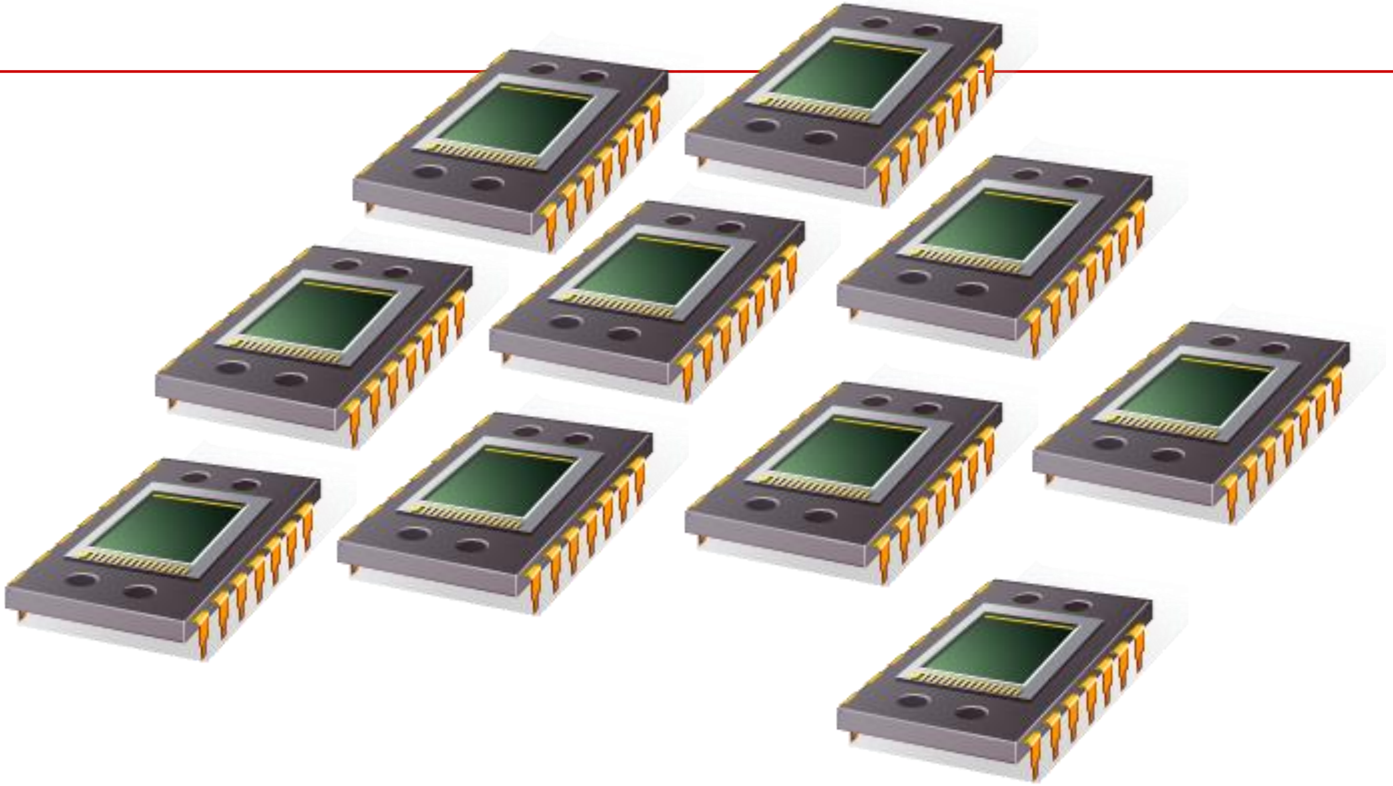


Parallel Software and Performance

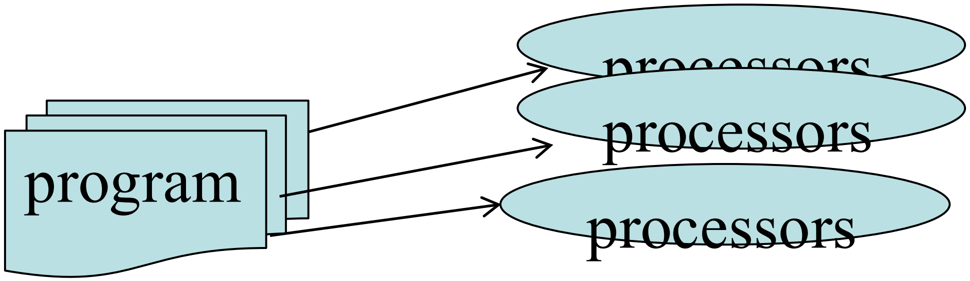
UCSB CS140, T. Yang

Outline

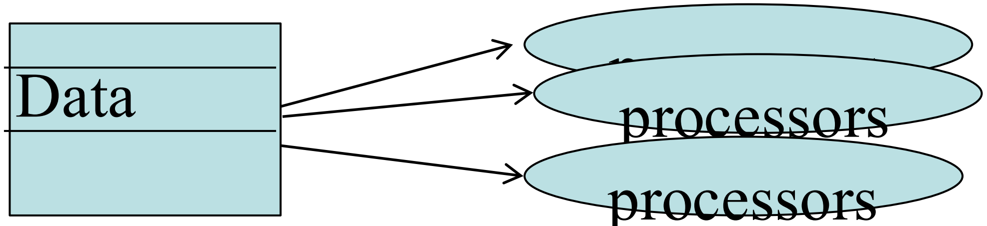
- **How to write parallel programs**
  - An abstract view with task graph model
  - SPMD (Single-program multiple data) coding style
- **Performance evaluation**
  - Parallel time, speedup, efficiency
  - What limits parallel performance
  - Parallel/serial performance assessment with FLOPS
- **Parallel program design strategies**

How do we write parallel programs?

- **Task parallelism by computation partitioning & mapping**
  - Divide code (computation) into a set of tasks
  - Map tasks to parallel processing units (processor cores, machines)

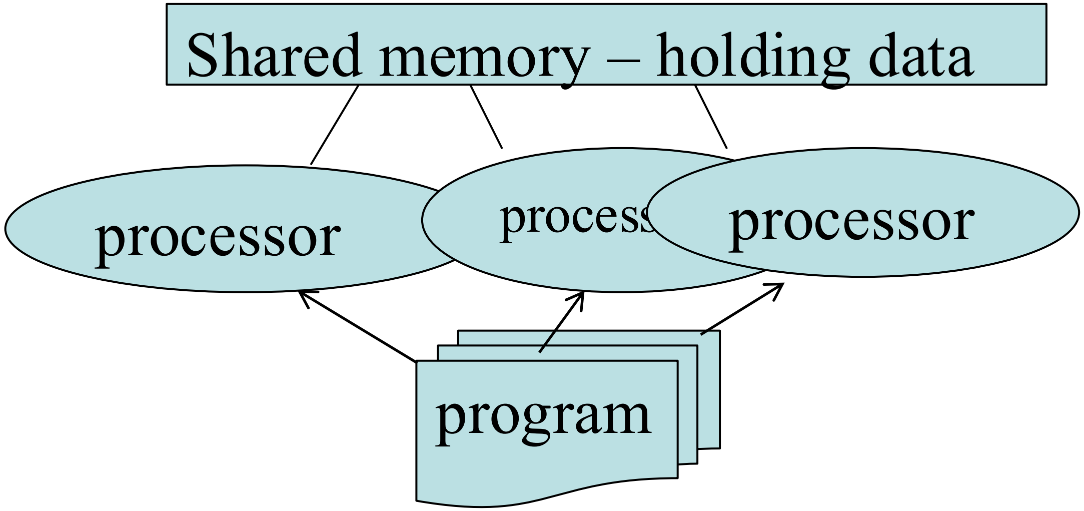


- **Data parallelism with data partitioning/mapping**
  - Divide data into a set of items. Computation for data items is partitioned accordingly
  - For a distributed architecture, map data items to separate machines



Shared vs. distributed memory programming

- Shared memory programming is easier
  - Task partitioning and mapping are required
  - Explicit data partitioning/mapping is not required because of shared memory

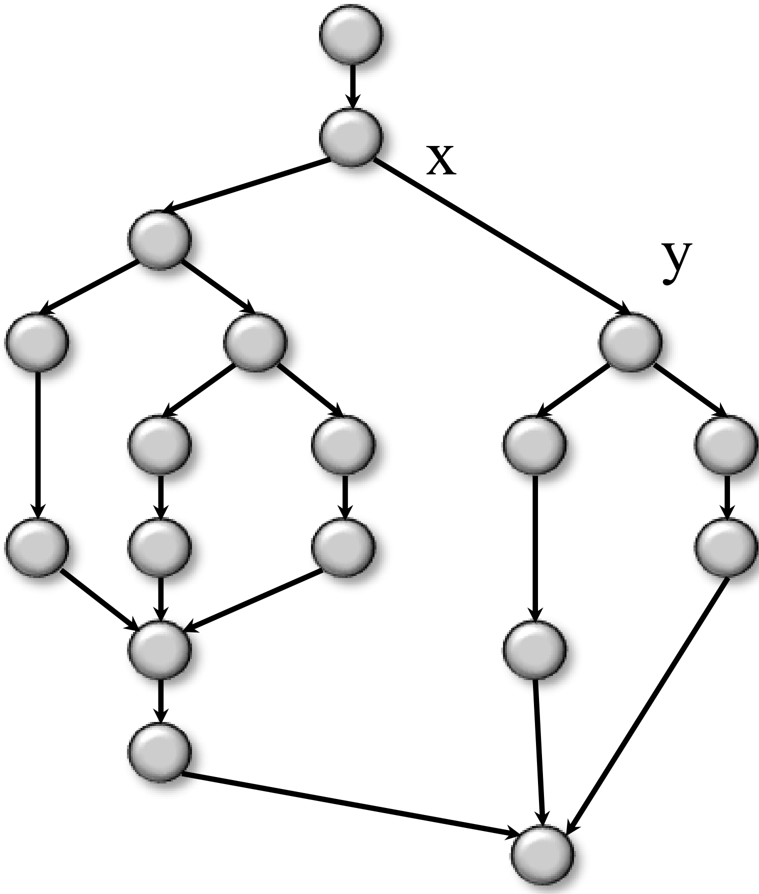


- For a distributed architecture, map data items to separate machines
  - More complex to manage distributed data naming and explicit communication among tasks that run on separate machines

Model Parallelism using a Directed Acyclic Task Graph

- A task is a basic computation unit that runs a program fragment,
- **A program consists of a set of tasks.**
    - Program partitioning produces a set of tasks. Tasks should not be too small to avoid excessive overhead

Each vertex is a task

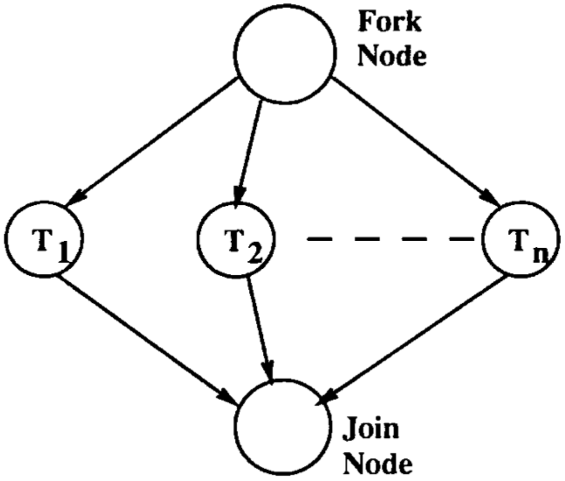


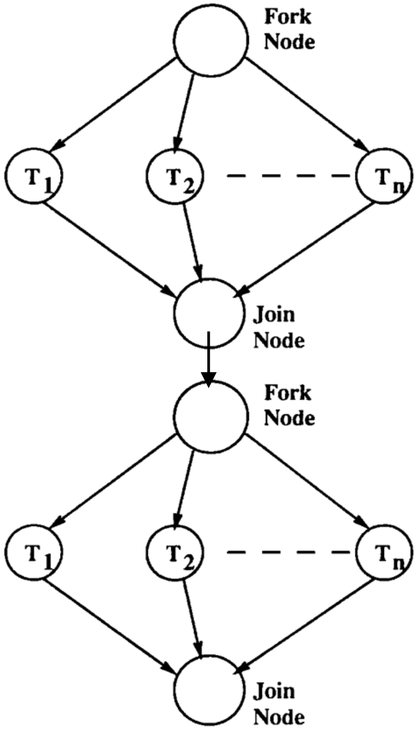
Each edge x->y represents task dependence: Task x has to be executed before y. Task x produces data used by y

