RESEARCH COMPUTING
Harvard University
Faculty of Arts and Sciences

INTRODUCTION TO PARALLEL COMPUTING

Plamen Krastev

Office: 38 Oxford, Room 117
Email: plamenkrastev@fas.harvard.edu

FAS Research Computing
Harvard University

OBJECTIVES:

To introduce you to the basic concepts and ideas in parallel computing

To familiarize you with the major programming models in parallel computing

To provide you with with guidance for designing efficient parallel programs

OUTLINE:

- Introduction to Parallel Computing / High Performance Computing (HPC)
- Concepts and terminology
- Parallel programming models
- Parallelizing your programs
- Parallel examples

What is High Performance Computing?

Pravetz 82 and 8M, Bulgarian Apple clones
Image credit: flickr

What is High Performance Computing?

Pravetz 82 and 8M, Bulgarian Apple clones
Image credit: flickr

What is High Performance Computing?

Odyssey supercomputer is the major computational resource of FAS RC:

- 2,140 nodes / 60,000 cores
- 14 petabytes of storage

What is High Performance Computing?

Odyssey supercomputer is the major computational resource of FAS RC:

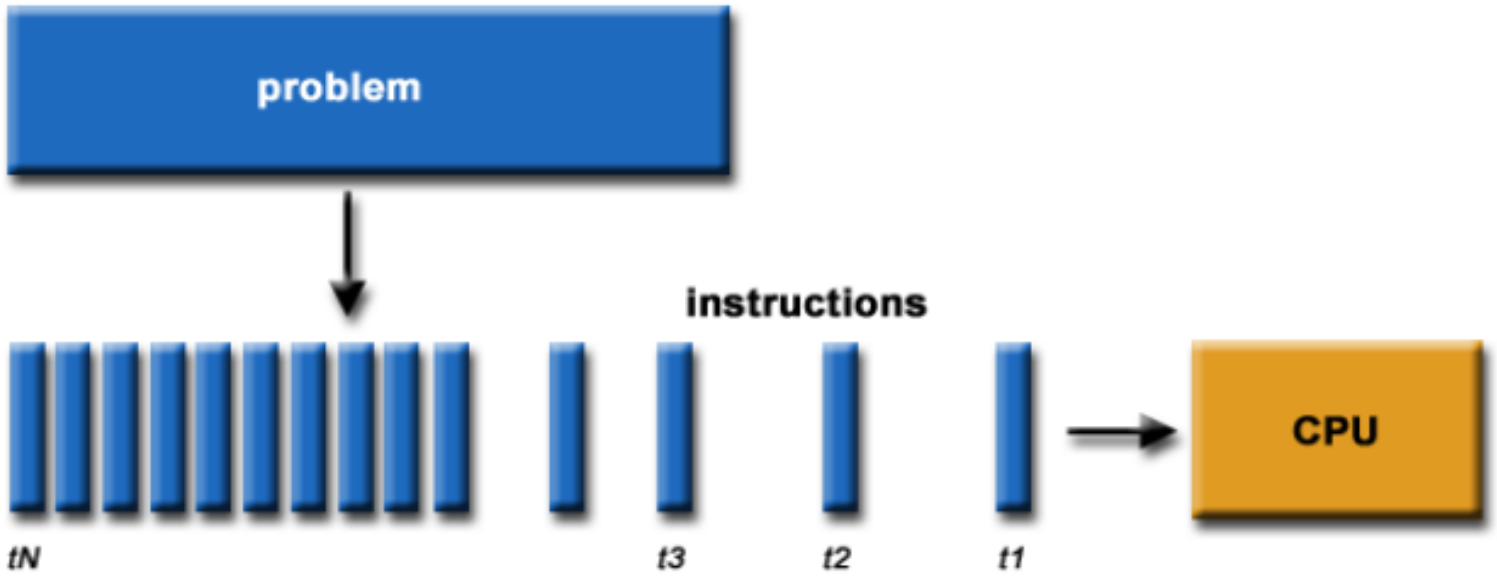
- 2,140 nodes / 60,000 cores
- 14 petabytes of storage

Using the world's fastest and largest computers to solve large and complex problems.

Serial Computation:

Traditionally software has been written for serial computations:

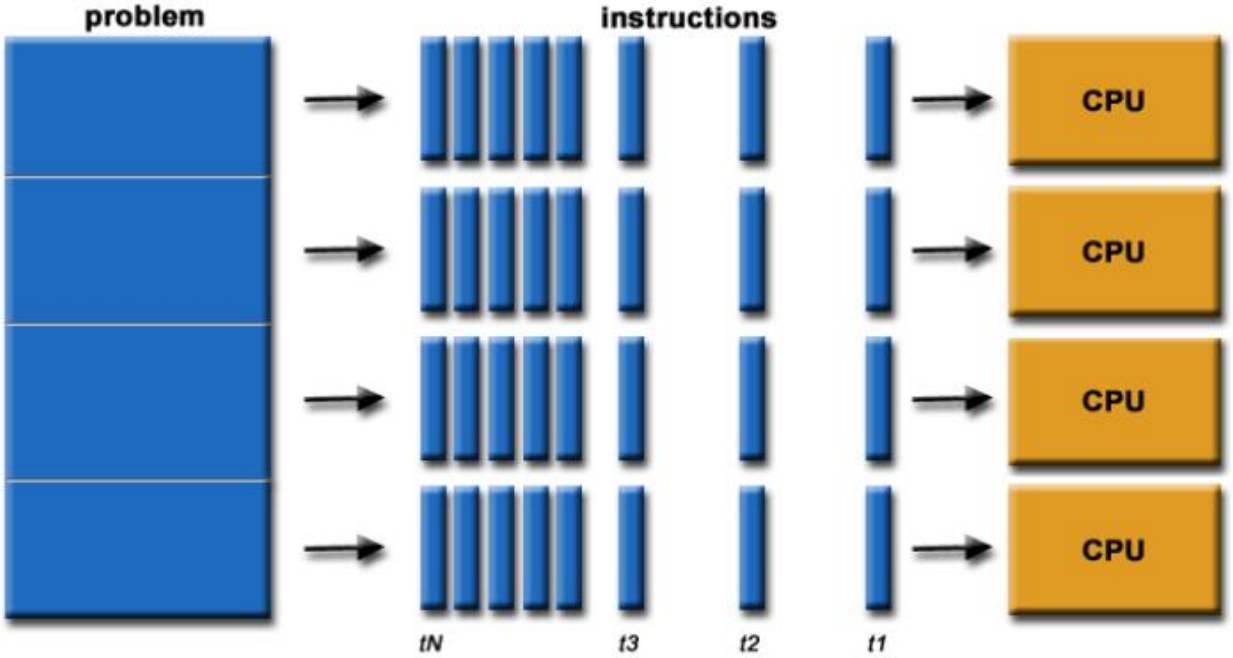
- To be run on a single computer having a single Central Processing Unit (CPU)
- A problem is broken into a discrete set of instructions
- Instructions are executed one after another
- Only one instruction can be executed at any moment in time



Parallel Computing:

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

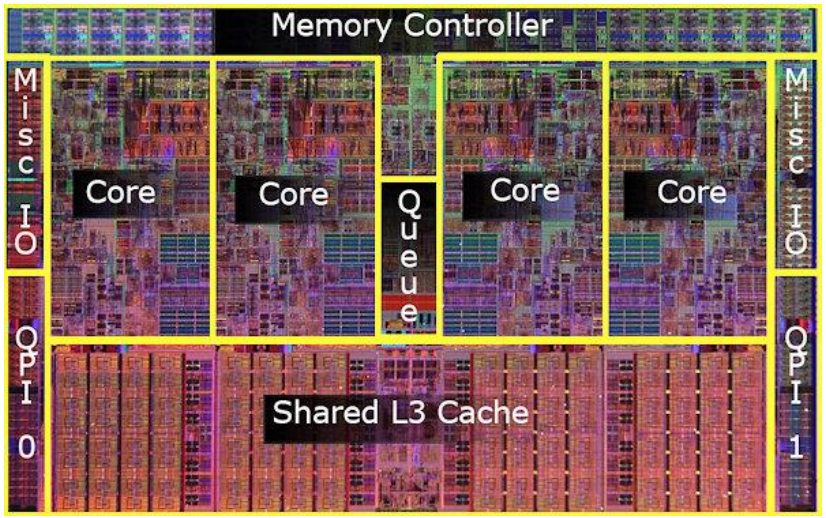
- To be run using multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs



