



# Parallelization

T. Powell

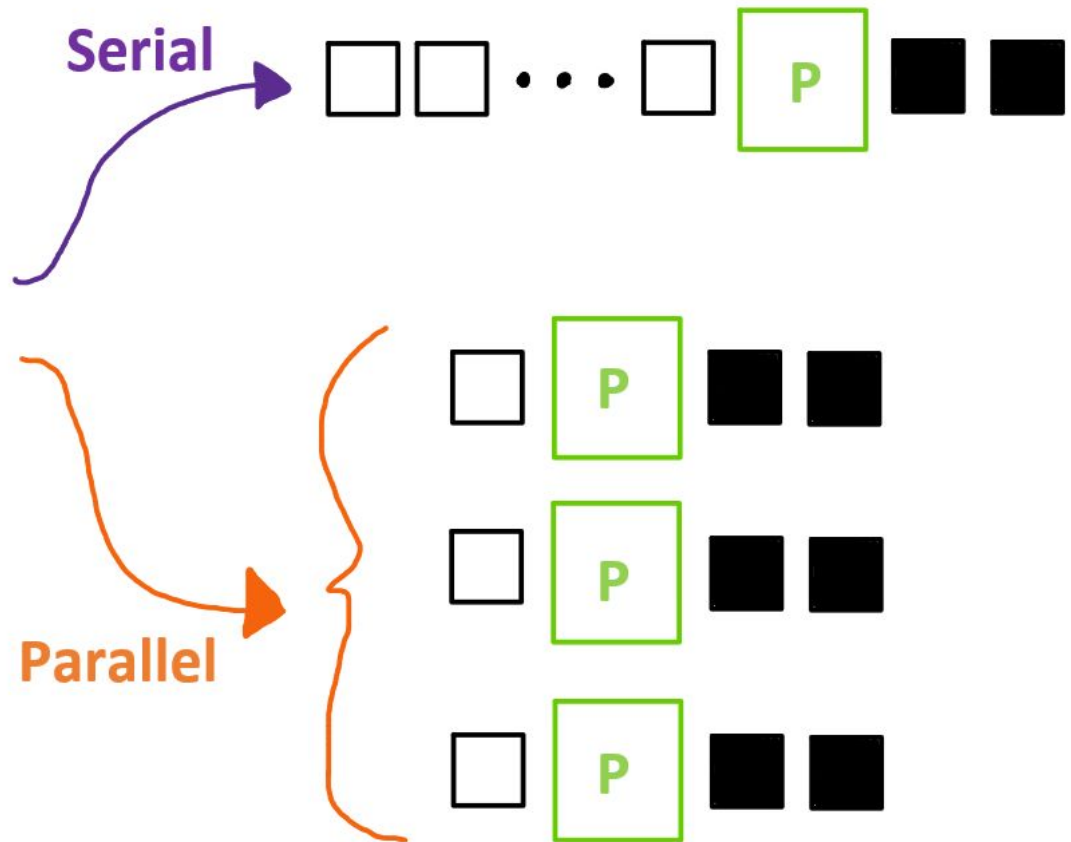
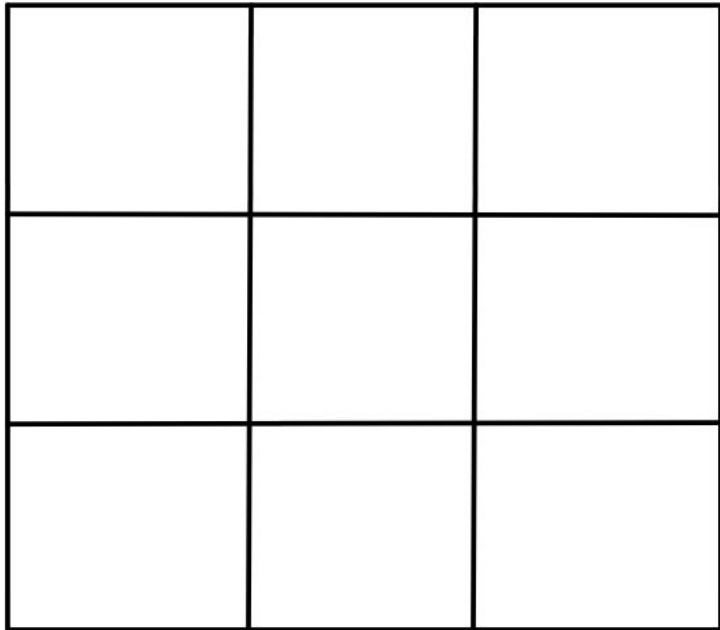
1. Computational Benefits & Speedup
2. Algorithms
3. OpenMP
4. Message-Passing Interface (MPI)

**Part 1:**

**Computational Benefits & Speedup**

# What is Parallelization?

## Tasks



## Where is parallelization useful?

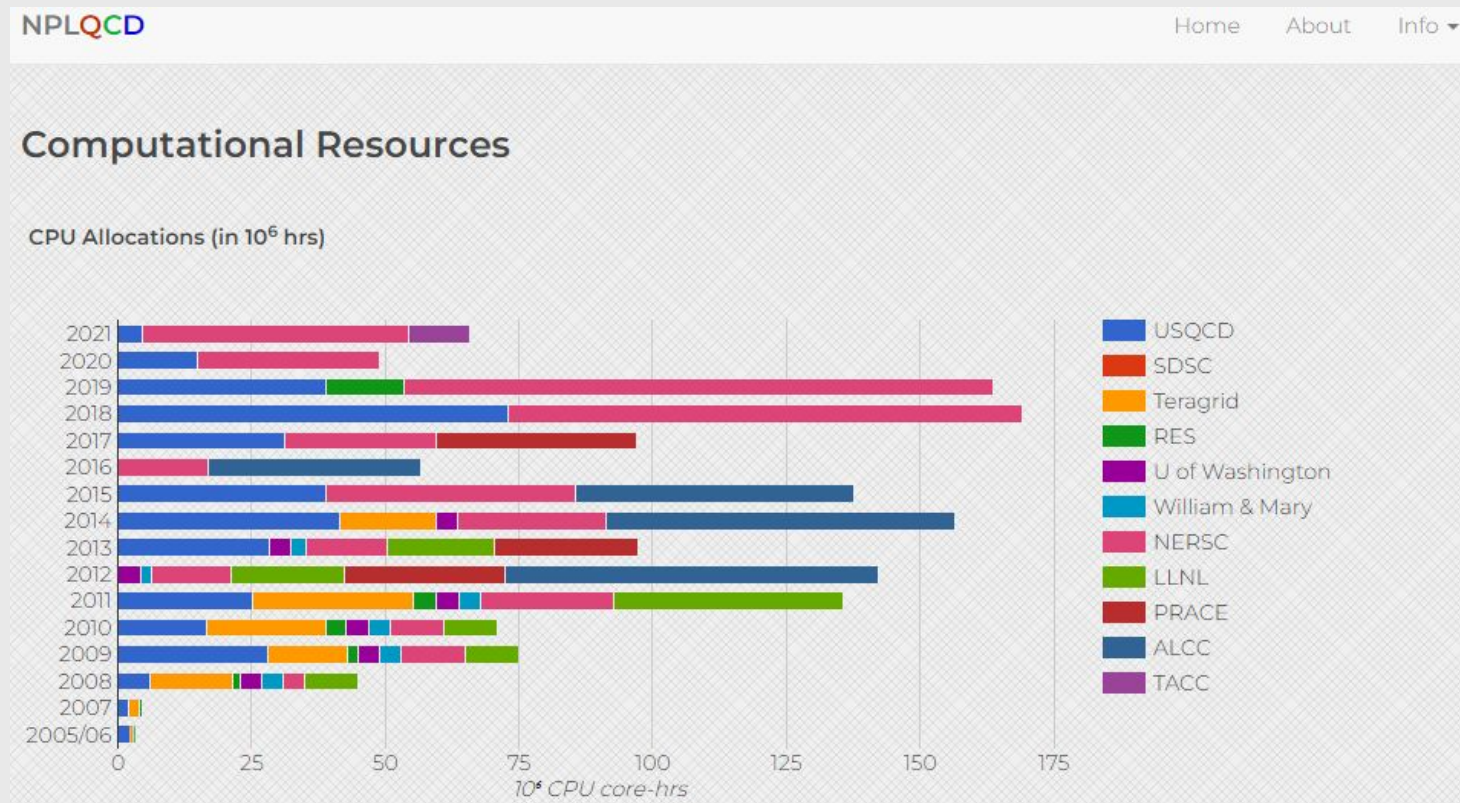
Most desktops and laptops now feature multi-core processors, which run dozens of concurrent processes at any given time to keep the PC running (just check your task manager!).

In the sciences, the push towards solving more complex and difficult problems has led to a similar need for more processing power to obtain results in a reasonable timeframe. The primary utility of supercomputers (HPCs) is the vast number of processors available to do simultaneous calculations on distinct portions of a single problem.

