

Programmazione di Sistemi ~~Embedded e~~ Multicore

Teacher: Daniele De Sensi

Recap

- The laws of physics have brought us to the doorstep of multicore technology. Existing applications can use >200,000 GPUs at once
- Serial programs typically don't benefit from multiple cores.
- Automatic parallel program generation from serial program code isn't the most efficient approach to get high performance from multicore computers.
- Learning to write parallel programs involves learning how to coordinate the cores.
- Several APIs depending on the type of parallel system
- Parallel programs are more complex than their serial counterpart and therefore, require sound program techniques and development.
- Keep in mind how the underlying hardware works
- C allows control at a low level, which is needed (especially on the shared memory part!)

Distributed memory programming with MPI

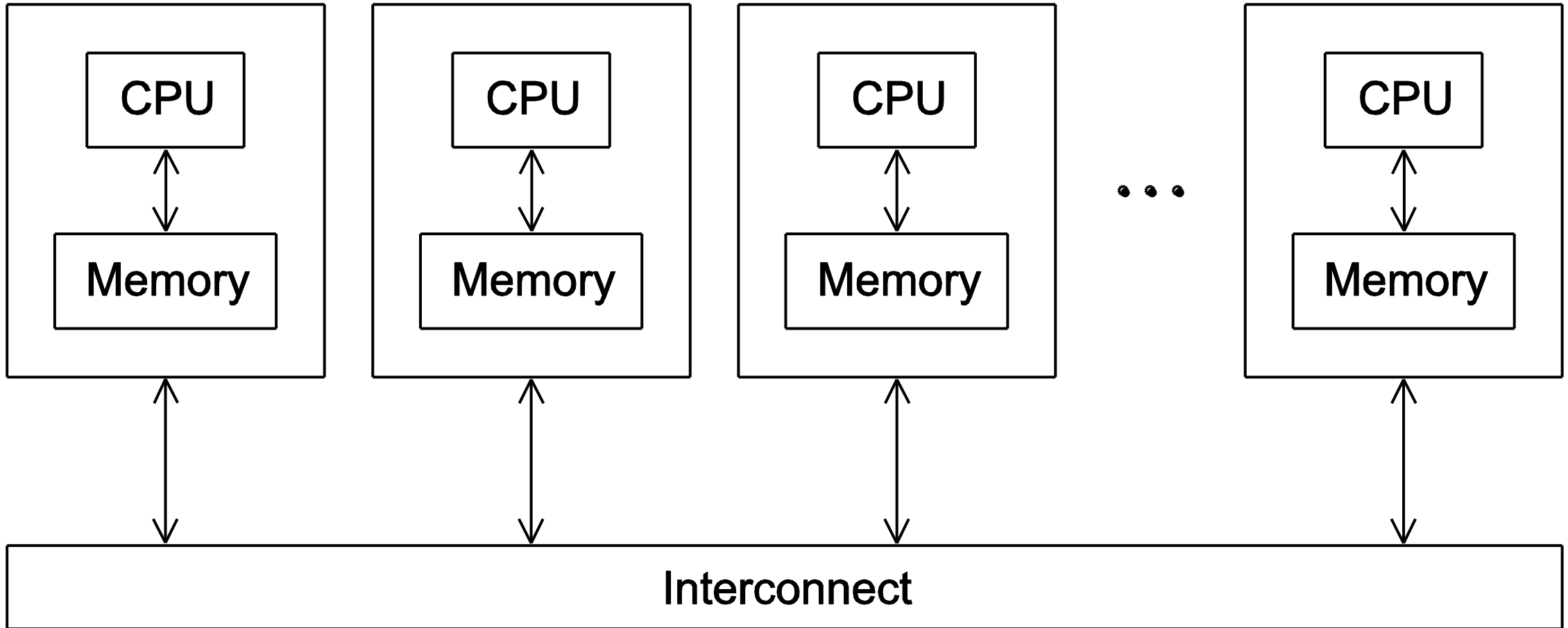
3

What about Chapter 2?

CHAPTER 2	Parallel hardware and parallel software	17	2.5	Input and output	60
2.1	Some background	17	2.5.1	MIMD systems	60
2.1.1	The von Neumann architecture	17	2.5.2	GPUs	61
2.1.2	Processes, multitasking, and threads	19	2.6	Performance	61
2.2	Modifications to the von Neumann model	20	2.6.1	Speedup and efficiency in MIMD systems	61
2.2.1	The basics of caching	21	2.6.2	Amdahl’s law	65
2.2.2	Cache mappings	23	2.6.3	Scalability in MIMD systems	66
2.2.3	Caches and programs: an example	24	2.6.4	Taking timings of MIMD programs	67
2.2.4	Virtual memory	25	2.6.5	GPU performance	70
2.2.5	Instruction-level parallelism	27	2.7	Parallel program design	71
2.2.6	Hardware multithreading	30	2.7.1	An example	71
2.3	Parallel hardware	31	2.8	Writing and running parallel programs	75
2.3.1	Classifications of parallel computers	31	2.9	Assumptions	77
2.3.2	SIMD systems	31	2.10	Summary	78
2.3.3	MIMD systems	34	2.10.1	Serial systems	78
2.3.4	Interconnection networks	37	2.10.2	Parallel hardware	79
2.3.5	Cache coherence	45	2.10.3	Parallel software	81
2.3.6	Shared-memory vs. distributed-memory	49	2.10.4	Input and output	82
2.4	Parallel software	49	2.10.5	Performance	82
2.4.1	Caveats	50	2.10.6	Parallel program design	83
2.4.2	Coordinating the processes/threads	50	2.10.7	Assumptions	84
2.4.3	Shared-memory	51	2.11	Exercises	84
2.4.4	Distributed-memory	55			
2.4.5	GPU programming	58			
2.4.6	Programming hybrid systems	60			

We will refer to parts of it when needed

Distributed Memory Systems



Single-Program Multiple-Data (SPMD)

- We compile one program
- The same program is executed by multiple processes
- You use if-else to specify what each process must do (similar to what happens when you fork a process)
- E.g.:
 - If I am process number 0, do X
 - If I am process number 1, do Y
 - etc...
- Processes do not share memory, so communications happen through *message passing*

Hello World!

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

```
int main(void) {  
    printf("hello, world\n");  
  
    return 0;  
}
```

Hello World! (v0)

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

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

```
int main(void) {  
    MPI_Init(NULL, NULL);  
    printf("hello, world\n");  
    MPI_Finalize();  
    return 0;  
}
```


MPI Programs

- Written in C (for this course, but it has bindings in many languages)
 - Has main.
 - Uses `stdio.h`, `string.h`, etc.
- Need to add `mpi.h` header file.
- Identifiers defined by MPI start with "MPI_".
- First letter following underscore is uppercase.
 - For function names and MPI-defined types.
 - Helps to avoid confusion.

MPI Components

- MPI_Init

- Tells MPI to do all the necessary setup.

```
int MPI_Init(  
    int*      argc_p  /* in/out */,  
    char***   argv_p  /* in/out */);
```

- MPI_Finalize

- Tells MPI we're done, so clean up anything allocated for this program.

```
int MPI_Finalize(void);
```

Basic Outline

```
. . .  
#include <mpi.h>  
  
. . .  
int main(int argc, char* argv[]) {  
    . . .  
    /* No MPI calls before this */  
    MPI_Init(&argc, &argv);  
    . . .  
    MPI_Finalize();  
    /* No MPI calls after this */  
    . . .  
    return 0;  
}
```

Questions?

