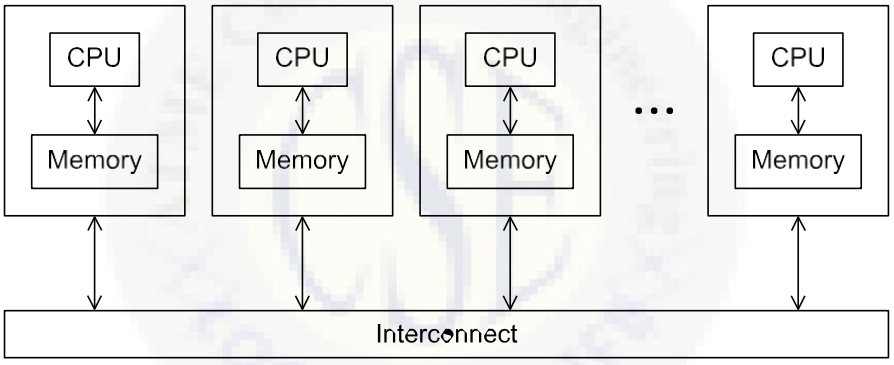
MODULE 3

Distributed memory programming with MPI

Distributed-Memory System (shown in Figure 3.1)

- Each CPU (core) has its own private memory.
- A CPU can directly access only its local memory.
- If one CPU needs data from another CPU's memory, it cannot access it directly and it must use message passing through the interconnect (network).



Shared-Memory System (Figure 3.2)

- All CPUs are connected to the **same global memory**.
- Any CPU can read/write from **any memory location** directly.
- But this requires hardware mechanisms (like cache coherence) to ensure consistency.

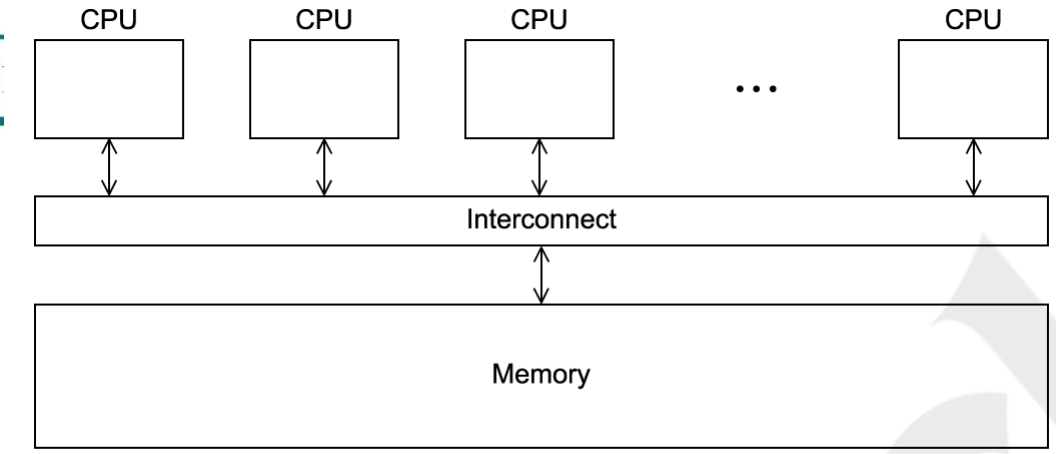


FIGURE 3.2

A shared memory system.

Programming Implication

- In **distributed memory**, communication through **message passing (MPI)**.
    - One process sends (send)
    - Another receives (recv).
- In **shared memory**, communication through **shared variables** (handled automatically by hardware/software).

MPI (Message Passing Interface)

- Not a programming language but it's a **library** for C/Fortran.
- Provides functions for:
    - **Point-to-point communication** to send/receive.
    - **Collective communication** to group operations (e.g., broadcast, scatter, gather).

Getting started with MPI:

Each process in MPI is given a rank (an integer ID).

We have p processes (say 4).

- Ranks are 0, 1, 2, …, p-1.

In this program:

- Process 0 is the "master" → it collects and prints the messages.
- Other processes (1, 2, …, p-1) send "hello" messages to process 0.

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>   // MPI header

int main(int argc, char* argv[]) {
   int my_rank;     // process rank (ID)
   int p;           // number of processes
   int source;      // rank of sender
   int dest = 0;    // destination rank (process 0)
   int tag = 0;     // message tag
   char message[100];
   MPI_Status status;

   // Initialize MPI
   MPI_Init(&argc, &argv);

   // Get my rank
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   // Get number of processes
   MPI_Comm_size(MPI_COMM_WORLD, &p);

   if (my_rank != 0) {
      // Non-zero processes send a message to process 0
      sprintf(message, "Hello from process %d of %d!", my_rank, p);
      MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
   } else {
      // Process 0 receives messages and prints them
      printf("Hello from process 0 of %d!\n", p);
```

```
    for (source = 1; source < p; source++) {
        MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
        printf("%s\n", message);
    }
  }
  // Finalize MPI
  MPI_Finalize();

  return 0;
}
```

Compilation and Execution: MPI programs are compiled using **mpicc** instead of plain gcc.

- mpicc = **wrapper script** around your system's C compiler.
  - It automatically knows where the **MPI header files** (mpi.h) and **MPI libraries** are located.
  - You don't have to manually specify them.

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

- -g → add debugging info.
- -Wall → show all warnings.
- -o mpi_hello → name of the output program.
- mpi_hello.c → your source file.

This produces an executable file named **mpi_hello**.

To run MPI programs, you use **mpiexec** (sometimes mpirun depending on your system). This command tells MPI to start **multiple processes** of your program.

```
mpiexec -n <number of processes> ./mpi_hello
```

Examples:

1. Run with **1 process**:

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

Output: Greetings from process 0 of 1!

2. Run with **4 processes**:

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

Output:

```
Greetings from process 0 of 4!
Greetings from process 1 of 4!
```

Greetings from process 2 of 4!
Greetings from process 3 of 4!
The order of greetings from processes 1–3 may vary because processes run **in parallel**.

1. mpiexec -n 4 ./mpi_hello → starts **4 copies** of the same program (mpi_hello).
2. Each copy is given a **unique rank** (0, 1, 2, 3).
3. They all run at the same time, possibly on different CPU cores.
4. MPI ensures they can **communicate** (via send/recv, collective ops, etc).

Compile with: mpicc
Run with: mpiexec -n <p>
MPI automatically assigns ranks and connects processes so they can exchange messages.

MPI Programs:

It's just a normal C program… plus MPI

- Starts with standard C headers:

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

Has a main() function. You can use all normal C syntax, loops, conditionals, functions, etc.

The special part:
#include <mpi.h>

- This is the MPI header file.
- It contains:
  - Function prototypes (MPI_Init, MPI_Send, etc).
  - Type definitions (MPI_Comm, MPI_Status, etc).
  - Macros/constants (MPI_COMM_WORLD, MPI_CHAR, etc).

Without this include, your compiler wouldn't know what MPI functions or types mean.

MPI Naming Convention

MPI makes its functions and identifiers very easy to recognize:

- Functions:
  Start with MPI_ and the next word starts with a capital.
  - Example: MPI_Init, MPI_Send, MPI_Recv
- Types: Same rule (capitalized after underscore).
  - Example: MPI_Comm, MPI_Datatype, MPI_Status
- Constants/Macros: All uppercase.
  - Example: MPI_COMM_WORLD, MPI_CHAR, MPI_SUCCESS

This way, you can always tell:

- MPI-provided starts with MPI_

MPI_Init and MPI_Finalize:

int MPI_Init(int* argc_p, char argv_p);***

Purpose: Starts the MPI environment.
What it does internally:

- Allocates resources (like message buffers).
- Decides ranks (which process is 0, 1, 2, …).
- Sets up communication among processes.

Arguments:

- argc_p and argv_p are pointers to argc and argv from main.
- This allows MPI to remove any "MPI-specific" command-line arguments before your program uses them.
- If your program doesn't need them → you can safely pass NULL, NULL.

Rule:
No MPI calls before MPI_Init. If you do, it's undefined behavior.
Return value:

- Returns an int error code (MPI_SUCCESS if all good).
- Usually ignored in beginner programs to keep code clean.

int MPI_Finalize(void);
Purpose: Shuts down the MPI environment.
What it does internally:

- Frees resources used by MPI.
- Ends communication.
- Cleans up runtime environment.

Rule:
No MPI calls after MPI_Finalize. The MPI system is no longer active.

Typical MPI Program Skeleton

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

int main(int argc, char* argv[]) {
   // No MPI calls before this
   MPI_Init(&argc, &argv);

   // ---- MPI Work happens here ----
   // e.g., MPI_Comm_rank, MPI_Comm_size, MPI_Send, MPI_Recv, etc.

   MPI_Finalize();
   // No MPI calls after this

   return 0;
```

Communicators, MPI_Comm_size, and MPI_Comm_rank:

- A communicator in MPI is a *collection of processes* that can talk to each other.
- Every communicator has:
  - A set of processes
  - A local numbering (ranks) for those processes

At program startup, MPI automatically creates a default communicator called: MPI_COMM_WORLD

It includes all processes started by your mpiexec -n <p> command. Most beginner MPI programs only use MPI_COMM_WORLD.

Two fundamental functions:

(a) MPI_Comm_size

int MPI_Comm_size(MPI_Comm comm, int* comm_sz_p);

- Input: comm → a communicator (e.g., MPI_COMM_WORLD).
- Output: comm_sz_p → number of processes in this communicator. Stores total process count in your variable (usually comm_sz).

(b) MPI_Comm_rank

int MPI_Comm_rank(MPI_Comm comm, int* my_rank_p);

- Input: comm → a communicator (e.g., MPI_COMM_WORLD).
- Output: my_rank_p → the *rank* (ID) of this process inside the communicator. Stores your process's ID (0 … comm_sz−1) in your variable (usually my_rank).

Example in Context (from Program 3.1)

```
int comm_sz;   // number of processes
int my_rank;   // my process ID

MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

SPMD programs:

1. One Program, Many Processes

- In MPI, you **compile just one program** (mpicc hello.c -o hello).
- When you run it with mpiexec -n P ./hello, MPI **starts P processes**, each running the *same executable*.
- So, at first, all processes are *identical*.

2. How Do Processes Do Different Work?

- Even though all processes run the same code, each process can **check its rank** (my_rank) and *branch* accordingly.

MPI will **start 4 copies** of the same program hello.

- These 4 copies are called **processes**.
- Each process has its **own memory**, **own registers**, and **runs independently**.
- At the very beginning, they all **start from main()** in exactly the same way.

So — at time **t = 0**:

- Process 0, Process 1, Process 2, Process 3 … are **running the same code**.
- They are "identical" in that they don't yet know they are different.

How do they become different?

Each process asks MPI:its rank

Process with rank **0** → gets my_rank = 0.

Process with rank **1** → gets my_rank = 1.

Process with rank **2** → gets my_rank = 2.

Process with rank **3** → gets my_rank = 3.

Example:

```
if (my_rank == 0) {
   // Process 0 acts like the master
   // Receives and prints messages
} else {
   // Other processes (rank 1, 2, …) act like workers
   // Send messages to process 0
}
```

This is exactly what happened in **Program 3.1**:

- **Rank 0** = the "coordinator" → receiving + printing messages.
- **Ranks 1, 2, …** = the "workers" → creating + sending messages.

This Approach Has a Name: SPMD

- **SPMD (Single Program, Multiple Data)**
    - One program binary is executed by all processes.
    - Each process may follow different execution paths depending on its **rank**.
    - Each process usually works with a different portion of the data.

Contrast:

- **MPMD (Multiple Program, Multiple Data)** is You actually write and run *different programs* for different process roles (less common in MPI).
- **SPMD** is preferred because it's **simpler and scalable**.

Why Scalability Matters

- The code doesn't "hard-wire" the number of processes.
- Instead, it queries:
    - MPI_Comm_size → total processes
    - MPI_Comm_rank → current process rank

This way, the same code runs correctly whether you launch:

- mpiexec -n 1 ./prog → runs with 1 process
- mpiexec -n 4 ./prog → runs with 4 processes
- mpiexec -n 1000 ./prog → runs with 1000 processes

Communication

In Lines 17–18, each process, other than process 0, creates a message it will send to process 0. (The function sprintf is very similar to printf, except that instead of writing to stdout, it writes to a string.) Lines 19–20 actually send the message to process 0. Process 0, on the other hand, simply prints its message using printf, and then uses a **for** loop to receive and print the messages sent by processes 1, 2, . . . , comm_sz − 1. Lines 25–26 receive the message sent by process *q*, for *q* = 1, 2, . . . , comm_sz − 1.

Sending a message with MPI_Send

The function is:

```
int MPI_Send(
  void* msg_buf_p,    // pointer to data to send
  int msg_size,       // number of data items
  MPI_Datatype type,  // type of each item (MPI_INT, MPI_CHAR, etc.)
  int dest,           // rank of the destination process
  int tag,            // a message "label"
  MPI_Comm comm       // communicator (usually MPI_COMM_WORLD)
);
```

msg_buf_p → pointer to the string (the greeting).
msg_size → number of characters in the string (strlen(greeting)+1 to include the \0).
msg_type → MPI_CHAR (since you're sending a string).
dest → 0 (all processes send their greeting to process 0).
tag → 0 (you can use tags to distinguish messages if needed).
comm → MPI_COMM_WORLD (all processes are in this communicator).

So if process 3 is running, it might execute:
MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);

That means:*"Send my greeting string to process 0 using communicator MPI_COMM_WORLD with tag 0."*

Receiving a message with MPI_Recv

The function is:

```
int MPI_Recv(
  void* msg_buf_p,    // buffer to store incoming message
  int buf_size,       // max number of items buffer can hold
  MPI_Datatype type,  // type of each item
  int source,         // rank of sender process
  int tag,            // must match sender's tag
  MPI_Comm comm,      // communicator
  MPI_Status* status_p // info about received message (can ignore)
);
```

msg_buf_p → buffer to store the incoming greeting (greeting).
buf_size → size of the buffer (e.g. 100 characters).
buf_type → MPI_CHAR.
source → the process rank you expect (e.g. 1, 2, 3, …).
tag → must match the sender's tag (in our case, 0).
comm → MPI_COMM_WORLD.
status_p → usually MPI_STATUS_IGNORE if you don't need details.

So process 0 might execute:
MPI_Recv(greeting, 100, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

That means: *"Receive a message of up to 100 chars from process q, with tag 0, in communicator MPI_COMM_WORLD."*

So the flow is:
Non-zero ranks (1,2,3,...) build a greeting string → send it to rank 0.
Rank 0 loops over all other ranks, receiving each greeting and printing them.
Rank 1 → "Hello from 1" → send → Rank 0 receives
Rank 2 → "Hello from 2" → send → Rank 0 receives
Rank 3 → "Hello from 3" → send → Rank 0 receives
...
MPI_Send = "Here's some data, send it to another process."
MPI_Recv = "Wait for data to arrive from a specific process."
Both must agree on **size, type, communicator, and tag** or the program will hang / fail.

In MPI, you use **MPI datatypes** (like MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_DOUBLE, etc.).
So MPI_CHAR just tells MPI: "This message is made of characters (bytes). Treat each element as a character." That's why when sending strings, we use MPI_CHAR.
If you were sending an integer array, you'd use MPI_INT.

The **tag** is like a label for the message.

- Imagine process 0 is expecting different types of messages (maybe greetings, results, control signals).
- Each message type could be given a different tag number (e.g., greetings=0, results=1, stop=2).
- The receiver can then say: *"I only want to receive messages with tag = 0."*

In the greetings program, all messages are of the same type (simple strings). So they just used tag = 0 everywhere for simplicity.

Table 3.1 Some predefined MPI datatypes.

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	

Message matching:

Message matching rules

Suppose process *q* calls MPI_Send with
MPI_Send (send_buf_p , send_buf_sz , send_type , dest , send_tag , send_comm) ;

Also suppose that process *r* calls MPI_Recv with
MPI_Recv (recv_buf_p , recv_buf_sz , recv_type , src , recv_tag , recv_comm , &status) ;

Then the message sent by *q* with the above call to MPI_Send can be received by *r* with the call to MPI_Recv if

- recv_comm = send_comm,
- recv_tag = send_tag,

• dest = r, and
• src = q.

For a send to match a recv, **four things must line up**:

1. **Communicator**:
   recv_comm = send_comm Both processes must be in the same "group of processes".
2. **Tag**:
   recv_tag = send_tag. The message label must match — unless receiver says "I don't care" with MPI_ANY_TAG.
3. **Destination = Receiver rank**:
   dest = r. The message must be sent to the correct process rank.
4. **Source = Sender rank**:
   src = q The receiver must be expecting the right sender — unless it says "I don't care" with MPI_ANY_SOURCE.
5. **Buffers must be compatible**:
   - Same datatype (MPI_INT vs MPI_CHAR must match).
   - Receiver buffer must be **big enough** to hold what's being sent.

The problem
If process 0 waits for messages **in a fixed order**:

```
MPI_Recv(result, sz, MPI_INT, 1, tag, comm, ...);
MPI_Recv(result, sz, MPI_INT, 2, tag, comm, ...);
...
```

but process 3 finishes first, its message will **sit waiting** in the system buffer until process 0 finally gets around to MPI_Recv(..., src=3, ...). That's inefficient.

The solution for that is using wildcards
MPI provides **wildcards** so receivers don't have to predict the exact order:

- **MPI_ANY_SOURCE** → "I don't care who sent it."
- **MPI_ANY_TAG** → "I don't care what the tag is."

Example:

```
for (i = 1; i < comm_sz; i++) {
   MPI_Recv(result, sz, MPI_INT,
         MPI_ANY_SOURCE,   // don't care which process sent
         MPI_ANY_TAG,      // don't care what tag it has
         comm, MPI_STATUS_IGNORE);
   Process_result(result);
}
```

Now process 0 receives results **in the order they arrive**, no matter who sends first.

Wildcards only for receivers

- Senders must say exactly *who* they are sending to and *what tag*.
- Receivers can say "I'll take anything from anyone."

No wildcard for communicators

- Both must belong to the same communicator.
- A communicator is like a **chat room**: if you're not in the same one, you can't talk.

Normal matching = all 4 (comm, tag, src, dest) must match.

Wildcards let the receiver relax some rules → very useful when the order of arrivals is unpredictable.

The status_p argument:

If you think about these rules for a minute, you'll notice that a receiver can receive a message without knowing

1. the amount of data in the message,
2. the sender of the message, or
3. the tag of the message. MPI gives you this info afterwards through the **status object**.

MPI_Status is a little struct that stores:

- MPI_SOURCE → the actual rank that sent the message
- MPI_TAG → the tag used by the sender
- MPI_ERROR → any error code
-

But **not** the number of elements. Because **"count" depends on datatype**.
Imagine Sender sends 20 bytes and Receiver could be asking for MPI_CHAR (→ 20 elements) or MPI_INT (4 bytes each → 5 elements). MPI can't store both "20 chars" and "5 ints" in status. So instead of guessing, MPI just stores the raw size internally. If you *want* the count, you call:
MPI_Get_count(&status, recv_type, &count);

Then MPI computes: count = (bytes_received) / (bytes_per_element_of_recv_type)

Why make you call separately?
Because of Efficiency. If you don't care about the number of elements, MPI doesn't waste time doing the division.

MPI_Status = "who sent, what tag, any error."
MPI_Get_count = "how many elements of this datatype actually arrived."

Semantics of MPI_Send and MPI_Recv

When you call MPI_Send, the system prepares your data for transmission:

1. **Envelope creation**: MPI attaches metadata (called the "envelope") to your message:
    - Destination process rank
    - Sender process rank
    - Message tag
    - Communicator
    - Message size

Think of this like an email: you don't just write the message, you also add *To, From, Subject,* etc.

2. **Two possible behaviors** (depends on MPI implementation + message size):
    - **Buffered send**: MPI copies your data (plus envelope) into its own internal buffer. Then, MPI_Send **returns immediately**. Your program can reuse/modify its send buffer, since MPI has made a safe copy.
    - **Blocking send**: If the message is large (above some *cutoff size* set by MPI), MPI waits until the receiver is ready or transmission starts before returning. Your program cannot move on until MPI ensures the data is safe.

When MPI_Send returns, you don't know whether the message *actually arrived* at the receiver. You only know it's safe to reuse the send buffer.

- MPI_Recv **always blocks** until:
    1. A matching message (correct source, tag, communicator) is found.
    2. The message is completely copied into the receive buffer.

This means when MPI_Recv returns, the message is guaranteed to be safely in your buffer.

Message ordering in MPI

MPI enforces **non-overtaking rule**:

- If process **q** sends two messages to process **r**, the **first one must arrive before the second**.
- BUT: If two different processes (**q** and **t**) send messages to the same process (**r**), MPI does **not** guarantee which arrives first. Because network delays are unpredictable.

Example:

- If **q** sends "Hello" then "Bye" to **r**, **r** will always receive "Hello" first.
- If **q** sends "Hello" and **t** sends "Bye" to **r**, the order is **not guaranteed**.

Why does MPI do this?

1. Efficiency: Small messages → buffered for speed. Large ones → blocking to avoid memory overhead.

2. Correctness: Receiver is guaranteed to have valid data before proceeding.

3. Flexibility: MPI does not impose unnecessary synchronization unless you explicitly request it (with special functions like MPI_Ssend, MPI_Bsend, MPI_Isend).

4. MPI_Send: May buffer or block, but guarantees send buffer is reusable when it returns.

5. MPI_Recv: Always blocks until data is received into the buffer.

Ordering: Same-sender messages are ordered, cross-sender messages are not.

Some potential pitfalls

Pitfalls with MPI_Send and MPI_Recv

1. **Hanging Receives (Deadlock)**
    - MPI_Recv always blocks until a matching MPI_Send arrives.
    - If no matching send ever comes (wrong source, wrong tag, or forgot to send), the process waits forever then program hangs.

Example:
MPI_Recv(buf, 10, MPI_INT, 1, 0, MPI_COMM_WORLD, &status); If process 1 never sends a message with tag 0, this process is stuck.

2. **Mismatched Tags or Sources**

- Communication only happens if rank + tag + communicator match.
- If you mix up the rank (e.g., send to 0 but receive from 1), or use different tags (0 vs 1), the messages won't match.
- Then one process waits forever, or worse → it might match a completely different message by accident.

3. **Hanging Sends (Deadlock)**

- Depending on implementation, MPI_Send may block if the message is large (not buffered).
- If no matching MPI_Recv is posted, the sender can also hang forever.

4. **Buffered Sends is Lost Messages**

- For small messages, MPI often uses buffering.
- That means MPI_Send might return immediately, even if no MPI_Recv is posted.
- But if no receive is ever called, the message just disappears (lost).

Trapezoidal rule: The trapezoidal rule is a numerical method to approximate the value of a definite integral: $I = \int_a^b f(x)\,dx$

Sometimes, we don't know how to compute this integral exactly (either because f(x) has no simple antiderivative, or it's too expensive). So instead of solving it analytically, we approximate it with geometry.

The area of one trapezoid (from **Figure 3.4**) is:

$$\frac{h}{2}[f(x_i) + f(x_i + 1)]$$

Thus if we call the leftmost endpoint *x*0, and the rightmost endpoint *xn*, we have that: *x*0 = *a*, *x*1 = *a* +*h*, *x*2 = *a* + 2*h*, . . . , *xn* − 1 = *a* + (*n* − 1)*h*, *xn* = *b*,

If we split [a,b] into n equal subintervals, each of width

$$\boldsymbol{h = (b - a) \div n},$$

Then the trapezoidal rule is: Add all trapezoid areas:

$$I \approx h[\frac{f(x_0)}{2} + f(x1) + f(x2) + \cdots + f(xn-1) + \frac{f(x_n)}{2}]$$

It is a numerical integration method used to approximate the area under a curve y=f(x)between two limits a and b.

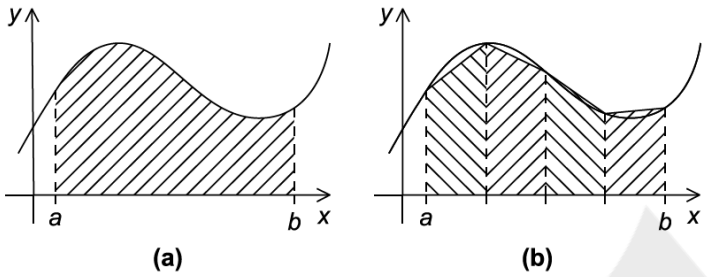


FIGURE 3.3

The trapezoidal rule: (a) area to be estimated and (b) approximate area using trapezoids.

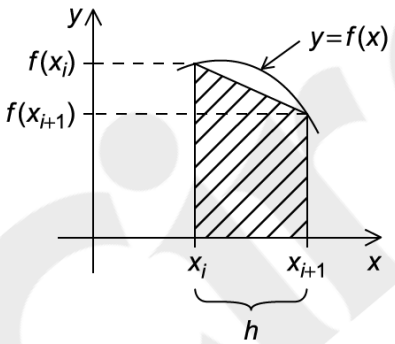


FIGURE 3.4

One trapezoid.

Figure 3.3(a): The actual area under the curve from a to b.
Figure 3.3(b): Approximation of the area using trapezoids.
Figure 3.4: A single trapezoid formed between x_i and x_{i+1}.

serial pseudocode for the **Trapezoidal Rule**

```
/ * 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 ;
```

- **Inputs**:

- a = lower limit
- b = upper limit
- n = number of trapezoids (subintervals)

- Compute step size:
  $h=(b-a)/n$

Initialize approximation with **half-weighted endpoints**. In the trapezoidal rule formula, f(a) and f(b) appear only **once**, while interior points are counted **twice**. That's why we start with half of the sum of endpoints. Loop over all **interior points** x1,x2,...,xn−1. For each, add f(xi) to the sum. Finally, multiply the accumulated sum by h to scale it to the correct area.

Parallelization of the trapezoidal rule with MPI

1. Partition into tasks

- Each trapezoid area can be computed **independently**.
- So instead of one process computing all trapezoids, we **split** them among MPI processes.

Example:

If you have n=8 trapezoids and comm_sz=4 processes,

- Process 0 handles trapezoids 0–1
- Process 1 handles trapezoids 2–3
- Process 2 handles trapezoids 4–5
- Process 3 handles trapezoids 6–7

---

2. Communication channels

- Each process computes its **local integral** (sum of its trapezoids).
- Then they must **communicate results back** to one process (usually rank 0).

So "Compute local area" , Send and then "Add areas together at rank 0". Local variables are variables whose contents are significant only on the process that's using them. Some examples from the trapezoidal rule program are local_a, local_b, and local_n.

Since we'll have far more trapezoids than cores, we **aggregate trapezoids into groups**, and each process computes a chunk.
That's why we calculate:

- local_n = n / comm_sz → trapezoids per process
- local_a, local_b → subinterval handled by each process

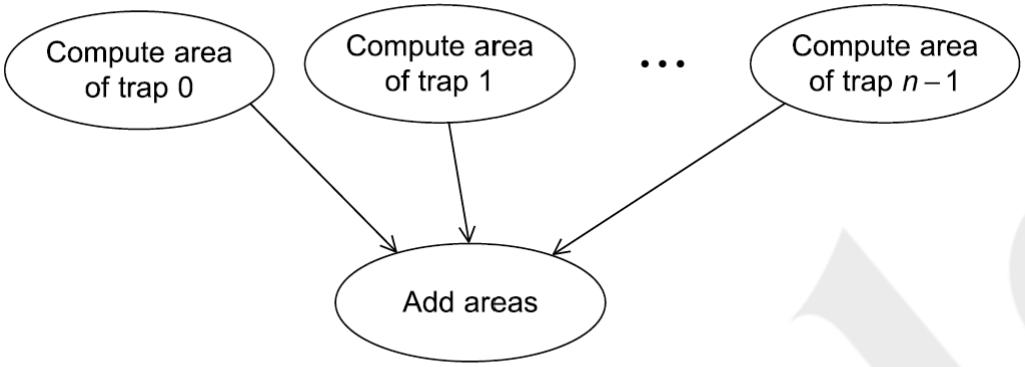


FIGURE 3.5
Tasks and communications for the trapezoidal rule.

4. Mapping to cores

- Each MPI rank gets **one subinterval**.
- Rank 0 later collects all partial results.

```
1 Get a , b , n ;
2 h = (b−a ) / n ;              // width of each trapezoid
3 local_n = n / comm_sz ;       // trapezoids per process
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) ;
```

Each process computes its **own part** using the serial trapezoidal rule (Trap function).

```
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 ;
```

Nonzero processes → **send results to rank 0**.
Rank 0 → **receives results, sums them, prints the final integral**.

Each MPI process computes a piece of the integral.
Everyone except rank 0 sends their result.
Rank 0 collects results and prints the answer.

First version of MPI trapezoidal rule.

```
int main(void) {
   int my_rank, comm_sz, n = 1024, local_n;
   double a = 0.0, b = 3.0, h, local_a, local_b;
   double local_int, total_int;
   int source;
  MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
h = (b - a) / n;         /* width of each trapezoid */
local_n = n / comm_sz;    /* trapezoids per process */
local_a = a + my_rank * local_n * h;
local_b = local_a + local_n * h;
local_int = Trap(local_a, local_b, local_n, h);
if (my_rank != 0) {
   MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
} else {
   total_int = local_int;
   for (source = 1; source < comm_sz; source++) {
      MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      total_int += local_int;
```

```
 }
}
if (my_rank == 0) {
   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.15e\n", a, b, total_int);
}
MPI_Finalize();
return 0;
}
```

n = 1024 → total number of trapezoids.
Interval: [a,b]=[0,3]

Each process will compute its **own piece** of the integral.
Start MPI.
my_rank → unique ID of the process (0,1,2,…).
comm_sz → total number of processes.
All processes have the **same trapezoid width**.
local_n = how many trapezoids each process handles.
Each process calculates its **own sub-interval** [locala,localb]. Calls Trap(...) (serial trapezoidal function) to compute **local integral**.

Non-root processes (rank != 0) → send their results to rank 0.
Rank 0:

- Initializes total_int with its own contribution.
- Loops over all other processes, **receives** their integrals, and sums them.

Only rank 0 prints the final answer. Clean shutdown of MPI.

Dealing with I/O

The program is **hardcoded** to integrate over [0,3] with **1024 trapezoids**.

- That's inflexible because every time you want a new range or number of trapezoids, you must edit and recompile.
- Better to allow the **user to provide input** (e.g., via command line or MPI input gathering).

MPI lets **all processes** call printf.

- BUT there's **no guarantee** about the **order of output**.
- Multiple processes writing to the same shared stdout leads to **nondeterminism**.
- Example: With 5 processes, you might get a neat sequence, but with 6, outputs may come in different/random orders. That's why sometimes you saw:

Proc 3 of 6 > ...
Proc 4 of 6 > ...
Proc 5 of 6 > ... and sometimes in another order.

Solution to Ordered Output

If you want **ordered, clean output**:

1. Let **each process** create its own message string.
2. Send the message to **process 0**.
3. Let **process 0 print them in rank order**.

This way no competition for stdout and Output is **predictable** and **clean**. That's exactly what the earlier "Greetings" MPI program did.

Why Required?

- **Trapezoidal Rule** is required to numerically approximate integrals when no closed form exists.
- **Input Handling** is required so the program isn't rigid.
- **Output Handling** is required to avoid messy, nondeterministic prints in parallel programs.

Trap:

It's just a **helper function** that computes the **trapezoidal rule estimate** of the integral over a **sub-interval** of [a,b]. Instead of repeating the trapezoid loop everywhere, they wrote it as a **separate function**:

```
double Trap(
   double left_endpt,   /* input: left endpoint */
   double right_endpt,  /* input: right endpoint */
   int trap_count,      /* input: number of trapezoids in this subinterval */
```

```
 double base_len      /* input: width of each trapezoid */
) {
   double estimate, x;
   int i;

   // First & last points (half weight)
   estimate = (f(left_endpt) + f(right_endpt)) / 2.0;

   // Middle points (full weight)
   for (i = 1; i <= trap_count - 1; i++) {
      x = left_endpt + i * base_len;
      estimate += f(x);
   }

   // Multiply by width to get area
   estimate = estimate * base_len;

   return estimate;
}
```

Why this design?

1. **Reusability**
   Each MPI process is assigned a **sub-interval** of [a,b].
   - Rank 0 integrates its chunk.
   - Rank 1 integrates its chunk.
   - … and so on.

Each process just calls Trap(left, right, trap_count, h) for its assigned region. Cleaner code (no need to duplicate trapezoid logic everywhere).

Mathematical reason Recall trapezoidal rule formula:
That's *exactly* what lines 9–14 implement:

- First & last terms are halved.
- Middle terms added in a loop.
- Multiply by base length.

$$\int_a^b f(x)\,dx \approx h\left[\frac{f(a)+f(b)}{2}+\sum_{i=1}^{n-1} f(a+ih)\right]$$

3. **MPI Parallelism Reason**
    - The *whole* interval [a,b][a,b][a,b] is split among processes.
    - Each process computes its **local integral** using Trap.
    - Finally, MPI Reduce combines all results into the **global integral**.

So Trap is a **serial worker** function that fits into the parallel MPI program.

Why Required?

- Without it, you'd need to repeat trapezoidal integration code inside every process → messy, error-prone.
- With it, each process just says: my_int = Trap(my_a, my_b, my_n, h); and then passes my_int to MPI for combining. It's like giving each worker in a team a calculator — they don't reinvent math; they just use the tool.
  The Trap function is required because it encapsulates **the trapezoidal rule calculation for a subinterval**, making it reusable, mathematically correct, and easy to plug into MPI parallel execution.

Output:

In MPI, **all processes can access stdout**, but there is **no built-in coordination** of output.

When multiple processes call printf at (roughly) the same time:

- The operating system decides the order in which their outputs go to the console.
- The result: **output order changes** across different runs (nondeterministic behavior).

Sometimes one process's message may even be **interleaved** with another's.

With **5 processes**, you might *by chance* get ordered output:

Proc 0 of 5 > Does anyone have a toothpick ?
Proc 1 of 5 > Does anyone have a toothpick ?
Proc 2 of 5 > Does anyone have a toothpick ?
Proc 3 of 5 > Does anyone have a toothpick ?

Proc 4 of 5 > Does anyone have a toothpick ?

But with **6 processes**, the scheduler may let process 5 print before process 3, so the order scrambles: Proc 0 of 6 > ...
Proc 1 of 6 > ...
Proc 2 of 6 > ...
Proc 5 of 6 > ...
Proc 3 of 6 > ...
Proc 4 of 6 > ...

This shows **nondeterminism** in parallel programs:

- It doesn't affect correctness if you only care about the values.
- But it can confuse debugging and result interpretation.
- If your algorithm relies on ordered or coordinated output, you **must handle it explicitly**.

To avoid jumbled output:

1. **Centralize printing**
    - Let only **process 0** print results.
    - Other processes send their messages/data to process 0 using MPI_Send.
    - Process 0 receives in rank order and prints sequentially.
2. **Ordered printing**
    - Each process prints only after the previous rank has finished, often enforced with MPI_Barrier or by sending tokens (like a baton pass).

```
# include < s t d i o . h>
# include <mpi . h>
i n t main ( void) {
i n t my_rank , comm_sz ;
MPI_Init ( NULL , NULL ) ;
MPI_Comm_size ( MPI_COMM_WORLD , &comm_sz ) ;
MPI_Comm_rank ( MPI_COMM_WORLD , &my_rank ) ;
printf ( " Proc %d of %d > Does anyone have a toothpick ?\n" ,my_rank , comm_sz) ;
MPI_Finalize ( ) ;
```

```
return 0 ;
}
```

Input:

- In MPI, stdin (keyboard input) is only accessible to process 0 in most implementations.
- If multiple processes could read stdin, it would cause chaos:
  - Should process 0 get the first line, process 1 the second?
  - Or process 0 the first character, process 1 the next?
- That ambiguity is avoided by letting only rank 0 read input.

Solution is to make process 0 read input and send it to all other processes.

