(V 1.0 09/25)

04. Launching and Structuring Parallel Programs with MPI

Supercomputing for Artificial Intelligence

Foundations, Architectures, and Scaling Deep Learning Workloads

Jordi TORRES.AI

Content

4.1 Foundations of Parallel Execution on Supercomputers

Foundations of Parallel Execution

- **Running parallel programs requires:**
  - A mechanism to **launch jobs**
  - A model to **express parallelism**

- **This section:**
  - Introduces **srun** (used throughout this book)
  - Explains the motivation behind **traditional parallel programming models** (MPI, OpenMP)

- **Goal: understand how workloads execute efficiently at scale**

Launching Parallel Jobs

- **Once SLURM allocates resources, it does not control execution**

- **Users must decide how to launch parallel work**

- **Two main approaches:**
  - mpirun (traditional MPI launcher)
    - Still widely used in HPC to launch MPI applications
    - Works without workload managers like SLURM
    - Starts multiple program instances across nodes
    - mpiexec is usually synonymous with mpirun
  - **srun** (SLURM-integrated launcher)

srun

- **Native SLURM launcher, tightly integrated with resource management**

- **Syntax: `srun [options] <executable>`**

- **Options:**
  - `-n, --ntasks` → number of tasks
  - `-c, --cpus-per-task` → CPU cores per task
  - `-N, --nodes` → number of nodes

- **Example:**

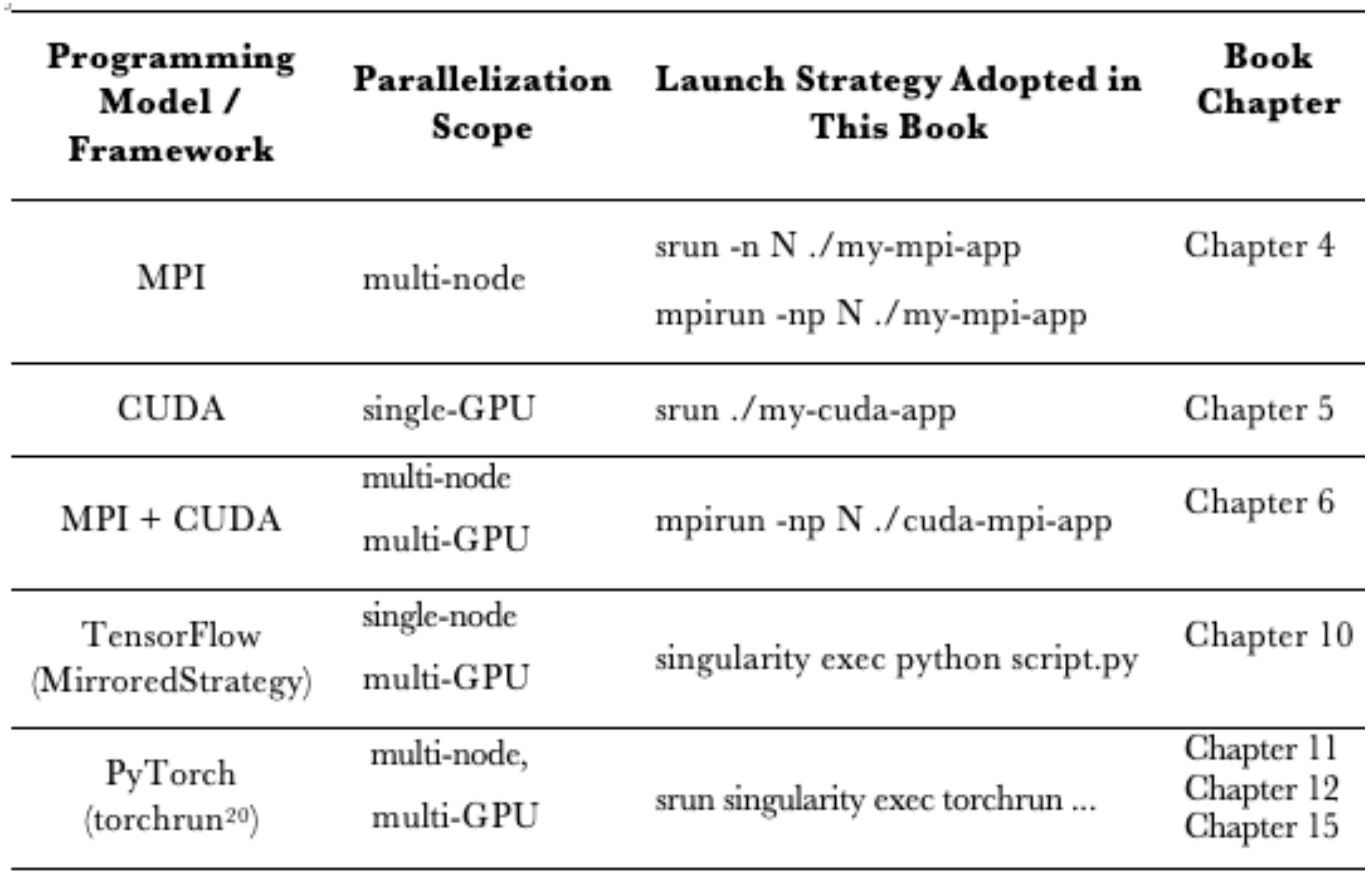
```
srun -n4 -c8 ./my_app
```

- **Inherits parameters from `#SBATCH` directives**

Parallel Job Strategies in this Book

- **Default: srun for most examples on MareNostrum 5**
- **Exceptions:**
  - **mpirun** examples → for learning
  - **TensorFlow (MirroredStrategy)** → no srun, handled internally
- **All frameworks benefit from Singularity containers**
- **Next Table summarizes launch strategies by framework & chapter**

Parallel Job Strategies in this Book



Programming Model / Framework	Parallelization Scope	Launch Strategy Adopted in This Book	Book Chapter
MPI	multi-node	srun -n N ./my-mpi-app mpirun -np N ./my-mpi-app	Chapter 4
CUDA	single-GPU	srun ./my-cuda-app	Chapter 5
MPI + CUDA	multi-node multi-GPU	mpirun -np N ./cuda-mpi-app	Chapter 6
TensorFlow (MirroredStrategy)	single-node multi-GPU	singularity exec python script.py	Chapter 10
PyTorch (torchrun[20])	multi-node, multi-GPU	srun singularity exec torchrun ...	Chapter 11 Chapter 12 Chapter 15

Why Parallel Programming Models Matter

- **Launching jobs is not enough**
  - **→ need to define *how work is divided***
- **Key historical idea: divide to conquer**
- **Early HPC systems → required abstractions for parallelism**
  - **MPI** for distributed-memory systems
  - **OpenMP** for shared-memory systems
- **These models hide low-level hardware details while enabling scalability**

MPI, NCCL, and OpenMP

- **MPI (1990s)**
  - 1990s
  - Portable, scalable, de facto standard in scientific HPC
  - Still vital in some scientific domains
- **Modern systems use CPUs + GPUs**
  - New models required
- **NCCL**
  - Optimized for GPU collectives, now default in PyTorch
- **OpenMP**
  - Still important for CPU-based shared-memory workloads, but not used in large-scale AI training
- **In this book:**
  - Focus on MPI + CUDA + NCCL hybrid approaches

4.2 Getting Started with MPI

What is MPI?