Parallel Computers:

Virtually all stand-alone computers today are parallel from a hardware perspective:

- ❑ Multiple functional units (floating point, integer, GPU, etc.)
- ❑ Multiple execution units / cores
- ❑ Multiple hardware threads



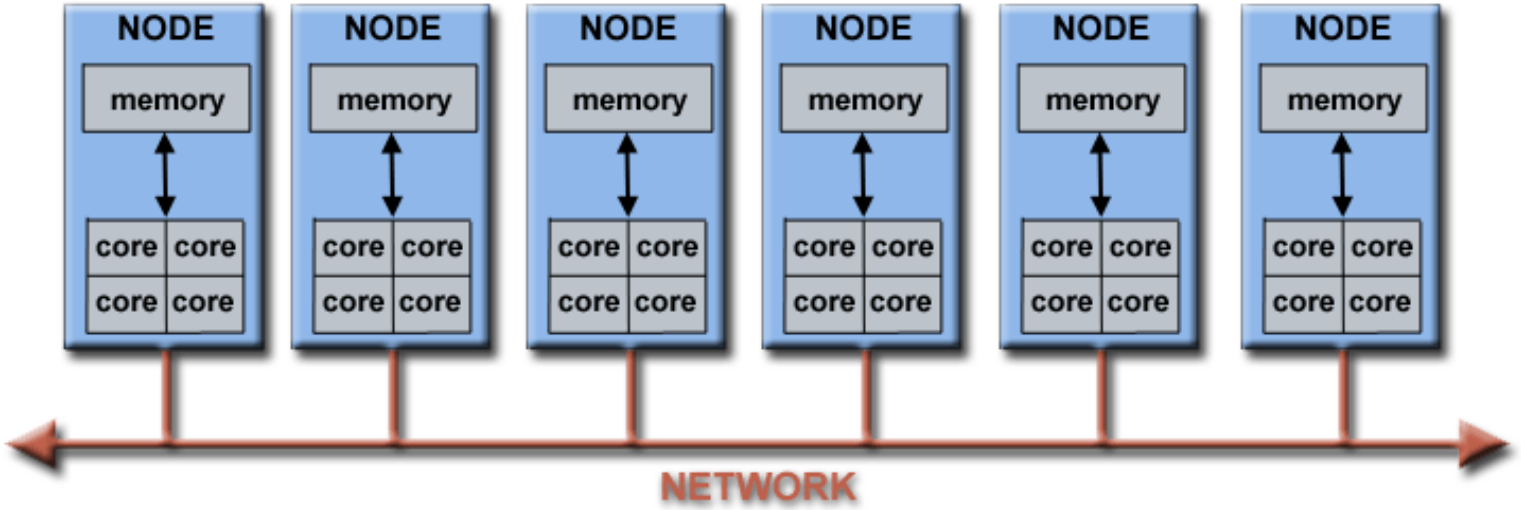
Intel Core i7 CPU and its major components

Image Credit: Intel

Parallel Computers:

Networks connect multiple stand-alone computers (nodes) to create larger parallel computer clusters

- Each compute node is a multi-processor parallel computer in itself
- Multiple compute nodes are networked together with an InfiniBand network
- Special purpose nodes, also multi-processor, are used for other purposes



Why Use HPC?

Major reasons:

Save time and/or money: In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel clusters can be built from cheap, commodity components.

Why Use HPC?

Major reasons:

Save time and/or money: In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel clusters can be built from cheap, commodity components.

Solve larger problems: Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.

Why Use HPC?

Major reasons:

Save time and/or money: In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel clusters can be built from cheap, commodity components.

Solve larger problems: Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.

Provide concurrency: A single compute resource can only do one thing at a time. Multiple computing resources can be doing many things simultaneously.

Why Use HPC?

Major reasons:

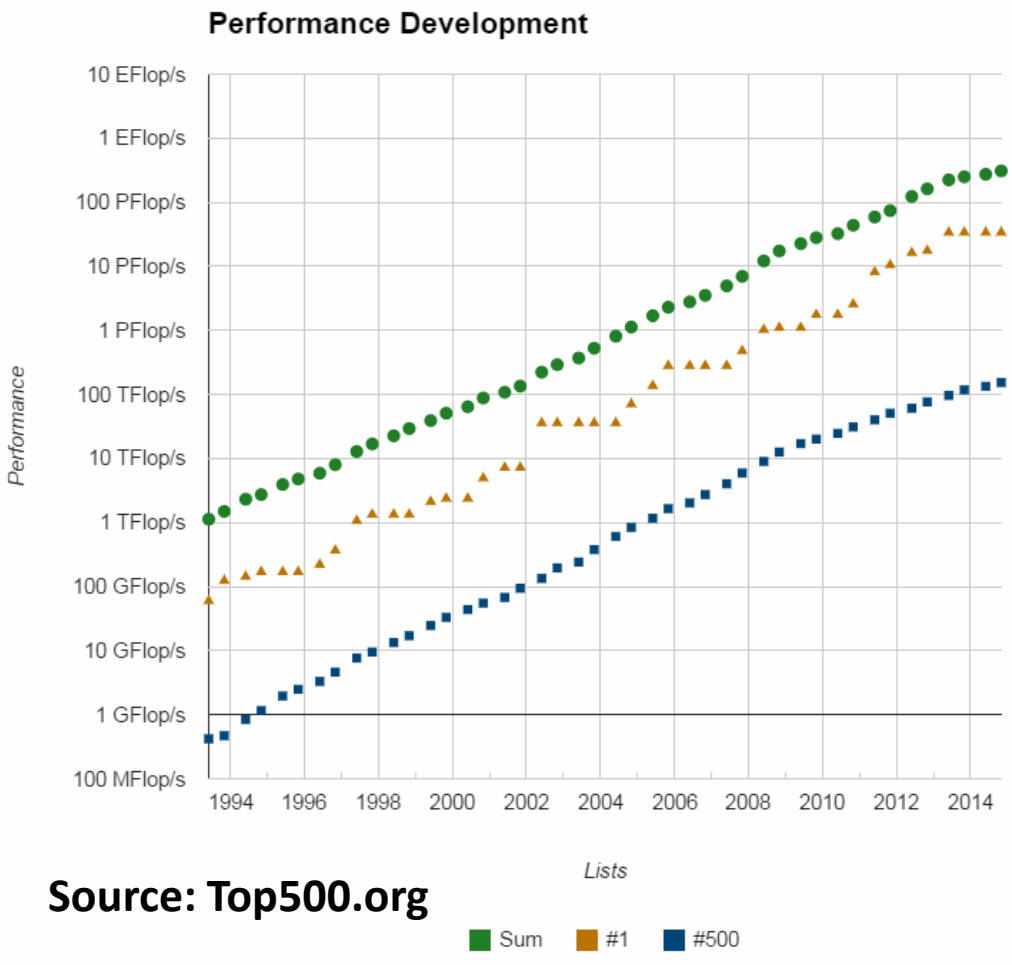
Save time and/or money: In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel clusters can be built from cheap, commodity components.

Solve larger problems: Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.

Provide concurrency: A single compute resource can only do one thing at a time. Multiple computing resources can be doing many things simultaneously.

Use of non-local resources: Using compute resources on a wide area network, or even the Internet when local compute resources are scarce.