Presently, HPCs are used for everything from maintaining the nuclear stockpile and weather forecasting to large-scale molecular dynamics simulations and simulating nuclear particle interactions.

# Where is parallelization useful?

The Nuclear Physics with Lattice Quantum Chromodynamics (NPLQCD) collaboration uses roughly 50-150 million core hours per year for calculations done by their various users



## Where is parallelization useful?

The Partnership for Advanced Computing in Europe (PRACE) has a call for proposals annually, and last awarded a total of 2 billion core hours to 43 different projects.

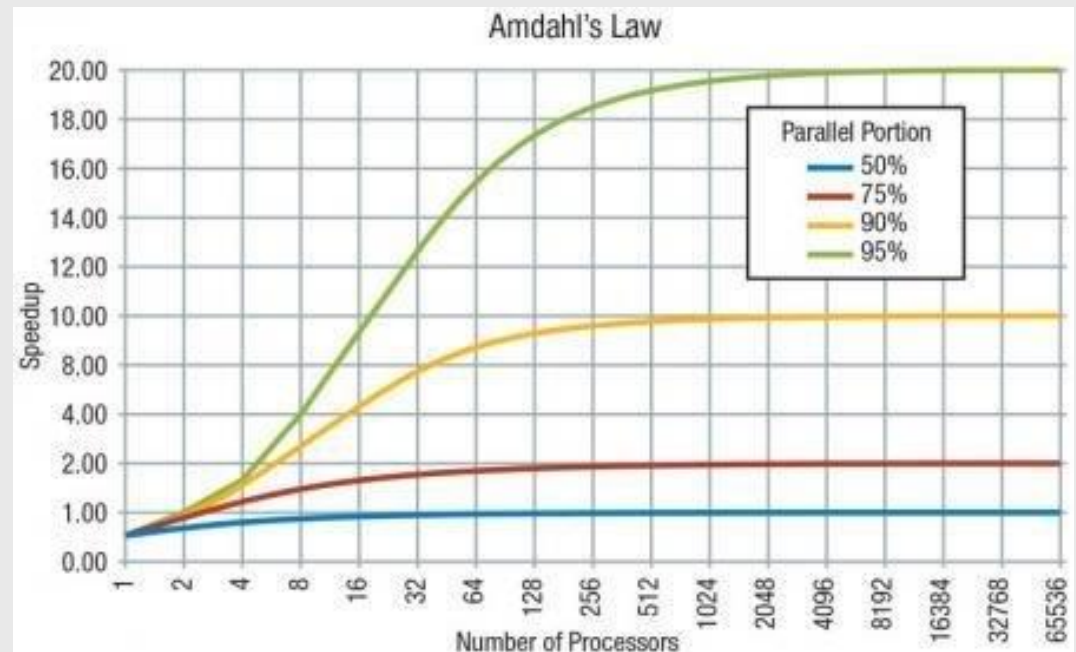
| System                      | Architecture                   | Site (Country)       | Core Hours (node hours)      | Minimum request (core hours) |
|-----------------------------|--------------------------------|----------------------|------------------------------|------------------------------|
| HAWK <sup>*</sup>           | HPE Apollo                     | GCS@HLRS (DE)        | 345.6 million (2.7 million)  | 100 million                  |
| Joliot-Curie KNL            | BULL Sequana X1000             | GENCI@CEA (FR)       | 37.5 million (0.6 million)   | 15 million                   |
| Joliot-Curie Rome           | BULL Sequana XH2000            | GENCI@CEA (FR)       | 195.3 million (1.5 million)  | 15 million                   |
| Joliot-Curie SKL            | BULL Sequana X1000             | GENCI@CEA (FR)       | 52.9 million (1.1 million)   | 15 million                   |
| JUWELS Booster <sup>*</sup> | BULL Sequana XH2000            | GCS@JSC (DE)         | 85.2 million (1.78 million)  | 7 million<br>Use of GPUs     |
| JUWELS Cluster <sup>*</sup> | BULL Sequana X1000             | GCS@JSC (DE)         | 35.04 million (0.73 million) | 35 million                   |
| Marconi100                  | IBM Power 9 AC922 Whisperspoon | CINECA (IT)          | 165 million (1.87 million)   | 35 million<br>Use of GPUs    |
| MareNostrum 4 <sup>*</sup>  | Lenovo System                  | BSC (ES)             | TBA                          | 30 million                   |
| Piz Daint                   | Cray XC50 System               | ETH Zurich/CSCS (CH) | 510 million (7.5 million)    | 68 million<br>Use of GPUs    |
| SuperMUC-NG <sup>*</sup>    | Lenovo ThinkSystem             | GCS@LRZ (DE)         | 91 million                   | 35 million                   |

## Speedup - Amdahl's Law

When parallelizing a program, one major consideration is the amount of speedup that can be achieved through allocating additional resources.

The theoretical speedup of task execution for a process with additional resources is given by Amdahl's law,

$$S = \frac{1}{(1 - p) + \frac{p}{s}}$$



## Limitations to Speedup

In the creation of parallelized programs, the primary theoretical limitation to the speedup of a process is the amount of dependence between portions of the problem.

**Inherently serial problems** - Some problems require information from previous steps in order to complete future steps.

e.g., Iterative numerical methods such as ODE solvers

**Embarrassingly parallel problems** - Other problems are ripe for parallelization because the individual steps are completely independent of one another

e.g., Numerical integration, Monte-Carlo



## Limitations to Speedup

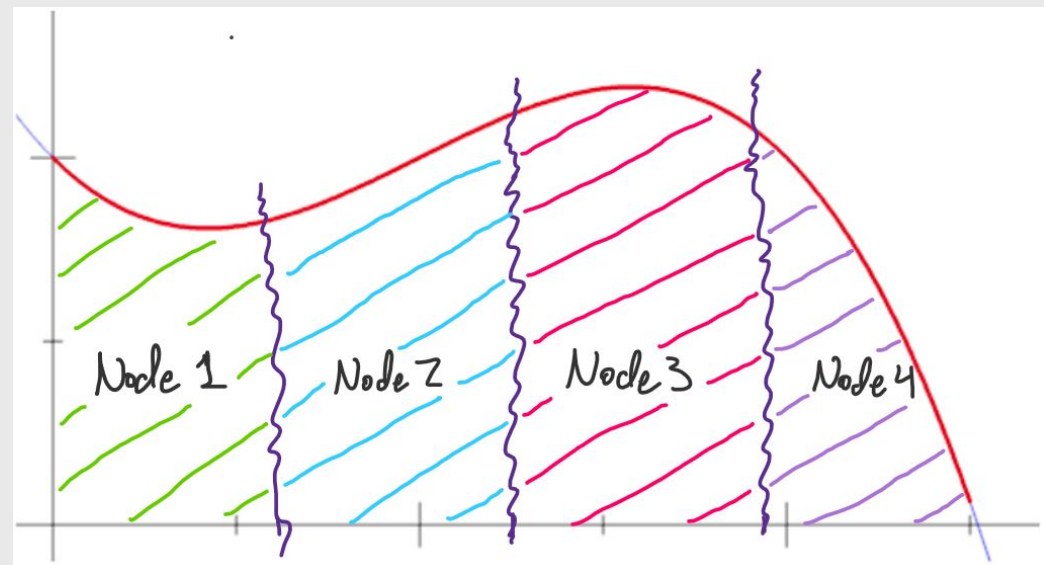
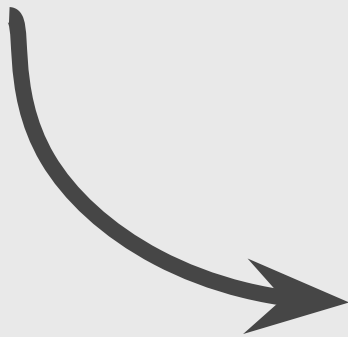
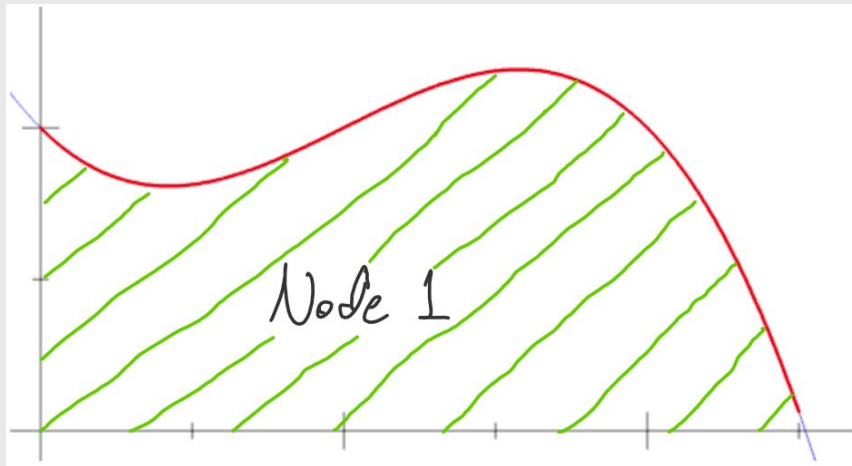
For most programs, tasks are a mixture of some serial portions and some parallelizable tasks.