- **Message Passing Interface (MPI)**
  - Standardized, portable library
  - Not a language → callable from C, C++, Fortran

  - Programmer must manage explicit communication
    - Enables processes to cooperate via explicit send/receive

  - De facto standard for distributed-memory HPC programming
    - Each process has its own address space
    - Enables growth by adding more nodes

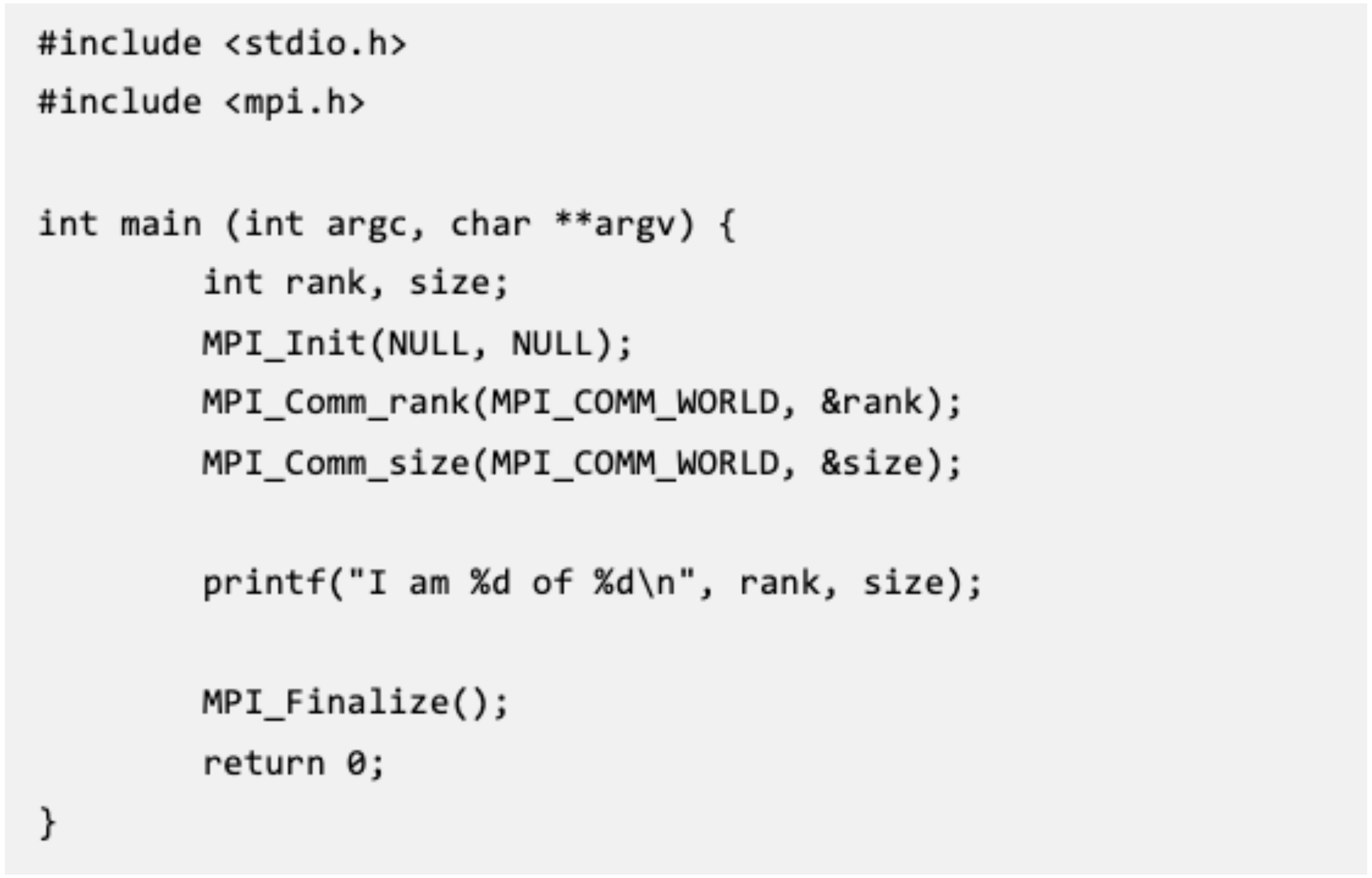
  - Portable across supercomputers worldwide

Distributed Memory in MPI

- **Each process has its own address space**
- **Excellent scalability: each node brings CPU, memory, storage, bandwidth**
- **Enables growth by adding more nodes**
- **Cost: programmer must manage explicit communication**
- **Shared data structures are harder to map**
  - → use message passing is required

MPI Hello World

- Each process prints:
  - Its rank (unique ID)
  - The total number of processes



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

int main (int argc, char **argv) {
        int rank, size;
        MPI_Init(NULL, NULL);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        printf("I am %d of %d\n", rank, size);

        MPI_Finalize();
        return 0;
}
```

Compiling with Intel MPI

- **Intel wrapper compiler: mpiicx**
  - Combines Intel icx with MPI flags & libraries

- **Steps:**
  - Load Intel oneAPI MPI module

```
module load oneapi/2023.2.0
```

  - Compile program

```
mpiicx mpi_helloworld.c -o mpi_helloworld
```

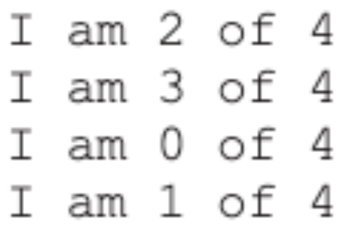
Launching with <u>mpirun</u>

- **mpirun is the “classic” MPI launcher**
- **Starts multiple program instances, one per process**
- **Handles:**
  - Process spawning
  - Rank assignment
  - Communication setup

- **Example:**

```
mpirun -np 4 ./mpi_helloworld
```

→ Output (unordered):



