

## **Designing Parallel Programs**

This review was developed from “Introduction to Parallel Computing”

Author: Blaise Barney, Lawrence Livermore National Laboratory

references: [https://computing.llnl.gov/tutorials/parallel\\_comp/#WhatIs](https://computing.llnl.gov/tutorials/parallel_comp/#WhatIs)

There are two ways to develop parallel programs

1. Automatic - compiler finds any possible parallelization points
2. Manual - programmer finds any possible parallelization points

### Manual Method

1. Determine if the problem is one that can be solved in parallel.

Example of Parallelizable Problem:

**Calculate the potential energy for each of several thousand independent conformations of a molecule. When done, find the minimum energy conformation.**

Example of a Non-parallelizable Problem:

**Calculation of the Fibonacci series (0,1,1,2,3,5,8,13,21,...) by use of the formula:**  
 **$F(n) = F(n-1) + F(n-2)$**

2. Identify Program Hotspots

Hotspots are areas where a lot of work is being done by the program.  
Concentrate on parallelizing the hot spots.

3. Identify Bottlenecks

Are there areas, such as I/O commands, that slow or even halt the parallel portion of the program.

4. Identify Inhibitors to Parallelism

Data dependence (Fib series) is a common inhibitor.

5. If possible look at other algorithms to see if they can be parallelized.

6. Use parallel programs and libraries from third party vendors.

## Designing Parallel Programs: Partitioning

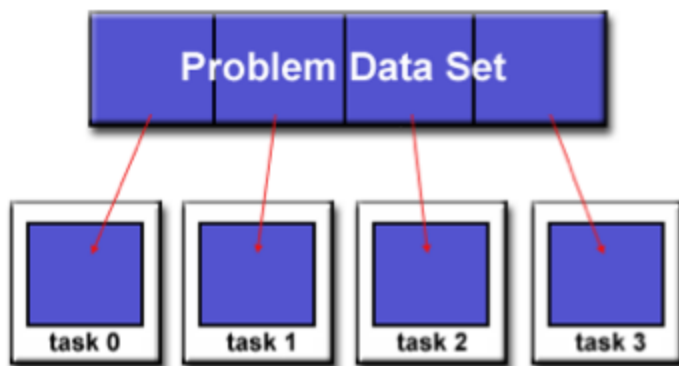
Divide the tasks to be done into discrete chunks.

There are two methods for partitioning:

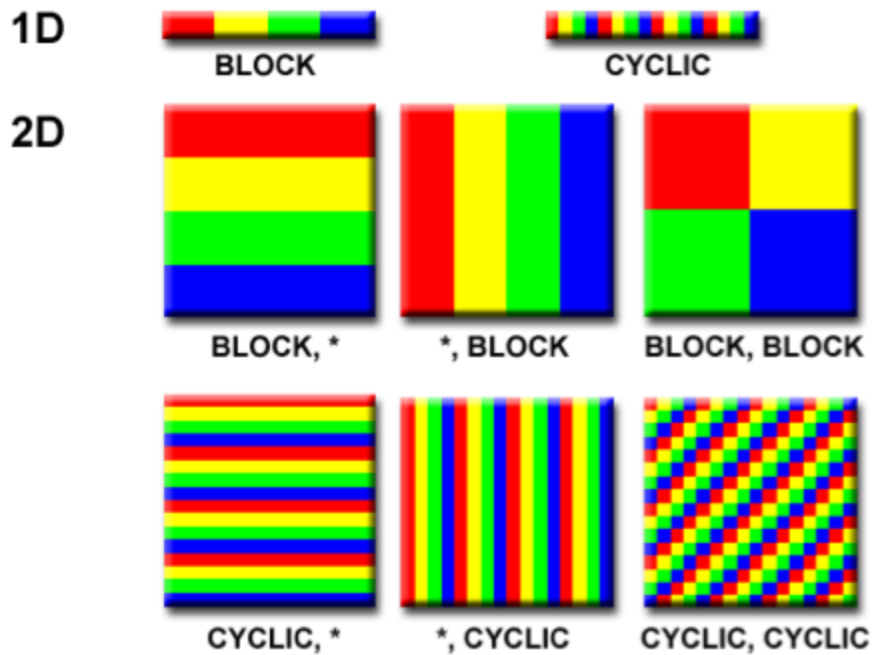
1. Domain Decomposition
2. Functional Decomposition

Domain Decomposition

The data is decomposed. Then each task works on a portion of the data.

