High Performance Computing

(HPC)

Introduction to HPC

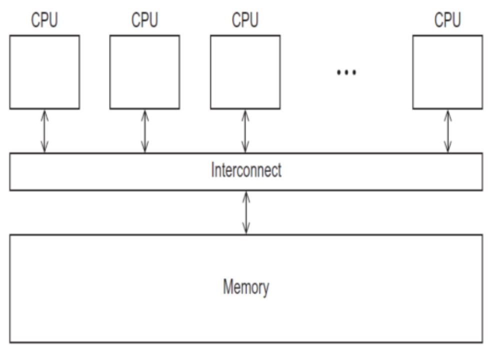
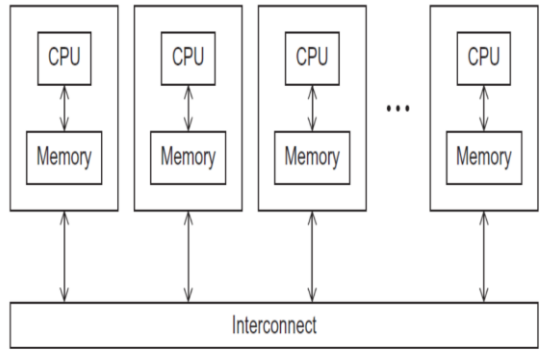
1) What is HPC
   - A collection of powerful
     - Hardware systems & architecture
     - Software tools
     - Programming languages **(Fortran, c++, c, ADA)**
     - Parallel programming paradigms

     And Computational Intelligence

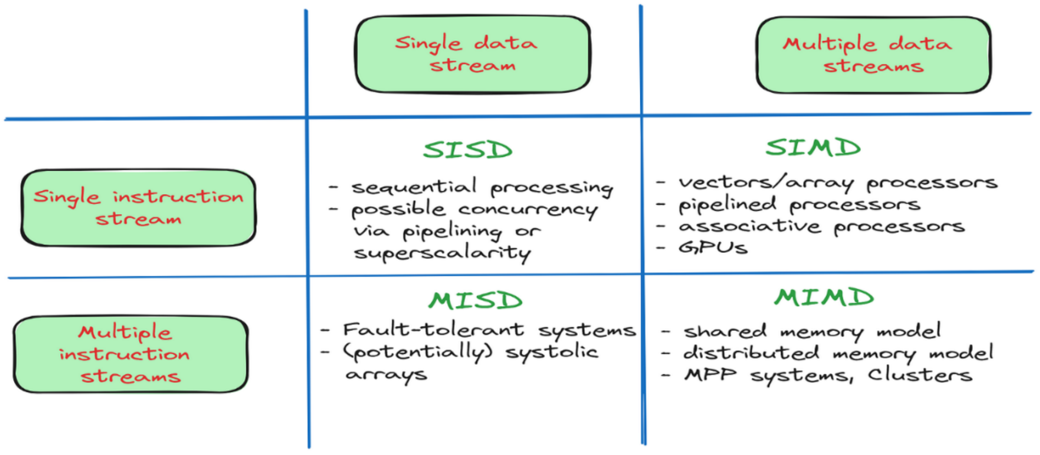
   - We use hpc to make unfeasible computation possible that means the problem itself must be computable
2) Types of science
   - Theoretical **(Abstract, mathematical)**
   - Experimental **(Hands-on, empirical)**
   - Computational science engineering **(Simulation-driven, data-intensive.)**
3) Applications of HPC
   - Applied Fluid Dynamics
   - Ecosystem simulation
   - Molecular Biology
   - Nuclear power & weapons

4) HPC Architectures

- Distributed Memory Machines vs Shared Memory Machines

	Shared Memory Machine	Distributed Memory Machine
Memory Arch.	All processors shares a global memory	Each processor has its own private memory.
Comm.	Implicit (shared variables)	Explicit (Message Passing)
Latency	Lower	Higher
Diagram		 CPU Memory CPU Memory CPU Memory ... CPU Memory Interconnect

- Most systems uses Hybrid Approach, Hybrid architecture combines shared-memory (e.g., multi-core CPUs) and distributed-memory (e.g., networked nodes) systems to optimize performance, scalability, and resource utilization. It leverages the best of both worlds for high-performance computing (HPC).
- CPUS, GPUS(SIMD) ML, AI, FPGAS
- Flynn's Taxonomy