```
void Get_input(
   int my_rank,  // current process ID
   int comm_sz,  // total number of processes
   double* a_p,  // output: interval start
   double* b_p,  // output: interval end
   int* n_p      // output: number of trapezoids
) {
   int dest;

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);

      // Send values to all other processes
      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 {
      // Other processes receive from process 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);
  }
}
```

How it works

1. **Rank 0 process**:
   - Prompts the user: "Enter a, b, and n".
   - Reads the values of a, b, n.
   - Sends them to all other processes using MPI_Send.
2. **All other processes**:
   - Receive the values from rank 0 using MPI_Recv.
   - After this, **every process has the same values** for a, b, and n.

To use this function (Get_input), we can simply insert a call to it inside our main function, being careful to put it after we've initialized my_rank and comm_sz:

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

Ensures **all processes get identical input data by** preventing confusion from multiple processes reading stdin. It uses the **same pattern as the greetings program**: one process (0) acts as a coordinator, others just listen.

3.4 Collective communication:

In the trapezoidal rule MPI program:

- Each process computes its **local integral estimate** over its subinterval.

- Then, instead of all processes contributing equally to the global sum, **process 0** receives all the partial results from the other processes and performs the addition.

So the communication pattern is:

Process 1 → sends result → Process 0

Process 2 → sends result → Process 0

...

Process N-1 → sends result → Process 0 And process 0 does **all the summing**.

Why is this a problem?

1. **Load imbalance**
   - Process 0 is doing extra work (summing everything).
   - The other processes do very little after sending their number.
2. **Communication bottleneck**
   - All messages funnel to process 0, which may cause delays if many processes are used.
3. **Scalability issues**
   - For a small number of processes, it's fine.
   - But with 1000+ processes, process 0 becomes a bottleneck because it has to handle 999 incoming messages.

So need for a better way of doing the global sum that distributes the effort more fairly across processes and that's where collective communication comes in.

3.4.1 Tree-structured communication

we might use a "binary tree structure," like that illustrated in Fig. 3.6. In this diagram, initially students or processes 1, 3, 5, and 7 send their values to processes 0, 2, 4, and 6, respectively. Then processes 0, 2, 4, and 6 add the received values to their original values, and the process is repeated twice:

1. a. Processes 2 and 6 send their new values to processes 0 and 4, respectively.
   b. Processes 0 and 4 add the received values into their new values.
2. a. Process 4 sends its newest value to process 0.
   b. Process 0 adds the received value to its newest value.

This solution may not seem ideal, since half the processes (1, 3, 5, and 7) are doing the same amount of work that they did in the original scheme. However,

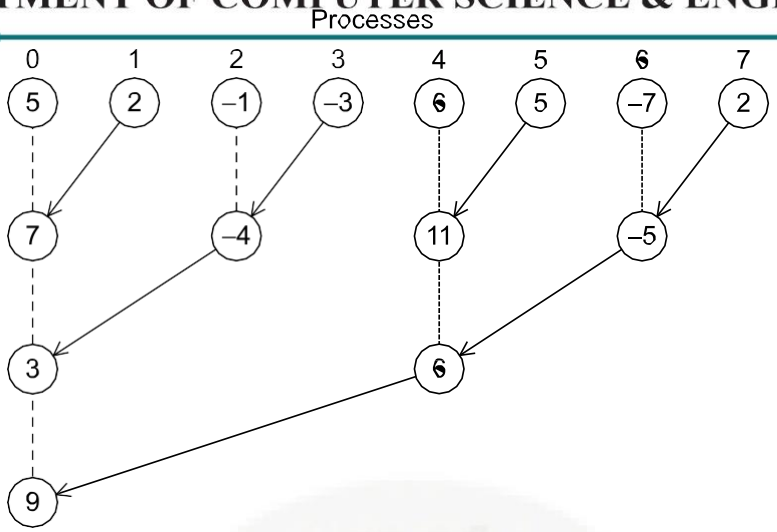


FIGURE 3.6

A tree-structured global sum.

you think about it, the original scheme required `comm_sz` 1 seven receives and seven adds by process 0, while the new scheme only requires three, and all the other processes do no more than two receives and adds.

For example, in the first phase, the receives and adds by processes 0, 2, 4, and 6 can all take place simultaneously. So, if the processes start at roughly the same time, the total time required to compute the global sum will be the time required by process 0, i.e., three receives and three additions. So we've reduced the overall time by more than 50%.

Furthermore, if we use more processes, we can do even better. For example, if `comm_sz` 1024, then the original scheme requires process 0 to do 1023 receives and additions, while it can be shown (Exercise 3.5) that the new scheme requires process 0 to do only 10 receives and additions. This improves the original scheme by more than a factor of 100

But coding this tree- structured global sum looks like it would take a quite a bit of work, and you'd be right. (See Programming Assignment 3.3.) In fact, the problem may be even harder. For example, it's perfectly feasible to construct a tree-structured global sum that uses different "process-pairings." For example, we might pair 0 and 4, 1 and 5, 2 and 6, and 3 and 7 in the first phase. Then we could pair 0 and 2, and 1 and 3 in the second, and 0 and 1 in the final. (See Fig. 3.7.)

Processes

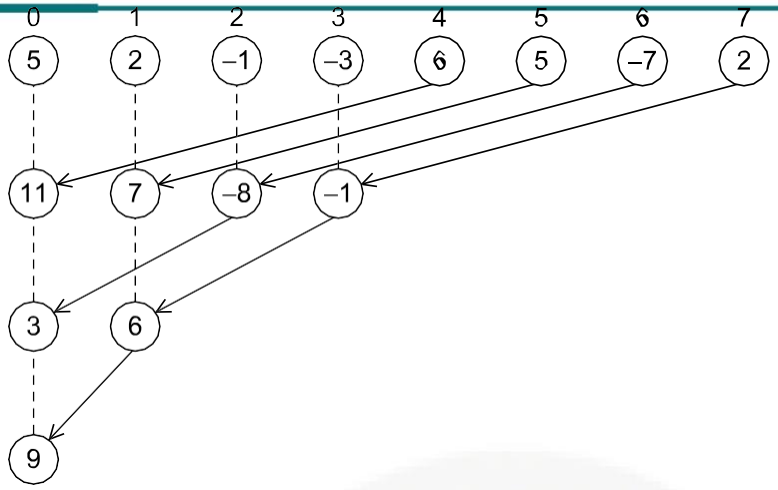


FIGURE 3.7

An alternative tree-structured global sum.

3.4.2 MPI_Reduce

MPI provides collective communication functions like MPI_Reduce instead of asking programmers to hand-craft tree-structured sums every time.

There are many possible ways (trees, pairings, orders) to implement a global sum. Choosing the "best" way depends on hardware (network topology, latency, etc.). It's not reasonable to expect every MPI programmer to reinvent and optimize this. So MPI shifts this responsibility to the MPI library implementer.

When *all processes* in a communicator (e.g., MPI_COMM_WORLD) participate in a communication, it's called a collective communication.

Example: global sum, every process has a value, and we need the sum across all processes. Contrast with point-to-point (MPI_Send, MPI_Recv), only two processes talk at a time.

Solution is MPI_Reduce

- Instead of just summing, MPI designed a single function to handle reduction operations like sum, max, min, product, etc.

Function prototype:

```
int MPI_Reduce(
   void* input_data_p,  // input buffer
   void* output_data_p,  // result buffer (only valid at dest_process)
   int count,          // number of elements
   MPI_Datatype datatype,// type (e.g. MPI_INT, MPI_DOUBLE)
   MPI_Op operator,     // operation (e.g. MPI_SUM, MPI_MAX)
   int dest_process,    // root process to collect result
   MPI_Comm comm        // communicator (e.g. MPI_COMM_WORLD)
);
```

global sum of scalars instead of manually coding send/receive loops.

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM,
    0, MPI_COMM_WORLD);
```

Each process provides its local number. MPI library takes care of efficiently summing them (using optimized tree-structured communication internally). The final result (total_int) appears only at process 0.

Table 3.2 Predefined Reduction Operators in MPI.

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

global sum of arrays

- If each process has an N-dimensional vector, MPI can sum them element-wise in **one call**:

```
double local_x[N], sum[N];
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Now process 0 has the **vector sum**.

- **Simplicity:** You don't have to write complicated code for tree-structured sums.
- **Performance:** The MPI library implementers know the hardware and optimize MPI_Reduce internally.
- **Flexibility:** Works not just for sums, but any associative operation (min, max, product, custom ops).

3.4.3 Collective vs. point-to-point communications

It's important to remember that collective communications differ in several ways from point-to-point communications:

1. All the processes in the communicator must call the same collective function. For example, a program that attempts to match a call to MPI_Reduce on one process with a call to MPI_Recv on another process is erroneous, and, in all likelihood, the program will hang or crash.
2. The arguments passed by each process to an MPI collective communication must be "compatible." For example, if one process passes in 0 as the dest_process and another passes in 1, then the outcome of a call to MPI_Reduce is erroneous, and, once again, the program is likely to hang or crash.

3. The output_data_p argument is only used on dest_process. However, all of the processes still need to pass in an actual argument corresponding to output_data_p, even if it's just NULL.
4. Point-to-point communications are matched on the basis of tags and communica- tors. Collective communications don't use tags. So they're matched solely on the basis of the communicator and the order in which they're called.

As an example, consider the calls to MPI_Reduce shown in Table 3.3. Suppose that each process calls MPI_Reduce with operator MPI_SUM, and destination process 0.
the matching of the calls to MPI_Reduce. The order of the calls will determine the matching, so the value stored in b will be 1 + 2 + 1 = 4, and the value stored in d will be 2 + 1 + 2 = 5.
A final caveat: it might be tempting to call MPI_Reduce using the same buffer for both input and output. For example, if we wanted to form the global sum of x on each process and store the result in x on process 0, we might try calling

```
MPI_Reduce (&x , &x , 1 , MPI_DOUBLE , MPI_SUM , 0 , comm );
```

However, this call is illegal in MPI. So its result will be unpredictable: it might produce an incorrect result, it might cause the program to crash; it might even produce a correct result. It's illegal, because it involves aliasing of an output argument. Two arguments are aliased if they refer to the same block of memory, and MPI prohibits aliasing of arguments if one of them is an output or input/output argument. This is because the MPI Forum wanted to make the Fortran and C versions of MPI as similar as possible, and Fortran prohibits aliasing.

Table 3.3 Multiple Calls to `MPI_Reduce`.

Time	Process 0	Process 1	Process 2
0	`a = 1; c = 2`	`a = 1; c = 2`	`a = 1; c = 2`
1	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`
2	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`

3.4.4 MPI_Allreduce

In our trapezoidal rule program, we just print the result. So it's perfectly natural for only one process to get the result of the global sum. However, it's not difficult to imagine a situation in which all of the processes need the result of a global sum to complete some larger computation. In this situation, we encounter some of the same problems we encountered with our original global sum. For example, if we use a tree to compute a global sum, we might "reverse" the branches to distribute the global sum (see Fig. 3.8). Alternatively, we might have the processes exchange partial re- sults instead of using one-way communications. Such a communication pattern is sometimes called a butterfly. (See Fig. 3.9.) Once again, we don't want to have to decide on which structure to use, or how to code it for optimal performance. For- tunately, MPI provides a variant of MPI_Reduce that will store the result on all the processes in the communicator:

```
int  MPI_Allreduce (
void *   input_data_p  /* in  */ ,
void *   output_data_p  /* out */ ,
int      count    /* in */ , MPI_Datatype datatype   /* in  */ , MPI_Op   operator   /* in  */ ,
MPI_Comm  comm       /* in */ );
```

The argument list is identical to that for MPI_Reduce, except that there is no dest_process since all the processes should get the result.

3.4.5 Broadcast

If we can improve the performance of the global sum in our trapezoidal rule program by replacing a loop of receives on process 0 with a tree structured communication, we ought to be able to do something similar with the distribution of the input data. In fact, if we simply "reverse" the communications in the tree-structured global sum in Fig. 3.6, we obtain the tree-structured communication shown in Fig. 3.10, and we can use this structure to distribute the input data. A collective communication in which data belonging to a single process is sent to all of the processes in the communicator is called a broadcast, and you've probably guessed that MPI provides a broadcast function:

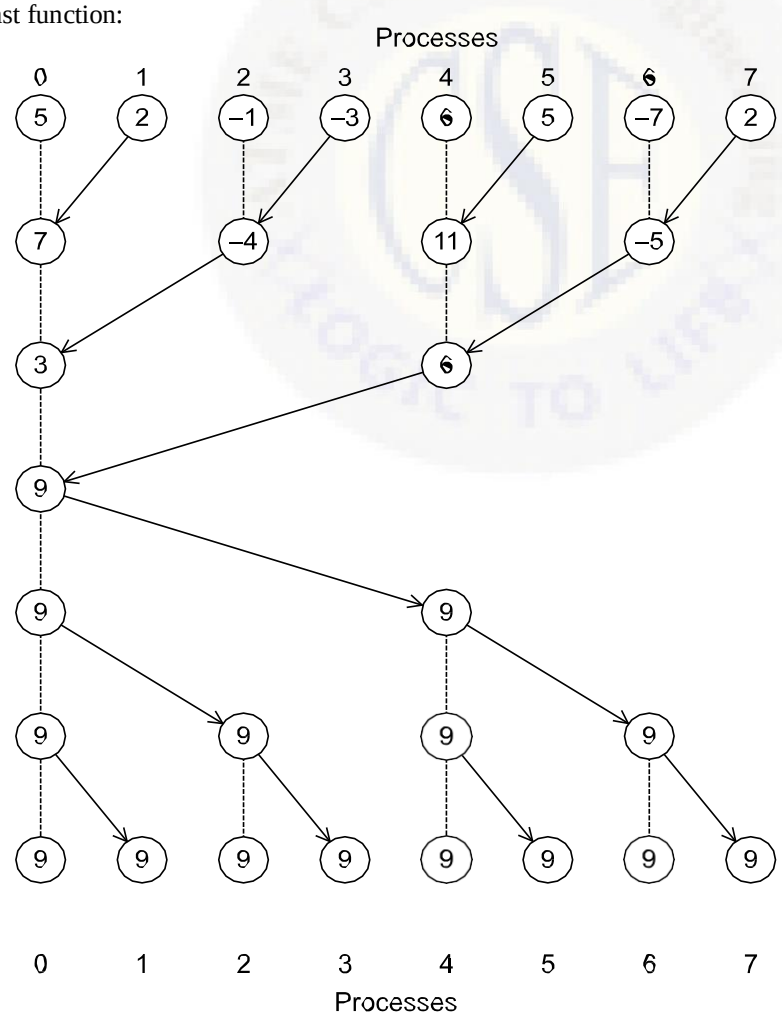


FIGURE 3.8

A global sum followed by distribution of the result.

```
int MPI_Bcast (
      void *          data_p          /* in/out */,
      int             count           /* in     */ ,
      MPI_Datatype    datatype        /* in     */ ,
      int             source_proc     /* in     */,
      MPI_Comm        comm            /* in     */);
```

The process with rank `source_proc` sends the contents of the memory referenced by `data_p` to all the processes in the communicator `comm`.

```
1  void fiet_input(
2        int        my_rank   /* in  */,
3        int        comm_sz   /* in  */,
4        double *   a_p       /* out */,
5        double *   b_p       /* out */,
6        int *      n_p       /* out */) {
7
8     if (my_rank == 0) {
9        printf("Enter a, b, and n\n");
10       scanf("%lf %lf %d", a_p, b_p, n_p);
11    }
12    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
13    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
14    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
15 }  /* Get_input */
```

Program 3.6: A version of `fiet_input` that uses `MPI_Bcast`.

Recall that in serial programs an in/out argument is one whose value is both used and changed by the function. For `MPI_Bcast`, however, the `data_p` argument is an input argument on the process with rank `source_proc` and an output argument on the other processes. Thus when an argument to a collective communication is labeled in/out, it's possible that it's an input argument on some processes and an output argument on other processes.

3.4.6 Data distributions

Suppose we want to write a function that computes a vector sum:

$$\begin{aligned}\mathbf{x} + \mathbf{y} &= (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \\ &= (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) \\ &= (z_0, z_1, \ldots, z_{n-1}) \\ &= \mathbf{z}\end{aligned}$$

If we implement the vectors as arrays of, say, **double**s, we could implement serial vector addition with the code shown in Program 3.7.

The work consists of adding the in- dividual components of the vectors, so we might specify that the tasks are just the additions of corresponding components. Then there is no communication between the tasks, and the problem of parallelizing vector addition boils down to aggregating the tasks and assigning them to the cores. If the number of components is *n* and we have comm_sz cores or processes, let's assume that *n* is evenly divisible by comm_sz and define local_n *n*/comm_sz. Then we can simply assign blocks of local_n consec- utive components to each process. The four columns on the left of Table 3.4 show an example when n 12 and comm_sz 3. This is often called a block partition of the vector.

An alternative to a block partition is a cyclic partition. In a cyclic partition, we assign the components in a round-robin fashion. The four columns in the middle of Table 3.4 show an example when n 12 and comm_sz 3. So process 0 gets component 0, process 1 gets component 1, process 2 gets component 2, process 0 gets component 3, and so on.

A third alternative is a block-cyclic partition. The idea here is that instead of using a cyclic distribution of individual components, we use a cyclic distribution of blocks of components. So a block-cyclic distribution isn't fully specified until we decide how large the blocks are. If comm_sz 3, n 12, and the blocksize b 2, an example is shown in the four columns on the right of Table 3.4.

each process will have local_n components of the vec- tor, and, to save on storage, we can just store these on each process as an array of local_n elements. Thus each process will execute the function shown in Program 3.8.

```
1  void Vector_sum(double x[], double y[], double z[], int n) {
2     int i;
3
4     for (i = 0; i < n; i++)
5        z[i] = x[i] + y[i];
6  }  /* Vector_sum */
```

Program 3.7: A serial implementation of vector addition.

Table 3.4 Different partitions of a 12-component vector among 3 processes.

	Components											
Process	Block				Cyclic				Block-cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

3.4.7

```
int MPI_Scatter(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */, void*
                     recv_buf_p  /* out */,
```

```
int             recv_count   /* in  */,
MPI_Datatype    recv_type    /* in  */, int
                src_proc     /* in  */,

MPI_Comm        comm         /* in  */);
```

If the communicator `comm` contains `comm_sz` processes, then `MPI_Scatter` divides the data referenced by `send_buf_p` into `comm_sz` pieces—the first piece goes to process 0, the second to process 1, the third to process 2, and so on. For example, suppose we're using a block distribution and process 0 has read in all of an *n*-component vector into `send_buf_p`. Then process 0 will get the first `local_n` *n*/`comm_sz` components, process 1 will get the next `local_n` components, and so on. Each process should pass its local vector as the `recv_buf_p` argument, and the `recv_count` argument should be `local_n`. Both `send_type` and `recv_type` should be `MPI_DOUBLE`, and `src_proc` should be 0. Perhaps surprisingly, `send_count` should also be `local_n`—`send_count` is the *amount of data going to each process*; it's *not* the amount of data in the memory referred to by `send_buf_p`. If we use a block distribution and `MPI_Scatter`, we can read in a vector using the function `Read_vector` shown in Program 3.9.

```
1  void  Read_vector (
2        double      local_a []    /* out */,
3        int         local_n       /* in  */,
4        int         n             /* in  */,
5        char        vec_name []   /* in  */,
6        int         my_rank       /* in  */,
7        MPI_Comm    comm          /* in  */) {
8
9     double * a = NULL;
10    int  i;
11
12    if  (my_rank  == 0) {
13       a = malloc (n*sizeof (double ));
14       printf ("Enter  the  vector  %s\n",  vec_name );
15       for  (i = 0; i < n; i++)
16          scanf ("%lf", &a[i]);
17       MPI_Scatter (a, local_n, MPI_DOUBLE, local_a, local_n,
18             MPI_DOUBLE , 0, comm );
19       free (a );
20    } else  {
21       MPI_Scatter (a, local_n, MPI_DOUBLE, local_a, local_n,
22             MPI_DOUBLE , 0, comm );
23    }
24 }  /* Read_vector */
```

Program 3.9: A function for reading and distributing a vector.

One point to note here is that `MPI_Scatter` sends the first block of `send_count` objects to process 0, the next block of `send_count` objects to process 1, and so on. So this approach to reading and distributing the input vectors will only be suitable if we're using a block distribution and *n*, the number of components in the vectors, is evenly divisible by `comm_sz`.

3.1.1 Allgather

As a final example, let's look at how we might write an MPI function that multiplies a matrix by a vector. Recall that if $A = (a_{ij})$ is an $m \times n$ matrix and $\mathbf{x}$ is a vector with n components, then $\mathbf{y} = A\mathbf{x}$ is a vector with m components, and we can find the ith component of y by forming the dot product of the ith row of A with $\mathbf{x}$:

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots a_{i,n-1}x_{n-1}.$$

(See Fig. 3.11.)

So we might write pseudocode for serial matrix multiplication as follows:

```
/* For each row of A */
for (i = 0; i < m; i++) {
/* Form dot product of ith row with x */
y[i] = 0.0;
for (j = 0; j < n; j++)
y[i] += A[i][j]*x[j];
}
```

In fact, this could be actual C code. However, there are some peculiarities in the way that C programs deal with two-dimensional arrays (see Exercise 3.14). So C programmers frequently use one-dimensional arrays to "simulate" two-dimensional arrays.

Module-1

Introduction to parallel programming, Parallel hardware and parallel software

Why Parallel Computing?

Shift in computer processor design around the early 2000s, when performance improvements from faster single-core processors started to slow down.

---

1. Why do we care? Aren't single-processor systems fast enough?

- At first glance, a 20% yearly increase still seems very good.
- But compared to the 50% yearly increase in the 1986–2002 era, it's much smaller:
  - 50% per year → ~60× faster in 10 years
  - 20% per year → ~6× faster in 10 years
- Many applications (like gaming, scientific computing, big data analysis, AI) need massive performance growth. A slowdown means we can't rely on just waiting for faster chips.

---

2. Why can't we just keep making faster single processors?

There are fundamental limits:

- Power and heat: Higher clock speeds cause chips to overheat ("power wall").
- Physics: Shrinking transistors further makes leakage currents and energy inefficiency worse.
- Memory bottleneck: CPUs are much faster than memory ("memory wall"), so the CPU often waits for data.

So, manufacturers couldn't just keep ramping up clock speed forever.

---

3. Why build multiprocessor (parallel) systems instead?

- By putting multiple cores (processors) on a single chip, manufacturers could still increase performance without increasing clock speed too much.
- This means instead of one super-fast brain, you get many brains working together.

- The catch: programs need to be written in a way that uses multiple processors (parallel programming).

---

4. Why can't we automatically convert serial programs into parallel ones?

- It's very hard because:
  - Many programs have dependencies (one step must finish before the next).
  - Automatic parallelization is limited — compilers can help, but they can't always figure out how to split work safely.
  - Some tasks are inherently sequential (Amdahl's Law).

So, developers must design software with parallelism in mind to take advantage of multiple cores.

<u>why we need ever-increasing performance:</u>

1. Past advances depended on computation

- Human genome decoding,
- Medical imaging improvements,
- Fast & accurate web searches,
- Realistic computer games.

These **wouldn't have been possible** without the huge leaps in processor performance over pastdecades.
Also, **new advances depend on older ones** → today's progress builds on yesterday's computational power.

---

2. As power grows, so do the problems we can solve

3. Key application areas needing more performance

- **Climate modeling**
  - Need highly accurate simulations including atmosphere, oceans, land, and ice.
  - Helps test the effects of interventions on climate change.
- **Protein folding**
  - Misfolded proteins are linked to diseases (Huntington's, Parkinson's, Alzheimer's).
  - Modeling proteins is extremely complex and limited by current computing power.
- **Drug discovery**
  - Computational genomics can identify alternative treatments when existing drugs fail for some patients.
  - Requires massive genome analysis.
- **Energy research**
  - Simulations of wind turbines, solar cells, and batteries can lead to cleaner, more efficient technologies.
- **Data analysis**
  - Data worldwide doubles every ~2 years.
  - Raw data (e.g., DNA sequences, collider data, medical imaging, astronomy, web search logs) is useless unless analyzed.
  - Analysis needs vast computational resources.

WHY WE'RE BUILDING PARALLEL SYSTEMS

1. The old model: Faster single processors

- Historically, performance improved by making **transistors smaller**.

- Smaller transistors leads to faster switching and faster processors.
- For decades, this gave huge performance gains (Moore's Law).

---

2. The problem: Power & heat (the "Power Wall")

- As transistor speed increased, **power consumption increased**.
- More power → more **heat dissipation**.
- By the early 2000s, **air cooling** couldn't handle the heat.
- Too much heat makes chips **unreliable**.
  So, we can't just keep raising clock speeds indefinitely.

---

3. But transistor density can still grow

- We can still **fit more transistors** on a chip (Moore's Law continues).
- The challenge: **How to use those extra transistors** if we can't just make them faster?

---

4. The solution is Parallelism

- Instead of **one super-fast core**, manufacturers build **many simpler cores** on one chip.
- Each **core = a complete processor (CPU)**.
- A chip with many cores is called as a **multicore processor**.
- Old-style processors with one CPU are now called **single-core systems**.

---

5. Why this matters

- **Economic reason**: The chip industry must keep improving products to survive.
- **Moral/innovation reason**: More computational power = more progress in science, medicine, energy, etc.

Why We Need to Write Parallel Programs

1. The problem with old (serial) programs

- Most existing programs were written for **single-core systems**.
- On a multicore system, you can run **multiple instances** of the same program (e.g., run 4 games at once), but that's **not useful** — users want one program to run **faster and better**, not more copies.
- Therefore: To use multiple cores effectively, programs must be **parallelized**.

2. Automatic conversion isn't enough

- Researchers have tried to create compilers that **translate serial code to parallel code**.
- Success has been **limited** because:
    - Translating each step independently into parallel code often leads to **inefficiency**.
    - Sometimes the **best parallel solution requires a completely new algorithm**, not just a step-by-step parallelization of the serial one.
    - Example: Matrix multiplication — turning it into parallel dot-products may be inefficient compared to designing a new parallel matrix multiplication algorithm.

Serial code(one core): Summation

```
sum = 0;
for (i = 0; i < n; i++) {
   x = ComputeNextValue(...);
   sum += x;
}
```

Parallel code (p cores, n ≫ p):

- Divide the loop into **p chunks**.
- Each core computes its **own partial sum**:

```
mysum = 0;
for(myi = myfirsti; myi < mylasti; myi++) {
```

```
    myx = ComputeNextValue(...);
    mysum += myx;
}
```

the earlier parallel summation example. This is about **reducing bottlenecks** in how the partial results are combined. **1. The first method: Centralized collection**

- Each core computes its own **partial sum (mysum)**.
- All cores send their results to the **master core (say, core 0)**.
- The master receives each value one by one and adds them up.

Problem: The **master core does all the work**.

- With 8 cores, master core handles **7 receives + 7 adds**.
- As the number of cores grows, the master becomes a **bottleneck**.

The second method: Pairwise (tree-style) reduction

- Instead of all cores sending to the master, we **combine results in stages**:
    - **Stage 1**: Pair the cores:
        - Core 0 + Core 1, Core 2 + Core 3, Core 4 + Core 5, Core 6 + Core 7.
    - **Stage 2**: Pair the winners (even-numbered cores now hold results):
        - Core 0 + Core 2, Core 4 + Core 6.
    - **Stage 3**: Final combination:
        - Core 0 + Core 4.
- Now, Core 0 ends up with the total sum, but the work was **spread across cores**.

Advantage:

- With 8 cores, Core 0 only does **3 adds** instead of **7**.
- In general, the number of steps is **$\log_2(p)$** (tree depth) instead of **p−1**.
    - Example:
        - 8 cores → $\log_2(8) = 3$ steps
        - 1024 cores → $\log_2(1024) = 10$ steps (instead of 1023 adds by master!)

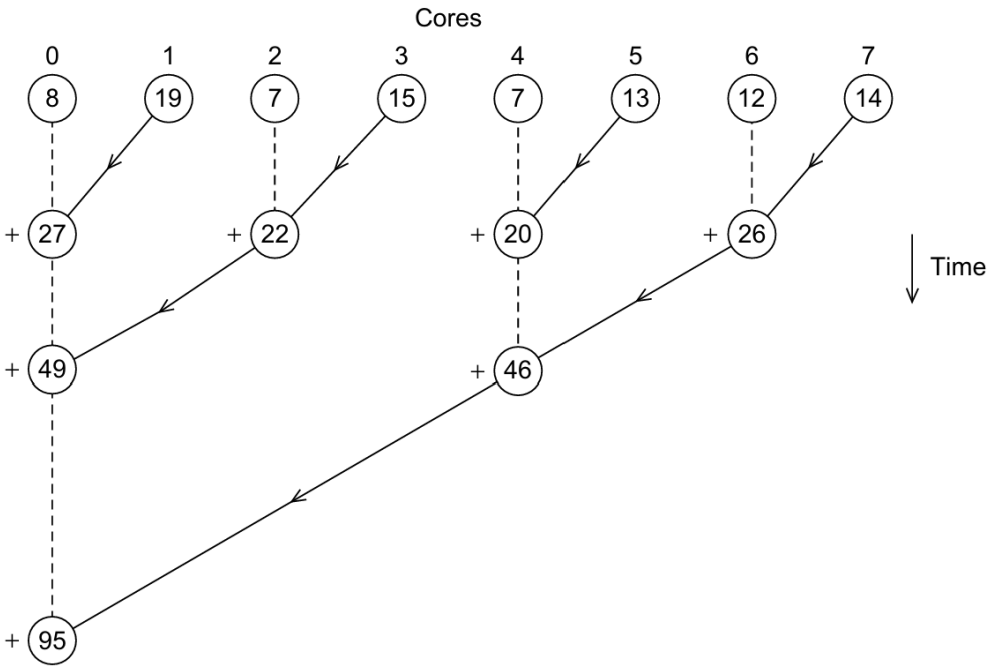


FIGURE 1.1

Multiple cores forming a global sum

1. This is called a **parallel reduction** (or tree-based reduction).

2. It avoids the **bottleneck of a single master core**.

3. The improvement grows dramatically with the number of cores

diagram shows Top row: Each core (0–7) starts with its local sum:

- Core 0: 8, Core 1: 19, Core 2: 7, Core 3: 15, Core 4: 7, Core 5: 13, Core 6: 12, Core 7: 14

First stage (pairwise sums):

- Core 0 + Core 1 → 27
- Core 2 + Core 3 → 22
- Core 4 + Core 5 → 20
- Core 6 + Core 7 → 26

Second stage (next level of pairing):

- 27 (Core 0) + 22 (Core 2) → 49
- 20 (Core 4) + 26 (Core 6) → 46

Third stage (final reduction):

- 49 + 46 → **95** (global sum, at Core 0).

Comparing the two global sum methods

- Method 1 (naïve / centralized):
  - Master adds up results from all cores.
  - Needs p − 1 operations (e.g., 999 adds for 1000 cores).
- Method 2 (tree reduction):
  - Results combined in pairs over stages.
  - Needs $\log_2(p)$ operations (e.g., only 10 adds for 1000 cores).
  - Much more efficient, especially as p grows.

how we actually write parallel programs and the main challenges

Two Main Approaches to Parallelism

1. **Task Parallelism**
   - Different cores do *different tasks*.
   - Example: In grading exams, one person grades only Question 1 (Shakespeare), another grades Question 2 (Milton), and so on.
   - Each is doing a *different job*, so the instructions differ.
2. **Data Parallelism**
   - Different cores do the *same task* on *different pieces of data*.
   - Example: Split the 100 exam papers into 5 piles of 20. Each TA grades *all questions* on their pile.
   - Same instructions, but applied to different data.

The Need for Coordination

Writing parallel programs isn't just about dividing work—it's also about **making the cores work together smoothly**. This requires:

1. **Communication**
    - Cores often need to send results to each other (e.g., partial sums in global sum).
    - One core may act as the "master" that collects and combines results.
2. **Load Balancing**
    - Work must be divided *fairly*.
    - If one core has too much work while others are idle, performance is wasted.
3. **Synchronization**
    - Cores don't all run at the exact same speed.
    - Sometimes they need to "wait for each other" before moving to the next step.
    - Example: If the master core is reading input data, other cores must wait until it's done before starting computation.

Parallel programming is about **dividing work** (task vs. data), **making cores cooperate** (communication, load balancing, synchronization), and **carefully coding** so that the system actually runs efficiently.

1. Concurrent computing

- **Definition**: Multiple tasks *appear* to be in progress at the same time.
- **Key idea**: It doesn't require multiple cores/processors. Even a single-core system with multitasking OS (like time-slicing between tasks) counts as concurrent.
- **Example**:
    - Your laptop running a browser, music player, and text editor *concurrently*.
    - Only one core may be executing at an instant, but the OS switches so fast that tasks appear to run simultaneously.

---

2. Parallel computing

- **Definition**: Multiple tasks are actually executed *at the same time* on multiple cores/processors.

- **Key idea**: Tasks are *tightly coupled* — they work together on one problem.
- **Typical setting**: Shared-memory systems or high-speed cluster.
- **Example**:
    - Adding 1 billion numbers using 8 CPU cores.
    - Each core sums a portion, and then partial sums are combined.
    - Requires **coordination** (synchronization, communication, load balancing).

---

3. Distributed computing

- **Definition**: Multiple tasks run on different computers (often geographically separated) that communicate via a network.
- **Key idea**: Tasks are *loosely coupled* — not necessarily created together.
- **Typical setting**: Cloud, internet-scale systems.
- **Example**:
    - Google's web search: thousands of servers across the world crawl, index, and serve results.
    - Each server does part of the work, but they aren't "sharing memory," they exchange messages.

Parallel Hardware and Parallel Software: SOME BACKGROUND

1. Von Neumann Architecture (Classical Computer Design)

The "classical" von Neumann architecture consists of main memory, a central processing unit (CPU) or processor or core, and an interconnection between the memory and the CPU. Main memory consists of a collection of locations, each of which is capable of storing both instructions and data. Every location consists of an address, which is used to access the location and the contents of the location—the instructions or data stored in the location.

The central processing unit is divided into a control unit and an arithmetic and logic unit (ALU). The control unit is responsible for deciding which instructions in a program should be executed, and the ALU is responsible for executing the actual instructions. Data in the CPU and information about the state of an executing program are stored in special, very fast storage called registers. The control unit has a special register called the program counter. It stores the address of the next instruction to be executed.

Instructions and data are transferred between the CPU and memory via the interconnect. This has traditionally been a bus, which consists of a collection of parallel wires and some hardware controlling access to the wires. A von Neumann machine executes a single instruction at a time, and each instruction operates on only a few pieces of data. See Figure 2.1.

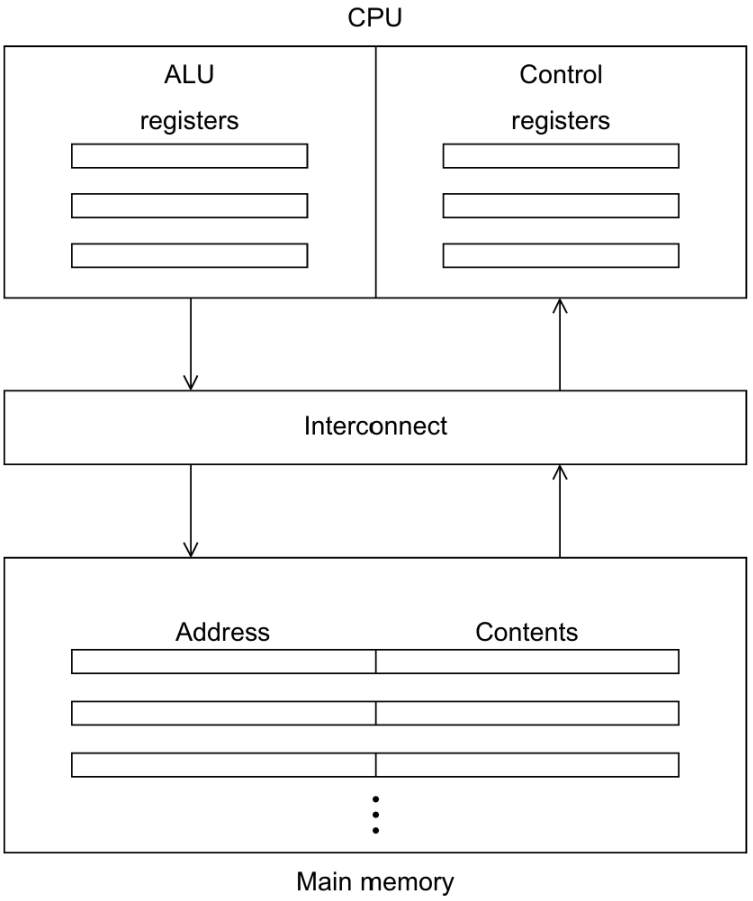


FIGURE 2.1

The von Neumann architecture

When data or instructions are transferred from memory to the CPU, we sometimes say the data or instructions are fetched or read from memory. When data are transferred from the CPU to memory, we some times say the data are written to memory or stored. The separation of memory and CPU is often called the **von Neumann bottleneck,**

Von Neumann Bottleneck

- Problem: The CPU is much faster than the memory access speed.
- CPU may execute 100+ instructions in the time it takes to fetch one piece of data from memory.
- The bus/interconnect limits how quickly data & instructions travel.

Analogy:

- CPU is like factory making products.
- Memory is warehouse storing raw materials (data) and finished products (results).
- Road (bus) is the transport system between them.
- If the road is too narrow (limited bandwidth), the factory workers sit idle because raw materials arrive too slowly.

Why This Matters

- This bottleneck makes computers inefficient.
- If CPU is starved of data/instructions, speed improvements in CPU alone don't help much.
- Hence, engineers started modifying this architecture (pipelining, caching, parallelism, etc.) → to reduce idle time.

Key Concepts: Processes, Multitasking, and Threads

1. Process

When you run a program (say, open Chrome), the OS creates a process. A process is an *active instance* of a program + its resources.

A process contains:

- The program code (executable machine instructions).
- Memory areas:
  - Call stack → tracks active functions.
  - Heap → dynamic memory (e.g., malloc/new).
  - Other memory → global vars, data.
- Resource descriptors → files, network sockets, etc.
- Security info → what it can/can't access.
- State info → is it running, waiting, blocked? Plus register values.

Think of a process like a workspace assigned to a worker in an office:

- Desk (memory),
- Tools (resources),
- Rules (security),
- Current task list (program counter + state).

---

2. Multitasking

- Modern OS allows multiple processes to appear to run at the same time.
- Even on a single CPU, the OS divides time into time slices (a few ms each).
- CPU runs process A → then switches to process B → then C → then back to A (this is context switching).If a process is waiting for something (e.g., reading from disk), it gets blocked → OS switches to another process that can keep working.

Analogy:

- One cook (CPU) works on 10 dishes (processes).
- Each dish gets a few minutes of attention before the cook moves to the next.

- If one dish has to simmer (blocked), the cook works on another.

3. Threads

- A thread is like a smaller unit of work inside a process.
- Processes usually have one “main” thread (the default execution path).
- But a process can create multiple threads and all share the same memory & resources.

Threads are lightweight compared to processes:

- Don’t need separate memory space, faster switching.
- Still need their own program counter and own call stack (so they can run independently).

Analogy:

- Process = restaurant.
- Threads = waiters inside that restaurant.
- They share the same kitchen & menu (resources), but each waiter keeps track of their own table (stack + program counter).

---

4. Thread Lifecycle (Fork–Join Model)

- When a process starts a new thread → the execution forks (splits).
- When a thread finishes, it joins back to the main process.Fig(2.2)

FIGURE 2.2

A process and two threads

Modifications to the von Neumann model

The **von Neumann bottleneck** means the CPU is very fast, but memory (RAM) is much slower. Since the CPU often has to wait for memory, overall performance suffers.

To fix this, computer engineers added **caching, virtual memory, and parallelism**. This part is about **caching**.

What is caching?

Think of it like this:

- CPU = **factory**
- Main memory = **warehouse**
- Road between them = **slow, two-lane road**

The CPU constantly needs raw materials (data & instructions) from memory. If every time it has to go to the warehouse far away, it wastes time.

Solution is to Build a small storeroom (cache) right next to the CPU. Cache stores a small amount of data that the CPU is very likely to need soon. It's much faster to access than main memory.

Key ideas behind caching

1. **Locality principle** – programs usually use data near what they just used.
    - **Spatial locality**: If you read z[0], you'll probably read z[1], z[2]... soon (arrays).
    - **Temporal locality**: If you use a variable once, you'll probably use it again soon.
2. **Cache lines (blocks)** – instead of fetching one item, CPU loads a whole block. Example: If each cache line = 16 floats, and program asks for z[0], the CPU loads z[0]..z[15]. The next 15 reads are super fast.
3. **Levels of cache (hierarchy)**
    - **L1**: Smallest, fastest (inside CPU core).

- **L2**: Bigger, slower.
- **L3**: Even bigger, shared between cores.
  CPU checks L1 → L2 → L3 → Main memory.
- If found = **cache hit**.
- If not = **cache miss** (stall, must fetch from slow RAM).

Writing data in cache

When CPU writes new data, cache & main memory may become **different**. Two ways to handle this:

1. **Write-through**: Update both cache & memory immediately (slower but consistent).
2. **Write-back**: Update cache only, mark it as *dirty*, and later update main memory when the cache line is replaced (faster but needs careful management).

Cache mappings

When the CPU asks for data from **main memory**, and we bring it into the cache, we must decide:

→ **Which cache slot should this memory block go into?**

There are 3 main strategies:

---

1. Fully Associative

- A memory block can go **anywhere** in the cache.
- Super flexible, but needs extra hardware to search everywhere.
- Example: Main memory line 0 could go into cache slot 0, 1, 2, or 3.

Think of this like parking in a shopping mall with **no assigned spots**—you can park anywhere.

---

2. Direct Mapped

- Each memory block has **exactly one place** in the cache.
- Simple & fast, but may cause many conflicts.
- Example: Cache has 4 slots, memory has 16 lines → slot = (line number mod 4).
    - Memory line 0, 4, 8, 12 → cache slot 0
    - Memory line 1, 5, 9, 13 → cache slot 1, etc.

Like a parking lot where **your car has only one assigned spot**.

---

3. N-way Set Associative

- A middle ground: each memory block can go into **one of n possible slots** (not anywhere, but not fixed to just one).
- Example: In a **2-way set associative cache**, cache slots are grouped into sets of 2:
    - Memory line 0 → can go into cache slot 0 **or** 1
    - Memory line 2 → can go into slot 2 **or** 3, etc.

Like parking in a lot where **you have 2 assigned spots to choose from**.

Eviction Policy (Which block gets kicked out?)

When the cache is full and a new block must be loaded:

- The most common rule = **Least Recently Used (LRU)** → Kick out the block that hasn't been used in the longest time.
    - Example: If line 0 (in slot 0) was just used, and line 2 (in slot 1) hasn't been used for a while, then line 2 will be replaced by line 4.

Like clearing out your fridge: you throw away the food you haven't touched for the longest time.

<u>Caches and programs: an example</u>

It's important to remember that the workings of the CPU cache are controlled by the system hardware, and we, the programmers, don't directly determine which data and

which instructions are in the cache. However, knowing the principle of spatial and temporal locality allows us to have some indirect control over caching. As an example, C stores two-dimensional arrays in “row-major” order. That is, although we think of a two-dimensional array as a rectangular block, memory is effectively a huge one-dimensional array. So in row-major storage, we store row 0 first, then row 1, and so on.

So memory layout of A[4][4] is:

A[0][0], A[0][1], A[0][2], A[0][3],

A[1][0], A[1][1], A[1][2], A[1][3],

A[2][0], A[2][1], A[2][2], A[2][3],

A[3][0], A[3][1], A[3][2], A[3][3]

Cache line:

- A cache doesn’t fetch single variables, it fetches a *block* (a "cache line").
- Example: 1 cache line = 4 elements.

Locality:

- **Spatial locality**: if you use A[0][0], you’ll probably use A[0][1], A[0][2] soon.
- **Temporal locality**: if you use a value once, you might use it again later.

