Hardware Parallelism

- This refers to the type of parallelism defined by the machine architecture and hardware multiplicity.
- Hardware parallelism is often a function of cost and performance tradeoffs.
- It displays the resource utilization patterns of simultaneously executable operations.
- It can also indicate the peak performance of the processor resource.
- One way to characterize the parallelism in a processor is by the number of instruction issues per machine cycle.
- If a processor issues k instructions pcr machine cycle, then it is called a **k-issue** processor.
- A conventional pipelined processor takes one machine cycle to issue a single instruction.
- These types of processors an: called one-issue machines, with a single instruction pipeline in the processor.
- In a modem processor, two or more instructions can be issued per machine cycle.

Software Parallelism

- This type of parallelism is revealed in the program profile or in the program flow graph.
- Software parallelism is it function of algorithm, programming style, and program design.
- The program flow graph displays the partems of simultaneously executable operations.

Program Decomposition

- General Ideas:
  - Identify the portions of code that can be done in parallel.
  - Mapping the code onto multiple processes.
  - Distributing the input, output, and intermediate data
  - Managing the access to shared resources.
  - Synchronizing the processes at various stages of the program.

Code Decomposition

- **Decomposition:** the operation of dividing the computation into smaller parts, some of which may be executed in parallel.
- **Task:** programmer-defined units of code resulting from decomposition.
- **Granularity:** the number / size of the tasks.
- **Fine-grained decomposition:** a large number of tasks
- **Coarse-grained decomposition:** small number of tasks.
- **Degree of concurrency:** the maximum number of tasks that can be executed in the same time.

Decomposition Techniques

- **Recursive decomposition:** used for traditional divide-and-conquer algorithms that are not easy to solve iteratively.
- **Data decomposition:** the data is partitioned and this induces a partitioning of the code in tasks.
- **Functional decomposition:** the functions to be performed on data are split into multiple tasks.
- **Exploratory decomposition:** decompose problems equivalent to a search of a space for solutions.
- **Speculative decomposition:** when a program may take one of many possible branches depending on results from computations preceding the choice.

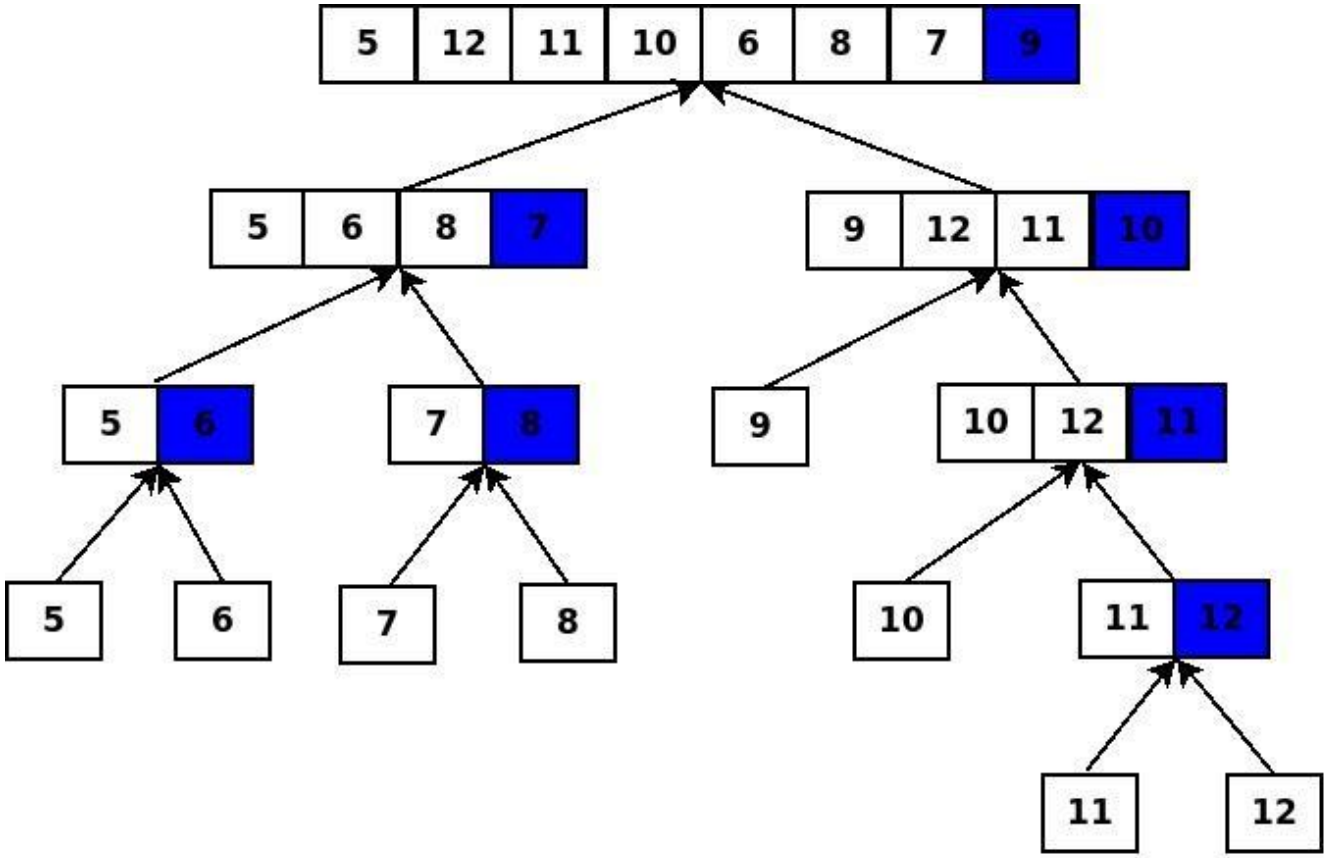
Characteristics of Tasks

- **Task generation:**
  - static - the tasks are known in advance (data decomposition)
  - dynamic - decided at runtime (recursive decomposition)
- **Task size:**
  - uniform (they require approximately the same amount of time) or
  - Non-uniform
  - known/not known.
- **Task Interaction**
  - Static: it happens at predetermined times and the set of tasks to interact with is known in advance.
  - Dynamic: the timing of the interaction or the set of tasks to interact with are unpredictable. Harder to implement.
  - Regular/irregular: it is regular if the interaction follows a pattern that can be exploited for efficiency.