```
I am 2 of 4
I am 3 of 4
I am 0 of 4
I am 1 of 4
```

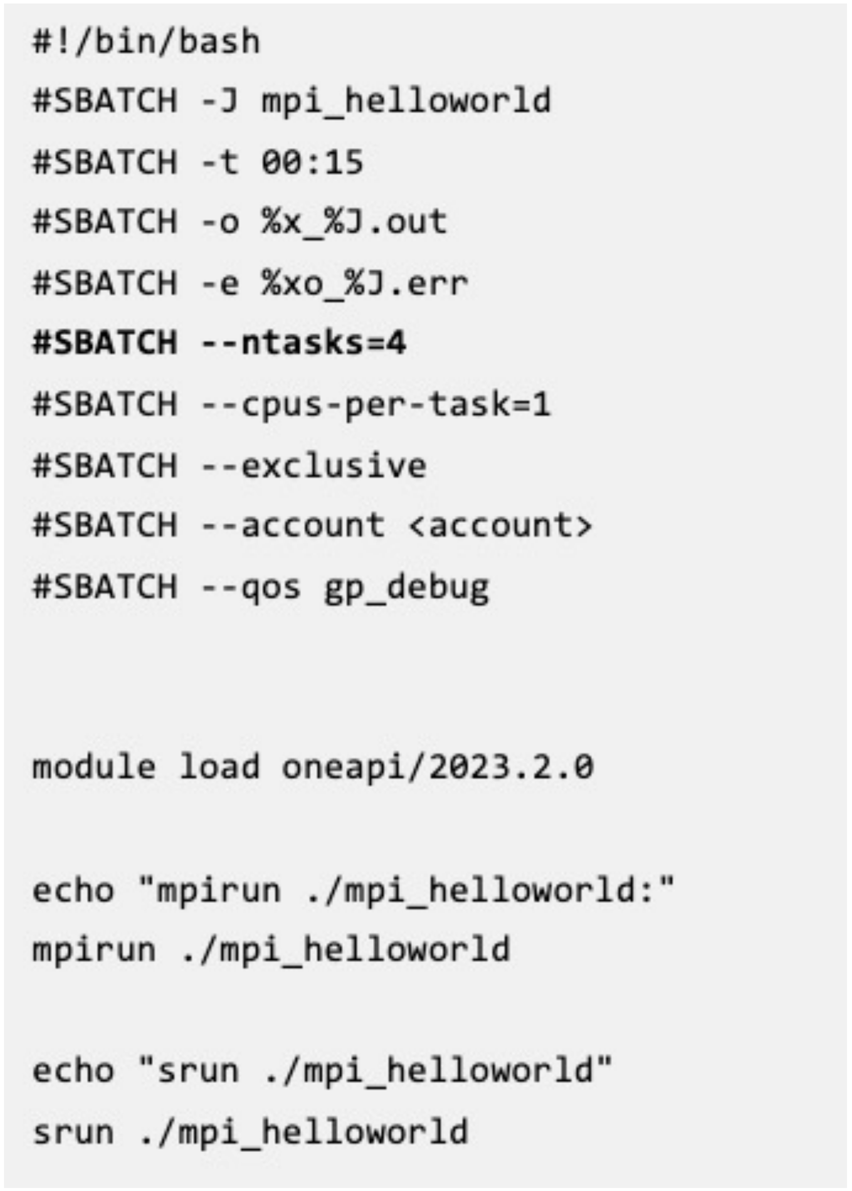
Launching with srun

- **Native SLURM launcher: srun**
- **Integrated with resource allocation & accounting**
- **mpirun and srun launchers work, but srun is preferred on MN5**
- **Example batch script runs program twice:**
  - First with **mpirun**
  - Then with **srun**

```
echo "mpirun ./mpi_helloworld:"
mpirun ./mpi_helloworld

echo "srun ./mpi_helloworld"
srun ./mpi_helloworld
```

Launching MPI in SLURM



```
#!/bin/bash
#SBATCH -J mpi_helloworld
#SBATCH -t 00:15
#SBATCH -o %x_%J.out
#SBATCH -e %xo_%J.err
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=1
#SBATCH --exclusive
#SBATCH --account <account>
#SBATCH --qos gp_debug


module load oneapi/2023.2.0

echo "mpirun ./mpi_helloworld:"
mpirun ./mpi_helloworld

echo "srun ./mpi_helloworld"
srun ./mpi_helloworld
```

– Example:
  - SLURM directive --ntasks=4 ensures 4 processes are launched
  - Both mpirun and srun launch 4 processes
  - Each prints its rank (0...3)
  - Order is not guaranteed → asynchronous output
  - **From now on:**
    use srun in SLURM-managed environments

Launching MPI in SLURM

```
mpirun ./mpi_helloworld:
I am 2 of 4
I am 1 of 4
I am 3 of 4
I am 0 of 4

srun ./mpi_helloworld
I am 3 of 4
I am 0 of 4
I am 2 of 4
I am 1 of 4
```

Task 4.1

- **Task 4.1 – Compile and run your first MPI program**
  a) Compile and execute the basic MPI Hello World example using the `mpiicx` compiler.
  b) Use the SLURM script provided to launch 4 MPI tasks.
  c) Examine the output and confirm that each process prints its rank and the total number of processes.

Multi-node Distribution Example

- Goal: run 16 tasks across 4 nodes (4 per node)

- SLURM directives:

```
#SBATCH --ntasks=16              # Total number of MPI tasks
#SBATCH --nodes=4               # Number of nodes to allocate
#SBATCH --ntasks-per-node=4     # Number of tasks per node
```

Multi-node Distribution Example

- C code prints process rank + hostname

```
#include <stdio.h>
#include <mpi.h>
#include <unistd.h> //  for gethostname

int main (int argc, char **argv) {
 int rank, size;
 MPI_Init(NULL, NULL);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);

 char hostname[256];
 gethostname(hostname, sizeof(hostname));
 printf("I am %d of %d running on %s\n", rank, size, hostname);

 MPI_Finalize();
 return 0;
}
```

Multi-node Distribution Example

- **Compile the program:**

```
module load oneapi/2023.2.0
mpiicx -o mpi_helloworld4x4 mpi_helloworld4x4.c
```

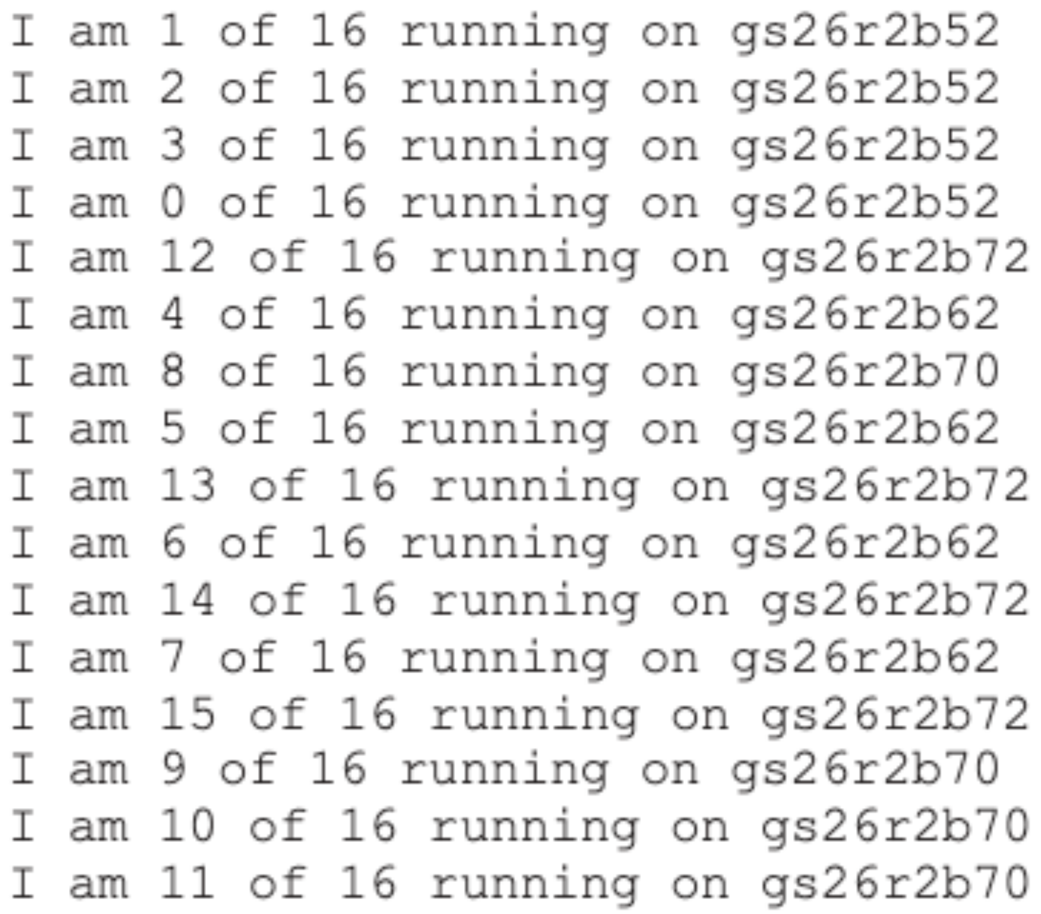
- **SLURM batch script:**

