CPSC/ECE 4780/6780

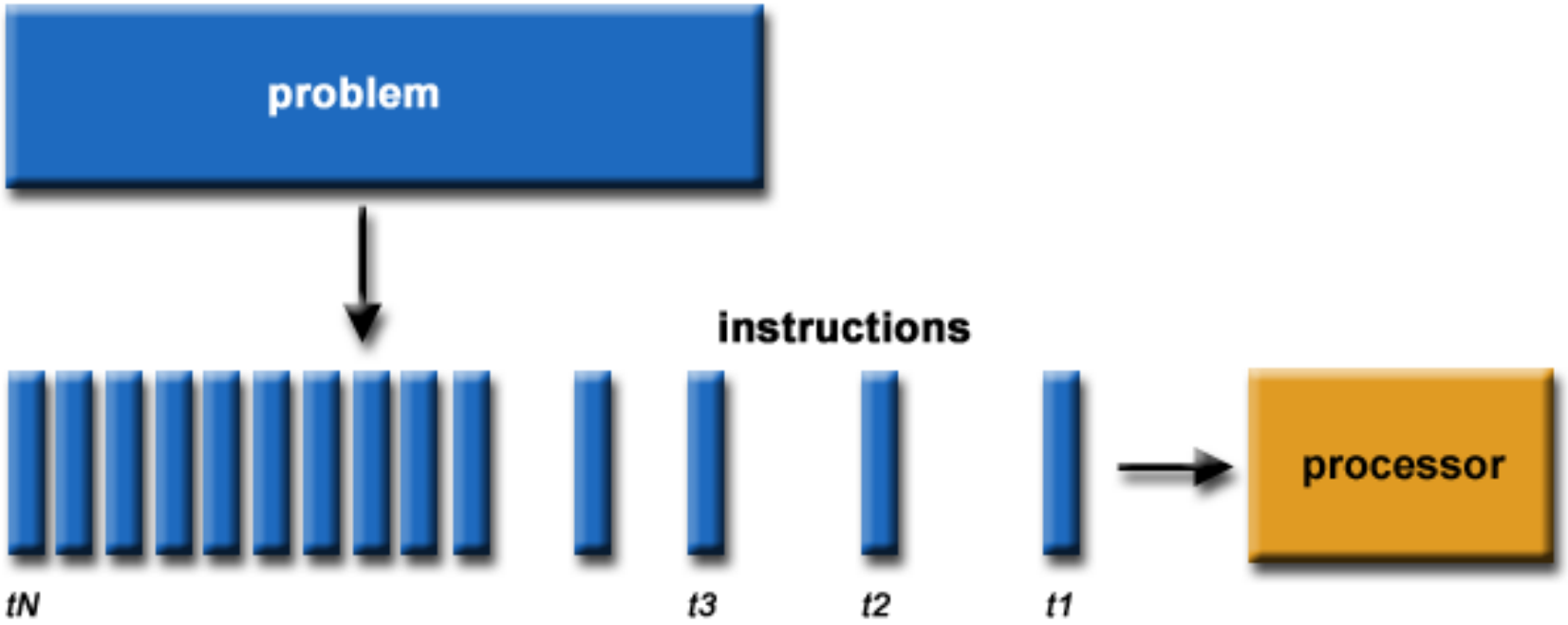
General-Purpose Computation on Graphical Processing Units (GPGPU)

Lecture 1: Parallel Computing and Palmetto Cluster

Basics of Parallel Computing

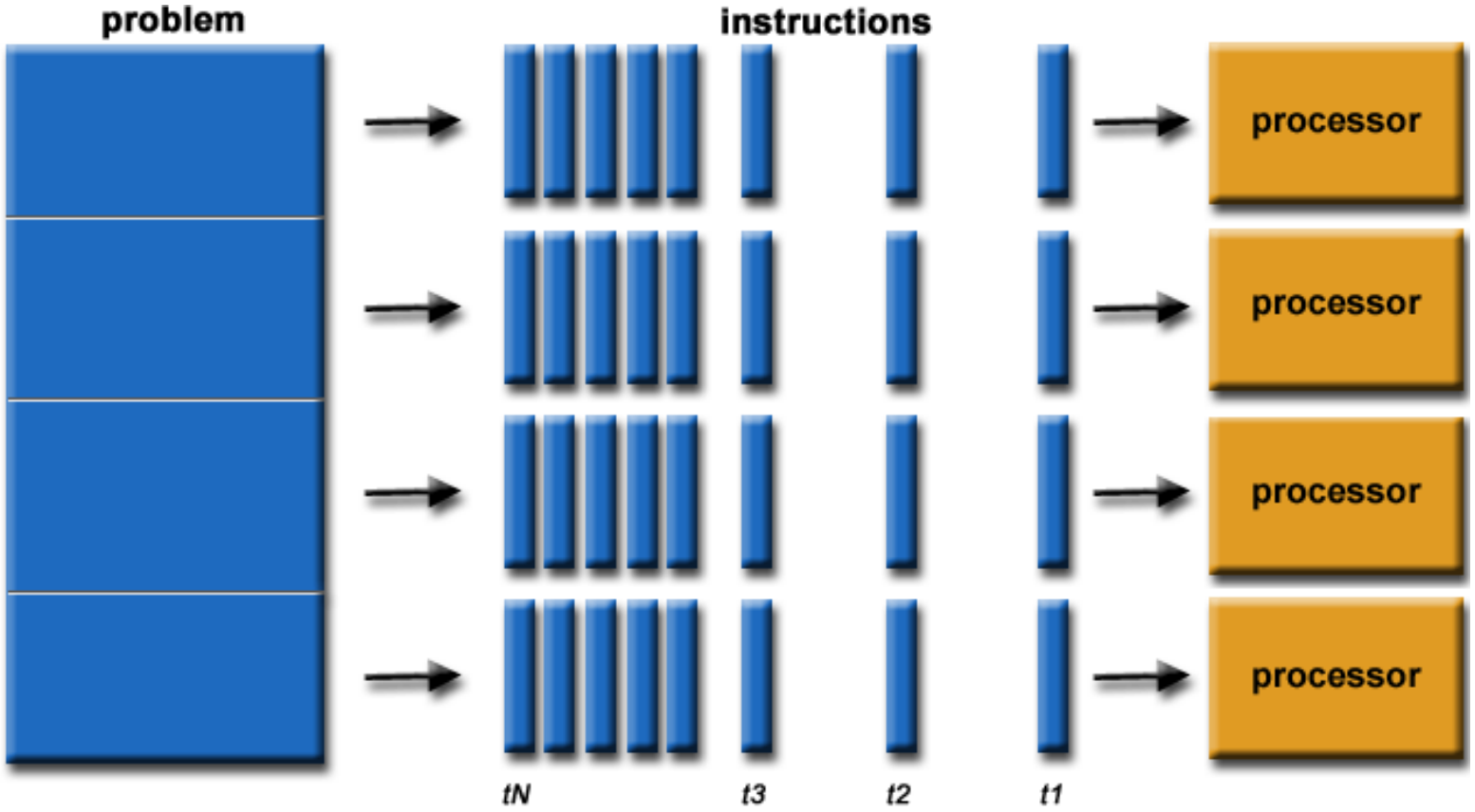
What is Serial Computing?

- Serial Computing



What is Parallel Computing?

- Parallel Computing – the simultaneous use of multiple compute resources to solve a computational problem.



Why Use Parallel Computing?