For those tasks which are parallelizable, it becomes important to design an algorithm which divides up the work as evenly as possible between the processors (*load-balancing*).

For tasks which require at least *some* communication between individual nodes, balancing communication overhead also becomes an important consideration to ensure an implementation remains efficient.

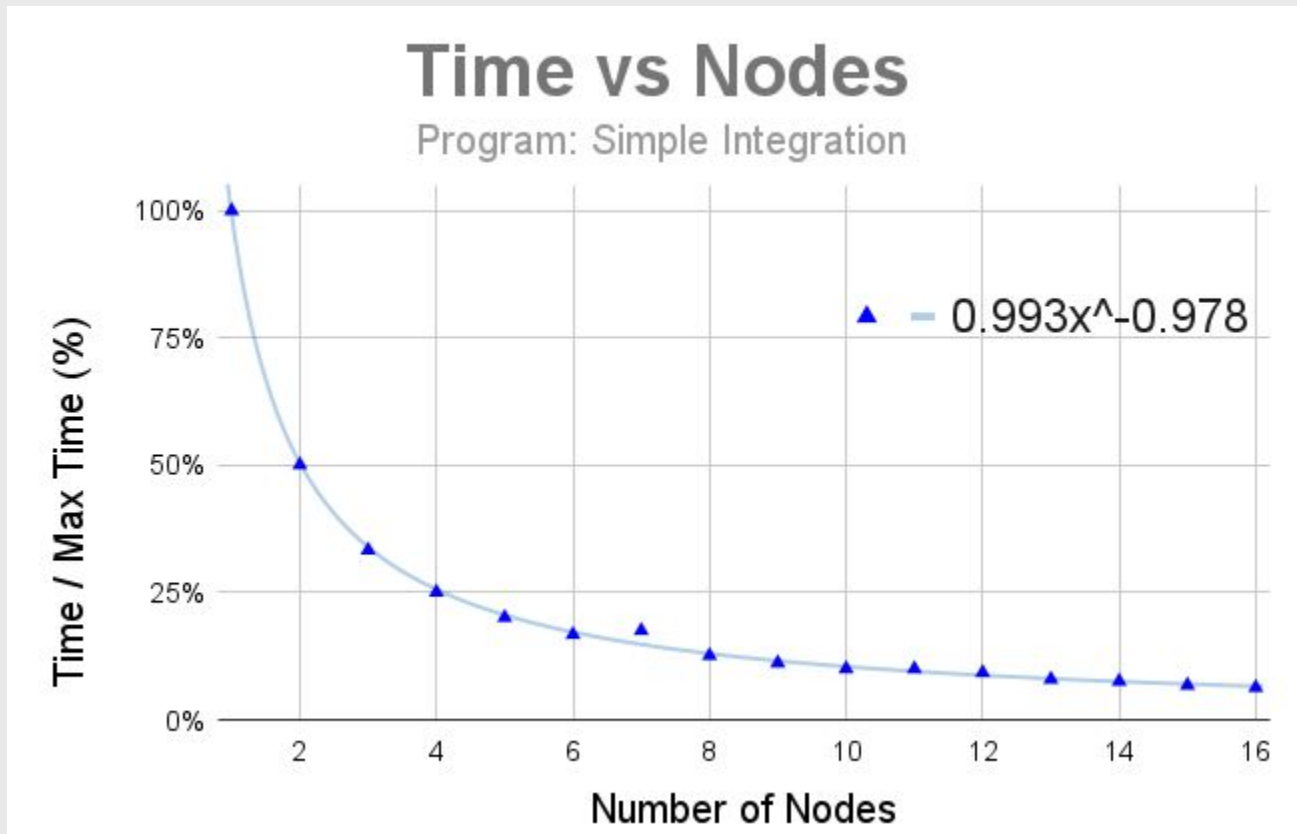
# Case Study: Improper Load Balancing

Consider a program which uses parallelization to split up the task of numerical integration over some region



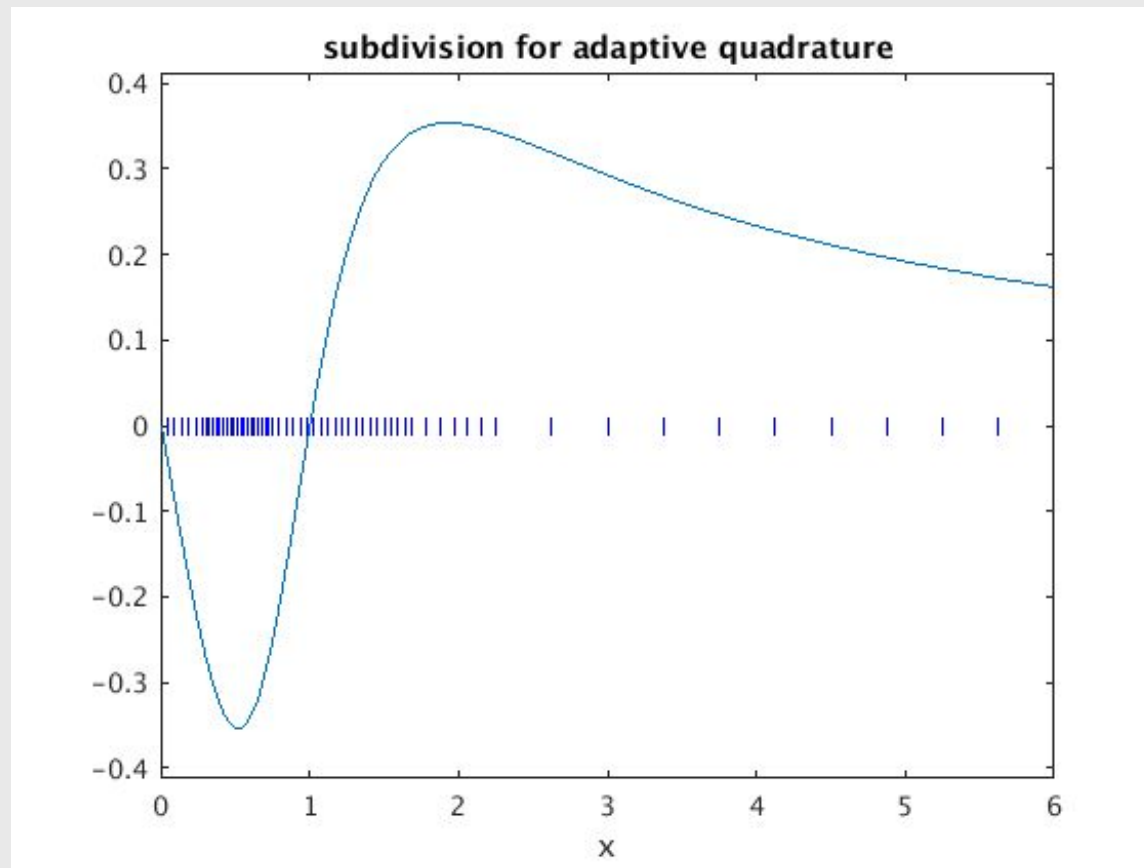
# Case Study: Improper Load Balancing

Performing numerical integration for some simple function with  $\sim 10^7$  steps, we can find the speedup from including more nodes for the computation.