Future Trends:

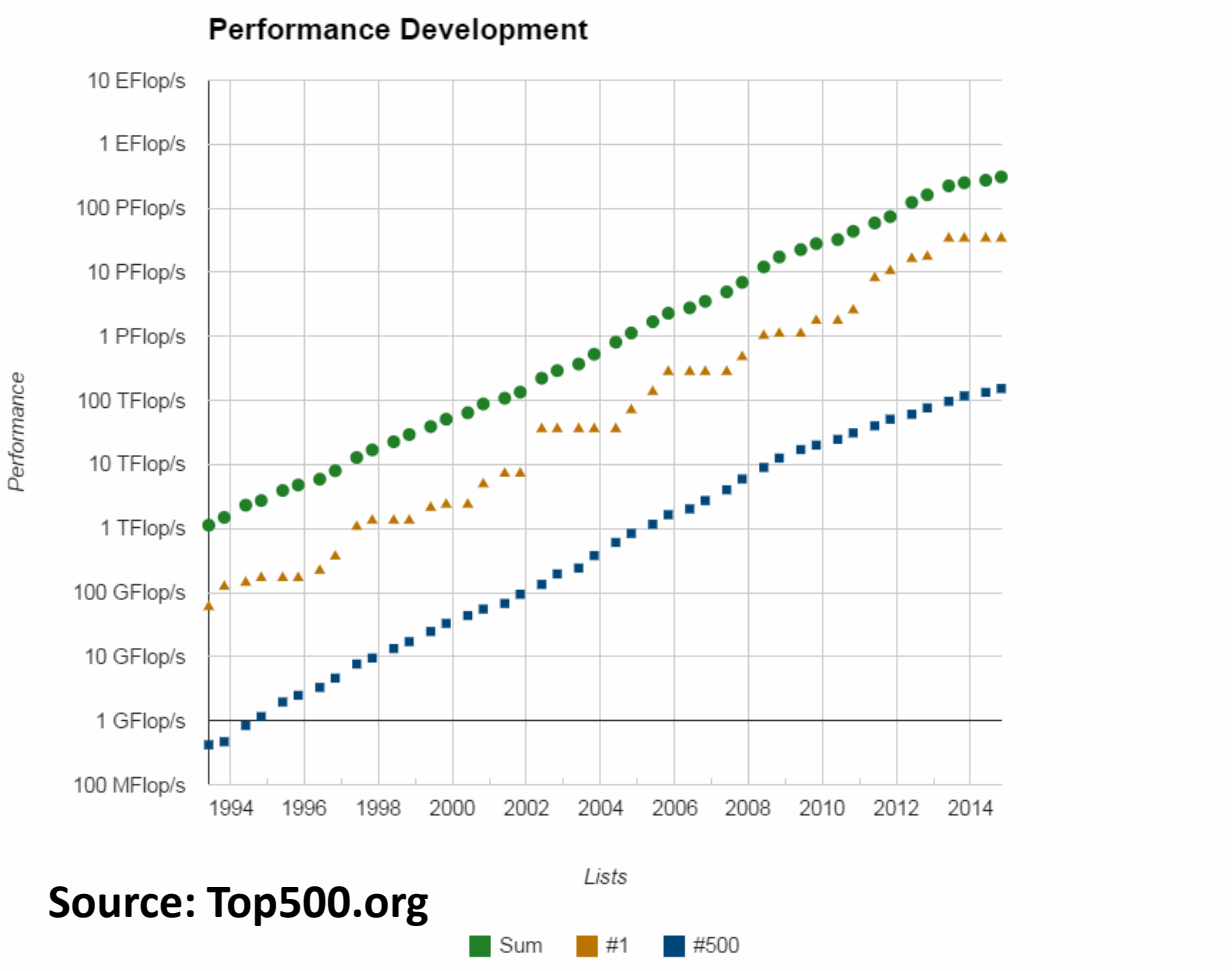


Source: Top500.org

Computer performance

Name	FLOPS
yottaFLOPS	10^{24}
zettaFLOPS	10^{21}
exaFLOPS	10^{18}
petaFLOPS	10^{15}
teraFLOPS	10^{12}
gigaFLOPS	10^{9}
megaFLOPS	10^{6}
kiloFLOPS	10^{3}

Future Trends:



Source: Top500.org

Computer performance

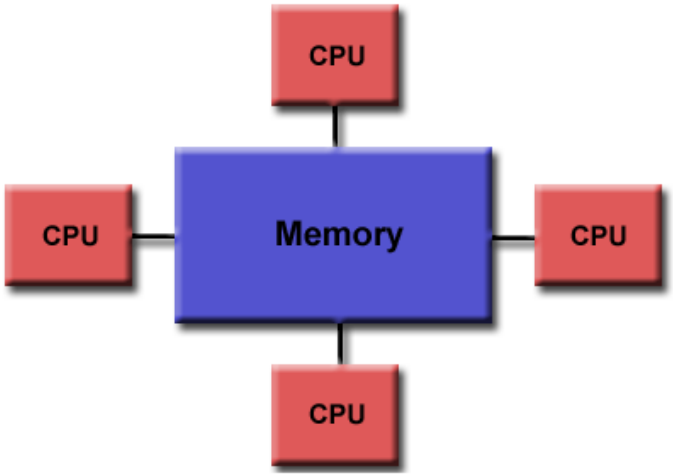
Name	FLOPS
yottaFLOPS	10^{24}
zettaFLOPS	10^{21}
exaFLOPS	10^{18}
petaFLOPS	10^{15}
teraFLOPS	10^{12}
gigaFLOPS	10^{9}
megaFLOPS	10^{6}
kiloFLOPS	10^{3}

The race is already on for Exascale Computing!

HPC Terminology:

- **Supercomputing / High-Performance Computing (HPC)**
- **Flop(s)** – Floating point operation(s)
- **Node** – a stand alone computer
- **CPU / Core** – a modern CPU usually has several cores (individual processing units)
- **Task** – a logically discrete section from the computational work
- **Communication** – data exchange between parallel tasks
- **Speedup** – time of serial execution / time of parallel execution
- **Massively Parallel** – refer to hardware of parallel systems with many processors (“many” = hundreds of thousands)
- **Pleasantly Parallel** – solving many similar but independent tasks simultaneously. Requires very little communication
- **Scalability** - a proportionate increase in parallel speedup with the addition of more processors

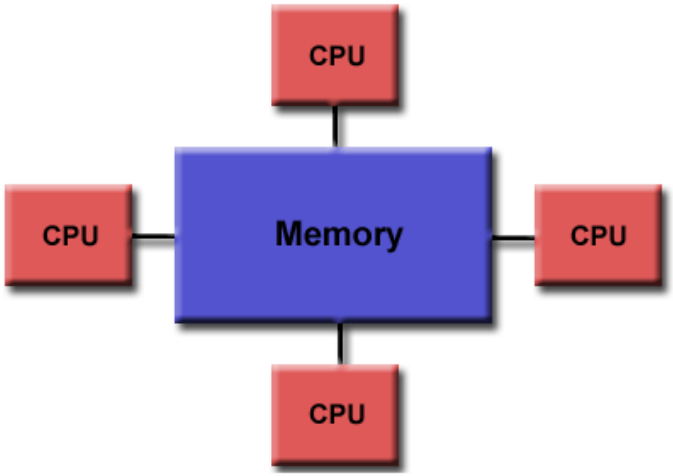
Parallel Computer Memory Architectures:



Shared Memory:

- Multiple processors can operate independently, but share the same memory resources
- Changes in a memory location caused by one CPU are visible to all processors

Parallel Computer Memory Architectures:



Shared Memory:

- Multiple processors can operate independently, but share the same memory resources
- Changes in a memory location caused by one CPU are visible to all processors

