CMOR 421/521
Distributed parallelism

Jesse Chan

What is distributed parallelism?

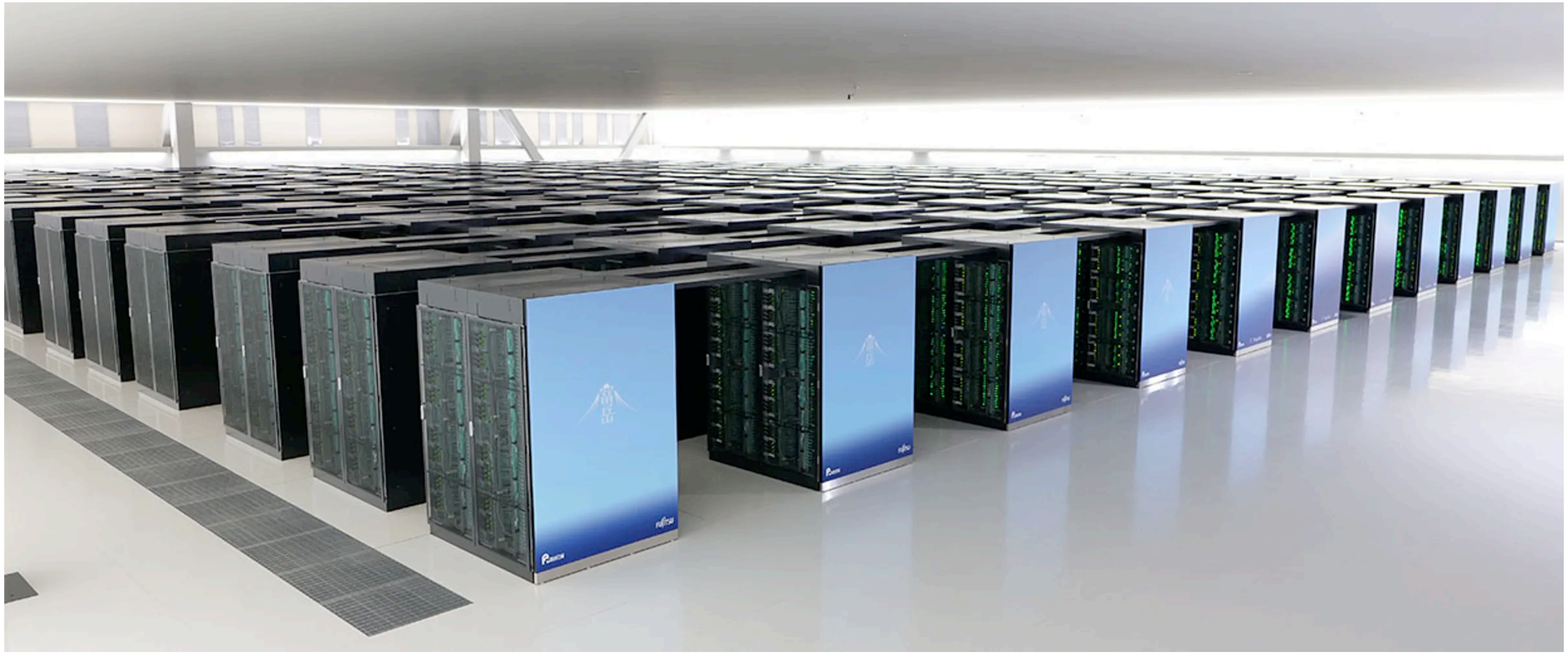
What is distributed parallelism?

And why do we need it?

- Physical limitations of hardware motivate different types of parallelism; shared memory parallelism overcomes limitations of single-core CPUs.
- Distributed memory parallelism addresses limitations of multi-core CPUs.
  - More cores increases cost of synchronization (cache coherency protocols).
  - Modern CPUs are still adding more cores, but are moving to alternative architectures (GPUs) to deal with scaling issues.
  - Memory: some problems are so large they need to be distributed across multiple processors just to be represented.

