

Introduction to Parallel Programming

Shaohao Chen

Office of Research Computing and Data (ORCD)

MIT

ORCD Overview

- Who are we?

- Long history of supporting the computational and data needs of MIT
- Official office status launched in September 2022

- What do we provide?

- Sizable, shared base computing and data resources and services
- Training and support
- Additional direct charge services
 - Purchasing and hosting compute resources
 - Storage



+off campus operations and remote support team

<https://orcd.mit.edu>

Outline of the course

Day 1

- Preliminary parallel programming
- OpenMP
- MPI

Day 2

- GPU basics
- CUDA
- Parallel and distributed deep learning

Preliminary Parallel Programming

Outline

- Basics of HPC
- Access to ORCD clusters
- Optimize programs
- Embarrassingly parallel
- Parallel computing: shared memory, distributed memory

What is HPC?

- High Performance Computing (HPC) refers to the practice of aggregating computing power in order to solve large problems in science, engineering, or business.
- Similar terminology: supercomputing.
- The purpose of HPC: accelerate computer programs and thus accelerate work processes.
- HPC cluster: A set of connected computers that work together. Computers are connected with high-speed network. They can be viewed as a single system.
- Parallel computing: many computations are carried out simultaneously, typically computed on a computer cluster.
- Parallel programming: MPI, OpenMP, CUDA.

General-purpose HPC

- More and more non-traditional HPC workloads.
- Artificial intelligence (AI) training as well as compute and data-driven analytics.
- Computational demands of deep learning applications: GPUs, large memory, fast I/O.
- General-purpose HPC refers to any applications designed to run a given workload as fast as the hardware will allow. The hardware stack can be CPU, memory, storage, network, GPU, PCI, a single node, or multiple nodes on a computer cluster.
- “The convergence of AI, data analytics and traditional simulation will result in systems with broader capabilities and configurability as well as cross pollination.” -- Dr. Al Gara, Intel

Reference: <https://hpcng.org/>

Basic structure of an HPC cluster

- Cluster – a collection of many computers/nodes.
- Rack – a closet to hold a bunch of nodes.
- **Node** – a computer (with processors, memory, hard drive, etc.)
- Socket/processor – one multi-core processor.
- **Core**/processor – one processing unit.
- Hyperthread: virtual (logical) core
- **Network devices**
- **Storage system**
- Power supply system
- Cooling system

Computer Clusters in MGHPCC



Inside a node

- **CPU** (e.g. multi-core processors)

To carry out program instructions. Built-in cache (fast memory).

- **Memory** (RAM)

Fast but temporary storage, to store data for immediate use.

- **Hard drives**

Relatively slow but permanent storage, to store data permanently.

- **Network devices** (e.g. Ethernet, Infiniband)

To transfer data between nodes or between sites.

- **Accelerator** (e.g. GPU)

To accelerate programs with parallel computing techniques.

Access to ORCD clusters

- Get started: <https://orcd-docs.mit.edu/getting-started/>

- Log in Engaging `ssh <user>@eofe10.mit.edu`

- Download slides and codes

```
git clone https://github.com/mit-orcd/parallel-programming.git
```

- Work on CPUs

```
srun -t 120 -p mit_normal -N 1 -n 8 --mem=20GB --pty bash  
module load gcc/12.2.0  
module load openmpi/4.1.4
```

Optimize programs

- Before parallelization, serial programs can be optimized and accelerated substantially!
- Compiler optimizations

```
gcc -O3 my_code.c -o my_program  
gfortran -O3 my_code.c -o my_program
```

```
icc -fast my_code.c -o my_program  
ifort -fast my_code.c -o my_program
```

- Compiling source code and GNU Make: <https://orcd-docs.mit.edu/software/compile/>
- Optimizing codes to speed up

Unnecessary work (1): redundant operations

- Avoid redundant operations in loops

bad

```
for i=1:N
    x = 10;
    .
    .
end
```

good

```
x = 10;
for i=1:N
    .
    .
end
```

Unnecessary work (2): reduce overhead

..from function calls

bad

```
function myfunc(i)
    % do stuff
end

for i=1:N
    myfunc(i);
end
```

good

```
function myfunc2(N)
    for i=1:N
        % do stuff
    end
end

myfunc2(N);
```

..from loops

bad

```
for i=1:N
    x(i) = i;
end
for i=1:N
    y(i) = rand();
end
```

good

```
for i=1:N
    x(i) = i;
    y(i) = rand();
end
```

Unnecessary work (3): logical tests

Avoid unnecessary logical tests...

