The background of the slide is a grayscale illustration of a circuit board. It features a network of black lines representing traces, with several circular pads or vias. A solid black horizontal band runs across the middle of the image, serving as a backdrop for the text. The top and bottom portions of the image show the circuit traces and pads in more detail.

Distributed Memory Concepts

Distributed Multiprocessing

Title and Content Layout

- Distributed Memory Concepts: Distributed Multiprocessing - An Analogy
- Distributed Parallelism
- MPI
-

Distributed Memory Concepts: Distributed Multiprocessing - Analogies

- Analogies:
- Painters have their own brushes, palettes, and canvases. All the canvases put together on the designated big wall to make a big picture. We can think the canvases as a set of puzzle pieces that will be put on the big wall to solve the puzzle which is the big picture. Additionally, each painter has a private assistant to carry, to communicate with other private assistants, and to arrange the canvas on the big wall. They work together simultaneously at the same time.
- Fishers have their own fishing poles. If only one person fishes at the lake, the fisher can only get some fishes. If many people fish together at the same lake on the same time, they can get many more fishes.

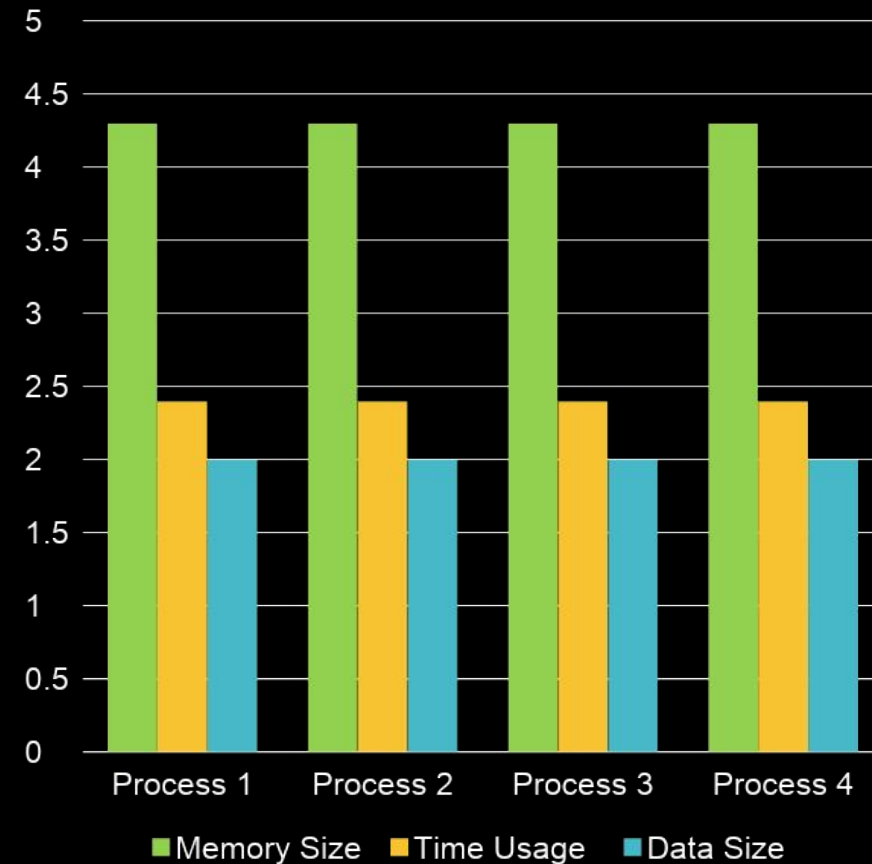
Distributed Multiprocessing

A big job divided into smaller jobs.
Each smaller job assigned to a processor.

All processors work simultaneously to do their jobs.

The communications among the processors will be done through network pipelines.

The more processors work simultaneously, the bigger job they can complete at less time.



What Is Parallelism?