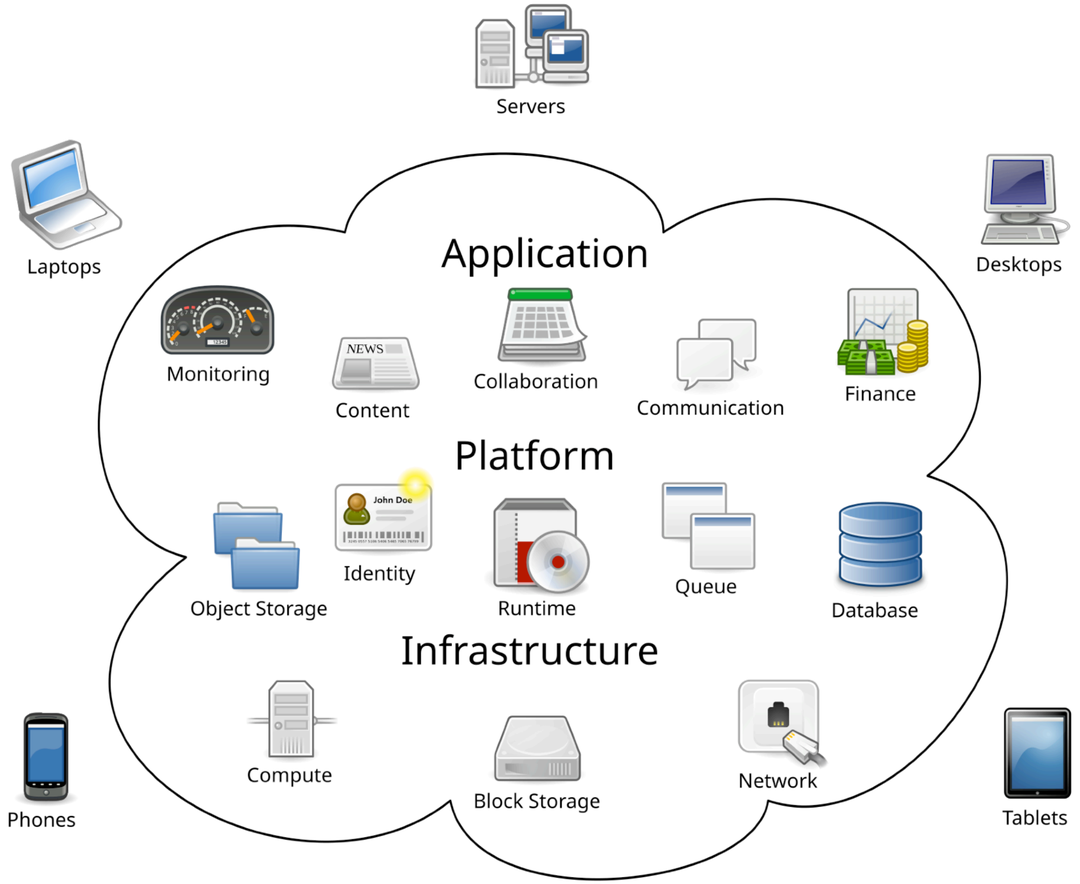
Some examples of distributed parallel machines

Japan’s Fugaku supercomputer



Some examples of distributed parallel machines

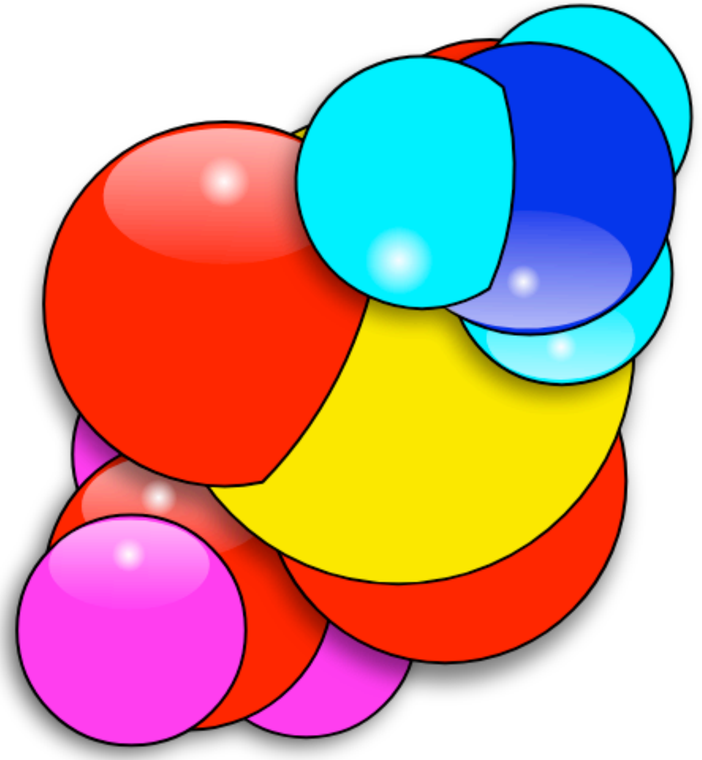
- Cloud computing is one type of distributed computing.
- Offload your main computations to the cloud (some distributed machine), communicate the result back through the internet.



Some examples of distributed parallel machines

Folding @ home

- Started in 2000, distributes work in molecular dynamics simulations to participating home computers as background processes.
- Technically became the first exascale “machine” during COVID.
- Highly heterogeneous computing: old/new CPUs, laptop/gaming GPUs.