```
#!/bin/bash
#SBATCH -J mpi_helloworld_4x4
#SBATCH -t 00:15
#SBATCH -o %x_%J.out
#SBATCH -e %x_%J.err
#SBATCH --ntasks=16
#SBATCH --nodes=4
```

```
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=1
#SBATCH --exclusive
#SBATCH --account <account>
#SBATCH --qos gp_debug

srun ./mpi_helloworld4x4
```

Multi-node Distribution Example



```
I am 1 of 16 running on gs26r2b52
I am 2 of 16 running on gs26r2b52
I am 3 of 16 running on gs26r2b52
I am 0 of 16 running on gs26r2b52
I am 12 of 16 running on gs26r2b72
I am 4 of 16 running on gs26r2b62
I am 8 of 16 running on gs26r2b70
I am 5 of 16 running on gs26r2b62
I am 13 of 16 running on gs26r2b72
I am 6 of 16 running on gs26r2b62
I am 14 of 16 running on gs26r2b72
I am 7 of 16 running on gs26r2b62
I am 15 of 16 running on gs26r2b72
I am 9 of 16 running on gs26r2b70
I am 10 of 16 running on gs26r2b70
I am 11 of 16 running on gs26r2b70
```

– Tasks are spread across nodes:
  - Output shows ranks mapped to hostnames (gs26r2b52, …)

Task 4.2

- **Task 4.2 – Observe node distribution using hostnames**
  a) Modify the Hello World program to print the hostname (use `gethostname()`).
  b) Then, rerun the job with `--ntasks=16` and `--nodes=2` to distribute 16 MPI tasks over 2 nodes.
  c) Use `--ntasks-per-node=8` to enforce the distribution.
  d) Verify by analyzing the standard output that the results are correct and that the execution took place across two different nodes, as expected.
  e) Try running the program using both `srun` and `mpirun`.

Key MPI Concepts

- **SPMD model: Single Program, Multiple Data**
  - Same binary → different behavior per rank
- **Every program must:**
  - MPI_Init() → enter MPI world
  - MPI_Finalize() → clean up MPI resources
- **When a parallel MPI program starts, all participating processes are grouped into a default communicator (called MPI_COMM_WORLD).**

Communicators and Ranks

- `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`
  - Returns rank (0 … size-1)

- `MPI_Comm_size(MPI_COMM_WORLD, &size)`
  - Returns total processes

- **Enables:**
  - Work distribution
  - Conditional logic (rank 0 often handles I/O, coordination, …)

- **IMPORTANT: Each process gets:**
  - **Rank**: unique ID
  - **Size**: total number of processes

Point-to-Point Communication

- **Core MPI operations:**
  - `MPI_Send(...)` → send data
  - `MPI_Recv(...)` → receive data

- **Requirements:**
  - Same communicator
  - Matching source/destination ranks
  - Matching tags and message size/type

  - If `MPI_Recv` has no matching send
    → blocks indefinitely

Example: Sending Greetings

- Non-zero processes send greetings to rank 0

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

int main(int argc, char* argv[]) {
    int rank, size;
    char message[100];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank != 0) {
        // All processes except 0 send a message to process 0
        sprintf(message, "Greetings from process %d!", rank);
        MPI_Send(message, strlen(message) + 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        // Process 0 receives messages from all other processes
        for (int source = 1; source < size; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("Process 0 received: %s\n", message);
        }
    }

    MPI_Finalize();
    return 0;
}
```

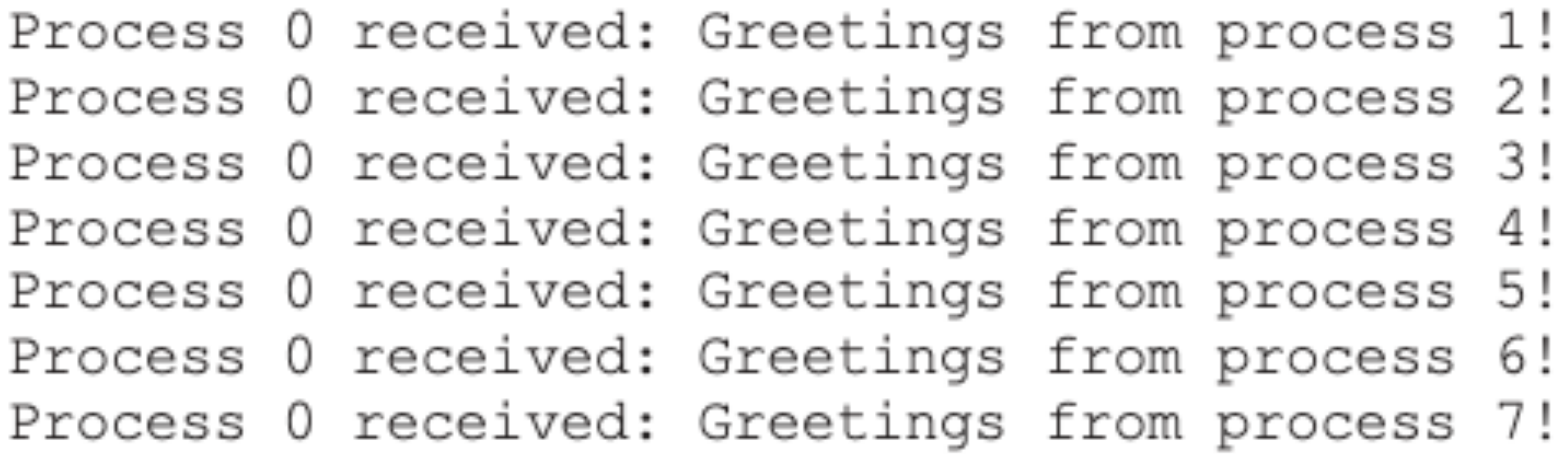
Example: Sending Greetings

- Rank 0 loops, receives messages sequentially
- Uses blocking MPI_Recv
  - ordered reception by rank

```
if (rank != 0) {
    // All processes except 0 send a message to process 0
    sprintf(message, "Greetings from process %d!", rank);
    MPI_Send(message, strlen(message) + 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
} else {
    // Process 0 receives messages from all other processes
    for (int source = 1; source < size; source++) {
        MPI_Recv(message, 100, MPI_CHAR, source, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 0 received: %s\n", message);
    }
}
```