Parallelism is the use of multiple processing units – either processors or parts of an individual processor – to solve a problem, and in particular the use of multiple processing units operating concurrently on different parts of a problem.

The different parts could be different tasks, or the same task on different pieces of the problem's data.

Kinds of Parallelism

- Instruction Level Parallelism
- Shared Memory Multithreading
- Distributed Memory Multiprocessing
- GPU Parallelism
- Hybrid Parallelism (Shared + Distributed + GPU)

Why Parallelism Is Good?

- **The Trees**: We like parallelism because, as the number of processing units working on a problem grows, we can solve the same problem in less time.
- **The Forest**: We like parallelism because, as the number of processing units working on a problem grows, we can solve bigger problems.

Parallelism Jargon

- Threads are execution sequences that share a single memory area (“address space”)
- Processes are execution sequences with their own independent, private memory areas

... and thus:

- Multithreading: parallelism via multiple threads
- Multiprocessing: parallelism via multiple processes

Generally:

- Shared Memory Parallelism is concerned with threads, and
- Distributed Parallelism is concerned with processes.

Jargon Alert!

In principle:

- “shared memory parallelism” □ “multithreading”
- “distributed parallelism” □ “multiprocessing”

In practice, sadly, these terms are often used interchangeably:

- Parallelism
- Concurrency (not as popular these days)
- Multithreading
- Multiprocessing

Typically, you have to figure out what is meant based on the context.

Load Balancing

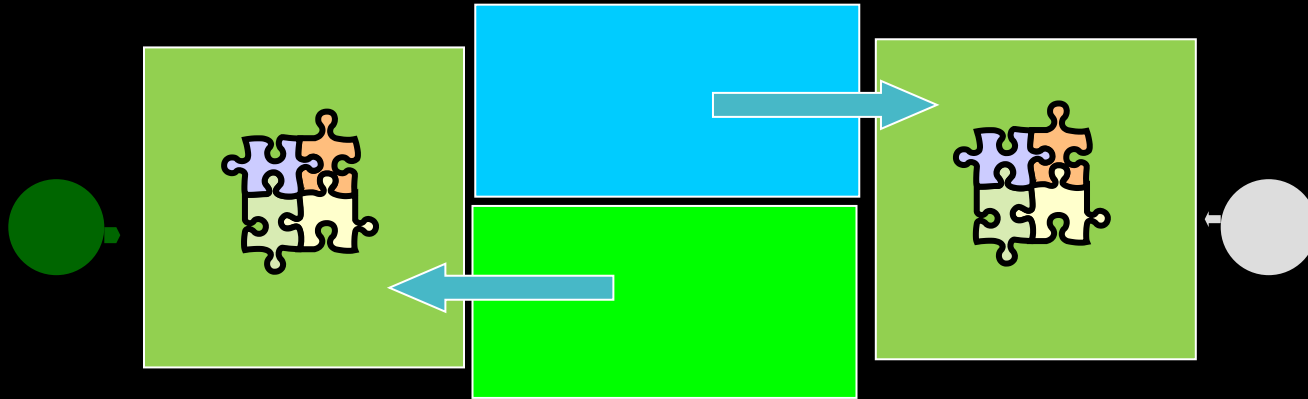
Suppose you have a distributed parallel code, but one process does 90% of the work, and all the other processes share 10% of the work.

Is it a big win to run on 1000 processes?

Now, suppose that each process gets exactly $1/N_p$ of the work, where N_p is the number of processes.