Recursive Decomposition

- **Examples: QuickSort or MergeSort.**
- In both cases the operation of sorting an array is divided into two subproblems that can be solved recursively.
- Both problems are hard to implement iteratively.
- For the **Quicksort** the task generation is dynamic and the task size is non-uniform.
- For the **MergeSort** the task generation is static and the task size is uniform.

Example: Quick sort



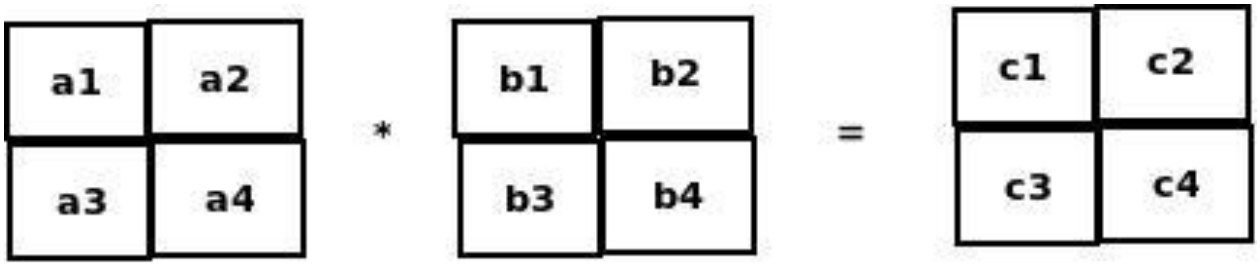
Example: The MergeSort

- Divides the array in 2, sorts the 2 parts recursively, then merges the arrays.
- The computations are organized in a binary tree.
- Each process receives an array to sort from the parent (except for the master).
- The process divides the array in 2 and sends the halves to the children.
- After the children are done computing, they send the sorted arrays back to the parent.
- The parent performed the merge and sends the array back up in the tree.

Example: even-odd transposition sort

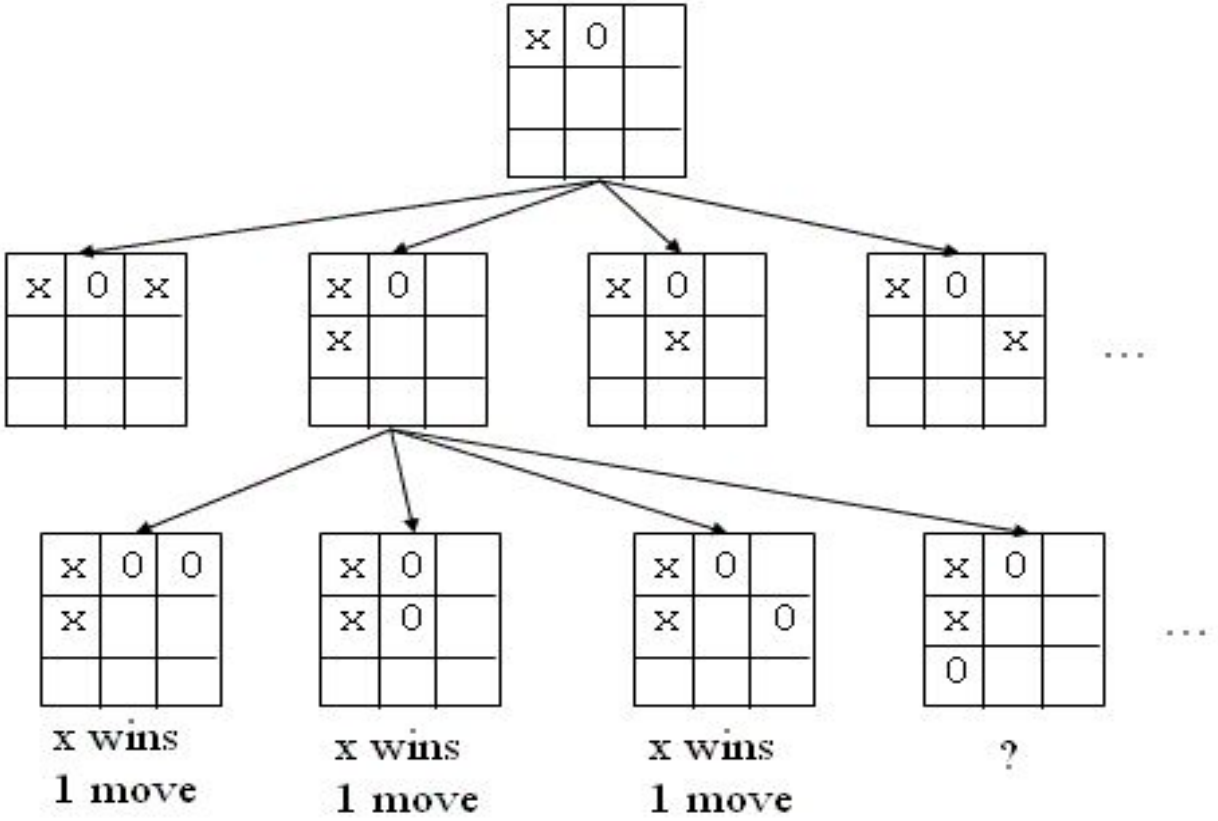
- It is based on the Bubble Sort technique, which compares every 2 consecutive numbers in the array and swap them if first is greater than the second to get an ascending order array.
- It consists of 2 phases – the odd phase and even phase:
- **Odd phase:** Every odd indexed element is compared with the next even indexed element (considering 1-based indexing).
- **Even phase:** Every even indexed element is compared with the next odd indexed element.