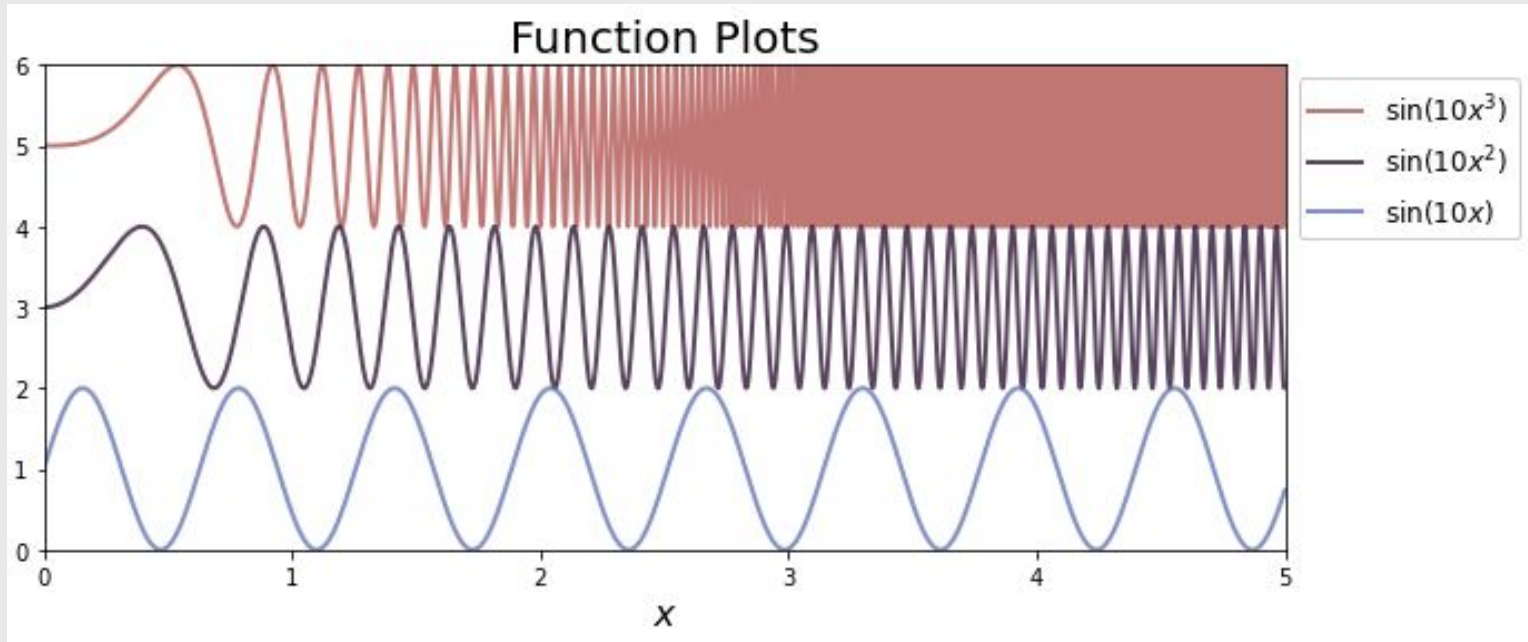
## Case Study: Improper Load Balancing

Now let's have this program implement adaptive integration. Each of the subdivided regions are now sent to nodes, which use an adaptive integration routine to obtain the local value for the integral,



## Case Study: Improper Load Balancing

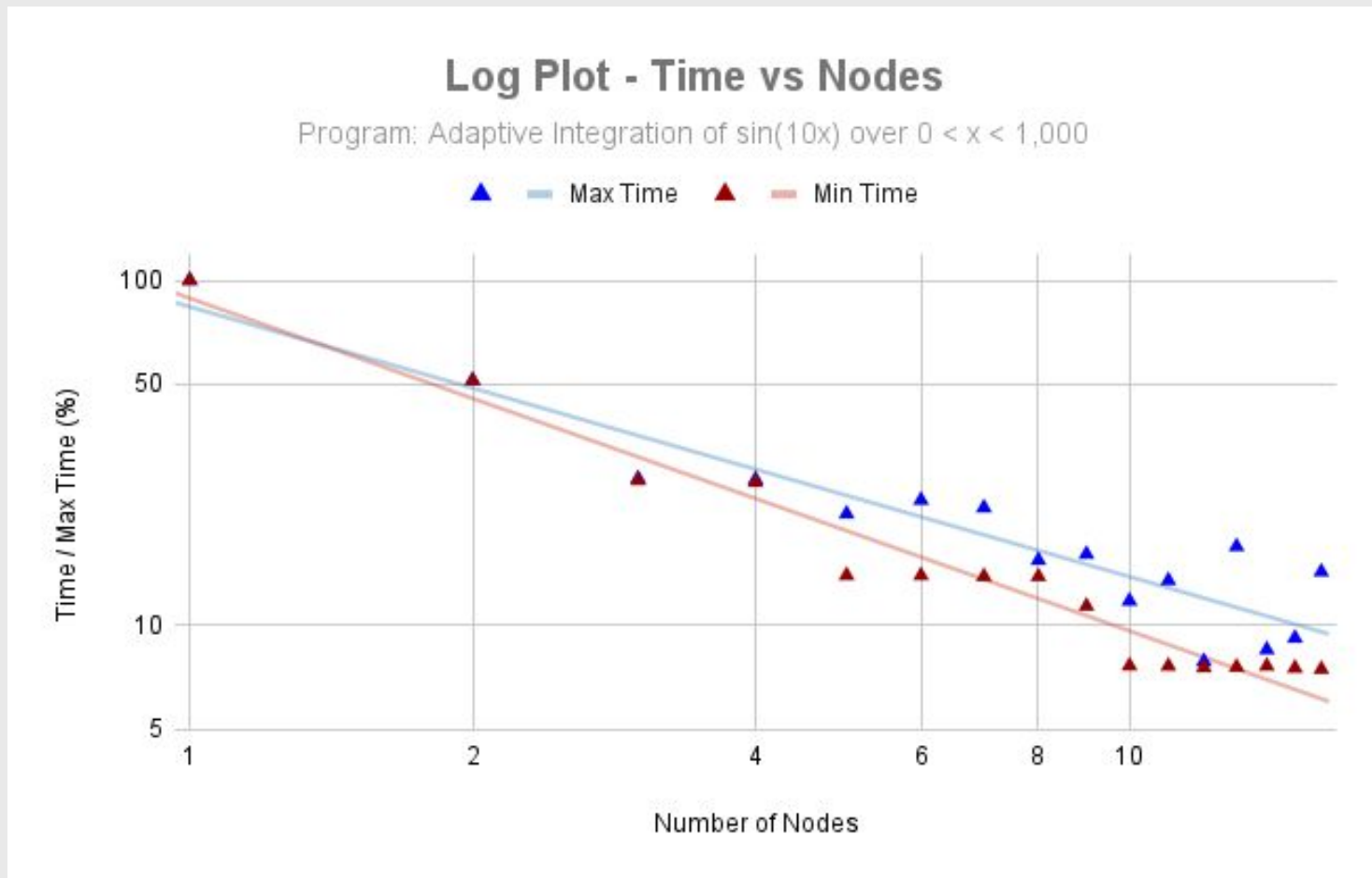
Finally, this program examines a set of sinusoids dependent on different powers of  $x$ ,



Clearly, for higher powers of  $x$ , processors which are assigned subsections further from the origin will take longer to perform the numerical integration than those closer to the origin.