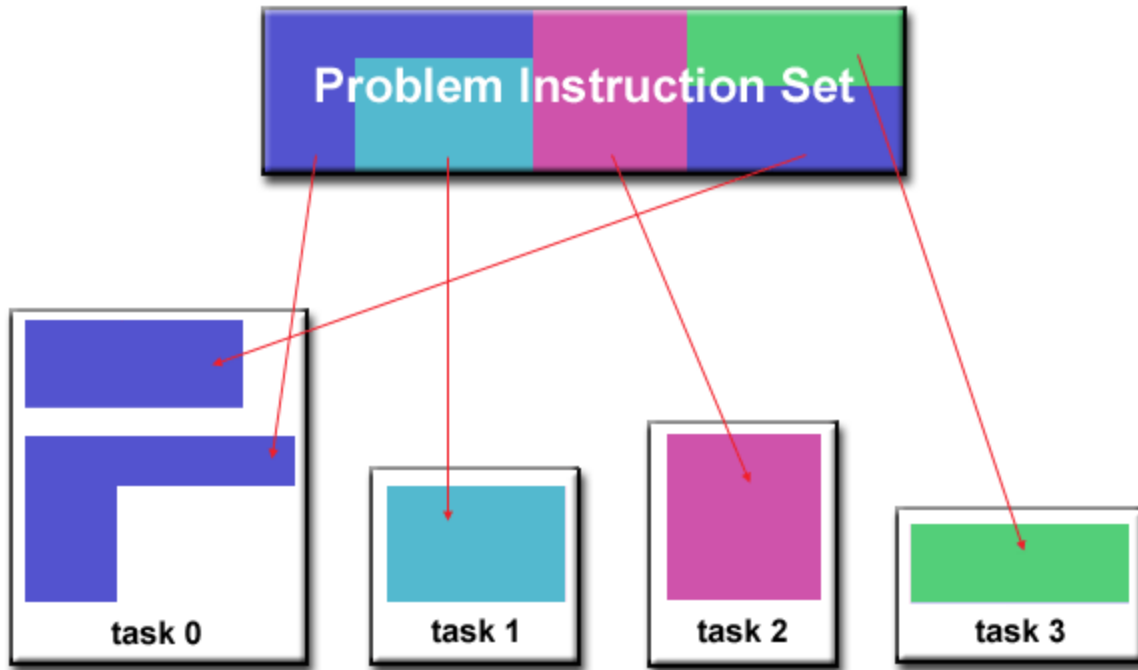


- There are different ways to partition data:



## Functional Decomposition

The problem is broken into smaller tasks. Then each task does part of the work that needs to be done. In functional decomposition the emphasis is on the computation, rather than the data.



### Examples of Functional Decomposition

1. Ecosystem modeling
2. Signal processing
3. Climate modeling

## **Communications**

Which tasks need to communicate with each other.

Communication is not needed:

If two tasks do not share data and the tasks are independent of each other, then this type of task does not need to communicate with other tasks.

An example: A black and white image needs to have its color reversed. Each pixel can be independently reversed, so there is no need for the tasks to exchange information.

Communication is needed:

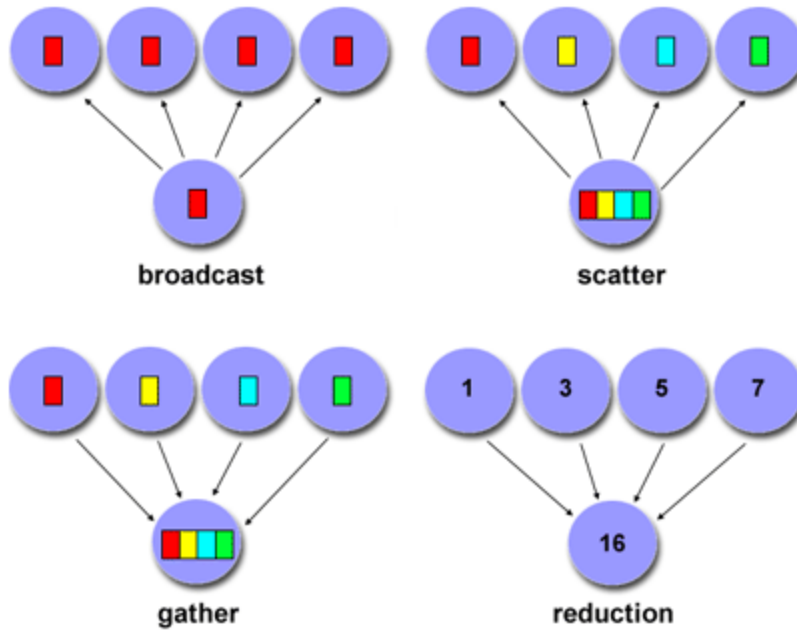
Tasks need information from other tasks to complete the work.

An example: A 3-D heat diffuser problem. Each task needs to know what neighboring portions of the problem have calculated so that it can adjust its own information.

### **Facts to Consider with Communication**

1. Cost of communication
2. Latency vs Bandwidth - small packages cause latency to dominate the communications overhead. Larger packages make fuller use of the bandwidth.
3. Visibility of Communication - With Message Passing communication is explicit and under control of the programmer. With the data parallel model the programmer does not know when communication is occurring and has no control over it.
4. Synchronous vs Asynchronous communication - Synchronous - called blocking - requires handshaking (send, receive). Asynchronous - called non-blocking - does not need to wait for communication to finish. Interleaving computation is the biggest advantage of asynchronous communication.
5. Scope of Communication - two scope levels 1. point to point = one task is the sender and one task is the receiver, 2. Collective = data sharing between two or more tasks. both can be synchronous or asynchronous.

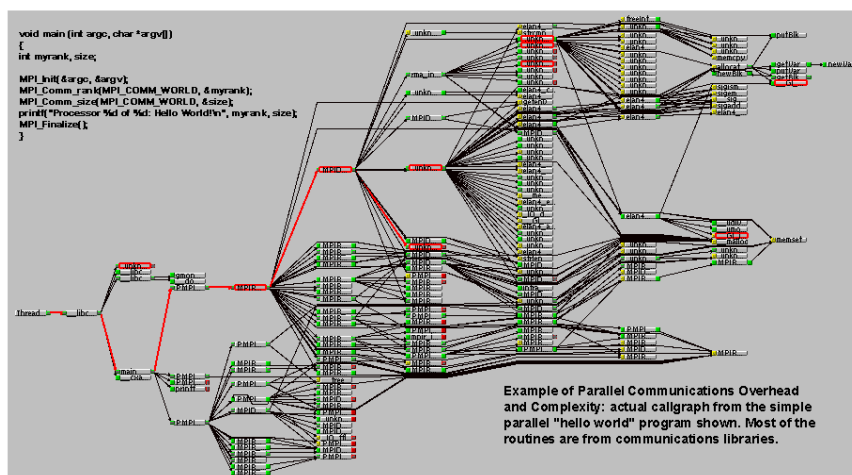
## Types of Collective Communication



## 6. Efficiency of Communications

Choosing the right type of communication will affect its efficiency.

## 7. Overhead and Complexity are increased with communications.



## **Synchronization**

Managing the work and the tasks required to complete the work - is a critical design consideration. Program performance can be significantly affected. Portions of the program may need to be serialized.

There are 3 kinds of Synchronization:

- barrier - when each task reaches the barrier it stops. When the last task reaches the barrier, all the tasks are synchronized and released. This is usually done to sync the tasks before a serial portion of the program.
- lock/semaphore - some of tasks are involved in a lock or semaphore. The purpose of the lock is to allow only one task at a time access to either a global variable or a serial portion of code. The first task to the lock sets it, then all other tasks must wait until the lock is released before they can acquire the variable or access to the code.
- synchronous communication operations - tasks that are communicating use this synchronization. Before a task can send a message it must get acknowledgement that the receiving task is ready for it.