...by using short-circuit
logical operators

```
if (i == 1 | j == 2) & k == 5
    % do something
end
```

bad

```
if (i == 1 || j == 2) && k == 5
    % do something
end
```

good

...by moving known cases
out of loops

bad

```
for i=1:N
    if i == 1
        % i=1 case
    else
        % i>1 case
    end
end
```

good

```
% i=1 case
for i=2:N
    % i>1 case
end
```

Unnecessary work (4): reorganize equations

Reorganize equations to use fewer or more efficient operators

Basic operators have different speeds:

Add	3- 6 cycles
Multiply	4- 8 cycles
Divide	32-45 cycles
Power, etc	(worse)

bad

```
c = 4;
for i=1:N
    x(i)=y(i)/c;
    v(i) = x(i) + x(i)^2 + x(i)^3;
    z(i) = log(x(i)) * log(y(i));
end
```

good

```
s = 1/4;
for i=1:N
    x(i) = y(i)*s;
    v(i) = x(i)*(1+x(i)*(1+x(i)));
    z(i) = log(x(i) + y(i));
end
```


Memory efficiency (1): preallocate arrays

- Arrays are always allocated in contiguous address space.
- If an array changes size, and runs out of contiguous space, it must be moved. For example,

```
x = 1;  
for i = 2:4  
    x(i) = i;  
end
```

- This can be very very bad for performance when variables become large.

Memory Address	Array Element
1	x(1)
...	...
2000	x(1)
2001	x(2)
2002	x(1)
2003	x(2)
2004	x(3)
...	...
10004	x(1)
10005	x(2)
10006	x(3)
10007	x(4)



Memory efficiency (1): preallocate arrays

- Preallocating array to its maximum size prevents intermediate array movement and copying.

```
A = zeros(n,m); % initialize A to 0
A(n,m) = 0;      % or touch largest element
```

- If maximum size is unknown, estimate with upper bound. Remove unused memory after.

```
A=rand(100,100);
% . . .
% if final size is 60x40, remove unused portion
A(61:end,:)=[]; A(:,41:end)=[]; % delete
```

Memory efficiency (2): loop order

- It is faster to access continuous memory addresses than separated ones.
- Column-major (Fortran, MATLAB) : multidimensional arrays are stored in memory along columns.

bad

```
n=5000; x = zeros(n);  
for i = 1:n          % rows  
    for j = 1:n      % columns  
        x(i,j) = i+(j-1)*n;  
    end  
end
```

good

```
n=5000; x = zeros(n);  
for j = 1:n          % columns  
    for i = 1:n      % rows  
        x(i,j) = i+(j-1)*n;  
    end  
end
```

- Row-major (C, Numpy) : switch the loop order.

Memory efficiency (3): avoid unnecessary variables

- Avoid time needed to allocate and write data to main memory.
- Compute and save array in-place improves performance and reduces memory usage.

bad

```
x = rand(5000);  
y = x.^2;
```

good

```
x = rand(5000);  
x = x.^2;
```

Embarrassingly Parallel

- Embarrassingly parallel, perfectly parallel, delightfully parallel, or pleasingly parallel.
- Run the same program with different input parameters independently
- No communications.
- Slurm job array.

Parallel Computing

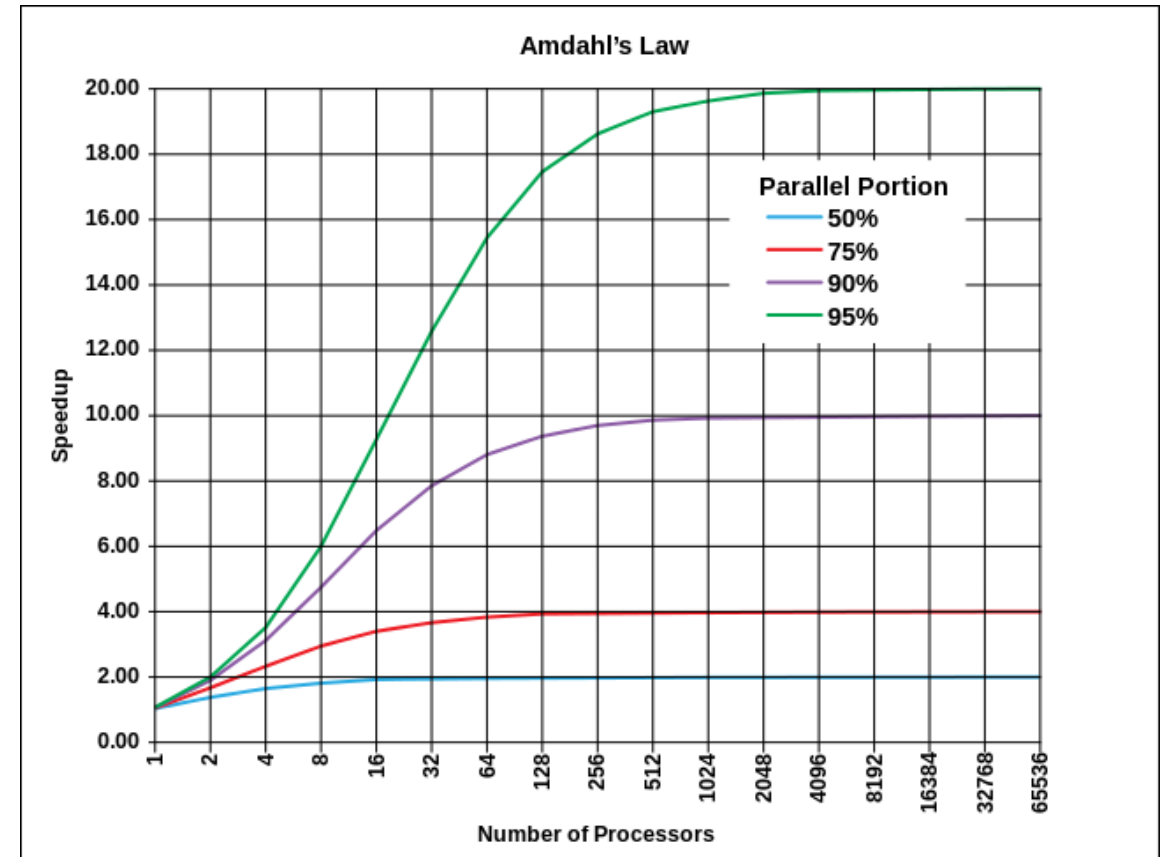
❑ Parallel computing is a type of computation in which many calculations are carried out **simultaneously**, based on the principle that large problems can often be divided into smaller ones, which are then solved at the same time.