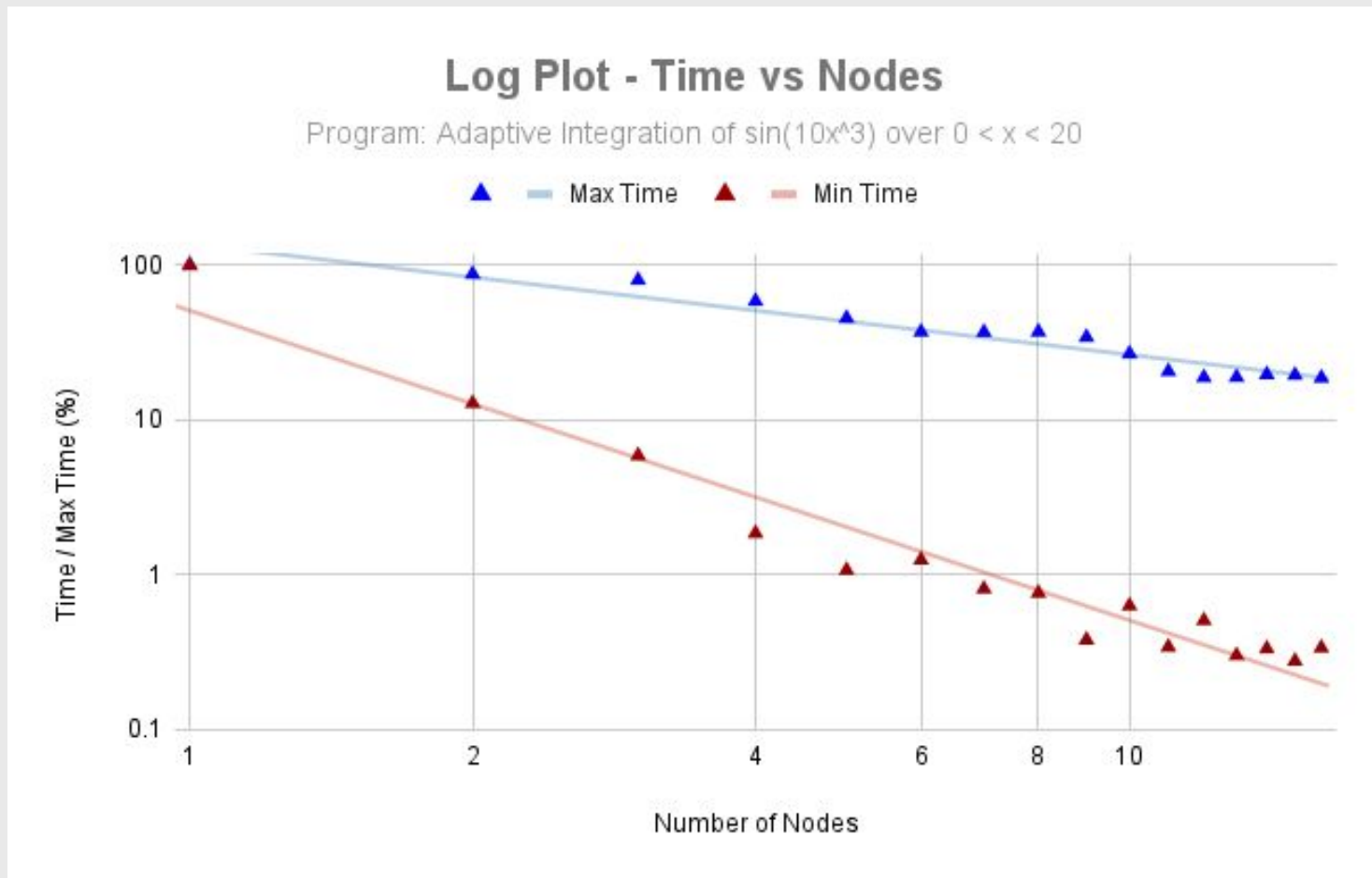
# Case Study: Improper Load Balancing

Linear

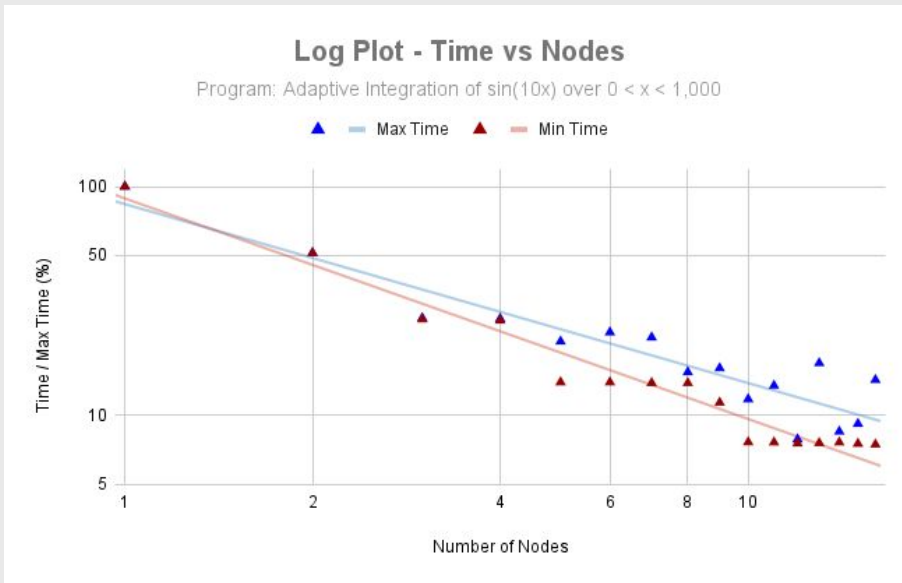


# Case Study: Improper Load Balancing

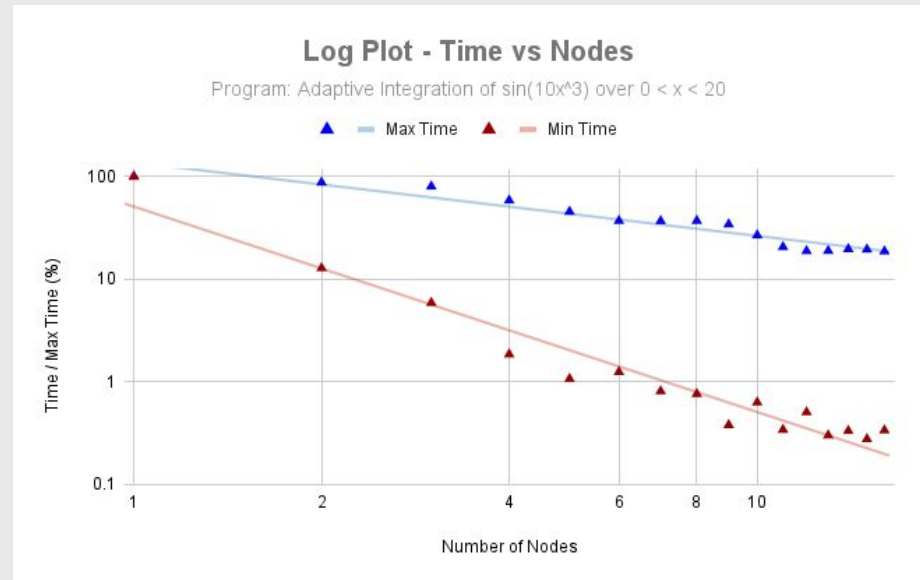
## Cubic



# Case Study: Improper Load Balancing



Linear