- The real world is massively parallel

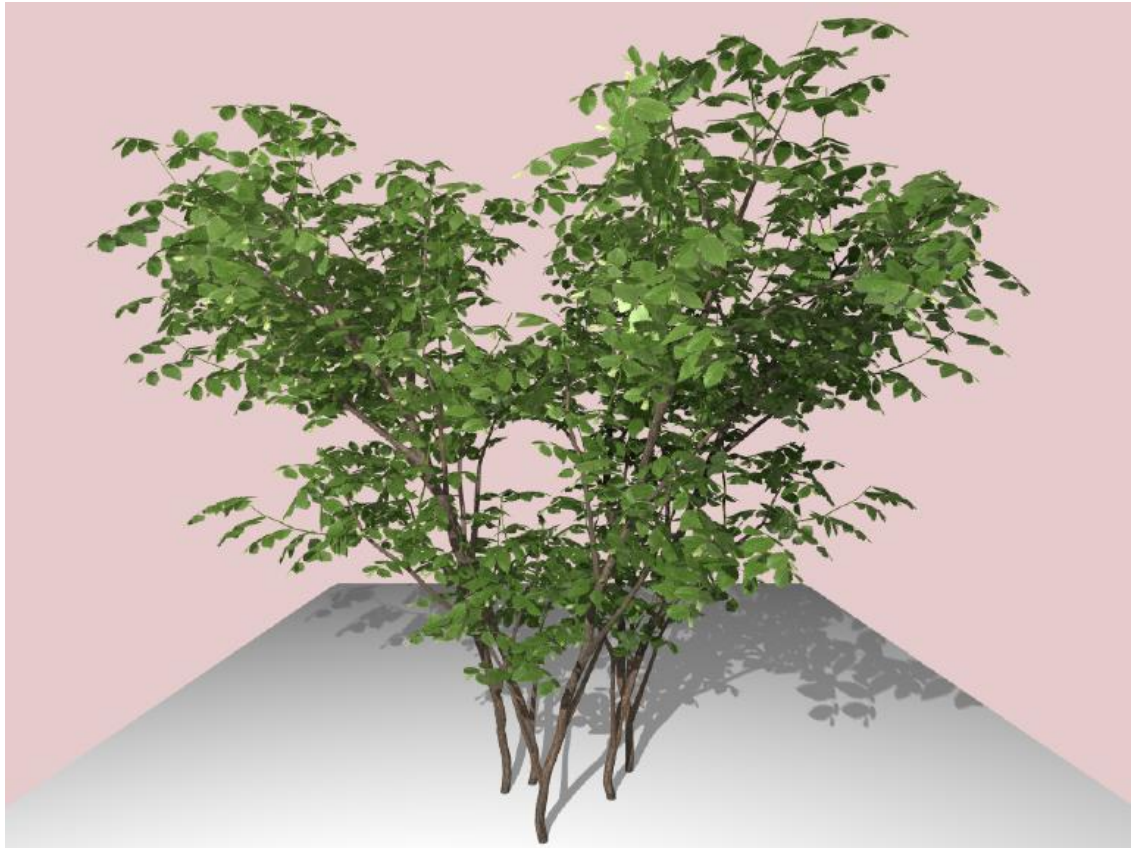


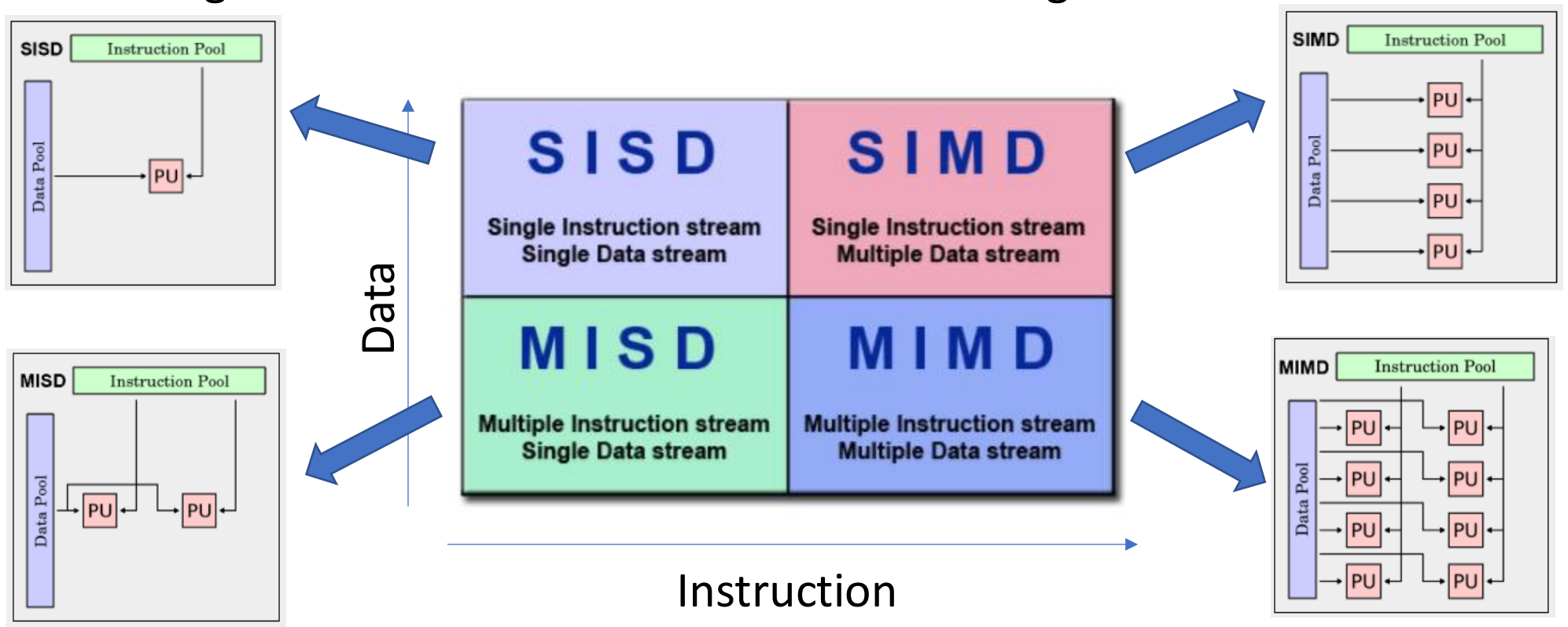
Figure 1: Global Illumination Example.

Main Reasons

- Save time and/or money
- Solve larger/more complex problems
  - E.g., Grand Challenge Problems, web search engines/database
- Provide concurrency
  - E.g., Collaborative Networks
- Take advantage of non-local resources
- Make better use of underlying parallel hardware

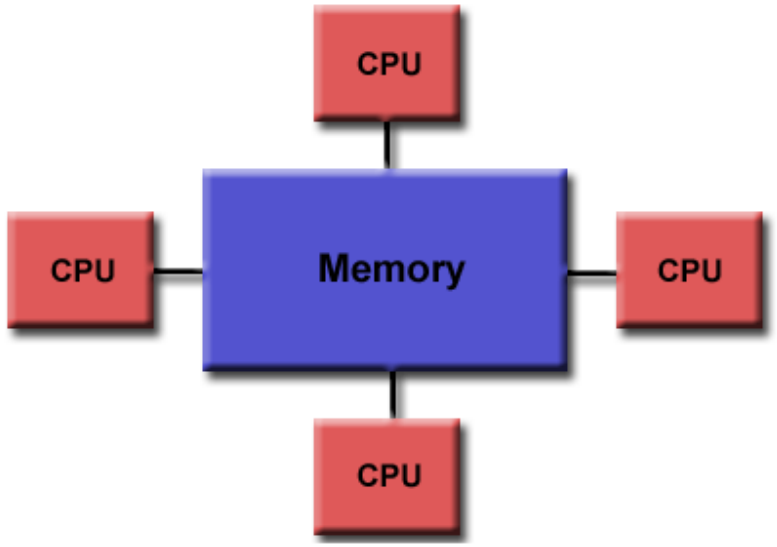
Computer Architecture

- Flynn's Taxonomy
  - Classifies multi-processor computer architectures into four different types according to how instructions and data flow through cores.



Multi-processor Computer Memory Architectures – Shared Memory

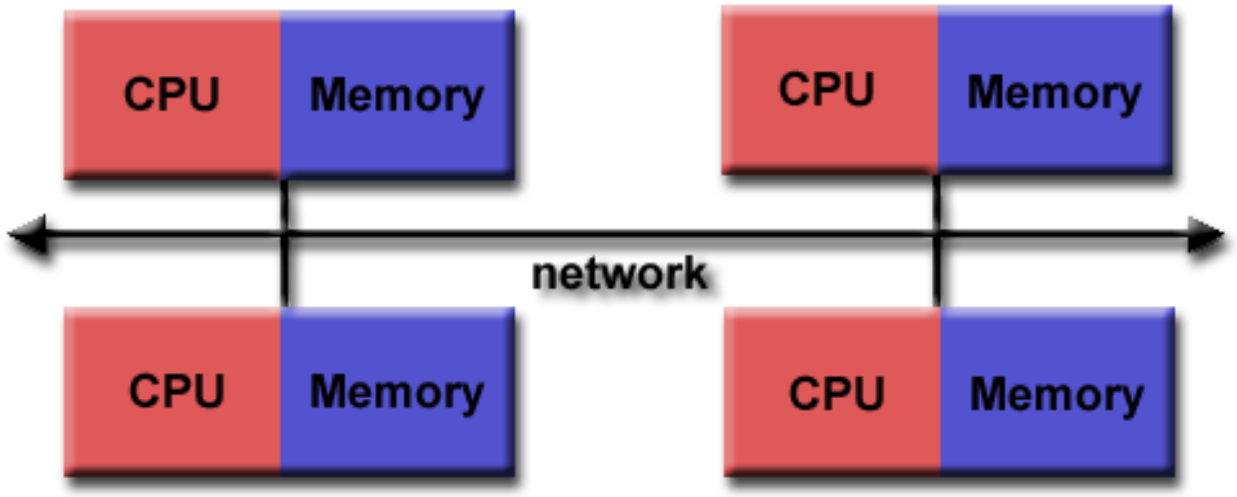
- General characteristics
  - Global memory address space
  - Global visibility
- Advantages
  - User-friendly programming
  - Fast and uniform data sharing
- Disadvantages
  - Lack of scalability between memory and CPUs