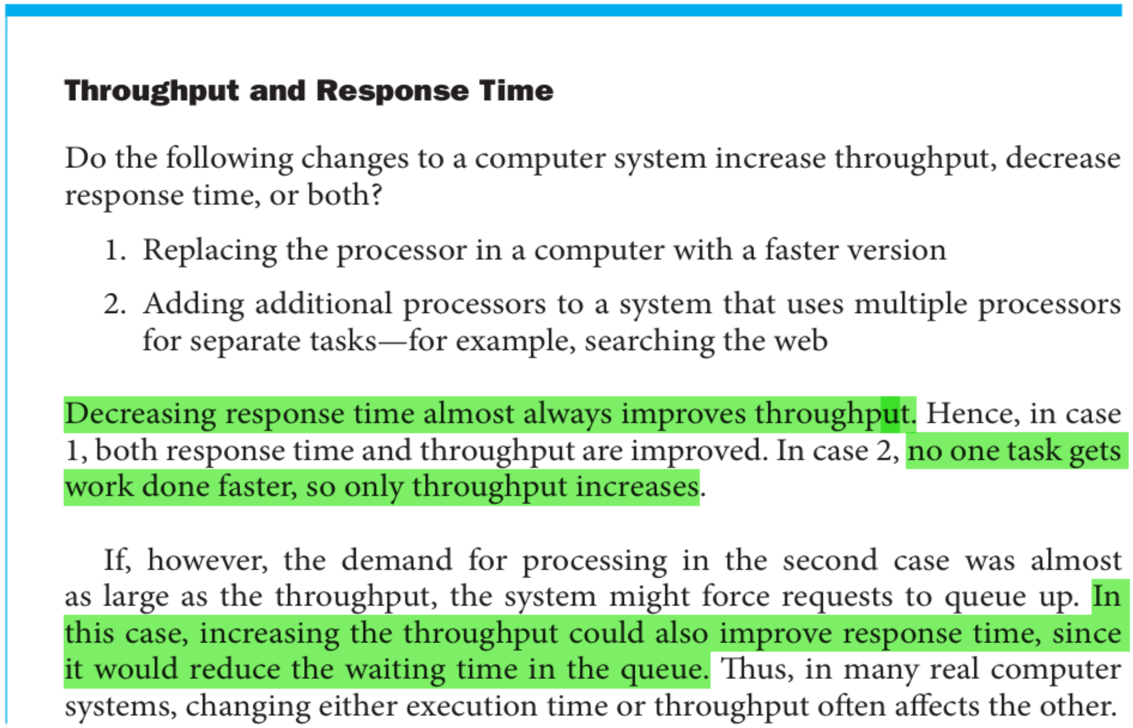


	Single data stream	Multiple data streams
Single instruction stream	SISD - sequential processing - possible concurrency via pipelining or superscalarity	SIMD - vectors/array processors - pipelined processors - associative processors - GPUs
Multiple instruction streams	MISD - Fault-tolerant systems - (potentially) systolic arrays	MIMD - shared memory model - distributed memory model - MPP systems, Clusters

5) Levels of parallelism
   - Data level parallelism (16-bit data is processed using 8-bit operations**)**
   - Instruction Level Parallelism (Pipelining)
   - OS level parallelism
     - Task-level parallelism
     - Process-level parallelism
     - Thread-level parallelism

Performance in HPC

1) Response time vs Throughput
   - **Response time:** Time between the start & completion of an event
   - **Throughput:** Total amount of work done in a given time
   - Decreasing response time almost always improves throughput but not vice versa