Compilation

wrapper script to compile *source file*

`mpicc -g -Wall -o mpi_hello mpi_hello.c`

produce debugging information *turns on all warnings* *create this executable file name (as opposed to default a.out)*

```
graph TD; WS[wrapper script to compile] --> mpicc; SF[source file] --> mpi_hello_c[mpi_hello.c]; mpicc --> g[-g]; mpicc --> Wall[-Wall]; mpicc --> o[-o]; mpicc --> mpi_hello[mpi_hello]; g --> g_desc[produce debugging information]; Wall --> Wall_desc[turns on all warnings]; o --> o_desc[create this executable file name (as opposed to default a.out)];
```

Execution

```
mpiexec -n <number of processes> <executable>
```

```
mpiexec -n 1 ./mpi_hello
```



run with 1 process

```
mpiexec -n 4 ./mpi_hello
```



run with 4 processes

Execution

```
mpiexec -n 1 ./mpi_hello
```

hello, world

```
mpiexec -n 4 ./mpi_hello
```

hello, world

hello, world

hello, world

hello, world

Debugging

Parallel debugging is trickier than debugging serial programs

- Many processes computing; getting the state of one failed process is usually hard

Using MPI with ddd (or gdb) on one process:

```
mpiexec -n 4 ./test : -n 1 ddd ./test : -n 1 ./test
```

- Launches the 5th process under “ddd” and all other processes normally

Identifying MPI processes

- Common practice to identify processes by nonnegative integer **ranks**.
- p processes are numbered $0, 1, 2, \dots, p-1$

Communicators

- A collection of processes that can send messages to each other.
- MPI_Init defines a communicator that consists of all the processes created when the program is started.
- Called `MPI_COMM_WORLD`.

Communicators

```
int MPI_Comm_size(  
    MPI_Comm comm      /* in */,  
    int* comm_sz_p     /* out */);
```


number of processes in the communicator

```
int MPI_Comm_rank(  
    MPI_Comm comm      /* in */,  
    int* my_rank_p     /* out */);
```


my rank (the process making this call)

Hello World! (v1)

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

int main(void) {
    int comm_sz, my_rank;
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    printf("hello, world from process %d out of %d\n", my_rank, comm_sz);
    MPI_Finalize();
    return 0;
}
```

Hello World! (v1)

```
mpiexec -n 4 ./mpi_hello
```

```
hello, world from process 2 out of 4  
hello, world from process 3 out of 4  
hello, world from process 0 out of 4  
hello, world from process 1 out of 4
```

Why?


```
mpiexec -n 4 ./mpi_hello
```

```
hello, world from process 1 out of 4  
hello, world from process 0 out of 4  
hello, world from process 3 out of 4  
hello, world from process 2 out of 4
```

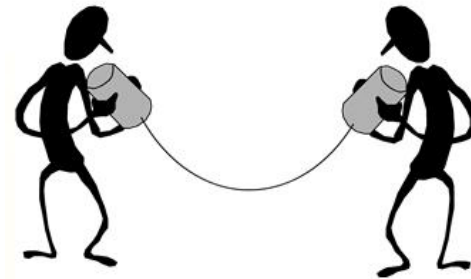
Questions?

Communication

```
int MPI_Send(  
    void*      msg_buf_p    /* in */,  
    int        msg_size     /* in */,  
    MPI_Datatype msg_type    /* in */,  
    int        dest         /* in */,  
    int        tag          /* in */,  
    MPI_Comm   communicator /* in */);
```




ATTENTION! Number of
elements, not number of bytes

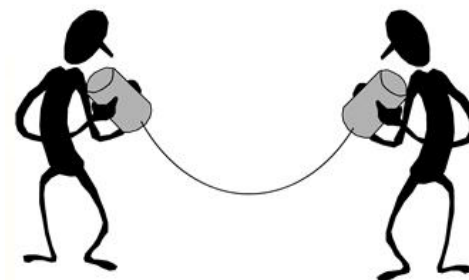


Communication

```
int MPI_Recv(  
    void*      msg_buf_p    /* out */,  
    int        buf_size     /* in  */,  
    MPI_Datatype buf_type    /* in  */,  
    int        source       /* in  */,  
    int        tag          /* in  */,  
    MPI_Comm   communicator /* in  */,  
    MPI_Status* status_p    /* out */);
```



ATTENTION! Number of
elements, not number of bytes



Hello World! (v2)

```
1 #include <stdio.h>
2 #include <string.h>  /* For strlen */
3 #include <mpi.h>     /* For MPI functions , etc */
4
5 const int MAX_STRING = 100;
6
7 int main(void) {
8     char    greeting[MAX_STRING];
9     int     comm_sz;  /* Number of processes */
10    int     my_rank;  /* My process rank */
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16    if (my_rank != 0) {
17        sprintf(greeting, "Greetings from process %d of %d!",
18                my_rank, comm_sz);
19        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20                 MPI_COMM_WORLD);
21    } else {
22        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
23        for (int q = 1; q < comm_sz; q++) {
24            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25                     0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26            printf("%s\n", greeting);
27        }
28    }
29
30    MPI_Finalize();
31    return 0;
32 } /* main */
```

Hello World! (v2)