```
for (i = 0; i < MAX; i++)
   for (j = 0; j < MAX; j++)
      y[i] += A[i][j] * x[j];
```

- Access pattern: row by row → contiguous memory.
- Example (MAX=4):
  - Access order: A[0][0], A[0][1], A[0][2], A[0][3] (all in **same cache line** → only **1 miss**).
  - Next row: A[1][0]..A[1][3] → again, just **1 miss**.

- **Total misses = 4** (one per row).

Cache lines are **used efficiently**. Once loaded, many elements are used before eviction.

Loop 2 (bad locality)

```
for (j = 0; j < MAX; j++)
   for (i = 0; i < MAX; i++)
      y[i] += A[i][j] * x[j];
```

- Access pattern: column by column is *non-contiguous* memory.
- Example (MAX=4):
  - Access order: A[0][0], A[1][0], A[2][0], A[3][0].
  - Each of these is in a **different cache line**, so each access = **cache miss**.
  - Then next column A[0][1], A[1][1]... → again **4 misses**.
- **Total misses = 16**. Every element reloads the cache, wasting most of the cache's capacity.

Performance Difference

- Loop 1: **4 misses**
- Loop 2: **16 misses**
- Larger arrays then difference becomes much bigger.
- That's why in experiments with MAX=1000, loop 1 was ~3x faster.

<u>virtual memory</u>

<u>Why Virtual Memory?</u>

- Programs think they each have their **own huge memory space** (virtual memory).
- In reality, many programs share the **same physical RAM**.

- Virtual memory gives:
    1. **Illusion of large memory** (even bigger than RAM, because disk is used as backup).
    2. **Protection** (one program cannot overwrite another's memory).
    3. **Flexibility** (any program can use any free RAM block).

Memory is divided into **pages** (usually 4 KB–16 KB).

Disk also has **swap space** divided into same-sized **pages**.

A program uses **virtual addresses** → these get mapped to **physical addresses** in RAM.

Page Table

- Each process has a **page table**: maps *virtual page number (VPN)* → *physical page number (PPN)*.
- Example:

Virtual page 5 → Physical page 20

Virtual page 6 → Physical page 11

- Virtual address is split into:
    - **Page offset** (within page, e.g., last 12 bits for 4 KB page).
    - **Virtual page number** (rest of the bits).

Translation = find VPN in page table, get PPN and combine with offset.

Problem with Page Tables

- To access memory, we first need to look into the **page table**.
- That's an *extra memory access* (slows things down).
- If the page table is big, it may not fit in cache → even more slowdowns.

Solution: TLB (Translation Lookaside Buffer)

- The TLB is a **cache for page table entries**.

- Stores most recently used **virtual and physical mappings**.
- Typically has **16–512 entries** (very small but very fast).
- Works like CPU cache:
    - **TLB hit** is fast translation, no need to check page table.
    - **TLB miss** is must check page table in RAM (slower).

Page Faults

- If a page is not in RAM at all (only on disk):
    - **Page fault** occurs.
    - OS must fetch the page from disk into RAM (super slow — millions of cycles).
- To reduce disk writes:
    - Virtual memory uses **write-back**, not write-through.
    - A **dirty bit** marks pages that were modified, only those are written back.

Who Manages What?

- **CPU hardware**:
    - Provides TLB.
    - Helps with address translation.
- **Operating system**:
    - Manages the page table.
    - Handles page faults.
    - Decides which pages to evict from RAM.

Instruction-Level Parallelism (ILP):

ILP = executing **multiple instructions at the same time** inside a CPU to improve performance.

There are two common approaches:

1. **Pipelining** → break instruction execution into stages (like an assembly line).

2. **Multiple issue** → start more than one instruction at the same time (superscalar processors).

Pipelining

Think of pipelining like a car assembly line:

- One worker bolts the engine.
- At the same time, another worker attaches the wheels on a different car.
- Another installs seats on yet another car.

Each worker specializes in one task → multiple cars are processed **in parallel**.

Example: Floating Point Addition

To add two floating point numbers (like 9.87×1049.87 \times 10^49.87×104 + 6.54×1036.54 \times 10^36.54×103), you need several steps:

1. **Fetch operands** (get the numbers).
2. **Compare exponents** (align the powers of 10).
3. **Shift operand** (so the exponents match).
4. **Add** the mantissas (the number parts).
5. **Normalize** (adjust so only one digit before decimal).
6. **Round** (keep precision).
7. **Store result** (save final answer).

Without pipelining → each addition takes **7 nanoseconds**. For 1000 additions, that's **7000 nanoseconds**.

With Pipelining

We build **7 hardware units**, one for each step.

- While step 1 (fetch operands) is working on the **2nd addition**,
- step 2 (compare exponents) is working on the **1st addition**,

- step 3 is idle (until the pipeline fills)

After the first few cycles (pipeline "warm-up"), the CPU produces **1 result every nanosecond** instead of every 7.

So instead of **7000 ns**, the whole loop takes only about **1006 ns** (almost 7x faster).

Practical Limits

Pipelining doesn't always give perfect speedup:

- If one stage is slower than others the whole pipeline slows to that stage's speed.
- If data needed by one instruction isn't ready yet, the pipeline **stalls**.
- Hazards (data hazards, control hazards, structural hazards) also cause delays.

Table 2.3 Pipelined Addition. Numbers in the Table Are Subscripts of Operands/Results

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

From the table 2.3 we see that after time 5, the pipelined loop produces a result every nanosecond, instead of every seven nanoseconds, so the total time to execute the for loop has been reduced from 7000 nanoseconds to 1006 nanoseconds—an improvement of almost a factor of seven.

Multiple Issue = instead of just one assembly line, you have **two (or more) parallel lines**.

- Example: if you have **two floating-point adders**, while one adds z[0] = x[0] + y[0], the other can add z[1] = x[1] + y[1].
- That way, the loop finishes in about **half the time**.

So:

- Pipeline ⇒ keeps the workers busy.
- Multiple issue ⇒ hires extra workers to do multiple tasks **at the same time**.

Static vs. Dynamic Multiple Issue

- **Static** = compiler decides in advance which instructions can run together. (Think: pre-planned schedule.)
- **Dynamic** = processor decides at run-time based on what's ready. This is **superscalar**.

peculation

Here's the clever trick: processors don't just wait for conditions to resolve — they **guess** what will happen and keep going.

1. **Branch speculation** (first example):

```
z = x + y;
if (z > 0) w = x;
else       w = y;
```

CPU guesses z > 0. It executes w = x **before knowing for sure**.

If the guess was right → great, no time lost.

If wrong → undo that step (rollback) and do w = y instead.

Memory speculation (second example):

```
z = x + y;
w = *ap;   // ap is a pointer
```

CPU guesses *ap is not pointing to z.

Executes both z = x + y; and w = *ap; in parallel.

If guess was right → saved time.

If wrong → discard speculative result and re-execute.

How CPUs Handle Wrong Guesses

- If **compiler speculates** → it inserts **checks** and corrective code.
- If **hardware speculates** → results are kept in a **buffer** until the CPU knows the guess was correct.
    - Correct guess ⇒ buffer committed to registers/memory.
    - Wrong guess ⇒ buffer discarded and instructions re-run.

In-Order vs. Out-of-Order

- Even superscalar CPUs **fetch in program order** and **commit results in order** (to avoid chaos in memory/registers).
- But they can **execute instructions out of order** internally — whichever ones are ready.
- Optimizing compilers, on the other hand, can actually **rearrange instruction order** ahead of time.

Thread-Level Parallelism (TLP) and hardware multithreading:

ILP = Instruction-Level Parallelism: The CPU tries to overlap or parallelize *instructions* within one program/thread. But ILP is limited when instructions are **dependent** on each other.

- Example: Fibonacci

```
f[0] = f[1] = 1;
for (i = 2; i <= n; i++)
  f[i] = f[i-1] + f[i-2];
```

Each f[i] depends on f[i-1] and f[i-2]. No chance to execute instructions in parallel inside one thread. So, ILP sometimes **runs out of parallelism**.

Thread-Level Parallelism (TLP)

Instead of squeezing parallelism *inside one thread*, we run **multiple threads** in parallel.

- Each thread is a *bigger unit of work* than an instruction.

- If one thread gets stuck (e.g., waiting for memory), the CPU can execute another thread.
- This is **coarser-grained parallelism** than ILP.

Hardware Multithreading

This is the **hardware trick** to keep the CPU busy by switching between threads.

1. Fine-Grained Multithreading

- CPU switches threads **after every instruction**.
- Advantage is No idle time if one thread stalls (say waiting for memory).
- Disadvantage is Even if one thread has plenty of ready instructions, it has to *share cycles* with others. Slows down that thread.

2. Coarse-Grained Multithreading

- CPU switches **only when a thread stalls** on a *long operation* (like memory access).
- Advantage is No unnecessary switching; a thread runs at full speed until it stalls.
- Disadvantage is Small/short stalls still waste CPU time.

---

3. Simultaneous Multithreading (SMT)

- This is the **modern solution** (used in Intel's Hyper-Threading, AMD's SMT).
- Runs **multiple threads at the same time** on a **superscalar CPU**.
- Each cycle, different threads can issue instructions to different functional units.
    - Example: Thread A issues an integer instruction while Thread B issues a floating-point instruction in the same cycle.

This combines ILP + TLP:

- Exploits **instruction-level parallelism** *inside each thread*.
- Exploits **thread-level parallelism** *across threads*.

If we also **prioritize "preferred threads"** (threads that have many instructions ready), SMT can reduce slowdown.

Distributed-memory interconnects

Distributed-memory interconnects are often divided into two groups: direct interconnects and indirect interconnects.

1. Direct interconnect

In direct interconnect each switch is directly connected to a processor-memory pair, and the switches are connected to each other. Fig. 2.8 shows a ring and a two-dimensional toroidal mesh. The circles are switches, the squares are processors, and the lines are bidirectional links.

One of the simplest measures of the power of a direct interconnect is the number of links. When counting links in a direct interconnect, it's customary to count only switch-to-switch links. This is because the speed of the processor-to-switch links may be very different from the speed of the switch-to-switch links.

a. Ring

A ring is superior to a simple bus, since it allows multiple simultaneous communications. However, it's easy to devise communication schemes, in which some of the processors must wait for other processors to complete their communications.

To get the total number of links, just add the number of processors to the number of switch-to-switch links. So, in the diagram for a ring (Fig. 2.8a), would ordinarily count 3 links instead of 6.

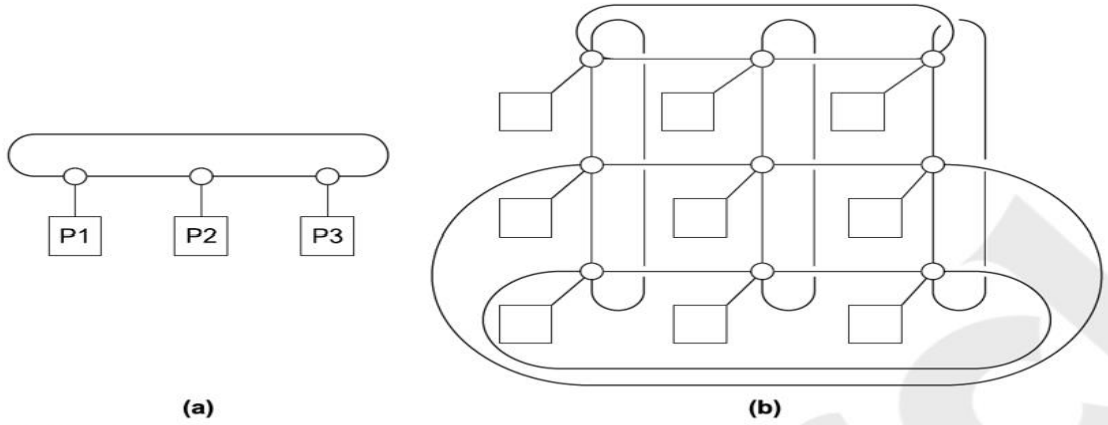


FIGURE 2.8
(a) A ring and (b) a toroidal mesh.

b. Toroidal mesh: The toroidal mesh will be more expensive than the ring, because the switches are more complex—they must support five links instead of three—and if there are *p* processors, the number of links is 2*p* in a toroidal mesh, while it's only *p* in a ring. However, the number of possible simultaneous communications patterns is greater with a mesh than with a ring. For the toroidal mesh (Fig. 2.8b), would count 18 links instead of 27.

Bisection width: To understand this measure, imagine that the parallel system is divided into two halves, and each half contains half of the processors or nodes. In Fig. 2.9(a) we've divided a ring with eight nodes into two groups of four nodes, and we can see that only two communications can take place between the halves.

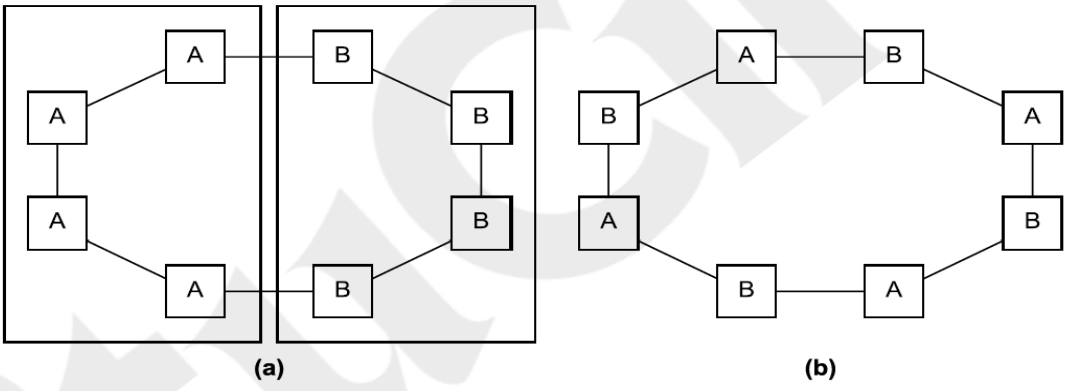


FIGURE 2.9
Two bisections of a ring: (a) only two communications can take place between the halves and (b) four simultaneous connections can take place.

the bisection width is to remove the minimum number of links needed to split the set of nodes into two equal halves. The number of links removed is the bisection width.

If we have a square two-dimensional toroidal mesh with $p = q2$ nodes (where q is even), then we can split the nodes into two halves by removing the "middle" horizontal links and the "wraparound" horizontal links. (See Fig. 2.10.) This suggests that the bisection width is at most $2q = 2\sqrt{p}$.

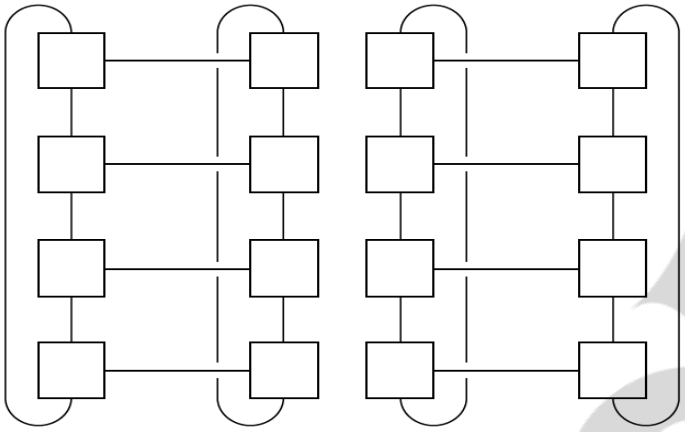


FIGURE 2.10
A bisection of a toroidal mesh.

The **bandwidth** of a link is the rate at which it can transmit data. It's usually given in megabits or megabytes per second. **Bisection bandwidth** is often used as a measure of network quality.

It's similar to bisection width. However, instead of counting the number of links joining the halves, it sums the bandwidth of the links. For example, if the links in a ring have a bandwidth of one billion bits per second, then the bisection bandwidth of the ring will be two billion bits per second or 2000 megabits per second.

c. **fully connected network:**

The ideal direct interconnect is a fully connected network, in which each switch is directly connected to every other switch. See Fig. 2.11. Its bisection width is $p^2/4$

However, it's impractical to construct such an interconnect for systems with more than a few nodes, since it requires a total of $p^2/2 - p/2$ links, and each switch must be capable of connecting to p links. It is therefore more a "theoretical best possible" interconnect than a practical one, and it is used as a basis for evaluating other interconnects.

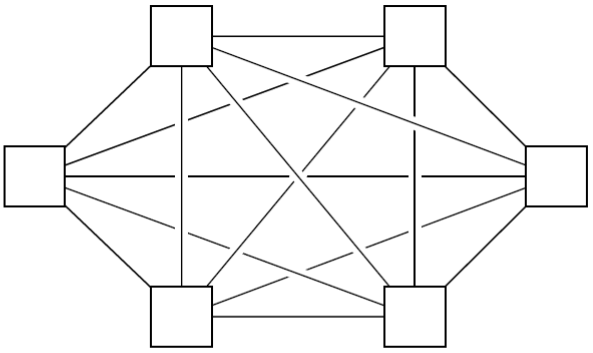


FIGURE 2.11

A fully connected network.

d. **hypercube:**

The hypercube is a highly connected direct interconnect that has been used in actual systems. Hypercubes are built inductively:

A one-dimensional hypercube is a fully connected system with two processors.

A two-dimensional hypercube is built from two one-dimensional hypercubes by joining "corresponding" switches.

A three-dimensional hypercube is built from two two-dimensional hypercubes. (See Fig. 2.12.) Thus a hypercube of dimension d has p = 2d nodes, and a switch in a d-dimensional hypercube is directly connected to a processor and d switches.

The bisection width of a hypercube is p/2, so it has more connectivity than a ring or toroidal mesh, but the switches must be more powerful, since they must support 1 + d = 1 + log2(p) wires, while the mesh switches only require five wires. So, a hypercube with p nodes is more expensive to construct than a toroidal mesh.

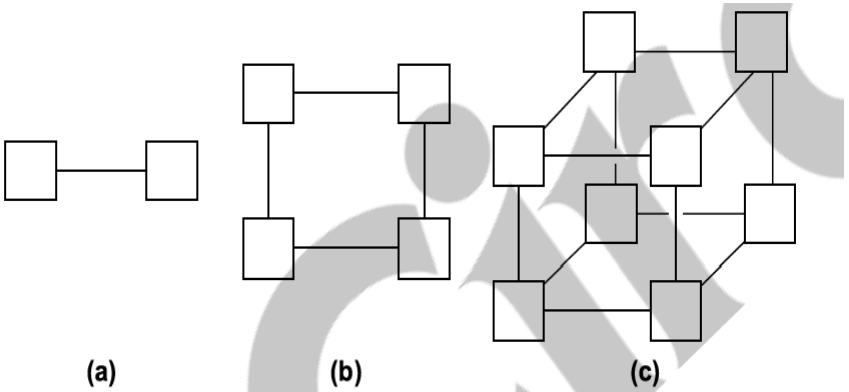


FIGURE 2.12

(a) One-, (b) two-, and (c) three-dimensional hypercubes.

Indirect interconnects: Indirect interconnects provide an alternative to direct interconnects. In an indirect interconnect, the switches may not be directly connected to a processor. They're often shown with unidirectional links and a collection of processors, each of which has an outgoing and an incoming link, and a switching network. (See Fig. 2.13.)

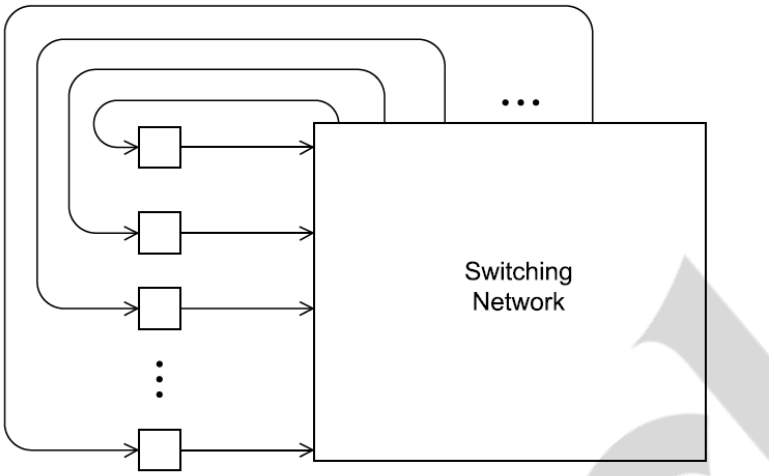


FIGURE 2.13

A generic indirect network.

The **crossbar** and the **omega network** are relatively simple examples of indirect networks.

The diagram of a distributed-memory crossbar in Fig. 2.14 has unidirectional links. Notice that as long as two processors don't attempt to communicate with the same processor, all the processors can simultaneously communicate with another processor.

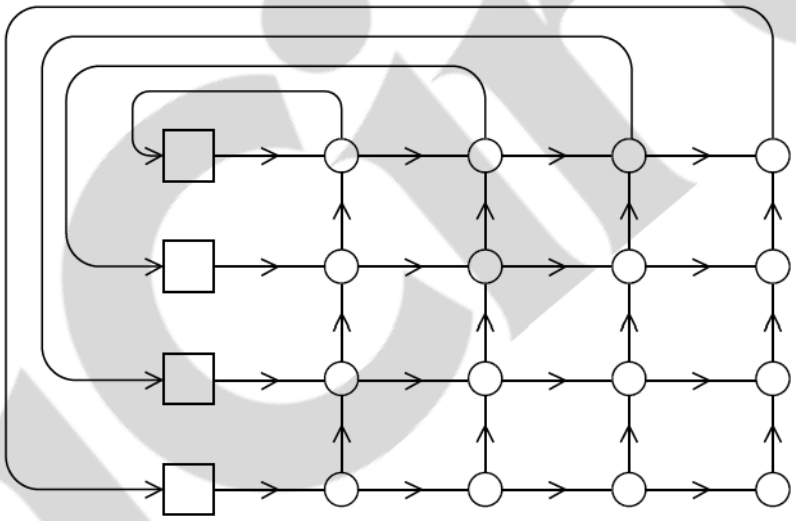


FIGURE 2.14

A crossbar interconnect for distributed-memory.

An omega network is shown in Fig. 2.15. The switches are two-by-two crossbars (see Fig. 2.16). Observe that unlike the crossbar, there are communications that cannot occur simultaneously.

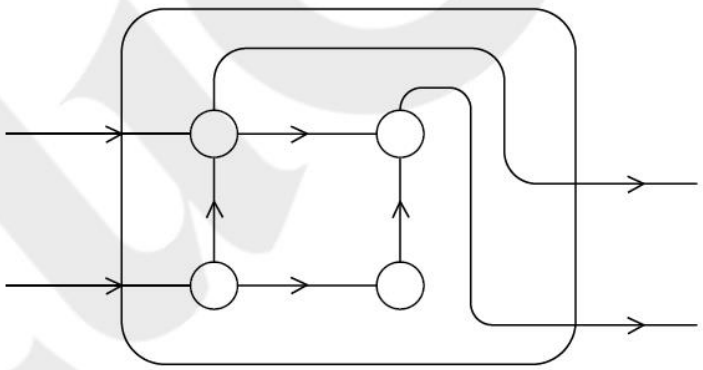


FIGURE 2.16

A switch in an omega network.

For example, in Fig. 2.15, if processor 0 sends a message to processor 6, then processor 1 cannot simultaneously send a message to processor 7. On the other hand, the omega

network is less expensive than the crossbar. The omega network uses $1/2p \log(p)$ of the 2×2 crossbar switches, so it uses a total of $2p \log(p)$ switches, while the crossbar uses $p2$.

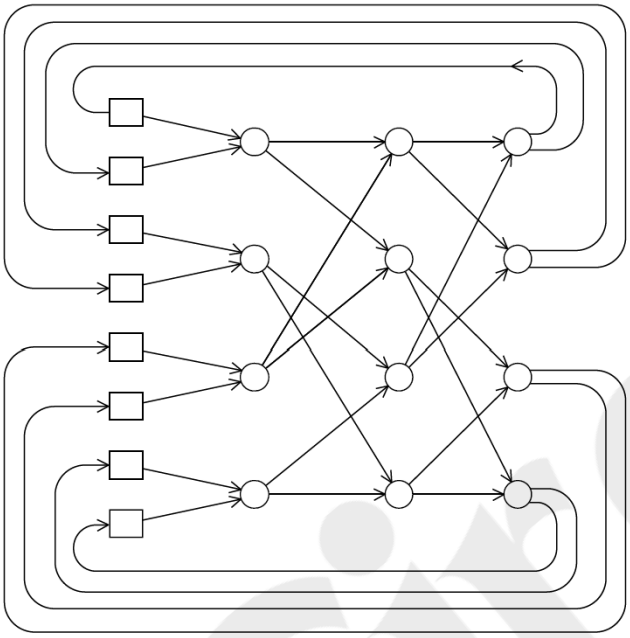


FIGURE 2.15
An omega network.

It's a little bit more complicated to define bisection width for indirect networks. However, the principle is the same: we want to divide the nodes into two groups of equal size and determine how much communication can take place between the two halves.

Alternatively, the minimum number of links that need to be removed so that the two groups can't communicate. The bisection width of a $p \times p$ crossbar is p, and the bisection width of an omega network is $p/2$.

Latency and bandwidth:

Any time data is transmitted, how long it will take for the data to reach its destination. Transmitting data between main memory and cache, cache and register, hard disk and memory, or between two nodes in a distributed-memory or hybrid system.

The latency is the time that elapses between the source's beginning to transmit the data and the destination's starting to receive the first byte.

The bandwidth is the rate at which the destination receives data after it has started to receive the first byte. So if the latency of an interconnect is l seconds and the bandwidth is

b bytes per second, then the time it takes to transmit a message of *n* bytes is message transmission time = *l* + *n*/*b*

Cache coherence: CPU caches are managed by system hardware: programmers don't have direct control over them. This has several important consequences for shared-memory systems.

To understand these issues, suppose we have a shared-memory system with two cores, each of which has its own private data cache. (See Fig. 2.17.)

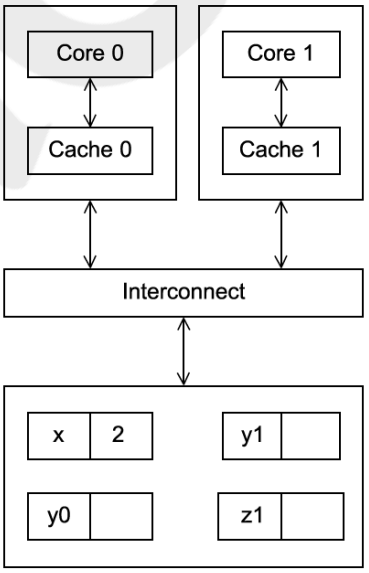


FIGURE 2.17

A shared-memory system with two cores and two caches.

As long as the two cores only read shared data, there is no problem. For example, suppose that x is a shared variable that has been initialized to 2, y0 is private and owned by core 0, and y1 and z1 are private and owned by core 1.

Now suppose the following statements are executed at the indicated times:

Time	Core 0	Core 1
0	`y0 = x;`	`y1 = 3*x;`
1	`x = 7;`	Statement(s) not involving x
2	Statement(s) not involving x	`z1 = 4*x;`

Then the memory location for y0 will eventually get the value 2, and the memory location for y1 will eventually get the value 6. However, it's not so clear what value z1 will get. It might at first appear that since core 0 updates x to 7 before the assignment to z1, z1 will get the value 4×7 = 28.

However, at time 0, x is in the cache of core 1. So, unless for some reason x is evicted from core 0's cache and then reloaded into core 1's cache, it actually appears that the original value x = 2 may be used, and z1 will get the value 4×2 = 8.

Note that this unpredictable behaviour will occur regardless of whether the system is using a write-through or a write-back policy. If it's using a write-through policy, the main memory will be updated by the assignment x = 7.
However, this will have no effect on the value in the cache of core 1. If the system is using a write-back policy, the new value of x in the cache of core 0 probably won't even be available to core 1 when it updates z1.

the caches for single processor systems provide no mechanism for ensuring that when the caches of multiple processors store the same variable, an update by one processor to the cached variable is "seen" by the other processors. That is, that the cached value stored by the other processors is also updated. This is called the **cache coherence** problem.

There are two main approaches to ensuring cache coherence: **snooping cache coherence** and **directory-based cache coherence**.

1. Snooping cache coherence: The idea behind snooping comes from bus-based systems: When the cores share a bus, any signal transmitted on the bus can be "seen" by all the cores connected to the bus.

Thus, when core 0 updates the copy of x stored in its cache, if it also broadcasts this information across the bus, and if core 1 is "snooping" the bus, it will see that x has been updated, and it can mark its copy of x as invalid.

This is more or less how snooping cache coherence works. The principal difference between our description and the actual snooping protocol is that the broadcast only informs the other cores that the cache line containing x has been updated, not that x has been updated.

First, it's not essential that the interconnect be a bus, only that it support broadcasts from each processor to all the other processors. Second, snooping works with both write-through and write-back caches.

In principle, if the interconnect is shared—as with a bus—with write-through caches, there's no need for additional traffic on the interconnect, since each core can simply "watch" for writes.

With write-back caches, on the other hand, an extra communication *is* necessary, since updates to the cache don't get immediately sent to memory.

2. Directory-based cache coherence:

In snooping cache coherence, whenever a variable changes in **any core's cache**, the update must be **broadcast** to *all* other cores.

- Example: Core 0 changes x = 5 → This info must be sent to **every other core** so their caches are updated or invalidated.

In a **small system** (like 4 cores), broadcasting is fast enough. All cores hear about changes almost instantly. But in a **large system** (hundreds/thousands of cores), broadcasting is **slow** as Too many "listeners" and more time to send update messages. The network connecting cores gets crowded. Distributed-memory system with a single address space each core has **its own** private memory.

But here, they set it up so any core can directly read/write to any variable in any other core's memory.

- Example: Core 0 can just do y = x even if x is in Core 1's memory.

The system *could* grow to thousands of cores, but snooping makes it not scalable because each write broadcast update, network overload and performance crash.

Directory-based cache coherence protocols attempt to solve this problem through the use of a data structure called a **directory.**

The directory is not stored in one place, it's **distributed**. Each core/memory pair keeps the directory info for the memory it owns. Directory has which cores have a copy of a given cache line? Whether that cache line is valid, not valid or shared?

When a core reads data Ex: Core 0 reads variable x which lives in Core 2's memory. Core 2:

1. Sends the data to Core 0.
2. Updates its directory entry → "Core 0 has a copy of x now."

When a core updates data Ex: Core 0 changes x = 10. Core 0's cache controller:

1. Checks the directory for x.

2. Finds only the cores that have a copy (say Core 3 and Core 5).
3. Sends invalidate messages only to those cores — not to all 1000 cores.

False sharing:

It's important to remember that CPU caches are implemented in hardware, so they operate on cache lines, not individual variables. This can have disastrous consequences for performance.

You have:

- **Two cores** (Core 0 and Core 1)
- An array y[8] of doubles (each double = 8 bytes)
- **Cache line size** = 64 bytes → can hold **8 doubles**
- So **all of y[0] to y[7]** fit in *one single cache line*.

Ex: for (i = 0; i < m; i++)

for (j = 0; j < n; j++)

y[i] += f(i, j);

We parallelize it:

- Core 0 → works on y[0] to y[3]
- Core 1 → works on y[4] to y[7]

```
/ * Pr i vat e v a r i a b l e s * /
i n t i , j , iter_count ;

/ * Shared v a r i a b l e s i n i t i a l i z e d by one core * /
i n t m , n , core_count

double y [ m ] ;
iter_count = m / core_count;

/ * Core 0 does t h i s * /
```

```
for ( i = 0; i < iter_count ; i ++)
for ( j = 0; j < n ; j ++)
y [ i] += f ( i , j ) ;
```