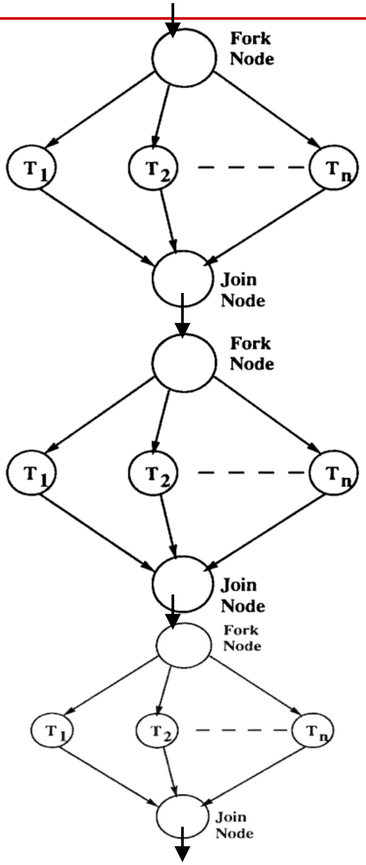
- Task y cannot start execution until the data produced by x is available in local memory

Popular Graph Pattern of Parallelism

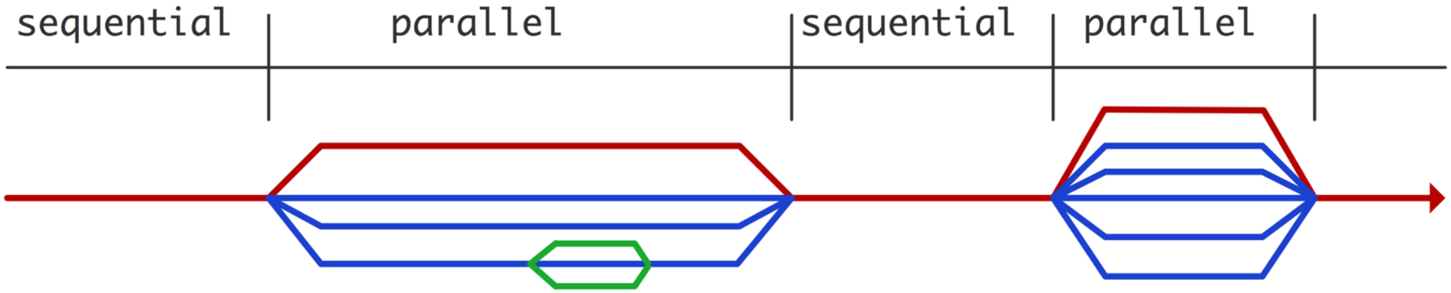
Fork-join parallelism







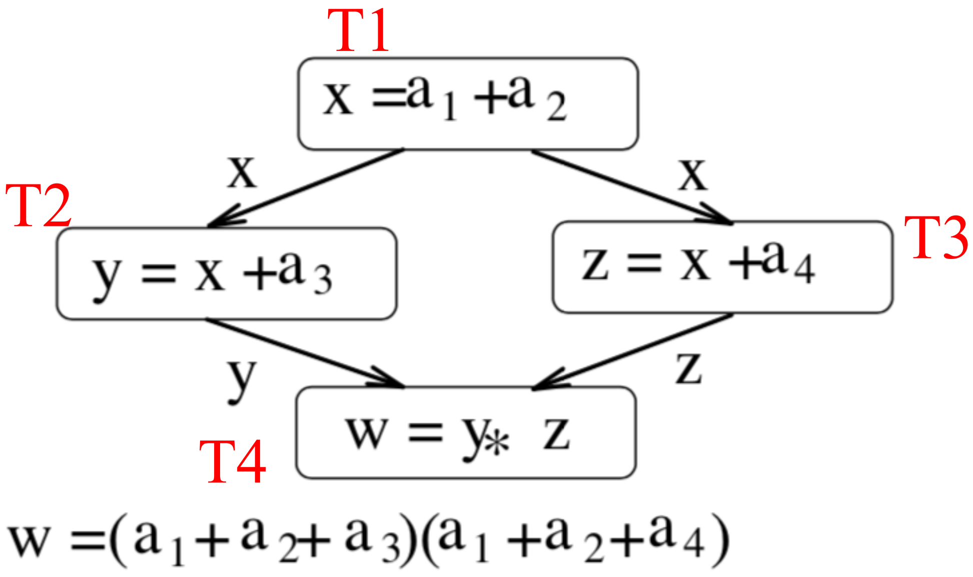
Parallel code structure:



Task Scheduling and Impact of Task Dependence Edges

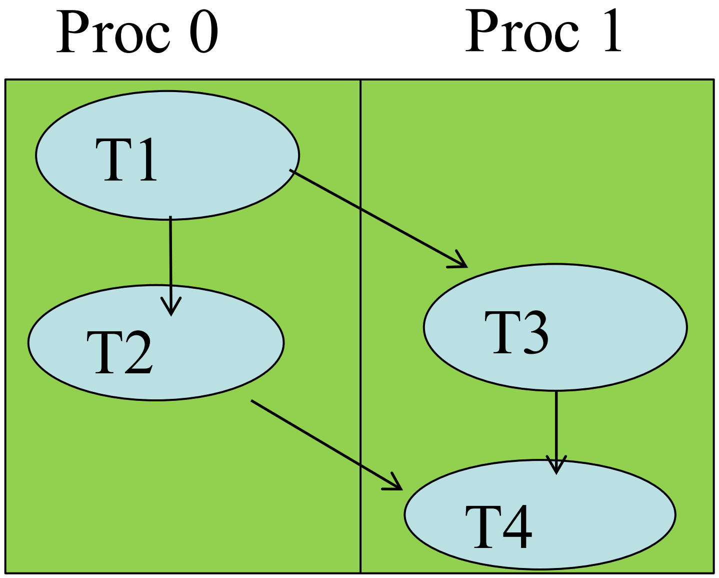
- **Scheduling** maps tasks to processors and also defines an order of execution at each processor
- **Task execution (a simplified model)**
  - A task waits to receive all data in parallel before it starts its execution.
  - As soon as the task completes its execution it sends the output data to all successors in parallel or makes data available through shared memory
- **Dependence constraint for edge x→y**
  - Let ST(x) be the starting execution time of task x and cost(x) is the execution time of x
  - ST(x)+ cost(x) ≥ ST(y)

An example of Task Scheduling



Task graph

Schedule

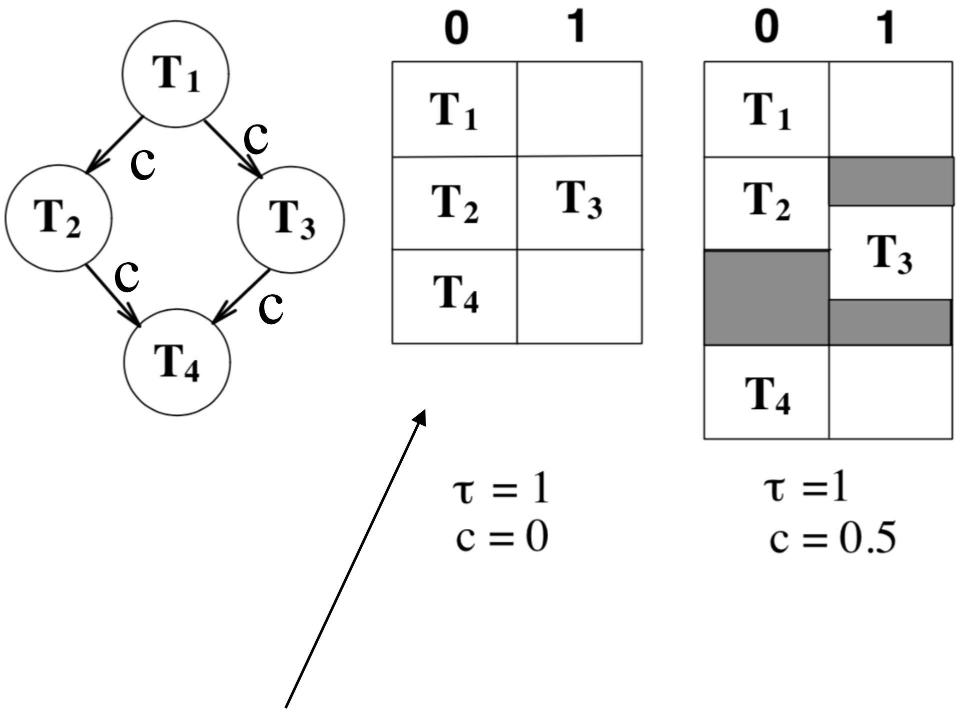


- Dependence edges are enforced by synchronization or message communication
- Scheduling can be done statically (fixed before running)

Or it can be derived dynamically during run-time

Gannt chart to represent a schedule

- We can use a Gannt chart to represent a schedule
- Assume there is cost c delay of sending a message between two processors. But overhead charged processors is 0. Message latency is 0 if two tasks communicated are in the same processor
- Computation cost is ***τ*** for each task



Left schedule can be expressed as

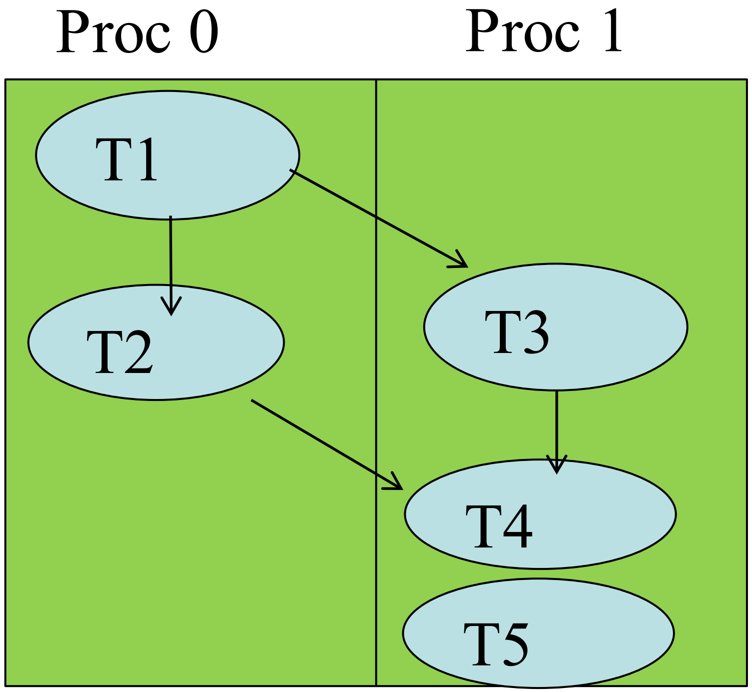
	T_1	T_2	T_3	T_4
Proc Assign.	0	0	1	0
Start time	0	1	1	2

What is a processor here?

- **Processor here is a logical processing unit**
  - Textbook calls it a core for shared memory programming
  - It can execute one or a number of tasks.
  - In general, we treat it as a logical execution unit. At runtime p logical units are executed in parallel by a fixed number of physical machines/cores.
- **In MPI programming or Linux programming**
  - It is a process
  - There is no data shared among processes
  - Process communication is done by message sending/receiving
- **In OpenMP/Pthread programming**
  - It is a thread. Memory is shared among threads

Estimate Parallel Time via Scheduling

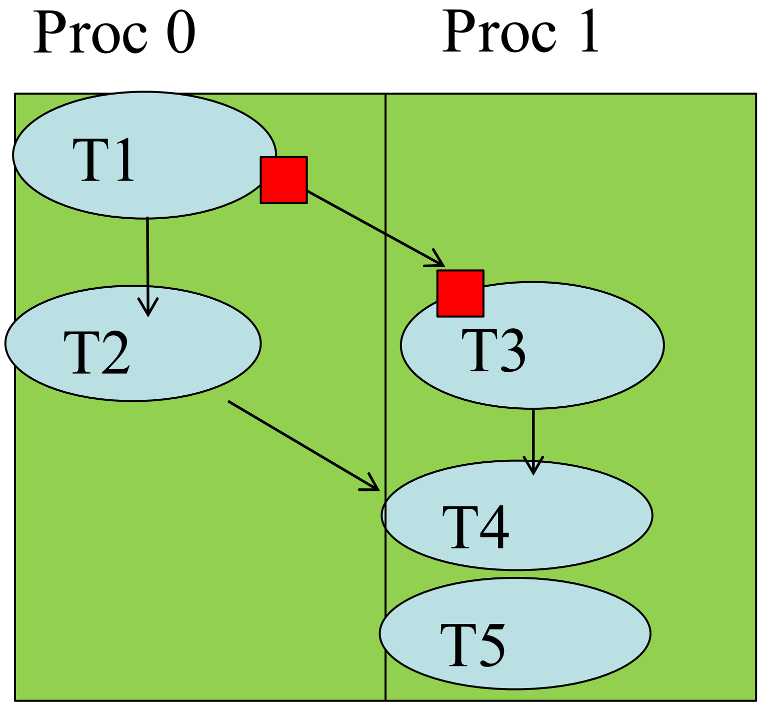
- Processor assignment of tasks determines where a task is executed.
- The execution order of tasks with each processor is further determined by a schedule
- Parallel time = Completion time of last task among all processors
- Parallel time = max(completion time of last task at each processor).