Some examples of distributed parallel machines

B

Some examples of distributed parallel machines

Beowulf (LAN-networked) clusters

- Local area network (LAN) connections could be used to build a “Beowulf” cluster for cheap parallel computing
- Linux, MPI-based.
- Single server node + client nodes
- “In most cases, client nodes in a Beowulf system are dumb, the dumber the better.”

Communication-based parallelism

- Distributed parallelism is more flexible since it doesn't depend on hardware.
  - Allows for highly heterogeneous architectures (CPUs vs GPUs)
  - The internet is a communication-based distributed network for file-sharing.
- How is distributed parallelism different? Independent workers use explicit **communication** to share information, synchronize, etc.
  - Can also be used for shared memory parallelism.
- How do we send messages between independent workers? Typically using the **Message Passing Interface (MPI)**.

Intro to the Message Passing Interface (MPI)

Message-passing interface (MPI)

- MPI is a standard (similar to the BLAS *specification* or the OpenMP *standard*)
  - BLAS is implemented by multiple libraries, OpenMP is implemented by different compilers.
- MPI also has two popular implementations: OpenMPI and MPICH
  - MPICH is the original implementation by Argonne National Lab, and implements every routine in the MPI specification.
  - OpenMPI is newer and doesn't implement every MPI specification feature, but can be easier to install and more performant.

Installing MPI

- On Mac, “brew update” then “brew install openmpi” should work
- OpenMPI and MPICH are not supported on Windows. Alternatives:
  - Use VSCode and SSH to directly access NOTSx
  - Install OpenMPI or MPICH through Cygwin or WSL.
  - Install MS-MPI. Note: I don’t have experience with this.

Hello world with MPI

- “MPI_Init” creates the MPI environment. It optionally takes in “argc” and “argv” for command-line input arguments.
- “MPI_Comm_rank” and “MPI_Comm_size” query MPI for the “rank” (current process) and “size” (total number of processes)
  - The MPI communicator (the list of ranks to communicate among) is “MPI_COMM_WORLD” (all ranks)

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

using namespace std;

int main(){

  MPI_Init(NULL, NULL);

  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  cout << "Hello world on rank " << rank
    << " of " << size << endl;

  MPI_Finalize();
  return 0;
}
```

hello_world_mpi.cpp

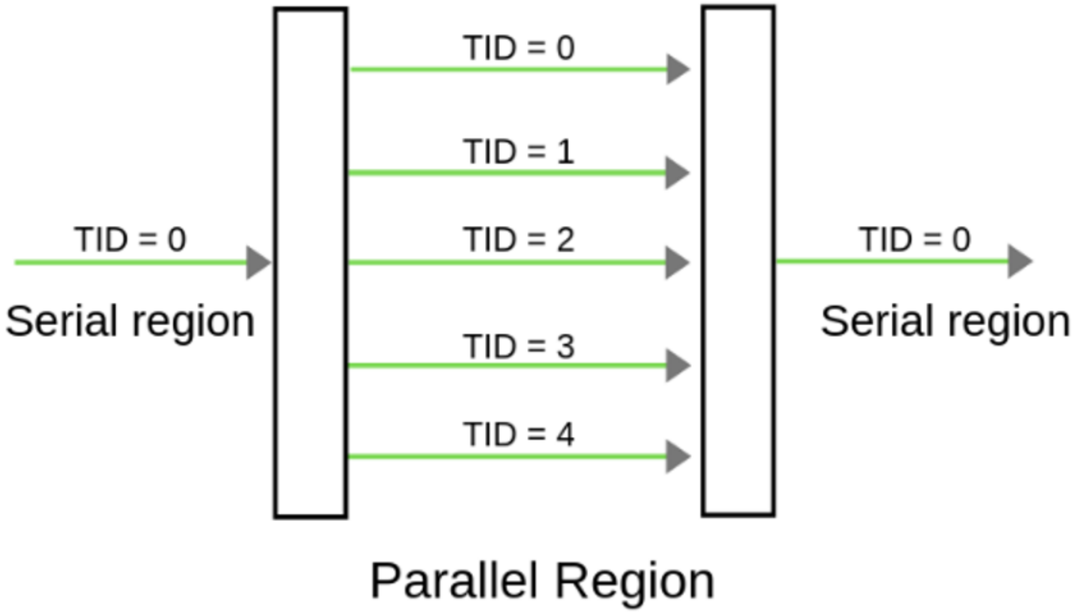
Running MPI “hello world”

- To compile, use “mpic++” to create an executable.
- To run your executable, you need to launch the program using “mpiexec” or “mpirun”
  - Just running the executable will execute with one single rank
  - E.g., “mpirun -n 4 ./a.out” will run “a.out” with 4 ranks.