```
mpirun -n 4 ./mpi_hello
```

Greetings from process 0 of 4

Greetings from process 1 of 4

Greetings from process 2 of 4

Greetings from process 3 of 4

Questions?

Sending order

- MPI requires that messages be **nonovertaking**. This means that if process q sends two messages to process r , then the first message sent by q must be available to r before the second message.
- However, there is no restriction on the arrival of messages sent from different processes
- No restriction on the network (?)

Data types

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Communicators

- MPI_Init defines a communicator called `MPI_COMM_WORLD`.
- MPI provides functions for creating new communicators
- User communicators are useful to integrate complex functionalities together

Communicators

- Suppose you have 2 MPI independent libraries of functions,
- they don't communicate with each other, but they do communicate internally.
- We can do it with tags, assigning tags $[1, n]$ and tags $[n+1, m]$ tot the two libraries OR
- we simply pass one communicator to one library functions and a different communicator to the other library.

Message matching

Rank q

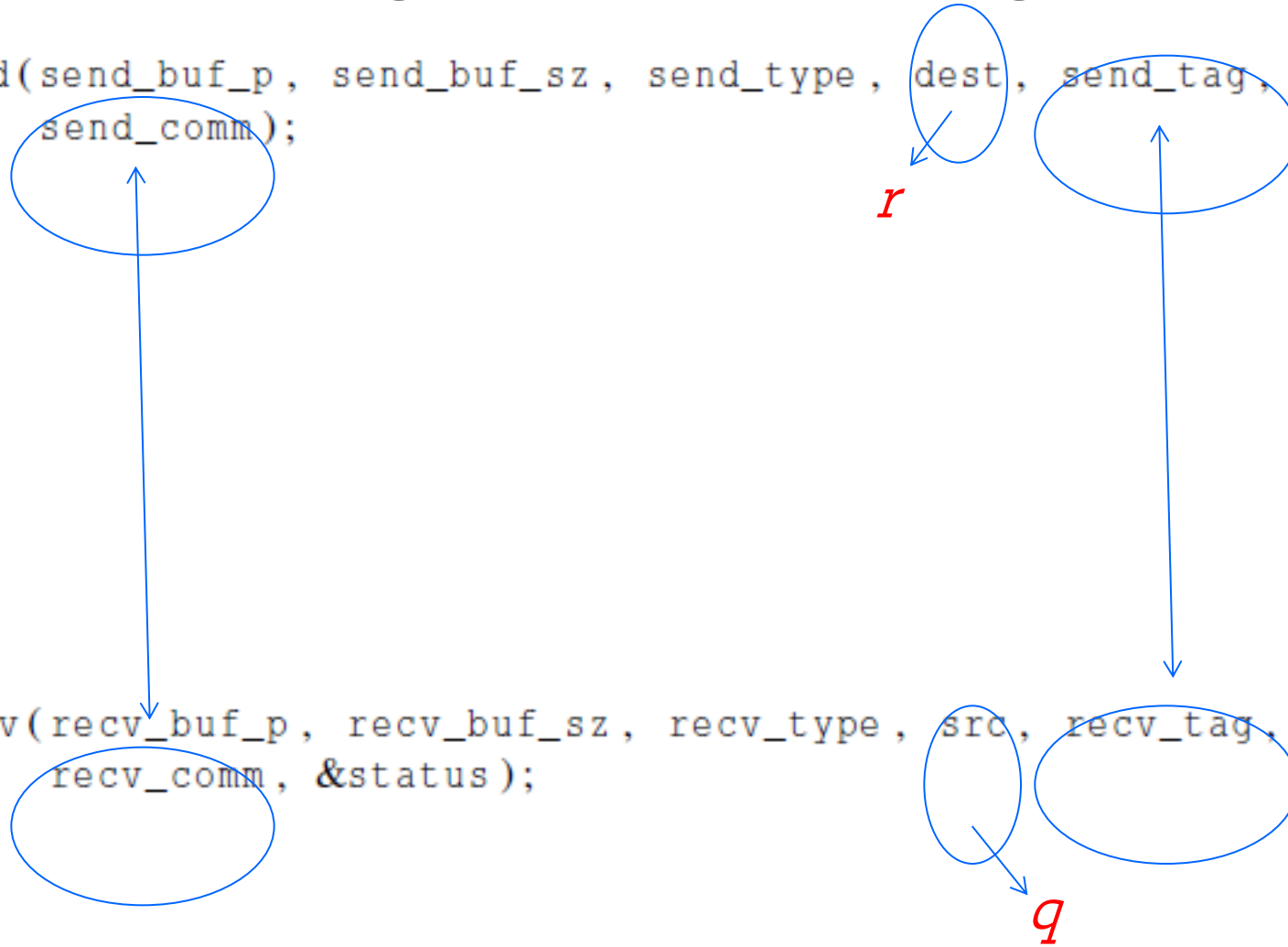
```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,  
send_comm);
```

r

Rank r

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,  
recv_comm, &status);
```

q

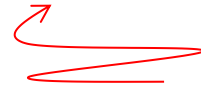


Message matching

- Message is *successfully* received if:
 - `recv_type = send_type`
 - `recv_buf_sz > send_buf_sz`
- A receiver can get a message without knowing:
 - the amount of data in the message,
 - the sender of the message (*`MPI_ANY_SOURCE`*),
 - or the tag of the message (*`MPI_ANY_TAG`*),

status_p argument

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,  
recv_comm, &status);
```



MPI_Status*



`MPI_Status* status;`

`status.MPI_SOURCE`

`status.MPI_TAG`

MPI_SOURCE

MPI_TAG

MPI_ERROR

How much data am I receiving?