Schedule of a task graph

Overhead Paid in Shared-Memory and Distributed-Memory Parallel Programming

- **Shared memory architecture**
  - Ensuring T3 is executed after T1 requires a synchronization whose cost cannot be fully overlapped with computation
- **Distributed memory architecture**
  - Sending a message from T1 to T3 requires Proc 0 to pay some start cost, even routing this message may be overlapped with other computation of Proc 0



Schedule of a task graph

Style of Parallel Programming

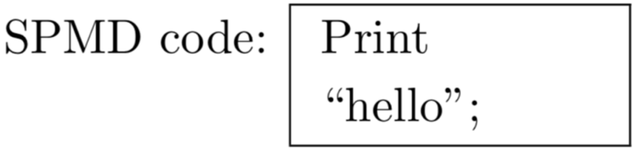
- **Express implicit parallelism with parallel operations in code**
  - Often assume shared data which may be implicitly distributed by the system
  - System assigns computation to multiple processing units automatically
  - Example: SIMD programming, OpenMP, MapReduce, Spark
- **Explicitly write the role of each processing unit**
  - Data may be shared or distributed
  - Code defines the behavior of each processing unit
    - Access data if shared
    - Allocate data locally and communicate if distributed
    - Compute
  - Example: MPI, Pthreads, CUDA

Explicitly Define Role of Processing Units: Coding Style

- **SPMD – single program multiple data**
  - Write one program, works for different data streams
  - Computation is distributed among processors. Code is executed based on a predetermined schedule.
  - Each processor executes the same program but operates on different data based on processor identification.
- **Master/slaves: One control process is called the master**
  - There are a number of slaves working for this master.
  - These slaves can be coded using an SPMD style.
- **MPMD – multiple programs for multiple data**
  - Write different programs for different processors handling different data streams
  - Flexible, but painful and unmanageable.

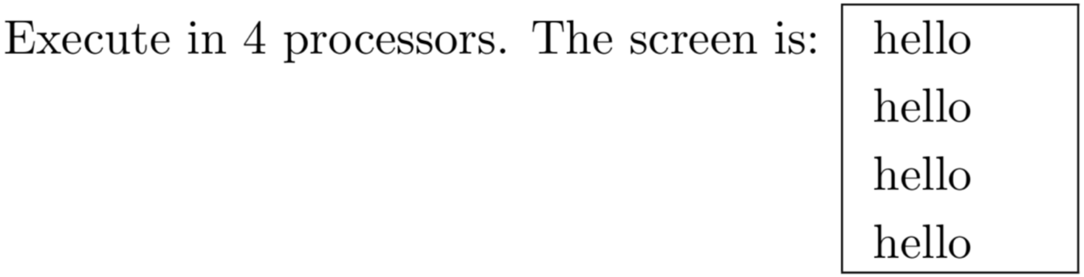
Generic Code Structure of SPMD

- **Processors are numbered as 0, 1, 2, …**
  - Each processor node executes the same program with a unique processor ID.
    - Differentiate the role of programs by their IDs
  - Assume two library functions (pseudo code)
    - **mynode()** – returns processor ID of the program executed on one processor.
    - **noproc()** - returns # of processors used
- **Example**



SPMD code:

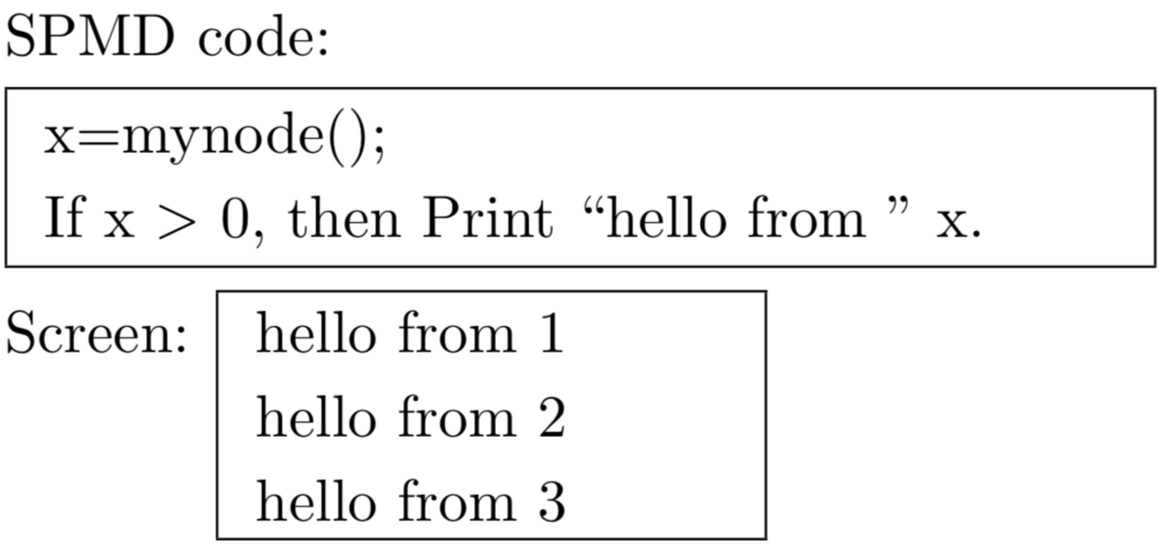
```
Print
“hello”;
```



Execute in 4 processors. The screen is:

```
hello
hello
hello
hello
```

Examples of SPMD code



SPMD code:

```
x=mynode();
If x > 0, then Print "hello from " x.
```

Screen:

```
hello from 1
hello from 2
hello from 3
```

- **Another sequential code example:**
  - For i = 0 to n-1

    x[i]= 3*i;

Task graph

a[0]=3*0
a[1]=3*1
…
a[n-1]=3*(n-1)

Example: Task graph, schedule, and SPMD code

Task graph

a[0]=3*0   a[1]=3*1   …   a[n-1]=3*(n-1)

Schedule

Proc 0	Proc 1		Proc n-1
a[0]=3*0	a[1]=3*1	…	a[n-1]=3*(n-1)

SPMD code:

```
me=mynode();
a[me]=3*me;
```

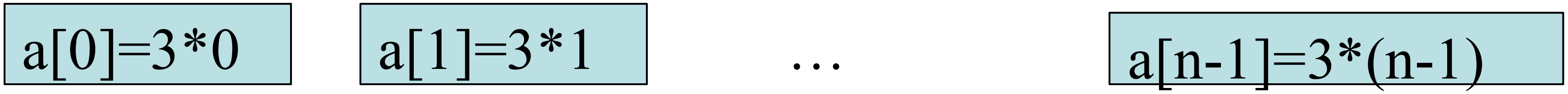
Runtime parallel execution

Proc 0	Proc 1		Proc n-1

me=mynode();
a[me]=3*me;
	me=mynode();
	a[me]=3*me;
		…	me=mynode();
			a[me]=3*me;

Example: Another schedule with p processors and n=k*p

Task graph

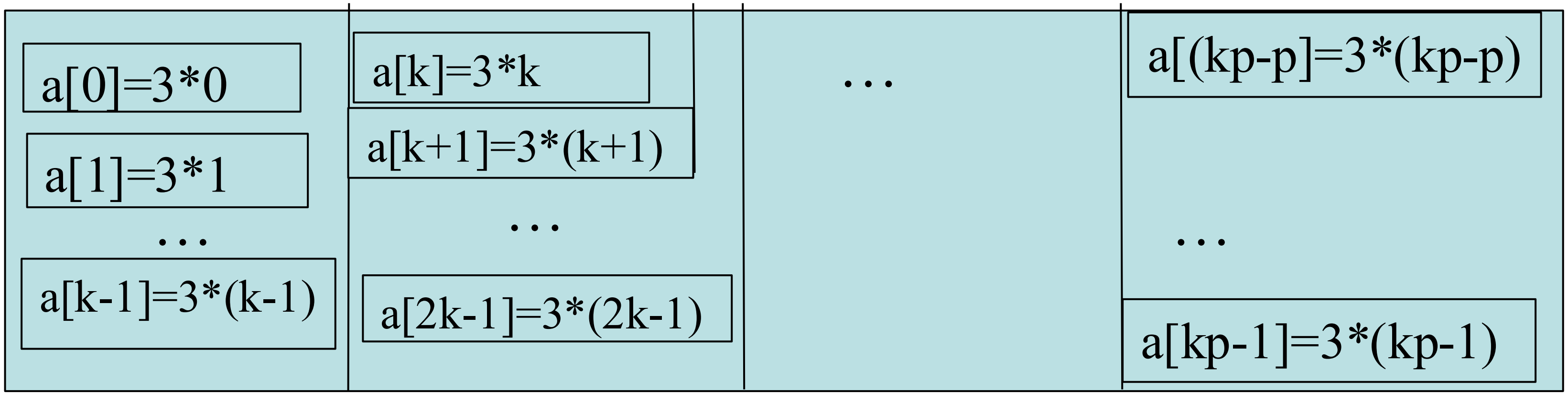

a[0]=3*0

a[1]=3*1

…

a[n-1]=3*(n-1)

Schedule Proc 0   Proc 1   Proc p-1


a[0]=3*0

a[1]=3*1

…

a[k-1]=3*(k-1)

a[k]=3*k

a[k+1]=3*(k+1)

…

a[2k-1]=3*(2k-1)

…

a[(kp-p]=3*(kp-p)

…

a[kp-1]=3*(kp-1)

SPMD code:

```
me=mynode();
p=noproc();
first=me*k
last = my_first+k-1
for (i=first; i<=last; i++)
        a[i]=3*i;
```

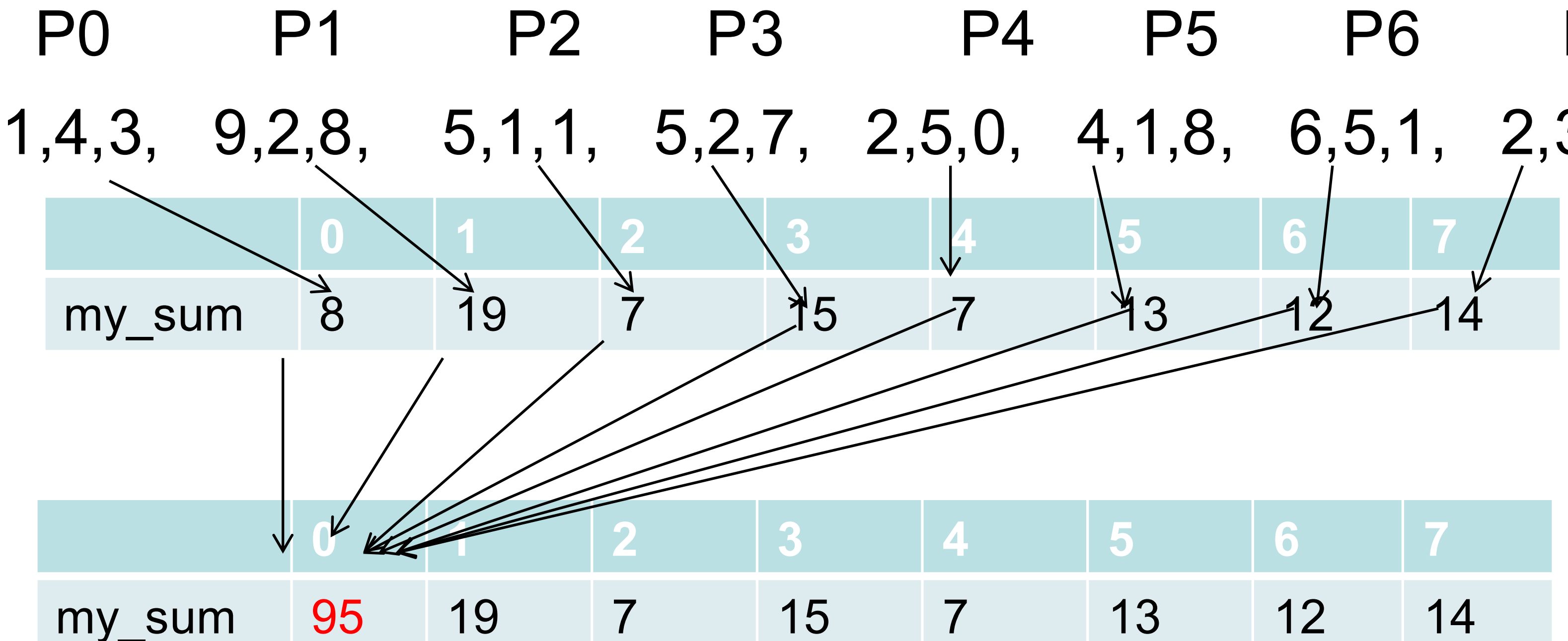
Steps in writing SPMD code

- **Step 1: Detect who I am**
  - my_rank= mynode(). p=noproc();
- **Step 2: Find the scope of computation performed in this processor based on my_rank and p values.**
  - For example, given n iterations in a sequential code
    - Derive first iteration to handle
    - Derive last iteration to handle
- **Step 3: Perform computation under the derived scope. Synchronize and communicate among tasks if needed.**

- When processors do not share memory
  - Use a message communication library
    - Send (msg, destination). Receive(msg) or Receive(msg, src)
- When processors share memory, synchronize through shared memory

Example of SPMD code structure with communication synchronization

- **Example: add n numbers on p processors**
  - Ex., p=8, n = 24
  - Once each processor adds its values to private my_sum, they form a global sum by sending results to a designated “master” core which adds the final result.