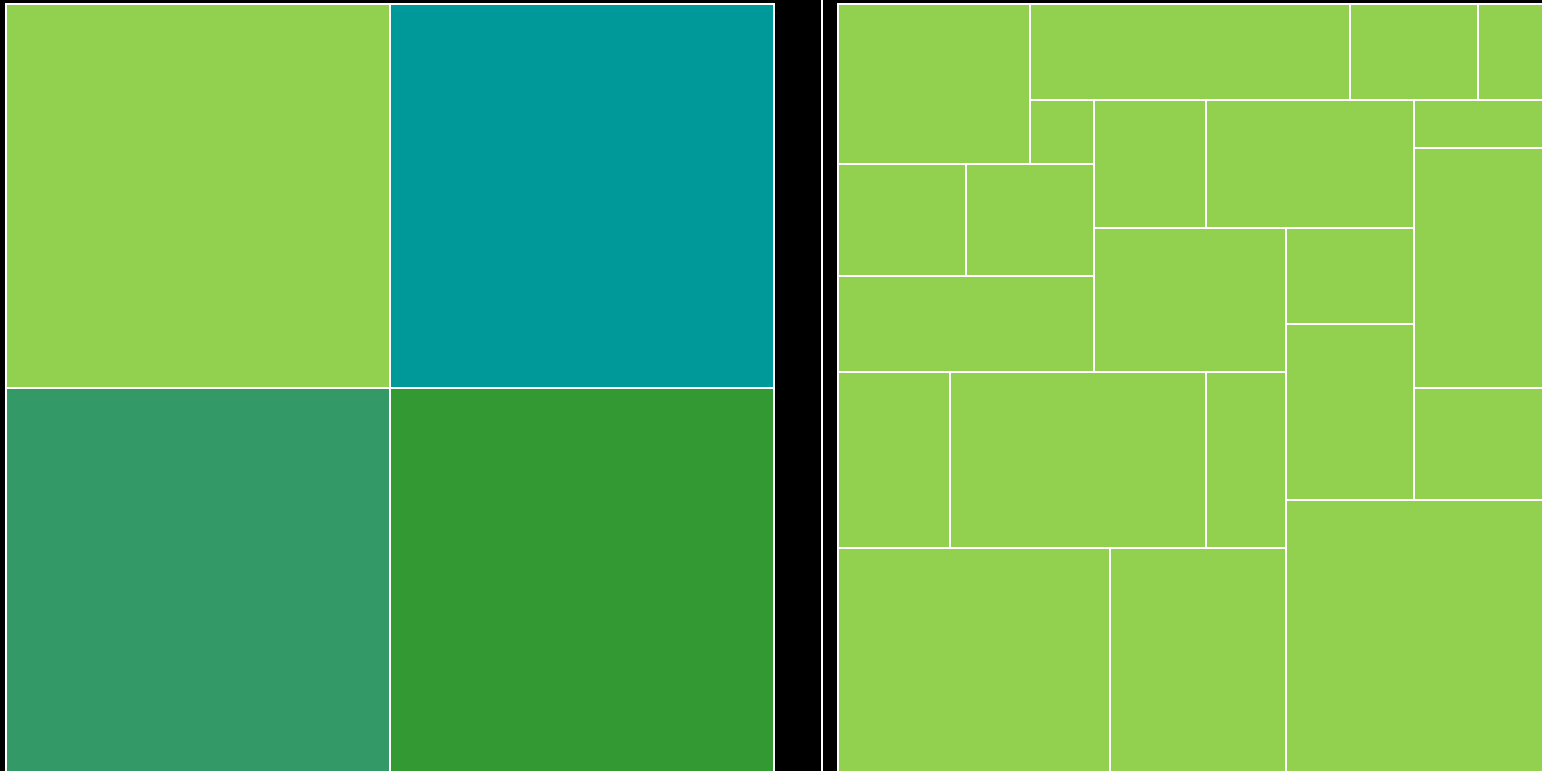
Now is it a big win to run on 1000 processes?