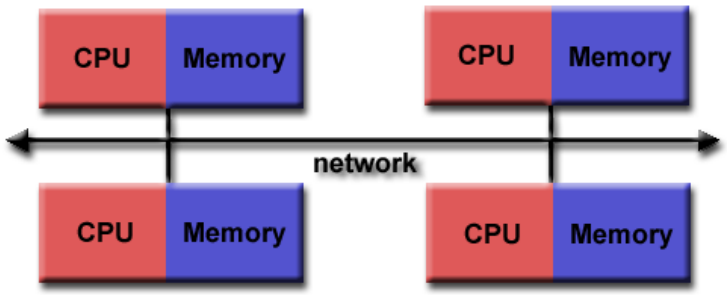
Advantages:

- Global address space provides a user-friendly programming perspective to memory
- Fast and uniform data sharing due to proximity of memory to CPUs

Disadvantages:

- Lack of scalability between memory and CPUs. Adding more CPUs increases traffic on the shared memory-CPU path
- Programmer responsibility for "correct" access to global memory

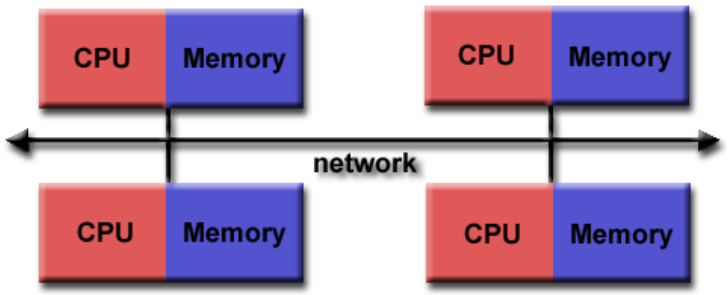
Parallel Computer Memory Architectures:



Distributed Memory:

- Requires a communication network to connect inter-processor memory
- Processors have their own local memory. Changes made by one CPU have no effect on others
- Requires communication to exchange data among processors

Parallel Computer Memory Architectures:



Distributed Memory:

- Requires a communication network to connect inter-processor memory
- Processors have their own local memory. Changes made by one CPU have no effect on others
- Requires communication to exchange data among processors

Advantages:

- Memory is scalable with the number of CPUs
- Each CPU can rapidly access its own memory without overhead incurred with trying to maintain global cache coherency

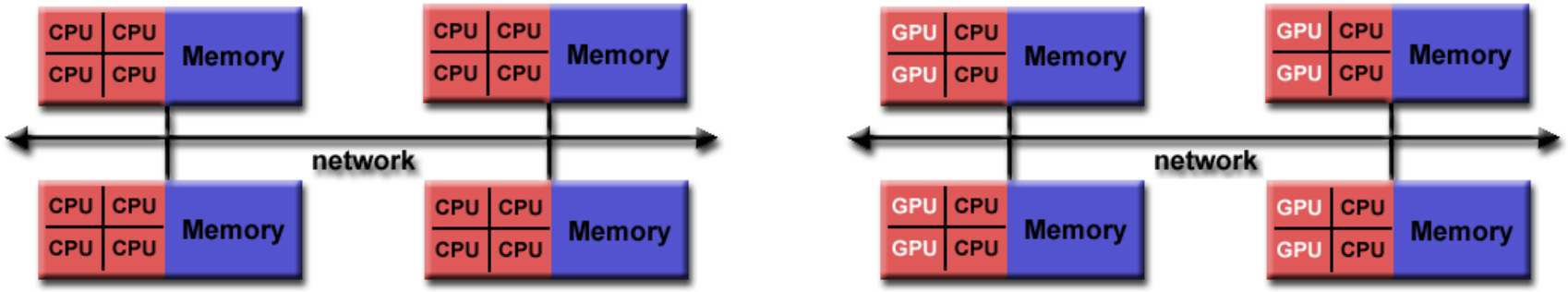
Disadvantages:

- Programmer is responsible for many of the details associated with data communication between processors
- It is usually difficult to map existing data structures to this memory organization, based on global memory

Parallel Computer Memory Architectures:

Hybrid Distributed-Shared Memory:

The largest and fastest computers in the world today employ both shared and distributed memory architectures.

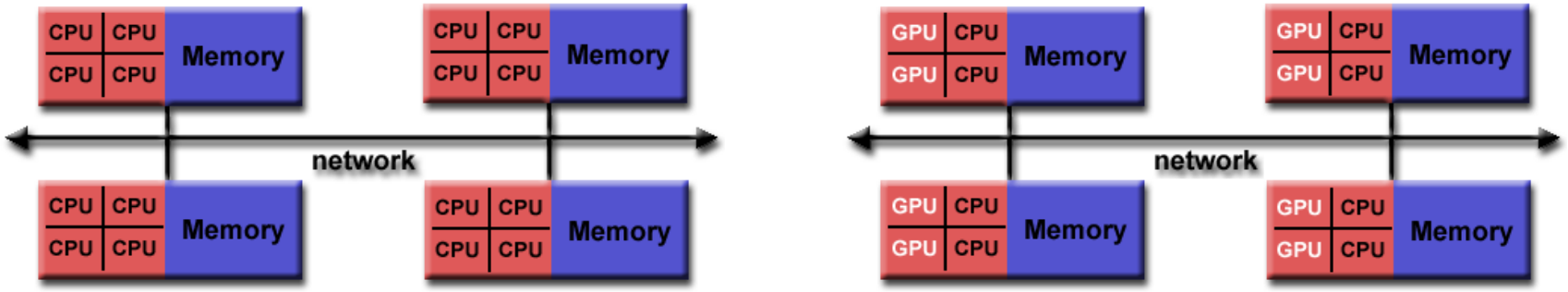


- Shared memory component can be a shared memory machine and/or GPU
- Processors on a compute node share same memory space
- Requires communication to exchange data between compute nodes

Parallel Computer Memory Architectures:

Hybrid Distributed-Shared Memory:

The largest and fastest computers in the world today employ both shared and distributed memory architectures.



- Shared memory component can be a shared memory machine and/or GPU
- Processors on a compute node share same memory space
- Requires communication to exchange data between compute nodes

Advantages and Disadvantages:

- Whatever is common to both shared and distributed memory architectures
- Increased scalability is an important advantage
- Increased programming complexity is a major disadvantage

Parallel Programming Models:

Parallel Programming Models exist as an abstraction above hardware and memory architectures

- Shared Memory (without threads)
- Shared Threads Models (Pthreads, OpenMP)
- Distributed Memory / Message Passing (MPI)
- Data Parallel
- Hybrid
- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)

Shared Threads Models:

POSIX Threads

- Library based; requires parallel coding
- C Language only; Interfaces for Perl, Python and others exist
- Commonly referred to as Pthreads
- Most hardware vendors now offer Pthreads in addition to their proprietary threads implementations
- Very explicit parallelism; requires significant programmer attention to detail

OpenMP

- Compiler directive based; can use serial code
- Jointly defined and endorsed by a group of major computer hardware and software vendors
- Portable / multi-platform, including Unix and Windows platforms
- Available in C/C++ and Fortran implementations
- Can be very easy and simple to use - provides for "incremental parallelism"

Distributed Memory / Message Passing Models:

- A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or 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 Interface (MPI) is the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. MPI implementations exist for virtually all popular parallel computing platforms

Data Parallel Model:

- May also referred to as the Partitioned Global Address Space (PGAS) model
- It displays these characteristics:
  - Address space is treated globally
  - Parallel work focuses on performing operations on a data set
  - Tasks work on different portions from the same data structure
  - Tasks perform the same operation

Data Parallel Model:

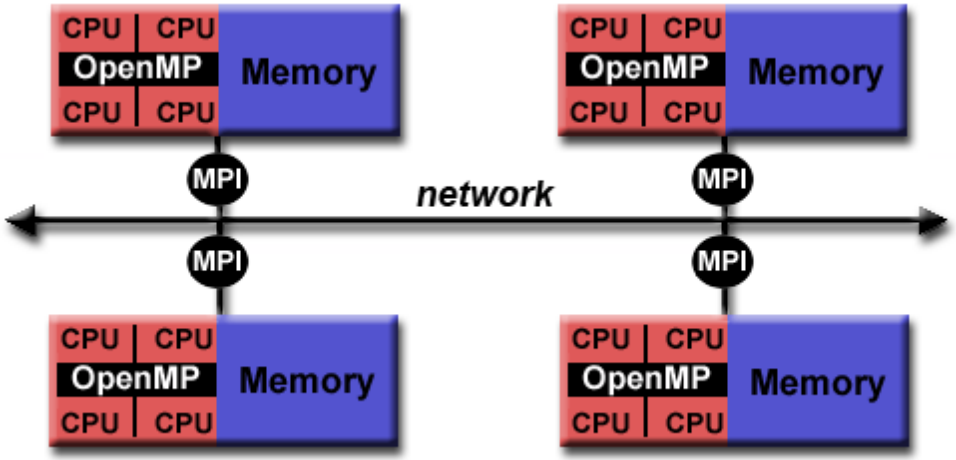
- May also referred to as the Partitioned Global Address Space (PGAS) model
- It displays these characteristics:
  - Address space is treated globally
  - Parallel work focuses on performing operations on a data set
  - Tasks work on different portions from the same data structure
  - Tasks perform the same operation

Example Implementations:

- **Coarray Fortran:** A small set of extension to Fortran 95. Compiler dependent

- **Unified Parallel C (UPC):** An extension to the C programming language. Compiler dependent

Hybrid Parallel Programming Models:

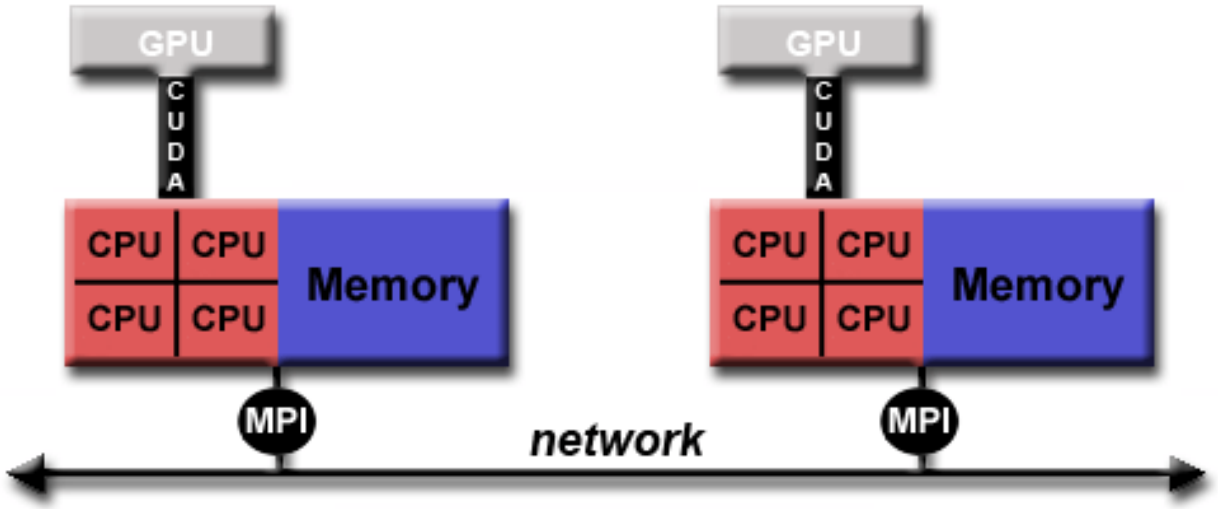


Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with the threads model (OpenMP)

- Threads perform computationally intensive kernels using local, on-node data
- Communications between processes on different nodes occurs over the network using MPI

This hybrid model lends itself well to the increasingly common hardware environment of clustered multi/many-core machines

Hybrid Parallel Programming Models:



Another similar and increasingly popular example of a hybrid model is using MPI with GPU (Graphics Processing Unit) programming

- GPUs perform computationally intensive kernels using local, on-node data
- Communications between processes on different nodes occurs over the network using MPI

Languages using parallel computing:

- C/C++
- Fortran
- MATLAB
- Python
- R
- Perl
- Julia
- And others

Can my code be parallelized?

- Does it have large loops that repeat the same operations?
- Does your code do multiple tasks that are not dependent on one another? If so is the dependency weak?
- Can any dependencies or information sharing be overlapped with computation? If not, is the amount of communications small?
- Do multiple tasks depend on the same data?
- Does the order of operations matter? If so how strict does it have to be?

Basic guidance for efficient parallelization:

- Is it even worth parallelizing my code?
  - Does your code take an intractably long amount of time to complete?
  - Do you run a single large model or do statistics on multiple small runs?
  - Would the amount of time it take to parallelize your code be worth the gain in speed?
- Parallelizing established code vs. starting from scratch
  - Established code: Maybe easier / faster to parallelize, but my not give good performance or scaling
  - Start from scratch: Takes longer, but will give better performance, accuracy, and gives the opportunity to turn a “black box” into a code you understand

Basic guidance for efficient parallelization:

- Increase the fraction of your program that can be parallelized. Identify the most time consuming parts of your program and parallelize them. This could require modifying your intrinsic algorithm and code's organization
- Balance parallel workload
- Minimize time spent in communication
- Use simple arrays instead of user defined derived types
- Partition data. Distribute arrays and matrices – allocate specific memory for each MPI process

Considerations about parallelization:

You parallelize your program to run faster, and to solve larger and more complex problems.

How much faster will the program run?

Speedup:

$$S(n) = \frac{T(1)}{T(n)}$$

Time to complete the job on **one** process

Time to complete the job on **n** process

Efficiency:

$$E(n) = \frac{S(n)}{n}$$

Tells you how efficiently you parallelize your code

Oversimplified example:

p → fraction of program that can be parallelized

1 - p → fraction of program that cannot be parallelized

n → number of processors

Then the time of running the parallel program will be

1 – p + p/n of the time for running the serial program

80% can be parallelized

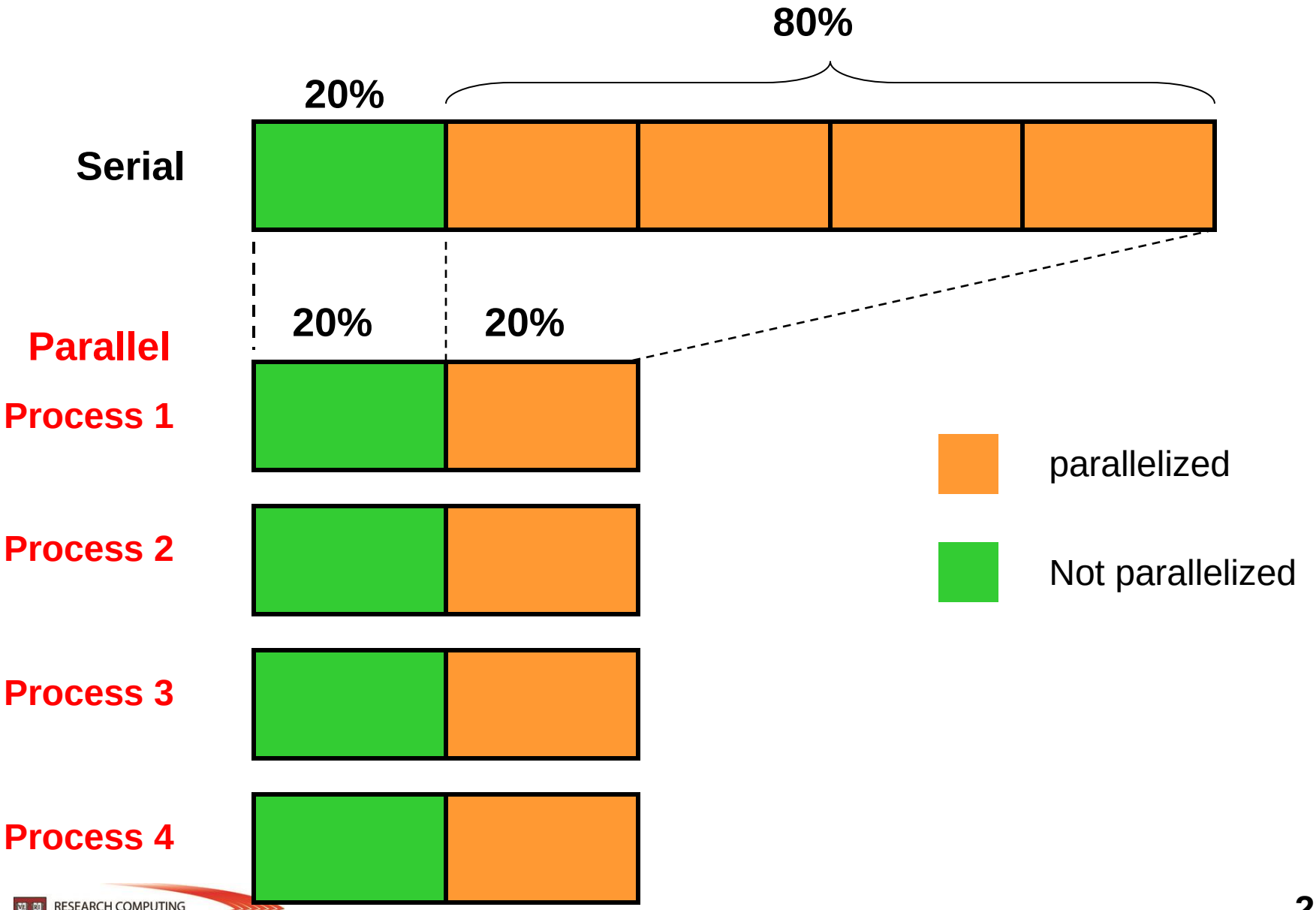
20 % cannot be parallelized

n = 4

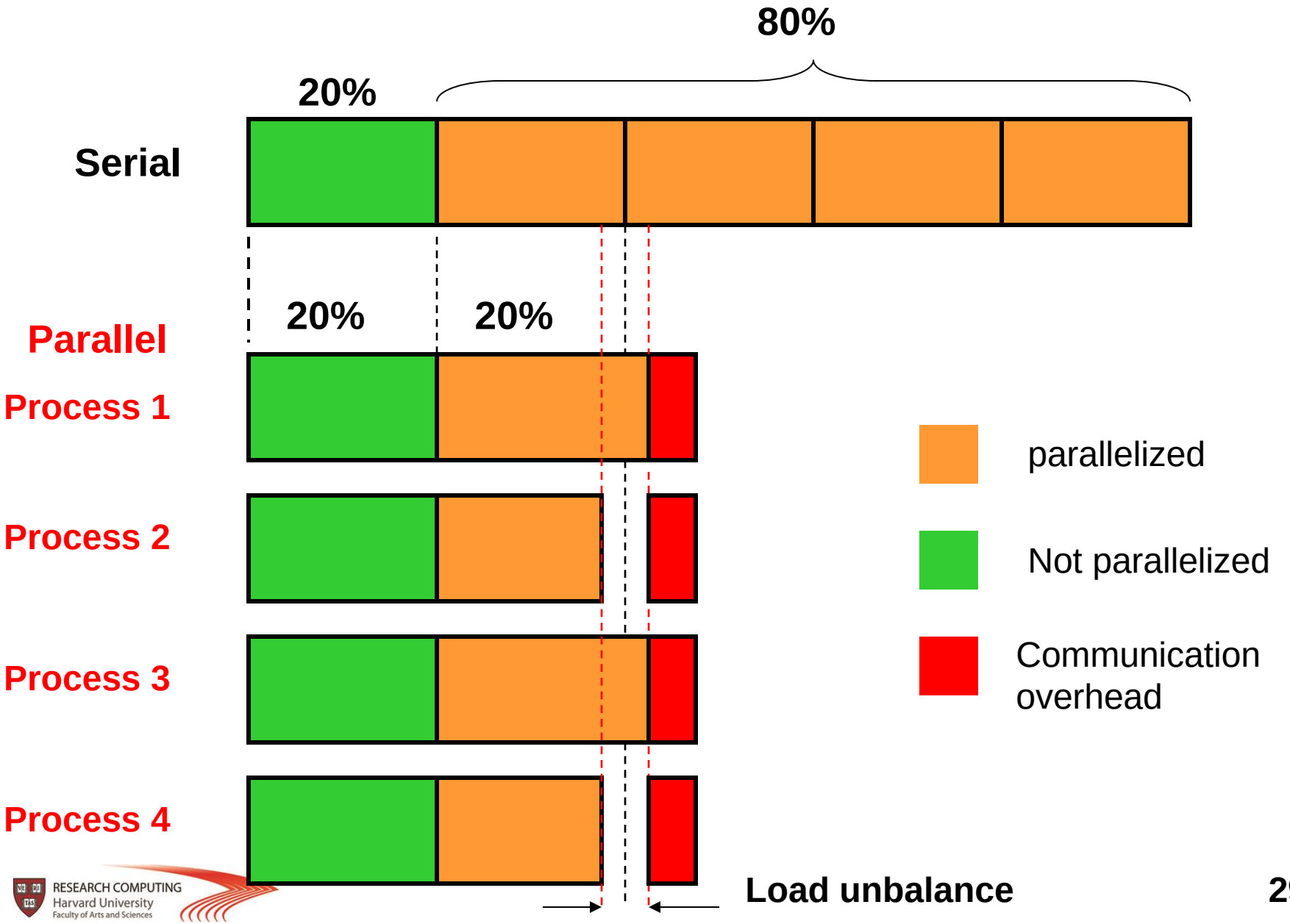
1 - 0.8 + 0.8 / 4 = 0.4 i.e., 40% of the time for running the serial code

You get 2.5 speed up although you run on 4 cores since only 80% of your code can be parallelized (assuming that all parts in the code can complete in equal time)