Cubic



**Part 2:**

**Algorithms**

# Algorithms - Standard Task Distribution

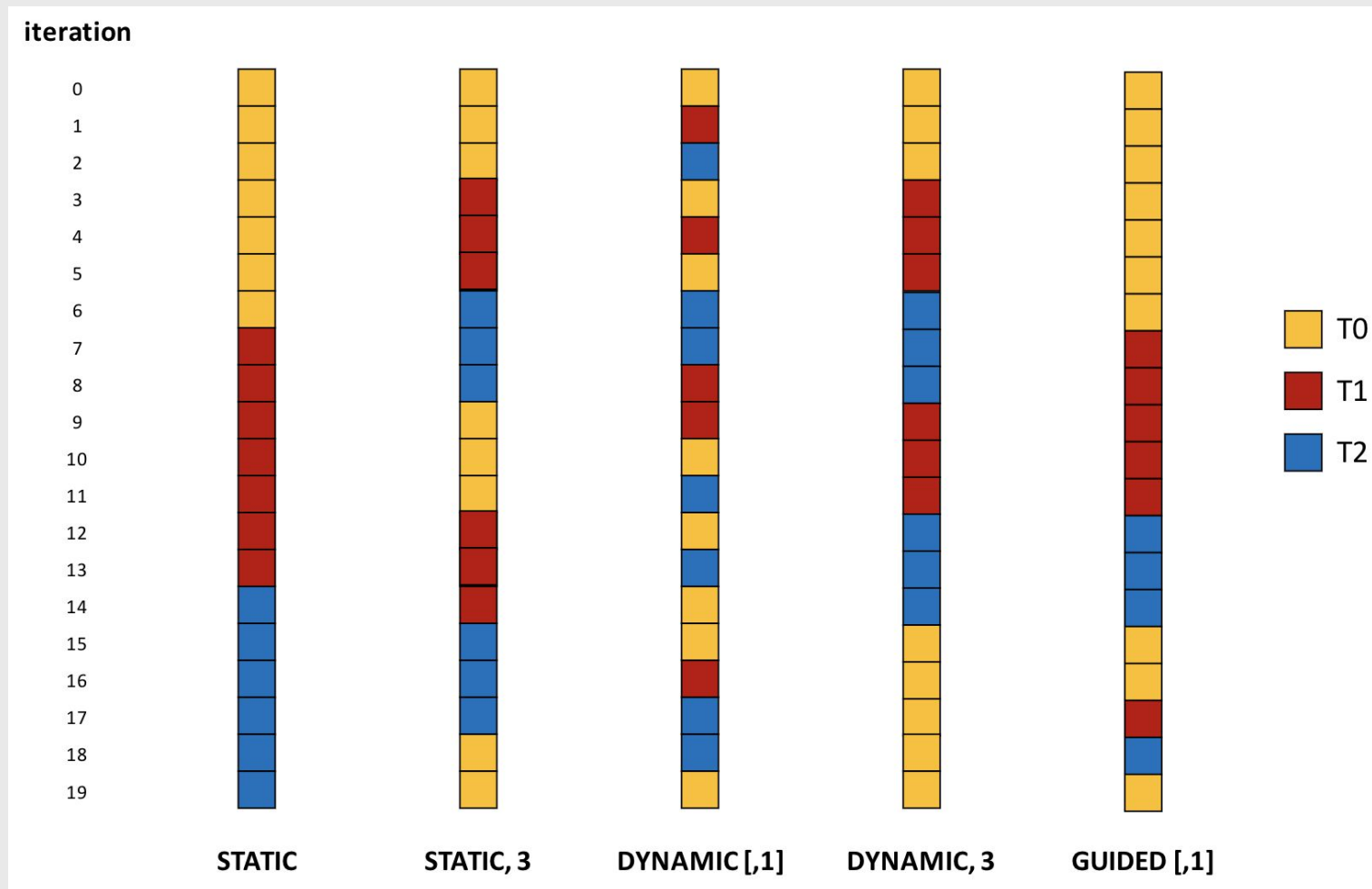
$P_1$

$P_2$

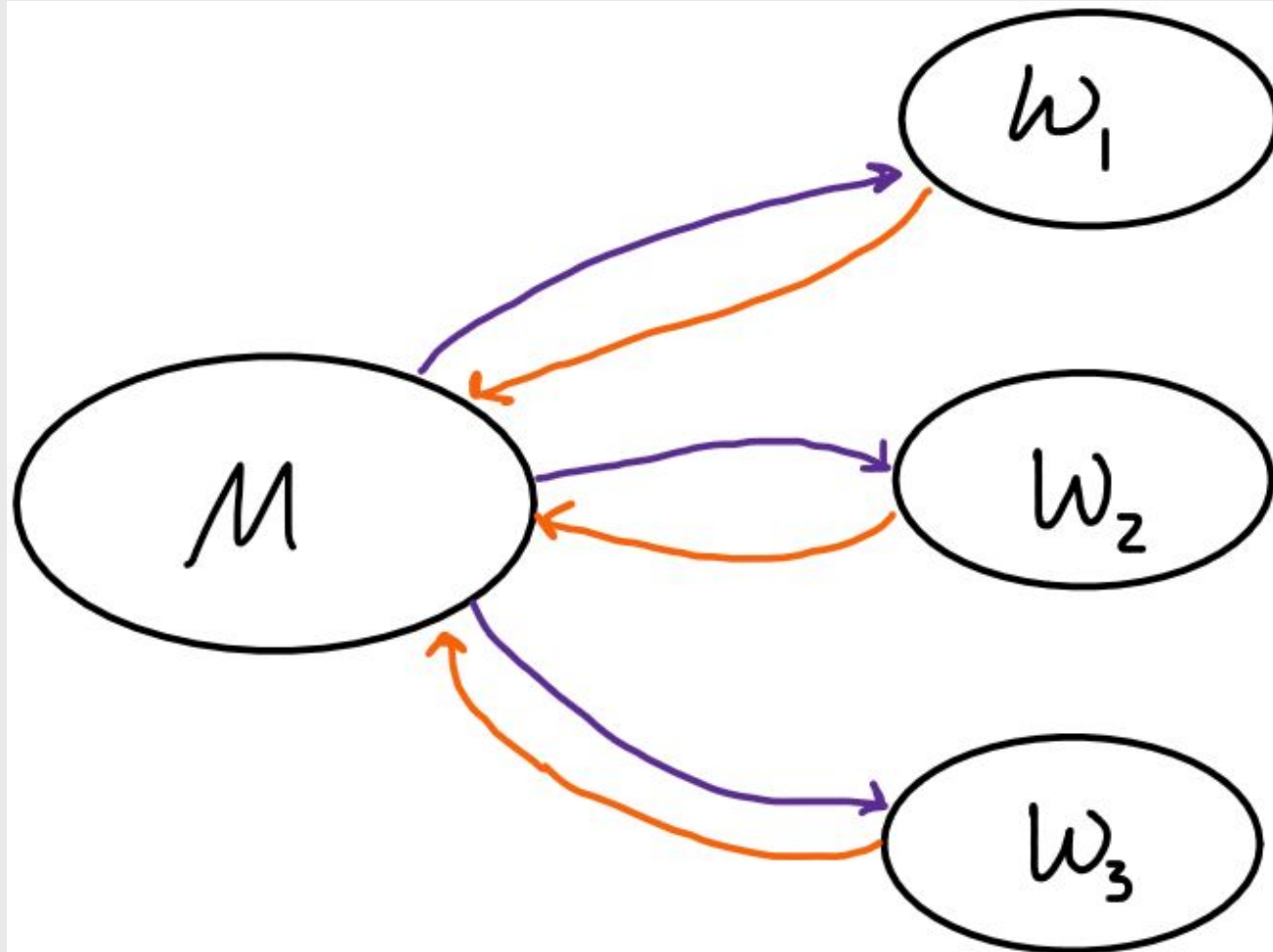
$P_3$

$P_4$

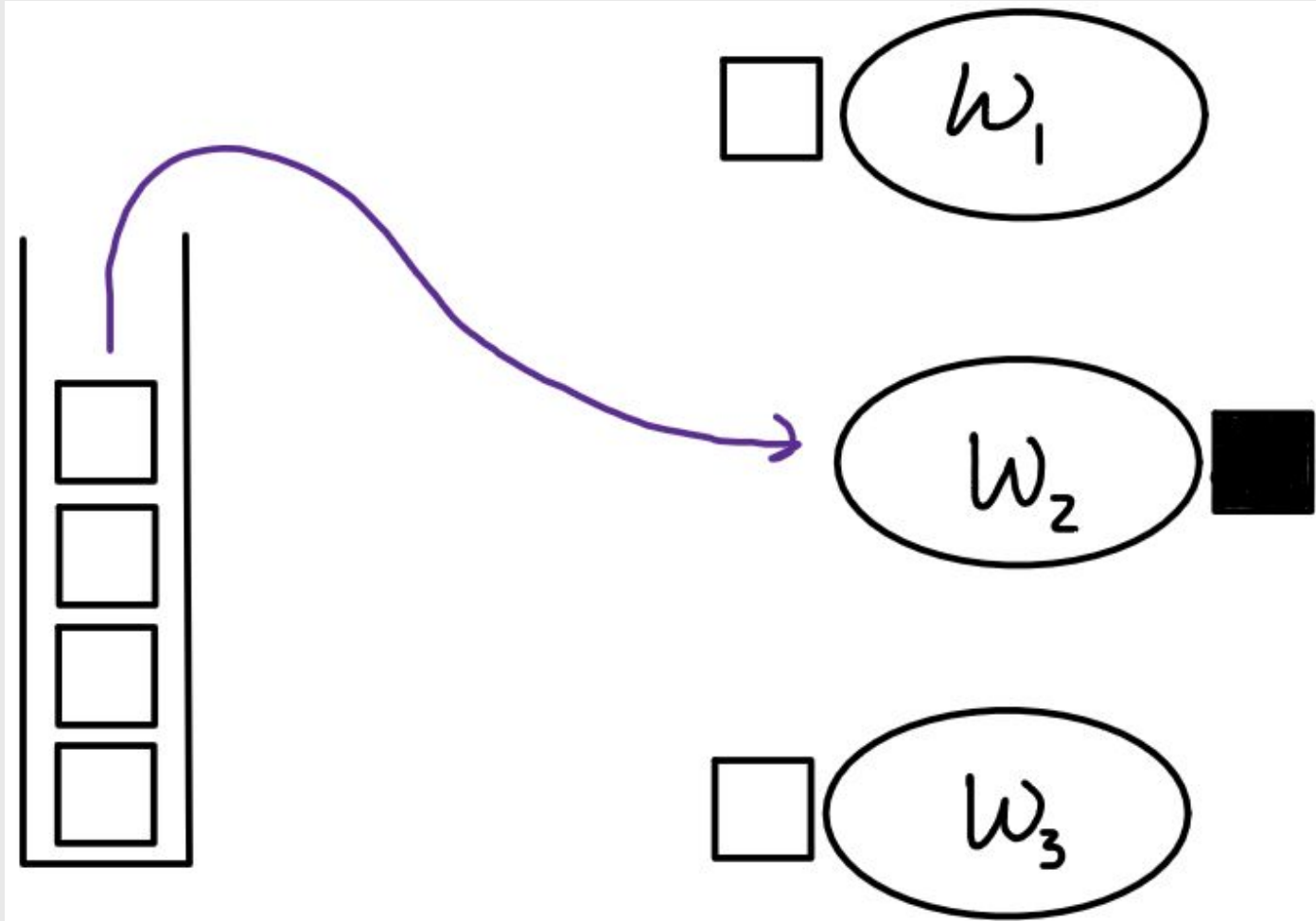
# Algorithms - Standard Task Distribution