Shared Memory Access

Multi-processor Computer Memory Architectures – Distributed Memory

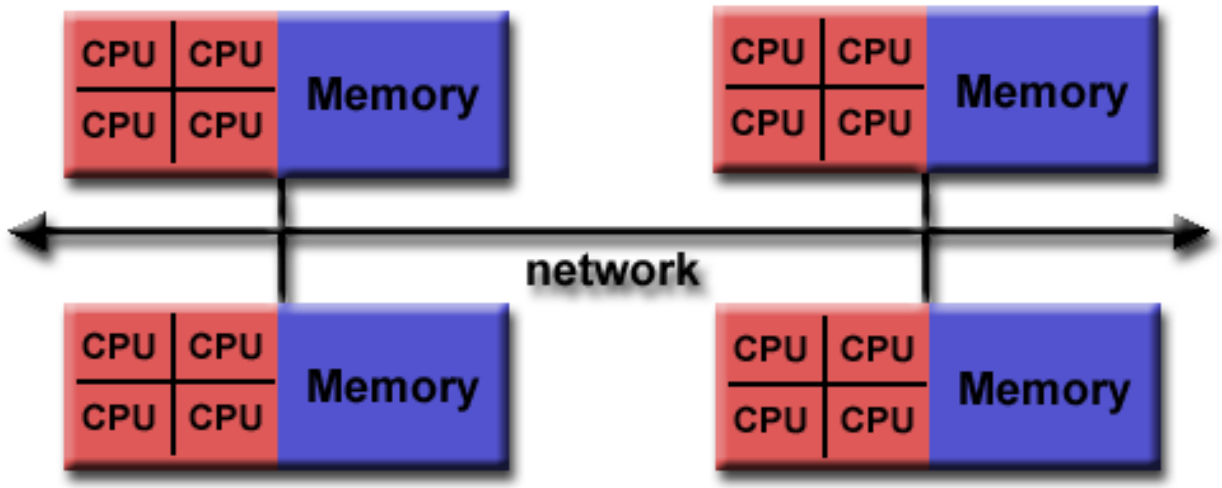
- General characteristics
  - Local memory
  - Communication network
- Advantages
  - Scalable memory
  - Rapid local memory access
  - Cost effective
- Disadvantages
  - Data communication
  - Data mapping
  - Non-uniform memory access times
  - Hard to program



Distributed Memory Access

Multi-processor Computer Memory Architectures – Hybrid Distributed-Shared Memory

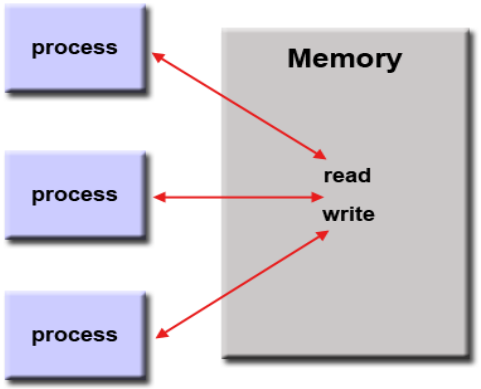
- Characteristics
  - Large and fast
  - The shared memory component can be a shared memory machine and/or graphics processing units
  - Network communication
- Advantages
  - Increased scalability
- Disadvantages
  - Increased programmer complexity



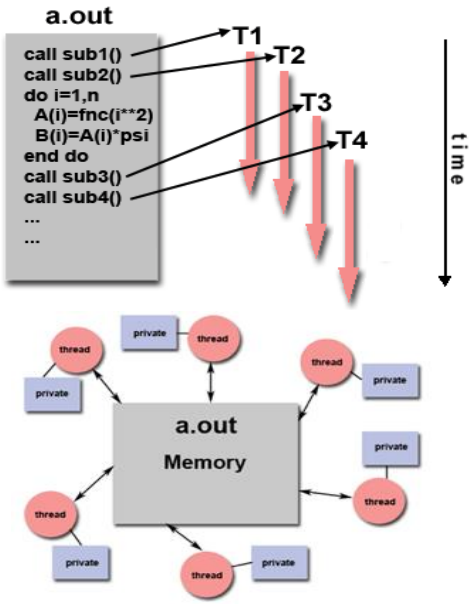
Hybrid Distributed-Shared Memory Access

Parallel Programming Models

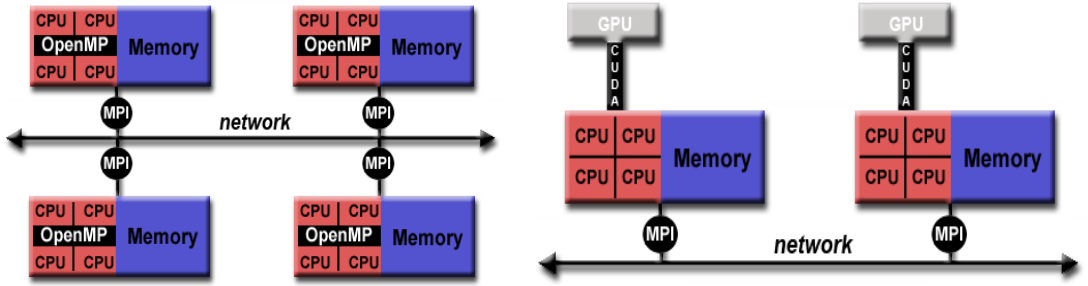
- An abstraction above hardware and memory architectures
  - Shared memory (without threads) model
  - Shared memory (with threads) model
    - POSIX Threads
    - OpenMP
    - CUDA threads for GPUs
  - Distributed memory / Message Passing model
    - MPI
  - Data parallel model (Partitioned global address space model)
  - Hybrid model



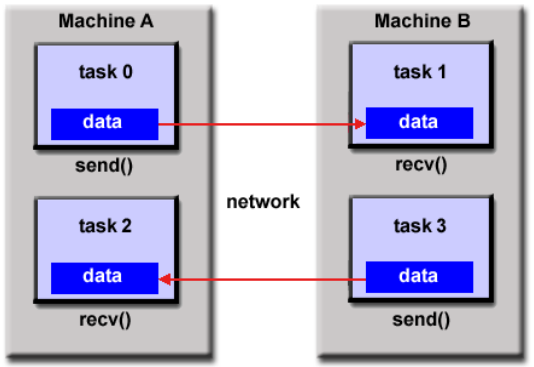
Shared Memory Model (without threads)



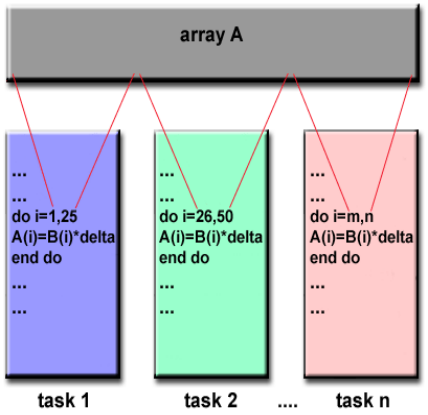
Shared Memory Model (with threads)



Hybrid models



MPI



Data parallel model

Types of Parallelism

- Data parallelism
  - Data parallelism arises when there are many data items that can be operated on at the same time.
  - Data parallelism focuses on distributing the data across multiple cores.
- Task-level parallelism
  - Task parallelism arises when there are many tasks or functions that can be operated independently and largely in parallel.
  - Task parallelism focuses on distributing functions across multiple cores.
- Bit-level parallelism
  - A form of parallel computing which is based on increasing processor word size.
- Instruction-level parallelism

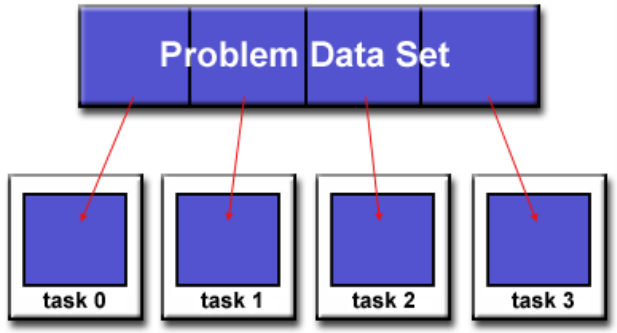
$$a \leftarrow b + c$$
$$d \leftarrow e * f$$