Data Decomposition



- c1=a1*b1+a2*b3
- c2=a1*b2+a2*b4
- c3=a3*b1+a4*b3
- c4=a3*b2+a4*b4

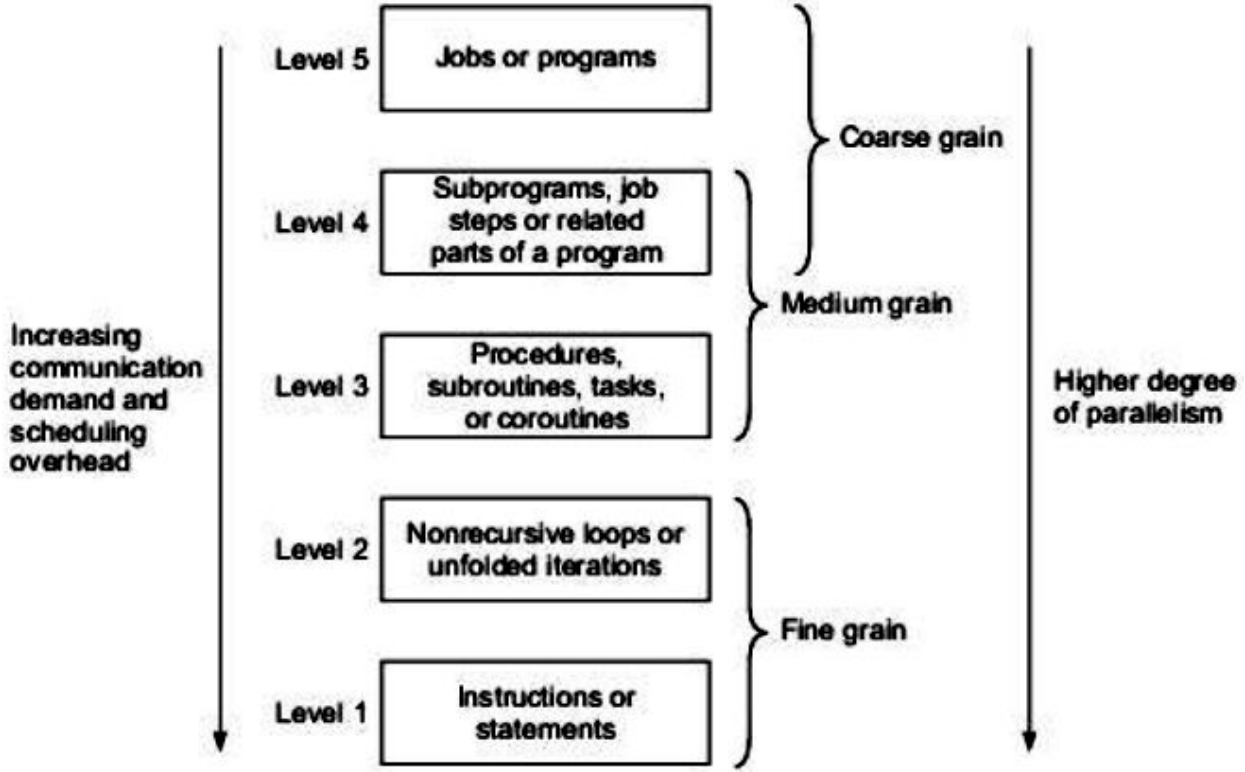
Exploratory Decomposition



Grain Sizes and Latency

- **Grain size** or **granularity** is a measure of the amount of computation involved in a software process.
  - The simplest measure is to count the number of instructions in a grain (program segment).
  - Grain size determines the basic program segment chosen for parallel processing.
  - Grain sizes are commonly described as fine, medium or course, depending on thc processing levels involved.
- **Latency** is a time measure of the communication overhead incurred between machine subsystems.
  - For example, the memory latency is the time required by a processor to access the memory.
  - The time required for two processes to synchronise with each other is called synchronisation latency.
  - Computational granularity and communication latency are closely related.

Levels of parallelism in program execution on modem computers



Grain Packing and Scheduling

- Fundamental questions in parallel programming:
  - How can we partition a program into parallel branches, program modules, microtasks, or grains to yield the shortest possible execution time? And
  - What is the optimal size of concurrent grains in a computation‘?
- Solution involves determination of both the number and the size of grains (or microtasks) in a parallel program, with a goal to produce a short schedule for fast execution of subdivided program modules with least computation and communication overheads.
- ***Answer to the Problem:*** The idea of **grain packing** is to apply fine grain first in order to achieve a higher degree of parallelism, then one combines (packs) multiple fine-grain nodes into a coarse grain node if it can eliminate unnecessary communications delays or reduce thc overall scheduling overhead.

Grain packing approach by Kruatraehue and Lewis (I938) for parallel programming applications.

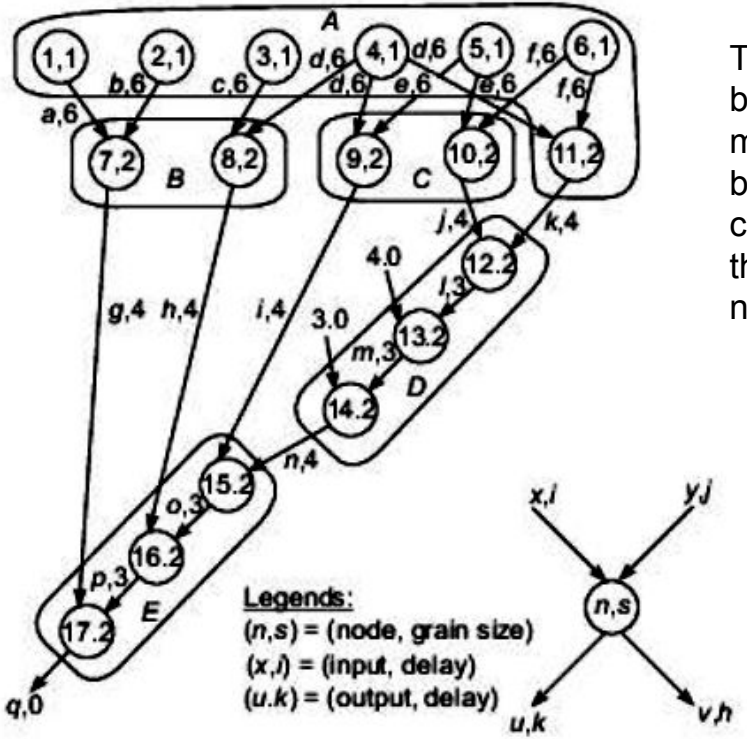
Example:

Var $a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q$

Begin

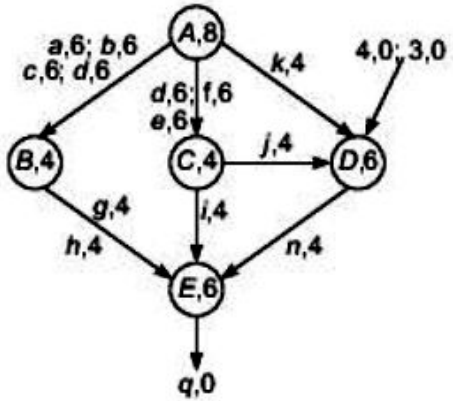
1. $a := 1$
2. $b := 2$
3. $c := 3$
4. $d := 4$
5. $e := 5$
6. $f := 6$
7. $g := a \times b$
8. $h := c \times d$
9. $i := d \times e$
10. $j := e \times f$
11. $k := d \times f$
12. $l := j \times k$
13. $m := 4 \times l$
14. $n := 3 \times m$
15. $o := n \times i$
16. $p := o \times h$
17. $q := p \times q$

