5.4. Threads and Processes

Parallel programs execute multiple units of execution simultaneously. These execution units are called Processes and Threads.

- A **Process** is an independent program in execution with its own memory space, program counter, registers, and system resources. Each process is isolated from other processes.
- A **Thread** is the smallest unit of execution within a process. Multiple threads share the same memory and resources of the parent process. Threads communicate easily because they operate in the same address space.

5.2 Processes and Threads in HPC

In HPC, the choice between processes and threads directly determines how a parallel program communicates and scales. MPI (Message Passing Interface) uses processes — each MPI rank runs as a fully independent program with its own private memory space, communicating with other ranks by explicitly sending and receiving messages over the network. Because processes do not share memory, MPI works perfectly across multiple nodes in a distributed cluster where shared memory is physically impossible. It is the backbone of virtually every large-scale HPC application running across hundreds or thousands of nodes.

OpenMP,

Open Multi-Processing (OpenMP) on the other hand, uses threads — lightweight execution units that all share the same memory space within a single process. A programmer simply inserts OpenMP directives into existing code, and the compiler automatically spawns threads that divide the work across all available cores on a node. Since threads communicate through shared memory rather than message passing, OpenMP is fast and easy to use within a single multi-core processor or node, but it cannot cross node boundaries.

Hybrid MPI + OpenMP

Modern supercomputers combine both into a Hybrid MPI + OpenMP model — one MPI process is launched per node, and within each node, multiple OpenMP threads spread the work across all available cores. This two-level approach captures the best of both worlds: MPI handles the distributed communication between nodes across the network, while OpenMP exploits all the cores within each node through shared

memory. The result is better scalability, lower communication overhead, and more efficient use of the hardware than either model alone could achieve.

5.5 Difference between Processes and threads

Process	Thread
Own separate memory space	Shares memory with other threads
Heavy to create	Lightweight to create
Communication via IPC (slow)	Communication via shared memory (fast)
Crash is isolated	One crash kills entire process
Expensive context switch	Cheap context switch
High OS resource usage	Low OS resource usage
Used across multiple machines	Used within a single machine
Example — MPI	Example — OpenMP

5.3. Load Balancing

Load Balancing is the process of distributing work evenly among all processors so that every processor remains busy during program execution. Good load balancing improves processor utilization and reduces execution time.

Types of Load Balancing

- Static Load Balancing (Work is divided before execution begins.)
- Dynamic (Task assigned during program execution as per processor availability)

Need for Load Balancing

Suppose four processors Core 1, 2, 3 and 4 execute a program of each 100 tasks. Core 4 finishes quickly and waits while the others continue working. Core 4's computing power is wasted. A balanced workload ensures maximum utilization.

Factors that causes Load imbalance

- Irregular Data Structures
- Adaptive Algorithms
- Heterogeneous Hardware
- Data-Dependent Computation

5.5. Synchronization and Overhead

Parallel programs require synchronization points barriers, locks, or reductions where threads/processes must wait for each other.

Barrier: All threads must reach this point before any can proceed. Introduces idle time if threads arrive at different times (load imbalance exposed here).

Race Condition: Two threads read/modify shared data simultaneously without synchronization → unpredictable, incorrect results.

Overhead sources in parallel programs:

- Thread/process creation and destruction
- Communication (sending/receiving messages in MPI)
- Synchronization (barriers, locks, atomic operations)
- Load imbalance (idle time)
- Extra computation introduced by parallelization

Rule of thumb: The parallel overhead must be much smaller than the computational speedup gained. If parallelizing a 10ms task takes 8ms of overhead, parallelization was not worthwhile.

5.4. Scheduling Strategies

Scheduling determines how tasks are assigned to processors. Proper scheduling minimizes idle time and improves overall performance.

- Round Robin Scheduling
- Block Scheduling
- Cyclic Scheduling

- Self-Scheduling
- Guided Self Scheduling
- Work Stealing

A. Round Robin Scheduling - Tasks are assigned to processors one after another in a circular (rotational) order. It ensures that each processor receives tasks fairly.

Example**:** **Core1:** Task1, Task5

Core2: Task2, Task6

Core3: Task3, Task7

Core4: Task4, Task8

B. Block Scheduling

Each processor is assigned one large, contiguous block of tasks or data before execution begins. It is efficient when all tasks require approximately the same execution time.

Example: **Processor1:** Rows 1–2500

Processor2: Rows 2501–5000

Processor3: Rows 5001–7500

Processor4: Rows 7501–10000

C. Cyclic Scheduling

Tasks are assigned to processors in an interleaved pattern to improve load balance when task execution times vary.

Example**:** **Core1:** Task1, Task5, Task 9

Core2: Task2, Task6, Task10

Core3: Task3, Task7, Task11

Core4: Task4, Task8, Task12

D. Self-Scheduling

Tasks are assigned to processors in an interleaved pattern to improve load balance when task execution times vary.

Example: **Shared Queue:** Task1, Task2, Task3,.,.,so on When **Core2** becomes idle, it takes the next available task from the queue.

E. Guided Self Scheduling

A dynamic scheduling strategy where an idle processor retrieves the next available task from a shared task queue at runtime, this approach is effective for unpredictable workloads.

Example: Initially: **Core1 → 100 tasks**, **Core2 → 100 tasks**

Later: **Core1 → 20 tasks**, **Core2 → 10 tasks**

Finally smaller chunks until all tasks are completed.

F. Work Stealing

Each processor maintains its own task queue. If a processor finishes early and becomes idle, it steals tasks from the queue of a busy processor.

Example: **Core1** completes its tasks early and steals unfinished tasks from **Core3**'s queue, allowing both cores to remain busy until execution finishes.

5.6 Scheduling in High Performance Computing

A scheduler is software that implements a *batch system* on a HPC (cluster). Users do not run their calculations directly and interactively (as they do on their personal workstations or laptops), instead they submit non-interactive *batch jobs* to the scheduler.

The scheduler stores the batch jobs, evaluates their resource requirements and priorities, and distributes the jobs to suitable *compute nodes*. These work horses make up the majority of HPC clusters (about 98%), being their most powerful, but also the most power consuming parts.

Strategies

There are two very basic strategies that schedulers can use to determine which job to run next. Note that modern schedulers do not stick strictly to just one of these algorithms, but rather employ a combination of the two. Besides, there are many more aspects a scheduler has to take into consideration, e. g. the current system load.

1. First Come, First Serve

Jobs are run in the exact same order in which they first enter the queue. The advantage is that every job will definitely be run, however, very tiny jobs might wait

for an inadequately long time compared to their actual execution time or all jobs would need to wait at some point if one user submits several thousand jobs at the same time.

2. Shortest Job First

Based on the execution time declared in the jobscript, the scheduler can estimate how long it will take to execute the job. Then, the jobs are ranked by that time from shortest to longest. While short jobs will start after a short waiting time, long running jobs (or at least jobs declared as such) might never actually start.

3. Backfilling

When backfilling the scheduler maintains the concept of First Come, First Serve, without preventing long running jobs to execute. The scheduler checks whether the first job in the queue can be executed. If that is true, the job is executed without further delay. But if not, the scheduler goes through the rest of the queue to check whether another job can be executed without extending the waiting time of the first job in queue. If it finds such a job, the scheduler simply runs the job. Since jobs, which only need a few compute resources, are easily "backfillable", small jobs will usually encounter short queue times.

FOR MORE INFORMATION REFER THIS BELOW LINK:

https://hpc-wiki.info/hpc/HPC_Wiki