Throughput and Response Time

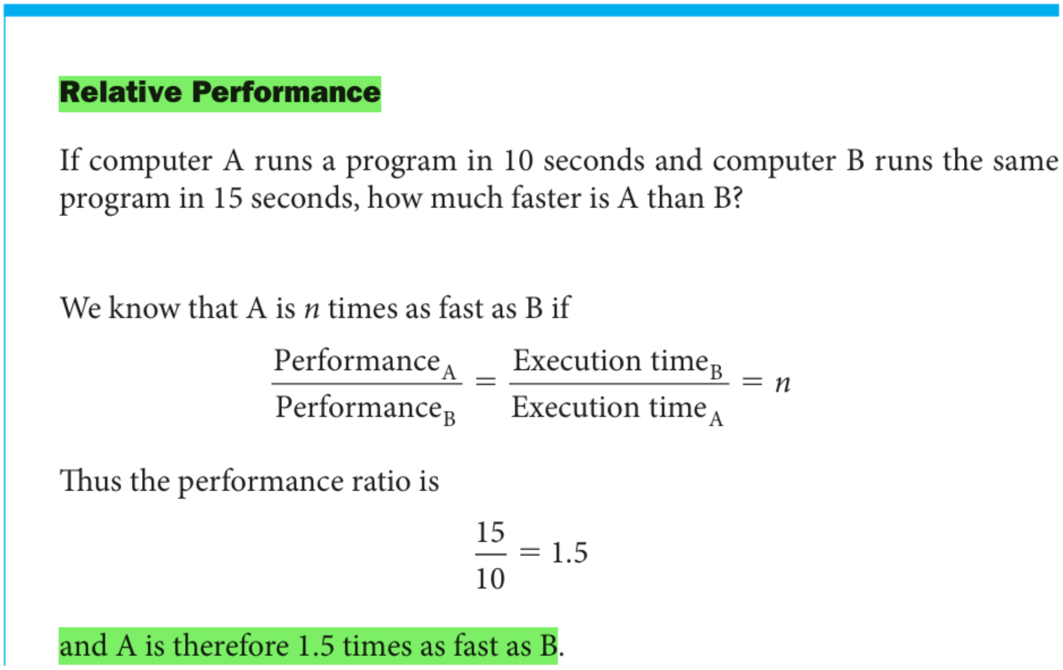
Do the following changes to a computer system increase throughput, decrease response time, or both?

1. Replacing the processor in a computer with a faster version
2. Adding additional processors to a system that uses multiple processors for separate tasks—for example, searching the web

Decreasing response time almost always improves throughput. Hence, in case 1, both response time and throughput are improved. In case 2, no one task gets work done faster, so only throughput increases.

If, however, the demand for processing in the second case was almost as large as the throughput, the system might force requests to queue up. In this case, increasing the throughput could also improve response time, since it would reduce the waiting time in the queue. Thus, in many real computer systems, changing either execution time or throughput often affects the other.

2) To maximize performance we need to minimize response time so they have an inverse relationship



Relative Performance

If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B?

We know that A is *n* times as fast as B if

$$\frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Thus the performance ratio is

$$\frac{15}{10} = 1.5$$

and A is therefore 1.5 times as fast as B.

3) Measuring Performance

- **Elapsed Time:** Total time to complete a task, including disk accesses, memory accesses, input/output (I/O) activities, operating system overhead—everything. We are not interested in the elapsed time we are interested in the CPU execution time
- **CPU Time** is the time the CPU spends computing for this task and does not include time spent waiting for I/O or running other programs. (Remember, though, that the response time experienced by the user will be the elapsed time of the program, not the CPU time.) CPU time can be further divided into the CPU time spent in the program, called **user CPU time**, and the CPU time spent in the operating system performing tasks on behalf of the program, called **system CPU time**.

$$\begin{matrix}\text{CPU execution time}\\ \text{for a program}\end{matrix} = \begin{matrix}\text{CPU clock cycles}\\ \text{for a program}\end{matrix} \times \text{Clock cycle time}$$

Alternatively, because clock rate and clock cycle time are inverses,

$$\begin{matrix}\text{CPU execution time}\\ \text{for a program}\end{matrix} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

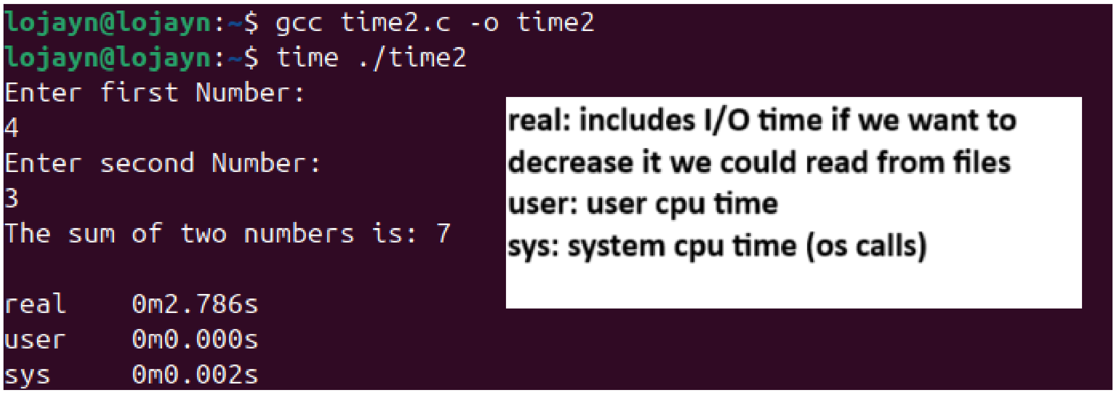
$$\text{CPU clock cycles} = \text{Instructions for a program} \times \begin{matrix}\text{Average clock cycles}\\ \text{per instruction}\end{matrix}$$