```
int MPI_Get_count(  
    MPI_Status* status_p /* in */,  
    MPI_Datatype type /* in */,  
    int* count_p /* out */);
```

Issues with send and receive

- Exact behavior is determined by the MPI implementation.
- MPI_Send may **behave differently** with regard to buffer size, cutoffs and blocking.
- MPI_Recv always blocks until a matching message is received.
- Even if you know your MPI implementation, stick to what defined by the standard (i.e., don't assume that the send returns immediately for small buffers)! Otherwise your code will not be portable.

Warnings

- If a process tries to receive a message and there's no matching send, then the process will **hang**.
- if a call to MPI_Send blocks and there's no matching receive, then the sending process can hang.
- if, a call to MPI_Send is buffered and there's no matching receive, then the message will be lost.
- if the rank of the destination process is the same as the rank of the source process, a process will hang, or, perhaps worse, the receive may match *another* send.

Questions?

Parallel Program Design

Foster's methodology

1. **Partitioning**: divide the computation to be performed and the data operated on by the computation into small tasks.

The focus here should be on identifying tasks that can be executed in parallel.

Foster's methodology

2. **Communication**: determine what communication needs to be carried out among the tasks identified in the previous step.

Foster's methodology

3. **Agglomeration or aggregation:** combine tasks and communications identified in the first step into larger tasks.

For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.

Why?

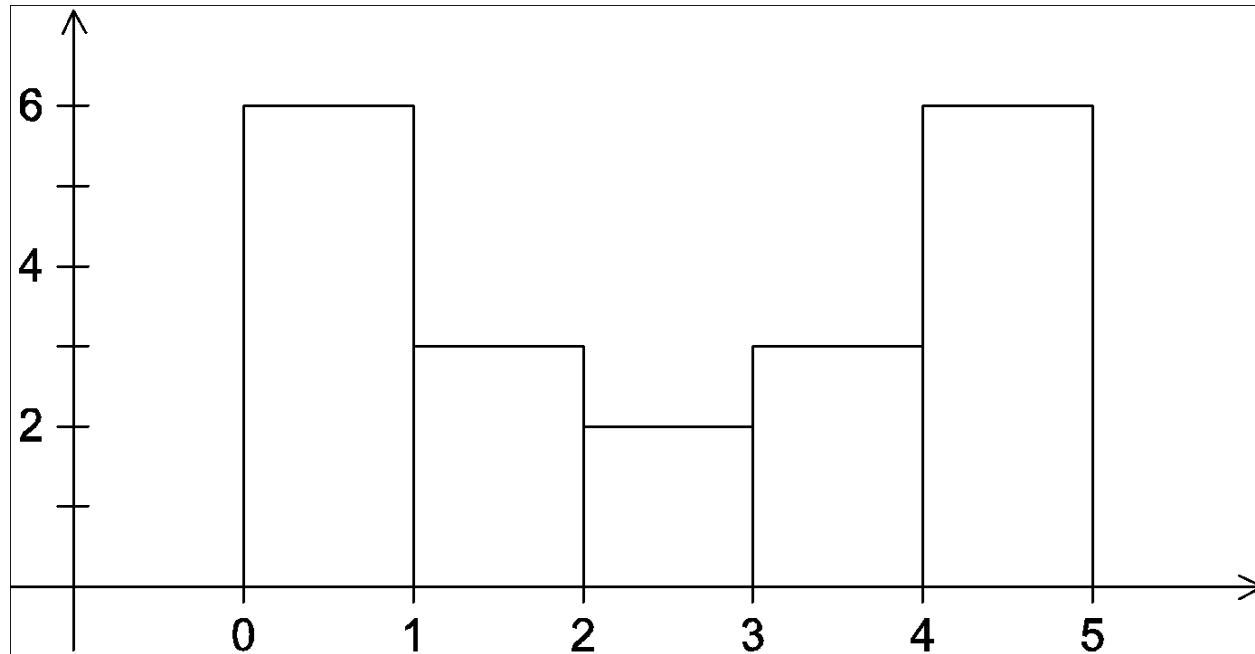
Foster's methodology

4. **Mapping**: assign the composite tasks identified in the previous step to processes/threads.

This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.

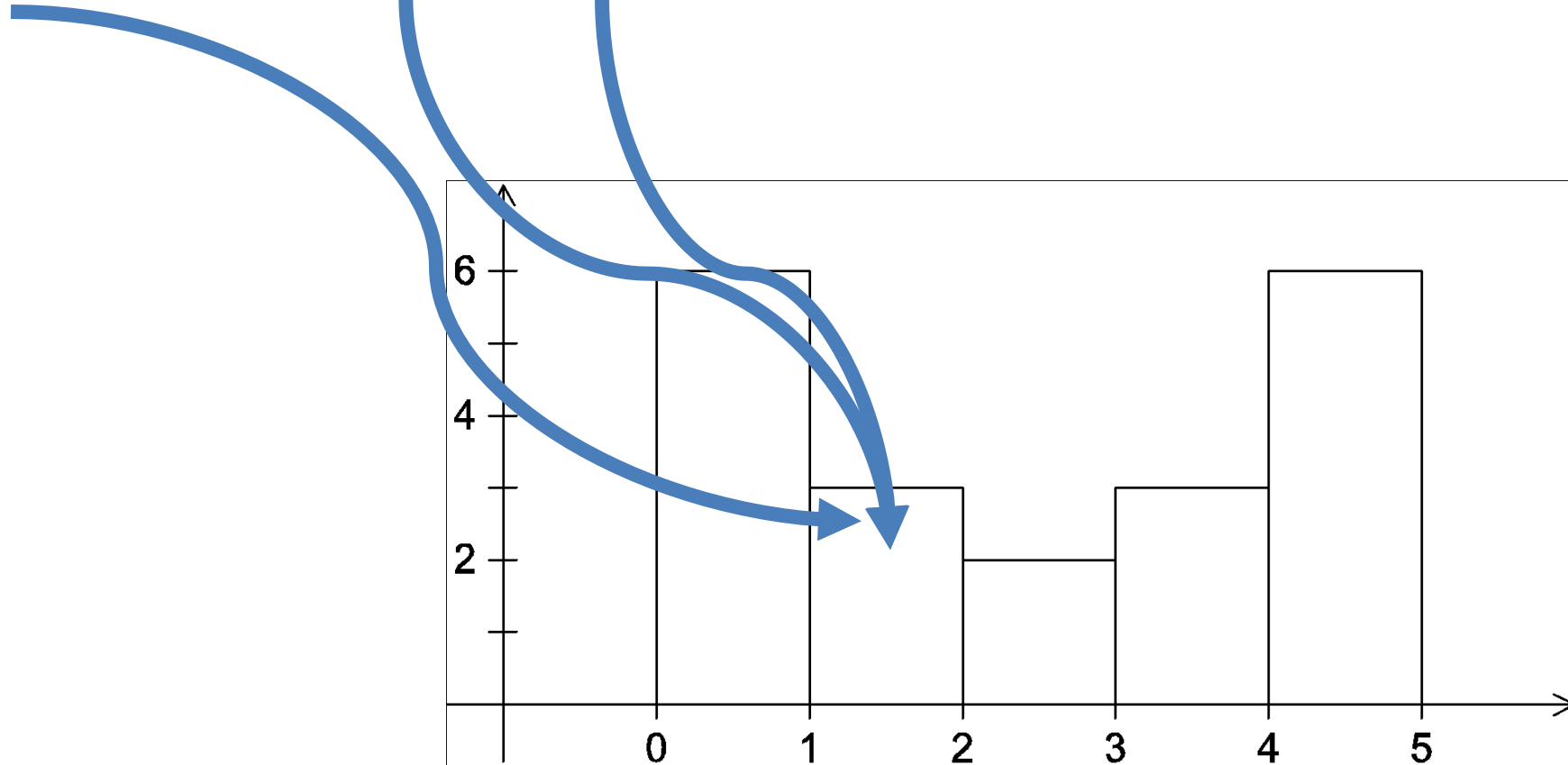
Example - histogram

1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9



Example - histogram

1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9



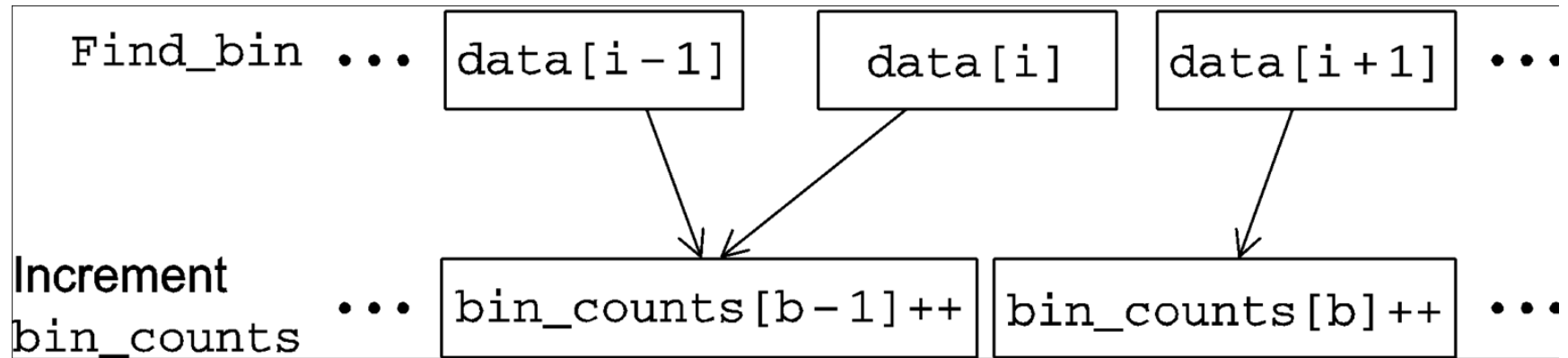
Serial program – input

1. The number of measurements: `data_count`
2. An array of `data_count` floats: `data`
3. The minimum value for the bin containing the smallest values: `min_meas`
4. The maximum value for the bin containing the largest values: `max_meas`
5. The number of bins: `bin_count`

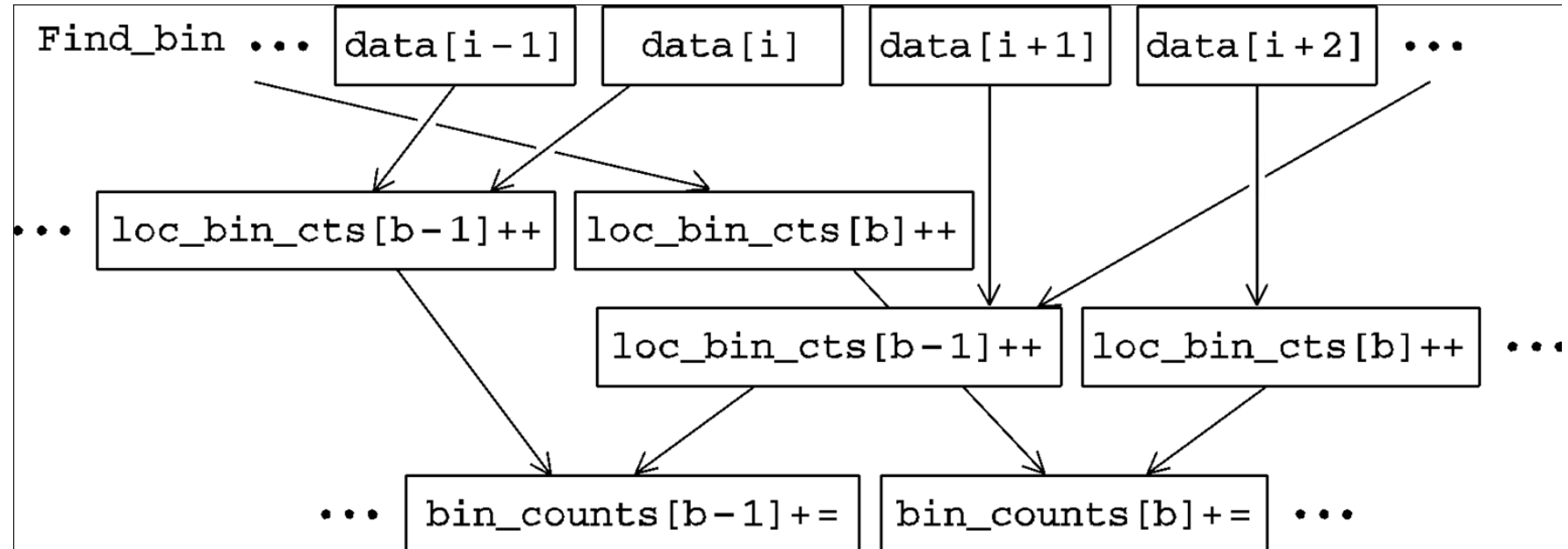
Serial program - output

1. `bin_maxes` : an array of `bin_count` floats
2. `bin_counts` : an array of `bin_count` ints

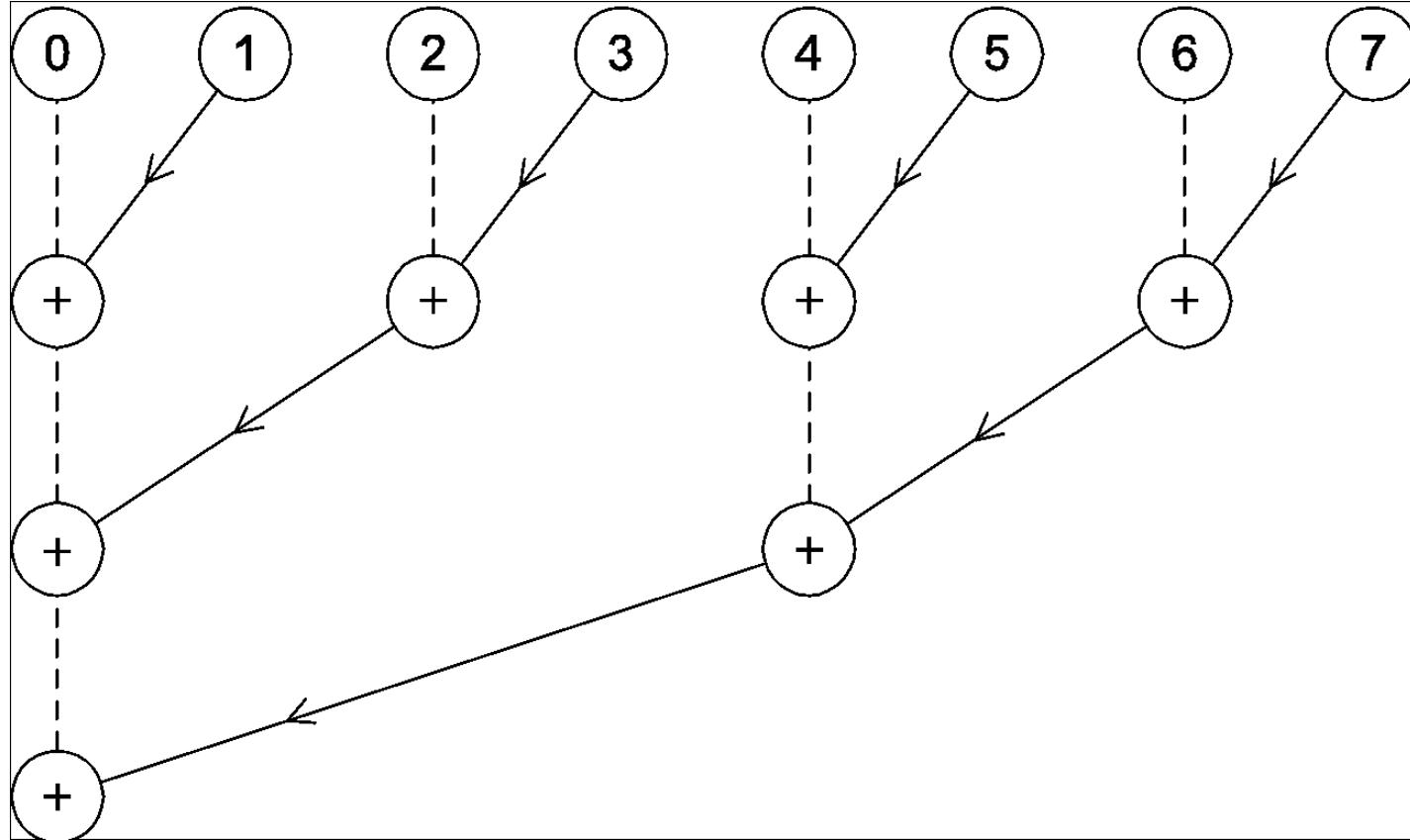
First two stages of Foster's Methodology



Alternative definition of tasks and communication



Adding the local arrays



Questions?

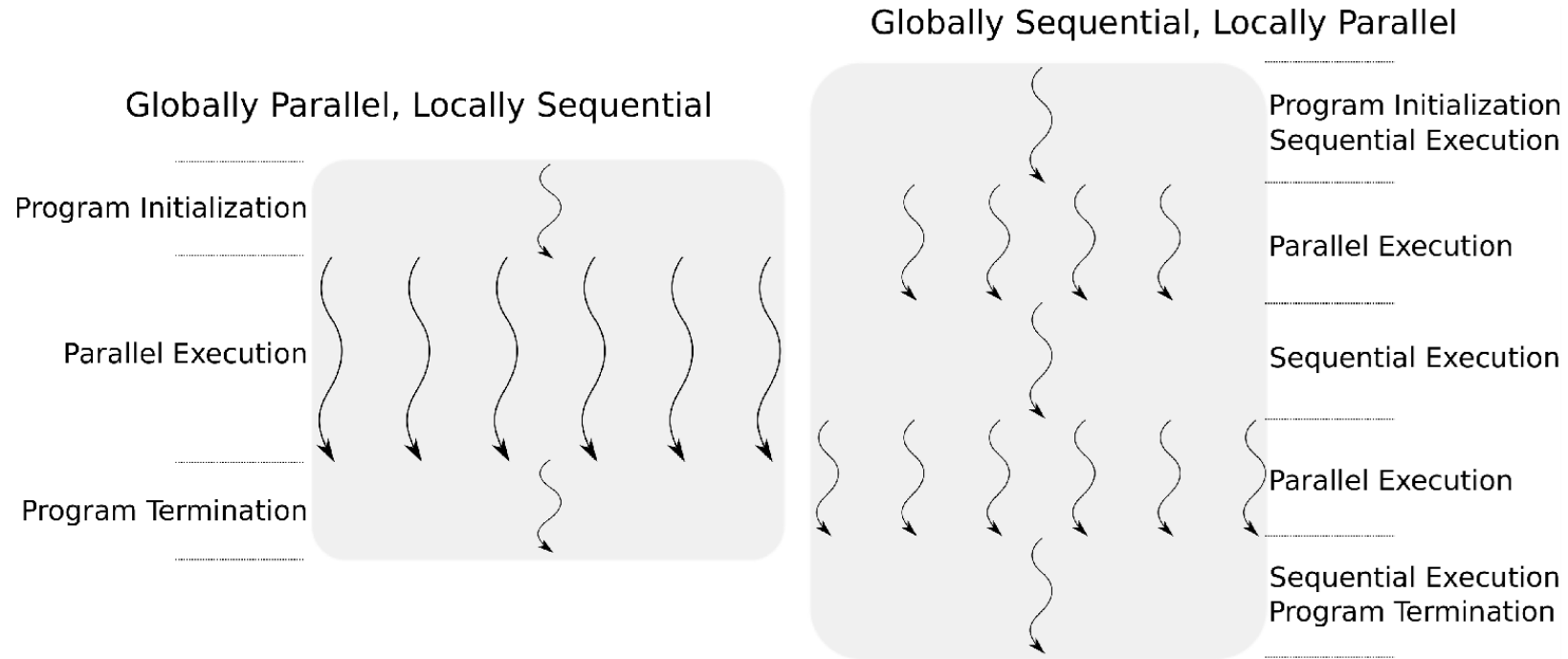
Parallel Design Patterns

Program Structure Patterns

We can distinguish the parallel program structure patterns into two major categories:

- **Globally Parallel, Locally Sequential (GPLS)** : this means that the application is able to perform multiple tasks concurrently, with each task running sequentially. Patterns that fall in this category include:
 - Single-Program, Multiple Data
 - Multiple-Program, Multiple Data
 - Master-Worker
 - Map-reduce
- **Globally Sequential, Locally Parallel (GSLP)** : this means that the application executes as a sequential program, with individual parts of it running in parallel when requested. Patterns that fall in this category include:
 - Fork/join