Example: Sending Greetings

- **Sample output with 8 tasks:**



```
Process 0 received: Greetings from process 1!
Process 0 received: Greetings from process 2!
Process 0 received: Greetings from process 3!
Process 0 received: Greetings from process 4!
Process 0 received: Greetings from process 5!
Process 0 received: Greetings from process 6!
Process 0 received: Greetings from process 7!
```

Task 4.3

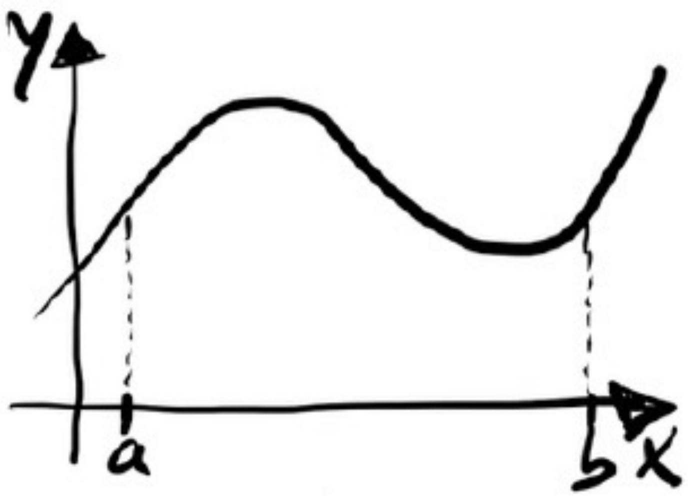
- **Task 4.3 – Point-to-point communication**
  a) Compile and run the MPI program using `MPI_Send` and `MPI_Recv` where all worker processes send a greeting to process 0.
  b) Create 16 tasks.
  c) Confirm that process 0 receives the messages in rank order.

4.3 Case Study: Parallelizing the Trapezoidal Rule

Case Study: Trapezoidal Rule (MPI)

- **Goal:**
  - approximate a definite integral with the trapezoidal rule
- **Strategy:**
  - split $[a, b]$ into n equal subintervals; sum trapezoids
- **Parallel idea:**
  - each process sums a subset of trapezoids, then reduce

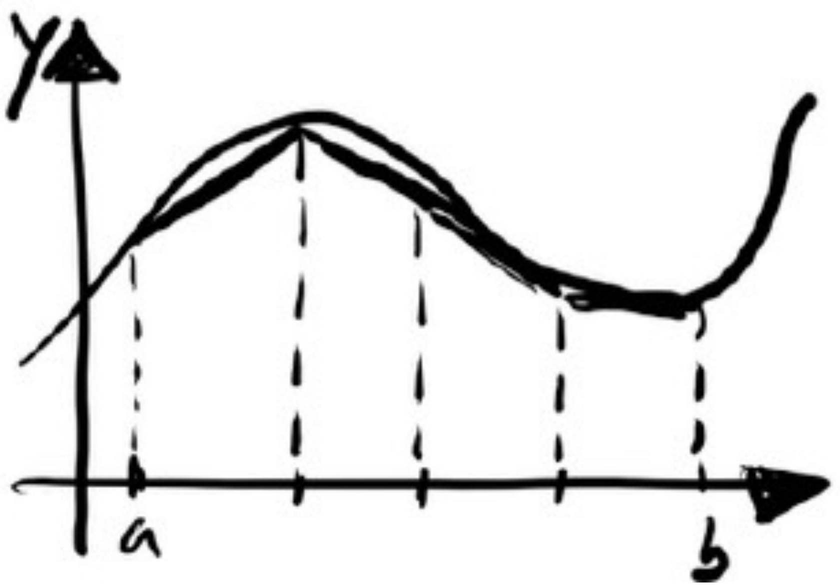
$$\int_a^b f(x)dx$$



Trapezoidal Rule

- **Pseudo-code:**

```
/* Input: a, b, n */
h = (b - a)/n;
approx = (f(a) + f(b))/2.0;
for i = 0 .. n-1:
    x_i = a + i*h
    approx += f(x_i)
approx *= h
```



- **Accuracy ↑ with n**

Serial Example: π

- **We approximate:**

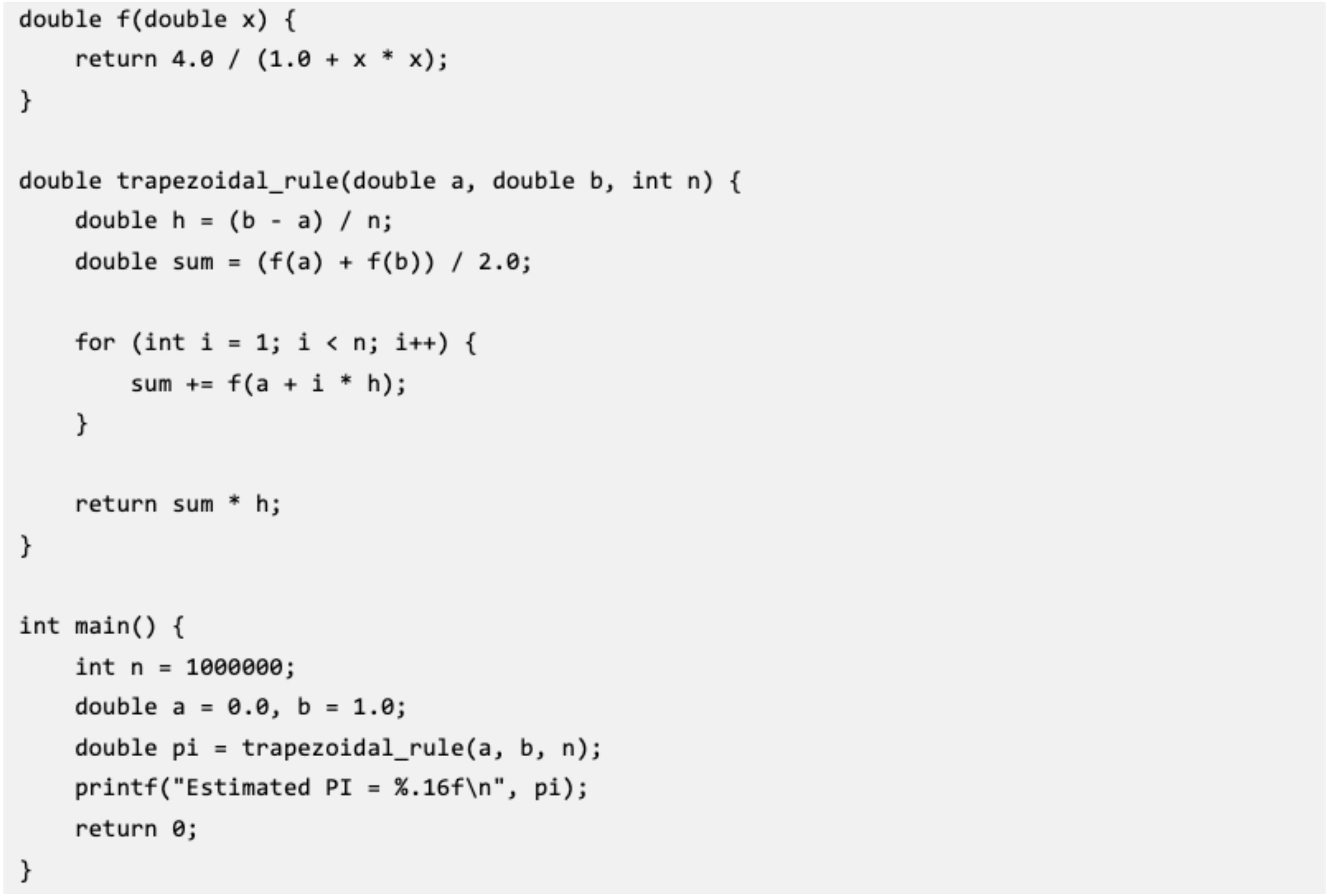
$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