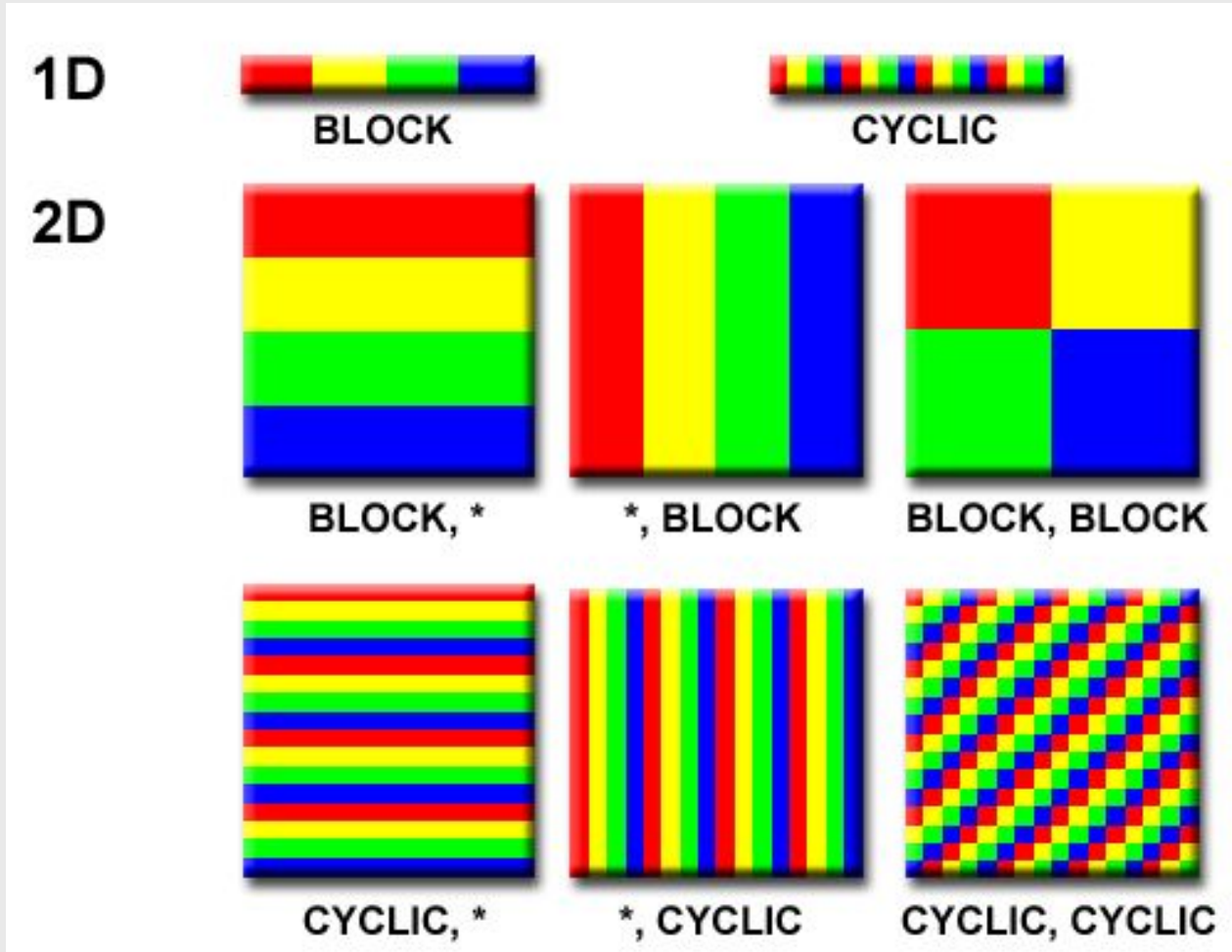
# Algorithms - Manager-Worker



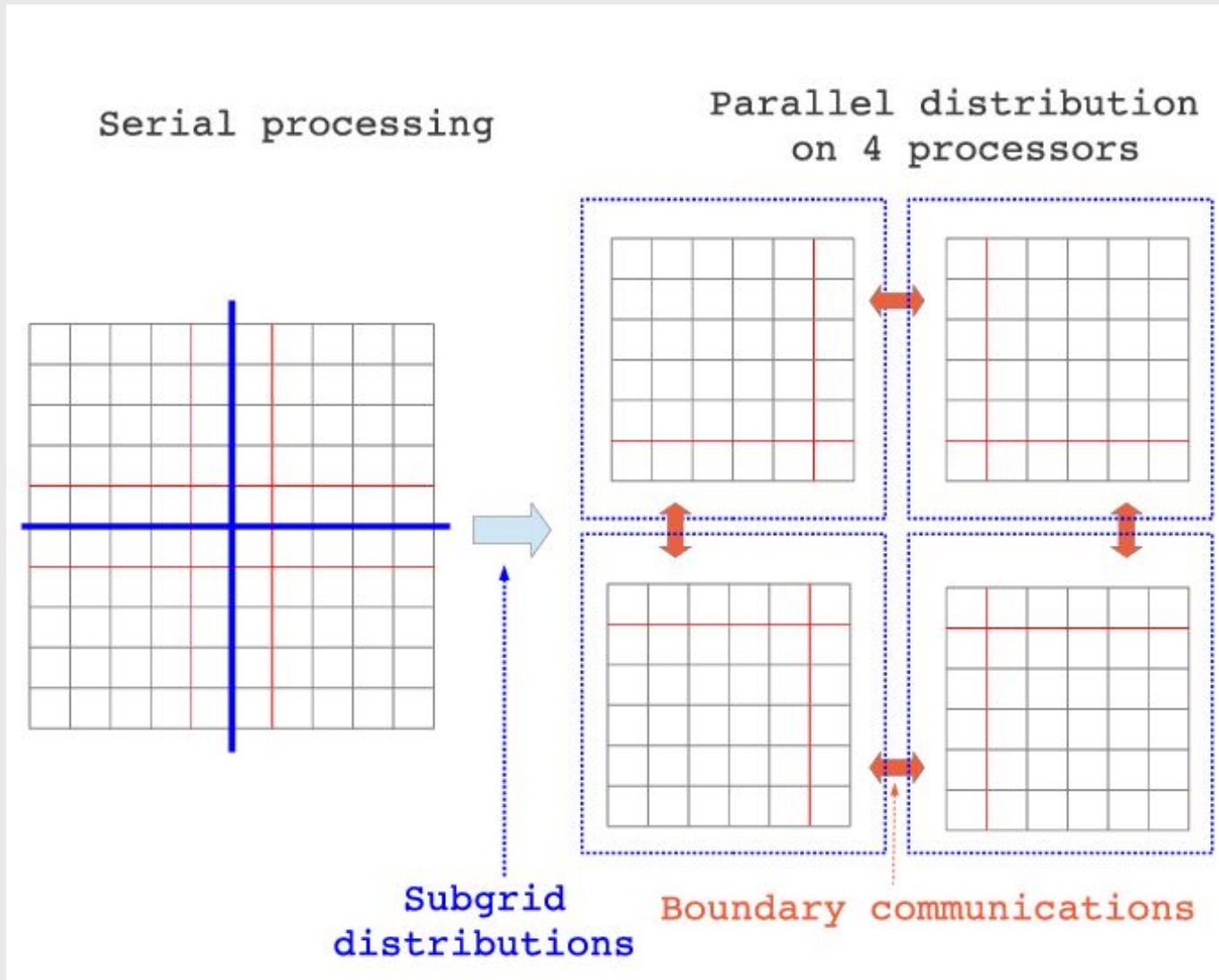
# Algorithms - Stack Allocation



# Algorithms - Domain Decomposition



# Algorithms - Domain Decomposition



**Part 3:**

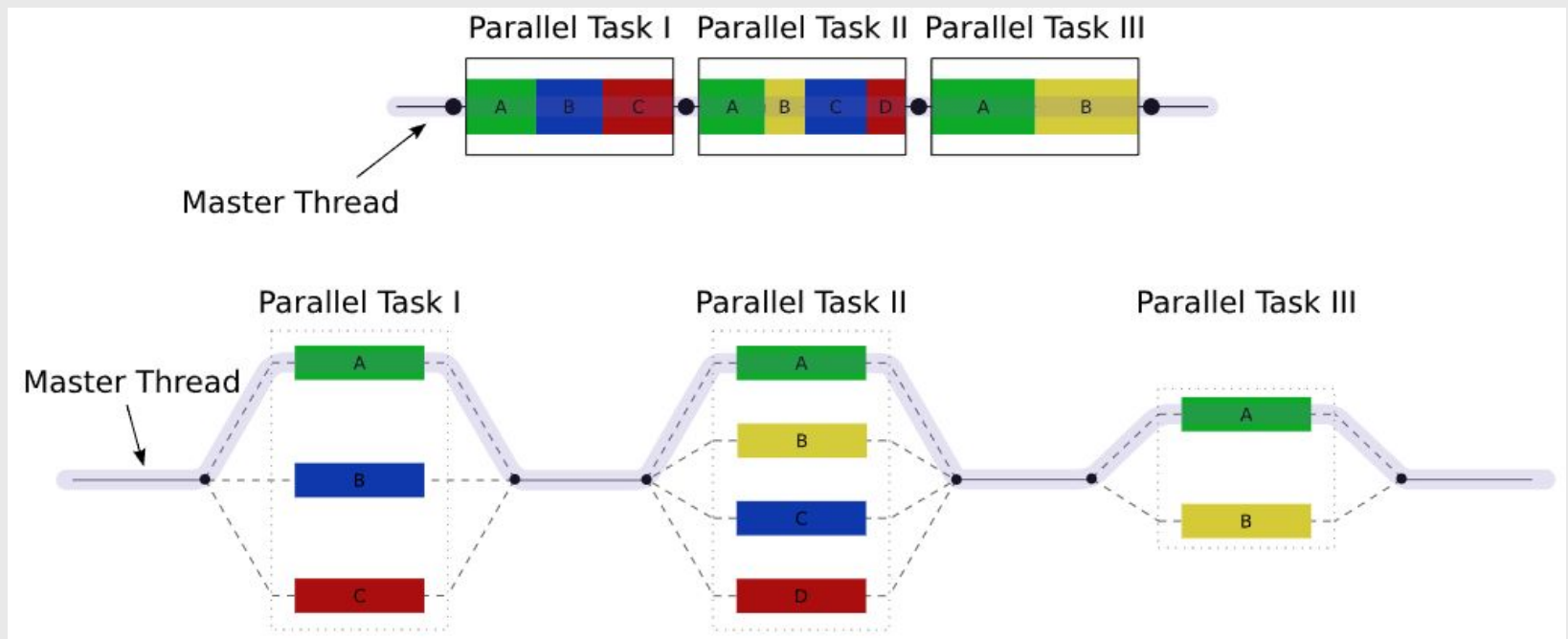
**OpenMP**



# OpenMP

Simplest way to get started with parallelization in C++ is the OpenMP library. Extremely fast to set up and can be used to parallelize anything that would be a **for** loop in a standard program.