Designing Parallel Programs

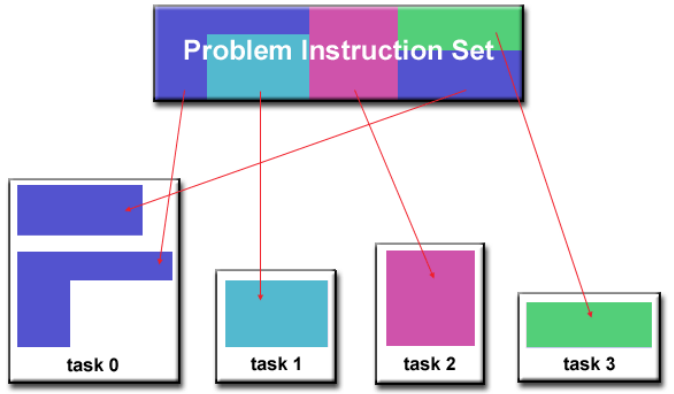
- Understand the problem and program!
  - Is it parallelizable?
    - Case 1: Calculate the potential energy for each of several thousand independent conformations of a molecule.
    - Case 2: Calculation of the Fibonacci series (0,1,1,2,3,5,8,13,21,...) by use of the formula: F(n) = F(n-1) + F(n-2)
  - What are the hotspots in the program?
  - What are the bottlenecks?
  - What are the inhibitors to parallelism?
    - Data dependence
  - Any optimized third party parallel software/math libraries available?

Partitioning

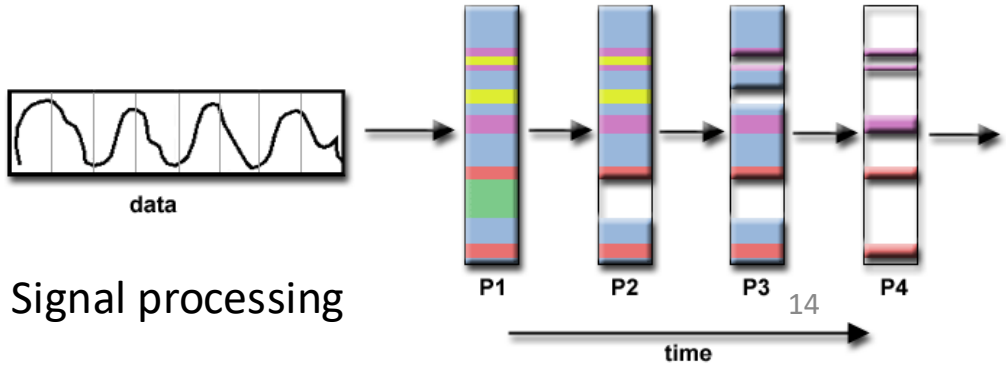
- Break the problem into discrete “chunks” of work that can be distributed to multiple tasks
  - Domain decomposition – data centered
    - Block: Each thread takes one potion, usually an equal potion of the data
    - Cyclic: Each thread takes more than one potion of the data
  - Functional decomposition – task centered
    - Functional decomposition lends itself well to problems that can be split into different tasks, e.g., signal processing



Domain decomposition



Functional decomposition

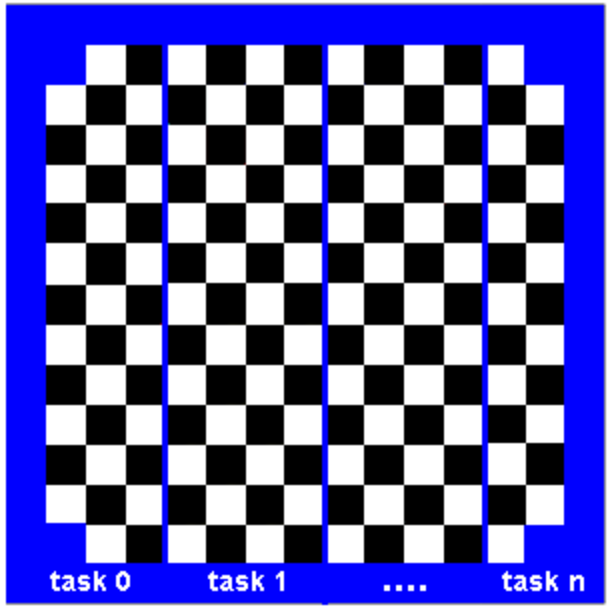


Signal processing

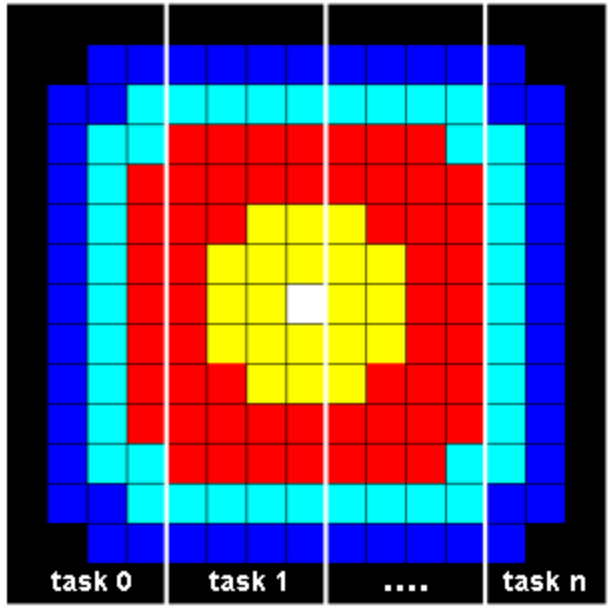
Communications

- Communication is to exchange data
- The need for communications between tasks depends upon your problem

Reverse color of pixels

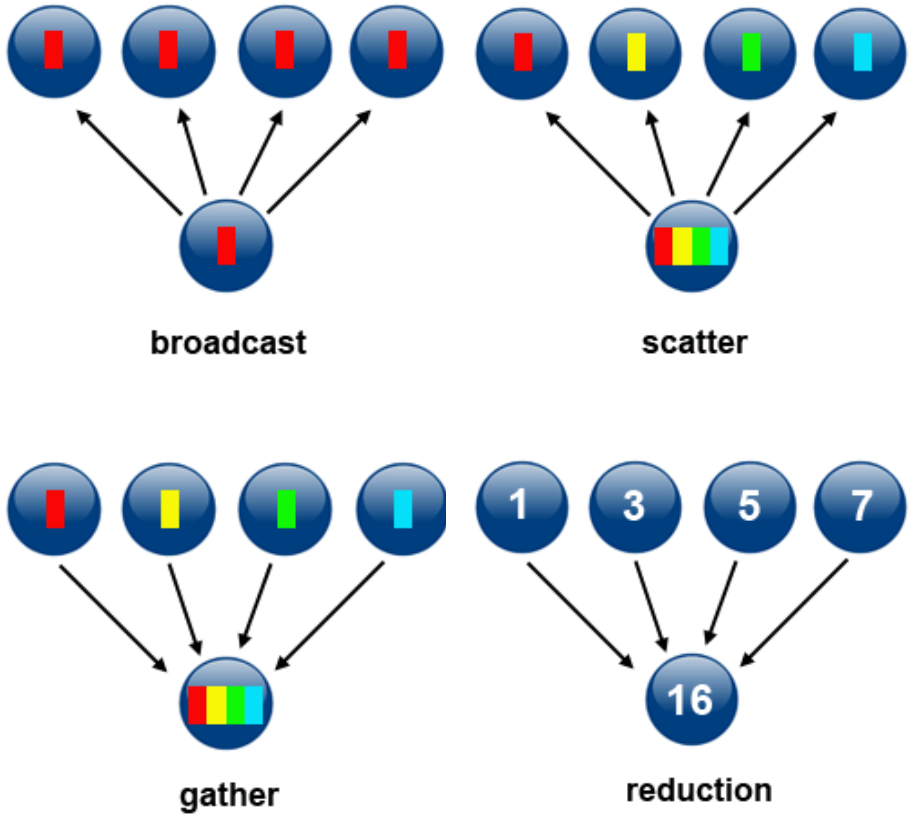


Heat diffusion problem



Communications (Cont.)