 - Loop parallelism

Program Structure Patterns



Program Structure Patterns

We can distinguish the parallel program structure patterns into two major categories:

- **Globally Parallel, Locally Sequential (GPLS)** : this means that the application is able to perform multiple tasks concurrently, with each task running sequentially. Patterns that fall in this category include:
 - Single-Program, Multiple Data
 - Multiple-Program, Multiple Data
 - Master-Worker
 - Map-reduce
- **Globally Sequential, Locally Parallel (GSLP)** : this means that the application executes as a sequential program, with individual parts of it running in parallel when requested. Patterns that fall in this category include:
 - Fork/join
 - Loop parallelism

Single Program Multiple Data

- Keeps all the application logic in a single program.
- Typical program structure involves:
 - **Program initialization** : e.g. runtime initialization
 - **Obtaining a unique identifier** : identifiers are numbered from 0, enumerating the threads or processes used. Some systems use vector identifiers (e.g. CUDA).
 - **Running the program** : execution path diversified based on ID.
 - **Shutting-down the program** : clean-up, saving results, etc.

Multiple Program Multiple Data

- SPMD fails when:
 - Memory requirements are too high for all nodes.
 - Heterogeneous platforms are involved.
- Execution steps are identical as SPMD.
- But deployment involves different programs.
- Both SPMD and MPMD are supported by MPI.

Master-Worker

Two kinds of components: Master and Workers

.Master (one or more) is responsible for:

- Handing out pieces of work to workers.
- Collecting the results of the computations from the workers.
- Performing I/O duties on behalf of the workers, i.e. sending them the data that they are supposed to process, or accessing a file.
- Interacting with the user.

.Good for implicit load balancing.

- No/few inter-worked data exchange

.Master can be a bottleneck.

- Use hierarchy of masters

Map-Reduce

- Variation of master-worker pattern.
- Old concept, made popular by Google's search engine implementation.
- Master coordinates the whole operation.
- Workers run two types of tasks:
 - **Map** : apply a function on data, resulting in a set of partial results
 - **Reduce** : collect the partial results and derive the complete one.
- Map and reduce workers can vary in number.
- Master-worker: Same function applied to different data items
- Map-reduce: Same function applied to different parts of a single data item

Program Structure Patterns

We can distinguish the parallel program structure patterns into two major categories:

- . **Globally Parallel, Locally Sequential (GPLS)** : this means that the application is able to perform multiple tasks concurrently, with each task running sequentially. Patterns that fall in this category include:
 - Single-Program, Multiple Data
 - Multiple-Program, Multiple Data
 - Master-Worker
 - Map-reduce
- . **Globally Sequential, Locally Parallel (GSLP)** : this means that the application executes as a sequential program, with individual parts of it running in parallel when requested. Patterns that fall in this category include:
 - Fork/join
 - Loop parallelism

Fork/Join

- Single, parent thread of execution
- Dynamic creation of children tasks at run-time.
- Tasks may run via spawning of threads, or via use of a static pool of threads.
- Children tasks have to finish for parent thread to continue.
- Used by OpenMP/Pthread

Fork/Join Example