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{C}_i)$$

4) Measuring time

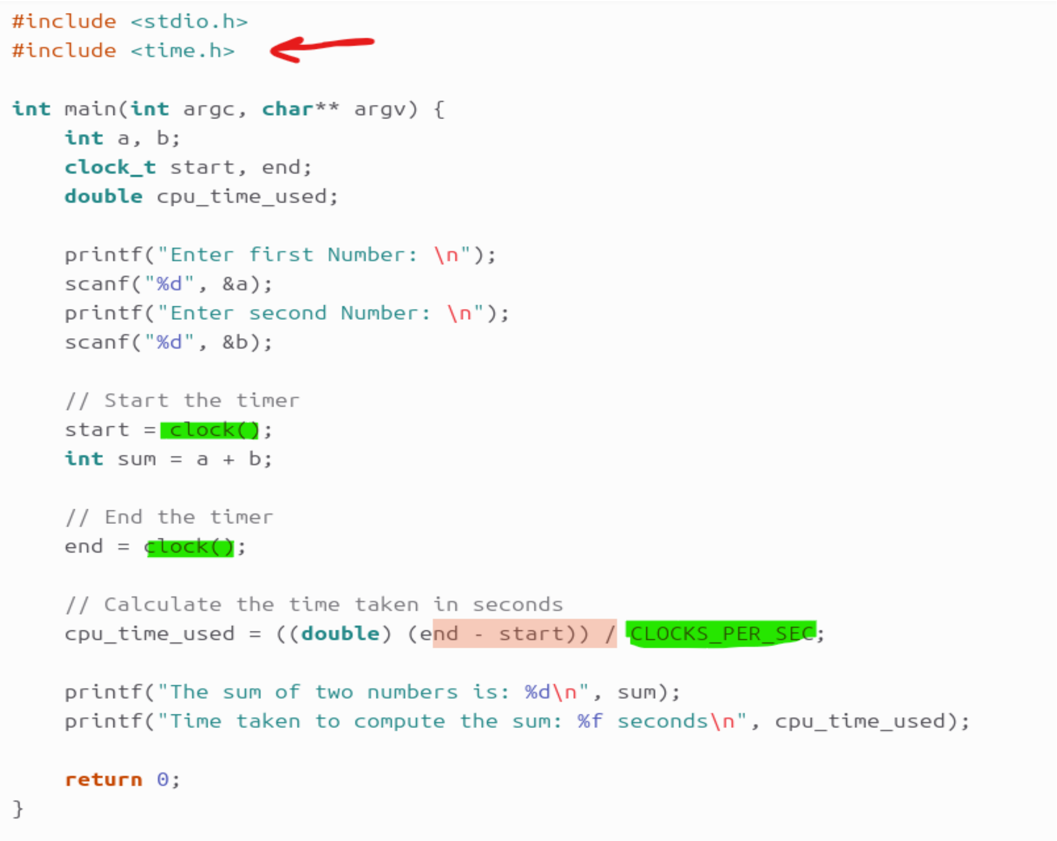
- Watches & stopwatches
- Time utility (Unix systems) **(Doesn't measure time in parts)**



```
lojayn@lojayn:~$ gcc time2.c -o time2
lojayn@lojayn:~$ time ./time2
Enter first Number:
4
Enter second Number:
3
The sum of two numbers is: 7

real	0m2.786s
user	0m0.000s
sys	0m0.002s
```

