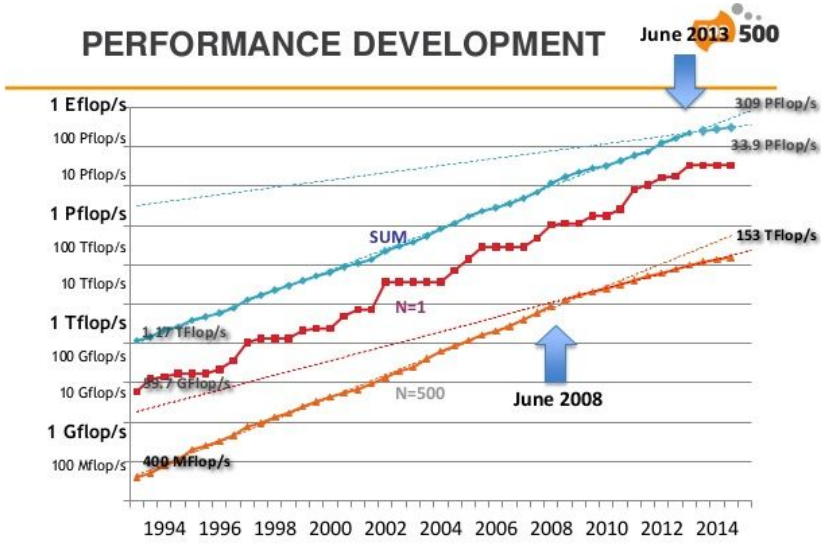
There are three problems with MIPS:

- MIPS specifies the instruction execution rate but not the capabilities of the instructions
- MIPS varies between programs on the same computer
- MIPS can vary inversely with performance

Metrics of Performance

- **Microprocessors:**
  - **MFLOPS (“Milion of FLoating-point Operations Per Second”)**

    This is the most used CPU metric It is independent of the set of instructions, but it depends totally of the FPU (Floating Point Unit). In addition, MFLOPS measure the same for an *ADD* operation than for a *DIV* operation, when the last one is more expe



Performance Index

- **Speedup**: Ratio between the serial and parallel execution time :

$$Sp = \frac{T_1}{T_P}$$

T_1 is the time taken to perform the computation on one processor,
T_P is the time taken to perform the same computation on p processors.
Normally: $1 <= S_p < p$, due to the parallel overhead.

- **Efficiency:** Ratio between the speedup factor and the number of processors. This is a measure of the cost-efficiency of computations.

$$\frac{1}{p} \leq E = \frac{S_p}{p} = \frac{T_1}{pT_p} \leq 1$$

Performance Index

- **Redundancy**: Ratio between the total number of operations (O_p) executed in performing some computation with p processors and the number of operations (O_1) required to execute the same computation with a uniprocessor. :

$$1 \leq R_p = \frac{O_p}{O_1} \leq p$$

- **Utilization**: Ratio between the actual number of operations O_p and the number of operations that could have been performed with p processors in T_p time units :

$$\frac{1}{p} \leq U = \frac{O_p}{pT_p} \leq 1$$