```
(base) jchan985@XXH62CK9GR MPI % mpic++ hello_world_mpi.cpp
ld: warning: dylib (/opt/homebrew/Cellar/open-mpi/5.0.7/lib/
libmpi.dylib) was built for newer macOS version (15.0) than
being linked (14.0)
(base) jchan985@XXH62CK9GR MPI % ./a.out
Hello world on rank 0 of 1
(base) jchan985@XXH62CK9GR MPI % mpirun -n 2 ./a.out
Hello world on rank 1 of 2
Hello world on rank 0 of 2
(base) jchan985@XXH62CK9GR MPI %
```

Differences between OpenMP and MPI

- OpenMP's threading model is "top down" where you can see all serial/ parallel sections.
- MPI's model is "bottom up", where each rank corresponds to a single thread ID.
  - The code executes from the perspective of a single parallel process, e.g., all workers will execute the same MPI code.



Sending and receiving data via MPI

- No shared memory in MPI.
- MPI routines are typically fairly low-level and C-like (lots of pointer usage and thinking about memory layout).
- Here, we use “MPI_Send” to send information from rank 0 to rank 3.
- Rank 3 uses “MPI_Recv” to receive that information.

```
MPI_Status status;
int x;
if (rank == 0)
{
  x = 123;
  MPI_Send(&x, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);
}
else if (rank == 3)
{
  MPI_Recv(&x, 1, MPI_INT, 0, MPI_ANY_TAG,
           MPI_COMM_WORLD, &status);
}
else
{
  x = -1;
}
```

What are the arguments of MPI_Send?

```
int MPI_Send(
    const void* buffer,
    int count,
    MPI_Datatype data_type,
    int dest_rank,
    int tag,
    MPI_Comm comm
);
```

Argument	Description
buffer	Pointer to the data to be sent
count	How many entries to send
data_type	Data type of the data to be sent
dest_rank	The rank to send to
tag	Identifier for the message
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)

```
x = 123;
MPI_Send(&x, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);
```

What are the arguments of MPI_Recv?

```
int MPI_Recv(
    void* buffer,
★   int count,
    MPI_Datatype data_type,
    int src_rank,
    int tag,
    MPI_Comm comm,
★   MPI_Status* status
);
```

Argument	Description
buffer	Pointer to where to copy the data
count	**AT MOST** how many entries to read
data_type	Data type of the data to be sent
src_rank	The rank to receive from
tag	Identifier for the message
comm	The communicator that the ranks are in (MPI_COMM_WORLD for us)
status	Information about the message

```
MPI_Recv(&x, 1, MPI_INT, 0, MPI_ANY_TAG,
         MPI_COMM_WORLD, &status);
```

MPI has several pre-defined data types

Primitive Type	MPI Data Type
int	MPI_INT
long long int	MPI_LONG_LONG
float	MPI_FLOAT
double	MPI_DOUBLE
char	MPI_BYTE

What does an instance of MPI_Status store?

Fields	Description
int count	Number of *received* entries
int cancelled	Was the request cancelled?
int MPI_SOURCE	Source rank
int MPI_TAG	Tag value
int MPI_ERROR	Any errors associated with the message

If we don’t care about the status:

```
MPI_STATUS_IGNORE
```

Sending and receiving an array via MPI

- What will the following code return for n=10?

```
for (int i = 0; i < n; ++i)
{
  x[i] = (double)rank + i;
}
```

```
if (rank == 0)
{
  MPI_Send(x, n / 2, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
}
else if (rank == 1)
{
  MPI_Recv(x + n / 2, n / 2, MPI_DOUBLE, 0, MPI_ANY_TAG,
           MPI_COMM_WORLD, &status);
  cout << "on rank " << rank << endl;
  print_vector(n, x);
}
```

Sending and receiving an array via MPI

- What will the following code return for n=10?

```
for (int i = 0; i < n; ++i)
{
  x[i] = (double)rank + i;
}
```

  - Initialize a vector's values to "0, 1, 2, 3, …" plus the current rank.
  - Sends the second half of a vector from rank 0 to rank 1.
  - Should return:
    x = [1, 2, 3, 4, 5, 0, 1, 2, 3, 4]

```
if (rank == 0)
{
  MPI_Send(x, n / 2, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
}
else if (rank == 1)
{
  MPI_Recv(x + n / 2, n / 2, MPI_DOUBLE, 0, MPI_ANY_TAG,
           MPI_COMM_WORLD, &status);
  cout << "on rank " << rank << endl;
  print_vector(n, x);
}
```