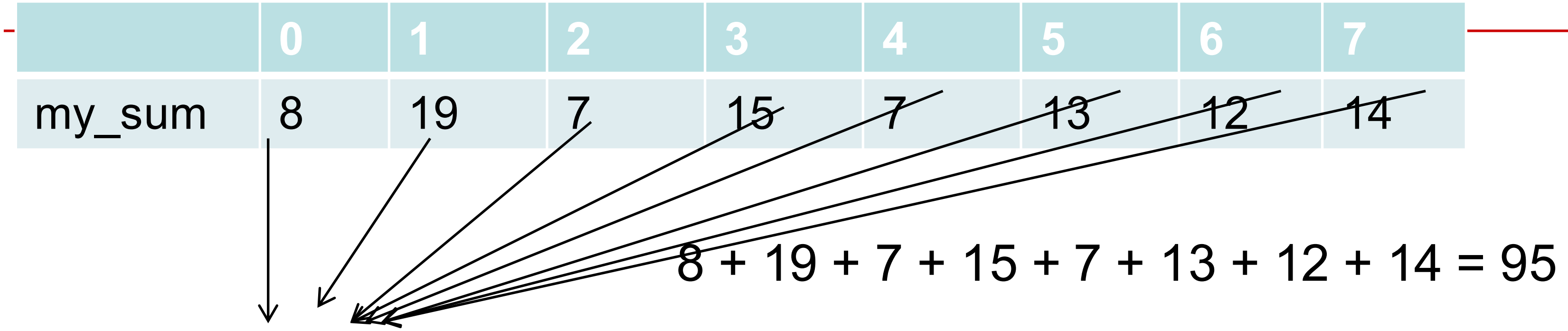
P0 P1 P2 P3 P4 P5 P6 P7

1,4,3, 9,2,8, 5,1,1, 5,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9

	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

	0	1	2	3	4	5	6	7
my_sum	95	19	7	15	7	13	12	14

SPMD code



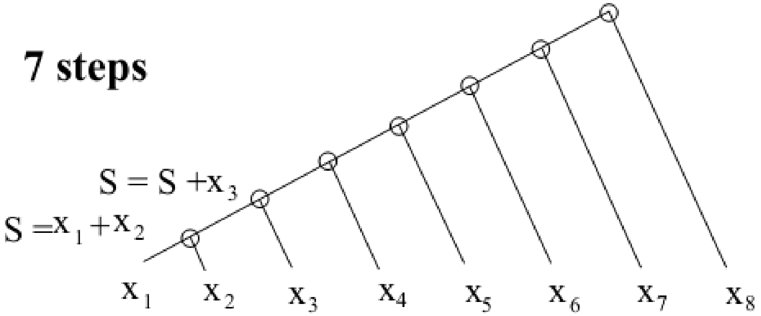
	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95

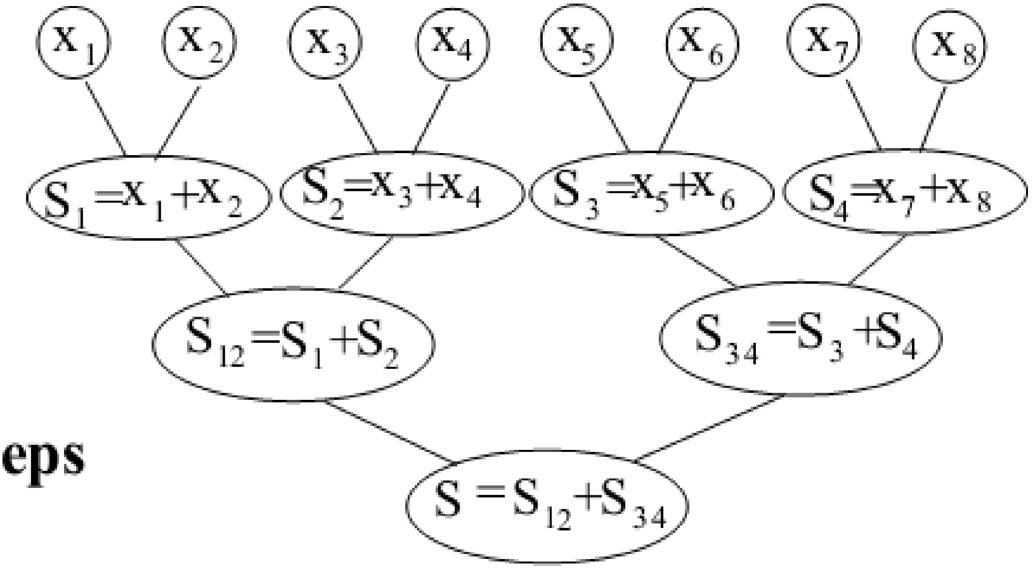
```
my_sum= Add all numbers assigned to this processor
If ( I am the master) {
        sum = my_sum;
        for  each of other processors do {
                Receive a value
                sum   = sum + received value
        }
} else {
        send my_sum to the master
}
```

Sequential vs tree-structured parallel addition

Sequential summation:
Master does all of the work to accumulate results



Tree summation structure for more parallelism.
What is the parallel time of executing this graph with p processors?

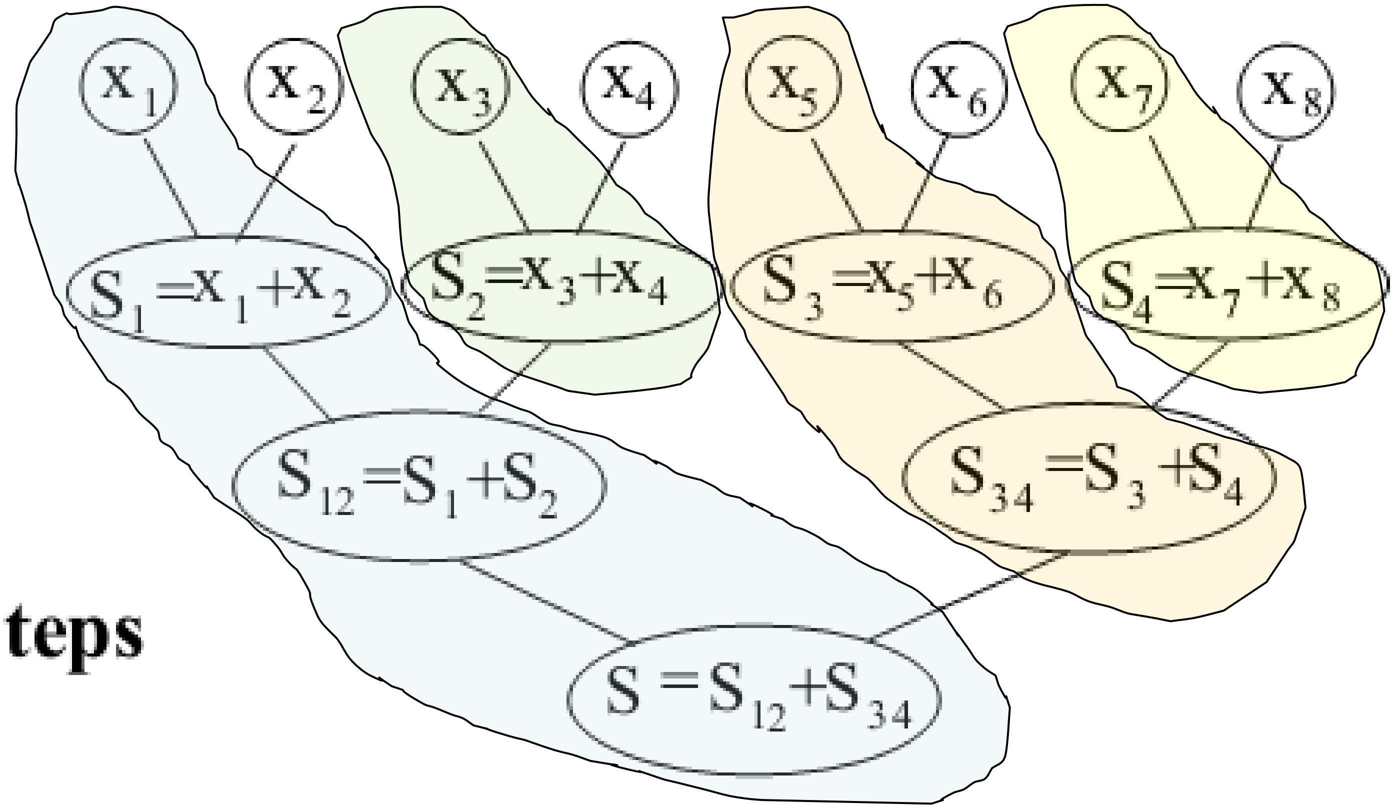


3 steps

Log (p)

Processor Assignment in Tree Summation for Parallel Addition

Processes: 0 1 2 3 4 5 6 7

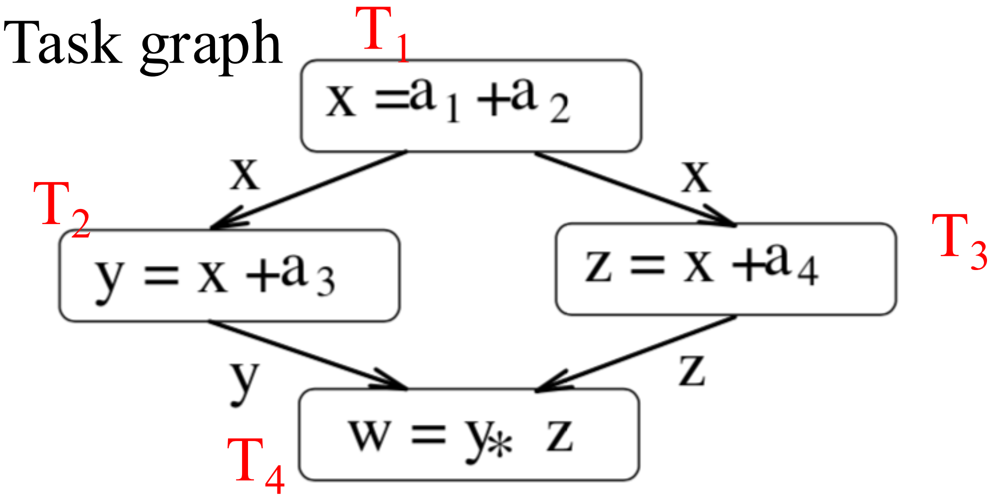


3 steps

When process communication is involved, what is the pattern of communication?

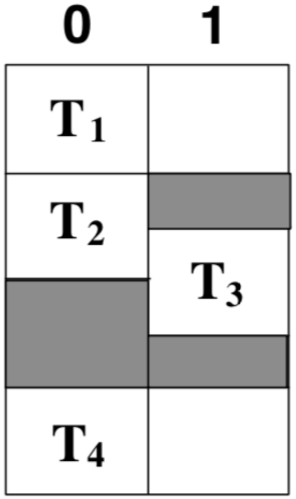
SPMD code for a task schedule with irregular pattern: inextensible with more processors

Task graph



$w = (a_1 + a_2 + a_3)(a_1 + a_2 + a_4)$

Schedule

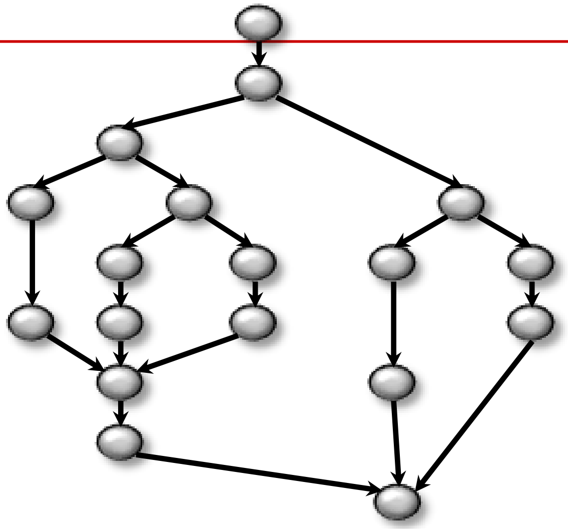


SPMD for 2 processors

```
int i, x, y, z, w,  a[5];
i = mynode();
if (i==0) then {
        x=a[1]+a[2];
        send(x, 1);
        y=x+a[3];
        receive(z);
        w=y*z;
}
else{
        receive(x);
        z=x+a[4];
        send(z,0);
}
```

Patterns of Graph Structure on Programming

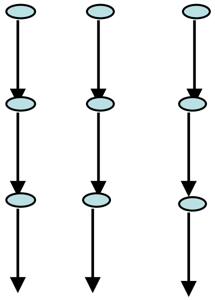
Difficult to write code and exploit parallelism for task graphs with irregular structure



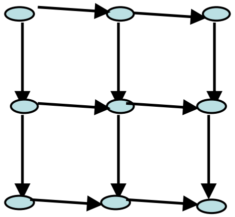
Difficult

Goal: Write simple SPMD code that scales well for a large number of processors.