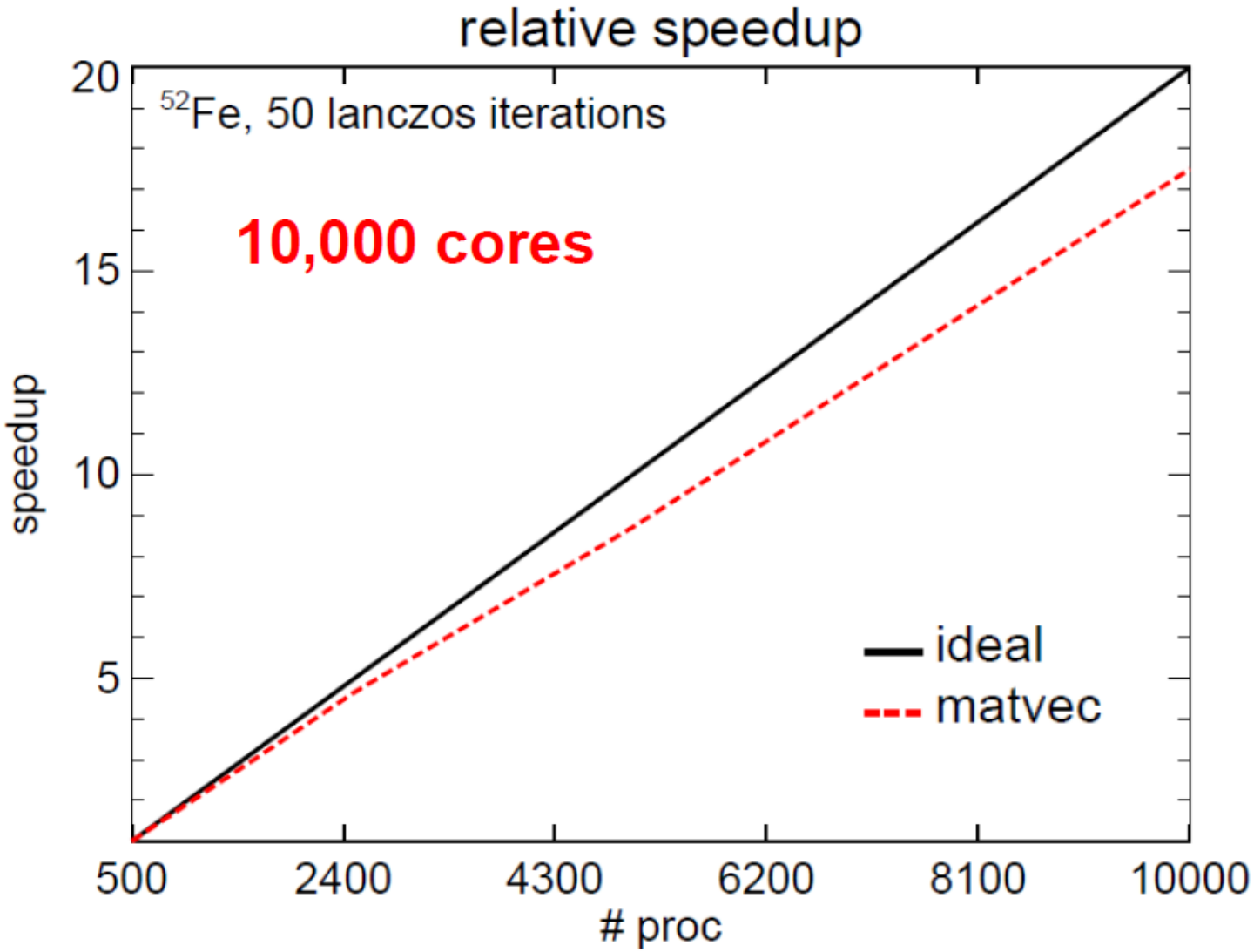
Oversimplified example, cont'd:



More realistic example:



Realistic example: Speedup of matrix vector multiplication in large scale shell-model calculations



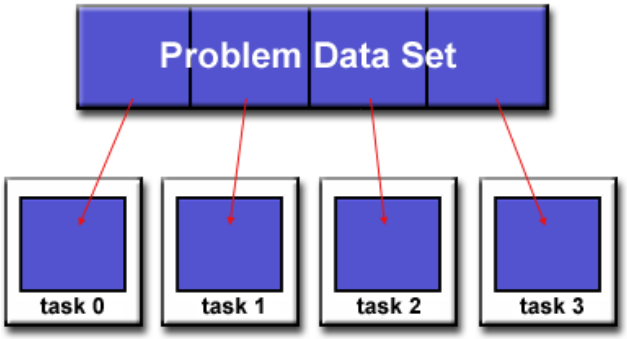
Designing parallel programs - partitioning:

One of the first steps in designing a parallel program is to break the problem into discrete "chunks" that can be distributed to multiple parallel tasks.

Designing parallel programs - partitioning:

One of the first steps in designing a parallel program is to break the problem into discrete "chunks" that can be distributed to multiple parallel tasks.

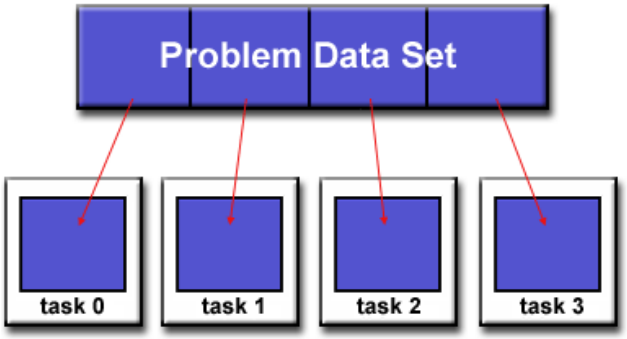
Domain Decomposition:
Data associate with a problem is partitioned – each parallel task works on a portion of the data

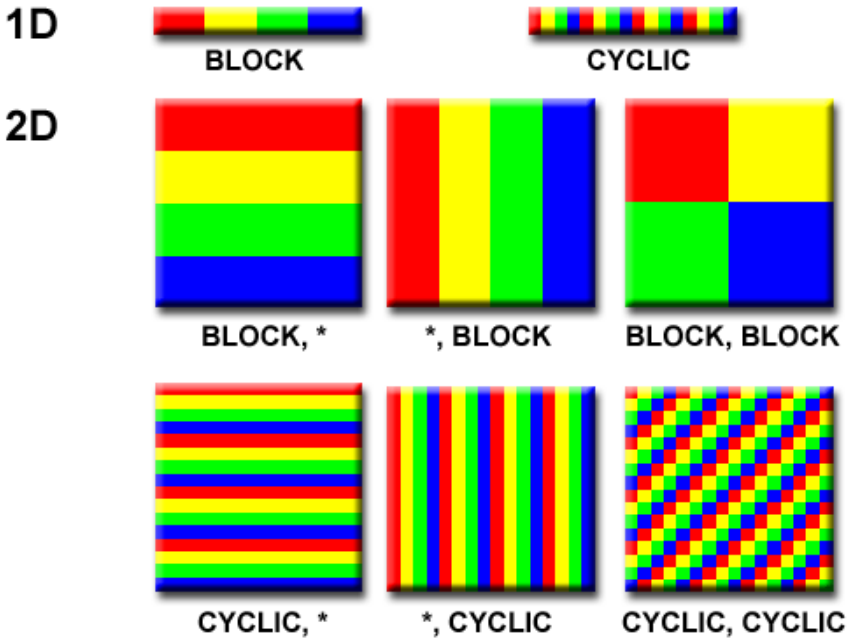


Designing parallel programs - partitioning:

One of the first steps in designing a parallel program is to break the problem into discrete "chunks" that can be distributed to multiple parallel tasks.

Domain Decomposition:
Data associate with a problem is partitioned – each parallel task works on a portion of the data



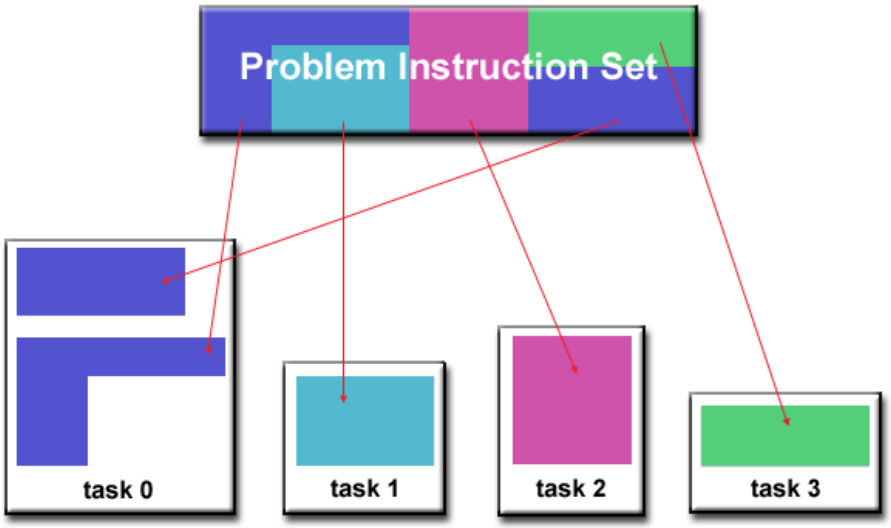


There are different ways to partition the data

Designing parallel programs - partitioning:

One of the first steps in designing a parallel program is to break the problem into discrete "chunks" that can be distributed to multiple parallel tasks.