## Data Dependencies

- A dependence occurs in a program when the order of execution affects the outcome of the program.
- A dependence occurs when more than one task is using a variable in a program.
- Dependencies are important in parallel programming because they are the main inhibitor to parallelism.

## Data Dependence Examples

### Loop carried Dependence

```
DO 500 J = MYSTART,MYEND  
  A(J) = A(J-1) * 2.0  
500 CONTINUE
```

A(J-1) needs to be calculated before A(J) can be calculated.  
A(J) is data dependent on A(J-1).

To properly compute the value of A(J):

Assume task2 calculates A(J) and task1 calculates A(J-1).

In a Distributed Memory Architecture this means task1 must finish before task2 can begin.

In a Shared Memory Architecture task1 must write the value of A(J-1) to memory before task2 can retrieve it.

### Loop Independent Data Dependence

task 1	task 2
-----	-----
X = 2	X = 4
.	.
.	.
Y = X**2	Y = X**3

In this instance, the order in which a task completes will affect the outcome of the program.

In a Distributed Memory Architecture the value of Y is dependent upon when the value of X is transferred between the two tasks.

In a Shared Memory Architecture the value of Y is dependent upon the last update of X.

Loop carried data dependencies are particularly important because loops are often a primary target of parallelization opportunities.



## How to Handle Data Dependencies

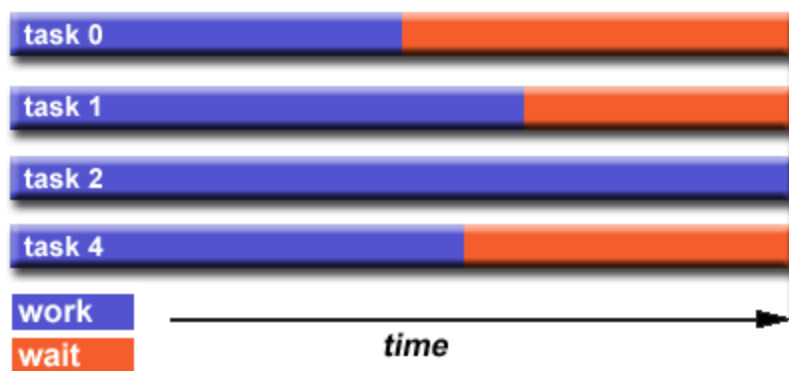
In Distributed Memory Architectures - data must be communicated at synchronization points.

In Shared Memory Architectures - synchronize read/write operations between tasks.

## Load Balancing

The goal of task balancing is to keep all tasks busy all of the time, it can also be seen as minimization of task idle time.

In parallel programs the slowest task will determine the overall performance.



## How to Achieve Load Balance

- Equally Partition the Work Each Task Receives

For array/matrix operations where each task performs similar work	Evenly distribute the data set among the tasks
For loop iterations where the work done in each iteration is similar	Evenly distribute the iterations across the tasks
If a heterogeneous mix of machines with varying performance characteristics are being used	Be sure to use some type of performance analysis tool to detect any load imbalances.

- Dynamic Work Assignment

Certain classes of problems result in load imbalances even if data is evenly distributed among tasks.

Sparse Arrays	Some tasks will have actual data to work on while others have mostly 'zeros'
Adaptive Grid Methods	Some tasks may need to refine their mesh while others don't
N-body simulations	Where some particles may migrate to/from their original task domain to another task's. Where particles owned by some tasks require more work than those owned by other tasks.

When to use a scheduler - task pool approach: then the amount of work of each task is variable or unpredictable. As each task finishes it joins the task pool to be assigned a new task.

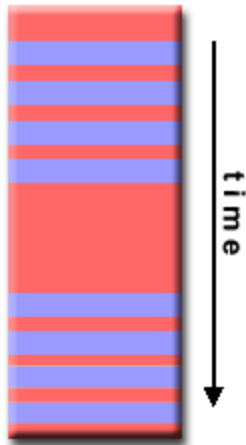
For some situations, it may be necessary to design an algorithm to detect load imbalances.