End



(a) Fine-grain program graph before packing

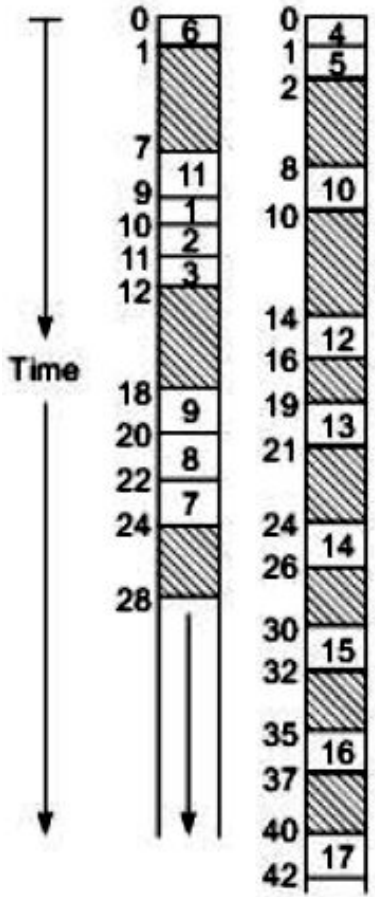
The **grain size** is measured by the number of basic machine cycles (including both processor and memory cycles) needed to execute all the operations within the node.



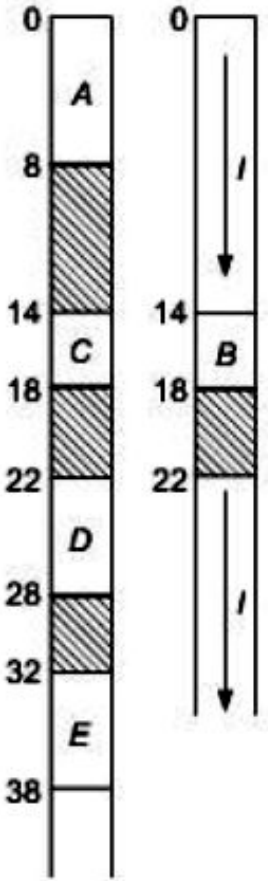
(b) Coarse-grain program graph after packing

Nodes **I, 2, 3, 4, 5**, and **6** are memory reference (data fetch} operations, each takes **1** cycle to address and **6** cycles to fetch from memory. All remaining nodes (**7 to 17**) are CPU operations, each requiring **2** cycles to complete.

Scheduling of the fine-grain and coarse-grain programs



Fine-Grain



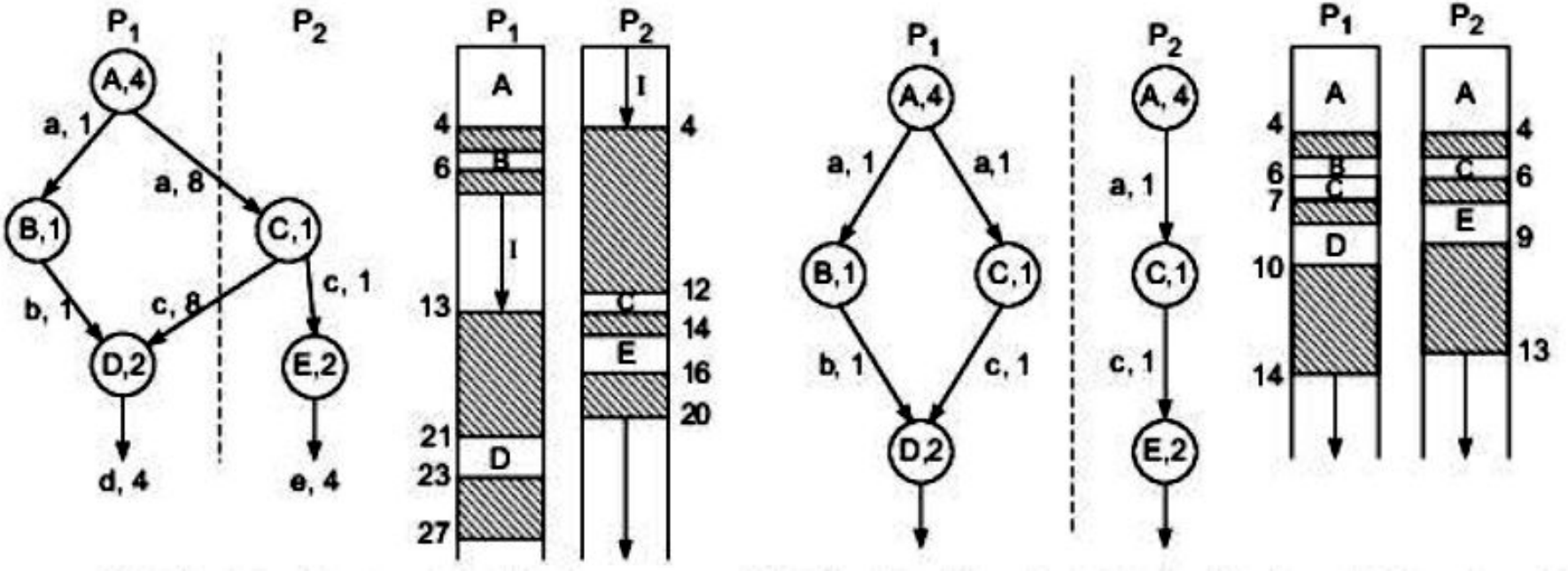
Coarse-Grain

- Here, we use 2 multiprocessor schedules.
- The fine-grain schedule is longer (**42** time units) because more communication delays were included as shown by the shaded area.
- The coarse-grain schedule is shorter (**38** time units) because communication delays among nodes 12, 13, and 14 within the same node D (and also the delays among 15, 115, and 17 within the node E) are eliminated after grain packing.