```
mergesort(A, lo, hi):  
    if lo < hi:                                     // at least one element of input  
        mid = [lo + (hi - lo) / 2]  
        fork mergesort(A, lo, mid) // process (potentially) in  
                                   // parallel with main task  
        mergesort(A, mid, hi) // main task handles  
                                //second recursion  
    join  
    merge(A, lo, mid, hi)
```



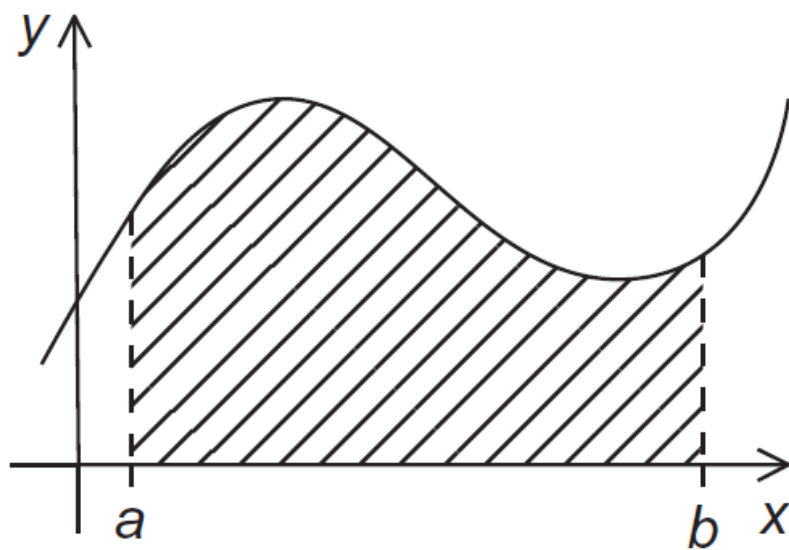
Loop parallelism

- Employed for migration of legacy/sequential software to multicore.
- Focuses on breaking up loops by manipulating the loop control variable.
- A loop has to be in a particular form to support this.
- Limited flexibility, but limited development effort as well.
- Supported by OpenMP.

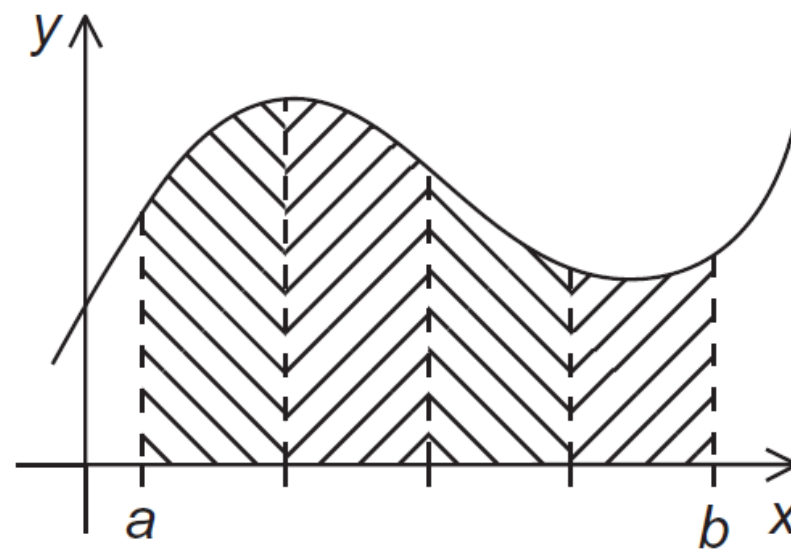
Questions?

Example: Trapezoidal rule in MPI

The Trapezoidal Rule

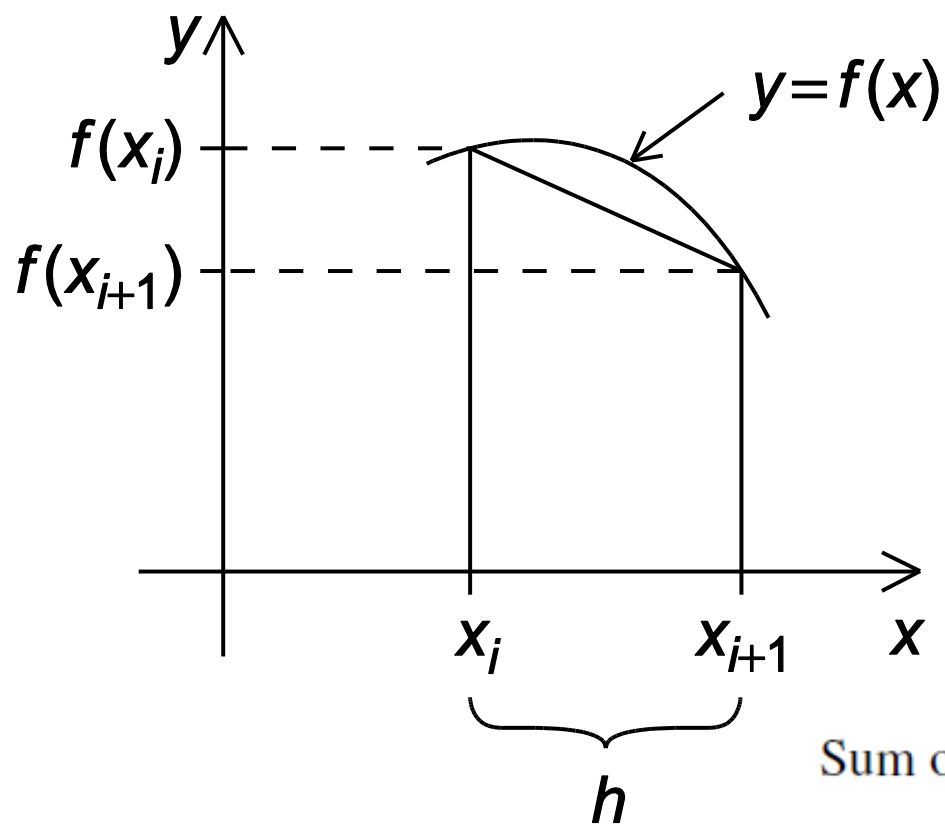


(a)



(b)

The Trapezoidal Rule



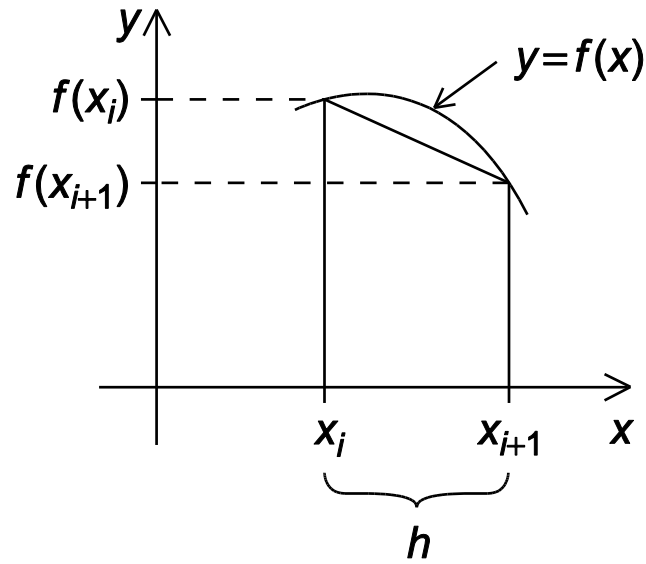
$$\text{Area of one trapezoid} = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

$$h = \frac{b-a}{n}$$

$$x_0 = a, x_1 = a + h, x_2 = a + 2h, \dots, x_{n-1} = a + (n-1)h, x_n = b$$

$$\text{Sum of trapezoid areas} = h[f(x_0)/2 + f(x_1) + f(x_2) + \dots + f(x_{n-1}) + f(x_n)/2]$$

Pseudo-code for a serial program



$$\text{Area of one trapezoid} = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

$$h = \frac{b-a}{n}$$

$$x_0 = a, x_1 = a+h, x_2 = a+2h, \dots, x_{n-1} = a+(n-1)h, x_n = b$$

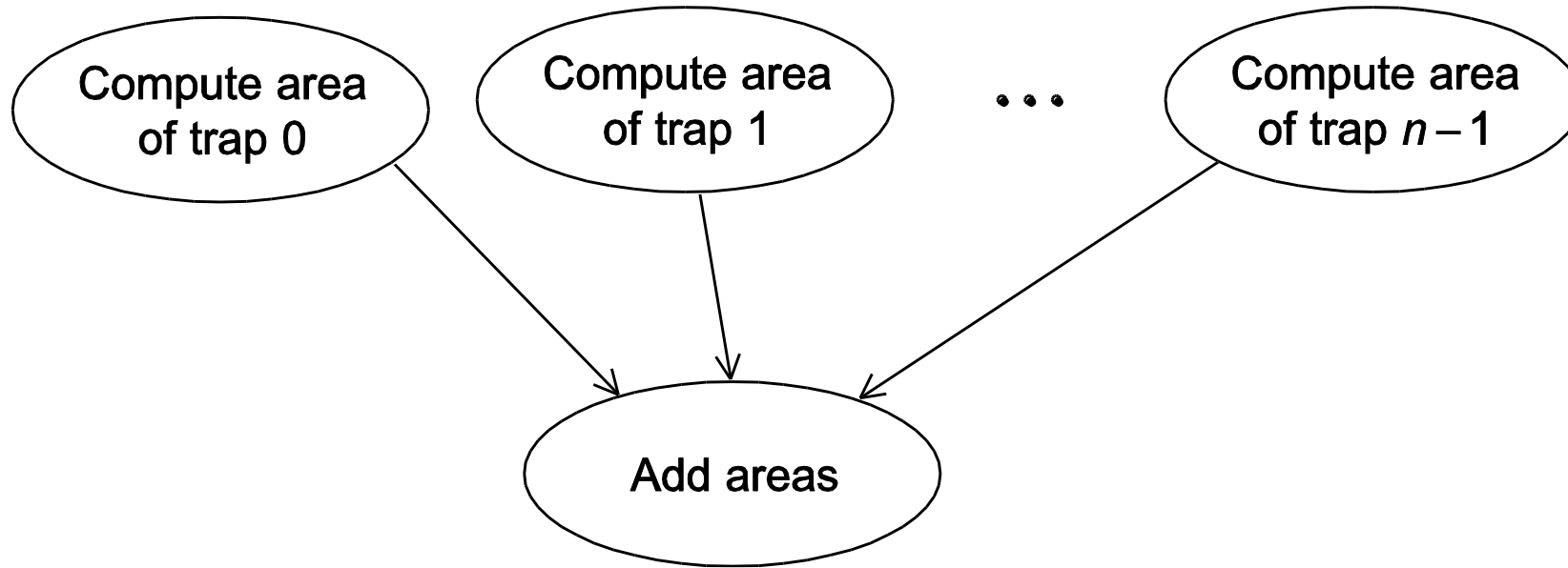
$$\text{Sum of trapezoid areas} = h[f(x_0)/2 + f(x_1) + f(x_2) + \dots + f(x_{n-1}) + f(x_n)/2]$$

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

Parallelizing the Trapezoidal Rule

1. Partition problem solution into tasks.
2. Identify communication channels between tasks.
3. Aggregate tasks into composite tasks.
4. Map composite tasks to cores.

Tasks and communications for Trapezoidal Rule



To which pattern does it look like?

Parallel pseudo-code