- **C implementation:**

```
double f(double x){ return 4.0/(1.0+x*x); }

double trap(double a,double b,int n){
  double h=(b-a)/n, sum=(f(a)+f(b))/2.0;
  for(int i=1;i<n;i++) sum += f(a + i*h);
  return sum*h;
}
```

Serial Example: π



```
double f(double x) {
    return 4.0 / (1.0 + x * x);
}

double trapezoidal_rule(double a, double b, int n) {
    double h = (b - a) / n;
    double sum = (f(a) + f(b)) / 2.0;

    for (int i = 1; i < n; i++) {
        sum += f(a + i * h);
    }

    return sum * h;
}

int main() {
    int n = 1000000;
    double a = 0.0, b = 1.0;
    double pi = trapezoidal_rule(a, b, n);
    printf("Estimated PI = %.16f\n", pi);
    return 0;
}
```

Serial Example: π

- **Compile and Run:**

```
$ module load intel
$ icx -O3 pi_seq.c -o pi_seq
$ ./pi_seq
```

```
Estimated PI = 3.1415926535895844
```

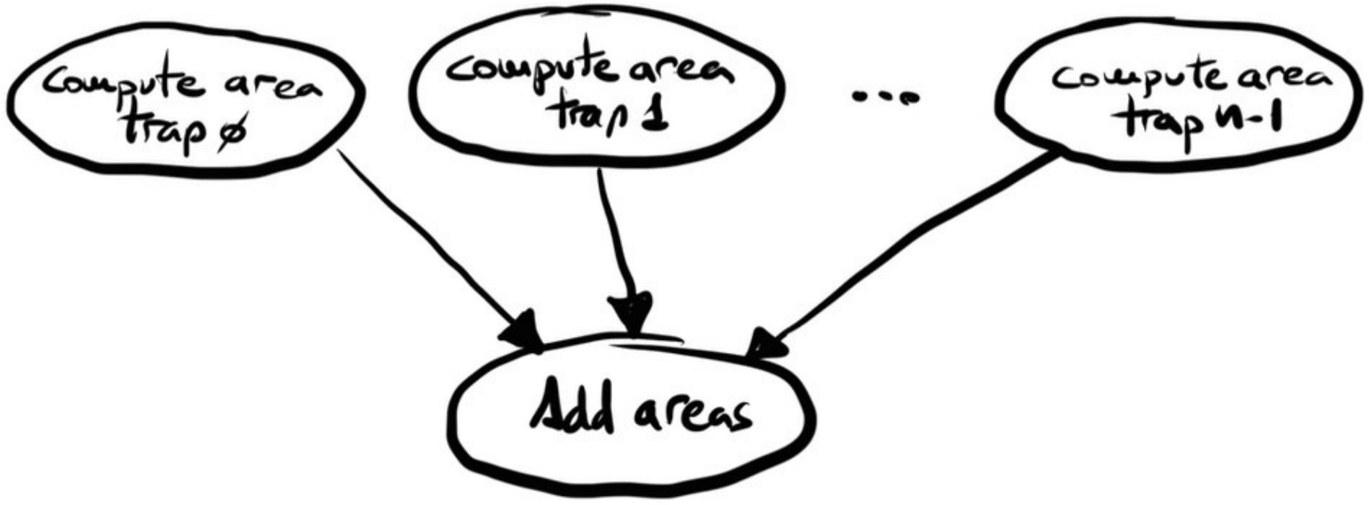
Task 4.4

- **Task 4.4 – Write and Run The Sequential Program That Estimate π**

  Use a standard C compiler and verify that the estimated value of π is printed to the terminal. This simple test ensures that your code is working before proceeding to the parallel MPI version.

Parallelization Plan

- **Partition work across p processes:**
  - Each process computes n/p trapezoids over its subinterval
- **Communication pattern:**
  - Local compute → send partial sums → root does final sum
- **Parallel reduction (manual or MPI_Reduce later)**



MPI Parallel Implementation

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

double f(double x) {
    return 4.0 / (1.0 + x * x);
}

double local_trap(double a, double b, int local_n) {
    double h = (b - a) / local_n;
    double sum = (f(a) + f(b)) / 2.0;

    for (int i = 1; i < local_n; i++) {
        sum += f(a + i * h);
    }

    return sum * h;
}
```

MPI Parallel Implementation

```
int main(int argc, char** argv) {
    int rank, size, n;
    double a = 0.0, b = 1.0;
    double local_a, local_b, h;
    int local_n;
    double local_result, total_result;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

MPI Parallel Implementation

```
// Ensuring that n is divisible by the number of MPI processes
n = 1000000;
n = (n / size) * size;

h = (b - a) / n;
local_n = n / size;
local_a = a + rank * local_n * h;
local_b = local_a + local_n * h;

local_result = local_trap(local_a, local_b, local_n);
```

MPI Parallel Implementation

```
    if (rank == 0) {
        total_result = local_result;
        for (int source = 1; source < size; source++) {
            double temp;
            MPI_Recv(&temp, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            total_result += temp;
        }
        printf("Estimated PI = %.16f\n", total_result);
    } else {
        MPI_Send(&local_result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

Compile & Run (MPI)

- Compile