Static Multiprocessor Scheduling

- Grain packing may not always produce a shorter schedule.
- In general, dynamic multiprocessor scheduling is an NP-hard problem.
- Very often heuristics are used to yield suboptimal solutions.
- **Node Duplication** (duplicate some of the nodes in more than one processor) for multiprocessor scheduling using static schemes provides feasible solution.

Static Multiprocessor Scheduling with Node Duplication



Schedule without node duplication **Schedule with node duplication**

The new schedule is almost 50% shorter. The reduction in schedule time is caused by elimination of the (a, 8) and (c, 8) delays between the two processors.

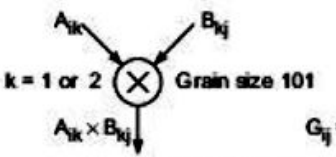
Combining Grain packing and node duplication

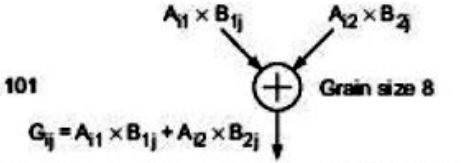
- Four major steps are involved in the grain determination and the process of scheduling optimization:
  - Step 1: Construct a fine-grain program graph.
  - Step 2: Schedule the fine-grain computation.
  - Step 3: Perform grain packing to produce the coarse grains.
  - Step 4: Generate a parallel schedule based on the packed graph.

Program decomposition for static multiprocessor scheduling

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$$
$$C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$$
$$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}$$
$$C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$$
$$\text{Sum} = C_{11} + C_{12} + C_{21} + C_{22}$$

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

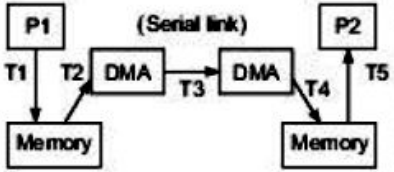




			CPU CYCLE
Move W	Axx,	D1	15
Move W	Bxx,	D2	15
MPTY D1,	D2		71
MOVE L	D2,	PAR	20

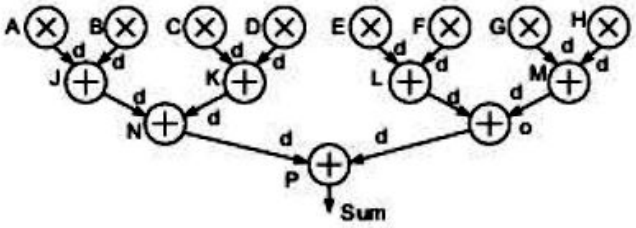
			CPU CYCLE
Move L	PAR1,	D1	20
Move L	PAR2,	D2	20
ADD L	D1,	D2	8
MOVE L	D2,	PSUM	20

(a) Grain size calculation in M68000 assembly code at 20-MHz cycle

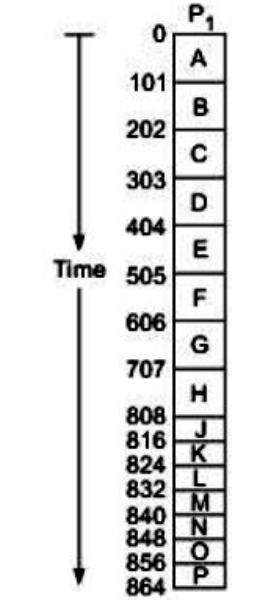


d = T1 + T2 + T3 + T4 + T5 + T6
= 20 + 20 + 32 + 20 + 20 + 100
= 212 cycles
T3 = 32-bit transmission time at 20 Mbps normalized to M68000 cycle at 20 MHz.
T6 = delay due to software protocols (assume 5 Move instructions, 100)

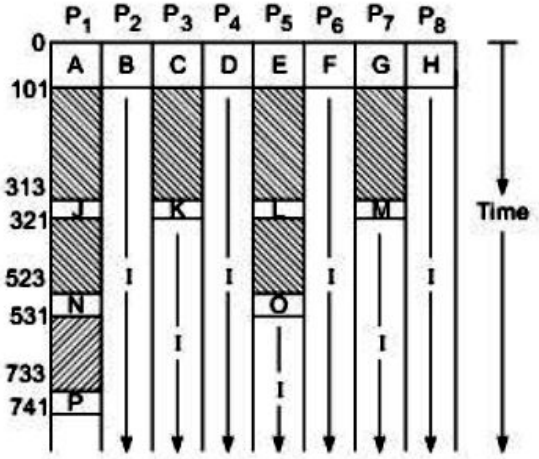
(b) Calculation of communication delay *d*