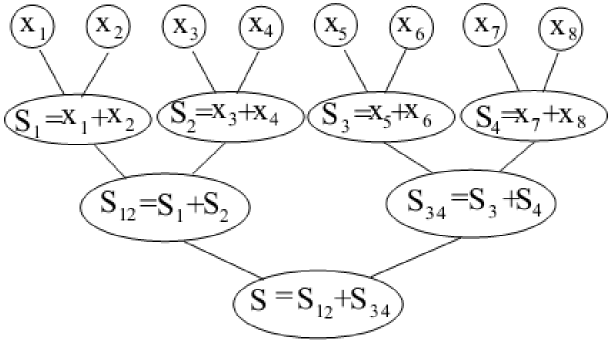
Strategies: Identify regularity of parallelism patterns



Trivial

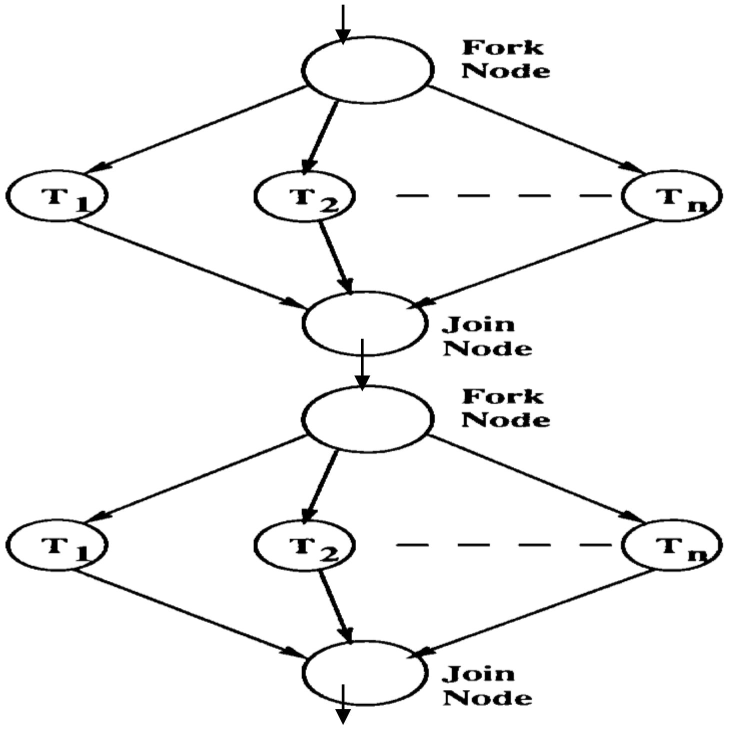


Easy/manageable



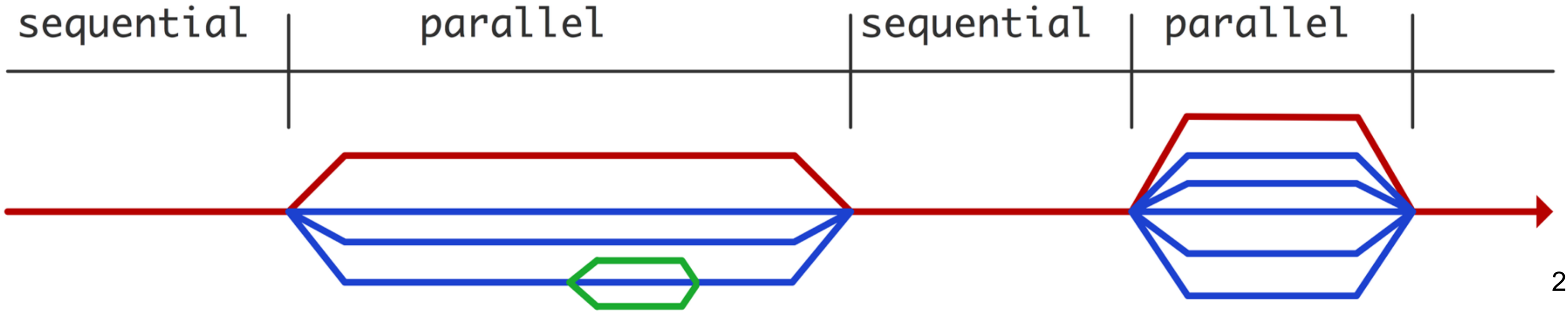
More difficult/manageable

Fork-Join Parallelism



Easy to write parallel code

Parallel code structure:



Coordination and Overhead

- **Coordination is needed among parallel tasks**
  - Communication – one or more cores send their current partial sums to another core.
    - How to communicate?
  - Load balancing – share the work evenly among the cores so that one is not heavily loaded.
  - Synchronization – because each core works at its own pace, make sure cores do not get too far ahead of the rest.
- **Pay attentions to overhead of coordination**
  - Not worthy to add a few numbers in parallel. Too much overhead.
  - Aggregation of small tasks is needed

What we'll be doing

- Learning to write programs that are explicitly parallel with different **extensions to C.**
  - Message-Passing Interface (MPI)
  - POSIX Threads (Pthreads)
  - OpenMP
  - GPU programming with Cuda
- **Terminology:**
  - Concurrent computing – a program is one in which multiple tasks can be in progress at any instant. These tasks can still run in one CPU.
  - Parallel computing – a program is one in which multiple tasks cooperate closely to solve a problem. These tasks often or periodically run on multiple CPUs or machines in parallel.
  - Distributed computing – a program may need to cooperate with other programs to solve a problem. These programs/tasks typically run on multiple machines without shared memory.They do not have to run in parallel in solving a problem

Concluding Remarks

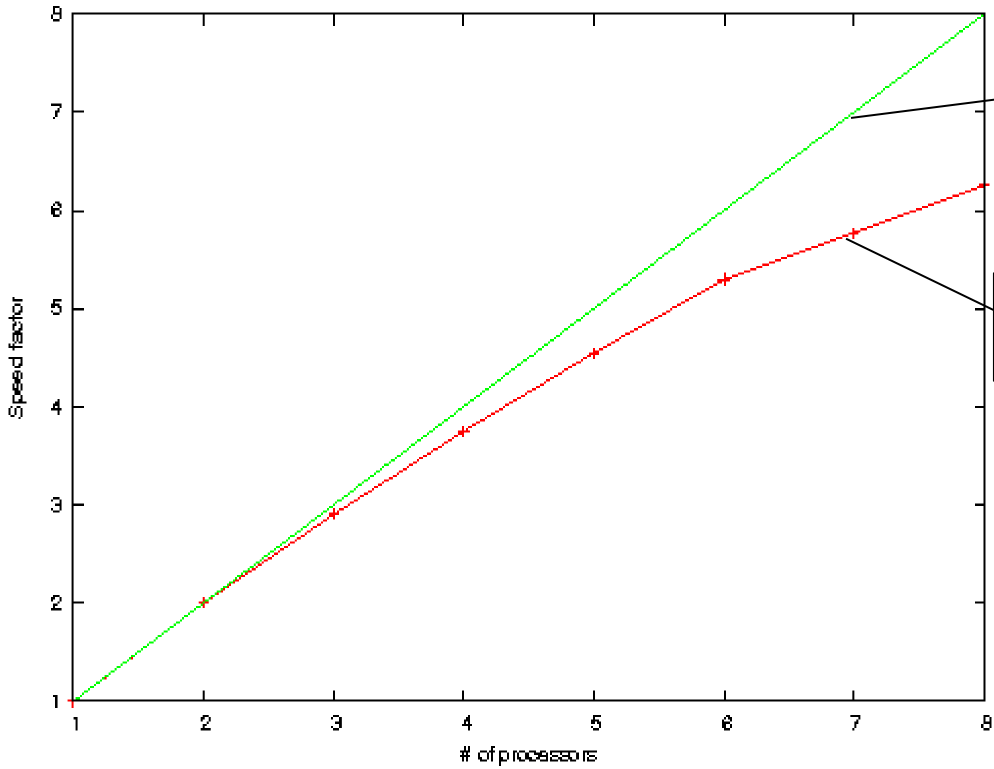
- **Task/data mapping is essential for writing parallel programs.**
  - Model parallelism using a set of tasks
  - Schedule tasks to a set of process units
  - Write SPMD code
- **Parallelism management involves coordination of cores/machines**
  - Coordination does cost extra overhead
  - Design goal is to let the benefits of parallel processing outweigh the cost of overhead paid.

Evaluation of Parallel Performance

Performance Terminology: Speedup

- **Number of processors = p**
- **Sequential time of execution = T_{serial}**
- **Parallel time = $T_{parallel}$**

Speedup $$S = \frac{T_{serial}}{T_{parallel}}$$



Perfect speedup

Actual speedup

Speedup Interpretation $T_{parallel} = T_{serial} / p$

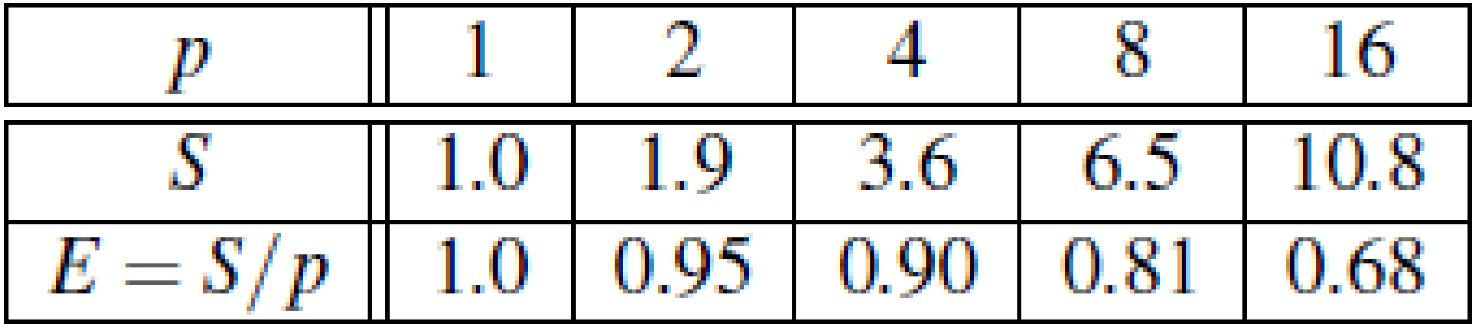
- *Linear speedup*
  - Speedup proportionally increases as p increases
- *Perfect linear speedup*
  - Speedup $= p$
- *Superlinear speedup*
  - Speedup $> p$
  - It is not possible in theory.
  - It is possible in practice
    - Data in sequential code does not fit into memory.
    - Parallel code divides data into many machines and they fit into memory.

Efficiency of a parallel program

$$E = \frac{\text{Speedup}}{p} = \frac{\left(\dfrac{T_{serial}}{T_{parallel}}\right)}{p} = \frac{T_{serial}}{p \cdot T_{parallel}}$$

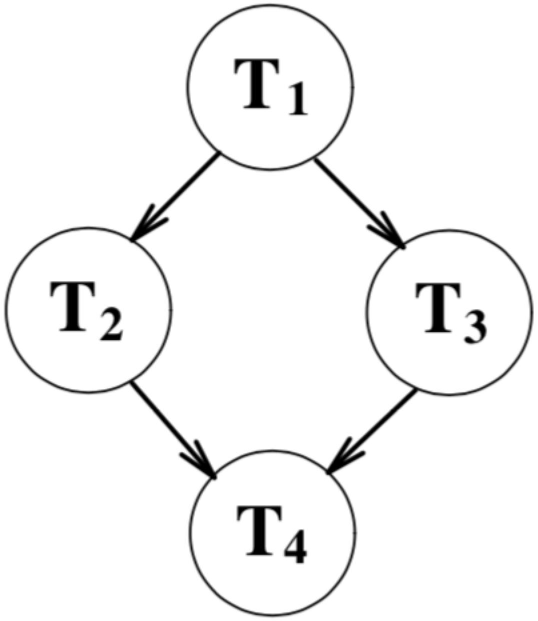
Measure how well-utilized the processors are, compared to effort wasted in communication and synchronization.

Example:



p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
$E = S/p$	1.0	0.95	0.90	0.81	0.68

Speedup & efficiency with two task schedules?