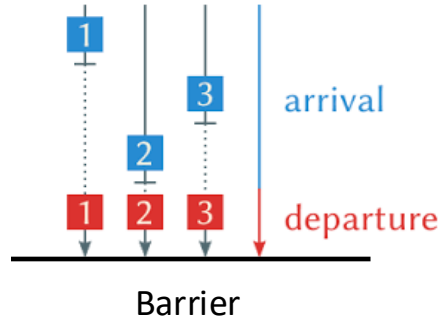
- Important factors
  - Communication overhead
  - Latency vs. Bandwidth
  - Visibility of communications
  - Synchronous vs. asynchronous communications
  - Scope of communications
    - Point-to-point
    - Collective: broadcast, scatter, gather, reduction

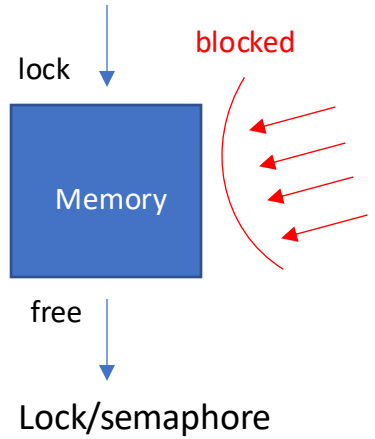


Collective communications

Synchronization

- Managing the sequence of work and the tasks performing it
- Types of synchronization
  - Barrier: each task performs its work until it reaches the barrier. It then stops, or "blocks"
  - Lock/semaphore: serialize (protect) access to global data or a section of code. Only one task at a time may use (own) the lock / semaphore / flag
  - Synchronous communication operations: When a task performs a communication operation, some form of coordination is required with the other task(s) participating in the communication





Data Dependencies

- A **dependence** exists between program statements when the order of statement execution affects the results of the program
- A **data dependence** results from multiple use of the same location(s) in storage by different tasks

Loop carried data dependence

```
DO J = MYSTART,MYEND
   A(J) = A(J-1) * 2.0
END DO
```

Loop independent data dependence

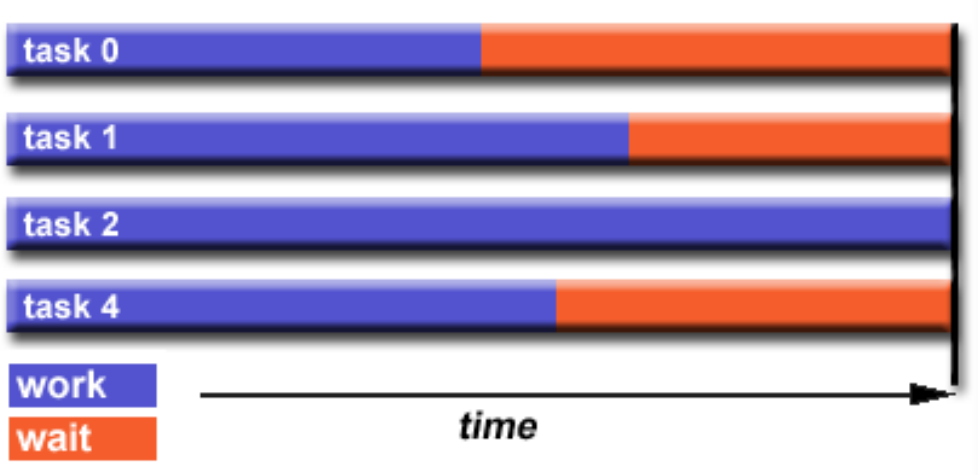
```
task 1          task 2
------          ------

X = 2           X = 4
  .               .
  .               .
Y = X**2        Y = X**3
```

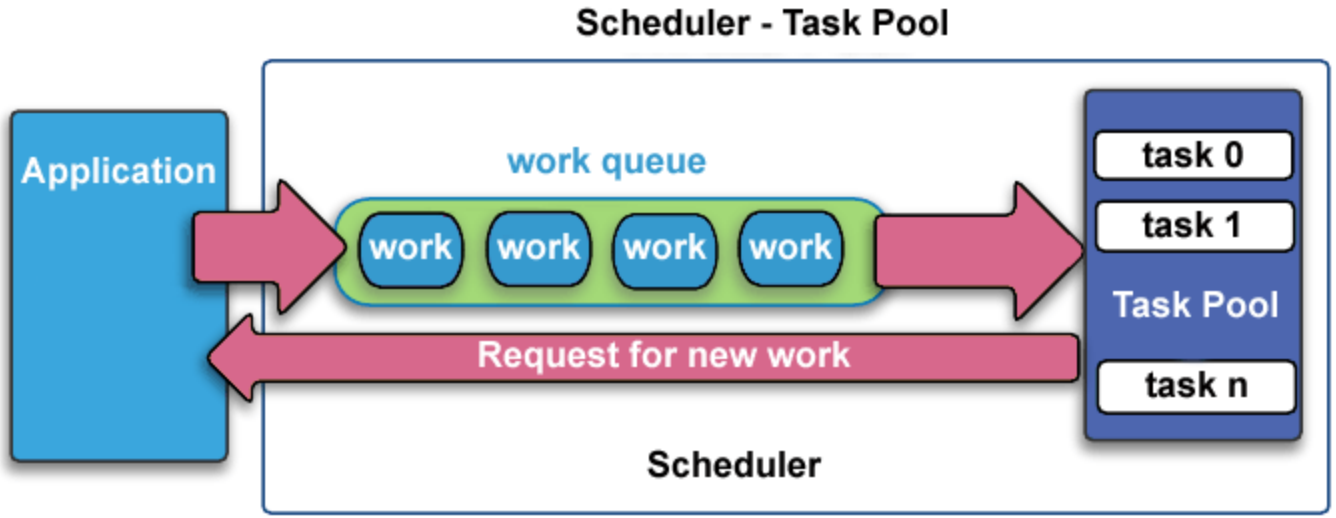
- How to handle data dependencies?
  - Distributed memory architecture – communicate required data at synchronization points
  - Shared memory architectures – synchronize read/write operations between tasks

Load Balancing

- **Load balancing** refers to the practice of distributing approximately equal amounts of work among tasks so that all tasks are kept busy all of the time. It can be considered a minimization of task idle time
- How to achieve load balance:
  - Equally partition the work each task receives
  - Dynamic work assignment

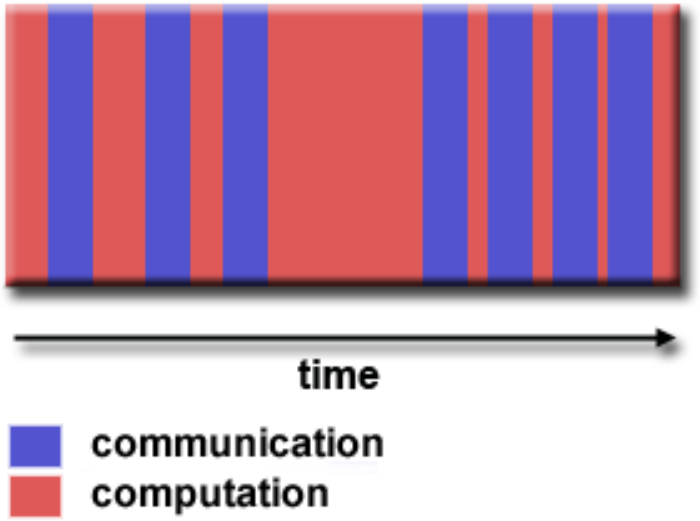


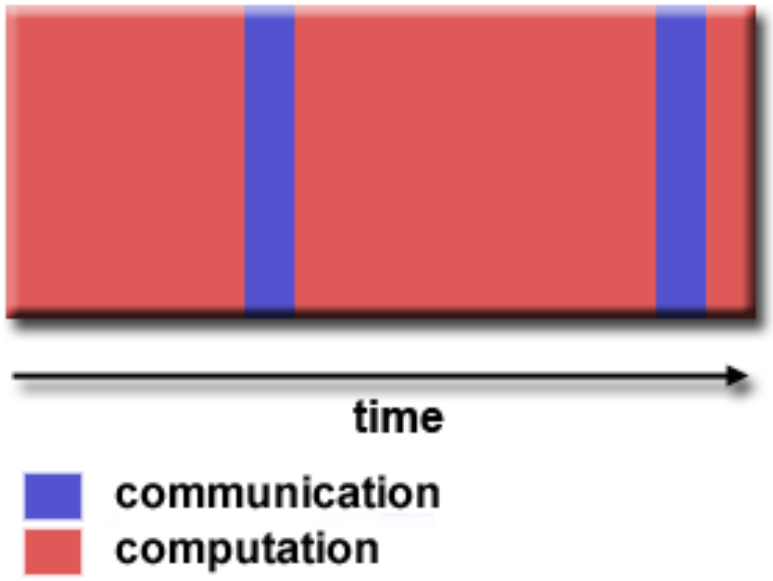
Slowest task determines overall performance



Granularity

- 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 vs. Coarse-grain parallelism