```
1  Get a, b, n;
2  h = (b-a)/n;
3  local_n = n/comm_sz;
4  local_a = a + my_rank*local_n*h;
5  local_b = local_a + local_n*h;
6  local_integral = Trap(local_a, local_b, local_n, h);
7  if (my_rank != 0)
8      Send local_integral to process 0;
9  else /* my_rank == 0 */
10     total_integral = local_integral;
11     for (proc = 1; proc < comm_sz; proc++) {
12         Receive local_integral from proc;
13         total_integral += local_integral;
14     }
15 }
16 if (my_rank == 0)
17     print result;
```

First version (part 1)

```
1  int main(void) {
2      int my_rank, comm_sz, n = 1024, local_n;
3      double a = 0.0, b = 3.0, h, local_a, local_b;
4      double local_int, total_int;
5      int source;
6
7      MPI_Init(NULL, NULL);
8      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
9      MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
10
11     h = (b-a)/n;          /* h is the same for all processes */
12     local_n = n/comm_sz; /* So is the number of trapezoids */
13
14     local_a = a + my_rank*local_n*h;
15     local_b = local_a + local_n*h;
16     local_int = Trap(local_a, local_b, local_n, h);
17
18     if (my_rank != 0) {
19         MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
20                 MPI_COMM_WORLD);
```


First version (part 2)