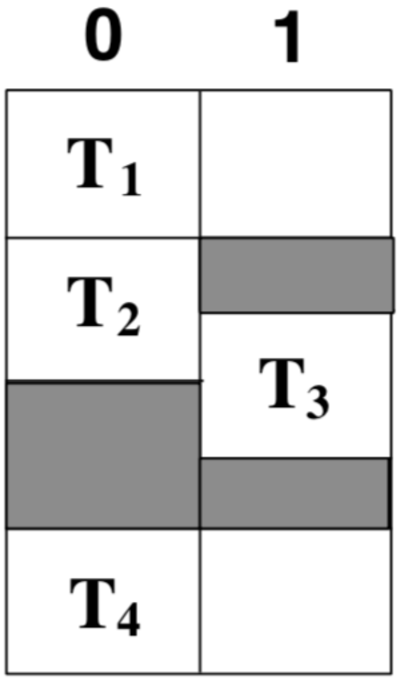


Sequential time = 4

0	1
T_1	
T_2	T_3
T_4	

$\tau = 1$
c = 0

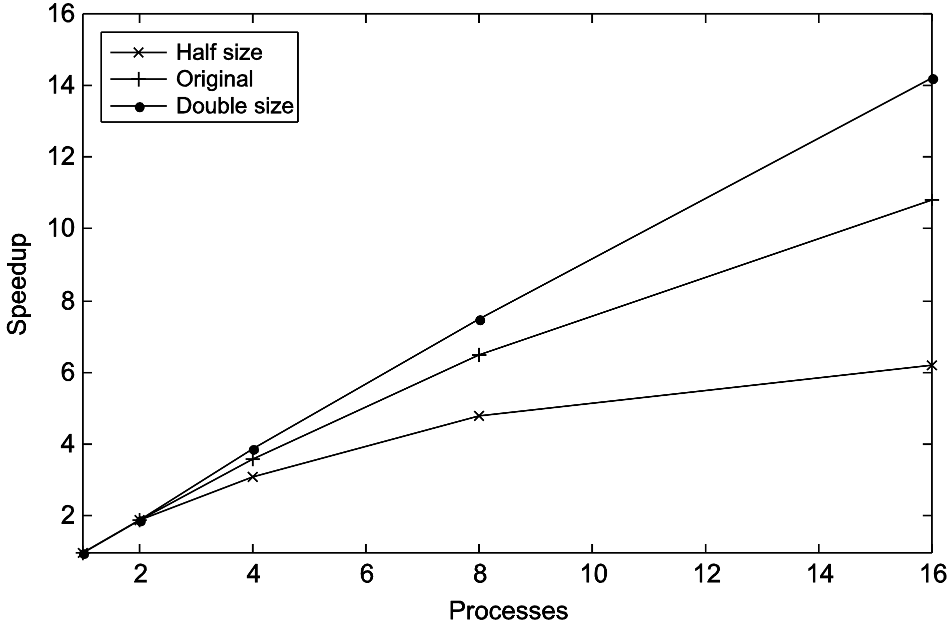
Parallel time= 3
Speedup = 4/3
Efficiency=2/3



$\tau = 1$
c = 0.5

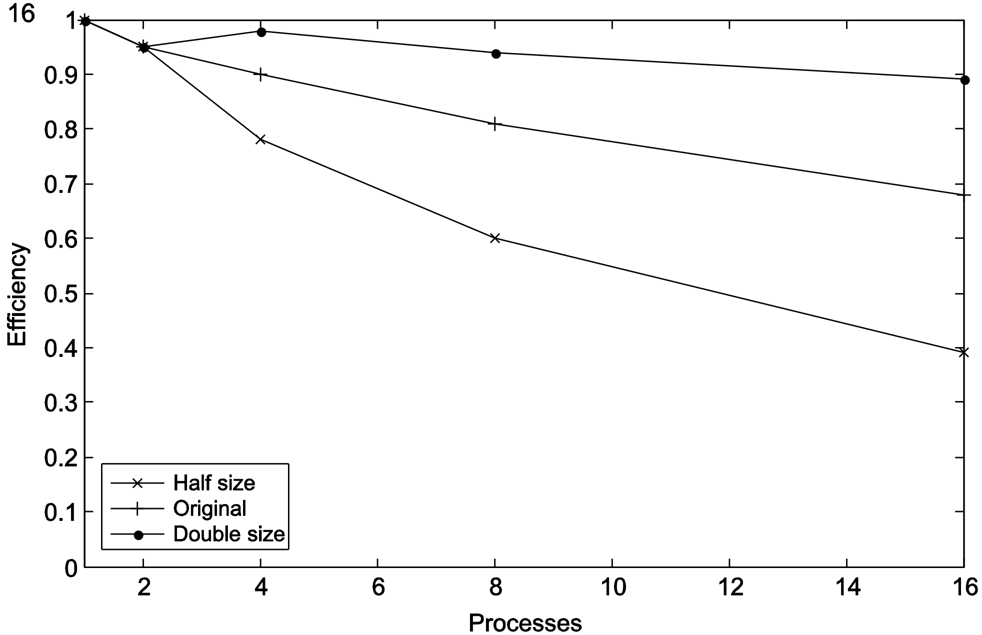
Parallel time= 4
Speedup = 1
Efficiency=1/2=50%

Problem Size Impact on Speedup and Efficiency



- Efficiency often goes down as more cores are used
- Efficiency often increases when increasing problem size

- Original problem size (e.g. measured by matrix size)
- Half of original problem size
- Double size



Impact of Limited Parallelism in a Program

- **Suppose only part of an application can be parallelized**
- **Amdahl's law**
  - let x be the fraction of work done sequentially, so (1-x) is fraction parallelizable
  - p = number of processors
  - Parallel time= (x + (1-x)/p) *Sequential time

$$\text{Speedup}(p) = \text{Sequential time/Parallel time}$$

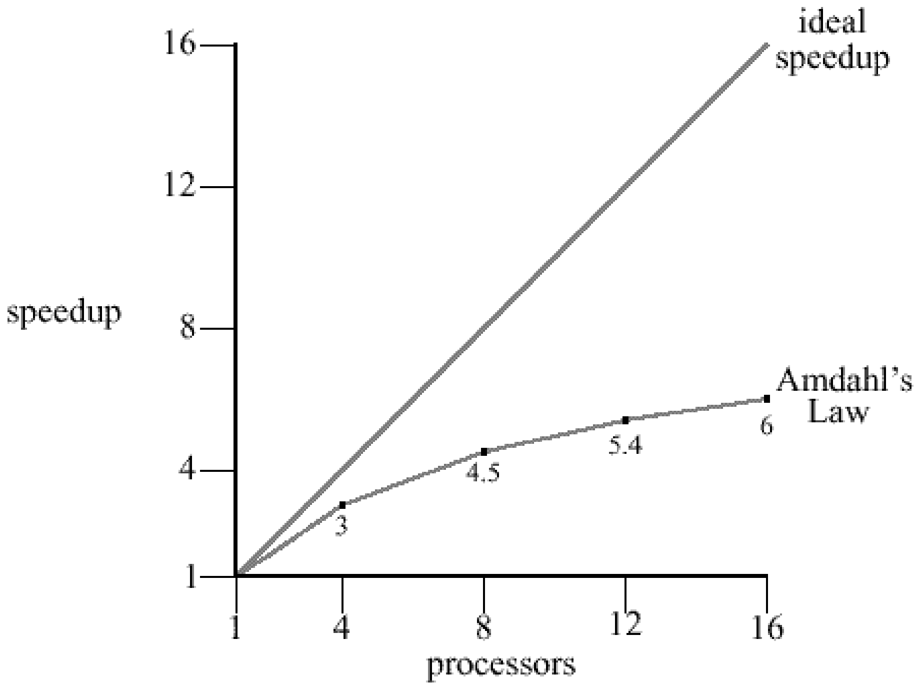
$$\leq 1/(x + (1\text{-}x)/p)$$

$$< 1/x$$

- Takeaway: Even if the parallel part speeds up perfectly performance is limited by the sequential part

Amdahl Law: Limitation of Parallel Performance

- The possible speedup is limited by available parallelism — regardless of the number of processors available.



X=1/9	1/[x+(1-x)/p] =9/[1+8/p]
p=4	3
p=8	4.5
P=12	5.4
P=16	6

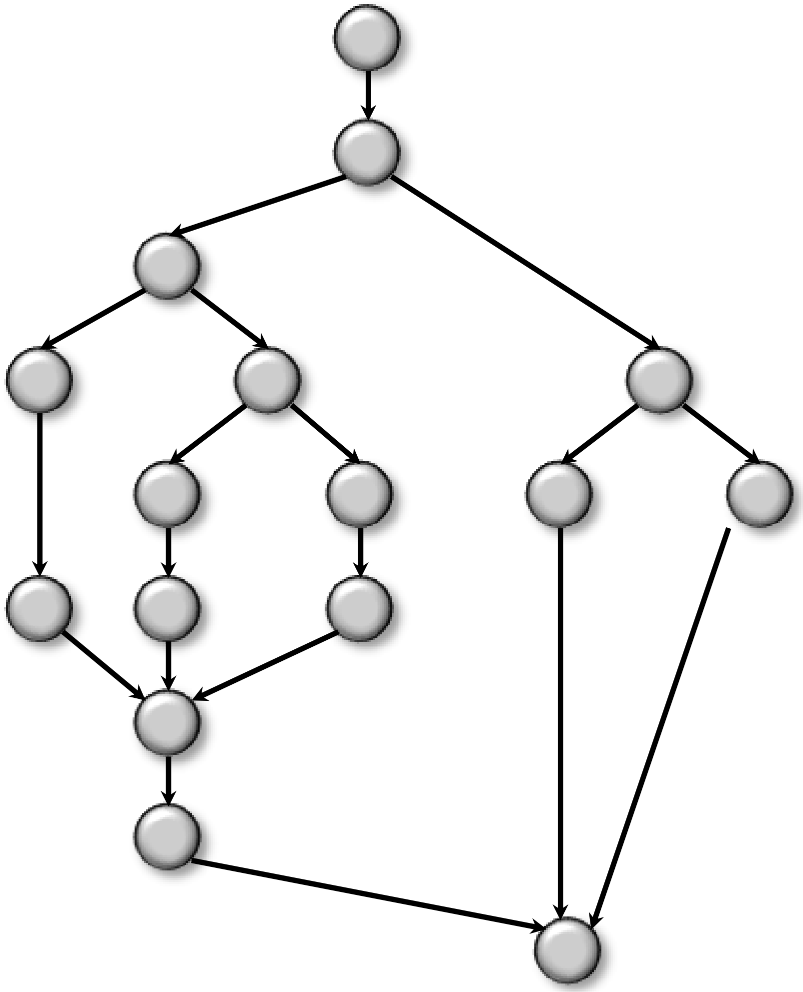
Speedup(p) < 1/x=9

Performance models and measurement in practice

- **Evaluate and predict with a theoretical analysis**
  - Model execution with directed acyclic graph
  - Model the latency / bandwidth for message communication and disk IO
  - Estimate parallel time and speedup.
  - Predict the best and average performance.
- **For time and speedup measurement of actual execution:**
  - Measure execution time with a timer function in binary code running
  - Assess the cost of data transmission and I/O with with a timer