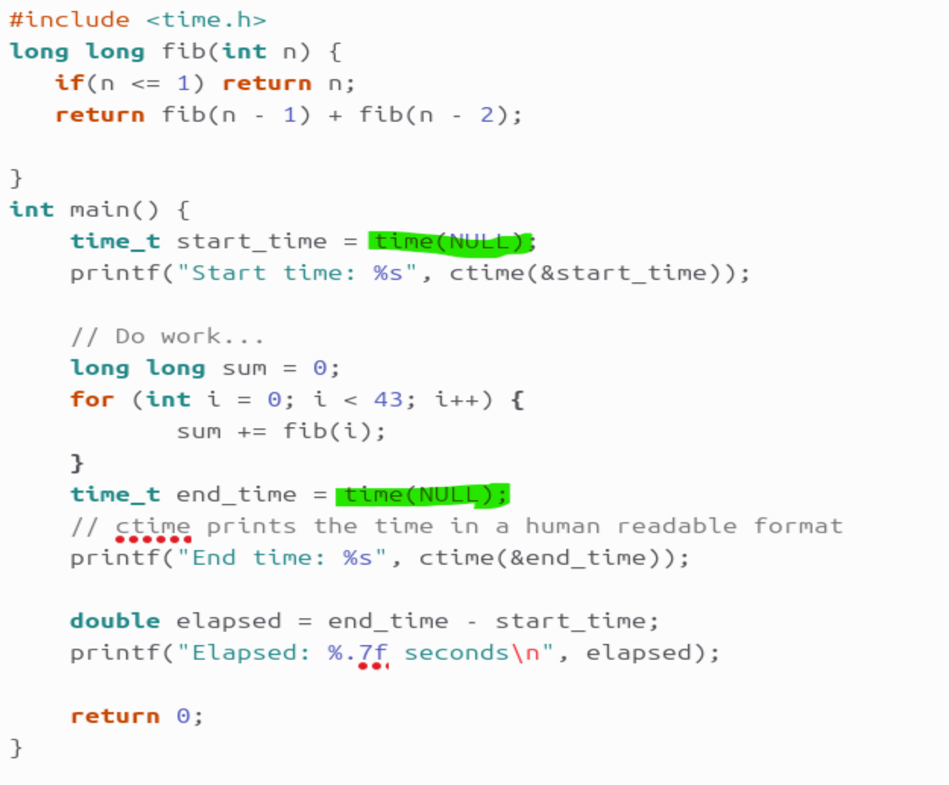
real: includes I/O time if we want to decrease it we could read from files
user: user cpu time
sys: system cpu time (os calls)

- Clock()



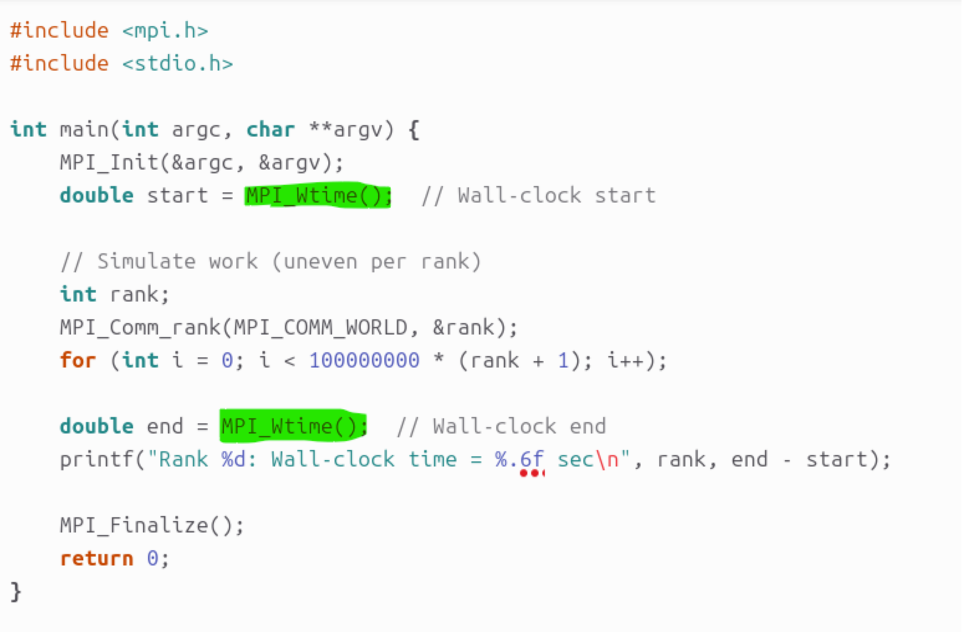
```c
#include <stdio.h>
#include <time.h>

int main(int argc, char** argv) {
    int a, b;
    clock_t start, end;
    double cpu_time_used;

    printf("Enter first Number: \n");
    scanf("%d", &a);
    printf("Enter second Number: \n");
    scanf("%d", &b);

    // Start the timer
    start = clock();
    int sum = a + b;

    // End the timer
    end = clock();

    // Calculate the time taken in seconds
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    printf("The sum of two numbers is: %d\n", sum);
    printf("Time taken to compute the sum: %f seconds\n", cpu_time_used);

    return 0;
}
```