Input/Output

- I/O operations are generally regarded as inhibitors to parallelism
  - I/O operations require orders of magnitude more time than memory operations
  - Parallel I/O systems may be immature or not available for all platforms
  - In an environment where all tasks see the same file space, write operations can result in file overwriting
  - Read operations can be affected by the file server's ability to handle multiple read requests at the same time
  - I/O that must be conducted over the network (NFS, non-local) can cause severe bottlenecks and even crash file servers
- Tips:
  - Reduce overall I/O as much as possible
  - Use a parallel file system
  - Write large chunks of data
  - Fewer, larger files performs better
  - Aggregate I/O operations across tasks

Quantifying Parallelism

- Execution time
  - Wall time: the time elapsed between start and completion of the program
  - User time: the actual runtime used by the program
  - System time: the time used by the operating system
- Sequential algorithm is usually evaluated in terms of its execution time
- Parallel program is usually evaluated with respect to the sequential program
  - Speedup, efficiency, scalability

Speedup

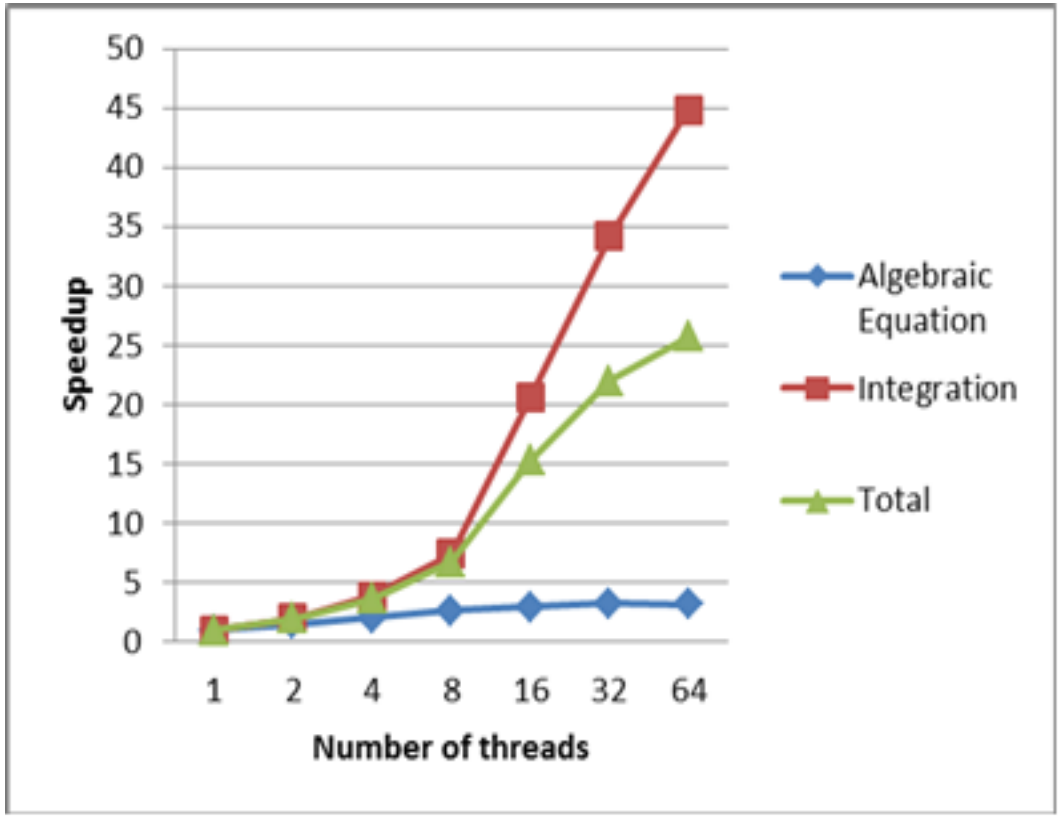
- A measure that captures the relative benefit of solving a problem in parallel
- Speedup $S = \frac{T_s}{T_p}$
  - Usually, $0 < S \leq p$
  - If $S = p$, linear speedup
  - Theoretically, S can never exceed p
  - In practice, it may happen that $S > p$: superlinear speedup

Efficiency

- A measure of processor utilization in a parallel program.
- Efficiency $E = \frac{S}{p}$
  - Usually, $0 < E \leq 1$
  - If $E = 1$, linear speedup
  - If $E > 1$: superlinear speedup since $S > p$
  - If $E < 1/p$: slowdown since $T_s < T_p$, the serial program is faster than the parallel program

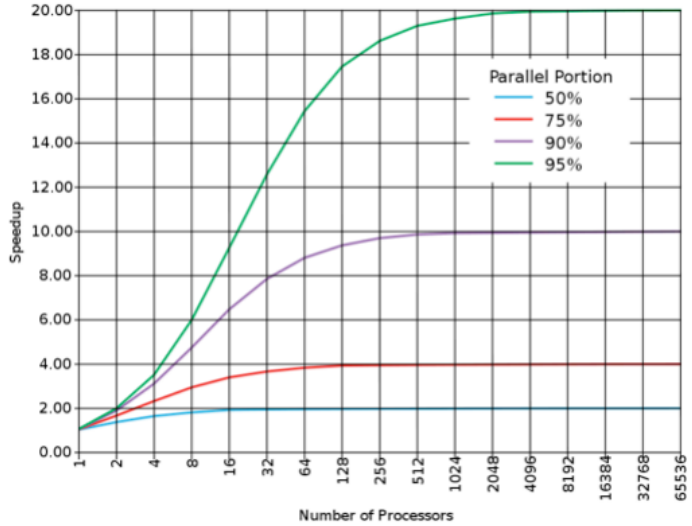
Scalability

- A measure of a parallel program's capacity to increase speedup in proportion to the number of processors
- Reflects a parallel program's ability to utilize increasing processing resource effectively

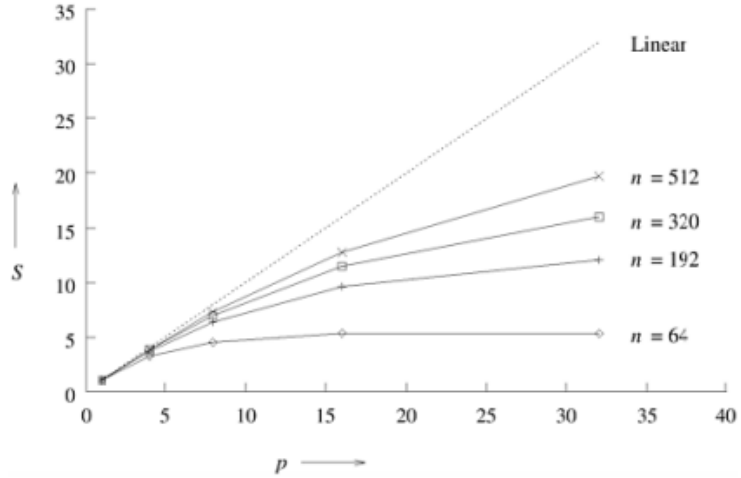


Factors Influencing the Performance

- Algorithm parallelizable
- Size of data set
- Synchronization and memory access overheads
- Load balancing