Model Performance in a Directed Acyclic Graph

t_p= execution time on p processors

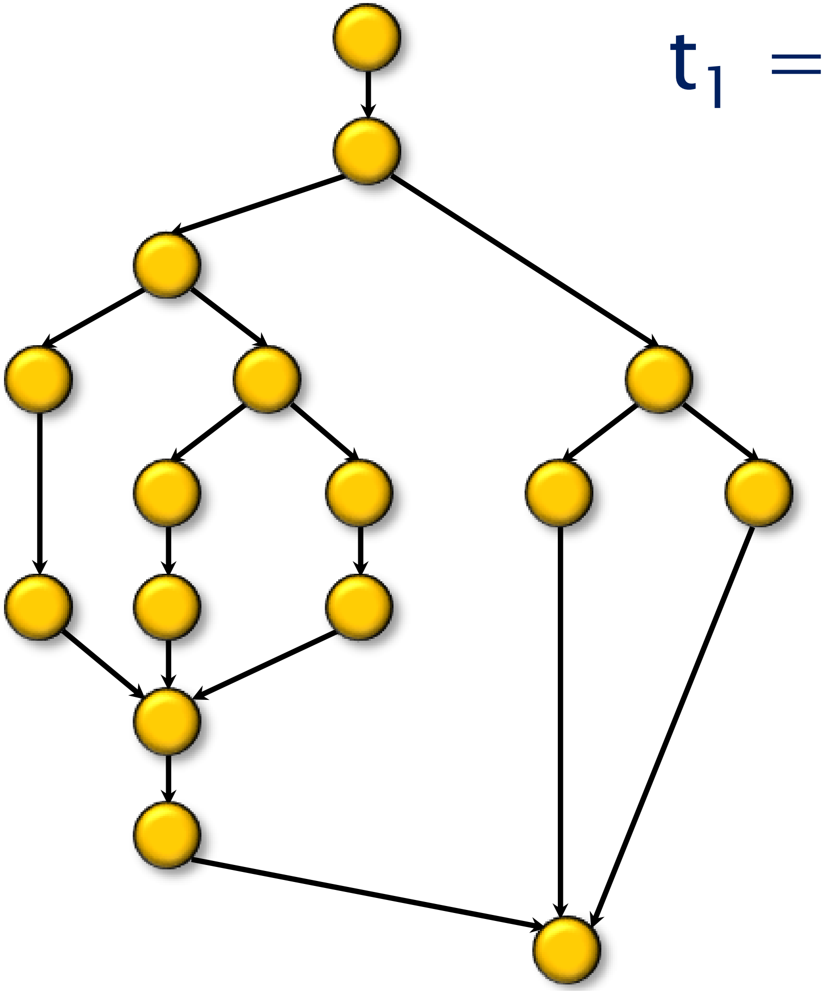


John Gilbert 240A

Bound analysis for parallel performance

t_p = execution time on p processors

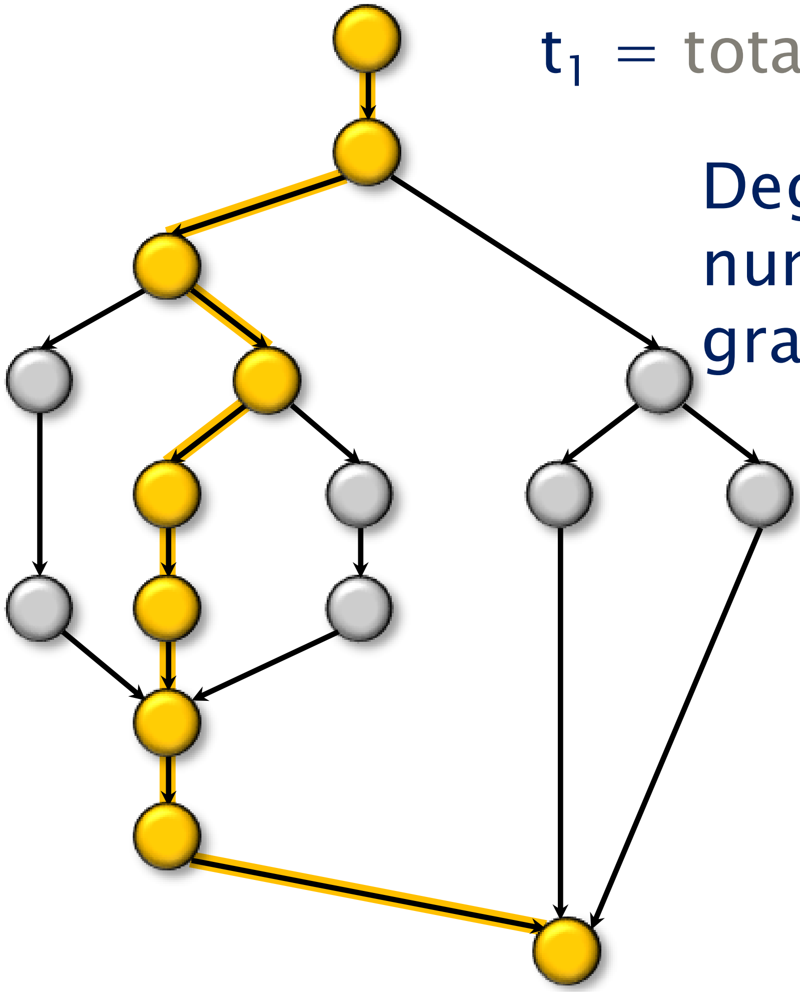
t_1 = ***workload***



Degree of parallelism –
Maximum number of
Independent tasks

Bound analysis for parallel performance

t_p = execution time on p processors



t_1 = total *workload* 16 with equal weights

Degree of parallelism – Maximum number of independent tasks in this graph 5

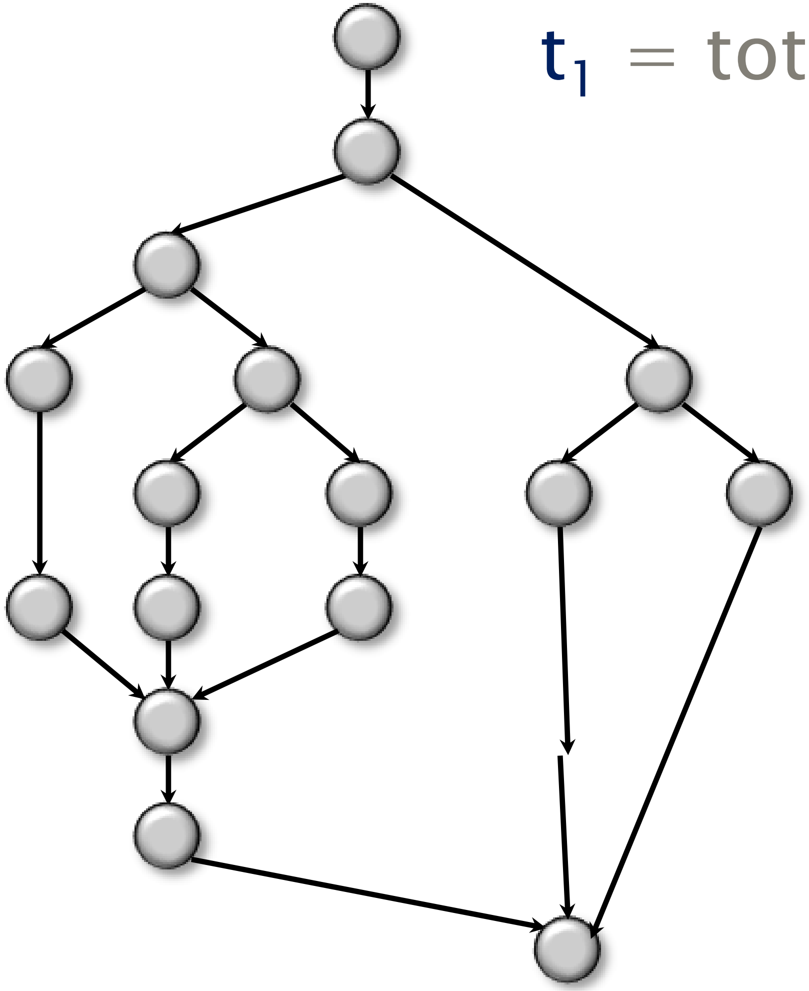
t_∞ = *critical path length*

length of longest path summing computation cost in this path

9

Bound analysis for parallel performance

t_p = execution time on p processors



t_1 = total *work*

t_∞ = *critical path length*

Minimum workload

- $t_p \geq t_1 / p$

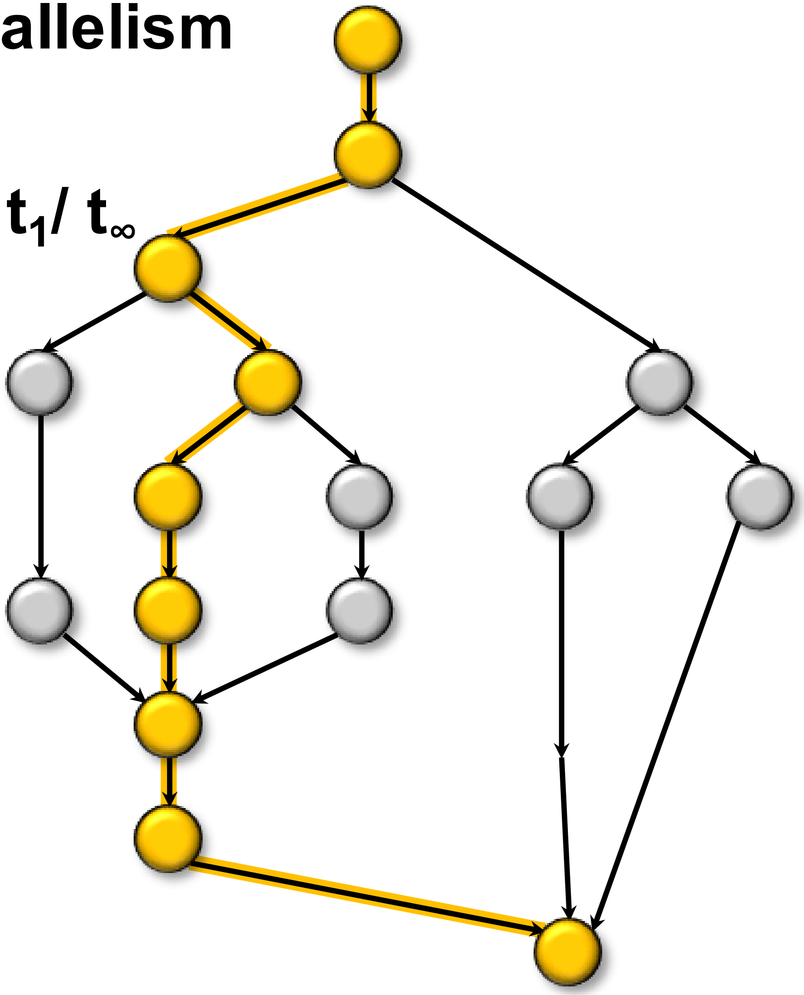
Limit by critical path length

- $t_p \geq t_\infty$

Maximum speed with *p* processors

Speedup = $t_1/t_P \leq$ degree of parallelism

Since $t_p \geq t_\infty$, Speedup = $t_1/t_P \leq t_1/\ t_\infty$



Degree of parallelism=5

$t_1/\ t_\infty$ = 16/9=1.778

Speedup ≤ min(5,1.778)=1.778

Estimate Time with Complexity Analysis

- Estimate cost of an sequential task (algorithm), given an input problem size (e.g. The size of an input array)
- Often, computer sciences uses Big O notation to describe how fast the run-time of an algorithm grows as problem size grows
- Examples of big-O categories with input size N

c	constant	O(1)
log N	logarithmic	O(*log N*)
N	linear	O(*N*)
N^2	quadratic	O(N^2)
N^3	cubic	O(N^3)
2^N	exponential	O(2^N)

This course does not use big-O but counts main task cost

Cost Analysis with Operation Counting

Analyze task cost by counting main arithmetic operations & space units

- Ignore less significant terms in cost expression
- Ignore loop overhead in algorithm implementation

- Example

```
for ( i=1to N ) {  a[i] =  2*i }    → N multiplications
```

```
y=2*x; for ( i=1to N ) {  a[i] =  a[i]+ i*b[y] }

  → 2N+1  addition/multiplications             → 2N
```

```
for ( i=1to N ) {  a[i] =  2*i }                 $N$
      for ( i=1 to N ) {
         x=i*2;
         for ( j=1 to N ) { a[i] = a[j] + x }    $N^2 + N$
      }
```

Total $N^2 + 2N$, which is still N^2

How to measure sequential and parallel time in code execution

- **Select a program segment of interest**
  - Setup startup time
  - Measure finish time
  - Report the time difference
- **What timing function to use?**
  - CPU time (e.g. as reported by Linux clock() call).
    - It only includes CPU time of a single process.
    - It does not work if your parallel program involves multiple processes.
  - Wall clock time
    - Suitable for a parallel program with multiple processes.
    - But impacted by other shared users when a system is not dedicated.

Performance assessment with FLOPS

- **How to assess if parallel/serial code is fully optimized, taking advantages of hardware**
  - Measure quality of implementation in terms of FLOPS

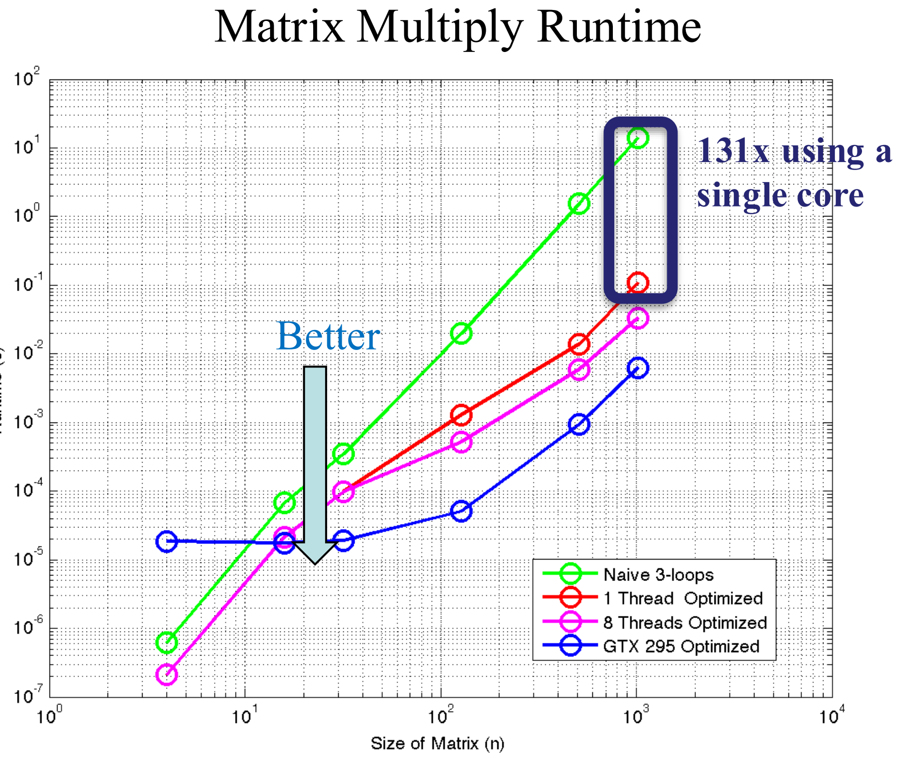
- FLOPS number (Megaflops, gigaflops or teraflops)

  = Core computation count / time spent

  - Only count core arithmetic operations

- Example: Matrix-matrix multiplication
  - Operation count = ~2 n^3 with matrix dimension n
  - 3 GFLOPS → 3 billion useful floating operations performed per second

Why care about serial performance in parallel computing?