```
module load oneapi/2023.2.0
mpiicx pi_mpi.c -o pi_mpi
```

- SLURM script (CPU only):

```
#!/bin/bash
#SBATCH --job-name=pi_mpi
#SBATCH --output=%x_%j.out
#SBATCH --error=%x_%j.err
#SBATCH --ntasks=8
#SBATCH --cpus-per-task=1
#SBATCH --time=00:05:00
#SBATCH --exclusive
#SBATCH --account=<account>
#SBATCH --qos=gp_debug

srun ./pi_mpi
```

Task 4.5

- **Task 4.5 – Write and Run the Parallel MPI Code to Estimate the Value of π**

  Execute the parallel version using 16 processes with SLURM, and verify that the estimated value of π is correct, as this will serve as the basis for the following tasks.

Taking Timing

- **We need performance metrics**
  - measure execution time of code under study

- **Not just total program time; often a specific region**

- **Typical timers:**
  - MPI: MPI_Wtime()
  - OpenMP: omp_get_wtime()
  - **POSIX (portable):** gettimeofday()

POSIX gettimeofday() [Appendix 17.4]

- **Portable Wall-Clock**
- **Works in serial, MPI, OpenMP, mixed codes**
- **Resolution: microseconds (10^{-6} s)**
- **Prototype:**

```
#include <sys/time.h>
int gettimeofday(struct timeval * tv, struct timezone * tz);
```

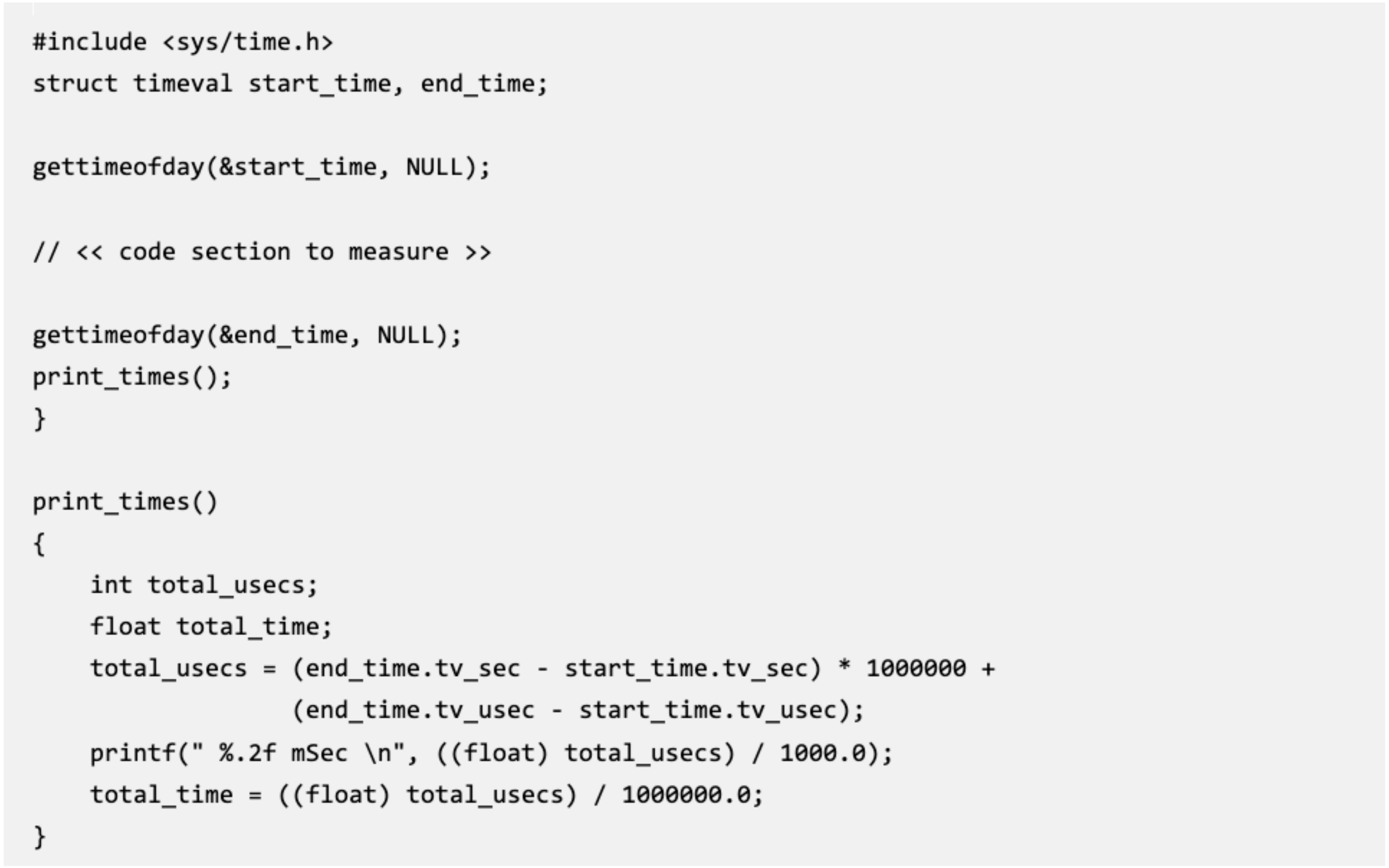
- **Structure:**

```
struct timeval {
    time_t tv_sec;      /* seconds */
    suseconds_t tv_usec; /* microseconds */
};
```

```
struct timezone {
    int tz_minuteswest; /* minutes west of Greenwich */
    int tz_dsttime;     /* type of DST correction */
};
```

Timing Pattern

- **(Copy/Paste Template)**



```
#include <sys/time.h>
struct timeval start_time, end_time;

gettimeofday(&start_time, NULL);

// << code section to measure >>

gettimeofday(&end_time, NULL);
print_times();
}

print_times()
{
    int total_usecs;
    float total_time;
    total_usecs = (end_time.tv_sec - start_time.tv_sec) * 1000000 +
                  (end_time.tv_usec - start_time.tv_usec);
    printf(" %.2f mSec \n", ((float) total_usecs) / 1000.0);
    total_time = ((float) total_usecs) / 1000000.0;
}
```

Mini Example (fills a matrix)

```c
/* timesample.c */
#include <stdio.h>
#include <sys/time.h>
#include <stdlib.h>

#define SIZE 1000
typedef double matrix[SIZE][SIZE];
matrix m1;

struct timeval start_time, end_time;

static void foo(void) {
    int i, j;
    for (i = 0; i < SIZE; ++i)
        for (j = 0; j < SIZE; ++j)
            m1[i][j] = 1.0;
}
```

Mini Example (fills a matrix)

```
void print_times() {
    int total_usecs;
    float total_time;
    total_usecs = (end_time.tv_sec - start_time.tv_sec) * 1000000 +
                  (end_time.tv_usec - start_time.tv_usec);
    printf(" %.2f mSec \n", ((float) total_usecs) / 1000.0);
    total_time = ((float) total_usecs) / 1000000.0;
}
```

Mini Example (fills a matrix)