Load Balancing



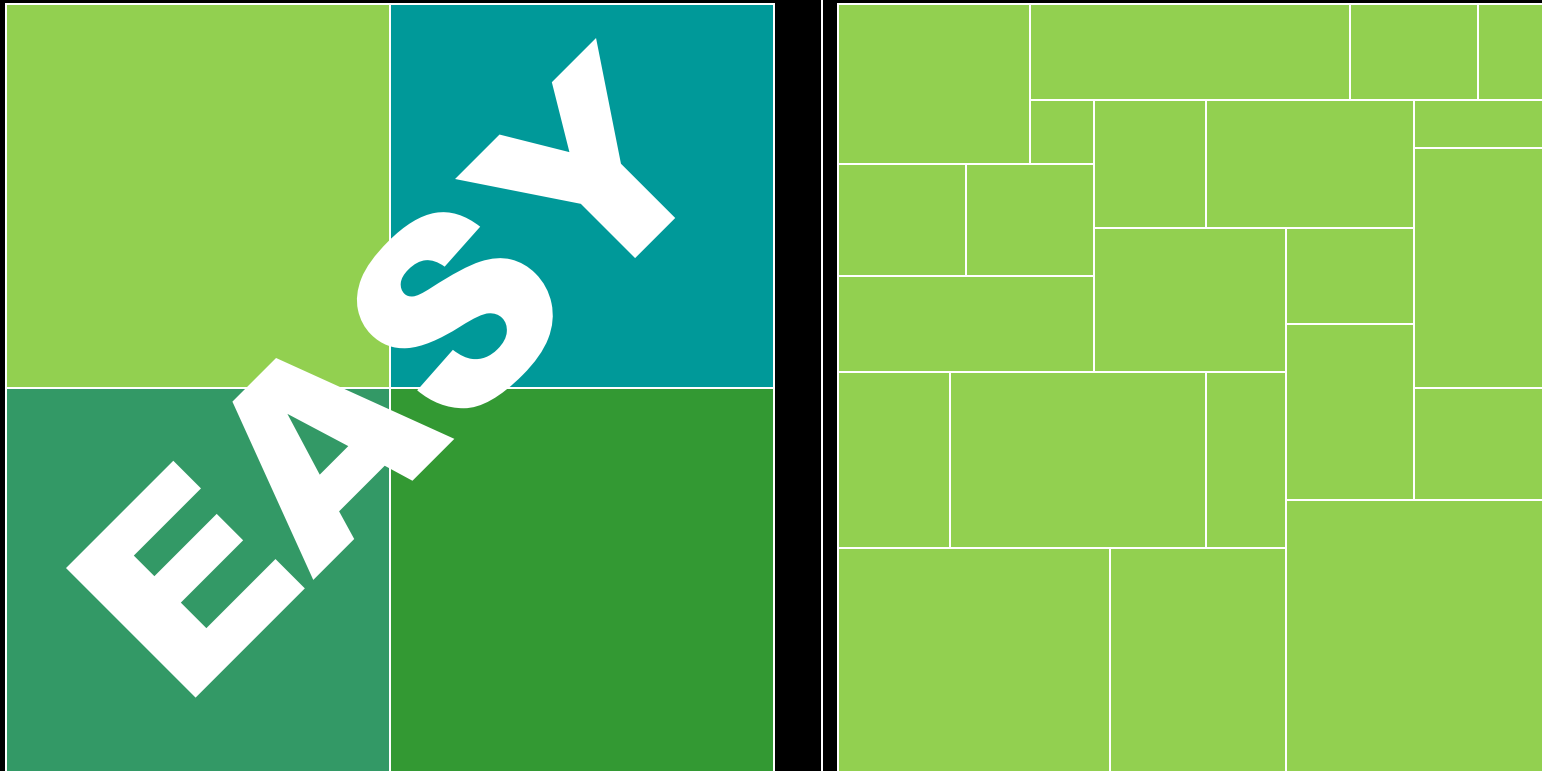
Load balancing means ensuring that everyone completes their workload at roughly the same time.