Based on a fork-join model, where forks are generated from a primary thread to complete a set number of tasks in parallel.



# OpenMP - Pragma Commands

Most of the commands used by OpenMP are compiler directives of the form

```
#pragma omp construct [clause]
```

The construct we will focus on is the ***parallel for*** construct, which simply parallelizes a ***for*** loop. Some common examples look like:

```
#pragma omp parallel for private (x, y) shared (z)
```

```
#pragma omp parallel for schedule(dynamic)
```

```
#pragma omp parallel for reduction (+:sum)
```

Other constructs which may be useful for more advanced implementation are ***barrier***, ***single***, ***critical***, and ***ordered***.

# OpenMP - Clauses

- **private** - Each thread creates its own private instance of the enclosed variables
- **shared** - All threads share the same value across memory
- **if** - executes the loop in parallel if the condition is satisfied
- **reduction** - reduces a shared variable according to the specified operation {+, -, \*, max, min}
- **schedule** - accepts the kind of schedule {dynamic, static, guided, runtime} as well as the chunk size of iterations (an integer).
- **nowait** - skips the barrier at the end of a parallel region (useful for multiple independent “parallel for” sections in sequence).

# OpenMP “Hello World” Example

```
#include "omp.h"
#include <iostream>

int main()
{
    double sum = 0;
    omp_set_num_threads(6);
    #pragma omp parallel for reduction (+:sum)
    for (int i = 0; i < 1001; i++)
        sum += i;
    std::cout << "Total is " << sum << "\n";
    return 0;
}
```

## OpenMP Advantages

This is by far the simplest way to parallelize any program where the bulk of the work is some repeated process within a **for** loop. The communication and overhead is entirely handled by the library and compiler, and the parallelization is handled by a single line of compiler directives.

It also doesn't require a full rewrite of serial code since parallelization can simply be added through the compiler directive and an “`#include <omp.h>`” statement

OpenMP also allows for dynamic load balancing between nodes on the fly as a program runs by adding “`schedule(dynamic)`” at the end of the directive

## OpenMP Drawbacks

OpenMP is a multithreading technique. As such, it is limited by the number of threads to which the program has access.

- On ODU's HPC cluster, Wahab, each core has 40 nodes, therefore an OpenMP-enabled program can create up to 40 threads at most.

No built-in support for parallel I/O from within parallel threads

Limited to the simplest of parallelization algorithms discussed in the previous section, since the programmer has little control over the communication between threads.

# General Tips for OpenMP Programming

1. Begin with a fully-optimized serial version of the program you wish to parallelize.
2. Add OpenMP in steps and do testing to evaluate the speedup achieved with varying number of threads.
3. If the program uses nested loops, start by parallelizing the outermost loops, since that will *typically* result in the best performance by reducing communication overhead
4. Experiment with different task scheduling.
  - a. In general, if the tasks are of equal time-complexity, static scheduling is fine. If the tasks vary in time-complexity between loop iterations, consider dynamic scheduling with small chunk size
5. Minimize the number of barriers used through a program (if any)

**Part 4:**

**Message-Passing Interface (MPI)**



## Message-Passing Interface (MPI)

MPI is considered the main workhorse for writing parallelized code in C++ and FORTRAN. It is a lower-level parallelization library than OpenMP, and therefore the programmer has to manually control the way in which nodes communicate and pass data.

Unlike OpenMP, MPI is scalable even to the largest architectures currently in use. Instead of being limited to the number of threads available to a single node, MPI program spawn independent processes, which are individually capable of both executing commands and communicating with other processes to transfer data.

## MPI - Advantages

MPI is capable of running on a variety of memory architectures, and is adaptable to heterogeneous hardware implementations.

Applicable to a wider variety of problems than OpenMP and can adapt to various designs to suit the needs of a particular project.

Distributed memory systems are generally cheaper and more accessible than shared memory systems.

## MPI - Drawbacks

It is not only more difficult to write a program with MPI, it is also more difficult in general to adapt a serial program into a parallelized version with MPI. This corresponds to additional development time in order to develop a good parallelized program with MPI even if a serial version already exists.

Debugging MPI program is generally more difficult and there are relatively few resources available for debugging in general.

## MPI - Basic Routines

*MPI\_Init(&argc, &argv)* - Required at the start of the program.

*MPI\_Finalize()* - Required at the end of the program

*MPI\_Comm\_rank(MPI\_COMM\_WORLD, &node)* - identifies the processor currently being used as an integer between 0 and N

*MPI\_Comm\_size(MPI\_COMM\_WORLD, &nproc)* - gives the total number of processes allocated for the program

*MPI\_Send(data, count, MPI\_Datatype, destination, tag, MPI\_COMM)* - blocking communication to send data to another process

*MPI\_Recv(data, count, MPI\_Datatype, source, tag, MPI\_COMM, status)* - blocking communication to receive data from another process

## MPI - Basic Datatypes

*MPI\_INT*

*MPI\_FLOAT*

*MPI\_DOUBLE*

*MPI\_LONG*

*MPI\_CHAR*

## MPI - Additional Routines

*MPI\_Barrier*

*MPI\_Bcast*

*MPI\_Gather*

*MPI\_Scatter*

*MPI\_Reduce*

*MPI\_Scan*

# MPI “Hello World” Example


```
#include <iostream>
#include <mpi.h>

int main(int argc, char** argv)
{
    int node, nproc;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &node);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);

    std::cout << "This is node " << node << "\n";

    MPI_Finalize();
    return 0;
}
```



```
C:\Users\taylo\source\repos\MPI_Basics\x64\Release>mpiexec -n 6 MPI_Basics.exe
This is node 4
This is node 5
This is node 1
This is node 3
This is node 0
This is node 2
```

# MPI Manager-Worker Example

```
/* -----  
Taylor Powell                                April 20, 2022  
Basic MPI Program - Manager-Worker Setup  
-----*/  
  
#include <iostream>  
#include <mpi.h>  
  
int main(int argc, char** argv)  
{  
    int node, nproc, Ntasks = 1000;  
    MPI_Init(&argc, &argv);  
    MPI_Comm_rank(MPI_COMM_WORLD, &node);  
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);  
    MPI_Status status;  
  
    if (nproc < 2) {  
        std::cout << "This application should run at least 2 processes.\n";  
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);  
    }  
}
```

# MPI Manager-Worker Example

```
if (node == 0) {
    int sender, nWorkers = nproc - 1, Nsent = 0;
    double x, sum = 0.0;

    for (int i = 1; i <= (nWorkers < Ntasks ? nWorkers : Ntasks); i++) {
        x = (Nsent + 1) * 1.0;
        MPI_Send(&x, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
        Nsent++;
    }
    for (int i = 0; i < Ntasks; i++) {
        MPI_Recv(&x, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        sum += x;
        sender = status.MPI_SOURCE;
        if (Nsent < Ntasks) {
            x = (Nsent + 1) * 1.0;
            MPI_Send(&x, 1, MPI_DOUBLE, sender, 0, MPI_COMM_WORLD);
            Nsent++;
        }
        else
            MPI_Send(MPI_BOTTOM, 0, MPI_DOUBLE, sender, 1, MPI_COMM_WORLD);
    }
    std::cout << "The total is " << sum << "\n";
}
else {
    if (node < Ntasks) {
        double x;
        MPI_Recv(&x, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        while (status.MPI_TAG == 0) {
            MPI_Send(&x, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
            MPI_Recv(&x, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        }
    }
}
MPI_Finalize();
return 0;
}
```



## Further Reading

### **OpenMP:**

<https://www.openmp.org/>

Lawrence Livermore tutorial series, <https://hpc-tutorials.llnl.gov/openmp/>

Tim Mattson's series on YouTube, [YouTube Link](#)

### **MPI:**

<https://www.open-mpi.org/>

Wes Kendall's tutorials, <https://mpitutorial.com/tutorials/>

Lawrence Livermore tutorial, <https://hpc-tutorials.llnl.gov/mpi/>

*Sample code used in this lecture is available on my GitHub at*

[https://github.com/Taylor-Powell/OpenMP\\_Basics](https://github.com/Taylor-Powell/OpenMP_Basics)

[https://github.com/Taylor-Powell/MPI\\_Basics](https://github.com/Taylor-Powell/MPI_Basics)