- Time()



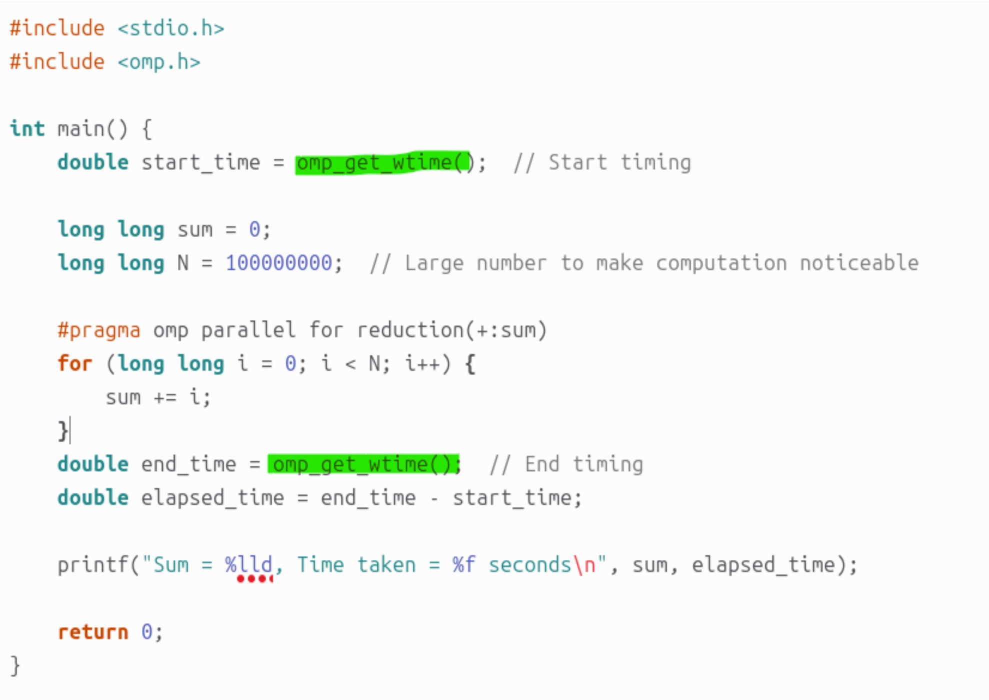
```c
#include <time.h>
long long fib(int n) {
   if(n <= 1) return n;
   return fib(n - 1) + fib(n - 2);

}
int main() {
    time_t start_time = time(NULL);
    printf("Start time: %s", ctime(&start_time));

    // Do work...
    long long sum = 0;
    for (int i = 0; i < 43; i++) {
           sum += fib(i);
    }
    time_t end_time = time(NULL);
    // ctime prints the time in a human readable format
    printf("End time: %s", ctime(&end_time));

    double elapsed = end_time - start_time;
    printf("Elapsed: %.7f seconds\n", elapsed);

    return 0;
}
```

- MPI_WTime()



```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    double start = MPI_Wtime();  // Wall-clock start

    // Simulate work (uneven per rank)
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    for (int i = 0; i < 100000000 * (rank + 1); i++);

    double end = MPI_Wtime();  // Wall-clock end
    printf("Rank %d: Wall-clock time = %.6f sec\n", rank, end - start);

    MPI_Finalize();
    return 0;
}
```