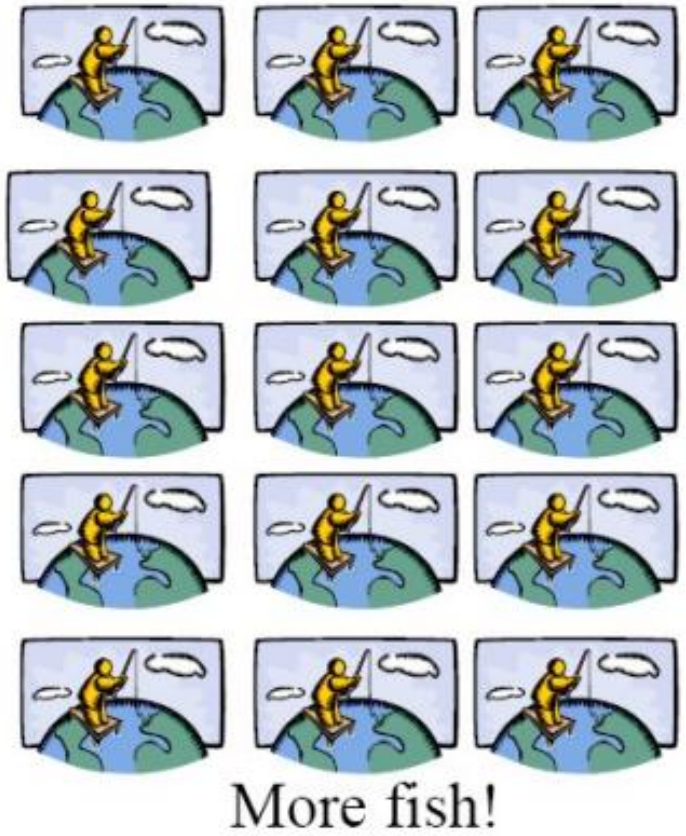
Algorithm effect



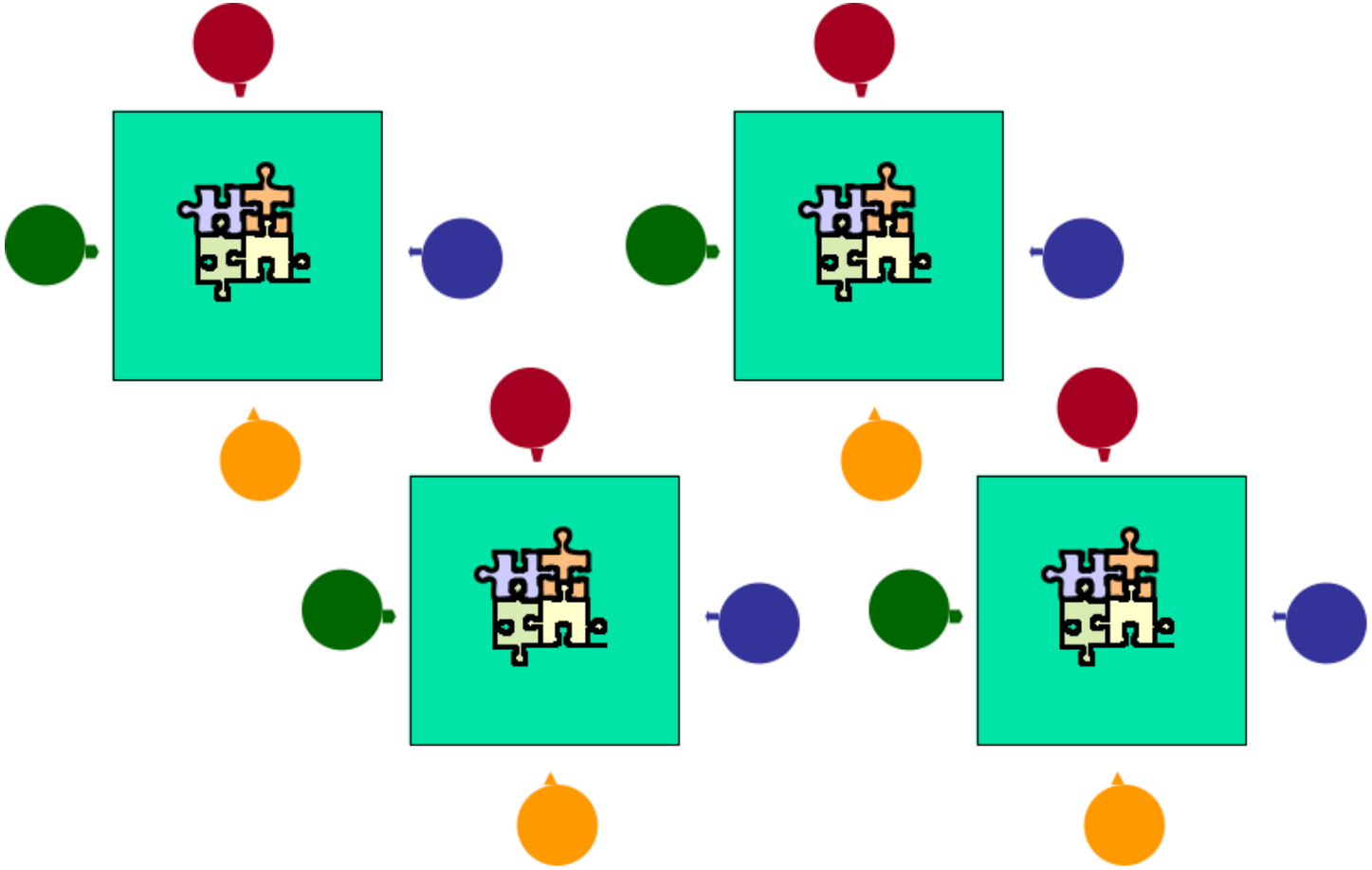
Data size effect

Recap with an Example

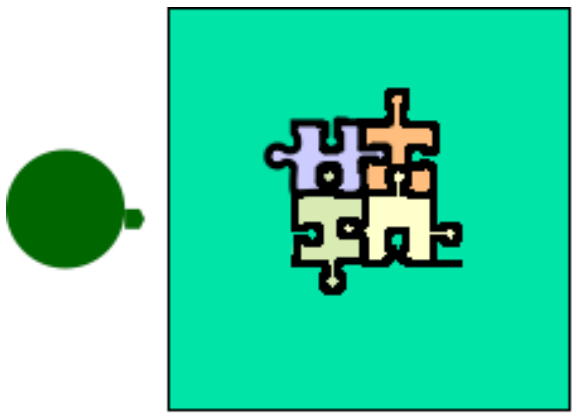
- Parallelism means doing multiple things at the same time; you can do more in less time



The Jigsaw Puzzle Analogy Example
-- Henry Neeman's "Supercomputing in Plain English An Introduction to High Performance Computing"



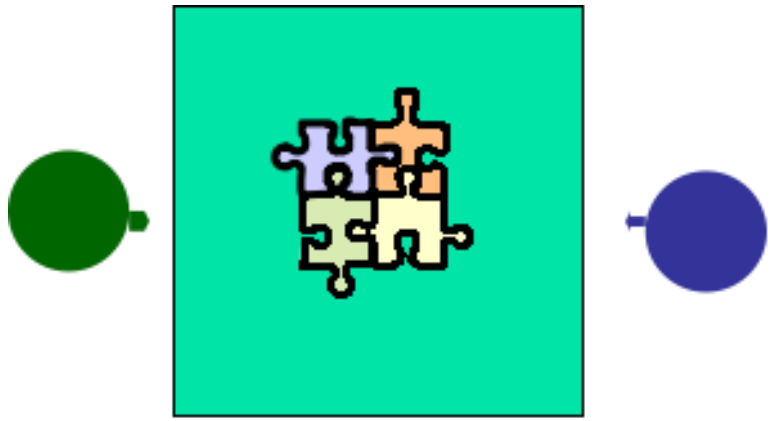
Serial Computing



Suppose you want to do a jigsaw puzzle that has, say, a thousand pieces.

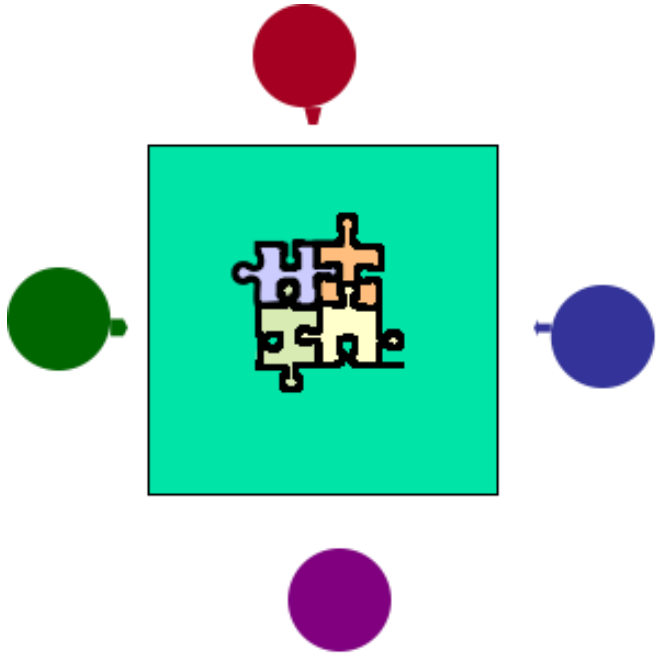
We can imagine that it'll take you a certain amount of time. Let's say that you can put the puzzle together in an hour.

Shared Memory Parallelism



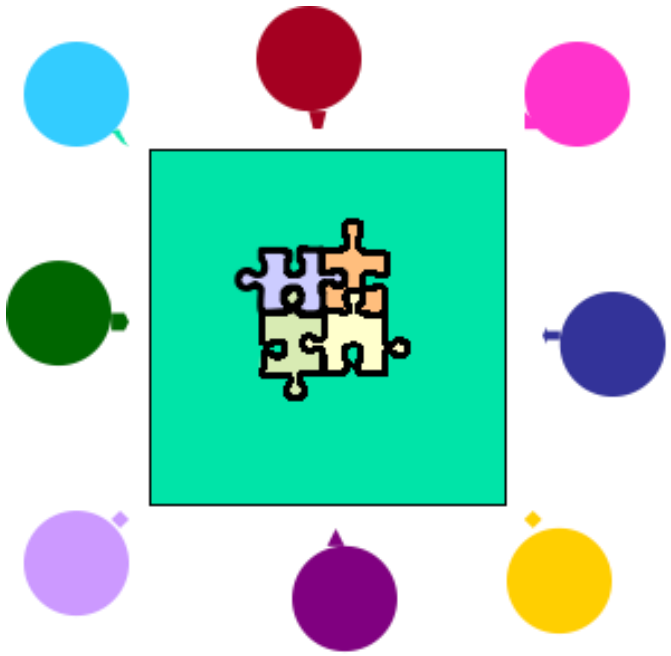
If Julie sits across the table from you, then she can work on her half of the puzzle and you can work on yours. Once in a while, you'll both reach into the pile of pieces at the same time (you'll contend for the same resource), which will cause a little bit of slowdown. And from time to time you'll have to work together (communicate) at the interface between her half and yours. The speedup will be nearly 2-to-1: y'all might take 35 minutes instead of 30.