```
int main() {
    int i;

    gettimeofday(&start_time, NULL);

    for (i = 0; i < 10; ++i) {
        foo();
    }

    gettimeofday(&end_time, NULL);
    print_times();

    return 0;
}
```

Compile & Run (example output)

```
$ gcc -O3 -o timesample timesample.c
$ ./timesample
```

```
3.81 mSec
```

How does it scale with varying numbers of parallel processes?

- **Compile**

```
module load oneapi/2023.2.0


icx -O3 pi_seq_timed.c -o pi_seq_timed
mpiicx -O3 pi_mpi_timed.c -o pi_mpi_timed
```

The files pi_mpi_timed.c and pi_seq_timed.c are the same as the previous versions, but now include the necessary code to measure execution time using the gettimeofday system call.

How does it scale with varying numbers of parallel processes?

- `pi_mpi_scaling.slurm`

```
#!/bin/bash
#SBATCH --job-name=pi_scaling
#SBATCH --output=%x_%j.out
#SBATCH --error=%x_%j.err
#SBATCH --ntasks=64
#SBATCH --cpus-per-task=1
#SBATCH --time=00:15:00
#SBATCH --exclusive
#SBATCH --account=<account>
#SBATCH --qos=gp_debug

./pi_seq_timed

for P in 2 4 8 16 32 64
do
  echo ""
  srun --ntasks=$P ./pi_mpi_timed
done
```

How does it scale with varying numbers of parallel processes?

Processes	Time (ms)	Speedup	Efficiency (%)
seq	4430.33	1.00	100.0%
2	2394.17	1.85	92.4%
4	1198.30	3.70	92.6%
8	601.04	7.37	92.1%
16	308.02	14.38	89.9%
32	160.61	27.58	86.2%
64	216.27	20.49	32.0%

How does it scale with varying numbers of parallel processes?

- We can observe that up to 32 processes, the efficiency remains reasonable; however, beyond that point, it drops significantly.
- In fact, the execution time with 64 processes is even higher than with 32, indicating diminishing returns.
- This raises the question: is the program's limited parallelism the cause?

→ See Amdhl’s Law & Gustafson’s Law sections in the book

Task 4.6 (from book)

- ~~**Task 4.6 – Analysis using Gustafson's Law to estimate the value of π**~~

  ~~Run the parallel version of the program that estimates the value of π using up to 128 processes, and analyze the results according to Gustafson's Law, following the same methodology presented earlier in this chapter.~~

Task 4.6 (new)

- **Task 4.6 new – Scalability Analysis of the MPI Trapezoidal Rule**
  a) Compile the provided sequential and parallel timed versions of the trapezoidal rule program (pi_seq_timed.c and pi_mpi_timed.c).
  b) Submit the SLURM script pi_mpi_scaling.slurm, which runs the parallel program with different numbers of processes (2, 4, 8, 16, 32) under the same job allocation.
  c) Compare execution times, calculate speedup and efficiency, and discuss at which point adding more processes no longer improves performance.
  d) Reflect on possible causes such as communication overhead or limited parallelism.

MPI I/O Basics

- **stdout from many ranks can interleave (unordered prints)**
- **stdin typically only from rank 0**
  - Distribute input to others:

```
void get_input(int* n_p, int my_rank, int comm_sz) {
    if (my_rank == 0) {
        printf("Enter number of intervals: ");
        scanf("%d", n_p);

        for (int dest = 1; dest < comm_sz; dest++) {
            MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

  - Later: simpler with **MPI_Bcast**

Barrier Synchronization

- Ensure all ranks reach a point before proceeding:

```
int MPI_Barrier(MPI_Comm comm);
```

- Use around timed regions to align start/stop
- Caveat: overuse introduces idle waiting → hurt perf

4.4 Collective Communication Primitives

Collective Communication in MPI

- **Point-to-point operations (`MPI_Send`, `MPI_Recv`) are flexible but …**

- **Many parallel algorithms need global communication patterns**

- **MPI provides collective operations:**
  - Simplify programming and code maintenance
  - Highly optimized in most implementations

- **All processes in the communicator must call them, otherwise deadlock**

MPI_Bcast (Broadcast)

- Used when one process (root) sends the same data to all others
- Function prototype:

```
MPI_Reduce(&local_result, &total_result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

- Example use case: input read by process 0 distributed to all others
- More elegant and efficient than multiple `MPI_Send` / `MPI_Recv`

Reduction Operations

- **Goal: combine partial results from all processes**
- **Traditional approach: manual send/receive loop**
- **Collective version:**

```
MPI_Reduce(&local_result, &total_result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

- **Variants:**
  - `MPI_Reduce` → result only at root
  - `MPI_Allreduce` → result delivered to all processes
- **Operations supported: `MPI_SUM`, `MPI_MAX`, `MPI_MIN`, ...**

MPI_Scatter and MPI_Gather

- **`MPI_Scatter`**
  - root splits dataset into chunks → each process gets one
- **`MPI_Gather`**
  - each process sends results → root collects into array
- **Typical workflow:**
  - Root distributes a vector (`MPI_Scatter`)
  - Processes compute locally
  - Root collects updated vector (`MPI_Gather`)
- **Example: parallel vector sum (available in repository)**

Task 4.7

- **Task 4.7 – Experimenting with Scatter and Gather**
  a) Take the example provided in the book's GitHub repository and run it.
  b) Once you have verified that it executes correctly, modify the code to test different vector sizes.
  c) Analyze how the size of the vector influences the program's speedup, efficiency, and scalability.

Advanced Collectives (I)

- **`MPI_Allgather`**
  - Every process gathers contributions from all others
  - Equivalent to Gather + Broadcast, but more efficient
  - Example: sharing local statistics with all processes

- **`MPI_Alltoall`**
  - Each process sends/receives different data to/from all others
  - General redistribution primitive, used in sorting, domain decomposition, matrix transpose

Advanced Collectives (II)

- **`MPI_Reduce_scatter`**
  - Combines reduction and scatter in a single step
  - Each process receives only its needed portion of the reduced result
  - Useful in distributed dot products, localized data aggregation

- **Advantages of advanced collectives:**
  - Reduce code complexity
  - Leverage optimized implementations for scalability

Pc: MPI

- **Tasks included:**
  - Task 4.1
  - Task 4.2
  - Task 4.3
  - Task 4.4
  - Task 4.5
  - Task 4.6 (new)
  - Task 4.7
- **Deliverable:**
  - Upload a single PDF (per group) to the intranet racó@FIB containing one slide per task. Each slide should report results or briefly explain how the task was completed.
  - In class (evaluation day), one group (chosen at random) will give a *"elevator pitch"-style* or PK presentation — clear, concise, and straight to the point.

These slides are based on the book
Supercomputing for Artificial Intelligence **(Torres, 2025).**
more info: https://torres.ai/hpc4aibook/

PDF slides are freely available for students.

Teachers using this book may request the PPTX version for classroom use.