Load Balancing



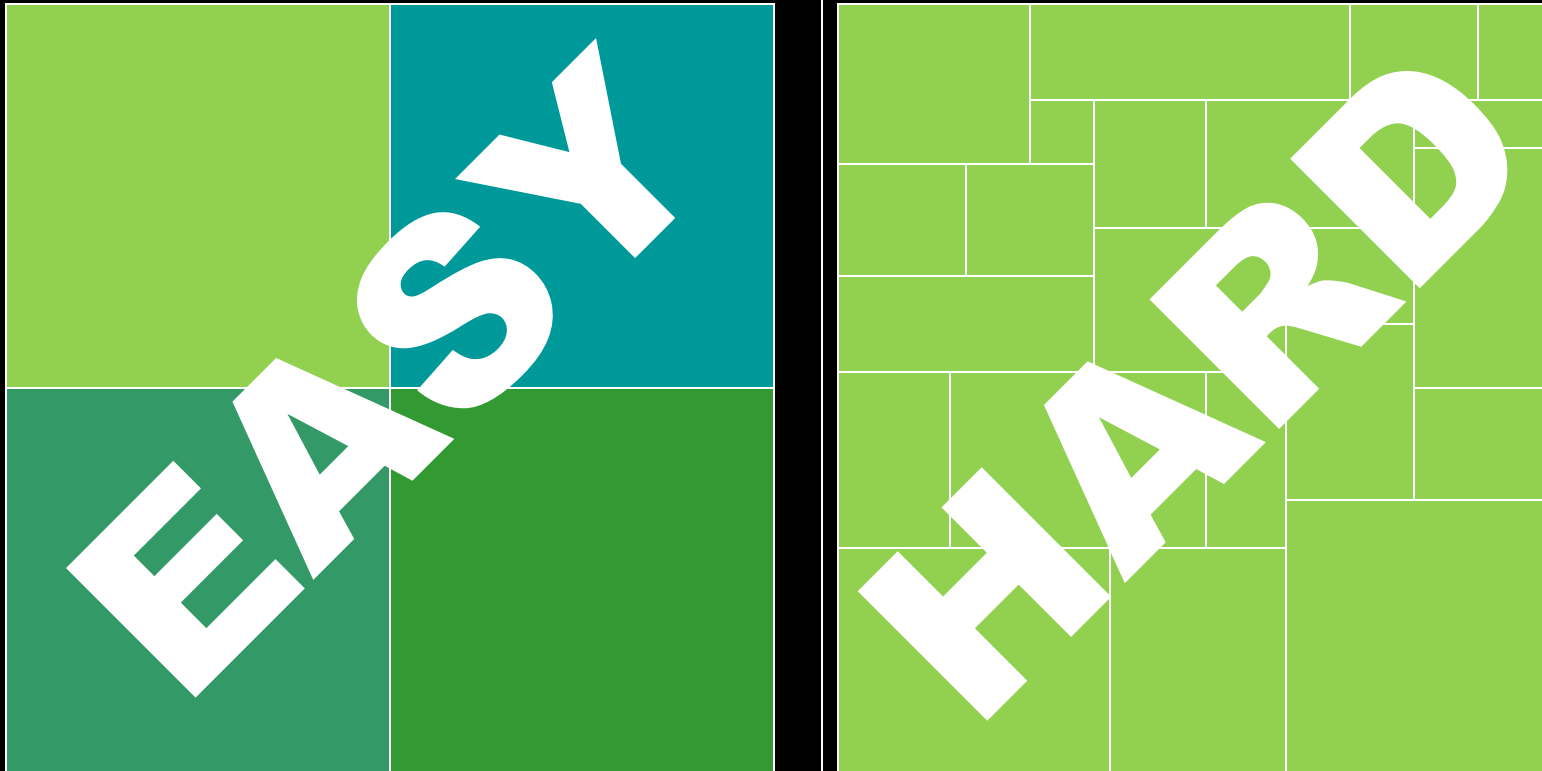
Load balancing can be easy, if the problem splits up into chunks of roughly equal size, with one chunk per processor. Or load balancing can be very hard.

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.

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.

Load Balancing Is Good

When every process gets the same amount of work, the job is load balanced.