❑ **Speedup** of a parallel program,

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

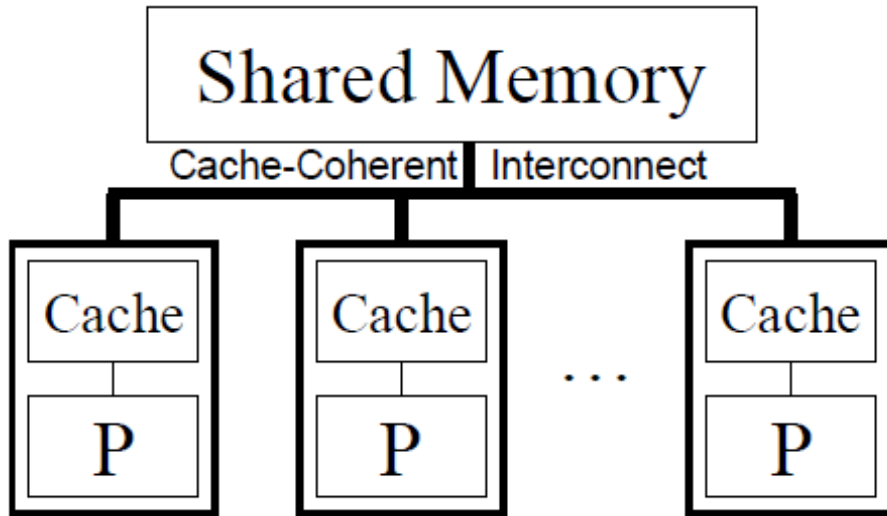
p : number of processors/cores,

α : fraction of the program that is serial.

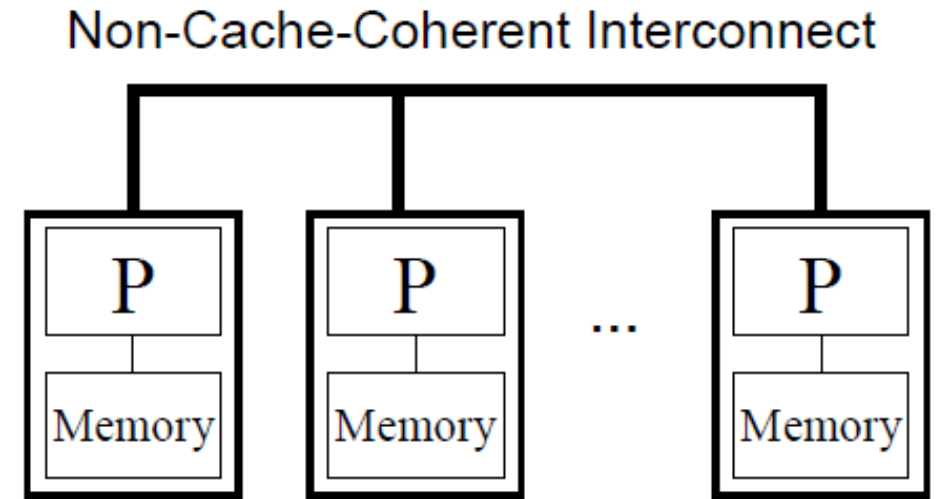


Ref: https://en.wikipedia.org/wiki/Parallel_computing

Distributed or shared memory systems



- Shared memory system
- Multiple cores on a single node
- Multi-processing (OpenMP, Numpy)



- Distributed memory system
- Multiple nodes on a cluster
- Message Passing Interface (MPI, MPI4Py)

MPI works on multiple cores on a node or multiple nodes.

Parallel programming languages

- C, Fortran

Compiling languages for performance, widely used in scientific computing for decades

Parallel library/protocol/platform: OpenMP, MPI, CUDA

- C++

Object-oriented design is not suitable for parallel programming.

- Python

High-level scripting languages for easy use. Call precompiled C libraries for performance.

Parallel packages: Numpy, Multiprocessing, MPI4py, CuPy

- Julia

Compiled for performance. Used as a scripting language. Multi-threading and distributed computing.

- MATLAB

Convenient to deal with matrices. Parallel toolbox. Parallel server.