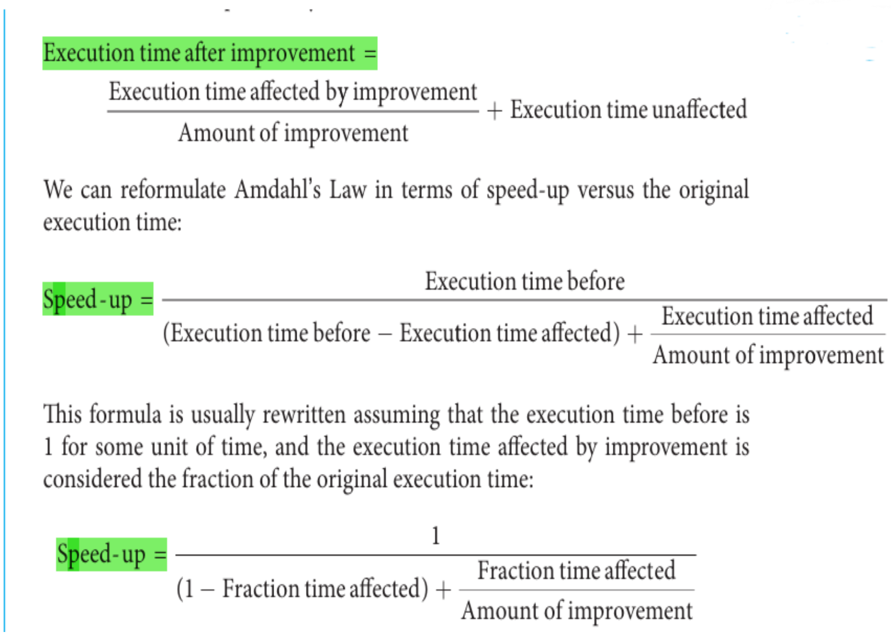
- Omp_get_wtime()



```c
#include <stdio.h>
#include <omp.h>

int main() {
    double start_time = omp_get_wtime();  // Start timing

    long long sum = 0;
    long long N = 100000000;  // Large number to make computation noticeable

    #pragma omp parallel for reduction(+:sum)
    for (long long i = 0; i < N; i++) {
        sum += i;
    }
    double end_time = omp_get_wtime();  // End timing
    double elapsed_time = end_time - start_time;

    printf("Sum = %lld, Time taken = %f seconds\n", sum, elapsed_time);

    return 0;
}
```

5) Benchmarks
    - Programs chosen to measure the performance of a computer system

6) Amdahl's law



$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

We can reformulate Amdahl's Law in terms of speed-up versus the original execution time:

$$\text{Speed-up} = \frac{\text{Execution time before}}{(\text{Execution time before} - \text{Execution time affected}) + \dfrac{\text{Execution time affected}}{\text{Amount of improvement}}}$$

This formula is usually rewritten assuming that the execution time before is 1 for some unit of time, and the execution time affected by improvement is considered the fraction of the original execution time:

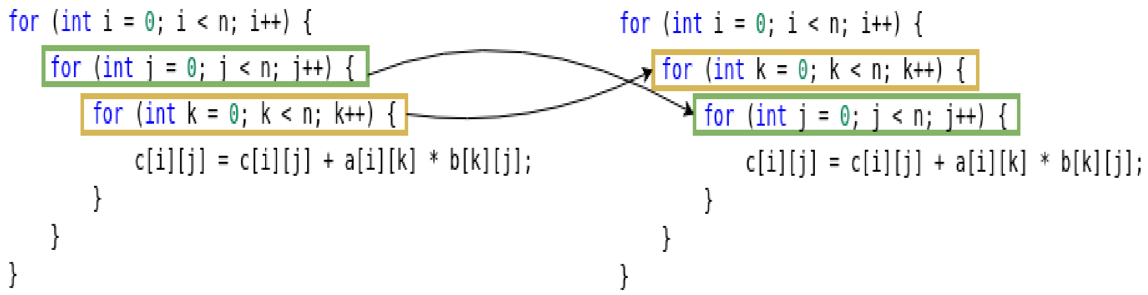
$$\text{Speed-up} = \frac{1}{(1 - \text{Fraction time affected}) + \dfrac{\text{Fraction time affected}}{\text{Amount of improvement}}}$$

Memory & Parallelism In HPC

1) Data Locality
   - Spatial Locality

     If an item is referenced, then items whose addresses are close by will tend to be referenced soon **(hardware)**
   - Temporal Locality

     If an item is referenced then it will tend to be referenced soon again **(software)**
2) Loop interchange



```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}
```

```
for (int i = 0; i < n; i++) {
    for (int k = 0; k < n; k++) {
        for (int j = 0; j < n; j++) {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}
```

3) Loop unrolling
   - **Loop unrolling** (or **loop unwinding**) is a compiler optimization technique that reduces the overhead of loop control by executing multiple iterations of a loop in a single step. Instead of running a loop normally (checking conditions and incrementing counters each time), loop unrolling replicates the loop body to decrease branching and improve performance.