Functional Decomposition:
Problem is decomposed according to the work that must be done. Each parallel task performs a fraction of the total computation.



Designing parallel programs - communication:

Most parallel applications require tasks to share data with each other.

Designing parallel programs - communication:

Most parallel applications require tasks to share data with each other.

Cost of communication: Computational resources are used to package and transmit data. Requires frequently synchronization – some tasks will wait instead of doing work. Could saturate network bandwidth.

Designing parallel programs - communication:

Most parallel applications require tasks to share data with each other.

Cost of communication: Computational resources are used to package and transmit data. Requires frequently synchronization – some tasks will wait instead of doing work. Could saturate network bandwidth.

Latency vs. Bandwidth: Latency is the time it takes to send a minimal message between two tasks. Bandwidth is the amount of data that can be communicated per unit of time. Sending many small messages can cause latency to dominate communication overhead.

Designing parallel programs - communication:

Most parallel applications require tasks to share data with each other.

Cost of communication: Computational resources are used to package and transmit data. Requires frequently synchronization – some tasks will wait instead of doing work. Could saturate network bandwidth.

Latency vs. Bandwidth: Latency is the time it takes to send a minimal message between two tasks. Bandwidth is the amount of data that can be communicated per unit of time. Sending many small messages can cause latency to dominate communication overhead.

Synchronous vs. Asynchronous communication: Synchronous communication is referred to as blocking communication – other work stops until the communication is completed. Asynchronous communication is referred to as non-blocking since other work can be done while communication is taking place.

Designing parallel programs - communication:

Most parallel applications require tasks to share data with each other.

Cost of communication: Computational resources are used to package and transmit data. Requires frequently synchronization – some tasks will wait instead of doing work. Could saturate network bandwidth.

Latency vs. Bandwidth: Latency is the time it takes to send a minimal message between two tasks. Bandwidth is the amount of data that can be communicated per unit of time. Sending many small messages can cause latency to dominate communication overhead.

Synchronous vs. Asynchronous communication: Synchronous communication is referred to as blocking communication – other work stops until the communication is completed. Asynchronous communication is referred to as non-blocking since other work can be done while communication is taking place.

Scope of communication: Point-to-point communication – data transmission between tasks. Collective communication – involves all tasks (in a communication group)

Designing parallel programs - communication:

Most parallel applications require tasks to share data with each other.

Cost of communication: Computational resources are used to package and transmit data. Requires frequently synchronization – some tasks will wait instead of doing work. Could saturate network bandwidth.

Latency vs. Bandwidth: Latency is the time it takes to send a minimal message between two tasks. Bandwidth is the amount of data that can be communicated per unit of time. Sending many small messages can cause latency to dominate communication overhead.

Synchronous vs. Asynchronous communication: Synchronous communication is referred to as blocking communication – other work stops until the communication is completed. Asynchronous communication is referred to as non-blocking since other work can be done while communication is taking place.

Scope of communication: Point-to-point communication – data transmission between tasks. Collective communication – involves all tasks (in a communication group)

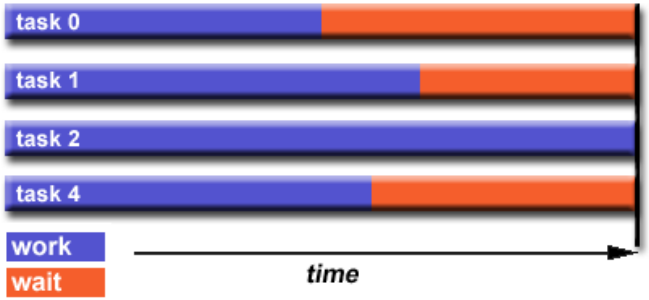
This is only partial list of things to consider!

Designing parallel programs – load balancing:

Load balancing is the practice of distributing approximately equal amount of work so that all tasks are kept busy all the time.

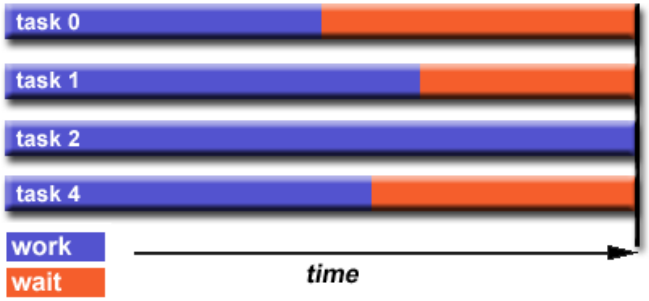
Designing parallel programs – load balancing:

Load balancing is the practice of distributing approximately equal amount of work so that all tasks are kept busy all the time.



Designing parallel programs – load balancing:

Load balancing is the practice of distributing approximately equal amount of work so that all tasks are kept busy all the time.



How to Achieve Load Balance?

Equally partition the work given to each task: For array/matrix operations equally distribute the data set among parallel tasks. For loop iterations where the work done for each iteration is equal, evenly distribute iterations among tasks.

Use dynamic work assignment: Certain class problems result in load imbalance even if data is distributed evenly among tasks (sparse matrices, adaptive grid methods, many body simulations, etc.). Use scheduler – task pool approach. As each task finishes, it queues to get a new piece of work. Modify your algorithm to handle imbalances dynamically.

Designing parallel programs – I/O:

The Bad News:

- I/O operations are inhibitors of parallelism
- I/O operations are orders of magnitude slower than memory operations
- Parallel file systems may be immature or not available on all systems
- I/O that must be conducted over network can cause severe bottlenecks

Designing parallel programs – I/O:

The Bad News:

- I/O operations are inhibitors of parallelism
- I/O operations are orders of magnitude slower than memory operations
- Parallel file systems may be immature or not available on all systems
- I/O that must be conducted over network can cause severe bottlenecks

The Good News:

- Parallel file systems are available (e.g., Lustre)
- MPI parallel I/O interface has been available since 1996 as a part of MPI-2

Designing parallel programs – I/O:

The Bad News:

- I/O operations are inhibitors of parallelism
- I/O operations are orders of magnitude slower than memory operations
- Parallel file systems may be immature or not available on all systems
- I/O that must be conducted over network can cause severe bottlenecks

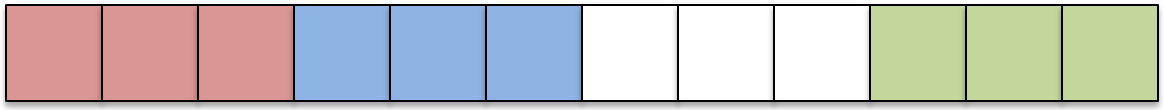
The Good News:

- Parallel file systems are available (e.g., Lustre)
- MPI parallel I/O interface has been available since 1996 as a part of MPI-2

I/O Tips:

- Reduce overall I/O as much as possible
- If you have access to parallel file system, use it
- Writing large chunks of data rather than small ones is significantly more efficient
- Fewer, larger files perform much better than many small files
- Have a subset of parallel tasks to perform the I/O instead of using all tasks, or
- Confine I/O to a single tasks and then broadcast (gather) data to (from) other tasks

Example – array processing:



task 1 task 2 … task N

Serial code

```
do i = 1, N
  a( i ) = fcn( i )
end do
```

Parallel code

```
Find out if I am MASTER or WORKER
if I am MASTER

  initiate the array
  send each WORKER info on part of array it owns
  send each WORKER its portion of initial array
  receive results from each WORKER

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

  # process my portion of array
  do i = mystart, myend
    a( i ) = fcn( i )
  end do

  send MASTER results

end if
```

Contact Information:

Harvard Research Computing Website:

http://rc.fas.harvard.edu

Email:

rchelp@fas.harvard.edu

plamenkrastev@fas.harvard.edu