```
/ * Core 1 does t h i s * /
for ( i = iter_count ; i < 2* iter_count ; i ++)
for ( j = 0; j < n ; j ++)
y [ i] += f ( i , j ) ;
```

Logically they are working on *different elements*, so there's no data dependency. But caches don't work on variables Caches store cache lines, not individual variables.

If y[0] to y[7] are in one cache line, then any update to any element in that line means:

1. The entire cache line gets marked as invalid in the other core's cache.
2. That core must fetch the updated line again from memory (or other core's cache) before continuing.

In practice Core 0 updates y[0], cache line is marked *modified* in Core 0's cache, invalid in Core 1's cache. Core 1 then tries to update y[4], Cache controller sees that its cache line is invalid and fetches the whole line from Core 0 (or main memory).

Now Core 1 owns the line, Core 0's copy is invalidated. Core 0 next tries to update y[1] but must fetch the whole line back again.

This keeps happening for every single update, even though they're working on different y[i] values.

Why this is called False Sharing

- **True sharing**: both cores actually read/write the *same variable*.
- **False sharing**: they read/write *different variables*, but those variables happen to be on the same cache line, so the hardware treats it like shared data.

Instead of using the fast L1 cache, most updates end up going through slow main memory or cross-core cache transfers.

L1 cache access: ~1–4 CPU cycles

Main memory access: ~100–300 cycles huge slowdown.

To reduce false sharing:

1. Pad the array so each core works on elements in a different cache line.
2. Use local temporary storage inside each thread and merge results at the end.
3. Arrange data structures so different threads don't share cache lines.

Shared-memory vs. distributed-memory

1. Shared-memory is easy for programmers

- In shared memory, all processors can just **access the same variables** without worrying about explicit message passing.
- Example: Two threads can just read/write to the same array without sending messages between them.

Shared-memory needs a high-speed interconnect (bus or crossbar) to connect all processors to the same memory.

Bus problem:

- When few processors are connected, it works fine.
- But as the number of processors increases, they **compete** for the same bus.
- More processors lead to **more conflicts** hence performance drops fast.
- So, buses are good only for **small systems** (like your laptop with 4–16 cores).

Crossbar problem:

- Allows more simultaneous connections than a bus.
- But **cost rises steeply** as you add processors.
- Large crossbars are *very expensive*, so rare in big systems.

Distributed-memory is cheaper to scale

- Uses **direct interconnects** like:
  - **Hypercube**
  - **Toroidal mesh**
- Each processor has **its own local memory** so No single "traffic jam" point.
- Can scale to **thousands of processors** at much lower cost.

- Great for **huge problems** needing massive computation or huge datasets.

Parallel software

Multicore processors are in:

- Desktops
- Servers
- Mobile phones and tablets

This means your device can do more than one thing at the same time in hardware. But software hasn't fully caught up. Back in **2011**, the book said: "We have parallel hardware, but not much parallel software."

In **2021**, the situation is better but still incomplete:

- **System software** (like your OS) now uses multiple cores.
- Many popular apps (Excel, Photoshop, Chrome) can use more than one core.
- But **many programs still use only a single core**.
- Many programmers have **never written parallel code**

In the past, we could just rely on:

- Faster CPUs
- Smarter compilers

This gave automatic speed boosts without changing the software.

Now, CPU clock speeds aren't rising much, performance growth comes mainly from adding more cores. If a program only uses one core, it won't get faster on newer hardware.

The solution is to Learn parallel programming: Developers must learn to write programs that can use:

- **Shared-memory architectures** (threads, common memory)
- **Distributed-memory architectures** (message passing)
- Both **MIMD** and **SIMD** execution models.

Caveats

This section is basically giving two warnings (caveats) before diving into parallel programming details and also clarifying SPMD. Mainly focus on what's often called single program, multiple data, or SPMD, programs.

SPMD is One program file, multiple cores running it at the same time. Each core runs the same code, but can do different things depending on its ID.

Ex: if (I'm thread/process 0)

do this;

else

do that;

Here, all cores are running the same executable, but thread 0 does one thing and thread 1 does another. The same operation is done on different chunks of data called as **data parallelism.**

if (I'm thread 0)

process first half of the array;

else // I'm thread 1

process second half of the array;

Task parallelism: Different threads do different tasks.

Example:

Thread 0: read data from disk

Thread 1: process the data

Thread 2: write results to a file

SPMD is flexible and it can implement both **data-parallel** and **task-parallel** designs using the same "single program" model, just by branching logic based on the thread or process ID.

Coordinating the processes/threads

For example,

suppose we have two arrays and we want to add them:

```
double x [ n ], y [n ] ;
. . .
for ( int i = 0; i < n ; i ++)
x [ i] += y [ i ] ;
```

Serial version: One loop runs start to finish.

Parallel version: If you have p threads:

Thread 0 → handles x[0] to x[n/p - 1]

Thread 1 → handles x[n/p] to x[2n/p - 1]

and so on.

This is easy because, each thread can work independently (no need to share intermediate results). No communication between threads is needed once the work is split.

When splitting work:

1. **Load balancing** — Each process/thread should get roughly the same amount of work (avoid one thread sitting idle while others are busy).
2. **Minimize communication** — Less data transfer between threads means faster performance.

If both are satisfied, you get high efficiency.

Load balancing → Ensuring even distribution of work.

Parallelization → Turning a serial program into a parallel one.

Embarrassingly parallel → Problems so easy to parallelize that no coordination is needed (e.g., image processing pixel-by-pixel, array addition). *Despite the name, it's a good thing, not something to be ashamed of.*

For complex problems: You need synchronization, making sure threads reach certain points together (e.g., waiting for all to finish before moving on).

You need communication for Sharing results, sending data to other threads.

Examples:

- **Shared-memory systems:** Communicate by directly reading/writing shared variables, but require synchronization to avoid conflicts.
- **Distributed-memory systems:** Communicate by sending messages between processes (MPI), which can also serve as synchronization points.

---

shared-memory:

Any thread can read/write **Shared variables**. Whereas **Private variables**, Belong to one thread only and Communication is done via shared variables (implicit — no need to send explicit messages like in distributed memory).

Dynamic Threads Paradigm

- **How it works:**
    1. Master thread waits for a new task (e.g., request from network).
    2. When work arrives → master **forks** a new worker thread.
    3. Worker thread does its job → **joins** back to master → ends.
    4. Repeat for each task.
- **Advantages:**
    - **Efficient resource usage** → Threads only exist while they are needed.
    - Ideal for systems where workload changes a lot over time.
- **Disadvantages:**
    - **Overhead of forking/joining** each time a task arrives.
    - Not the fastest for continuous or heavy workloads.

Static Threads Paradigm

How it works:

1. Master thread creates all worker threads at the start.
2. All threads stay alive until all work is done.
3. At the end, all workers join the master, and then the program cleans up.

Advantages:

- Lower overhead → No repeated thread creation/destruction.
- Better performance for workloads that are steady or predictable.

Disadvantages:

- Less efficient resource usage since idle threads still consume memory & CPU stack space.
- Wastes resources if there's not always enough work to keep all threads busy.

Nondeterminism?

- In **MIMD systems** (Multiple Instruction, Multiple Data) where multiple processors (or threads) run at the same time, they usually don't stay perfectly in sync.
- This means **the same input** might produce **different outputs** depending on how the processors finish their tasks. This unpredictability is called **nondeterminism**.

Example 1: Printing with Two Threads

- Imagine we have two threads:
  - **Thread 0** → has a private variable my_x = 7
  - **Thread 1** → has a private variable my_x = 19

Both threads run this code:

printf("Thread %d > my_x = %d\n", my_rank, my_x);

Possible outputs: Case 1:

Thread 0 > my_x = 7

Thread 1 > my_x = 19

Case 2:

Thread 1 > my_x = 19

Thread 0 > my_x = 7

Their outputs might get **mixed together** (e.g., half of one line, half of another), because both are writing to the screen at the same time. The point is since the threads are independent, we **cannot predict** the order.

Is nondeterminism always bad?

- **Sometimes it's okay**, in the print example, since each output is labelled with the thread ID, order doesn't matter.

- **Sometimes it's dangerous**, especially in shared-memory programs, where threads work on the same data.

Example 2: Race Condition with Shared Variable

Suppose:

- Each thread calculates some value and stores it in its own variable my_val.
- Then, both threads try to add their values into a shared variable x, which starts at 0.

The code looks like this:

```
my_val = Compute_val(my_rank);
x += my_val;
```

How computers actually do x += my_val

To simplify, assume:

1. Load x from memory into a register.
2. Load my_val into another register.
3. Add them.
4. Store result back into memory.

Time	Core 0 (Thread 0)	Core 1 (Thread 1)
0	Done computing my_val=7	Still computing my_val
1	Loads x=0	Finished computing my_val
2	Loads my_val=7	Loads x=0
3	Adds → result=7	Loads my_val=19
4	Stores x=7	Adds 0+19=19
5	(Leaves)	Stores x=19

Final result of x = 19 (Thread 0's contribution is lost!)

This is called a **race condition** where both threads are "racing" to update x, and the outcome depends on who finishes first.

Fixing Race Conditions using Critical Sections

- The update x += my_val must be done **atomically** (all steps happen as one unit).
- To guarantee that, we put it inside a **critical section** → a block of code that **only one thread can run at a time**.

Using Mutex (Lock)

- A **mutex** (mutual exclusion lock) is a special object provided by the system/hardware.
- Process:
  1. A thread must **lock** the mutex before entering the critical section.
  2. Do the critical work (e.g., x += my_val).
  3. **Unlock** the mutex when done.

Example:

```
my_val = Compute_val(my_rank);
Lock(&add_my_val_lock);
x += my_val;
Unlock(&add_my_val_lock);
```

This ensures only one thread updates x at a time. But it doesn't force any specific order (thread 0 or 1 can go first). Downside it makes that part of the program **serial** (not parallel). So we want **critical sections to be as short as possible**.

Alternative 1: Busy-Waiting

- Instead of a mutex, a thread can just **keep checking a condition** until it's allowed to proceed.
- Example: Thread 1 must wait for Thread 0.

```
my_val = Compute_val(my_rank);
if (my_rank == 1)
  while (!ok_for_1);   // busy-wait loop

x += my_val; // critical section

if (my_rank == 0)
  ok_for_1 = true; // allow thread 1 to continue
```

Here, Thread 1 will loop endlessly until ok_for_1 is set to true by Thread 0. Simple but **wasteful** → the CPU is busy "waiting" instead of doing real work.

Alternative 2: Semaphores

- Similar to mutexes but slightly more flexible.
- A **semaphore** can allow multiple threads to enter at once (depending on its count).
- Some synchronization problems are easier to solve with semaphores than mutexes.

Alternative 3: Monitors

- A **monitor** is a higher-level concept.
- Think of it like a special **object** where only one thread can call its methods at a time.
- Provides **mutual exclusion automatically** without explicit lock/unlock calls.

Nondeterminism → Output is unpredictable when threads run independently.
Race Condition → Happens when threads update shared data at the same time.
Critical Section → Code that must only be executed by one thread at a time.
Mutex (Lock/Unlock) → Most common way to protect critical sections.
Busy-Waiting → Thread keeps looping until allowed (simple but wasteful).
Semaphores → Like advanced locks, more flexible.
Monitors → Higher-level locks built into objects.

Thread Safety

What is Thread Safety?

- In many cases, functions written for **serial (single-threaded) programs** can also be used safely in **parallel (multithreaded) programs**.
- BUT there are exceptions, some functions don't behave correctly if multiple threads call them at the same time. When that happens, the function is said to be **not thread safe**.

Why does this happen?

It usually happens because of **static local variables** in C.

- **Normal local variables**:
  - Declared inside a function.
  - Stored on the **stack**.

- Each thread has its **own stack**, so each thread gets its own copy of the variable. Safe in multithreading.

- **Static local variables**:
  - Declared inside a function but with static.
  - They **remember their value between function calls** (they don't get destroyed).
  - All threads share the same variable. This can cause problems when multiple threads use the function.

Example: strtok in C

- The function strtok splits a string into smaller parts (substrings).
- It works like this:
  - On the **first call**, you pass it a string, and it remembers that string using a **static pointer**.
  - On the **next calls**, it keeps giving you the next part of the string, using that stored static variable.

Problem is What if **two threads** use strtok at the same time? Suppose: Thread 0 calls strtok first with "apple orange". Before Thread 0 finishes splitting, Thread 1 calls strtok with "cat dog". Now the **static variable inside strtok gets overwritten**.

- Thread 0's string ("apple orange") is lost.
- On its next call, Thread 0 might get pieces of Thread 1's string ("cat dog") instead.

What does this mean?

- strtok is **not thread safe**.
- If used in a multithreaded program it may produce **wrong results** or **random errors**.

General Rule:

- A function is **not thread safe** if multiple threads can **access or modify the same shared data** inside it.
- Many functions from single-threaded libraries **are thread safe**, but you must be careful with the ones that rely on shared state (like strtok).

Distributed-Memory

- In **shared-memory systems**, all processors (cores/threads) can access the same memory.
- But in **distributed-memory systems**, **each processor has its own private memory**. A processor can directly use **only its own memory**. So, if one processor wants data from another processor's memory, it **cannot read it directly**. Instead,

- they must **communicate**. The most widely used way is **message passing**. Other APIs exist, but message passing is the standard.

You can even use message-passing on a shared-memory system by pretending memory is private for each thread and exchanging data using messages.

How are distributed-memory programs run?

- They are usually started as **multiple processes**, not threads. Because processors in distributed-memory systems may be:
  - Independent CPUs
  - Running their own operating systems
  - Without shared infrastructure to create threads across nodes

So **Processes** (not threads) each with its own memory.

Message Passing

A **message-passing API** provides at least two functions:

1. **Send** → to send data to another process
2. **Receive** → to get data from another process

Each process is identified by a **rank** (like an ID number).

- If there are p processes, the ranks are 0, 1, …, p–1.

Example: Process 1 sends to Process 0

Pseudocode:

```
char message[100];
...
my_rank = Get_rank();

if (my_rank == 1) {
   sprintf(message, "Greetings from process 1");
   Send(message, MSG_CHAR, 100, 0);
}
else if (my_rank == 0) {
   Receive(message, MSG_CHAR, 100, 1);
   printf("Process 0 > Received: %s\n", message);
```

What happens here?

- Get_rank() → tells the process its ID.
- If rank = 1 → it creates a message ("Greetings from process 1") and sends it to process 0.
- If rank = 0 → it waits to receive the message and then prints it.

SPMD Program (Single Program, Multiple Data)

- Both processes run the **same code** (same executable).
- But their behavior differs based on rank.

Local Variables → Each process's message variable refers to **different memory blocks** (private). Some programmers emphasize this by naming them my_message or local_message.

Output (stdout/stderr) → Usually, all processes can print to the screen.

- Even though message-passing APIs don't guarantee it, most implementations allow it.

Behaviour of Send and Receive

Different implementations may behave differently:

- **Blocking Send** → The Send call waits until the matching Receive starts.
- **Buffered Send** → The Send call copies data into its own buffer, so the sender can continue immediately.
- **Blocking Receive** → The Receive call usually waits until the data actually arrives.

There are other variations too (non-blocking send/receive, synchronous send, etc.).

More Functions in Message Passing

Message-passing APIs usually provide **extra communication tools**:

- **Broadcast** → One process sends the same data to all processes.
- **Reduction** → Combine results from all processes into one (e.g., sum, max, min).
- **Process Management** → Start/stop/manage processes.
- **Complex Data Handling** → Sending structured data across processes.

The most widely used API is **MPI (Message-Passing Interface)**.

Advantages and Disadvantages of Message Passing

Advantages:

- Very **powerful** and flexible.

- Used in almost all **supercomputers** (the most powerful computers in the world).

Disadvantages:

- Very **low-level** (programmer must handle lots of details).
- To parallelize a serial program, you often need to **rewrite most of it**.
- Data structures must either:
  - Be **copied** for each process, or
  - Be **split/distributed** across processes.
- Incremental rewriting (parallelizing small parts step by step) is usually **not practical**.
- Managing data movement between processes is **tedious and error-prone**. Because of this, message passing is often called **"the assembly language of parallel programming."** It's powerful but very detailed and hard to work with.

Key Points

In **distributed-memory systems**, each process has its **own private memory**.
Processes communicate by **sending and receiving messages**.
Example: Process 1 sends "Hello" to Process 0, and Process 0 prints it.
Message-passing programs are usually **SPMD** (same code, behavior depends on rank).
Send/Receive can block or not, depending on the API.
APIs also support collective operations like **broadcast** and **reduction**.
MPI is the most common message-passing API.
Message passing is powerful but **hard to use** (like assembly language).

One-Sided Communication

Two-Sided (Normal) Message Passing In **regular message-passing**, communication always needs **two processes**:

- One process **sends** data.
- Another process **receives** data.

Both must participate at the same time (a **send** must match a **receive**).

In **one-sided communication** (also called **remote memory access, RMA**):

- Only **one process** makes a function call.
- That single call either:

Reads data **from another process's memory** into its own memory, or Writes data **from its own memory** into another process's memory. The other process does **not explicitly call send/receive** at that moment.

Why use One-Sided Communication?
Simpler communication – only one process actively participates. **Faster communication** – less overhead (no need for matching send/receive). **Fewer function calls** – only one call instead of two.

But there are challenges, Imagine **Process 0 writes data into Process 1's memory**.
Problems:

1. **Safety problem** → Process 0 must know it's safe to overwrite memory in Process 1.
2. **Awareness problem** → Process 1 must know when its memory has been updated.

How to Solve These Problems?

- **Synchronization before writing** to Make sure Process 1 is ready before Process 0 writes.
- **Synchronization after writing** to Let Process 1 know data has arrived.

Ways to do this:

- **Second synchronization step** where Process 0 and 1 coordinate after the write.
- **Flag variable**
  - Process 0 sets a special variable (e.g., done = true) after copying.
  - Process 1 keeps checking (polling) the flag until it sees the update.

But **Polling wastes time** because Process 1 may repeatedly check the flag without doing real work. Since only one process does the communication call, the **other process doesn't "see" it happening directly**.

This can cause:

- **Hard-to-detect bugs**
- **Overwriting wrong memory**
- **Processes waiting forever** if synchronization isn't handled properly

Partitioned global address space languages:

- **shared-memory programming** is easier for many programmers (all processes just read/write from the same memory).
- But in **distributed-memory systems** (like clusters of computers or multi-core processors with private memory), memory is *not actually shared*. Each core or node has its **own local memory**.
- If we pretend the entire system has one "giant shared memory," performance can become terrible:
  - Accessing **local memory** is super fast
  - Accessing **remote memory** (another core's memory) is very slow 🐌 (100x–1000x slower).

So, if a program frequently accesses remote memory without control, it will run very inefficiently.

Solution: PGAS Languages

Partitioned Global Address Space (PGAS) languages try to give programmers the **ease of shared memory** but with **awareness of locality** (which data is local vs. remote).

- **Global Address Space**, All memory across processes looks like one shared memory (like in shared-memory programming).
- **Partitioned**, The memory is divided so that:
  - Some parts of the shared memory are **local** to each process/core.
  - Other parts may be **remote** (belonging to another process).

This way, a programmer knows which part of the array is in local memory and can design the program to minimize slow remote memory accesses.

```
shared int n = ...;
shared double x[n], y[n];
```

```
private int i, my_first_element, my_last_element;

my_first_element = ...;
my_last_element = ...;

/* Initialize x and y */
...

for (i = my_first_element; i <= my_last_element; i++)
   x[i] += y[i];
```

Here, arrays x and y are **shared** across all processes. Each process only works on its own part of the array (my_first_element to my_last_element). If the compiler/runtime ensures that each process's assigned portion of x and y is stored in its **local memory**, the code is **fast**. But if x is all on core 0 and y is all on core 1, then each access x[i] += y[i] would involve **remote memory access** is **very slow**.

Why PGAS Helps

- PGAS languages let the **programmer control** the distribution of shared data across processes.
- The programmer **knows where each piece of data lives**, so they can write efficient code that avoids unnecessary remote access.
- Private variables are always local and no performance problem.

Examples of PGAS Languages

Some well-known PGAS programming models and languages are:

- **UPC (Unified Parallel C)**
- **Coarray Fortran**
- **Chapel (from Cray)**
- **X10 (IBM project)**
- **Titanium (Java-based PGAS language)**

MODULE – 2

GPU PROGRAMMING, PROGRAMMING HYBRID SYSTEMS, MIMD SYSTEMS, GPUs, PERFORMANCE

GPU programming

GPUs are usually not "standalone" processors. They don't ordinarily run an operating system and system services, such as direct access to secondary storage. So, program ming a GPU also involves writing code for the CPU "host" system, which runs on an ordinary CPU. The memory for the CPU host and the GPU memory are usually separate. So, the code that runs on the host typically allocates and initializes storage on both the CPU and the GPU. It will start the program on the GPU, and it is responsible for the output of the results of the GPU program. Thus, GPU programming is really heterogeneous programming, since it involves programming two different types of processors.

The GPU itself will have one or more processors. Each of these processors is capable of running hundreds or thousands of threads. In the systems we'll be using, the processors share a large block of memory, but each individual processor has a small block of much faster memory that can only be accessed by threads running on that processor. These blocks of faster memory can be thought of as a programmer managed cache.

The threads running on a processor are typically divided into groups: the threads within a group use the SIMD model, and two threads in different groups can run independently. The threads in a SIMD group may not run in lockstep. That is, they may not all execute the same instruction at the same time. However, no thread in the group will execute the next instruction until all the threads in the group have completed executing the current instruction. If the threads in a group are executing a branch, it may be necessary to idle some of the threads. For example, suppose there are 32 threads in a SIMD group, and each thread has a private variable rank_in_gp that ranges from 0 to 31. Suppose also that the threads are executing the following code:

```
// Thread private variables
int rank_in_gp, my_x;
...
if (rank_in_gp < 16)
   my_x += 1;
else
   my_x -= 1;
```

Then the threads with rank < 16 will execute the first assignment, while the threads with rank ≥ 16areidle. After the threads with rank < 16 are done, the roles will be re versed: the threads with rank < 16 will be idle, while the threads with rank ≥ 16 will execute the second assignment. Of course, idling half the threads for two instructions isn't a very efficient use of the available resources. So, it's up to the programmer to minimize branching, where the threads within a SIMD group take different branches.

Programming hybrid systems

Before moving on, we should note that it is possible to program systems such as clusters of multicore processors using a combination of a shared-memory API on the nodes and a distributed-memory API for internode communication. However, this is usually only done for programs that require the highest possible levels of performance, since the complexity of this "hybrid" API makes program development much more difficult. See, for example, [45]. Rather, such systems are often programmed using a single, distributed-memory API for both inter- and intra-node communication.

INPUT AND OUTPUT

MIMD systems

We've generally avoided the issue of input and output. There are a couple of reasons. First and foremost, parallel I/O, in which multiple cores access multiple disks or other devices, is a subject to which one could easily devote a book. See, for example, [35]. Second, the vast majority of the programs we'll develop do very little in the way of I/O. The amount of data they read and write is quite small and easily managed by the standard C I/O functions printf, fprintf, scanf, and fscanf. However, even the limited use we make of these

functions can potentially cause some problems. Since these functions are part of standard C, which is a serial language, the standard says nothing about what happens when they're called by different processes. On the other hand, threads that are forked by a single process *do* share stdin, stdout, and stderr. However, (as we've seen), when multiple threads attempt to access one of these, the outcome is nondeterministic, and it's impossible to predict what will happen.

When we call printf from multiple processes, we, as developers, would like the output to appear on the console of a single system, the system on which we started the program. In fact, this is what the vast majority of systems do. However, there is no guarantee, and we need to be aware that it is possible for a system to do something else, for example, only one process has access to stdout or stderr or even *no* processes have access to stdout or stderr.

What *should* happen with calls to scanf when we're running multiple processes is a little less obvious. Should the input be divided among the processes? Or should only a single process be allowed to call scanf? The vast majority of systems allow at least one process to call scanf—usually process 0—while some allow more processes. Once again, there are some systems that don't allow any processes to call scanf.

When multiple processes *can* access stdout, stderr, or stdin, as you might guess, the distribution of the input and the sequence of the output are usually nondeterministic. For output, the data will probably appear in a different order each time the program is run, or, even worse, the output of one process may be broken up by the output of another process. For input, the data read by each process may be different on each run, even if the same input is used.

In order to partially address these issues, we'll be making these assumptions and following these rules when our parallel programs need to do I/O:

In distributed-memory programs, only process 0 will access stdin. In shared-memory

programs, only the master thread or thread 0 will access stdin.

. In both distributed-memory and shared-memory programs, all the processes/ threads can access stdout and stderr.

. However, because of the nondeterministic order of output to stdout, in most cases only a single process/thread will be used for all output to stdout. The principal exception will be output for debugging a program. In this situation, we'll often have multiple processes/threads writing to stdout.

. Only a single process/thread will attempt to access any single file other than stdin, stdout, or stderr. So, for example, each process/thread can open its own, private file for reading or writing, but no two processes/threads will open the same file.

. Debug output should always include the rank or id of the process/thread that's generating the output.

GPUs

In most cases, the host code in our GPU programs will carry out all I/O. Since we'll only be running one process/thread on the host, the standard C I/O functions should behave as they do in ordinary serial C programs.

The exception to the rule that we use the host for I/O is that when we are debug ging our GPU code, we'll want to be able to write to stdout and/or stderr. In the systems we use, each thread can write to stdout, and, as with MIMD programs, the order of the output is nondeterministic. Also, in the systems we use, no GPU thread has access to stderr, stdin, or secondary storage.

PERFORMANCE

Of course our main purpose in writing parallel programs is usually increased performance. So what can we expect? And how can we evaluate our programs?

In this section, we'll start by looking at the performance of homogeneous MIMD systems. So we'll assume that all of the cores have the same architecture. Since this is not the case for GPUs, we'll talk about the performance of GPUs in a separate subsection.

1. Speedup and efficiency in MIMD systems

Usually the best we can hope to do is to equally divide the work among the cores, while at the same time introducing no additional work for the cores. If we succeed in doing this, and we run our program with p cores, one thread or process on each core, then our parallel program will run p times faster than the serial program. If we call the serial run-time T_{serial} and our parallel run-time $T_{parallel}$, then the best we can hope for is $T_{parallel} = T_{serial}/p$. When this happens, we say that our parallel program has linear speedup.

In practice, we're unlikely to get linear speedup because the use of multiple processes/threads almost invariably introduces some overhead. For example, shared-memory programs will almost always have critical sections, which will require that we use some mutual exclusion mechanism such as a mutex. The calls to the mutex functions are overhead that's not present in the serial program, and the use of the mutex forces the parallel program to serialize execution of the critical section. Distributed-memory programs will almost always need to transmit data across the network, which is usually much slower than local memory access. Serial programs, on the other hand, won't have these overheads. Thus, it will be very unusual for us to find that our parallel programs get linear speedup. Furthermore, it's likely that the overheads will increase as we increase the number of processes or threads, that is, more threads will probably mean more threads need to access a critical section. More processes will probably mean more data needs to be transmitted across the network.

So if we define the speedup of a parallel program to be

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}},$$

then linear speedup has $S = p$, which is unusual. Furthermore, as p increases, we expect S to become a smaller and smaller fraction of the ideal, linear speedup p. Another way of saying this is that $S=p$ will probably get smaller and smaller as p increases. Table 2.4 shows an example of the changes in S and S/p as p increases.

This value, S/p, is sometimes called the efficiency of the parallel program. If we substitute the formula for S, we see that the efficiency is

$$E = \frac{S}{p} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}.$$

Table 2.4 Speedups and Efficiencies of a Parallel Program

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
E = S/p	1.0	0.95	0.90	0.81	0.68

Table 2.5 Speedups and Efficiencies of a Parallel Program on Different Problem Sizes

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

It's clear that T_{parallel}, S, and E depend on p, the number of processes or threads. We also need to keep in mind that T_{parallel}, S, E, and T_{serial} all depend on the problem size. For example, if we halve and double the problem size of the program whose speedups are shown in Table 2.4, we get the speedups and efficiencies shown in Table 2.5. The speedups are plotted in Figure 2.18, and the efficiencies are plotted in Figure 2.19.

We see that in this example, when we increase the problem size, the speedups and the efficiencies increase, while they decrease when we decrease the problem size. This behavior is

quite common. Many parallel programs are developed by dividing the work of the serial program among the processes/threads and adding in the nec-essary "parallel overhead" such as mutual exclusion or communication. Therefore, if T_{overhead} denotes this parallel overhead, it's often the case that

$$T_{\text{parallel}} = T_{\text{serial}}/p + T_{\text{overhead}}.$$

Furthermore, as the problem size is increased, T_{overhead} often grows more slowly than T_{serial}. When this is the case the speedup and the efficiency will increase. See Exercise 2.16. This is

what your intuition should tell you: there's more work for the processes/threads to do, so the relative amount of time spent coordinating the work of the processes/threads should be less.

A final issue to consider is what values of T_{serial} should be used when report-ing speedups and efficiencies. Some authors say that T_{serial} should be the run-time of the fastest program on the fastest processor available. In practice, most authors use a serial program on which the parallel program was based and run it on a single processor of the parallel system. So if we were studying the performance of a par-allel shell sort program, authors in the first group might use a serial radix sort or quicksort on a single core of the fastest system available, while authors in the second group would use a serial shell sort on a single processor of the parallel system. We'll generally use the second approach.

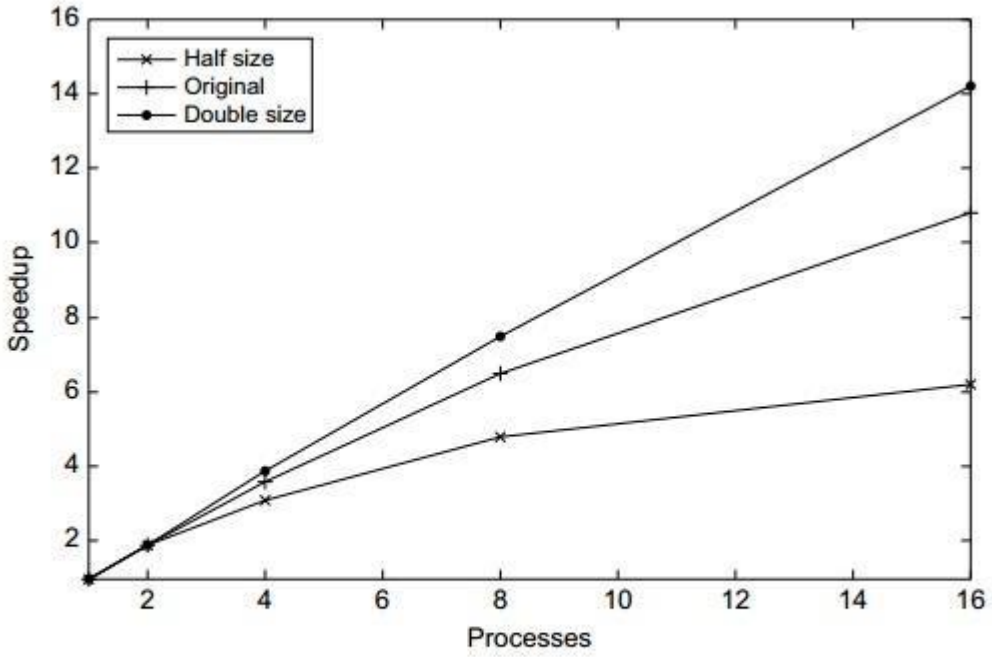


FIGURE 2.18

Speedups of parallel program on different problem sizes

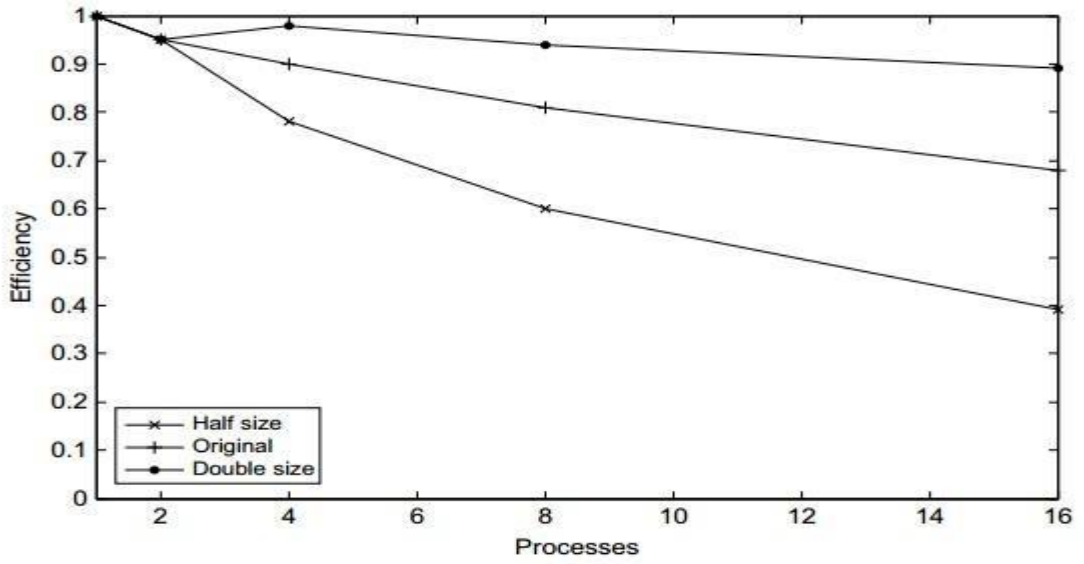


FIGURE 2.19

Efficiencies of parallel program on different problem sizes

2. Amdahl's law

Back in the 1960s, Gene Amdahl made an observation [2] that's become known as Amdahl's law. It says, roughly, that unless virtually all of a serial program is paral-lelized, the possible speedup is going to be very limited—regardless of the number of cores available. Suppose, for example, that we're able to parallelize 90% of a serial program. Further suppose that the parallelization is "perfect," that is, regardless of the number of cores p we use, the speedup of this part of the program will be p. If the serial run-time is T_{serial}= 20 seconds, then the run-time of the parallelized part will be 0.9 xT_{serial}/p = 18=p and the run-time of the "unparallelized" part will be 0.1 x T_{serial} = 2. The overall parallel run-time will be

$$T_{\text{parallel}} = 0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}} = 18/p + 2,$$

and the speedup will be

$$S = \frac{T_{\text{serial}}}{0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}}} = \frac{20}{18/p + 2}.$$

Now as p gets larger and larger, $0.9\ T_{serial}/p = 18/p$ gets closer and closer to 0, so the total parallel run-time can't be smaller than $0.1\ T_{serial} = 2$. That is, the denominator in S can't be

$$S \leq \frac{T_{serial}}{0.1 \times T_{serial}} = \frac{20}{2} = 10.$$

smaller than $0.1\ T_{serial} = 2$. The fraction S must therefore be smaller than

That is, $S <= 10$. This is saying that even though we've done a perfect job in parallelizing 90% of the program, and even if we have, say, 1000 cores, we'll never get a speedup better than 10.

More generally, if a fraction r of our serial program remains unparallelized, then Amdahl's law says we can't get a speedup better than $1/r$. In our example, r = 1- 0.9 = 1/10, so we couldn't get a speedup better than 10. Therefore, if a fraction r of our serial program is "inherently serial," that is, cannot possibly be parallelized, then we can't possibly get a speedup better than $1/r$. Thus, even if r is quite small— say 1/100—and we have a system with thousands of cores, we can't possibly get a speedup better than 100.

This is pretty daunting. Should we give up and go home? Well, no. There are several reasons not to be too worried by Amdahl's law. First, it doesn't take into con-sideration the problem size. For many problems, as we increase the problem size, the "inherently serial" fraction of the program decreases in size; a more mathematical version of this statement is known as Gustafson's law [25]. Second, there are thousands of programs used by scientists and engineers that routinely obtain huge speedups on large distributed-memory systems. Finally, is a small speedup so awful? In many cases, obtaining a speedup of 5 or 10 is more than adequate, especially if the effort involved in developing the parallel program wasn't very large.

3. Scalability in MIMD systems

The word "scalable" has a wide variety of informal uses. Indeed, we've used it several times already. Roughly speaking, a technology is scalable if it can handle ever-increasing problem sizes. However, in discussions of parallel program performance, scalability has a somewhat more formal definition. Suppose we run a parallel program with a fixed number of processes/threads and a fixed input size, and we obtain an efficiency E. Suppose we now increase the number of processes/threads that are used by the program. If we can find a corresponding rate of increase in the problem size so that the program always has efficiency E, then the program is scalable. As an example, suppose that $T_{serial} = n$, where the units of T_{serial} are in microseconds, and n is also the problem size. Also suppose that $T_{parallel} = n/p + 1$. Then

$$E = \frac{n}{p(n/p+1)} = \frac{n}{n+p}.$$

To see if the program is scalable, we increase the number of processes/threads by a factor of k, and we want to find the factor x that we need to increase the problem size by so that E is unchanged. The number of processes/threads will be kp and the problem size will be xn, and we want to solve the following equation for x:

$$E = \frac{n}{n+p} = \frac{xn}{xn+kp}.$$

Well, if $x = k$, there will be a common factor of k in the denominator $xn + kp = kn + kp = k(n + p)$, and we can reduce the fraction to get

$$\frac{xn}{xn+kp} = \frac{kn}{k(n+p)} = \frac{n}{n+p}.$$

In other words, if we increase the problem size at the same rate that we increase the number of processes/threads, then the efficiency will be unchanged, and our program is scalable.

There are a couple of cases that have special names. If when we increase the number of processes/threads, we can keep the efficiency fixed *without* increasing the problem size, the program is said to be *strongly scalable.* If we can keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, then the program is said to be *weakly scalable.* The program in our example would be weakly scalable.

4. Taking timings of MIMD programs

You may have been wondering how we find T_{serial} and $T_{parallel}$. There are a *lot* of different approaches, and with parallel programs the details may depend on the API. However, there are a few general observations we can make that may make things a little easier.

The first thing to note is that there are at least two different reasons for taking tim-ings. During program development we may take timings in order to determine if the program is behaving as we intend. For example, in a distributed-memory program we might be interested in finding out how much time the processes are spending waiting for messages, because if this value is large, there is almost certainly something wrong either with our design or our implementation. On the other hand, once we've completed development of the program, we're often interested in determining how good its performance is. Perhaps surprisingly, the way we take these two timings is usually different. For the first timing, we usually need very detailed information: How much time did the program spend in this part of the program? How much time did it spend in that part? For the second, we usually report a single value. Right now we'll talk about the second type of timing. See Exercise 2.22 for a brief discussion of some issues in taking the first type of timing.

Second, we're usually *not* interested in the time that elapses between the pro-gram's start and the program's finish. We're usually interested only in some part of the program. For example, if we write a program that implements bubble sort, we're probably only interested in the time it takes to sort the keys, not the time it takes to read them in and print them out. We probably can't use something like the Unix shell command time, which reports the time taken to run a program from start to finish.

Third, we're usually *not* interested in "CPU time." This is the time reported by the standard C function clock. It's the total time the program spends in code executed as part of the program. It would include the time for code we've written; it would include the time we spend in library functions such as pow or sin; and it would include the time the operating system spends in functions we call, such as printf and scanf. It would not include time the program was idle, and this could be a problem. For example, in a distributed-memory program, a process that calls a receive function may have to wait for the sending process to execute the matching send, and the operating system might put the receiving process to sleep while it waits. This idle time wouldn't be counted as CPU time, since no function that's been called by the process is active. However, it should count in our evaluation of the overall run-time, since it may be a real cost in our program. If each time the program is run, the process has to wait, ignoring the time it spends waiting would give a misleading picture of the actual run-time of the program.

Thus, when you see an article reporting the run-time of a parallel program, the reported time is usually "wall clock" time. That is, the authors of the article report the time that has elapsed between the start and finish of execution of the code that the user is interested in. If the user could see the execution of the program, she would hit the start button on her stopwatch when it begins execution and hit the stop button when it stops execution. Of course, she can't see her code executing, but she can modify the source code so that it looks something like this:

```
double start, finish;
. . .
start = Get_current_time();

/*      Code that we want to time      */

. . .
finish = Get_current time();
```

```
printf("The elapsed time = %e seconds\n", finish-start);
```

The function Get_current_time() is a hypothetical function that's supposed to at's supposed to return the number of seconds that have elapsed since some fixed time in the past. It's just a placeholder. The actual function that is used will depend on the API. For example, MPI has a function MPI Wtime that could be used here, and the OpenMP API for shared-memory programming has a function omp get wtime. Both functions return wall clock time instead of CPU time.

There may be an issue with the resolution of the timer function. The resolution is the unit of measurement on the timer. It's the duration of the shortest event that can have a nonzero time. Some timer functions have resolutions in milliseconds (10^{-3} seconds), and when instructions can take times that are less than a nanosecond (10^{-9} seconds), a program may have to execute millions of instructions before the timer reports a nonzero time. Many APIs provide a function that reports the resolution of the timer. Other APIs specify that a timer must have a given resolution. In either case we, as the programmers, need to check these values.

When we're timing parallel programs, we need to be a little more careful about how the timings are taken. In our example, the code that we want to time is probably being executed by multiple processes or threads and our original timing will result in the output of *p* elapsed times.