Matrix Multiply Runtime

Better

131x using a single core

x: Matrix size
y: Running time
Both x and y axes are log-scale

From Berkeley's CS267 Lecture by A Buluc and J. Demmel

- Parallelizing slow serial code only produces slow parallel code

- Linear speedup is not necessarily fast code

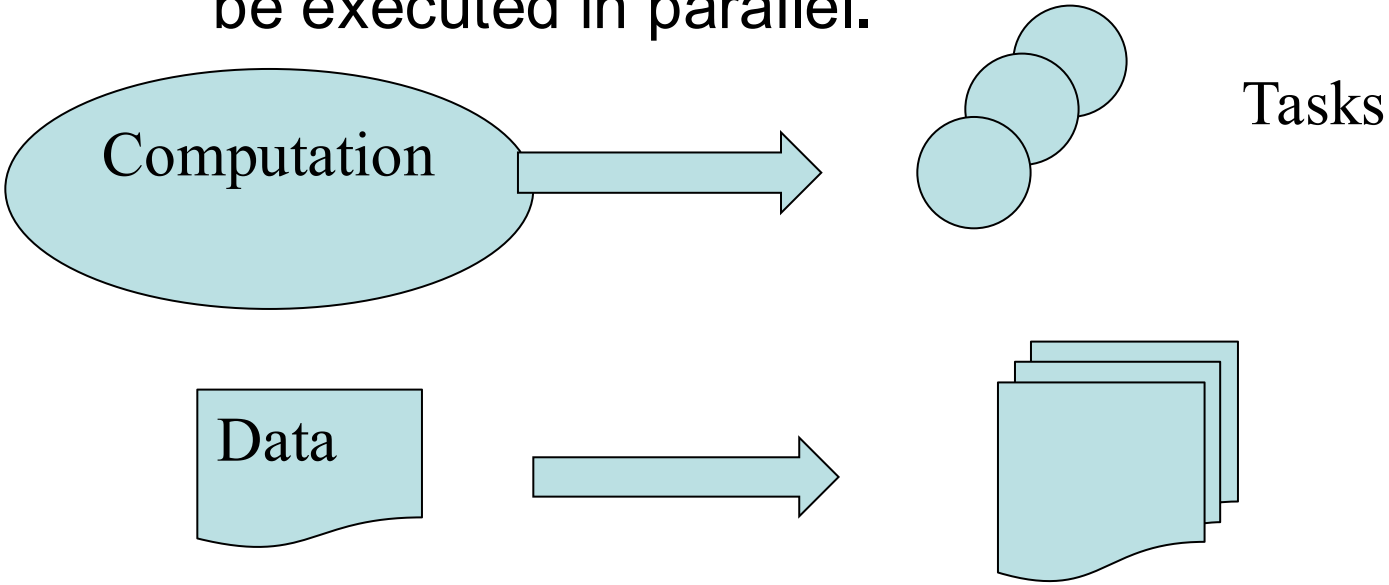
- As an example, 10x optimization for serial matrix multiply is possible with
  - SIMD vectorization
  - Blocked computation for better cache utilization
  - Or by using optimized library functions

Revisit: Steps on Parallel Program Design

Based on Foster's Methodology

1. **Partitioning:** divide the computation to be performed and the data operated on by the computation into small tasks.

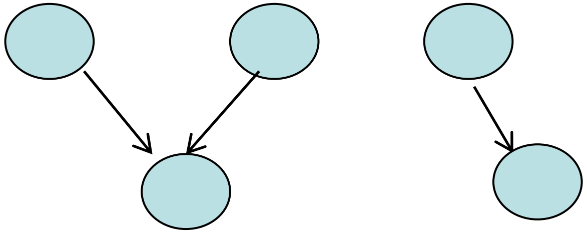
   The focus here should be on identifying tasks that can be executed in parallel.



Steps 2&3: Task dependence and aggregation

Step 2: Task dependence & ommunication

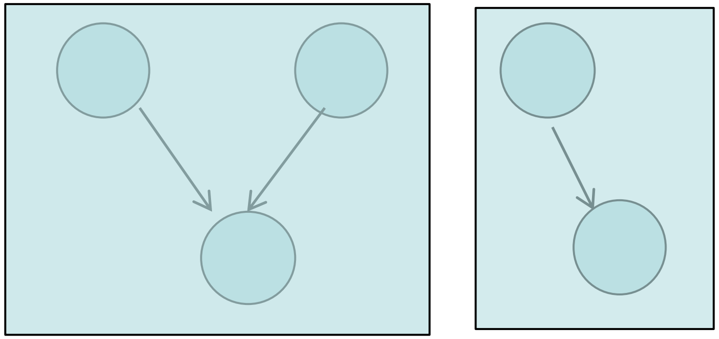
- Identify dependence among tasks
- Determine inter-task communication



Step 3: Task **aggregation**

Combine small tasks and communications identified in the first step into larger tasks.

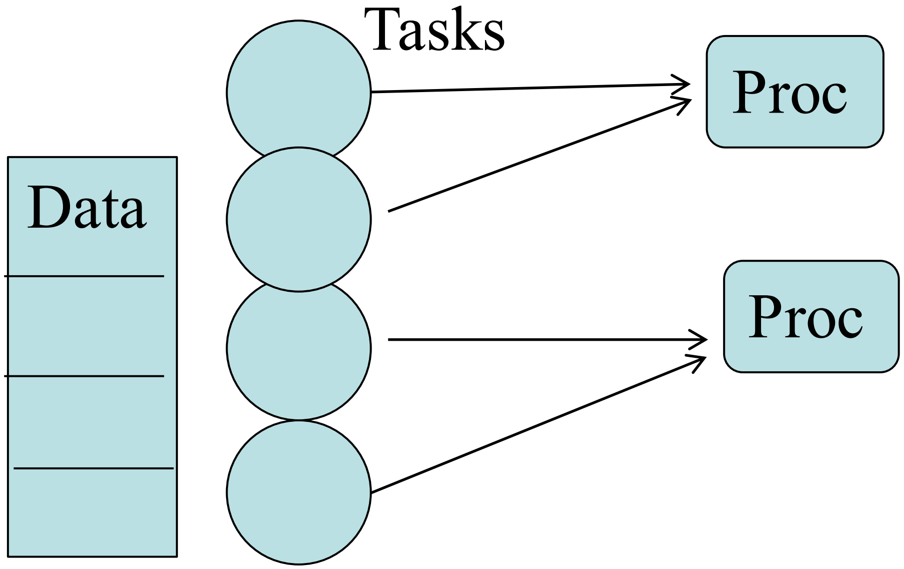
- Reduce communication overhead →Coarse grain tasks
- May reduce parallelism sometime



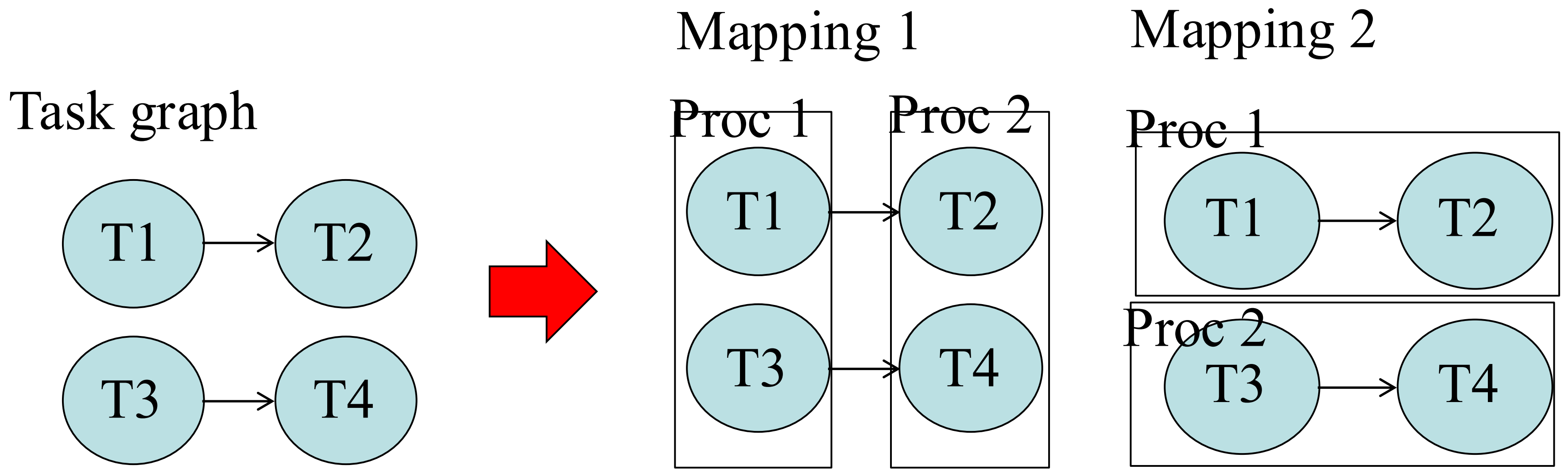
Step 4: Mapping/Scheduling

- Assign the composite tasks identified in the previous step to processes/threads. Distribute data if needed

  This should be done so that communication is minimized, and each process/thread gets even workload.
- Develop a schedule for task execution



Optimization for Less Communication and Faster Schedule on Distributed Memory Architecture



Task graph

T1 T2

T3 T4

Mapping 1

Proc 1 Proc 2

T1 T2

T3 T4

Mapping 2

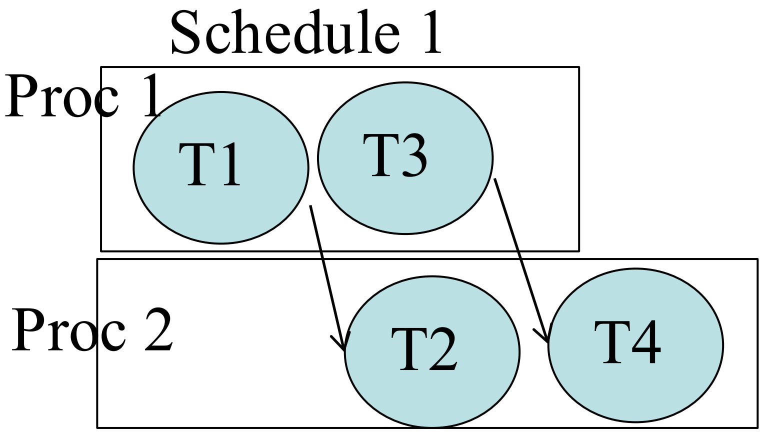
Proc 1

T1 T2

Proc 2

T3 T4

Which mapping is better? #2



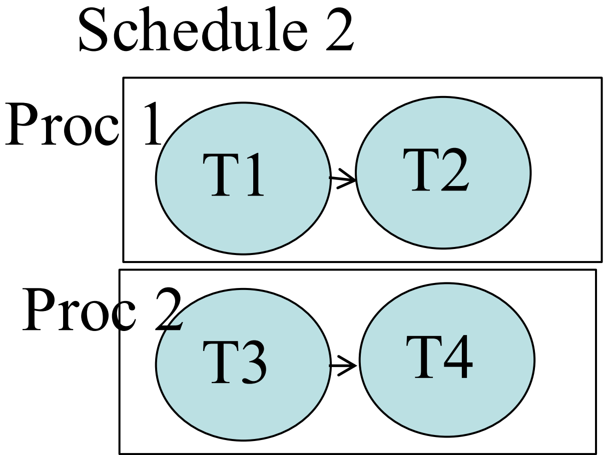
Schedule 1

Proc 1

T1 T3

Proc 2

T2 T4



Schedule 2

Proc 1

T1 T2

Proc 2

T3 T4

Concluding Remarks

- **Parallel hardware**
  - Shared memory and distributed memory architectures
  - Network topology for interconnect
- **Parallel software with SPMD programs**
- **Performance**
  - Speedup/Efficiency. Amdahl's law. Scalability
- **Parallel Program Design**
  - Partitioning-->dependence →mapping/scheduling

Class Update: Exercise 1 due Wed.

Expanse account: 1) Create an ACCESS account in access-ci.org.

2) Inform your account name by filling a Google sheet posted in Piazza.

3) After CPU allocation addition to your account, you can login to the Expanse cluster