```
21     } else {
22         total_int = local_int;
23         for (source = 1; source < comm_sz; source++) {
24             MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
25                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26             total_int += local_int;
27         }
28     }
29
30     if (my_rank == 0) {
31         printf("With n = %d trapezoids, our estimate\n", n);
32         printf("of the integral from %f to %f = %.15e\n",
33               a, b, total_int);
34     }
35     MPI_Finalize();
36     return 0;
37 } /*  main  */
```

First version (part 3)

```
1  double Trap(  
2      double left_endpt  /* in */,  
3      double right_endpt /* in */,  
4      int    trap_count  /* in */,  
5      double base_len    /* in */) {  
6      double estimate, x;  
7      int i;  
8  
9      estimate = (f(left_endpt) + f(right_endpt))/2.0;  
10     for (i = 1; i <= trap_count-1; i++) {  
11         x = left_endpt + i*base_len;  
12         estimate += f(x);  
13     }  
14     estimate = estimate*base_len;  
15  
16     return estimate;  
17 } /*  Trap  */
```

Input

- Most MPI implementations only allow process 0 in `MPI_COMM_WORLD` access to `stdin`.
- Process 0 must read the data (`scanf`) and send to the other processes.

```
. . .  
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
  
Get_data(my_rank, comm_sz, &a, &b, &n);  
  
h = (b-a)/n;  
. . .
```

Function for reading user input

```
void Get_input(  
    int      my_rank    /* in */,  
    int      comm_sz    /* in */,  
    double*  a_p        /* out */,  
    double*  b_p        /* out */,  
    int*     n_p        /* out */) {  
    int dest;  
  
    if (my_rank == 0) {  
        printf("Enter a, b, and n\n");  
        scanf("%lf %lf %d", a_p, b_p, n_p);  
        for (dest = 1; dest < comm_sz; dest++) {  
            MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);  
            MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);  
            MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);  
        }  
    } else { /* my_rank != 0 */  
        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,  
                 MPI_STATUS_IGNORE);  
        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,  
                 MPI_STATUS_IGNORE);  
        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,  
                 MPI_STATUS_IGNORE);  
    }  
} /* Get_input */
```

Hands-on break

Setup

Su linux (ubuntu)

```
sudo apt install openmpi
```

Oppure

```
sudo apt install mpich
```

Esecuzione MPI

- Esecuzione su host locale di tre processi

```
$ mpirun -np 3 ./mpi_hello_world
```

- Esecuzione su 3 hosts di 3 processi

```
$ mpirun -np 3 --host node1,node2,node3 ./mpi_hello_world
```

- Esecuzione su 3 hosts di 10 processi

```
$ mpirun -np 10 --host node1:4,node2:3,node3:3 ./mpi_hello_world
```

Esercizi

PROGRAMMING ASSIGNMENTS

3.2. Suppose we toss darts randomly at a square dartboard, whose bullseye is at the origin, and whose sides are 2 feet in length. Suppose also that there's a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and its area is π square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation

$$\frac{\text{number in circle}}{\text{total number of tosses}} = \frac{\pi}{4},$$

i.e., number in circle : π = total number of tosses : 4

Remember that if origin (0,0), the circle must respect the condition $x^2+y^2=r^2$

We can use this formula to estimate the value of π with a random number generator:

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
    x = random double between -1 and 1;
    y = random double between -1 and 1;
    distance_squared = x*x + y*y;
    if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```

This is called a “Monte Carlo” method, since it uses randomness (the dart tosses). Write an MPI program that uses a Monte Carlo method to estimate π .

Exercises

3.2. Modify the trapezoidal rule so that it will correctly estimate the integral even if *comm_sz* doesn't evenly divide *n*. (You can still assume that $n \geq comm_sz$)

3.4. Modify the program that just prints a line of output from each process (*mpi_output.c*) so that the output is printed in process rank order: process 0s output first, then process 1s, and so on.

PROGRAMMING ASSIGNMENTS

3.1. Use MPI to implement the histogram program. Have process 0 read in the input data and distribute it among the processes. Also have process 0 print out the histogram.

PROGRAMMING ASSIGNMENTS

3.3. Write an MPI program that computes a tree-structured global sum. First write your program for the special case in which *comm_sz* is a power of two. Then, after you've gotten this version working, modify your program so that it can handle any *comm_sz*