The More the Merrier?



Now let's put Lloyd and Jerry on the other two sides of the table. Each of you can work on a part of the puzzle, but there'll be a lot more contention for the shared resource (the pile of puzzle pieces) and a lot more communication at the interfaces. So y'all will get noticeably less than a 4-to-1 speedup, but you'll still have an improvement, maybe something like 3-to-1: the four of you can get it done in 20 minutes instead of an hour.

Diminishing Returns



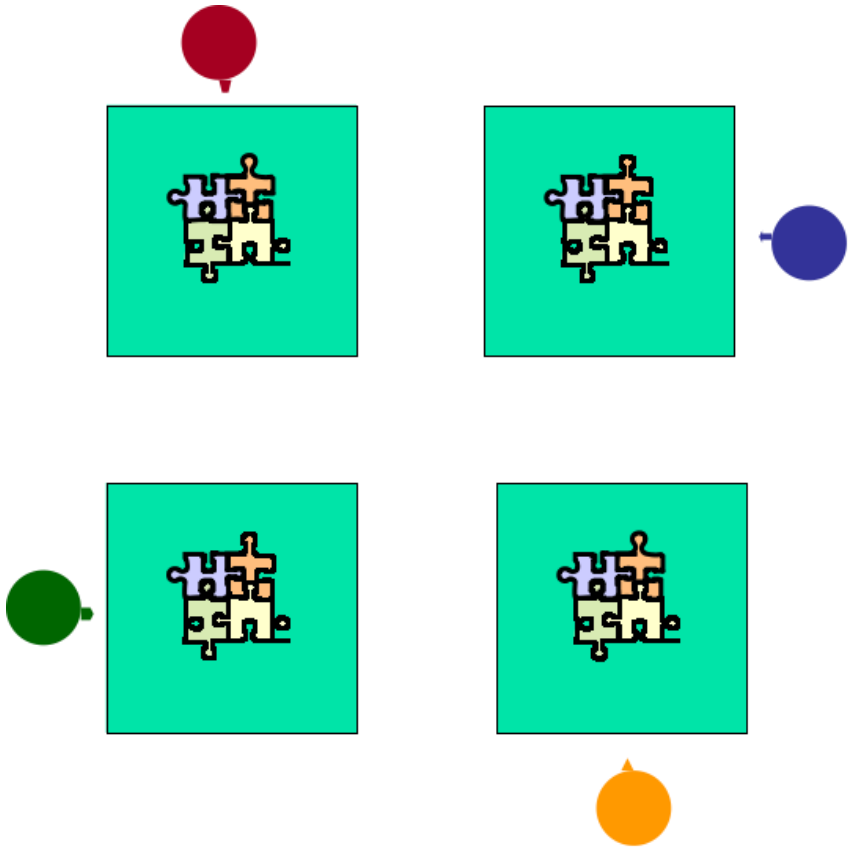
If we now put Cathy and Denese and Chenmei and Nilesh on the corners of the table, there's going to be a whole lot of contention for the shared resource, and a lot of communication at the many interfaces. So the speedup y'all get will be much less than we'd like; you'll be lucky to get 5-to-1.

So we can see that adding more and more workers onto a shared resource is eventually going to have a diminishing return.

Distributed Parallelism

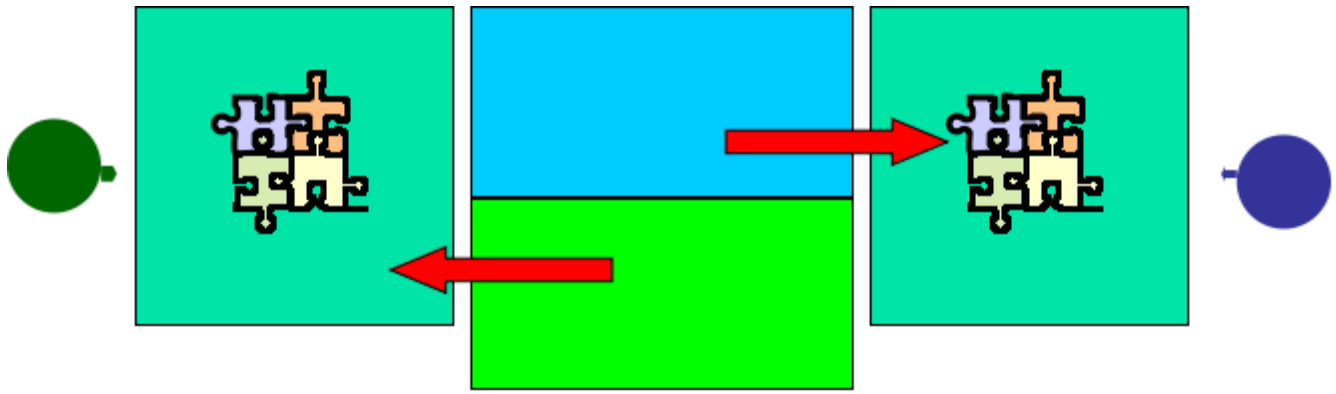
Now let's try something a little different. Let's set up two tables, and let's put you at one of them and Julie at the other. Let's put half of the puzzle pieces on your table and the other half of the pieces on Julie's. Now y'all can work completely independently, without any contention for a shared resource. **BUT**, the cost of communicating is **MUCH** higher (you have to scootch your tables together), and you need the ability to split up (decompose) the puzzle pieces reasonably evenly, which may be tricky to do for some puzzles.

More Distributed Processors



It's a lot easier to add more processors in distributed parallelism. But, you always have to be aware of the need to decompose the problem and to communicate between the processors. Also, as you add more processors, it may be harder to load balance the amount of work that each processor gets.

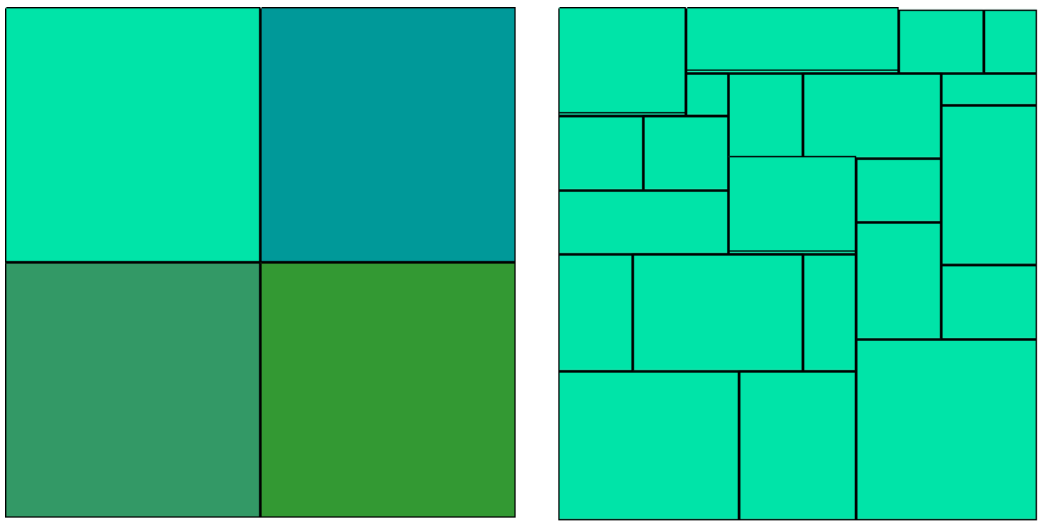
Load Balancing



Load balancing means giving everyone roughly the same amount of work to do.

For example, if the jigsaw puzzle is half grass and half sky, then you can do the grass and Julie can do the sky, and then y'all only have to communicate at the horizon – and the amount of work that each of you does on your own is roughly equal. So you'll get pretty good speedup.

Load Balancing



Load balancing can be easy, if the problem splits up into chunks of roughly equal size, with one chunk per processor. Or load balancing can be very hard.

Palmetto Cluster

https://docs.rcd.clemson.edu/palmetto/