```
private double start, finish;
. . .

start = Get_current_time();

/*      Code that we want to time      */

. . .
finish = Get_current_time();

printf("The elapsed time = %e seconds\n", finish-start);
```

However, what we're usually interested in is a single time: the time that has elapsed from when the first process/thread began execution of the code to the time the last process/thread finished execution of the code. We often can't obtain this exactly, since there may not be any correspondence between the clock on one node and the clock on another node. We usually settle for a compromise that looks something like this:

```
shared double global_elapsed;

private double my_start, my_finish, my_elapsed;
/*      Synchronize all processes/threads     */

Barrier();
my_start = Get_current_time();

/*      Code that we want to time      */

. . .

my_finish = Get_current_time();
my_elapsed = my_finish -  my_start;

/* Find the max across all processes/threads */
```

```
global_elapsed = Global max(my_elapsed);
if (my_rank == 0)
printf("The elapsed time = %e seconds\n", global elapsed);
shared double global elapsed;
```

Here, we first execute a barrier function that approximately synchronizes all of the processes/threads. We would like for all the processes/threads to return from the call simultaneously, but such a function usually can only guarantee that all the process-es/threads have started the call when the first process/thread returns. We then execute the code as before and each process/thread finds the time it took. Then all the process-es/threads call a global maximum function, which returns the largest of the elapsed times, and process/thread 0 prints it out.

We also need to be aware of the *variability* in timings. When we run a program several times, it's extremely likely that the elapsed time will be different for each run. This will be true even if each time we run the program we use the same input and the same systems. It might seem that the best way to deal with this would be to report either a mean or a median run-time. However, it's unlikely that some outside event could actually make our program run faster than its best possible run-time. So instead of reporting the mean or median time, we usually report the *minimum* time.

Running more than one thread per core can cause dramatic increases in the variability of timings. More importantly, if we run more than one thread per core, the system will have to take extra time to schedule and de-schedule cores, and this will add to the overall run-time. Therefore, we rarely run more than one thread per core.

Finally, as a practical matter, since our programs won't be designed for high-performance I/O, we'll usually not include I/O in our reported run-times.

5. GPU performance

We can compare the performance of a GPU program to the performance of a serial program, and it's quite common to see reported speedups of GPU programs over serial programs or parallel MIMD programs.

Since the cores on the GPU are fundamentally different from conventional CPUs, it doesn't make sense to talk about linear speedup of a GPU program relative to a serial CPU program.

Since efficiency of a GPU program relative to a CPU program doesn't make sense, the formal definition of the scalability of a MIMD program can't be applied to a GPU program. However, the informal usage of scalability is routinely applied to GPUs: a GPU program is scalable if we can increase the size of the GPU and obtain speedups over the performance of the program on a smaller GPU.

If we run the inherently serial part of a GPU program on a conventional, serial processor, then Amdahl's law can be applied to GPU programs, and the resulting upper bound on the possible speedup will be the same as the upper bound on the possible speedup for a MIMD program. That is, if a fraction r of the original serial program isn't parallelized, and this fraction is run on a conventional serial processor, then the best possible speedup of the program running on the GPU and the serial processor will be less than 1/r.

It should be noted that the same caveats that apply to Amdahl's law on MIMD systems also apply to Amdahl's law on GPUs: It's likely that the "inherently serial" fraction will depend on the problem size, and if it gets smaller as the problem size increases, the bound on the best possible speedup will increase. Also, many GPU programs obtain huge speedups, and, finally, a relatively small speedup may be perfectly adequate.

The same basic ideas about timing that we discussed for MIMD programs also apply to GPU programs. However, since a GPU program is ordinarily started and finished on a conventional CPU, as long as we're interested in the performance of the entirety of the program running on the GPU, we can usually just use the timer on the CPU, starting it before the GPU part(s) of the program are started, and stopping it after the GPU part(s) are done. There are more complicated scenarios—e.g., running a program on multiple CPU-GPU pairs—that require more care, but we won't be dealing with these types of programs. If we only want to time a subset of the code running on the GPU, we'll need to use a timer defined by the API for the GPU.

MODULE 5

CHAPTER 6: GPU programming with CUDA

6.1 GPUs and GPGPU

In the late 1990s and early 2000s, the computer industry responded to the demand for highly realistic computer video games and video animations by developing extremely powerful **graphics processing units** or **GPUs**. These processors, as their name suggests, are designed to improve the performance of programs that need to render many detailed images.

The existence of this computational power was a temptation to programmers who didn't specialize in computer graphics, and by the early 2000s they were trying to ap- ply the power of GPUs to solving general computational problems, problems such as searching and sorting, rather than graphics. This became known as **General Purpose computing on GPUs** or **GPGPU**.

One of the biggest difficulties faced by the early developers of GPGPU was that the GPUs of the time could only be programmed using computer graphics APIs, such as Direct3D and OpenGL. So programmers needed to reformulate algorithms for general computational problems so that they used graphics concepts, such as vertices, triangles, and pixels. This added considerable complexity to the development of early GPGPU programs, and it wasn't long before several groups started work on developing languages and compilers that allowed programmers to implement general algorithms for GPUs in APIs that more closely resembled conventional, high-level languages for CPUs.

These efforts led to the development of several APIs for general purpose programming on GPUs. Currently the most widely used APIs are CUDA and OpenCL. CUDA was developed for use on Nvidia GPUs. OpenCL, on the other hand, was de- signed to be highly portable. It was designed for use on arbitrary GPUs *and* other processors—processors such as field programmable gate arrays (FPGAs) and digital signal processors (DSPs). To ensure this portability, an OpenCL program must include a good deal of code providing information about which systems it can be run on and information about how it should be run. Since CUDA was developed to run only on Nvidia GPUs, it requires relatively modest setup, and, as a consequence, we'll use it instead of OpenCL.

6.2 GPU architectures

As we've seen (see Chapter 2), CPU architectures can be extremely complex. How- ever, we often think of a conventional CPU as a SISD device in Flynn's Taxonomy (see Section 2.3): the processor fetches an instruction from memory and executes the instruction on a small number of data items. The instruction is an element of the *Single Instruction stream*—the "SI" in SISD—and the data items are elements of the *Single Data stream*—the "SD" in SISD. GPUs, however, are composed of SIMD or *Single Instruction stream, Multiple Data stream* processors. So, to understand how to program them, we need to first look at their architecture.

Recall (from Section 2.3) that we can think of a SIMD processor as being com- posed of a single control unit and multiple datapaths. The control unit fetches an instruction from memory and broadcasts it to the datapaths. Each datapath either executes the instruction on *its* data or is idle.

For example, suppose there are *n* datapaths that share an *n*-element array x. Also suppose that the *i*th datapath will work with x[i]. Now suppose we want to add 1 to the nonnegative elements of x and subtract 2 from the negative elements of x. We might implement this with the following code:

```
/ Datapath i executes the following code /
if ( x [ i ] >= 0 )
x [ i ] += 1 ;
else
x [ i ] −= 2 ;
```

In a typical SIMD system, each datapath carries out the test x[i] >= 0. Then the datapaths for which the test is true execute x[i] += 1, while those for which x[i] < 0 are *idle*. Then the roles of the datapaths are reversed: those for which x[i] >= 0 are idle while the other datapaths execute x[i] = 2. See Table 6.1.

Table 6.1 Execution of branch on a SIMD system.

Time	Datapaths with x[i] >= 0	Datapaths with x[i] < 0
1	Test x[i] >= 0	Test x[i] >= 0
2	x[i] += 1	Idle
3	Idle	x[i] -= 2

A typical GPU can be thought of as being composed of one or more SIMD processors. Nvidia GPUs are composed of **Streaming Multiprocessors** or **SMs**.[1] One SM can have several control units and many more datapaths. So an SM can be thought of as consisting of one or more SIMD processors. The SMs, however, operate asynchronously: there is no penalty if one branch of an **if −else** executes on one SM, and

the other executes on another SM. So in our preceding example, if all the threads with x[i] >= 0 were executing on one SM, and all the threads with x[i] < 0 were executing on another, the execution of our **if else** example would require only two stages. (See Table 6.2.)

Table 6.2 Execution of branch on a system with multiple SMs.

Time	Datapaths with x[i] >= 0 (on SM A)	Datapaths with x[i] < 0 (on SM B)
1	Test x[i] >= 0	Test x[i] >= 0
2	x[i] += 1	x[i] -= 2

In Nvidia parlance, the datapaths are called cores, **Streaming Processors**, or **SPs**. Currently,[2] one of the most powerful Nvidia processor has 82 SMs, and each SM has 128 SPs for a total of 10,496 SPs. Since we use the term “core” to mean something else when we’re discussing MIMD architectures, we’ll use SP to denote an Nvidia datapath. Also note that Nvidia uses the term **SIMT** instead of SIMD. SIMT stands for Single Instruction Multiple Thread, and the term is used because threads on an SM that are executing the same instruction may not execute simultaneously: to hide memory access latency, some threads may block while memory is accessed and other threads, that have already accessed the data, may proceed with execution.

Each SM has a relatively small block of memory that is shared among its SPs. As we’ll see, this memory can be accessed very quickly by the SPs. All of the SMs on a single chip also have access to a much larger block of memory that is shared among all the SPs. Accessing this memory is relatively slow. (See Fig. 6.1.)

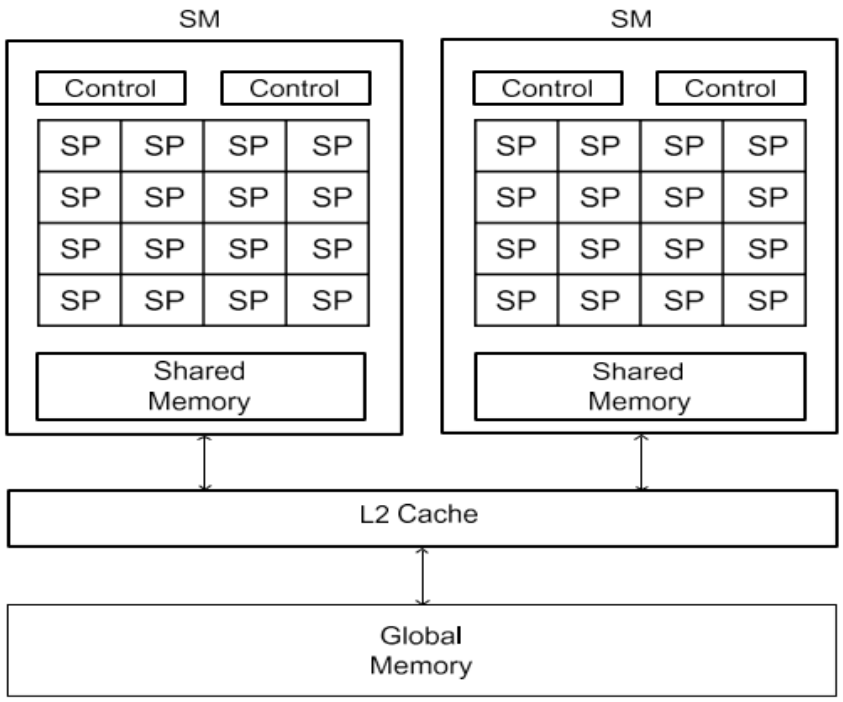


FIGURE 6.1
Simplified block diagram of a GPU.

The GPU and its associated memory are usually physically separate from the CPU and its associated memory. In Nvidia documentation, the CPU together with its associated memory is often called the **host**, and the GPU together with its memory is called the **device**. In earlier systems the physical separation of host and device memories required that data was usually explicitly transferred between CPU memory and GPU memory. That is, a function was called that would transfer a block of data from host memory to device memory or vice versa. So, for example, data read from a file by the CPU or output data generated by the GPU would have to be transferred between the host and device with an explicit function call. However, in more recent Nvidia systems (those with compute capability 3.0), the explicit transfers in the source code aren't needed for correctness, although they may be able to improve overall performance. (See Fig. 6.2.)

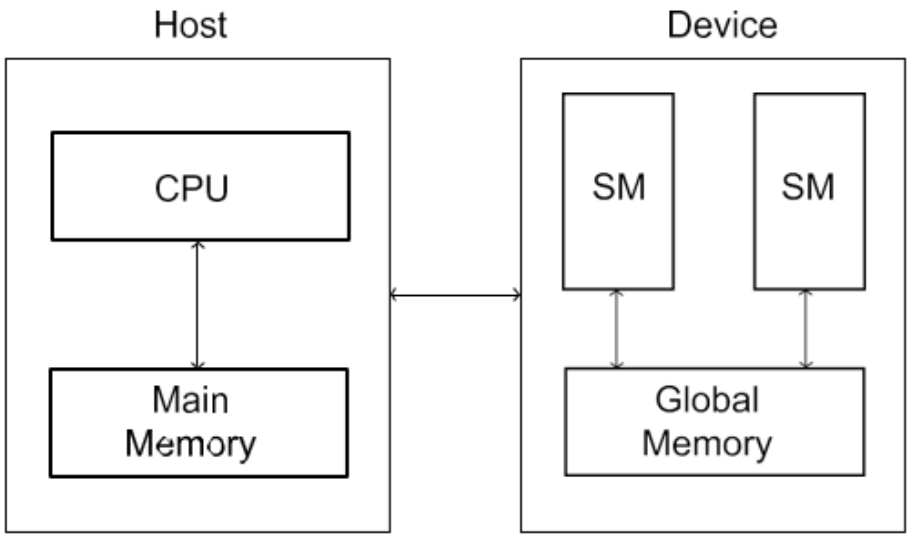


FIGURE 6.2
Simplified block diagram of a CPU and a GPU.

6.3 Heterogeneous computing

Up to now we've implicitly assumed that our parallel programs will be run on systems in which the individual processors have identical architectures. Writing a program that runs on a GPU is an example of **heterogeneous** computing. The reason is that the programs make use of both a host processor—a conventional CPU—and a de- vice processor—a GPU—and, as we've just seen, the two processors have different architectures.

We'll still write a single program (using the SPMD approach—see Section 2.4.1), but now there will be functions that we write for conventional CPUs and functions that we write for GPUs. So, effectively, we'll be writing two programs.

Heterogeneous computing has become much more important in recent years. Re- call from Chapter 1 that from about 1986 to 2003, the single-thread performance of conventional CPUs was increasing, on average, more than 50% per year, but since 2003, the improvement in single-thread performance has decreased to the point that from 2015 to 2017, it has been growing at less than 4% per year [28]. So programmers are leaving no stone unturned in their search for ways to bolster performance, and one possibility is to make use of other types of processors, processors other than CPUs. Our focus is GPUs, but other possibilities include **Field Programmable Gate Arrays** or **FPGAs**, and **Digital Signal Processors** or **DSPs**. FPGAs contain programmable logic blocks and interconnects that can be configured prior to program execution. DSPs contain special circuitry for manipulating (e.g., compressing, filter- ing) signals, especially "real-world" analog signals.

6.4 CUDA hello

So let's start talking about the CUDA API, the API we'll be using to program heterogeneous CPU–GPU systems.

CUDA is a software platform that can be used to write GPGPU programs for heterogeneous systems equipped with an Nvidia GPU. CUDA was originally an acronym for "Compute Unified Device Architecture," which was meant to suggest that it provided a single interface for programming both CPU and GPU. More recently, however, Nvidia has decided that CUDA is not an acronym; it's simply the name of an API for GPGPU programming.

There is a language-specific CUDA API for several languages; for example, there are CUDA APIs for C, C++, Fortran, Python, and Java. We'll be using CUDA C, but we need to be aware that sometimes we'll need to use some C++ constructs. This is because the CUDA C compiler can compile both C and C++ programs, and it can do this because it is a modified C++ compiler. So where the specifications for C and C++ differ the CUDA compiler sometimes uses C++. For example, since the C library function malloc returns a **void*** pointer, a C program doesn't need a cast in the

instruction

float *x = malloc (n * **sizeof** (**float**));

However, in C++ a cast is required

float *x = (**float** *) malloc (n * **sizeof** (**float**));

As usual, we'll begin by implementing a version of the "hello, world" program. We'll write a CUDA C program in which each CUDA thread prints a greeting.[3] Since the program is heterogeneous, we will, effectively, be writing two programs: a host or CPU program and a device or GPU program.

Note that even though our programs are written in CUDA C, CUDA programs cannot be compiled with an ordinary C compiler. So unlike MPI and Pthreads, CUDA is not just a library that can be linked in to an ordinary C program: CUDA requires a special compiler. For example, an ordinary C compiler (such as gcc) generates a ma- chine language executable for a single CPU (e.g., an x86 processor), but the CUDA compiler must generate machine language for two different processors: the host processor and the device processor.

6.4.1 The source code

The source code for a CUDA program that prints a greeting from each thread on the GPU is shown in Program 6.1.

```
1  #include <stdio.h>
2  #include <cuda.h>    /* Header file for CUDA */
3
4  /* Device code:  runs on GPU */
5  __global__ void Hello(void) {
6
7      printf("Hello from thread %d!\n", threadIdx.x);
8  }  /* Hello */
9
10
11 /* Host code:  Runs on CPU */
12 int main(int argc, char* argv[]) {
13     int thread_count;          /* Number of threads to run on GPU */
14
15     thread_count = strtol(argv[1], NULL, 10);
16                         /* Get thread_count from command line */
17
18     Hello <<<1, thread_count>>>();
19                         /* Start thread_count threads on GPU, */
20
21     cudaDeviceSynchronize();              /* Wait for GPU to finish */
22
23     return 0;
24 }  /* main */
```

Program 6.1: CUDA program that prints greetings from the threads.

As you might guess, there's a header file for CUDA programs, which we include in Line 2.

The Hello function follows the include directives and starts on Line 5. This function is run by each thread on the GPU. In CUDA parlance, it's called a **kernel**, a function that is started by the host but runs on the device. CUDA kernels are identified by the keyword _global_, and they always have return type **void**.

The main function follows the kernel on Line 12. Like ordinary C programs, CUDA C programs start execution in main, and the main function runs on the *host*. The function first gets the number of threads from the command line. It then starts the re- quired number of copies of the kernel on Line 18. The call to cudaDeviceSynchronize will cause the main program to wait until all the threads have finished executing the kernel, and when this happens, the program terminates as usual with **return** 0.

6.4.2 Compiling and running the program

A CUDA program file that contains both host code and device code should be stored in a file with a ".cu" suffix. For example, our hello program is in a file called cuda_hello.cu. We can compile it using the CUDA compiler nvcc. The command should look something like this[4]:

$ nvcc -o cuda_hello cuda_hello.cu

If we want to run one thread on the GPU, we can type

$ ./cuda_hello 1

and the output will be

```
Hello from thread 0!
```

If we want to run ten threads on the GPU, we can type

```
$ ./cuda_hello 10
```

and the output of will be

```
Hello from thread 0! Hello from thread 1! Hello from thread 2! Hello from thread 3! Hello from thread 4! Hello from thread 5! Hello from thread 6! Hello from thread 7! Hello from thread 8! Hello from thread 9!
```

6.5 A closer look

So what exactly happens when we run cuda_hello? Let's take a closer look.

As we noted earlier, execution begins on the host in the main function. It gets the number of threads from the command line by calling the C library strtol function.

Things get interesting in the call to the kernel in Line 18. Here we tell the system how many threads to start on the GPU by enclosing the pair

1, thread_count

in triple angle brackets. If there were any arguments to the Hello function, we would enclose them in the following parentheses.

The kernel specifies the code that each thread will execute. So each of our threads will print a message

" Hello from thread %d\n"

The decimal int format specifier (%d) refers to the variable threadIdx.x. The struct threadIdx is one of several variables defined by CUDA when a kernel is started. In our example, the field x gives the relative index or rank of the thread that is executing. So we use it to print a message containing the thread's rank.

After a thread has printed its message, it terminates execution.

Notice that our kernel code uses the Single-Program Multiple-Data or SPMD paradigm: each thread runs a copy of the same code on its own data. In this case, the only thread-specific data is the thread rank stored in threadIdx.x.

One very important difference between the execution of an ordinary C function and a CUDA kernel is that kernel execution is **asynchronous**. This means that the call to the kernel on the host *returns* as soon as the host has notified the system that it should start running the kernel, and even though the call in main has re- turned, the threads executing the kernel may not have finished executing. The call to cudaDeviceSynchronize in Line 21 forces the main function to wait until all the threads executing the kernel have completed. If we omitted the call to cudaDeviceSynchronize, our program could terminate before the threads produced any output, and it might ap- pear that the kernel was never called.

When the host returns from the call to cudaDeviceSynchronize, the main function then terminates as usual with **return** 0.

To summarize, then:

- Execution begins in main, which is running on the host.
- The number of threads is taken from the command line.
- The call to Hello starts the kernel.
  - The <<<1, thread_count>>> in the call specifies that thread_count copies of the kernel should be started on the device.
  - When the kernel is started, the struct threadIdx is initialized by the system, and in our example the field threadIdx.x contains the thread's index or rank.
  - Each thread prints its message and terminates.
- The call to cudaDeviceSynchronize in main forces the host to wait until all of the threads have completed kernel execution before continuing and terminating.

6.6 Threads, blocks, and grids

You're probably wondering why we put a "1" in the angle brackets in our call to Hello:

Hello <<<1, thread_count > > >();

Recall that an Nvidia GPU consists of a collection of streaming multiprocessors (SMs), and each streaming multiprocessor consists of a collection of streaming processors (SPs). When a CUDA kernel runs, each individual thread will execute its code on an SP. With "1" as the first value in angle brackets, all of the threads that are started by the kernel call will run on a single SM. If our GPU has two SMs, we can try to use both of them with the kernel call

Hello <<<2, thread_count / 2 >> >();

If thread_count is even, this kernel call will start a total of thread_count threads, and the threads will be divided between the two SMs: thread_count/2 threads will run on each SM. (What happens if thread_count is odd?)

CUDA organizes threads into blocks and grids. A **thread block** (or just a **block** if the context makes it clear) is a collection of threads that run on a single SM. In a kernel call the first value in the angle brackets specifies the number of thread blocks. The second value is the number of threads in each thread block. So when we started the kernel with

Hello <<<1, thread_count > > >();

we were using one thread block, which consisted of thread_count threads, and, as a consequence, we only used one SM.

We can modify our greetings program so that it uses a user-specified number of blocks, each consisting of a user-specified number of threads. (See Program 6.2.)

```
1  #include <stdio.h>
2  #include <cuda.h>    /* Header file for CUDA */
3
4  /* Device code:  runs on GPU */
5  __global__ void Hello(void) {
6
7      printf("Hello from thread %d in block %d\n",
8             threadIdx.x, blockIdx.x);
9  }  /* Hello */
10
11
12 /* Host code:  Runs on CPU */
13 int main(int argc, char* argv[]) {
14     int blk_ct;                    /* Number of thread blocks */
15     int th_per_blk;          /* Number of threads in each block */
16
17     blk_ct = strtol(argv[1], NULL, 10);
18                     /* Get number of blocks from command line */
19     th_per_blk = strtol(argv[2], NULL, 10);
20          /* Get number of threads per block from command line */
21
22     Hello <<<blk_ct, th_per_blk>>>();
23             /* Start blk_ct*th_per_blk threads on GPU, */
24
25      cudaDeviceSynchronize();              /* Wait for GPU to finish */
26
27     return 0;
28 }  /* main */
```

Program 6.2: CUDA program that prints greetings from threads in multiple blocks.

In this program we get both the number of thread blocks and the number of threads in each block from the command line. Now the kernel call starts blk_ct thread blocks, each of which contains th_per_blk threads.

When the kernel is started, each block is assigned to an SM, and the threads in the block are then run on that SM. The output is similar to the output from the original program, except that now we're using two system-defined variables: threadIdx.x and blockIdx.x. As you've probably guessed, threadIdx.x gives a thread's rank or index in its block, and blockIdx.x gives a block's rank in the *grid*.

A **grid** is just the collection of thread blocks started by a kernel. So a thread block is composed of threads, and a grid is composed of thread blocks.

There are several built-in variables that a thread can use to get information on the grid started by the kernel. The following four variables are structs that are initialized in each thread's memory when a kernel begins execution:

- threadIdx: the rank or index of the thread in its thread block.
- blockDim: the dimensions, shape, or size of the thread blocks.
- blockIdx: the rank or index of the block within the grid.
- gridDim: the dimensions, shape, or size of the grid.

All of these structs have three fields, x, y, and z,5 and the fields all have unsigned integer types. The fields are often convenient for applications. For example, an application that uses graphics may find it convenient to assign a thread to a point in two- or three-dimensional space, and the fields in threadIdx can be used to indicate the point's position. An application that makes extensive use of matrices may find it convenient to assign a thread to an element of a matrix, and the fields in threadIdx can be used to indicate the column and row of the element. When we call a kernel with something like

```
int  blk_ct ,  th_per_blk ;
. . .
Hello <<<blk_ct ,  th_per_blk > > ( ) ;
```

the three-element structures gridDim and blockDim are initialized by assigning the values in angle brackets to the x fields. So, effectively, the following assignments are made:

```
gridDim . x = blk_ct ;
blockDim . x = th_per_blk ;
```

The y and z fields are initialized to 1. If we want to use values other than 1 for the y and z fields, we should declare two variables of type dim3, and pass them into the call to the kernel. For example,

```
dim3 grid_dims ,  block_dims ;
grid_dims . x = 2 ; grid_dims . y = 3 ; grid_dims . z = 1 ; block_dims . x = 4 ; block_dims . y = 4 ; block_dims . z = 4 ;
. . .
```

Kernel <<<grid_dims , block_dims >>> (. . .) ;

This should start a grid with 23 1 6 blocks, each of which has 4^3 64 threads. Note that all the blocks must have the same dimensions. More importantly, CUDA requires that thread blocks be independent. So one thread block must be able to com- plete its execution, regardless of the states of the other thread blocks: the thread blocks can be executed sequentially in any order, or they can be executed in par- allel. This ensures that the GPU can schedule a block to execute solely on the basis

of the state of that block: it doesn't need to check on the state of any other block.[6]

6.7 Nvidia compute capabilities and device architectures[7]

There are limits on the number of threads and the number of blocks. The limits depend on what Nvidia calls the **compute capability** of the GPU. The compute capability is a number having the form *a.b*. Currently the *a*-value or major revision number can be 1, 2, 3, 5, 6, 7, 8. (There is no major revision number 4.) The possible *b*-values or minor revision numbers depend on the major revision value, but currently they fall in the range 0–7. CUDA no longer supports devices with compute capability

< 3.

For devices with compute capability > 1, the maximum number of threads per block is 1024. For devices with compute capability 2.*b*, the maximum number of threads that can be assigned to a single SM is 1536, and for devices with compute capability > 2, the maximum is currently 2048. There are also limits on the sizes of the dimensions in both blocks and grids. For example, for compute capability > 1, the maximum x- or y-dimension is 1024, and the maximum z-dimension is 64. For further information, see the appendix on compute capabilities in the CUDA C++ Programming Guide [11].

Nvidia also has names for the microarchitectures of its GPUs. Table 6.3 shows the current list of architectures and some of their corresponding compute capabilities. Somewhat confusingly, Nvidia also uses Tesla as the name for their products targeting GPGPU.

Table 6.3 GPU architectures and compute capabilities.

Name	Ampere	Tesla	Fermi	Kepler	Maxwell	Pascal	Volta	Turing
Compute capability	8.0	1.*b*	2.*b*	3.*b*	5.*b*	6.*b*	7.0	7.5

We should note that Nvidia has a number of "product families" that can consist of anything from an Nvidia-based graphics card to a "system on a chip," which has the main hardware components of a system, such as a mobile phone in a single integrated circuit.

Finally, note that there are a number of versions of the CUDA API, and they do

not correspond to the compute capabilities of the different GPUs.

6.8 Vector addition

GPUs and CUDA are designed to be especially effective when they run data-parallel programs. So let's write a very simple, data-parallel CUDA program that's embarrass- ingly parallel: a program that adds two vectors or arrays. We'll define three *n*-element arrays, x, y, and z. We'll initialize x and y on the host. Then a kernel can start at least *n* threads, and the *i*th thread will add

```
z [ i ] = x [ i ] + y [ i ];
```

Since GPUs tend to have more 32-bit than 64-bit floating point units, let's use arrays of floats rather than doubles:

```
float *x , *y , *z ;
```

After allocating and initializing the arrays, we'll call the kernel, and after the kernel completes execution, the program checks the result, frees memory, and quits. See Program 6.3, which shows the kernel and the main function.

Let's take a closer look at the program.

```
1  __global__ void Vec_add(
2        const float x[]  /* in  */,
3        const float y[]  /* in  */,
4        float       z[]  /* out */,
5        const int   n    /* in  */) {
6     int my_elt = blockDim.x * blockIdx.x + threadIdx.x;
7
8     /* total threads = blk_ct*th_per_blk may be > n */
9     if (my_elt < n)
10       z[my_elt] = x[my_elt] + y[my_elt];
11 }  /* Vec_add */
12
13 int main(int argc, char* argv[]) {
14    int n, th_per_blk, blk_ct;
15    char i_g;  /* Are x and y user input or random? */
16    float *x, *y, *z, *cz;
17    double diff_norm;
18
19    /* Get the command line arguments, and set up vectors */
20    Get_args(argc, argv, &n, &blk_ct, &th_per_blk, &i_g);
21    Allocate_vectors(&x, &y, &z, &cz, n);
22    Init_vectors(x, y, n, i_g);
23
24    /* Invoke kernel and wait for it to complete          */
25    Vec_add <<<blk_ct, th_per_blk>>>(x, y, z, n);
26    cudaDeviceSynchronize();
27
28    /* Check for correctness */
29    Serial_vec_add(x, y, cz, n);
30    diff_norm = Two_norm_diff(z, cz, n);
31    printf("Two-norm of difference between host and ");
32    printf("device = %e\n", diff_norm);
33
34    /* Free storage and quit */
35    Free_vectors(x, y, z, cz);
36    return 0;
37 }  /* main */
```

Program 6.3: Kernel and `main` function of a CUDA program that adds two vectors.

6.8.1 The kernel

In the kernel (Lines 1–11), we first determine which element of z the thread should compute. We've chosen to make this index the same as the *global* rank or index of the thread. Since we're only using the x fields of the blockDim and threadIdx structs, there are a total of

gridDim . x ∗ blockDim . x

threads. So we can assign a unique "global" rank or index to each thread by using the formula

rank = blockDim . x ∗ blockIdx . x + threadIdx . x

For example, if we have four blocks and five threads in each block, then the global ranks or indexes are shown in Table 6.4.

Table 6.4 Global thread ranks or indexes in a grid with 4 blocks and 5 threads per block.

	threadIdx.x				
blockIdx.x	**0**	**1**	**2**	**3**	**4**
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

In the kernel, we assign this global rank to my_elt and use this as the subscript for accessing each thread's elements of the arrays x, y, and z.

Note that we've allowed for the possibility that the total number of threads may not be exactly the same as the number of components of the vectors. So before car- rying out the addition,

z [my_elt] = x [my_elt] + y [my_elt];

we first check that my_elt < n. For example, if we have n 997, and we want at least two blocks with at least two threads per block, then, since 997 is prime, we can't possibly have exactly 997 threads. Since this kernel needs to be executed by at least n threads, we must start more than 997. For example, we might use four blocks of 256 threads, and the last 27 threads in the last block would skip the line

z [my_elt] = x [my_elt] + y [my_elt];

Note that if we needed to run our program on a system that didn't support CUDA, we could replace the kernel with a serial vector addition function. (See Program 6.4.)

```
1  void Serial_vec_add(
2          const float  x[]     /* in  */,
3          const float  y[]     /* in  */,
4          float        cz[]    /* out */,
5          const int    n       /* in  */) {
6
7      for (int i = 0; i < n; i++)
8          cz[i] = x[i] + y[i];
9  }  /* Serial_vec_add */
```

Program 6.4: Serial vector addition function.

So we can view the CUDA kernel as taking the serial **for** loop and assigning each iteration to a different thread. This is often how we start the design process when we want to parallelize a serial code for CUDA: assign the iterations of a loop to individual threads.

Also note that if we apply Foster's method to parallelizing the serial vector sum, and we make the tasks the additions of the individual components, then we don't need to do anything for the communication and aggregation phases, and the mapping phase simply assigns each addition to a thread.

6.8.2 Get_args

After declaring the variables, the main function calls a Get_args function, which re- turns *n*, the number of elements in the arrays, blk_ct, the number of thread blocks, and th_per_blk, the number of threads in each block. It gets these from the command line. It also returns a **char** i_g. This tells the program whether the user will *i*nput x and y or whether it should *g*enerate them using a random number generator. If the user doesn't enter the correct number of command line arguments, the function prints a usage summary and terminates execution. Also if *n* is greater than the total number of threads, it prints a message and terminates. (See Program 6.5.) Note that Get_args is written in standard C, and it runs completely on the host.

```
1  void Get_args(
2        const int argc           /* in  */,
3        char*     argv[]         /* in  */,
4        int*      n_p            /* out */,
5        int*      blk_ct_p       /* out */,
6        int*      th_per_blk_p   /* out */,
7        char*     i_g            /* out */) {
8     if (argc != 5) {
9        /* Print an error message and exit */
10       ...
11    }
12
13    *n_p = strtol(argv[1], NULL, 10);
14    *blk_ct_p = strtol(argv[2], NULL, 10);
15    *th_per_blk_p = strtol(argv[3], NULL, 10);
16    *i_g = argv[4][0];
17
18    /* Is n > total thread count = blk_ct*th_per_blk? */
19    if (*n_p > (*blk_ct_p)*(*th_per_blk_p)) {
20       /* Print an error message and exit */
21       ...
22    }
23 }  /* Get_args */
```

Program 6.5: `Get_args` function from CUDA program that adds two vectors.

6.8.3 Allocate_vectors and managed memory

After getting the command line arguments, the main function calls Allocate_vectors, which allocates storage for four *n*-element arrays of **float** :

x , y , z , cz

The first three arrays are used on both the host and the device. The fourth array, cz, is only used on the host: we use it to compute the vector sum with one core of the host. We do this so that we can check the result computed on the device. (See Program 6.6.)

```
1  void  Allocate_vectors(
2        float** x_p      /* out */,
3        float** y_p      /* out */,
4        float** z_p      /* out */,
5        float** cz_p     /* out */,
6        int     n        /* in  */) {
7
8     /* x, y, and z are used on host and device */
9     cudaMallocManaged(x_p, n*sizeof(float));
10    cudaMallocManaged(y_p, n*sizeof(float));
11    cudaMallocManaged(z_p, n*sizeof(float));
12
13    /* cz is only used on host */
14    *cz_p = (float*) malloc(n*sizeof(float));
15 }  /* Allocate_vectors */
```

Program 6.6: Array allocation function of CUDA program that adds two vectors.

First note that since cz is only used on the host, we allocate its storage using the standard C library function malloc. For the other three arrays, we allocate storage in Lines 9–11 using the CUDA function

__host__ cudaError_t cudaMallocManaged (

void ** devPtr /* *out* */, size_t size /* *in* */, **unsigned** flags /* *in* */);

The _host_ qualifier is a CUDA addition to C, and it indicates that the function should be called and run on the host. This is the default for functions in CUDA programs, so it can be omitted when we're writing our own functions, and they'll only be run on the host.

The return value, which has type cudaError_t, allows the function to return an er- ror. Most CUDA functions return a cudaError_t value, and if you're having problems with your code, it is a very good idea to check it. However, always checking it tends to clutter the code, and this can distract us from the main purpose of a program. So in the code we discuss we'll generally ignore cudaError_t return values.

The first argument is a pointer to a pointer: it refers to the pointer that's being allocated. The second argument specifies the number of bytes that should be allo- cated. The flags argument controls which kernels can access the allocated memory. It defaults to cudaMemAttachGlobal and can be omitted.

The function cudaMallocManaged is one of several CUDA memory allocation func- tions. It allocates memory that will be automatically managed by the "unified memory system." This

is a relatively recent addition to CUDA,[8] and it allows a programmer to write CUDA programs as if the host and device shared a single memory: pointers referring to memory allocated with cudaMallocManaged can be used on both the device and the host, even when the host and the device have separate physical memories. As you can imagine this greatly simplifies programming, but there are some cautions. Here are a few:

1. Unified memory requires a device with compute capability 3.0, and a 64-bit host operating system.
2. On devices with compute capability < 6.0 memory allocated with cudaMallocManaged cannot be simultaneously accessed by both the device and the host. When a kernel is executing, it has exclusive access to memory allocated with cudaMallocManaged.
3. Kernels that use unified memory can be slower than kernels that treat device mem- ory as separate from host memory.

The last caution has to do with the transfer of data between the host and the device. When a program uses unified memory, it is up to the system to decide when to transfer from the host to the device or vice versa. In programs that explicitly transfer data, it is up to the programmer to include code that implements the transfers, and she may be able to exploit her knowledge of the code to do things that reduce the cost of transfers, things such as omitting some transfers or overlapping data transfer with computation.

At the end of this section we'll briefly discuss the modifications required if you want to explicitly handle the transfers between host and device.

6.8.4 Other functions called from main

Except for the Free_vectors function, the other host functions that we call from main are just standard C.

The function Init_vectors either reads x and y from stdin using scanf or generates them using the C library function random. It uses the last command line argument i_g to decide which it should do.

The Serial_vec_add function (Program 6.4) just adds x and y on the host using a **for** loop. It stores the result in the host array cz.

The Two_norm_diff function computes the "distance" between the vector z com- puted by the kernel and the vector cz computed by Serial_vec_add. So it takes the difference between corresponding components of z and cz, squares them, adds the squares, and takes the square root:

$$\sqrt{(z[0]-cz[0])^2 + (z[1]-cz[1])^2 + \cdots + (z[n-1]-cz[n-1])^2}.$$

See Program 6.7.