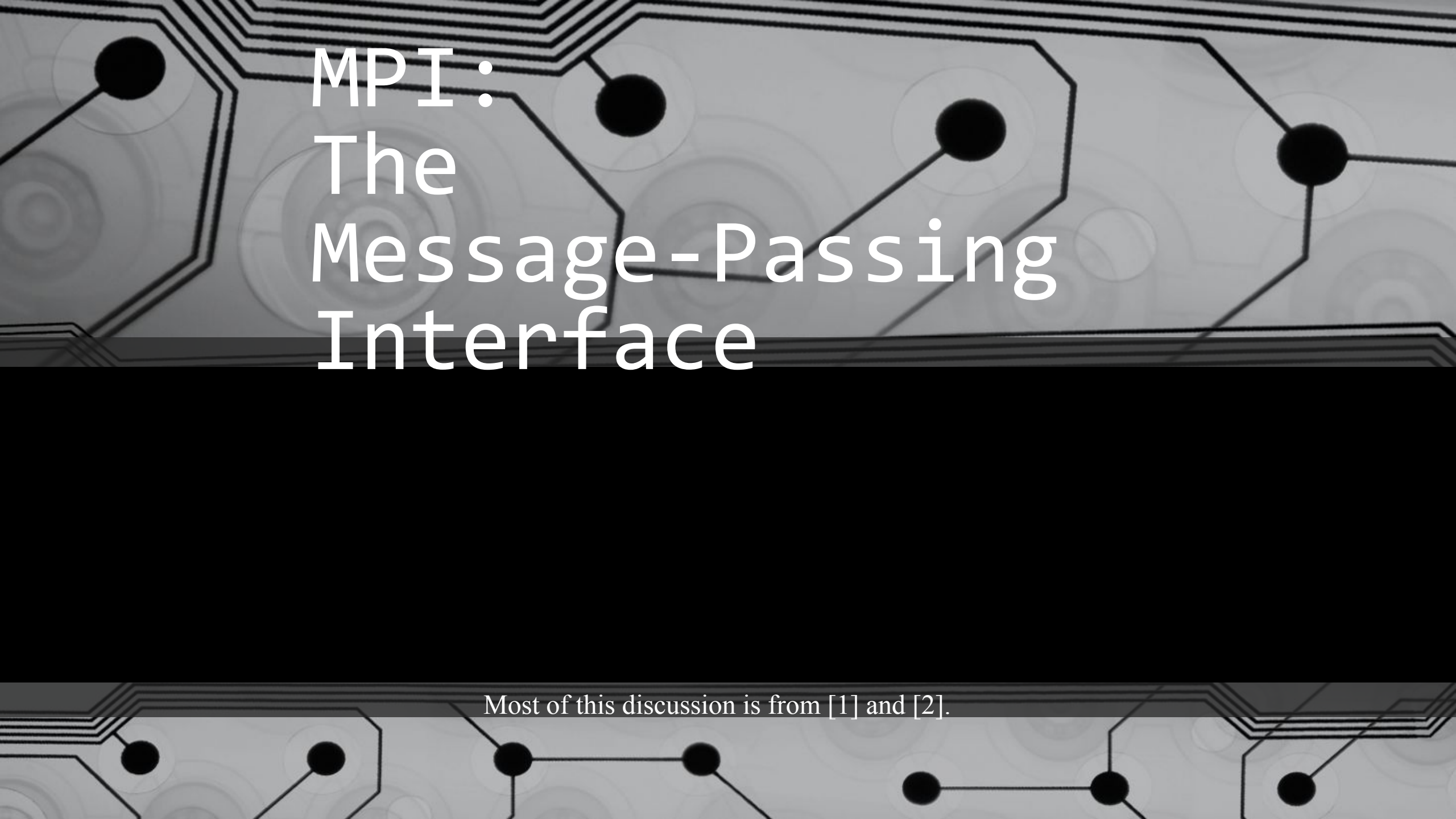
We like load balancing, because it means that our speedup can potentially be linear:
if we run on N_p processes, it takes $1/N_p$ as much time as on one.