## Granularity

### Computation/ Communication Ratio:

- Granularity is the ratio of computation to communication.
- Periods of computation are separated from communication periods through the use of synchronous events.

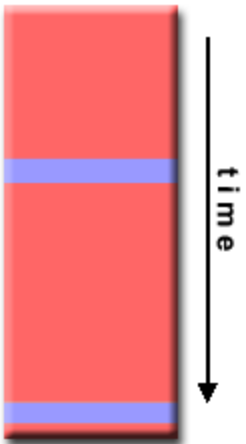
■ communication  
■ computation



### Fine-grain Parallelism:

- The work is divided into many small amounts between communication events
- Low computation to communication ratio
- Facilitates load balancing
- Results in high communication costs and lower performance enhancement
- If granularity is too fine communication and synchronization overhead can be greater than computation.

### Coarse Grain Parallelism



■ communication  
■ computation

- In between communication / synchronization events large amounts of computational work is performed.
- There is a high computation to communication ratio
- There is more opportunity for increases in performance.
- It is harder to load balance effectively

## **Fine Grain - Coarse Grain Which is Best?**

Best = most efficient granularity

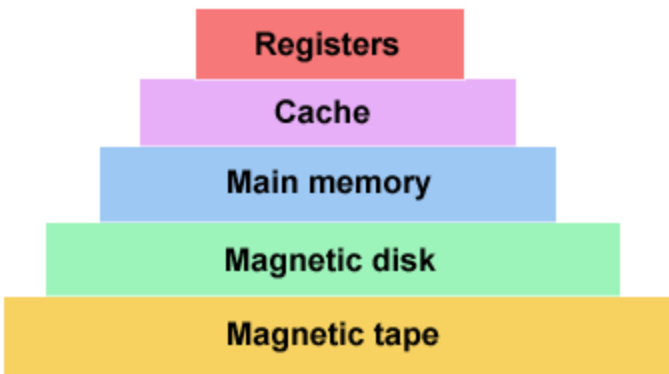
The algorithm and hardware environment play important roles in the efficiency of a run.

Coarse granularity is usually advantageous because of the overhead associated with communications and synchronization is high compared to the execution speed.

Fine grain can help reduce load imbalance overheads.

## I/O

### Memory Hierarchy



A pyramid diagram representing the memory hierarchy. The pyramid is divided into five horizontal layers, each with a different color and text. From top to bottom: a red layer labeled 'Registers', a purple layer labeled 'Cache', a blue layer labeled 'Main memory', a green layer labeled 'Magnetic disk', and a yellow layer labeled 'Magnetic tape'. To the right of the pyramid, there are two columns of text corresponding to each layer, showing time and relative speed.

Registers	1 ns	1x
Cache	10 ns	10x
Main memory	100 ns	100x
Magnetic disk	100 ms	100,000,000x
Magnetic tape	10 s	1e+10x

#### The Bad News

- I/O operations are generally considered to inhibitors to parallelism
- I/O operations require much more time than memory operations (by orders of magnitude)
- Parallel I/O systems may not be available for all platforms.
- In environments where all tasks see the same file space, write operations can result in file overwriting.
- Read operations can be affected by the file servers ability to handle multiple simultaneous reads.
- Over network I/O can cause severe bottlenecks and crash servers.

#### The Good

- Parallel file systems are available
- MPI has had I/O programming interface since 1996.

#### I/O Pointers

- REDUCE I/O AS MUCH AS POSSIBLE
- Use a parallel file system - if you have it.
- Writing large blocks of data is more efficient than multiple small blocks.
- Confine I/O to the serial portions of the code.
- Have a small subset of tasks perform I/O

## **Debugging**

Debugging parallel code can be very difficult.

Good debuggers are:

- For threaded programs: pthreads and OpenMP
- MPI
- GPU / accelerator
- Hybrid

## **Performance Analysis and Tuning**

- Analyzing and tuning is very difficult.
- The use of tuning tools is highly recommended.