```
1  double Two_norm_diff(
2         const float  z[]     /* in */,
3         const float  cz[]    /* in */,
4         const int    n       /* in */) {
5     double diff,    sum = 0.0;
6
7     for (int i = 0; i < n; i++) {
8        diff = z[i] - cz[i];
9        sum += diff*diff;
10    }
11    return sqrt(sum);
12 } /* Two_norm_diff */
```

Program 6.7: C function that finds the distance between two vectors.

The Free_vectors function just frees the arrays allocated by Allocate_vectors. The array cz is freed using the C library function free, but since the other arrays are allocated using cudaMallocManaged, they must be freed by calling cudaFree:

__host__ __devic e__ cudaError_t cudaFree (**void** * ptr)

The qualifier _device_ is a CUDA addition to C, and it indicates that the function can be called from the device. So cudaFree can be called from the host or the device. However, if a pointer is allocated on the device, it cannot be freed on the host, and vice versa.

It's important to note that unless memory allocated on the device is explicitly freed by the program, it won't be freed until the program terminates. So if a CUDA program calls two (or more) kernels, and the memory used by the first kernel isn't explicitly freed before the second is called, it will remain allocated, regardless of whether the second kernel actually uses it.

See Program 6.8.

```
1  void  Free_vectors(
2          float *  x      /* in/out */,
3          float *  y      /* in/out */,
4          float *  z      /* in/out */,
5          float *  cz     /* in/out */) {
6
7      /* Allocated with cudaMallocManaged */
8      cudaFree(x);
9      cudaFree(y);
10     cudaFree(z);
11
12     /* Allocated with malloc */
13     free(cz);
14 }  /* Free_vectors */
```

Program 6.8: CUDA function that frees four arrays.

6.8.5 Explicit memory transfers[9]

Let's take a look at how to modify the vector addition program for a system that doesn't provide unified memory. Program 6.9 shows the kernel and main function for the modified program.

```
1  __global__ void Vec_add(
2          const float  x[]   /* in  */,
3          const float  y[]   /* in  */,
4          float        z[]   /* out */,
5          const int    n     /* in  */) {
6      int my_elt = blockDim.x * blockIdx.x + threadIdx.x;
7
8      if (my_elt < n)
9          z[my_elt] = x[my_elt] + y[my_elt];
10 }  /* Vec_add */
11
12 int main(int argc, char* argv[]) {
13     int n, th_per_blk, blk_ct;
14     char i_g;    /* Are x and y user input or random? */
15     float *hx, *hy, *hz, *cz;      /* Host arrays          */
16     float *dx, *dy, *dz;           /* Device arrays        */
17     double diff_norm;
18
19     Get_args(argc, argv, &n, &blk_ct, &th_per_blk, &i_g);
20     Allocate_vectors(&hx, &hy, &hz, &cz, &dx, &dy, &dz, n);
21     Init_vectors(hx, hy, n, i_g);
22
23     /* Copy vectors x and y from host to device */
24     cudaMemcpy(dx, hx,     n*sizeof(float), cudaMemcpyHostToDevice);
25     cudaMemcpy(dy, hy,     n*sizeof(float), cudaMemcpyHostToDevice);
26
27
28     Vec_add <<<blk_ct, th_per_blk>>>(dx, dy, dz, n);
29
30     /* Wait for kernel to complete and copy result to host */
31     cudaMemcpy(hz, dz,     n*sizeof(float), cudaMemcpyDeviceToHost);
32
33     Serial_vec_add(hx,       hy, cz, n);
34     diff_norm = Two_norm_diff(hz, cz, n);
35     printf("Two-norm of difference between host and ");
36     printf("device = %e\n", diff_norm);
37
38     Free_vectors(hx, hy, hz, cz, dx, dy, dz);
39
40     return 0;
41 }  /* main */
```

Program 6.9: Part of CUDA program that implements vector addition without unified memory.

The first thing to notice is that the kernel is unchanged: the arguments are x, y, z, and n. It finds the thread's global index, my_elt, and if this is less than n, it adds the elements of x and y to get the corresponding element of z.

The basic structure of the main function is almost the same. However, since we're assuming unified memory is unavailable, pointers on the host aren't valid on the de- vice, and vice versa: an address on the host may be illegal on the device, or, even worse, it might refer to memory that the device is using for some other purpose. Similar problems occur if we try to use a device address on the host. So instead of declaring and allocating storage for three arrays that are all valid on both the host and the device, we declare and allocate storage for three arrays that are valid on the host hx, hy, and hz, *and* we declare and allocate storage for three arrays that are valid on the device, dx, dy, and dz. The declarations are in Lines 15–16, and the allocations are in the Allocate_vectors function called in Line 20. The function itself is in Program 6.10.

```
1  void Allocate_vectors(
2          float** hx_p    /* out */,
3          float** hy_p    /* out */,
4          float** hz_p    /* out */,
5          float** cz_p    /* out */,
6          float** dx_p    /* out */,
7          float** dy_p    /* out */,
8          float** dz_p    /* out */,
9          int     n       /* in  */)  {
10
11     /* dx, dy, and dz are used on device */
12     cudaMalloc(dx_p, n*sizeof(float));
13     cudaMalloc(dy_p, n*sizeof(float));
14     cudaMalloc(dz_p, n*sizeof(float));
15
16     /* hx, hy, hz, cz are used on host */
17     *hx_p = (float*) malloc(n*sizeof(float));
18     *hy_p = (float*) malloc(n*sizeof(float));
19     *hz_p = (float*) malloc(n*sizeof(float));
20     *cz_p = (float*) malloc(n*sizeof(float));
21 }  /* Allocate_vectors */
```

Program 6.10: Allocate_vectors function for CUDA vector addition program that doesn't use unified memory.

Since unified memory isn't available, instead of using cudaMallocManaged, we use the C library function malloc for the host arrays, and the CUDA function cudaMalloc for the device arrays:

```
__host__ __device__ cudaError_t cudaMalloc (
        void ** dev_p       /* out */,
        size_t size     /* in  */);
```

The first argument is a reference to a pointer that will be used *on the device*. The second argument specifies the number of bytes to allocate on the device.

After we've initialized hx and hy on the host, we copy their contents over to the device, storing the transferred contents in the memory allocated for dx and dy, respec- tively. The copying is done in Lines 24–26 using the CUDA function cudaMemcpy:

```
__host__ cudaError_t cudaMemcpy (
void * dest /* out */, c o n s t void * source /* in */, size_t     count /* in  */,
cudaMemcpyKind kind  /* in  */);
```

This copies count bytes from the memory referred to by source into the memory referred to by dest. The type of the kind argument, cudaMemcpyKind, is an enumer- ated type defined by CUDA that specifies where the source and dest pointers are located. For our purposes the two values of interest are cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost. The first indicates that we're copying from the host to the device, and the second indicates that we're copying from the device to the host.

The call to the kernel in Line 28 uses the pointers dx, dy, and dz, because these are addresses that are valid on the *device*.

After the call to the kernel, we copy the result of the vector addition from the device to the host in Line 31 using cudaMemcpy again. A call to cudaMemcpy is *syn- chronous*, so it waits for the kernel to finish executing before carrying out the transfer. So in this version of vector addition we do *not* need to use cudaDeviceSynchronize to ensure that the kernel has completed before proceeding.

After copying the result from the device back to the host, the program checks the result, frees the memory allocated on the host and the device, and terminates. So for this part of the program, the only difference from the original program is that we're freeing seven pointers instead of four. As before, the Free_vectors function frees the storage allocated on the host with the C library function free. It uses cudaFree to free the storage allocated on the device.

6.9 Returning results from CUDA kernels

There are several things that you should be aware of regarding CUDA kernels. First, they always have return type **void**, so they can't be used to return a value. They also can't return anything to the host through the standard C pass-by-reference. The reason for this is that addresses on the host are, in most systems, invalid on the device, and vice versa. For example, suppose we try something like this:

```
__global__ void Add(int x, int y, int *       sum_p) {
    *sum_p = x + y;
}   /*  Add */

 int main(void) {
    int sum = _5;
    Add <<<1, 1>>> (2, 3, &sum);
    cudaDeviceSynchronize();
    printf("The sum is %d\n", sum);

    return 0;
}
```

It's likely that either the host will print -5 or the device will hang. The reason is that the address &sum is probably invalid on the device. So the dereference

∗ sum_p = x + y ;

is attempting to assign x + y to an invalid memory location.

There are several possible approaches to "returning" a result to the host from a kernel. One is to declare pointer variables and allocate a single memory location. On a system that supports unified memory, the computed value will be automatically copied back to host memory:

```
__global__ void Add ( int x , int y , int ∗ sum_p ) {

        ∗ sum_p = x + y ;

}

/∗      Add ∗ /
```

```
int main(void) {
    int * sum_p;
    cudaMallocManaged(&sum_p, sizeof(int));
    *sum_p = 5;
    Add <<<1, 1>>> (2, 3, sum_p);
    cudaDeviceSynchronize();
    printf("The sum is %d\n", *sum_p);
    cudaFree(sum_p);

    return 0;
}
```

If your system doesn't support unified memory, the same idea will work, but the result will have to be explicitly copied from the device to the host:

```
__global__ void Add(int x, int y, int *sum_p) {
    *sum_p = x + y;
} /* Add */

int main(void) {
    int *hsum_p, *dsum_p;
    hsum_p = (int *) malloc(sizeof(int));
    cudaMalloc(&dsum_p, sizeof(int));
    *hsum_p = 5;
    Add <<<1, 1>>> (2, 3, dsum_p);
    cudaMemcpy(hsum_p, dsum_p, sizeof(int),
            cudaMemcpyDeviceToHost);
    printf("The sum is %d\n", *hsum_p);
    free(hsum_p);
    cudaFree(dsum_p);

    return 0;
}
```

Note that in both the unified and non-unified memory settings, we're returning a *single* value from the device to the host.

If unified memory is available, another option is to use a global managed variable for the sum:

```
__managed__ int sum;

__global__ void Add(int x, int y) {
    sum = x + y;
}  /* Add */

int main(void) {
    sum = _5;
    Add <<<1, 1>>> (2, 3);
    cudaDeviceSynchronize();
    printf("After kernel:          The sum is %d\n", sum);

    return 0;
}
```

The qualifier _managed_ declares sum to be a managed **int** that is accessible to all the functions, regardless of whether they run on the host or the device. Since it's man- aged, the same restrictions apply to it that apply to managed variables allocated with cudaMallocManaged. So this option is unavailable on systems with compute capability < 3.0, and on systems with compute capability < 6.0, sum can't be accessed on the host while the kernel is running. So after the call to Add has started, the host can't access sum until after the call to cudaDeviceSynchronize has completed.

Since this last approach uses a global variable, it has the usual problem of reduced modularity associated with global variables.

6.10 CUDA trapezoidal rule I

6.10.1 The trapezoidal rule

Let's try to implement a CUDA version of the trapezoidal rule. Recall (see Section 3.2.1) that the trapezoidal rule estimates the area between an interval on the x-axis and the graph of a function by dividing the interval into subintervals and approximating the area between each subinterval and the graph by the area of a trape- zoid. (See Fig. 3.3.) So if the interval is $[a, b]$ and there are n trapezoids, we'll divide

$[a, b]$ into n equal subintervals, and the length of each subinterval will be

$$h = (b - a)/n.$$

Then if x_i is the left end point of the ith subinterval,

$$x_i = a + ih,$$

for i 0, 1, 2,...,n 1. To simplify the notation, we'll also denote b, the right end point of the interval, as

$$b = x_n = a + nh.$$

Recall that if a trapezoid has height h and base lengths c and d, then its area is

$$\frac{h}{2}(c + d).$$

So if we think of the length of the subinterval x_i, x_{i+1} as the height of the ith trapezoid, and $f(x_i)$ and $f(x_{i-1})$ as the two base lengths (see Fig. 3.4), then the area of the ith trapezoid is

$$\frac{h}{2}[f(x_i) + f(x_{i+1})].$$

This gives us a total approximation of the area between the graph and the x-axis as

$$\frac{h}{2}[f(x_0) + f(x_1)] + \frac{h}{2}[f(x_1) + f(x_2)] + \cdots + \frac{h}{2}[f(x_{n-1}) + f(x_n)],$$

and we can rewrite this as

$$h\left[\frac{1}{2}(f(a) + f(b)) + (f(x_1) + f(x_2) + \cdots + f(x_{n-1}))\right].$$

We can implement this with the serial function shown in Program 6.11.

```
1  float Serial_trap(
2        const float  a   /* in */,
3        const float  b   /* in */,
4        const int    n   /* in */) {
5     float x, h = (b-a)/n;
6     float trap = 0.5*(f(a) + f(b));
7
8     for (int i = 1; i <= n-1; i++) {
9        x = a + i*h;
10       trap += f(x);
11    }
12    trap = trap*h;
13
14    return trap;
15 } /* Serial_trap */
```

Program 6.11: A serial function implementing the trapezoidal rule for a single CPU.

6.10.2 A CUDA implementation

If n is large, the vast majority of the work in the serial implementation is done by the **for** loop. So when we apply Foster's method to the trapezoidal rule, we're mainly interested in two types of tasks: the first is the evaluation of the function f at x_i, and the second is the addition of $f(x_i)$ into trap. Here i 1,...,n 1. The second type of task depends on the first. So we can aggregate these two tasks.

This suggests that each thread in our CUDA implementation might carry out one iteration of the serial **for** loop. We can assign a unique integer rank to each thread as we did with the vector addition program. Then we can compute an x-value, the function value, and add in the function value to the "running sum":

```
/* h and trap are formal arguments to the kernel */
int my_i = blockDim.x * blockIdx.x + threadIdx.x;
float my_x = a + my_i * h; float
my_trap = f(my_x); float trap
+= my_trap;
```

However, it's immediately obvious that there are several problems here:

1. We haven't initialized h or trap.
2. The my_i value can be too large or too small: the serial loop ranges from 1 up to and including n 1. The smallest value for my_i is 0 and the largest is the total number of threads minus 1.
3. The variable trap must be shared among the threads. So the addition of my_trap forms a race condition: when multiple threads try to update trap at roughly the same time, one thread can overwrite another thread's result, and the final value in trap may be wrong. (For a discussion of race conditions, see Section 2.4.3.)
4. The variable trap in the serial code is returned by the function, and, as we've seen, kernels must have **void** return type.
5. We see from the serial code that we need to multiply the total in trap by h after all of the threads have added their results.

Program 6.12 shows how we might deal with these problems. In the following sections, we'll look at the rationales for the various choices we've made.

```
1  __global__ void Dev_trap(
2          const float a          /* in       */,
3          const float b          /* in       */,
4          const float h          /* in       */,
5          const int   n          /* in       */,
6          float *     trap_p     /* in/out   */) {
7      int my_i = blockDim.x * blockIdx.x + threadIdx.x;
8
9      /* f(x_0) and f(x_n) were computed on the host.  So */
10     /* compute f(x_1), f(x_2), ..., f(x_(n-1))          */
11     if (0 < my_i && my_i < n) {
12         float my_x = a + my_i*h;
13         float my_trap = f(my_x);
14         atomicAdd(trap_p, my_trap);
15     }
16 }  /* Dev_trap */
17
18 /* Host code */
19 void Trap_wrapper(
20         const float a              /* in    */,
21         const float b              /* in    */,
22         const int   n              /* in    */,
23         float *     trap_p_blk_ct  /* out   */,
24         const int   th_per_blk     /* in    */,
25         const int                  /* in    */) {
26
27     /* trap_p storage allocated in main with
28      * cudaMallocManaged */
29     *trap_p = 0.5*(f(a) + f(b));
30     float h = (b-a)/n;
31
32     Dev_trap<<<blk_ct, th_per_blk>>>(a, b, h, n, trap_p);
33     cudaDeviceSynchronize();
34
35     *trap_p = h*(*trap_p);
36 }  /* Trap_wrapper */
```

Program 6.12: CUDA kernel and wrapper implementing trapezoidal rule.

6.10.3 Initialization, return value, and final update

To deal with the initialization and the final update (Items 1 and 5), we could try to select a single thread—say, thread 0 in block 0—to carry out the operations:

```
int  my_i = blockDim . x       blockIdx . x + threadIdx . x ;

if  ( my_i == 0 ) {

h = ( b−a )/ n ;

trap = 0 . 5 ∗ ( f ( a ) + f ( b ));

}

. . .

if  ( my_i == 0 )

trap = trap ∗h ;
```

There are (at least) a couple of problems with these options: formal arguments to functions are private to the executing thread and **thread synchronization**.

Kernel and function arguments are private to the executing thread.

Like the threads started in Pthreads and OpenMP, each CUDA thread has its own stack and, and since formal arguments are allocated on the thread's stack, each thread has its own private variables h and trap. So, any changes made to one of these variables by one thread won't be visible to the other threads. We could have each thread initialize h, but we could also just do the initialization once in the host. If we do this before the kernel is called, each thread will get a copy of the value of h.

Things are more complicated with trap. Since it's updated by multiple threads, it must be *shared* among the threads. We can achieve the *effect* of sharing trap by allocating storage for a memory location before the kernel is called. This allocated memory location will correspond to what we've been calling trap. Now we can pass a *pointer* to the memory location to the kernel. That is, we can do something like this:

```
/* Host code */
float * trap_p;
cudaMallocManaged(&trap_p, sizeof(float));
...
*trap_p = 0.5*(f(a) + f(b));

/* Call kernel */
...

/* After return from kernel */
*trap_p = h*(*trap_p);
```

When we do this, each thread will get its own copy of trap_p, but all of the copies of trap_p will refer to the same memory location. So ∗trap_p will be shared.

Note that using a pointer instead of a simple float also solves the problem of returning the value of trap in Item 4.

A wrapper function

If you look at the code in Program 6.12, you'll see that we've placed most of the code we use before and after calling the kernel in a **wrapper function**, Trap_wrapper. A wrapper function is a function whose main purpose is to call another function. It can perform any preparation needed for the call. It can also perform any additional work needed after the call.

Using the correct threads

We assume that the number of threads, blk_ct∗th_per_blk, is at least as large as the number of trapezoids. Since the serial **for** loop iterates from 1 up to n−1, thread 0 and any thread with my_i >*n* 1, shouldn't execute the code in the body of the serial **for** loop. So we should include a test before the main part of the kernel code

```
if (0 < my_i && my_i < n ) {
/* Compute x , f ( x ) , and add i n t o *trap_p */
. . .
}
```

See Line 11 in Program 6.12.

6.10.4 Updating the return value and atomicAdd

This leaves the problem of updating ∗trap_p (Item 3 in the list above). Since the memory location is shared, an update such as

```
∗ trap_p += my_trap ;
```

forms a race condition, and the actual value ultimately stored in ∗trap_p will be unpredictable. We're solving this problem by using a special CUDA library function, atomicAdd, to carry out the addition.

An operation carried out by a thread is **atomic** if it appears to all the other threads as if it were "indivisible." So if another thread tries to access the result of the operation or an operand used in the operation, the access will occur either before the operation started or after the operation completed. Effectively, then, the operation appears to consist of a single, indivisible, machine instruction.

As we saw earlier (see Section 2.4.3), addition is not ordinarily an atomic op- eration: it consists of several machine instructions. So if one thread is executing an addition, it's possible for another thread to access the operands and the result while the addition is in progress. Because of this, the CUDA library defines several atomic addition functions. The one we're using has the following syntax:

```
__device__ float atomicAdd (
      float ∗ float_p /∗ i n / out ∗ / ,
      float  val    /∗ in  ∗ / );
```

This atomically adds the contents of val to the contents of the memory referred to by float_p and stores the result in the memory referred to by float_p. It returns the value of the memory referred to by float_p at the beginning of the call. See Line 14 of Program 6.12.

6.10.5 Performance of the CUDA trapezoidal rule

We can find the run-time of our trapezoidal rule by finding the execution time of the Trap_wrapper function. The execution of this function includes all of the computa- tions carried out by the serial trapezoidal rule, including the initialization of trap_p (Line 29) and h (Line 30), and the final update to trap_p (Line 35). It also includes all of the calculations in the body of the serial **for** loop in the Dev_trap kernel. So we can effectively determine the run-time of the CUDA trapezoidal rule by timing a host function, and we only need to insert calls to our timing functions before and after the call to Trap_wrapper. We use the GET_TIME macro defined in the timer.h header file on the book's website:

```
double start , finish ;
. . .
GET_TIME ( start );
Trap_wrapper ( a , b , n , trap_p , blk_ct , th_per_blk );
GET_TIME ( finish );
printf ( " Elapsed time for cuda = % e seconds \ n " ,
finish−start );
```

The same approach can be used to time the serial trapezoidal rule:

```
GET_TIME ( start )
trap = Serial_trap ( a , b , n );
GET_TIME ( finish );
printf ( " Elapsed time for cpu = % e seconds \ n " ,
finish−start );
```

Recall from the section on taking timings (Section 2.6.4) that we take a number of timings, and we ordinarily report the minimum elapsed time. However, if the vast majority of the times are much greater (e.g., 1% or 0.1% greater), then the minimum time may not be reproducible. So other users who run the program may get a time much larger than ours. When this happens, we report the mean or median of the elapsed times.

Now when we ran this program on our hardware, there *were* a number of times that were within 1% of the minimum time. However, we'll be comparing the run- times of this program with programs that had very few run-times within 1% of the minimum. So for our discussion of implementing the trapezoidal rule using CUDA (Sections 6.10–6.13), we'll use the *mean* run-time, and the means are taken over at least 50 executions.

When we run the serial trapezoidal and the CUDA trapezoidal rule functions many times and take the means of the elapsed times, we get the results shown in Table 6.5.

Table 6.5 Mean run-times for serial and CUDA trapezoidal rule (times are in ms).

System	ARM Cortex-A15	Nvidia GK20A	Intel Core i7	Nvidia GeForce GTX Titan X
Clock	2.3 GHz	852 MHz	3.5 GHz	1.08 GHz
SMs, SPs		1, 192		24, 3072
Run-time	33.6	20.7	4.48	3.08

These were taken using $n = 2^{20} = 1{,}048{,}576$ trapezoids with $f(x) = x^2 + 1$, $a = -3$, and $b = 3$. The GPUs use 1024 blocks with 1024 threads per block for a total of 1,048,576 threads. The 192 SPs of the GK20A are clearly much faster than a fairly slow conventional processor, an ARM Cortex-A15, but a single core of an Intel Core i7 is much faster than the GK20A. The 3072 SPs on a Titan X *were* 45% faster than the single core of the Intel, but it would seem that with 3072 SPs, we should be able to do better.

6.11 CUDA trapezoidal rule II: improving performance

If you've read the Pthreads or OpenMP chapter, you can probably make a good guess at how to make the CUDA program run faster. For a thread's call to atomicAdd to

actually be atomic, no other thread can update ∗trap_p while the call is in progress. In other words, the updates to ∗trap_p can't take place simultaneously, and our program may not be very parallel at this point.

One way to improve the performance is to carry out a tree-structured global sum that's similar to the tree-structured global sum we introduced in the MPI chapter (Section 3.4.1). However, because of the differences between the GPU architecture and the distributed-memory CPU architecture, the details are somewhat different.

Tree-structured communication

We can visualize the execution of the "global sum" we implemented in the CUDA trapezoidal rule as a more or less random, linear ordering of the threads. For example, suppose we have only 8 threads and one thread block. Then our threads are 0, 1,..., 7, and one of the threads will be the first to succeed with the call to atomicAdd. Say it's thread 5. Then another thread will succeed. Say it's thread 2. Continuing in this fashion we get a sequence of atomicAdds, one per thread. Table 6.6 shows how this might proceed over time.

Table 6.6 Basic global sum with eight threads.

Time	Thread	my_trap	*trap_p
Start	—	—	9
t_0	5	11	20
t_1	2	5	25
t_2	3	7	32
t_3	7	15	47
t_4	4	9	56
t_5	6	13	69
t_6	0	1	70
t_7	1	3	73

Here, we're trying to keep the computations simple: we're assuming that $f(x) = 2x + 1$, $a = 0$, and $b = 8$. So $h = (8 - 0)/8 = 1$, and the value referenced by trap_p at the start of the global sum is

$$0.5 \times (f(a) + f(b)) = 0.5 \times (1 + 17) = 9.$$

What's important is that this approach may serialize the threads. So the com- putation may require a *sequence* of 8 calculations. Fig. 6.3 illustrates a possible computation.

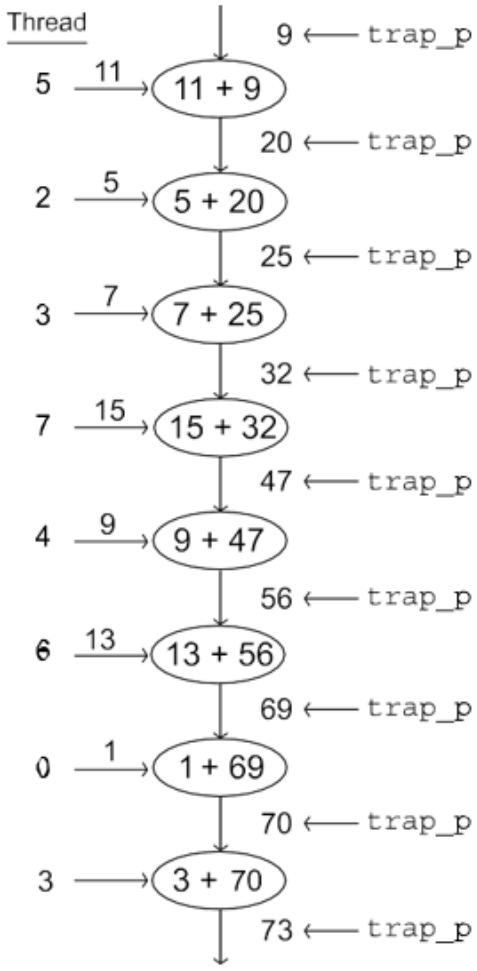


FIGURE 6.3

Basic sum.

So rather than have each thread wait for its turn to do an addition into ∗trap_p, we can pair up the threads so that half of the "active" threads add their partial sum to their partner's partial sum. This gives us a structure that resembles a tree (or, perhaps better, a shrub). See Fig. 6.4.

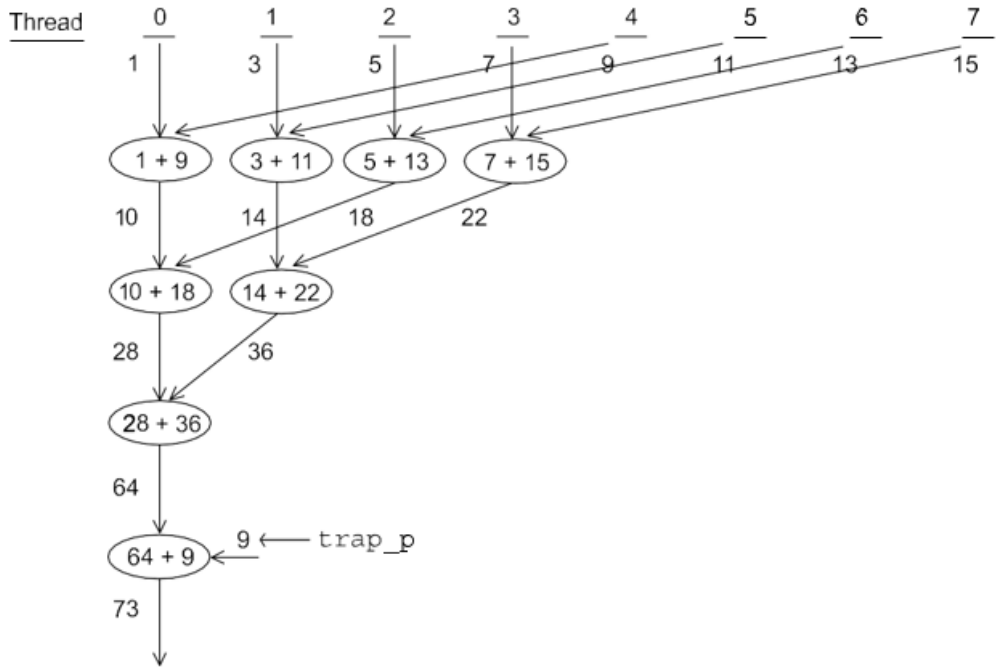


FIGURE 6.4

Tree-structured sum.

In our figures, we've gone from requiring a sequence of 8 consecutive additions to a sequence of 4. More generally, if we double the number of threads and values (e.g., increase from 8 to 16), we'll double the length of the sequence of additions using the basic approach, while we'll only add one using the second, tree-structured approach. For example, if we increase the number of threads and values from 8 to 16, the first approach requires a sequence of 16 additions, but the tree-structured approach only requires 5., if there are t threads and t values, the first approach requires a sequence of t additions, while In fact the tree-structured approach requires $\log_2(t) + 1$. For example, if we have 1000 threads and values, we'll go from 1000 communications and sums using the basic approach to 11 using the tree-structured approach, and if we have 1,000,000, we'll go from 1,000,000 to 21!

There are two standard implementations of a tree-structured sum in CUDA. One implementation uses shared memory, and in devices with compute capability < 3 this is the best implementation. However, in devices with compute capability >= 3 there are several functions called **warp shuffles**, that allow a collection of threads within a *warp* to read variables stored by other threads in the warp.

6.11.1 Local variables, registers, shared and global memory

Before we explain the details of how warp shuffles work, let's digress for a moment and talk about memory in CUDA. In Section 6.2 we mentioned that SMs in an Nvidia processor have access to two collections of memory locations: each SM has access to its own "shared" memory, which is accessible only to the SPs belonging to the SM. More precisely, the shared memory allocated for a thread block is only accessible to the threads in that block. On the other hand, all of the SPs and all of the threads have access to "global" memory. The number

of shared memory locations is rela- tively small, but they are quite fast, while the number of global memory locations is relatively large, but they are relatively slow. So we can think of the GPU memory as a hierarchy with three "levels." At the bottom, is the slowest, largest level: global memory. In the middle is a faster, smaller level: shared memory. At the top is the fastest, smallest level: the registers. For example, Table 6.7 gives some information on relative sizes.

Table 6.7 Memory statistics for some Nvidia GPUs.

GPU	Compute Capability	Registers: Bytes per Thread	Shared Mem: Bytes per Block	Global Mem: Bytes per GPU
Quadro 600	2.1	504	48K	1G
GK20A (Jetson TK1)	3.2	504	48K	2G
GeForce GTX Titan X	5.2	504	48K	12G

Access times also increase dramatically. It takes on the order of 1 cycle to copy a 4-byte int from one register to another. Depending on the system it can take up to an order of magnitude more time to copy from one shared memory location to another, and it can take from two to three orders of magnitude more time to copy from one global memory location to another.

An obvious question here: what about local variables? How much storage is avail- able for them? And how fast is it? This depends on total available memory and program memory usage. If there is enough storage, local variables are stored in registers. However, if there isn't enough register storage, local variables are "spilled" to a region of global memory that's thread private, i.e., only the thread that owns the local variables can access them.

So as long as we have sufficient register storage, we expect the performance of a kernel to improve if we increase our use of registers and reduce our use of shared and/or global memory. The catch, of course, is that the storage available in registers is tiny compared to the storage available in shared and global memory.

6.6.1 Warps and warp shuffles

In particular, if we can implement a global sum in registers, we expect its performance to be superior to an implementation that uses shared or global memory, and the **warp shuffle** functions introduced in CUDA 3.0 allow us to do this.

In CUDA a **warp** is a set of threads with consecutive ranks belonging to a thread block. The number of threads in a warp is currently 32, although Nvidia has stated that this could change. There is a variable initialized by the system that stores the size of a warp:

int warpSize

The threads in a warp operate in SIMD fashion. So threads in different warps can execute different statements with no penalty, while threads within the same warp must execute the same statement. When the threads within a warp attempt to execute different statements—e.g., they take different branches in an **if else** statement—the threads are said to have **diverged**. When divergent threads finish executing different statements, and start executing the same statement, they are said to have **converged**. The rank of a thread within a warp is called the thread's **lane**, and it can be com-

puted using the formula

lane = threadIdx . x % warpSize ;

The warp shuffle functions allow the threads in a warp to read from registers used by another thread in the same warp. Let's take a look at the one we'll use to implement a tree-structured sum of the values stored by the threads in a warp[10]:

```
__device__ float __shfl_down_sync(
              unsigned mask                    /* in */,
              float    var                     /* in */,
              unsigned diff                    /* in */,
                int      width = warpSize /* in */);
```

The mask argument indicates which threads are participating in the call. A bit, representing the thread's lane, must be set for each participating thread to ensure that all of the threads in the call have converged—i.e., arrived at the call—before any thread begins executing the call to _s mask = 0 xffffffff ;

Recall that 0x denotes a hexadecimal (base 16) value and 0xf is 15_10, which is 1111_2.[11] So this value of mask is 32 1's in binary, and it indicates that every thread in the warp participates in the call to _shfl_down_sync. If the thread with lane *l* calls

_shfl_down_sync, then the value stored in var on the thread with

lane = *l* + diff

is returned on thread *l*. Since diff has type **unsigned**, it is 0. So the value that's returned is from a *higher*-ranked thread. Hence the name "shuffle *down*".

We'll only use width = warpSize, and since its default value is warpSize, we'll omit it from our calls.

There are several possible issues:

- What happens if thread *l* calls _shfl_down_sync but thread *l* diff doesn't? In this case, the value returned by the call on thread *l* is *undefined*.

- What happens if thread *l* calls _shfl_down_sync but *l* diff warpSize? In this case the call will return the value in var already stored on thread *l*.
- What happens if thread *l* calls _shfl_down_sync, and *l* diff < warpSize, but *l* diff > largest lane in the warp. In other words, because the thread block size is not a multiple of warpSize, the last warp in the block has fewer than warpSize threads. Say there are *w* threads in the last warp, where 0 <*w* < warpSize. Then if

$$l + \text{diff} \geq w,$$

the value returned by the call is also undefined. So to avoid undefined results, it's best if

- All the threads in the warp call _shfl_down_sync, and
- All the warps have warpSize threads, or, equivalently, the thread block size (blockDim.x) is a multiple of warpSize.

6.6.1 Implementing tree-structured global sum with a warp shuffle

So we can implement a tree-structured global sum using the following code:

```
__device__ float Warp_sum ( float var ) {
        unsigned mask = 0 xffffffff ;
                for ( int diff = warpSize /2; diff > 0 ; diff = diff /2)
                        var += __shfl_down_sync_sync ( mask , var , diff );
        return var ;
}       / * Warp_sum * /
```

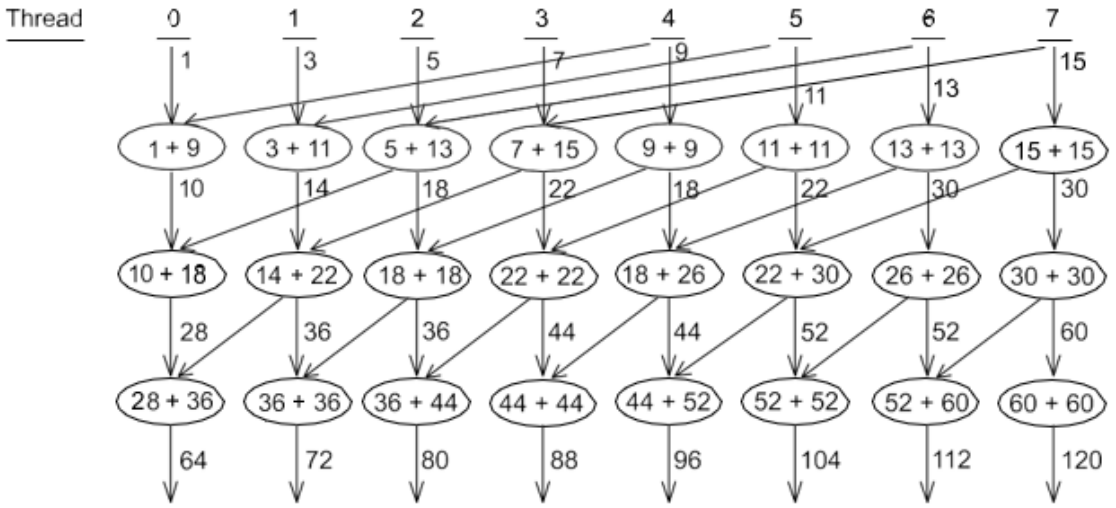


FIGURE 6.5

Tree-structured sum using warp shuffle.

Fig. 6.5 shows how the function would operate if warpSize were 8. (The diagram would be illegible if we used a warpSize of 32.) Perhaps the most confusing point in the behavior of shfl_down_sync is that when the lane ID

$$l + \text{diff} \geq \text{warpSize},$$

the call returns the value in the *caller's* var. In the diagram this is shown by having only one arrow entering the oval with the sum, and it's labeled with the value just calculated by the thread carrying out the sum. In the row corresponding to diff = 4 (the first row of sums), the threads with lane IDs l = 4, 5, 6, and 7 all have l 4 8. So the call to shfl_down_sync returns their current var values, 9, 11, 13, and 15, respectively, and these values are doubled, because the return value of the call is added into the calling thread's variable var. Similar behavior occurs in the row corresponding to the sums for diff = 2 and lane IDs l = 6 and 7, and in the last row when diff = 1 for the thread with lane ID l = 7.

From a practical standpoint, it's important to remember that this implementation will only return the correct sum on the thread with lane ID 0. If all of the threads need the result, we can use an alternative warp shuffle function, _shfl_xor. See Eercise 6.6.

6.11.5 Shared memory and an alternative to the warp shuffle

If your GPU has compute capability < 3.0, you won't be able to use the warp shuffle functions in your code, and a thread won't be able to directly access the registers of other threads. However, your code *can* use shared memory, and threads in the same thread block can all access the same shared memory locations. In fact, although shared memory access is

slower than register access, we'll see that the shared memory implementation can be just as fast as the warp shuffle implementation.

Since the threads belonging to a single warp operate synchronously, we can implement something very similar to a warp shuffle using shared memory instead of registers.