(c) Fine-grain program graph



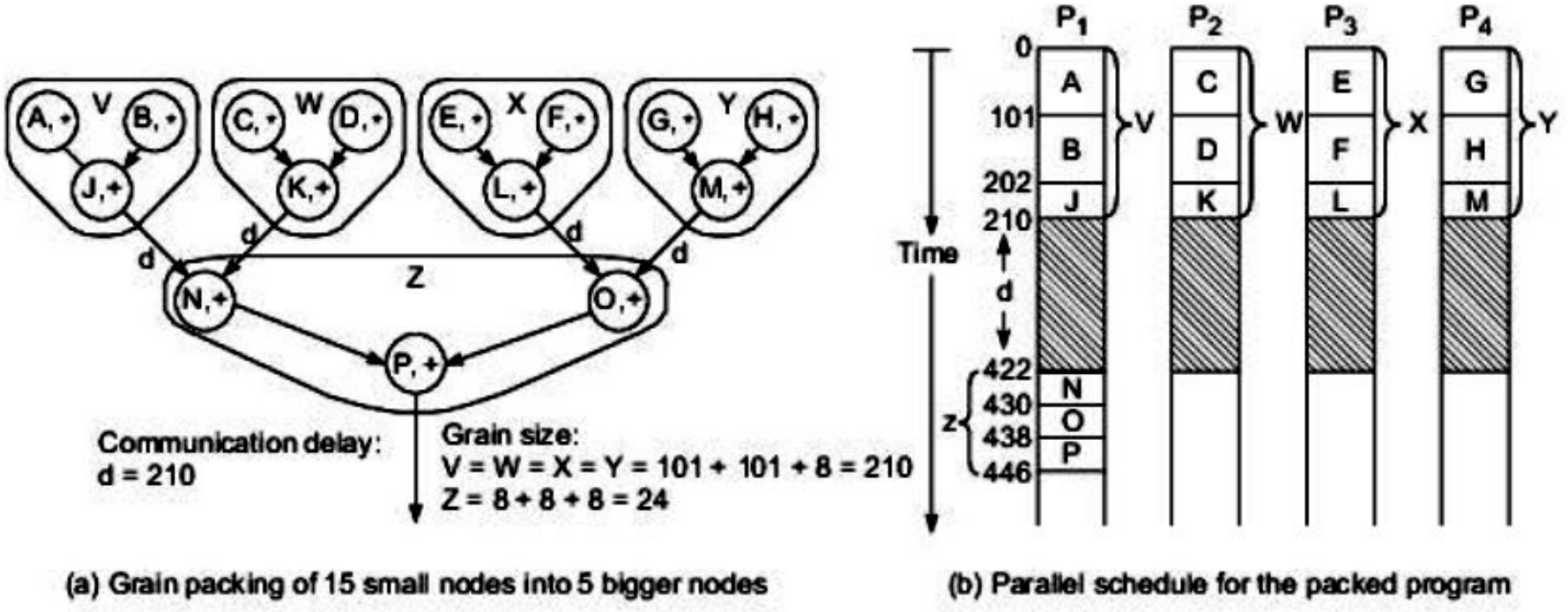
(a) A sequential schedule



(b) A parallel schedule

Speed-up factor = 864/741 = 1.16

Program decomposition for static multiprocessor scheduling (Contd…)



Since the maximum degree of parallelism is now reduced to **4** in the program graph, we use only four processors to execute this coarse-grain program.

Improved Speed-up = 864/446 = 1.94

Shared Memory Programming (pthreads)

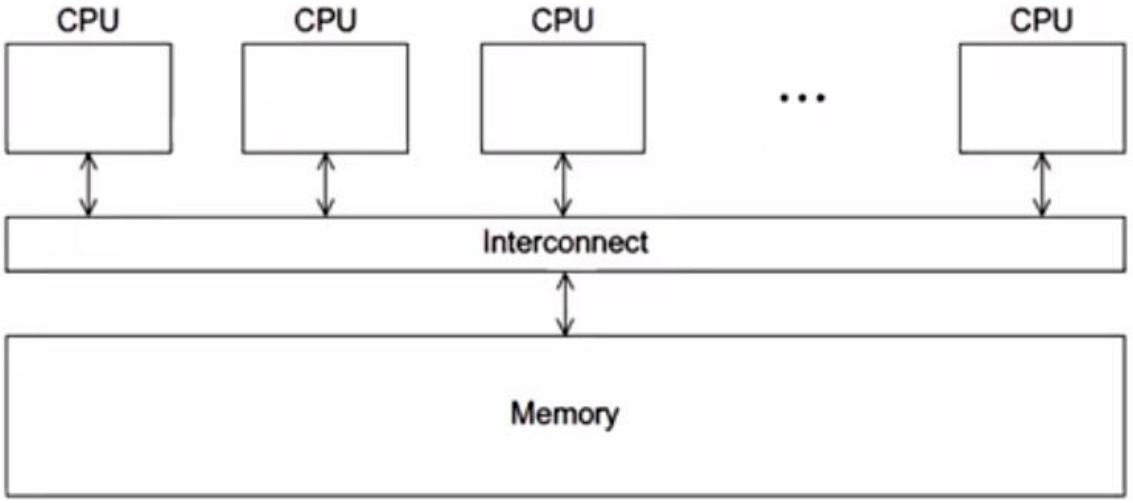
- Shared memory helps programs communicate faster.
- Programs may use one or more processors and as a result, a process may have several threads.
- Threads are referred to as lightweight processes.
- They are referred to as shared lightweight processes because they are formed by dividing a single process into many processes called threads.
- Threading is a process that achieves parallelism. Parallelism is the simultaneous execution of many processes.
- Browser tabs are threads. The same threading technology is used in Microsoft Word.

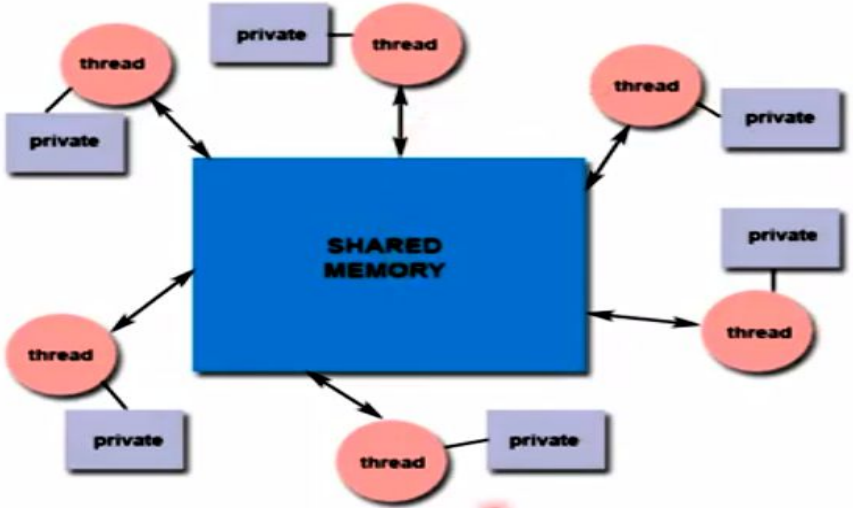
POSIX (Portable Operating System Interface for Unix) **Thread**

- Also known as pthreads.
- It is an IEEE standard for Unix-like OS.
- It is a library which can be linked with C program.
- It specifies an API for multi-threaded programming.
- It forms the basic building block for many parallel libraries like: OpenMP, Intel thread building block.
- They are needed to implement algorithms that have irregular decomposition and communication.