Parallel Programming in HPC (MPI)

1) Structure of parallel programs: the program begins sequential & at a certain point we can start a parallel section
2) Message Passing interface (MPI) used to program **multi-node machines** but also could be used in shared memory machines
3) MPI Functions
   - MPI_Init(int* argc, char*** argv)
     - Initializes the MPI environment
     - Sets up communicators and assigns ranks to processes.
     - No MPI Function could be called before it
   - MPI_Finalize(void)
     - Cleans up the MPI Environment
     - No MPI function could be called after it
   - MPI_Comm_size(MPI_Comm communicator, int * size)
     - Retrieves the total number of processes in the specified communicator (**MPI_COMM_WORLD**), Stores the result in **size**
   - MPI_Comm_rank(MPI_Comm communicator, int * rank)
     - Retrieves the rank (unique ID) of the current process within the communicator (Ranks start from 0 to size – 1) & stores it in rank variable
   - MPI_Get_processor_name(char* name, int* name_length)
     - Gets the name of the host processor (machine) running the process. Stores the name in name variable & the length of that name in name_length
   - MPI_Send(void* data, int count , MPI_Datatype dt, int dest, MPI_Comm Communicator)
     - Sends message from one process to another
   - MPI_Recv(void* data, int count , MPI_Datatype dt, int source, int tag, MPI_Comm Communicator, MPI_Status* status);
     - Receives message sent by other process

- MPI_Bcast(void*data, int count, MPI_Datatype dt, int root, MPI_Comm comm);
    - Sends data from root process to all other processes
    - All processes must allocate memory for the received data before calling bcast
- MPI_Scatter(void* sendBuf, int sendcount, MPI_Datatype dt, void* recvbuf, int recvcount, int root, MPI_Comm comm);
    - Root process splits sendbuf into chunks and sends one chunk to each process (including itself).
- MPI_Gather(void* sendBuf, int sendcount, MPI_Datatype dt, void* recvbuf, int recvcount, int root, MPI_Comm comm);
    - Root collects all chunks into recvbuf in rank order
- MPI_Reduce(void* send_data, void*recv_data, int count, MPI_Datatype dt, MPI_Op op, MPI_Comm comm);
    - Aggregates data from all processes using a specified operation (e.g., sum, max) and stores the result only on the root process
- MPI_AllReduce(void* send_data, void*recv_data, int count, MPI_Datatype dt, MPI_Op op, MPI_Comm comm);
    - Same as MPI_Reduce, but broadcasts the result to all processes (no root needed)
- MPI_Abort(MPI_Comm comm, int errorcode);
    - Forces all processes in an mpi_program to terminate
    - Useful for handling errors

4) Peer to peer vs Collective communication

	P2P	Collective
Basic Concept	Direct communication between two processes (sender and receiver)	Involves all processes in a communicator (or a defined group).
MPI Function	MPI_Send , MPI_Recv	MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Allgather, MPI_Reduce, MPI_Allreduce,

Numerical Computing & OpenMP

1) Numerical Integration
   - Numerical integration is a computational technique used to approximate the value of definite integrals when an analytical (exact) solution is difficult or impossible to obtain. It is widely used in scientific computing, engineering, physics, and finance. Some common methods include **Trapezoidal Rule** (approximates area under a curve using trapezoids)
   - Numerical integration can be computationally intensive because it often requires evaluating the integrand function many times. As the step size decreases (leading to finer discretization), the approximation becomes more accurate—but this also increases the total number of computations needed, resulting in higher computational cost.
   - Applications: **Quantum Mechanics, Fluid Dynamics**
2) Parallel vs Sequential Programming
   - While parallel computing can significantly speed up computations, it is not always the optimal solution why?
     1. Poorly designed parallel programs may perform worse than their sequential counterparts due to communication overhead, synchronization costs, or inefficient workload distribution.
     2. Some algorithms (e.g., matrix multiplication on small matrices) only outperform sequential versions when problem sizes are very large.
     3. Some algorithms (like the Collatz conjecture) are inherently sequential.

     Therefore, before parallelizing, it's crucial to analyze the problem's structure, assess scalability, and apply common sense—parallelism isn't a one-size-fits-all remedy

3) What is OpenMp

- **OpenMP (Open Multi-Processing)** is an **API** (Application Programming Interface) for **shared-memory parallel programming** in C, C++, and Fortran. It enables developers to parallelize code efficiently using **directives**
- **Coarse-grained = Big chunks** – OpenMP divides work into large tasks

4) **Comparison between OpenMP & MPI**

	OpenMP	MPI
Parallelism model	Shared memory	Distributed Memory
Communication	Implicit (shared variables)	Explicit (Message Passing)
Overhead	Low(threads share memory)	Higher
Scalability	Limited to single-node cores	Scales to thousands of nodes
Parallelization unit	Threads	Processes
example	#pragma omp parallel for	MPI_Bcast(), MPI_Send(), MPI_Recv