```
__device__ float Shared_mem_sum( float shared_vals[]) {
   int my_lane = threadIdx.x % warpSize;

   for (int diff = warpSize/2; diff > 0; diff = diff/2) {
      /* Make sure 0 <= source < warpSize */
      int source = (my_lane + diff) % warpSize;
      shared_vals[my_lane] += shared_vals[source];
   }
   return shared_vals[my_lane];
}
```

This should be called by all the threads in a warp, and the array shared_vals should be stored in the shared memory of the SM that's running the warp. Since the threads in the warp are operating in SIMD fashion, they effectively execute the code of the function in lockstep. So there's no race condition in the updates to shared_vals: all the threads read the values in shared_vals[source] before any thread updates

shared_vals[my_lane].

Technically speaking, this isn't a tree-structured sum. It's sometimes called a **dissemination sum** or **dissemination reduction**. Fig. 6.6 illustrates the copying and additions that take place.

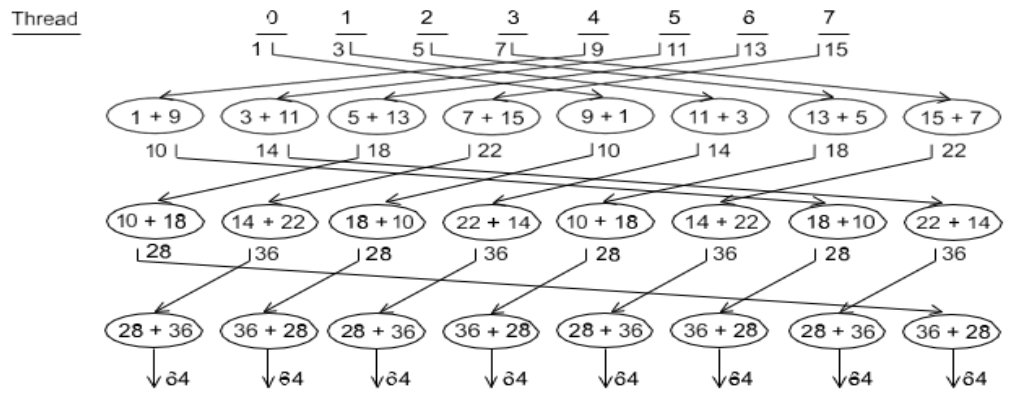


FIGURE 6.6
Dissemination sum using shared memory.

Unlike the earlier figures, this figure doesn't show the di- rect contributions that a thread makes to its sums: including these lines would have made the figure too difficult to read. Also note that every thread reads a value from another thread in each pass through the **for** statement. After all these values have been added in, every thread has the correct sum—not just thread 0. Although we won't need this for the trapezoidal rule, this can be useful in other applications. Furthermore, in any cycle in which the threads in a warp are working, each thread either executes the current instruction or it is idle. So the cost of having every thread exe- cute the same instruction shouldn't be any greater than having some of the threads execute one instruction and the others idle.

An obvious question here is: how does Shared_mem_sum make use of Nvidia's shared memory? The answer is that it's not required to use shared memory. The function's argument, the array shared_vals, could reside in either global memory or shared memory. In either case, the function would return the sum of the elements of shared_vals.

However, to get the best performance, the argument shared_vals should be defined to be shared_ in a kernel. For example, if we know that shared_vals will need to store at most 32 floats in each thread block, we can add this definition to our kernel:

```
__shared__ float shared_vals [32];
```

For each thread block this sets aside storage for a collection of 32 floats in the shared memory of the SM to which the block is assigned.

Alternatively, if it isn't known at compile time how much shared memory is needed, it can be declared as

```
extern __shared__ float shared_vals [];
```

and when the kernel is called, a third argument can be included in the triple angle brackets specifying the size *in bytes* of the block of shared memory. For example, if we were using Shared_mem_sum in a trapezoidal rule program, we might call the kernel Dev_trap with

```
Dev_trap <<<blk_ct , th_per_blk , th_per_blk sizeof ( float )>>> ( . . . args to Dev_trap . . . );
```

This would allocate storage for th_per_blk floats in the shared_vals array in each thread block.

This would allocate storage for th_per_blk floats in the shared_vals array in each thread block.

Implementation of trapezoidal rule with warpSize thread blocks

Let's put together what we've learned about more efficient sums, warps, warp shuf- fles, and shared memory to create a couple of new implementations of the trapezoidal rule.

For both versions we'll assume that the thread blocks consist of warpSize threads, and we'll use one of our "tree-structured" sums to add the results of the threads in the warp. After computing the function values and adding the results within a warp, the thread with lane ID 0 in the warp will add the warp sum into the total using

Atomic_add.

6.11.2 Host code

For both the warp shuffle and the shared memory versions, the host code is virtually identical to the code for our first CUDA version. The only substantive difference is that there is no th_per_blk variable in the new versions, since we're assuming that each thread block has warpSize threads.

6.11.3 Kernel with warp shuffle

Our kernel is shown in Program 6.13. Initialization of my_trap is the same as it was in our original implementation (Program 6.12). However, instead of adding each thread's calculation directly into ∗trap_p, each warp (or, in this case, thread block) calls the Warp_sum function (Fig. 6.5) to add the values computed by the threads in the warp. Then, when the warp returns, thread (or lane) 0 adds the warp sum for its thread block (result) into the global total. Since, in general, this version will use multiple thread blocks, there will be multiple warp sums that need to be added to ∗trap_p. So if we didn't use atomicAdd, the addition of result to ∗trap_p would form a race condition.

```
1  __global__ void Dev_trap(
2          const float  a          /*  in     */,
3          const float  b          /*  in     */,
4          const float  h          /*  in     */,
5          const int    n          /*  in     */,
6          float *      trap_p     /* in/out */) {
7      int my_i = blockDim.x * blockIdx.x + threadIdx.x;
8
9      float my_trap = 0.0f;
10     if (0 < my_i && my_i < n) {
11         float my_x = a + my_i*h;
12         my_trap = f(my_x);
13     }
14
15     float result = Warp_sum(my_trap);
16
17     /* result is correct only on thread 0 */
18     if (threadIdx.x == 0) atomicAdd(trap_p, result);
19 }  /* Dev_trap */
```

Program 6.13: CUDA kernel implementing trapezoidal rule and using Warp_sum.

6.11.4 Kernel with shared memory

The kernel that uses shared memory is shown in Program 6.14.

```
1  __global__ void Dev_trap(
2          const float  a           /* in  */,
3          const float  b           /* in  */,
4          const float  h           /* in  */,
5          const int    n           /* in  */,
6          float *      trap_p      /* out */) {
7      __shared__   float  shared_vals[WARPSZ];
8      int  my_i = blockDim.x * blockIdx.x + threadIdx.x;
9      int  my_lane = threadIdx.x % warpSize;
10
11     shared_vals[my_lane] = 0.0f;
12     if (0 < my_i && my_i < n) {
13         float  my_x = a + my_i*h;
14         shared_vals[my_lane] = f(my_x);
15     }
16
17     float  result = Shared_mem_sum(shared_vals);
18
19     /* result is the same on all threads in a block. */
20     if (threadIdx.x == 0) atomicAdd(trap_p, result);
21 } /* Dev_trap */
```

Program 6.14: CUDA kernel implementing trapezoidal rule and using shared memory.

It is almost identical to the version that uses the warp shuffle. The main differences are that it declares an array of shared memory in Line 7; it initializes this array in Lines 11 and 14; and, of course, the call to Shared_mem_sum is passed this array rather than a scalar register.

Since we know at compile time how much storage we'll need in shared_vals, we can define the array by simply preceding the ordinary C definition with the CUDA qualifier _shared_:

__shared__ **float** shared_vals [WARPSZ];

Note that the CUDA defined variable warpSize is *not* defined at compile-time. So our program defines a preprocessor macro

define WARPSZ 32

6.11.5 Performance

Of course, we want to see how the various implementations perform. (See Table 6.8.) The problem is the same as the problem we ran earlier (see Table 6.5): we're integrating $f(x)=x^2+1$ on the interval [-3, 3], and there are 2^{20} = 1,048,576 trapezoids. However, since the thread block size is 32, we're using 32,768 thread blocks (32 × 32,768 = 1,048,576).

Table 6.8 Mean run-times for trapezoidal rule using block size of 32 threads (times in ms).

System	ARM Cortex-A15	Nvidia GK20A	Intel Core i7	Nvidia GeForce GTX Titan X
Clock	2.3 GHz	852 MHz	3.5 GHz	1.08 GHz
SMs, SPs		1, 192		24, 3072
Original	33.6	20.7	4.48	3.08
Warp Shuffle		14.4		0.210
Shared Memory		15.0		0.206

We see that on both systems and with both sum implementations, the new pro- grams do significantly better than the original. For the GK20A, the warp shuffle version runs in about 70% of the time of the original, and the shared memory version runs in about 72% of the time of the original. For the Titan X, the improvements are much more impressive: both versions run in less than 7% of the time of the original. Perhaps most striking is the fact that on the Titan X, the warp shuffle is, on average, slightly slower than the shared memory version.

6.12 CUDA trapezoidal rule III: blocks with more than one warp

Limiting ourselves to thread blocks with only 32 threads reduces the power and flex- ibility of our CUDA programs. For example, devices with compute capability 2.0 can have blocks with as many as 1024 threads or 32 warps, and CUDA provides a fast barrier that can be used to synchronize *all* the threads in a block. So if we limited ourselves to only 32 threads in a block, we wouldn't be using one of the most useful features of CUDA: the ability to efficiently synchronize large numbers of threads.

So what would a "block" sum look like if we allowed ourselves to use blocks with up to 1024 threads? We could use one of our existing warp sums to add the values computed by the threads in each warp. Then we would have as many as 1024/32 = 32 warp sums, and we could use one warp in the thread block to add the warp sums.

Since two threads belong to the same warp if their ranks in the block have the same quotient when divided by warpSize, to add the warp sums, we can use warp 0, the threads with ranks 0, 1, ..., 31 in the block.

6.12.1 syncthreads

We might try to use the following pseudocode for finding the sum of the values com- puted by all the threads in a block:

```
Each thread computes its contribution ;
Each warp adds its threads ' contributions ; Warp 0 in block adds warp sums ;
```

However, there's a race condition. Do you see it? When warp 0 tries to compute the total of the warp sums in the block, it doesn't know whether all the warps in the block have completed their sums. For example, suppose we have two warps, warp 0 and warp 1, each of which has 32 threads. Recall that the threads in a warp operate in SIMD fashion: no thread in

the warp proceeds to a new instruction until all the threads in the warp have completed (or skipped) the current instruction. But the threads in warp 0 can operate independently of the threads in warp 1. So if warp 0 finishes computing its sum before warp 1 computes its sum, warp 0 could try to add warp 1's sum to its sum before warp 1 has finished, and, in this case, the block sum could be incorrect.

So we must make sure that warp 0 doesn't start adding up the warp sums until all of the warps in the block are done. We can do this by using CUDA's fast barrier:

__device__ **void** __syncthreads (**void**);

This will cause the threads in the thread block to wait in the call until all of the threads have started the call. Using _syncthreads, we can modify our pseudocode so that the race condition is avoided:

```
Each thread computes its contribution ;
Each warp adds its threads ' contributions ;
__syncthreads ();
Warp 0 in block adds warp sums ;
```

Now warp 0 won't be able to add the warp sums until every warp in the block has completed its sum.

There are a couple of important caveats when we use _syncthreads. First, it's critical that *all* of the threads in the block execute the call. For example, if the block contains at least two threads, and our code includes something like this:

```
int my_x = threadIdx . x ;
if ( my_x < blockDim . x /2)
__syncthreads ();
my_x ++;
```

then only half the threads in the block will call _syncthreads, and these threads can't proceed until *all* the threads in the block have called _syncthreads. So they will wait forever for the other threads to call _syncthreads.

The second caveat is that _syncthreads only synchronizes the threads in a block. If a grid contains at least two blocks, and if all the threads in the grid call

_syncthreads then the threads in different blocks will continue to operate indepen- dently of each other. So we can't synchronize the threads in a general grid with

_syncthreads.12

6.12.2 More shared memory

If we try to implement the pseudocode in CUDA, we'll see that there's an important detail that the pseudocode doesn't show: after the call to _syncthreads, how does warp 0 obtain

access to the sums computed by the other warps? It can't use a warp shuffle and registers: the warp shuffles only allow a thread to read a register belonging to another thread when that thread belongs to the same warp, and, for the final warp sum we would like the threads in warp 0 to read registers belonging to threads in *other* warps.

You may have guessed that the solution is to use shared memory. If we use warp shuffles to compute the warp sums, we can just declare a shared array that can store up to 32 floats, and the thread with lane 0 in warp w can store its warp sum in element w of the array:

```
__shared__ float warp_sum_arr [ WARPSZ ];
int my_warp = threadIdx . x / warpSize ;
int my_lane = threadIdx . x % warpSize ;
// Threads calculate their contributions ;
. . .
float my_result = Warp_sum ( my_trap );
if ( my_lane == 0 ) warp_sum_arr [ my_warp ] = my_result ;
__syncthreads ();
// Warp 0 adds the sums in warp_sum_arr
. . .
```

6.12.3 Shared memory warp sums

If we're using shared memory instead of warp shuffles to compute the warp sums, we'll need enough shared memory for each warp in a thread block. Since shared variables are shared by *all* the threads in a thread block, we need an array large enough to hold the contributions of all of the threads to the sum. So we can declare an array with 1024 elements—the largest possible block size—and partition it among the warps:

```
// Make max thread block size available at compile time
# define MAX_BLKSZ 1024
. . .
___shared__ float thread_calcs [ MAX_BLKSZ ];
```

Now each warp will store its threads' calculations in a subarray of thread_calcs:

```
float * shared_vals = thread_calcs + my_warp * warpSize ;
```

In this setting a thread stores its contribution in the subarray referred to by shared_vals:

```
shared_vals [ my_lane ] = f ( my_x );
```

Now each warp can compute the sum of its threads' contributions by using our shared memory implementation that uses blocks with 32 threads:

```
float my_result = Shared_mem_sum ( shared_vals );
```

To continue we need to store the warp sums in locations that can be accessed by the threads in warp 0 in the block, and it might be tempting to try to make a subarray of thread_calcs do "double duty." For example, we might try to use the first 32 elements for both the contributions of the threads in warp 0, and the warp sums computed by the warps in the block. So if we have a block with 32 warps of 32 threads, warp w might store its sum in thread_calcs[w] for $w = 0, 1, 2, \ldots, 31$.

The problem with this approach is that we'll get another race condition. When can the other warps safely overwrite the elements in warp 0's block? After a warp has completed its call to Shared_mem_sum, it would need to wait until warp 0 has finished its call to Shared_mem_sum before writing to thread_calcs:

```
float my_result = Shared_mem_sum ( shared_vals );
__syncthreads ();
if ( my_lane == 0 ) thread_calcs [ my_warp ] = my_result .
```

This is all well and good, but warp 0 still can't proceed with the final call to Shared_mem_sum: it must wait until all the warps have written to thread_calcs. So we would need a *second* call to _syncthreads before warp 0 could proceed:

```
if ( my_lane == 0 ) thread_calcs [ my_warp ] = my_result .
__syncthreads ();
// It's safe for warp 0 to proceed . . .
if ( my_warp == 0 )
my_result = Shared_mem_sum ( thread_calcs );
```

Calls to _syncthreads are fast, but they're not free: every thread in the thread block will have to wait until all the threads in the block have called _syncthreads. So this can be costly. For example, if there are more threads in the block than there are SPs in an SM, the threads in the block won't all be able to execute simultaneously. So some threads will be delayed reaching the second call to _syncthreads, and all of the threads in the block will be delayed until the last thread is able to call _syncthreads. So we should only call _syncthreads() when we have to.

Alternatively, each warp could store its warp sum in the "first" element of its subarray:

```
float my_result = Shared_mem_sum ( shared_vals );
if ( my_lane == 0 ) shared_vals [0] = my_result ;
__syncthreads ();
. . .
```

It might at first appear that this would result in a race condition when the thread with lane 0 attempts to update shared_vals, but the update is OK. Can you explain why?

6.12.4 Shared memory banks

However, this implementation may not be as fast as possible. The reason has to do with details of the design of shared memory: Nvidia divides the shared memory on an SM into 32 "banks" (16 for GPUs with compute capability < 2.0). This is done so that the 32 threads in a warp can simultaneously access shared memory: the threads in a warp can simultaneously access shared memory when each thread accesses a different bank.

Table 6.9 Shared memory banks: Columns are memory banks. The entries in the body of the table show subscripts of elements of thread_calcs.

	Bank					
	0	**1**	**2**	**...**	**30**	**31**
Subscripts	0	1	2	...	30	31
	32	33	34	...	62	63
	64	65	66	...	94	95
	96	97	98	...	126	127
	.	.	.	⋱	.	.
	992	993	994	...	1022	1023

Table 6.9 illustrates the organization of thread_calcs. In the table, the columns are banks, and the rows show the subscripts of consecutive elements of thread_calcs. So the 32 threads in a warp can simultaneously access the 32 elements in any one of the rows, or, more generally, if each thread access is to a different column.

When two or more threads access different elements in a single bank (or column in the table), then those accesses must be serialized. So the problem with our approach to saving the warp sums in elements 0, 32, 64, ..., 992 is that these are all in the same bank. So when we try to execute them, the GPU will serialize access, e.g., element 0 will be written, then element 32, then element 64, etc. So the writes will take something like 32 times as long as it would if the 32 elements were stored in different banks, e.g., a row of the table.

The details of bank access are a little complicated and some of the details depend on the compute capability, but the main points are

- If each thread in a warp accesses a different bank, the accesses can happen simultaneously.
- If multiple threads access different memory locations in a single bank, the accesses must be serialized.
- If multiple threads read the same memory location in a bank, the value read is broadcast to the reading threads, and the reads are simultaneous.

The CUDA programming Guide [11] provides full details.

Thus we could exploit the use of the shared memory banks if we stored the results in a contiguous subarray of shared memory. Since each thread block can use at least 16 Kbytes of

shared memory, and our "current" definition of shared_vals only uses at most 1024 floats or 4 Kbytes of shared memory, there is plenty of shared memory available for storing 32 more floats.

So if we're using shared memory warp sums, a simple solution is to declare *two* arrays of shared memory: one for storing the computations made by each thread, and another for storing the warp sums.

```
__shared__ float  thread_calcs [ MAX_BLKSZ ];
__shared__ float  warp_sum_arr [ WARPSZ ];
float    shared_vals = thread_calcs + my_warp warpSize ;
. . .
float  my_result =  Shared_mem_sum ( shared_vals );
if  ( my_lane == 0 )  warp_sum_arr [ my_warp ] = my_result ;
__syncthreads ();
. . .
```

6.6.1 Finishing up

The remaining codes for the warp sum kernel and the shared memory sum kernel are very similar. First warp 0 computes the sum of the elements in warp_sum_arr. Then thread 0 in the block adds the block sum into the total across all the threads in the grid using atomicAdd. Here's the code for the shared memory sum:

```
if  ( my_warp == 0 ) {
    if  ( threadIdx . x >= blockDim . x / warpSize )
        warp_sum_arr [ threadIdx . x ]  =  0.0;
    blk_result =  Shared_mem_sum ( warp_sum_arr );
}
if  ( threadIdx . x == 0 )  atomicAdd ( trap_p ,  blk_result );
```

In the test threadIdx.x > blockDim.x/warpSize we're checking to see if there are fewer than 32 warps in the block. If there are, then the final elements in warp_sum_arr won't have been initialized. For example, if there are 256 warps in the block, then

blockDim . x / warpSize = 256/32 = 8

So there are only 8 warps in a block and we'll have only initialized elements 0, 1,..., 7 of warp_sum_arr. But the warp sum function expects 32 values. So for the threads with threadIdx.x >= 8, we assign

```
warp_sum_arr [ threadIdx . x ] = 0.0;
```

For the sake of completeness, Program 6.15 shows the kernel that uses shared memory. The main differences between this kernel and the kernel that uses warp shuffles are that the declaration of the first shared array isn't needed in the warp shuffle version, and, of course, the warp shuffle version calls Warp_sum instead of Shared_mem_sum.

```
1  __global__ void Dev_trap(
2        const float a          /* in  */,
3        const float b          /* in  */,
4        const float h          /* in  */,
5        const int   n          /* in  */,
6        float *     trap_p     /* out */) {
7     __shared__ float thread_calcs[MAX_BLKSZ];
8     __shared__ float warp_sum_arr[WARPSZ];
9     int my_i = blockDim.x * blockIdx.x + threadIdx.x;
10    int my_warp = threadIdx.x / warpSize;
11    int my_lane = threadIdx.x % warpSize;
12    float * shared_vals = thread_calcs + my_warp*warpSize;
13    float blk_result = 0.0;
14
15    shared_vals[my_lane] = 0.0f;
16    if (0 < my_i && my_i < n) {
17       float my_x = a + my_i*h;
18       shared_vals[my_lane] = f(my_x);
19    }
20
21    float my_result = Shared_mem_sum(shared_vals);
22    if (my_lane == 0) warp_sum_arr[my_warp] = my_result;
23    __syncthreads();
24
25    if (my_warp == 0) {
26          if (threadIdx.x >= blockDim.x/warpSize)
27             warp_sum_arr[threadIdx.x] = 0.0;
28          blk_result = Shared_mem_sum(warp_sum_arr);
29    }
30
31    if (threadIdx.x == 0) atomicAdd(trap_p, blk_result);
32 } /* Dev_trap */
```

Program 6.15: CUDA kernel implementing trapezoidal rule and using shared memory. This version can use large thread blocks.

MODULE 4

CHAPTER 5: Shared-memory programming with OpenMP

The "MP" in OpenMP stands for "multiprocessing," a term that is synonymous with shared-memory MIMD computing. Thus OpenMP is designed for systems in which each thread or process can potentially have access to all available memory, and when we're programming with OpenMP, we view our system as a collection of autonomous cores or CPUs, all of which have access to main memory, as in Fig. 5.1.

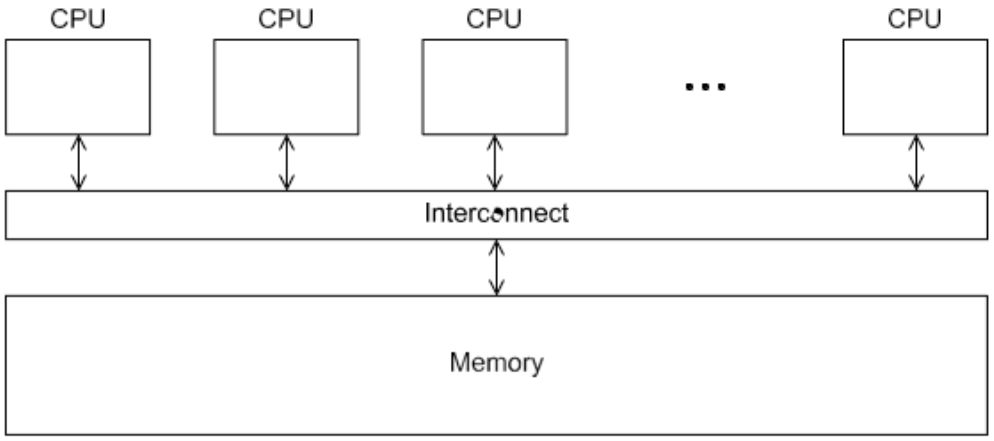


FIGURE 5.1

A shared-memory system.

OpenMP allows the programmer to simply state that a block of code should be executed in parallel, and the precise determination of the tasks and which thread should execute them is left to the compiler and the run-time system.

OpenMP requires compiler support for some operations, and hence it's entirely possible that you may run across a C compiler that can't compile OpenMP programs into parallel programs.

OpenMP allows the compiler and run-time system to determine some of the details of thread behavior, so it can be simpler to code some parallel behaviors using OpenMP. The cost is that some low-level thread interactions can be more difficult to program.

OpenMP was developed by a group of programmers and computer scientists who believed that writing large-scale high-performance programs using APIs, such as Pthreads, was too difficult, and they defined the OpenMP specification so that shared-memory programs could be developed at a higher level.

5.1 Getting started

OpenMP provides what's known as a "directives-based" shared-memory API. In C and C++, this means that there are special preprocessor instructions known as pragmas. Pragmas are typically added to a system to allow behaviors that aren't part of the basic C specification. Compilers that don't support the pragmas are free to ignore them. This allows a program that uses the pragmas to run on platforms that don't support them. So, in principle, if you have a carefully written OpenMP program, it can be compiled and run on any system with a C compiler, regardless of whether the compiler supports OpenMP. If OpenMP is not supported, then the directives are simply ignored and the code will execute sequentially.

Pragmas in C and C++ start with

pragma

As usual, we put the pound sign, #, in column 1, and like other preprocessor directives, we shift the remainder of the directive so that it is aligned with the rest of the code. Pragmas (like all preprocessor directives) are, by default, one line in length, so if a pragma won't fit on a single line, the newline needs to be "escaped"—that is, pre- ceded by a backslash \. The details of what follows the **#pragma** depend entirely on which extensions are being used.

Let's take a look at a *very* simple example, a "hello, world" program that uses OpenMP. (See Program 5.1.)

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

void Hello(void);     /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n",
          my_rank, thread_count);

}  /* Hello */
```

Program 5.1: A "hello, world" program that uses OpenMP.

5.1.1 Compiling and running OpenMP programs

To compile this with gcc we need to include the −fopenmp option[1]:

```
$ gcc −g −Wall −fopenmp −o omp_hello omp_hello.c
```

To run the program, we specify the number of threads on the command line. For example, we might run the program with four threads and type

```
$ ./ omp_hello 4
```

If we do this, the output might be

```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

However, it should be noted that the threads are competing for access to stdout, so there's no guarantee that the output will appear in thread-rank order. For example, the output might also be

```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

Or

```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

or any other permutation of the thread ranks.

If we want to run the program with just one thread, we can type

```
$ ./ omp_hello 1
```

and we would get the output

```
Hello from thread 0 of 1
```

5.1.2 The program

Let's take a look at the source code. In addition to a collection of directives, OpenMP consists of a library of functions and macros, so we usually need to include a header file with prototypes and macro definitions. The OpenMP header file is omp.h, and we include it in Line3.

In our Pthreads programs, we specified the number of threads on the command line. We'll also usually do this with our OpenMP programs. In Line 9 we therefore use the strtol function from stdlib.h to get the number of threads. Recall that the syntax of this function is

```
long strtol (
```

```
const char *   number_p     /* in  */,
char **        end_p                /* out */,
int  base                   /* in  */);
```

The first argument is a string—in our example, it's the command-line argument, a string—and the last argument is the numeric base in which the string is represented— in our example, it's base 10. We won't make use of the second argument, so we'll just pass in a NULL pointer. The return value is the command-line argument converted to a C **long int** .

If you've done a little C programming, there's nothing really new up to this point. When we start the program from the command line, the operating system starts a single-threaded process, and the process executes the code in the main function. How- ever, things get interesting in Line 11. This is our first OpenMP directive, and we're using it to specify that the program should start some threads. Each thread should execute the Hello function, and when the threads return from the call to Hello, they should be terminated, and the process should then terminate when it executes the **return** statement.

That's a lot of bang for the buck (or code). If you studied the Pthreads chapter, you'll recall that we had to write a lot of code to achieve something similar: we needed to allocate storage for a special struct for each thread, we used a **for** loop to start all the threads, and we used another **for** loop to terminate the threads. Thus it's immediately evident that OpenMP provides a higher-level abstraction than Pthreads provides.

We've already seen that pragmas in C and C++ start with

#pragma

OpenMP pragmas always begin with

#pragma omp

Our first directive is a parallel directive, and, as you might have guessed, it speci- fies that the **structured block** of code that follows should be executed by multiple threads. A structured block is a C statement or a compound C statement with one point of entry and one point of exit, although calls to the function exit are allowed. This definition simply prohibits code that branches into or out of the middle of the structured block.

Recall that **thread** is short for *thread of execution*. The name is meant to suggest a sequence of statements executed by a program. Threads are typically started or **forked** by a process, and they share most of the resources of the process that starts them—for example, access to stdin and stdout—but each thread has its own stack and program counter. When a thread completes execution, it **joins** the process that started it. This terminology comes from diagrams that show threads as directed lines. (See Fig. 5.2.)

At its most basic the parallel directive is simply

#pragma omp parallel

and the number of threads that run the following structured block of code will be determined by the run-time system. The algorithm used is fairly complicated; see the OpenMP Standard [47] for details. However, if there are no other threads started, the system will typically run one thread on each available core.

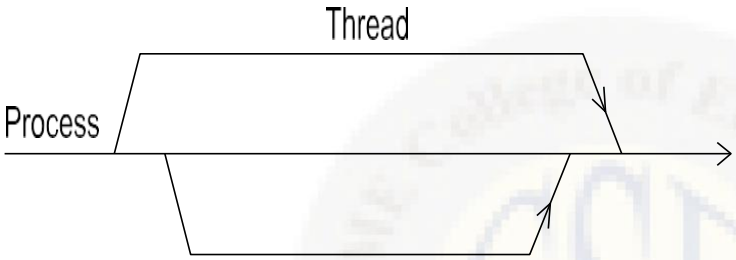


Fig 5.2: A process forking and joining two threads.

As we noted earlier, we'll usually specify the number of threads on the command line, so we'll modify our parallel directives with the num_threads *clause*. A clause in OpenMP is just some text that modifies a directive. The num_threads clause can be added to a parallel directive. It allows the programmer to specify the number of threads that should execute the following block:

#pragma omp parallel num_threads (thread_count)

It should be noted that there may be system-defined limitations on the number of threads that a program can start. The OpenMP Standard doesn't guarantee that this will actually start thread_count threads. However, most current systems can start hun- dreds or even thousands of threads, so unless we're trying to start *a lot* of threads, we will almost always get the desired number of threads.

What actually happens when the program gets to the parallel directive? Prior to the parallel directive, the program is using a single thread, the process started when the program started execution. When the program reaches the parallel direc- tive, the original thread continues executing and thread_count − 1 additional threads

are started. In OpenMP parlance, the collection of threads executing the parallel

block—the original thread and the new threads—is called a team. OpenMP thread terminology includes the following:

- master: the first thread of execution, or *thread 0*.
- parent: thread that encountered a parallel directive and started a team of threads. In many cases, the parent is also the master thread.

child: each thread started by the parent is considered a *child* thread.

Each thread in the team executes the block following the directive, so in our example, each thread calls the Hello function.

When the block of code is completed—in our example, when the threads return from the call to Hello—there's an implicit barrier. This means that a thread that has completed the block of code will wait for all the other threads in the team to complete the block—in our example, a thread that has completed the call to Hello will wait for all the other threads in the team to return. When all the threads have completed the block, the child threads will terminate and the parent thread will continue executing the code that follows the block. In our example, the parent thread will execute the return statement in Line 14, and the program will terminate.

Since each thread has its own stack, a thread executing the Hello function will create its own private, local variables in the function. In our example, when the function is called, each thread will get its rank or ID and the number of threads in the team by calling the OpenMP functions omp_get_thread_num and omp_get_num_threads, respectively. The rank or ID of a thread is an int that is in the

range 0, 1,..., thread_count − 1. The syntax for these functions is

int omp_get_thread_num (void);

int omp_get_num_threads (void);

Since stdout is shared among the threads, each thread can execute the printf state- ment, printing its rank and the number of threads.

As we noted earlier, there is no scheduling of access to stdout, so the actual order in which the threads print their results is nondeterministic.

5.1.3 Error checking

To make the code more compact and more readable, our program doesn't do any error checking. Of course, this is dangerous, and, in practice, it's a *very* good idea—one might even say mandatory—to try to anticipate errors and check for them. In this example, we should definitely check for the presence of a command-line argument, and, if there is one, after the call to strtol, we should check that the value is positive. We might also check that the number of threads actually created by the parallel directive is the same as thread_count, but in this simple example, this isn't crucial.

A second source of potential problems is the compiler. If the compiler doesn't support OpenMP, it will just ignore the parallel directive. However, the attempt to include omp.h and the calls to omp_get_thread_num and omp_get_num_threads *will* cause errors. To handle these problems, we can check whether the preprocessor macro_OPENMP is defined. If this is defined, we can include omp.h and make the calls to the OpenMP functions. We might make the modifications that follow to our program.

Instead of simply including omp.h:

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

we can check for the definition of _OPENMP before trying to include it:

```
# ifdef _OPENMP
#    include <omp . h>
# endif
```

Also, instead of just calling the OpenMP functions, we can first check whether _OPENMP is defined:

```
#  ifdef _OPENMP
      int  my_rank =  omp_get_thread_num ();
      int  thread_count = omp_get_num_threads ();
 #else
      int  my_rank = 0 ;
      int  thread_count = 1 ;
#   endif
```

Here, if OpenMP isn't available, we assume that the Hello function will be single- threaded. Thus the single thread's rank will be 0, and the number of threads will be 1.

The book's website contains the source for a version of this program that makes these checks. To make our code as clear as possible, we'll usually show little, if any, error checking in the code displayed in the text. We'll also assume that OpenMP is available and supported by the compiler.

5.2 The trapezoidal rule

Let's take a look at a somewhat more useful (and more complicated) example: the trapezoidal rule for estimating the area under a curve. Recall that if $y\ f\ (x)$ is a reasonably nice function, and $a \lessapprox b$ are real numbers, then we can estimate the area between the graph of $f\ (x)$, the vertical lines $x\ \ a$ and $x\ \ b$, and the x-axis by dividing the interval a, b into n subintervals and approximating the area over each subinterval by the area of a trapezoid. See Fig. 5.3.

Also recall that if each subinterval has the same length and if we define $h =$

$(b - a)/n$, $x_i = a + ih$, $i = 0, 1, \ldots, n$, then our approximation will be

$$h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2].$$

Thus we can implement a serial algorithm using the following code:

```
/* 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 ;
```

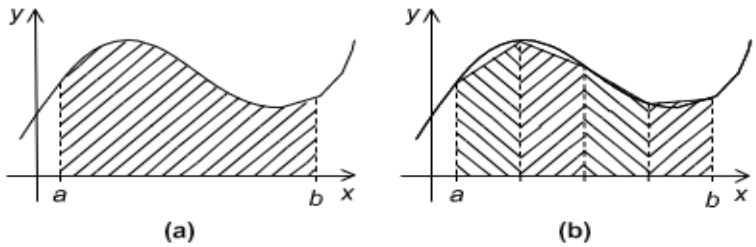


FIGURE 5.3

The trapezoidal rule.

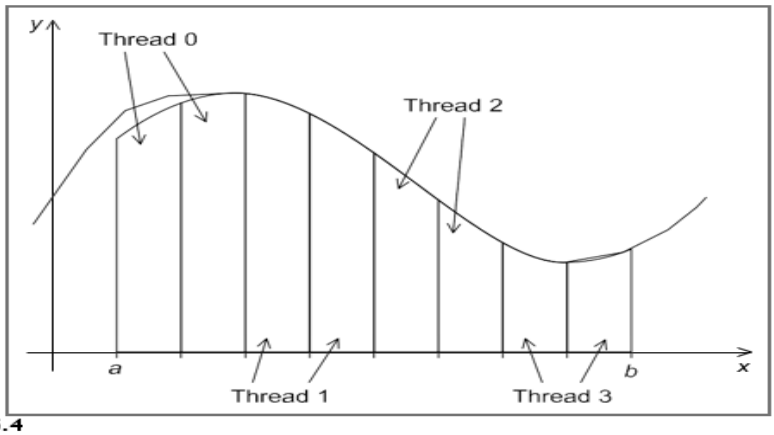


FIGURE 5.4

Assignment of trapezoids to threads.

5.2.1 A first OpenMP version

Recall that we applied Foster's parallel program design methodology to the trape- zoidal rule as described in the following list :

1. We identified two types of jobs:

   a.Computation of the areas of individual trapezoids, and

   b.Adding the areas of trapezoids.

2.There is no communication among the jobs in the first collection, but each job in the first collection communicates with job 1b.

3.We assumed that there would be many more trapezoids than cores, so we aggregated jobs by assigning a contiguous block of trapezoids to each thread (and a single thread to each core).[2] Effectively, this partitioned the interval $[a, b]$ into larger subintervals, and each thread simply applied the serial trapezoidal rule to its subinterval. See Fig. 5.4.

We aren't quite done, however, since we still need to add up the threads' results. An obvious solution is to use a shared variable for the sum of all the threads' results, and each thread can add its (private) result into the shared variable. We would like to have each thread execute a statement that looks something like

global_result += my_result ;

However, as we've already seen, this can result in an erroneous value for global_result—if two (or more) threads attempt to simultaneously execute this state- ment, the result will be unpredictable. For example, suppose that global_result has been initialized to 0, thread 0 has computed my_result = 1, and thread 1 has com- puted my_result = 2. Furthermore, suppose that the threads execute the statement global_result += my_result according to the following timetable:

Time	Thread 0	Thread 1
0	global_result = 0to register	finish my_result
1	my_result = 1to register	global_result = 0to register
2	add my_resultto global_result	my_result = 2to register
3	store global_result = 1	add my_resultto global_result
4		store global_result = 2

We see that the value computed by thread 0 (my_result = 1) is overwritten by thread 1.

Of course, the actual sequence of events might be different, but unless one thread finishes the computation global_result += my_result before the other starts, the result will be incorrect. Recall that this is an example of a **race condition**: multiple threads are attempting to access a shared resource, at least one of the accesses is an update, and the accesses can result in an error. Also recall that the code that causes the race condition, global_result += my_result, is called a **critical section**. A critical section is code executed by multiple threads that updates a shared resource, and the shared resource can only be updated by one thread at a time.

We therefore need some mechanism to make sure that once one thread has started executing global_result += my_result, no other thread can start executing this code until the first thread has finished. In Pthreads we used mutexes or semaphores. In OpenMP we can use the critical directive

#pragma omp critical

global_result += my_result ;

This directive tells the compiler that the system needs to arrange for the threads to have **mutually exclusive** access to the following structured block of code.[3] That is, only one thread can execute the following structured block at a time. The code for this version is shown in Program 5.2. We've omitted any error checking. We've also omitted code for the function *f (x)*.

In the main function, prior to Line 17, the code is single-threaded, and it simply gets the number of threads and the input (*a*, *b*, and *n*). In Line 17 the parallel directive specifies that the Trap function should be executed by thread_count threads. After returning from the call to Trap, any new threads that were started by the parallel directive are terminated, and the program resumes execution with only one thread. The one thread prints the result and terminates.

In the Trap function, each thread gets its rank and the total number of threads in the team started by the parallel directive. Then each thread determines the following:

1.The length of the bases of the trapezoids (Line 33),

2.The number of trapezoids assigned to each thread (Line 34),

Program 5.2: First OpenMP trapezoidal rule program

3.The left and right endpoints of its interval (Lines 35 and 36, respectively)

4.Its contribution to global_result (Lines 37–42).

The threads finish by adding in their individual results to global_result in Lines 44–45.

We use the prefix local_ for some variables to emphasize that their values may differ from the values of corresponding variables in the main function—for example, local_a may differ from a, although it is the *thread's* left endpoint.

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

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
    /* We'll store our result in global_result: */
    double global_result = 0.0;
    double a, b;     /* Left and right endpoints     */
    int n;           /* Total number of trapezoids */
    int thread_count;

    thread_count = strtol(argv[1], NULL, 10);
    printf("Enter a, b, and n\n");
    scanf("%lf %lf %d", &a, &b, &n);
#   pragma omp parallel num_threads(thread_count)
    Trap(a, b, n, &global_result);

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.14e\n",
        a, b, global_result);
    return 0;
}  /* main */

void Trap(double a, double b, int n, double* global_result_p) {
    double h, x, my_result; local_a
    double , local_b; local_n;
    int i,
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    h = (b-a)/n;
    local_n = n/thread_count;
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    my_result = (f(local_a) + f(local_b))/2.0;
    for (i = 1; i <= local_n-1; i++) {
       x = local_a + i*h;
       my_result += f(x);
    }
    my_result = my_result*h;

#   pragma omp critical
    *global_result_p += my_result;
}  /* Trap */
```

Notice that unless *n* is evenly divisible by thread_count, we'll use fewer than *n* trapezoids for global_result. For example, if *n* 14 and thread_count = 4, each thread will compute

local_n = n/thread_count = 14/4 = 3.