Shared Memory Model

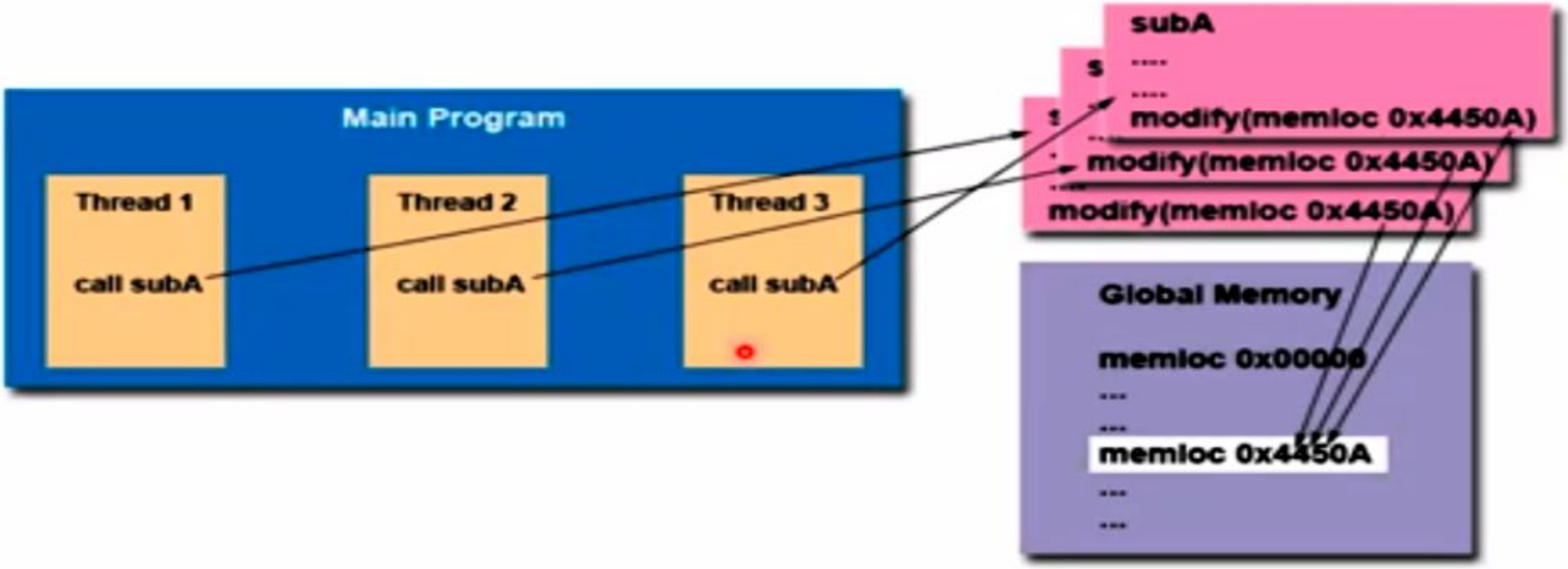
- A multi-core CPU processes one thread per core.
- The threads of a process are separate, but they share a memory space.
- A process contains atleast one thread.
- Programmers are responsible for synchronizing access (protecting) globally shared data.





Thread Safeness

- It refers to an application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating race condition while accessing shared global memory.



Pthread limitations

- Although the Pthreads API is an ANSI/IEEE standard, implementations can, and usually do, vary in ways not specified by the standard.
- Because of this, a program that runs fine on one platform, may fail or produce wrong results on another platform.
- For example, the maximum number of threads permitted, and the default thread stack size are two important limits to consider when designing your program.

The Pthreads API

- The Pthreads API can be informally divided into four major groups.
  - **Thread Management:** handles creating, terminating, and joining threads, etc.
  - **Mutexes:** deals with synchronization, called a “mutex”, an abbreviation of mutual exclusion. Mutex functions provide for creating, destroying, locking, and unlocking mutexes.
  - **Condition variables:** address communication among the threads that share a mutex. This specifies functions to create, destroy, wait, and signal based upon specified variable values.
  - **Synchronization:** manage read/write locks and barriers.

Posix thread system calls

- **int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);**
- ***thread:** is a pointer to the thread id number (the type pthread_t is an int). The calling program may need this value later for thread synchronization.
- **pthread_attr_t** allows the calling program to set some of the attibutes of the thread, such as the size of the stack.
- **start_routine:** the C routine that the thread will execute once it is created. (replacing start_routine with the name of the function.)
- **arg:** a single argument that may be passed to start_routine.
- As is usually the case, the pthread_create call will return a zero if it successfully created the new thread and a negative value if it failed, and if it failed, the external global variable errno will be set to a value which indicates why it failed.

Posix thread system calls

- To terminate a single thread without killing the entire process, use the system call:
- **void pthread_exit(void *value_ptr);**
- The argument is a pointer to a return value, but it can be set to NULL.
- The calling thread can wait for a particular thread to terminate with the call:
- **int pthread_join(pthread_t thread, void **value_ptr);**
- The first argument is the thread id that the process is waiting for, and the second is a pointer to the argument passed back by the thread. This can be set to NULL. This function will block until the thread terminates.

Posix Thread Synchronization

- **int pthread_mutex_lock(pthread_mutex_t *mutex);**
- This function checks to see if the mutex is locked. If it is not, it locks the mutex and returns, thus allowing the calling thread to continue. If the mutex is locked, the thread blocks until the mutex becomes unlocked. At this time the mutex is set to the locked state again but the function returns, allowing the thread to continue. This should be called before a thread enters its critical section.
- **int pthread_mutex_trylock(pthread_mutex_t *mutex);**
- This is similar to the above function but it always returns immediately. The return value is zero if the mutex had previously been unlocked and the thread has successfully locked the mutex. If the mutex is locked, the function returns a non-zero value.
- **int pthread_mutex_unlock(pthread_mutex_t *mutex);**
- This unlocks a locked mutex. This should be called by a thread after it leaves its critical section.

Single Program, Multiple Data (SPMD) Model

- It is a technique employed to achieve parallelism.
- It is a subcategory of MIMD.
- Tasks are split up and run simultaneously on multiple processors with different input in order to obtain results faster.
- SPMD is the most common style of parallel programming.
- Difference between SIMD and SPMD:
  - SIMD is vectorization at the instruction level - each CPU instruction processes multiple data elements. SPMD is a much higher level abstraction where processes or programs are split across multiple processors and operate on different subsets of the data.
  - With SPMD, tasks can be executed on general purpose CPUs; SIMD requires vector processors to manipulate data streams.

Single Program, Multiple Data (SPMD) Model

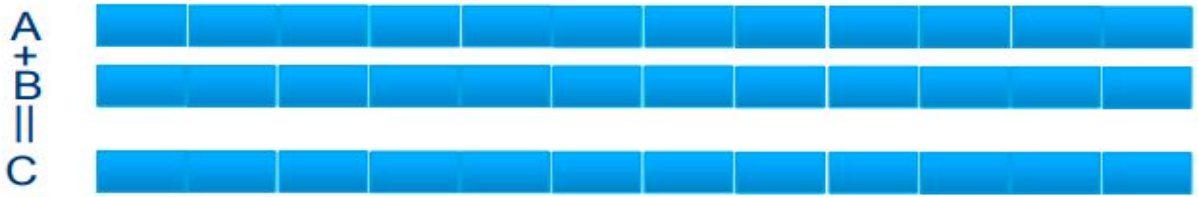
- SPMD usually refers to **message passing** programming on distributed memory computer architectures.
- A distributed memory computer consists of a collection of independent computers, called nodes.
- Each node starts its own program and communicates with other nodes by sending and receiving messages, calling send/receive routines for that purpose.
- Barrier synchronization may also be implemented by messages.