For some codes, figuring out how to balance the load is trivial (for example, breaking a big unchanging array into sub-arrays).

For others, load balancing is very tricky (for example, a dynamically evolving collection of arbitrarily many blocks of arbitrary size).

Parallel Strategies

- **Client-Server**: One worker (the server) decides what tasks the other workers (clients) will do; for example, Hello World, Monte Carlo.
- **Data Parallelism**: Each worker does exactly the same tasks on its unique subset of the data; for example, distributed meshes for transport problems (weather etc.).
- **Task Parallelism**: Each worker does different tasks on exactly the same set of data (each process holds exactly the same data as the others); for example, N-body problems (molecular dynamics, astrophysics).
- **Pipeline**: Each worker does its tasks, then passes its set of data along to the next worker and receives the next set of data from the previous worker.

The background of the slide is a grayscale image of a circuit board. It features a complex network of black lines representing traces, with several large black circular pads or vias. The pattern is symmetrical and geometric, typical of high-speed digital circuitry.

MPI: The Message-Passing Interface

Most of this discussion is from [1] and [2].

What Is MPI?

The Message-Passing Interface (MPI) is a standard for expressing distributed parallelism via message passing.

MPI consists of a header file, a library of routines and a runtime environment.

When you compile a program that has MPI calls in it, your compiler links to a local implementation of MPI, and then you get parallelism; if the MPI library isn't available, then the compile will fail.

MPI can be used in Fortran, C and C++.

MPI Calls

MPI calls in Fortran look like this:

```
CALL MPI_Funcname(..., mpi_error_code)
```

In C, MPI calls look like:

```
mpi_error_code = MPI_Funcname(...);
```

In C++, MPI calls look like:

```
mpi_error_code = MPI::Funcname(...);
```

Notice that `mpi_error_code` is returned by the MPI routine `MPI_Funcname`, with a value of `MPI_SUCCESS` indicating that `MPI_Funcname` has worked correctly.

MPI is an API

MPI is actually just an Application Programming Interface (API).

An API specifies what a call to each routine should look like, and how each routine should behave.

An API does not specify how each routine should be implemented, and sometimes is intentionally vague about certain aspects of a routine's behavior.

Each platform has its own MPI implementation.

WARNING!

In principle, the MPI standard provides bindings for:

- C
- C++
- Fortran 77
- Fortran 90

In practice, you should do this:

- To use MPI in a C++ code, use the C binding.
- To use MPI in Fortran 90, use the Fortran 77 binding.

This is because the C++ and Fortran 90 bindings are less popular, and therefore less well tested.

Example MPI Routines

- `MPI_Init` starts up the MPI runtime environment at the beginning of a run.
- `MPI_Finalize` shuts down the MPI runtime environment at the end of a run.
- `MPI_Comm_size` gets the number of processes in a run, N_p (typically called just after `MPI_Init`).
- `MPI_Comm_rank` gets the process ID that the current process uses, which is between 0 and $N_p - 1$ inclusive (typically called just after `MPI_Init`).

Example: Greetings

1. Start the MPI system.
2. Get the rank and number of processes.
3. If you're not the server process:
 1. Create a greeting string.
 2. Send it to the server process.
4. If you are the server process:
 1. For each of the client processes:
 1. Receive its greeting string.
 2. Print its greeting string.
5. Shut down the MPI system.

helloworldmpi.c

```
    include <mpi.h>

#include <stdio.h>

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

    printf("Hello, World!\n");

    MPI_Finalize();

    return 0;
```


Acknowledgements

1) This module adopted this slide:

[Presentation #6: Distributed Memory Parallelism](#)

2) The example code from this website:

<http://www.shodor.org/petascale/materials/UPModules/AreaUnderCurve/>

3) mmm

References

- [1] P.S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann Publishers, 1997.
- [2] W. Gropp, E. Lusk and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd ed. MIT Press, 1999.