SPMD Example

- Consider the following vector addition example

```
for( i = 0:11 ) {
   C[ i ] = A[ i ] + B[ i ]
}
```

Serial program:
one program completes the entire task



- Combining SPMD with loop strip mining allows multiple copies of the same program execute on different data in parallel

```
for( i = 0:3 ) {
   C[ i ] = A[ i ] + B[ i ]
}
```

```
for( i = 4:7 ) {
   C[ i ] = A[ i ] + B[ i ]
}
```

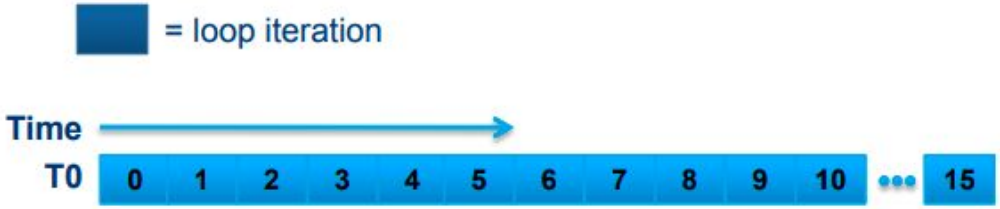
```
for( i = 8:11 ) {
   C[ i ] = A[ i ] + B[ i ]
}
```

SPMD program:
multiple copies of the same program run on different chunks of the data

SPMD Example

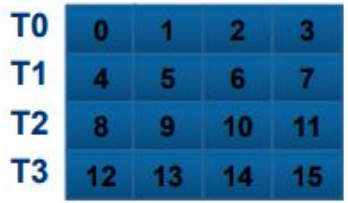
Single-threaded (CPU)

```
// there are N elements
for(i = 0; i < N; i++)
  C[i] = A[i] + B[i]
```



Multi-threaded (CPU)

```
// tid is the thread id
// P is the number of cores
for(i = 0; i < tid*N/P; i++)
  C[i] = A[i] + B[i]
```



Massively Multi-threaded (GPU)

```
// tid is the thread id
C[tid] = A[tid] + B[tid]
```

Message Passing Interface (MPI)

- It is an Application Program Interface that defines a model of parallel computing where each parallel process has its own local memory, and data must be explicitly shared by passing messages between processes.
- Using MPI allows programs to scale beyond the processors and shared memory of a single compute server, to the distributed memory and processors of multiple compute servers combined together.

MPI Communication Functions

- **Blocking Communication Functions**
- Blocking communication are routines where the completion of the call is dependent on certain “events”.
- For sends, the data must be successfully sent or safely copied to system buffer space and for receives, the data must be safely stored in the receive buffer.
- The functions are as follows:
  - **MPI_Send**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  - **MPI_Recv**(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

MPI Communication Functions

- **Non-blocking Communication Functions:**
- A communication routine is non-blocking if the call returns without waiting for the communications to complete.
- It is the programmer's responsibility to insure that the buffer is free for reuse.
- These are primarily used to increase performance by overlapping computation with communication. It is recommended to first get your program working using blocking communication before attempting to use non-blocking functions. The functions are as follows:
  - **MPI_Isend**(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
  - **MPI_Irecv**(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

MPI Communication Functions

- **Synchronization –** processes wait until all members of the group have reached the synchronization point. The function used to do this is:
  - **MPI_Barrier (comm)**
- This causes each process, when reaching the MPI_Barrier call, to block until all tasks in the group reach the same MPI_Barrier call.

OpenMP

- It is a library for parallel programming in the SMP (symmetric multi-processors, or shared-memory processors) model.
- When programming with OpenMP, all threads share memory and data. OpenMP supports C, C++ and Fortran.
- The OpenMP functions are included in a header file called omp.h
- An OpenMP program has sections that are sequential and sections that are parallel. In general an OpenMP program starts with a sequential section in which it sets up the environment, initializes the variables, and so on.
- When run, an OpenMP program will use one thread (in the sequential sections), and several threads (in the parallel sections).
- There is one thread that runs from the beginning to the end, and it's called the **master thread**. The parallel sections of the program will cause additional threads to fork. These are called the **slave threads**.

OpenMP

- A section of code that is to be executed in parallel is marked by a special directive (omp pragma). When the execution reaches a parallel section (marked by omp pragma), this directive will cause slave threads to form.
- Each thread executes the parallel section of the code independently. When a thread finishes, it joins the master.
- When all threads finish, the master continues with code following the parallel section.
- Each thread has an ID attached to it that can be obtained using a runtime library function (called omp_get_thread_num()). The ID of the master thread is 0.

Why OpenMP?

- More efficient, and lower-level parallel code is possible, however OpenMP hides the low-level details and allows the programmer to describe the parallel code with high-level constructs, which is as simple as it can get.
- OpenMP has directives that allow the programmer to:
  - specify the parallel region
  - specify whether the variables in the parallel section are private or shared
  - specify how/if the threads are synchronized
  - specify how to parallelize loops
  - specify how the works is divided between threads (scheduling)

Synchronization with OpenMP

- **critical:** the enclosed code block will be executed by only one thread at a time, and not simultaneously executed by multiple threads. It is often used to protect shared data from race conditions.
- **atomic:** the memory update (write, or read-modify-write) in the next instruction will be performed atomically. It does not make the entire statement atomic; only the memory update is atomic. A compiler might use special hardware instructions for better performance than when using critical.
- **ordered:** the structured block is executed in the order in which iterations would be executed in a sequential loop
- **barrier:** each thread waits until all of the other threads of a team have reached this point. A work-sharing construct has an implicit barrier synchronization at the end.
- **nowait:** specifies that threads completing assigned work can proceed without waiting for all threads in the team to finish. In the absence of this clause, threads encounter a barrier synchronization at the end of the work sharing construct.