Thus each thread will only use 3 trapezoids, and global_result will be computed with 4*3=12 trapezoids instead of the requested 14. So in the error checking (which isn't shown), we check that *n* is evenly divisible by thread_count by doing something like this:

```
if ( n % thread_count != 0 ) {
fprintf ( stderr ,
" n must be evenly divisible by thread_count \ n " );
exit (0);
}
```

Since each thread is assigned a block of local_n trapezoids, the length of each thread's interval will be local_n∗h, so the left endpoints will be

thread 0: a + 0*local_n*h thread 1: a + 1*local_n*h thread 2: a + 2*local_h*h

...

So in Line 35, we assign

local_a = a + my_rank ∗ local_n ∗h ;

Furthermore, since the length of each thread's interval will be local_n h, its right endpoint will just be

local_b = local_a + local_n ∗h ;

5.3 Scope of variables

In serial programming, the *scope* of a variable consists of those parts of a program in which the variable can be used. For example, a variable declared at the beginning of a C function has "function-wide" scope, that is, it can only be accessed in the body of the function. On the other hand, a variable declared at the beginning of a . c file but outside any function has "file-wide" scope, that is, any function in the file in which the variable is declared can access the variable. In OpenMP, the **scope** of a variable refers to the set of threads that can access the variable in a parallel block. A variable that can be accessed by all the threads in the team has **shared** scope, while a variable that can only be accessed by a single thread has **private** scope.

In the "hello, world" program, the variables used by each thread (my_rank and thread_count) were declared in the Hello function, which is called inside the parallel block. Consequently, the variables used by each thread are allocated from the thread's (private) stack, and hence all of the variables have private scope. This is *almost* the case in the trapezoidal rule program; since the parallel block is just a function call, all of the variables used by each thread in the Trap function are allocated from the thread's stack.

However, the variables that are declared in the main function (a, b, n, global_result, and thread_count) are all accessible to all the threads in the team started by the parallel directive. Hence, the *default* scope for variables declared before a parallel block is shared. In fact, we've made implicit use of this: each thread in the team gets the values of a, b, and n from the call to Trap. Since this call takes place in the parallel block, it's essential that each thread has access to a, b, and n when their values are copied into the corresponding formal arguments.

Furthermore, in the Trap function, although global_result_p is a private variable, it refers to the variable global_result which was declared in main before the parallel directive, and the value of global_result is used to store the result that's printed out after the parallel block. Thus in the code

∗ global_result_p += my_result ;

it's essential that ∗global_result_p have shared scope. If it were private to each thread, there would be no need for the critical directive. Furthermore, if it were private, we would have a hard time determining the value of global_result in main after completion of the parallel block.

5.4 The reduction clause

If we developed a serial implementation of the trapezoidal rule, we'd probably use a slightly different function prototype. Rather than

```
void Trap(
double a,
double b,
int n,
double* global_result_p);
```

we would probably define

```
double Trap(double a, double b, int n);
```

and our function call would be

```
global_result = Trap(a, b, n);
```

This is somewhat easier to understand and probably more attractive to all but the most fanatical believers in pointers.

We resorted to the pointer version, because we needed to add each thread's local calculation to get global_result. However, we might prefer the following function prototype:

```
double Local_trap(double a, double b, int n);
```

With this prototype, the body of Local_trap would be the same as the Trap function in Program

5.2, except that there would be no critical section. Rather, each thread would return its part of the calculation, the final value of its my_result variable. If we made this change, we might try modifying our parallel block so that it looks like this:

```
global_result = 0.0;
#pragma omp parallel num_threads(thread_count)
{
#pragma omp critical
global_result += Local_trap(double a, double b, int n);
}
```

Can you see a problem with this code? It should give the correct result. However, since we've specified that the critical section is

```
global_result += Local_trap(double a, double b, int n);
```

the call to Local_trap can only be executed by one thread at a time, and, effectively, we're forcing the threads to execute the trapezoidal rule sequentially. If we check the run-time of this version, it may actually be *slower* with multiple threads than one thread (see Exercise 5.3).
We can avoid this problem by declaring a private variable inside the parallel block and moving the critical section after the function call:

```
global_result = 0.0;
#                    pragma omp parallel num_threads(thread_count)
{
double my_result = 0.0;                 *       * / private /
    my_result += Local_trap(double a, double b, int n); #    pragma
omp critical
global_result += my_result;
}
```

Now the call to Local_trap is outside the critical section, and the threads can execute their calls simultaneously. Furthermore, since my_result is declared in the parallel block, it's private, and before the critical section each thread will store its part of the calculation in its my_result variable.
OpenMP provides a cleaner alternative that also avoids serializing execution of Local_trap: we can specify that global_result is a *reduction* variable. A reduction operator is an associative binary operation (such as addition or multiplication), and a reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands to get a single result. Furthermore, all of the intermediate results of the operation should be stored in the same variable: the reduction variable.

For example, if A is an array of n ints, the computation

```
int sum = 0;
for (i = 0; i < n; i++)
sum += A[i];
```

is a reduction in which the reduction operator is addition.

In OpenMP it may be possible to specify that the result of a reduction is a reduc- tion variable. To do this, a reduction clause can be added to a parallel directive. In our example, we can modify the code as follows:

```
global_result = 0.0;
#pragma omp parallel num_threads(thread_count) \
  reduction(+: global_result)
global_result += Local_trap(double a, double b, int n);
```

First note that the parallel directive is two lines long. Recall that C preprocessor directives are, by default, only one line long, so we need to "escape" the newline character by putting a backslash (\) immediately before it.

The code specifies that global_result is a reduction variable, and the plus sign ("+") indicates that the reduction operator is addition. Effectively, OpenMP creates a private variable for each thread, and the run-time system stores each thread's result in this private variable. OpenMP also creates a critical section, and the values stored in the private variables are added in this critical section. Thus the calls to Local_trap can take place in parallel.

The syntax of the reduction clause is

```
reduction(<operator>: <variable list>)
```

In C, operator can be any one of the operators +, , , &, ⊥, ^, &&, ||. You may wonder whether the use of subtraction is problematic, though, since subtraction isn't

associative or commutative. For example, the serial code

```
result = 0;
for (i = 1; i <= 4; i++)
result −= i;
```

stores the value −10 in result. However, if we split the iterations among two threads, with thread 0 subtracting 1 and 2 and thread 1 subtracting 3 and 4, then thread 0 will compute 3 and thread 1 will compute 7. This results in an incorrect calculation, 3 − (−7). Luckily, the OpenMP standard states that partial results of a sub- traction reduction are *added* to form the final value, so the reduction will work as intended.

It should also be noted that if a reduction variable is a float or a double, the results may differ slightly when different numbers of threads are used. This is due to the fact that floating point arithmetic isn't associative. For example, if *a*, *b*, and *c* are floats, then *(a + b) + c* may not be exactly equal to *a + (b + c)*. See Exercise 5.5.

When a variable is included in a reduction clause, the variable itself is shared. However, a private variable is created for each thread in the team. In the parallel block each time a thread executes a statement involving the variable, it uses the private variable. When the parallel block ends, the values in the private variables are combined into the shared variable. Thus our latest version of the code

```
global_result = 0.0;
#pragma omp parallel num_threads(thread_count) \
reduction(+: global_result)
global_result += Local_trap(double a, double b, int n);
```

effectively executes code that is identical to our previous version:

```
global_result = 0.0;
#pragma omp parallel num_threads(thread_count)
{
double my_result = 0.0;
      my_result += Local_trap(double a, double b, int n); #
omp critical
global_result += my_result;
}
```

5.5 The parallel for directive

As an alternative to our explicit parallelization of the trapezoidal rule, OpenMP provides the parallel for directive. Using it, we can parallelize the serial trapezoidal rule

```
h = (b a)/ n;
approx = (f(a) + f(b))/2.0; for (i = 1; i <= n−1; i++) approx += f(a + i∗h);
approx = h∗approx;
```

by simply placing a directive immediately before the for loop:

```
h = (b a)/ n;
approx = (f(a) + f(b))/2.0;
```

```
#pragma omp parallel for num_threads(thread_count) \
reduction(+: approx)
for (i = 1; i <= n-1; i++)
approx += f(a + i*h);
approx = h*approx;
```

Like the parallel directive, the parallel for directive forks a team of threads to execute the following structured block. However, the structured block following the parallel for directive must be a for loop. Furthermore, with the parallel for directive the system parallelizes the for loop by dividing the iterations of the loop among the threads. So the parallel for directive is therefore very different from the parallel directive, because in a block that is preceded by a parallel directive, in general, the work must be divided among the threads by the threads themselves.

In a for loop that has been parallelized with a parallel for directive, the default partitioning of the iterations among the threads is up to the system. However, most systems use roughly a block partitioning, that is, if there are *m* iterations, then roughly the first *m*/thread_count are assigned to thread 0, the next *m*/thread_count are assigned to thread 1, and so on.

Note that it was essential that we made approx a reduction variable. If we hadn't, it would have been an ordinary shared variable, and the body of the loop

```
approx += f(a + i*h);
```

would be an unprotected critical section, leading to inconsistent values of approx.

However, speaking of scope, the default scope for all variables in a parallel directive is shared, but in our parallel for if the loop variable i were shared, the variable update, i++, would also be an unprotected critical section. Hence, in a loop that is parallelized with a parallel for directive the default scope of the loop variable is *private*; in our code, each thread in the team has its own copy of i.

5.5.1 Caveats

This is truly wonderful: It may be possible to parallelize a serial program that consists of one large for loop by just adding a single parallel for directive. It may be possible to incrementally parallelize a serial program that has many for loops by successively placing parallel for directives before each loop.

However, things may not be quite as rosy as they seem. There are several caveats associated with the use of the parallel for directive. First, OpenMP will only parallelize for loops—it won't parallelize while loops or do−while loops directly. This may not seem to be too much of a limitation, since any code that uses a while loop or a do−while loop can be converted to equivalent code that uses a for loop instead. However, OpenMP will only parallelize for loops for which the number of iterations can be determined:

- from the for statement itself (that is, the code for (. . . ; . . . ; . . .)), and
- prior to execution of the loop. For example, the "infinite loop"

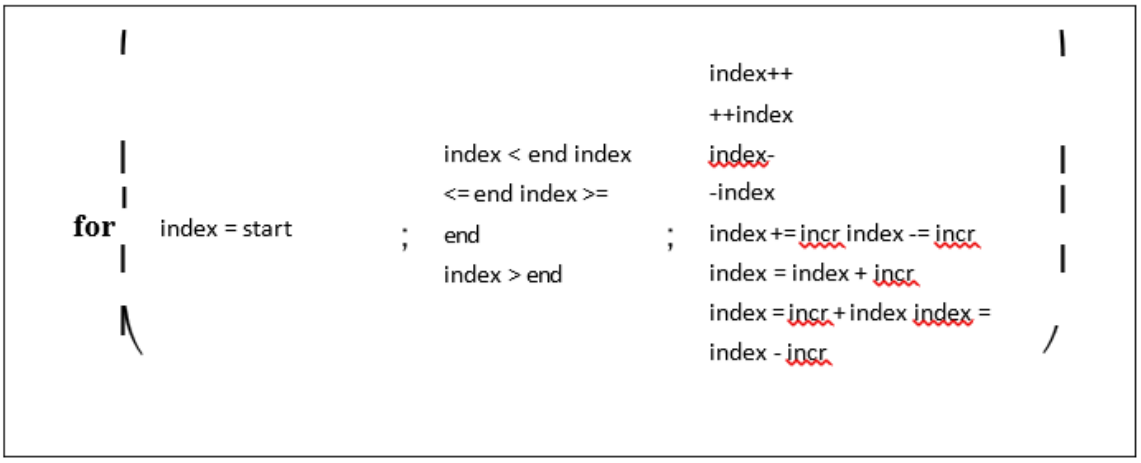
```
for ( ; ; ) {
. . .
}
```

cannot be parallelized. Similarly, the loop

```
for (i = 0; i < n; i++) {
if ( . . . ) break;
. . .
}
```

cannot be parallelized, since the number of iterations can't be determined from the for statement alone. This for loop is also not a structured block, since the break adds another point of exit from the loop.

In fact, OpenMP will only parallelize for loops that are in canonical form. Loops in canonical form take one of the forms shown in Program 5.3. The variables and expressions in this template are subject to some fairly obvious restrictions:



Program 5.3: Legal forms for parallelizable **for** statements.

- The variable index must have integer or pointer type (e.g., it can't be a **float**).
- The expressions start, end, and incr must have a compatible type. For example, if index is a pointer, then incr must have integer type.
- The expressions start, end, and incr must not change during execution of the loop.
- During execution of the loop, the variable index can only be modified by the "increment expression" in the **for** statement.

These restrictions allow the run-time system to determine the number of iterations prior to execution of the loop.

The sole exception to the rule that the run-time system must be able to determine the number of iterations prior to execution is that there can be a call to exit in the body of the loop.

5.5.2 Data dependences

If a **for** loop fails to satisfy one of the rules outlined in the preceding section, the compiler will simply reject it. For example, suppose we try to compile a program with the following linear search function:

```
int Linear_search(int key, int A[], int n) {
int i;
/thread_count *is global /          *
#pragma omp parallel for num_threads(thread_count)
for (i = 0; i < n; i++)
if (A[i] == key) return i;
return -1;                  /* key not in list */
}
```

The gcc compiler reports:

```
Line 6: error: invalid exit from OpenMP structured block
```

A more insidious problem occurs in loops in which the computation in one it- eration depends on the results of one or more previous iterations. As an example, consider the following code, which computes the first *n* Fibonacci numbers:

```
fibo[0] = fibo[1] = 1;
for (i = 2; i < n; i++)
fibo[i] = fibo[i-1] + fibo[i-2];
```

Although we may be suspicious that something isn't quite right, let's try parallelizing the for loop with a parallel for directive:

```
fibo[0] = fibo[1] = 1;
#pragma omp parallel for num_threads(thread_count)
for (i = 2; i < n; i++)
fibo[i] = fibo[i-1] + fibo[i-2];
```

The compiler will create an executable without complaint. However, if we try running it with more than one thread, we may find that the results are, at best, unpredictable. For example, on one of our systems (if we try using two threads to compute the first 10 Fibonacci numbers), we sometimes get

1 1 2 3 5 8 13 21 34 55,

which is correct. However, we also occasionally get

1 1 2 3 5 8 0 0 0 0.

What happened? It appears that the run-time system assigned the computation of fibo[2],

fibo[3], fibo[4], and fibo[5] to one thread, while fibo[6], fibo[7], fibo[8], and fibo[9] were assigned to the other. (Remember, the loop starts with i = 2.) In some runs of the program, everything is fine, because the thread that was assigned fibo[2], fibo[3], fibo[4], and fibo[5] finishes its computations before the other thread starts. However, in other runs, the first thread has evidently not computed fibo[4] and fibo[5] when the second computes fibo[6]. It appears that the system has initialized the entries in fibo to 0, and the second thread is using the values fibo[4] = 0 and fibo[5] = 0 to compute fibo[6]. It then goes on to use fibo[5] = 0 and fibo[6] = 0 to compute fibo[7], and so on.

We see two important points here:

1. OpenMP compilers don't check for dependences among iterations in a loop that's being parallelized with a parallel for directive. It's up to us, the programmers, to identify these dependences.
2. A loop in which the results of one or more iterations depend on other iterations *cannot*, in general, be correctly parallelized by OpenMP without using features such as the Tasking API. (See Section 5.10).

The dependence of the computation of fibo[6] on the computation of fibo[5] is called a data dependence. Since the value of fibo[5] is calculated in one iteration, and the result is used in a subsequent iteration, the dependence is sometimes called a loop-carried dependence.

5.5.3 Finding loop-carried dependences

Perhaps the first thing to observe is that when we're attempting to use a parallel for directive, we only need to worry about loop-carried dependences. We don't need to worry about more general data dependences. For example, in the loop

```
for (i = 0; i < n; i++) {
x[i] = a + i h;          *
y[i] = exp(x[i]);
}
```

there is a data dependence between Lines 2 and 3. However, there is no problem with the parallelization

```
#pragma omp parallel for num_threads(thread_count)
for (i = 0; i < n; i++) {
x[i] = a + i h;          *
y[i] = exp(x[i]);
}
```

since the computation of x[i] and its subsequent use will always be assigned to the same thread.

Also observe that at least one of the statements must write or update the variable in order for the statements to represent a dependence, so to detect a loop-carried dependence, we should only concern ourselves with variables that are updated by the loop body. That is, we should look for variables that are read or written in one iteration, and written in another. Let's look at a couple of examples.

5.5.4 Estimating π

One way to get a numerical approximation to π is to use many terms in the formula[4]

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right) = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}.$$

We can implement this formula in serial code with

```
1        double factor = 1.0;
2        double sum = 0.0;
3        for (k = 0; k < n; k++) {
4            sum += factor/(2*k+1);
5            factor = -factor;
6        }
7        pi_approx = 4.0*sum;
```

[4] This is by no means the best method for approximating π, since it requires a *lot* of terms to get a reasonably accurate result. However, in this case, lots of terms will be better to demonstrate the effects of parallelism, and we're more interested in the formula itself than the actual estimate.

(Why is it important that factor is a double instead of an int or a long?)

How can we parallelize this with OpenMP? We might at first be inclined to do something like this:

```
double factor = 1.0;
double sum = 0.0;
#pragma omp parallel for num_threads(thread_count) \
reduction(+: sum)
for (k = 0; k < n; k++) {
sum += factor/(2 k+1); *
factor = -factor; -
}
pi_approx = 4.0*sum;
```

However, it's pretty clear that the update to factor in Line 7 in iteration k and the subsequent increment of sum in Line 6 in iteration k+1 is an instance of a loop-carried dependence. If

iteration k is assigned to one thread and iteration k+1 is assigned to another thread, there's no guarantee that the value of factor in Line 6 will be correct. In this case, we can fix the problem by examining the series

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}.$$

We see that in iteration k, the value of factor should be $(-1)^k$, which is $+1$ if k is even and -1 if k is odd, so if we replace the code

```
1        sum += factor/(2*k+1);
2        factor = -factor;
```

by

```
1        if (k % 2 == 0)
2           factor = 1.0;
3        else
4           factor = -1.0;
5        sum += factor/(2*k+1);
```

or, if you prefer the ?: operator,

```
1        factor = (k % 2 == 0) ? 1.0 : -1.0;
2        sum += factor/(2*k+1);
```

we will eliminate the loop dependence.

However, things still aren't quite right. If we run the program on one of our systems with just two threads and $n=1000$, the result is consistently wrong. For example,

```
1     With n = 1000 terms and 2 threads,
2        Our estimate of pi = 2.97063289263385
3     With n = 1000 terms and 2 threads,
4        Our estimate of pi = 3.22392164798593
```

On the other hand, if we run the program with only one thread, we always get

1.With n = 1000 terms and 1 threads,
2.Our estimate of pi = 3.14059265383979

What's wrong here?

Recall that in a block that has been parallelized by a parallel **for** directive, by default any variable declared before the loop—with the sole exception of the loop variable—is shared among the threads. So factor is shared and, for example, thread 0 might assign it the value 1, but before it can use this value in the update to sum, thread 1 could assign it the value 1. Therefore, in addition to eliminating the loop- carried dependence in the calculation of factor, we need to ensure that each thread has its own copy of factor. That is, to make our code correct, we need to also ensure that factor has private scope. We can do this by adding a private clause to the parallel **for** directive.

```
1          double sum = 0.0;
2  #       pragma omp parallel for num_threads(thread_count) \
3              reduction(+: sum) private(factor)
4          for (k = 0; k < n; k++) {
5              if (k % 2 == 0)
6                  factor = 1.0;
7              else
8                  factor = -1.0;
9              sum += factor/(2*k+1);
10         }
```

The private clause specifies that for each variable listed inside the parentheses, a private copy is to be created for each thread. Thus, in our example, each of the thread_count threads will have its own copy of the variable factor, and hence the updates of one thread to factor won't affect the value of factor in another thread.

It's important to remember that the value of a variable with private scope is un- specified at the beginning of a parallel block or a parallel **for** block. Its value is also unspecified after completion of a parallel or parallel **for** block. So, for example, the output of the first printf statement in the following code is unspecified, since it prints the private variable x before it's explicitly initialized. Similarly, the output of the final printf is unspecified, since it prints x after the completion of the parallel block.

```
1      int x = 5;
2  #   pragma omp parallel num_threads(thread_count) \
3          private(x)
4      {
5          int my_rank = omp_get_thread_num();
6          printf("Thread %d > before initialization, x = %d\n",
7                 my_rank, x);
8          x = 2*my_rank + 2;

9          printf("Thread %d > after initialization, x = %d\n",
10                my_rank, x);
11     }
12     printf("After parallel block, x = %d\n", x);
```

5.5.5 More on scope

Our problem with the variable factor is a common one. We usually need to think about the scope of each variable in a parallel block or a parallel **for** block. There- fore, rather than letting OpenMP decide on the scope of each variable, it's a very good practice for us as programmers to specify the scope of each variable in a block. In fact, OpenMP provides a clause that will explicitly require us to do this: the **default** clause. If we add the clause

```
default (none)
```

to our parallel or parallel **for** directive, then the compiler will require that we spec- ify the scope of each variable we use in the block and that has been declared outside the block. (Variables that are declared within the block are always private, since they are allocated on the thread's stack.)

For example, using a **default** (none) clause, our calculation of π could be written as follows:

```
double sum = 0.0;
#pragma omp parallel for num_threads(thread_count) \
default(none) reduction(+: sum) private(k, factor) \
shared(n)
for (k = 0; k < n; k++) {
if (k % 2 == 0)
factor =1.0;
else
factor = -1.0;
sum += factor/(2*k+1);
}
```

In this example, we use four variables in the **for** loop. With the default clause, we need to specify the scope of each. As we've already noted, sum is a reduction variable (which has properties of both private and shared scope). We've also already noted that factor and the loop variable k should have private scope. Variables that are never updated in the parallel or parallel **for** block, such as n in this example, can be safely shared. Recall that unlike private variables, shared variables have the same value in the parallel or parallel **for** block that they had before the block, and their value after the block is the same as their last value in the block. Thus if n were initialized before the block to 1000, it would retain this value in the parallel **for** statement, and since the value isn't changed in the **for** loop, it would retain this value after the loop has completed.

5.6 More about loops in OpenMP: sorting

5.6.1 Bubble sort

Recall that the serial *bubble sort* algorithm for sorting a list of integers can be implemented as follows:

```
for (list_length = n; list_length >= 2; list_length--)
for (i = 0; i < list_length-1; i++)
if (a[i] > a[i+1]) {
tmp = a[i];
a[i] = a[i+1];
a[i+1] = tmp;
}
```

Here, a stores *n* **int**s and the algorithm sorts them in increasing order. The outer loop first finds the largest element in the list and stores it in a[n−1]; it then finds the next-to- the-largest element and stores it in a[n−2], and so on. So, effectively, the first pass is working with the full *n*-element list. The second is working with all of the elements, except the largest; it's working with an *n* 1-element list, and so on.

The inner loop compares consecutive pairs of elements in the current list. When a pair is out of order (a[i] > a[i+1]) it swaps them. This process of swapping will move the largest element to the last slot in the "current" list, that is, the list consisting of the elements

```
a[0], a[1], . . . , a[list_length−1]
```

It's pretty clear that there's a loop-carried dependence in the outer loop; in any iteration of the outer loop the contents of the current list depend on the previous iterations of the outer loop. For example, if at the start of the algorithm a = {3, 4, 1, 2}, then the second iteration of the outer loop should work with the list {3, 1, 2}, since the 4 should be moved to the last position by the first iteration. But if the first two iterations are executing simultaneously, it's possible that the effective list for the second iteration will contain 4.

The loop-carried dependence in the inner loop is also fairly easy to see. In iteration *i*, the elements that are compared depend on the outcome of iteration $i - 1$. If in iteration $i - 1$, a[i−1] and a[i] are not swapped, then iteration *i* should compare a[i] and a[i+1]. If, on the other hand, iteration $i - 1$ swaps a[i−1] and a[i], then iteration *i* should be comparing the original a[i−1] (which is now a[i]) and a[i+1]. For example, suppose the current list is {3,1,2}. Then when $i = 1$, we should compare 3 and 2,

but if the $i = 0$ and the $i = 1$ iterations are happening simultaneously, it's entirely possible that the $i = 1$ iteration will compare 1 and 2.

It's also not at all clear how we might remove either loop-carried dependence without completely rewriting the algorithm. It's important to keep in mind that even though we can always find loop-carried dependences, it may be difficult or impossible to remove them. The parallel **for** directive is not a universal solution to the problem of parallelizing **for** loops.

5.6.2 Odd-even transposition sort

Odd-even transposition sort is a sorting algorithm that's similar to bubble sort, but it has considerably more opportunities for parallelism. Recall from Section 3.7.1 that serial odd-even transposition sort can be implemented as follows:

```
for (phase = 0; phase < n; phase++)
    if (phase % 2 == 0)
        for (i = 1; i < n; i += 2)
            if (a[i-1] > a[i]) Swap(&a[i-1],&a[i]);
    else
        for (i = 1; i < n 1;_i += 2)
            if (a[i] > a[i+1]) Swap(&a[i] , &a[i+1]);
```

The list a stores *n* ints, and the algorithm sorts them into increasing order. During an "even phase" (phase %2 == 0), each odd-subscripted element, a[i], is compared to the element to its "left," a[i−1], and if they're out of order, they're swapped. During an "odd" phase, each odd-subscripted element is compared to the element to its right, and if they're out of order, they're swapped. A theorem guarantees that after *n* phases,

the list will be sorted.

As a brief example, suppose a = {9, 7, 8, 6}. Then the phases are shown in Table 5.2. In this case, the final phase wasn't necessary, but the algorithm doesn't bother checking whether the list is already sorted before carrying out each phase.

It's not hard to see that the outer loop has a loop-carried dependence. As an ex- ample, suppose as before that a = {9, 7, 8, 6}. Then in phase 0 the inner loop will compare elements in the pairs *(9, 7)* and *(8, 6)*, and both pairs are swapped. So for phase 1, the list should be {7, 9, 6, 8}, and during phase 1 the elements in the pair *(9, 6)* should be compared and swapped. However, if phase 0 and phase 1 are executed simultaneously, the pair that's checked in phase 1 might be *(7, 8)*, which is in order. Furthermore, it's not clear how one might eliminate this loop-carried dependence, so it would appear that parallelizing the outer for loop isn't an option.

The *inner* for loops, however, don't appear to have any loop-carried dependences. For example, in an even phase loop variable i will be odd, so for two distinct values of i, say $i = j$ and $i = k$, the pairs $\{j - 1, j\}$ and $\{k - 1, k\}$ will be disjoint. The comparison and possible swaps of the pairs (a[j−1], a[j]) and (a[k−1], a[k]) can therefore proceed simultaneously.

```
1     for (phase = 0; phase < n; phase++) {
2        if (phase % 2 == 0)
3  #        pragma omp parallel for num_threads(thread_count) \
4              default(none) shared(a, n) private(i, tmp)
5           for (i = 1; i < n; i += 2) {
6              if (a[i-1] > a[i]) {
7                 tmp = a[i-1];
8                 a[i-1] = a[i];
9                 a[i] = tmp;
10             }
11          }
12       else
13 #        pragma omp parallel for num_threads(thread_count) \
14                default(none) shared(a, n) private(i, tmp)
15          for (i = 1; i < n-1; i += 2) {
16             if (a[i] > a[i+1]) {
17                tmp = a[i+1];
18                a[i+1] = a[i];
19                a[i] = tmp;
20             }
21          }
22    }
```

Program 5.4: First OpenMP implementation of odd-even sort.

Thus we could try to parallelize odd-even transposition sort using the code shown in Program 5.4, but there are a couple of potential problems. First, although any iteration of, say, one even phase doesn't depend on any other iteration of that phase, we've already noted that this is not the case for iterations in phase *p* and phase *p+1*.

We need to be sure that all the threads have finished phase p before any thread starts phase p+1. However, like the parallel directive, the parallel **for** directive has an implicit barrier at the end of the loop, so none of the threads will proceed to the next phase, phase p+1, until all of the threads have completed the current phase, phase p. A second potential problem is the overhead associated with forking and joining the threads. The OpenMP implementation may fork and join thread_count threads on each pass through the body of the outer loop. The first row of Table 5.3 shows run- times for 1, 2, 3, and 4 threads on one of our systems when the input list contained 20,000 elements.

These aren't terrible times, but let's see if we can do better. Each time we execute one of the inner loops, we use the same number of threads, so it would seem to be superior to fork the threads once and reuse the same team of threads for each execution of the inner loops. Not surprisingly, OpenMP provides directives that allow us to do just this. We can fork our team of thread_count threads before the outer loop with a parallel directive. Then, rather than forking a new team of threads with each execution of one of the inner loops, we use a **for** directive,

which tells OpenMP to parallelize the **for** loop with the existing team of threads. This modification to the original OpenMP implementation is shown in Program 5.5.

```
1  #  pragma omp parallel num_threads(thread_count) \
2         default(none) shared(a, n) private(i, tmp, phase)
3     for (phase = 0; phase < n; phase++) {
4        if (phase % 2 == 0)
5  #        pragma omp for
6           for (i = 1; i < n; i += 2) {
7              if (a[i-1] > a[i]) {
8                 tmp = a[i-1];
9                 a[i-1] = a[i];
10                a[i] = tmp;
11             }
12          }
13       else
14 #        pragma omp for
15          for (i = 1; i < n-1; i += 2) {
16             if (a[i] > a[i+1]) {
17                tmp = a[i+1];
18                a[i+1] = a[i];
19                a[i] = tmp;
20             }
21          }
22    }
```

Program 5.5: Second OpenMP implementation of odd-even sort.

The **for** directive, unlike the parallel **for** directive, doesn't fork any threads. It uses whatever threads have already been forked in the enclosing parallel block. There is an implicit barrier at the end of the loop. The results of the code—the final list—will therefore be the same as the results obtained from the original parallelized code.

Run-times for this second version of odd-even sort are in the second row of Table 5.3. When we're using two or more threads, the version that uses two **for** directives is at least 17% faster than the version that uses two parallel **for** directives, so for this system the slight effort involved in making the change is well worth it.

5.7 Scheduling loops

When we first encountered the parallel **for** directive, we saw that the exact assignment of loop iterations to threads is system dependent. However most OpenMP implementations use roughly a block partitioning: if there are *n* iterations in the se- rial loop, then in the parallel loop the first *n*/thread_count are assigned to thread 0, the next *n*/thread_count are assigned to thread 1, and so on. It's not difficult to think of situations in which this assignment of iterations to threads would be less than optimal. For example, suppose we want to parallelize the loop

```
sum = 0.0;
for (i = 0; i <= n; i++)
    sum += f(i);
```

Also suppose that the time required by the call to *f* is proportional to the size of the argument i. Then a block partitioning of the iterations will assign much more work to thread thread_count 1 than it will assign to thread 0. A better assignment of work to threads might be obtained with a **cyclic** partitioning of the iterations among the threads. In a cyclic partitioning, the iterations are assigned, one at a time, in a "round-robin" fashion to the threads. Suppose *t* thread_count. Then a cyclic partitioning will assign the iterations as follows:

Thread	Iterations
0	$0, n/t, 2n/t, \ldots$
1	$1, n/t + 1, 2n/t + 1, \ldots$
.	.
$t-1$	$t-1, n/t + t - 1, 2n/t + t - 1, \ldots$

To get a feel for how drastically this can affect performance, we wrote a program in which we defined

```
double f(int i) {
    int j, start = i*(i+1)/2, finish = start + i;
    double return_val = 0.0;

    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
} /* f */
```

The call *f (i)* calls the sin function *i* times, and, for example, the time to execute *f (2i)* requires approximately twice as much time as the time to execute *f (i)*.

When we ran the program with *n* =10,000 and one thread, the run-time was 3.67 seconds. When we ran the program with two threads and the default assignment— iterations 0–5000 on thread 0 and iterations 5001–10,000 on thread 1—the run-time was 2.76 seconds. This is a speedup of only 1.33. However, when we ran the program with two threads and a cyclic assignment, the run-time was decreased to 1.84 seconds. This is a speedup of 1.99 over the one-thread run and a speedup of 1.5 over the two- thread block partition!

We can see that a good assignment of iterations to threads can have a very sig- nificant effect on performance. In OpenMP, assigning iterations to threads is called **scheduling**, and the schedule clause can be used to assign iterations in either a parallel **for** or a **for** directive.

5.7.1 The schedule clause

In our example, we already know how to obtain the default schedule: we just add a parallel **for** directive with a reduction clause:

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
            reduction(+: sum)
        for (i = 0; i <= n; i++)
            sum += f(i);
```

To get a cyclic schedule, we can add a schedule clause to the parallel **for** directive:

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
            reduction(+: sum) schedule(static,1)
        for (i = 0; i <= n; i++)
            sum += f(i);
```

In general, the schedule clause has the form

```
schedule(<type> [, <chunksize>])
```

The type can be any one of the following:

- static. The iterations can be assigned to the threads before the loop is executed.
- dynamic or guided. The iterations are assigned to the threads while the loop is executing, so after a thread completes its current set of iterations, it can request more from the run-time system.

- auto. The compiler and/or the run-time system determine the schedule.
- runtime. The schedule is determined at run-time based on an environment variable (more on this later).

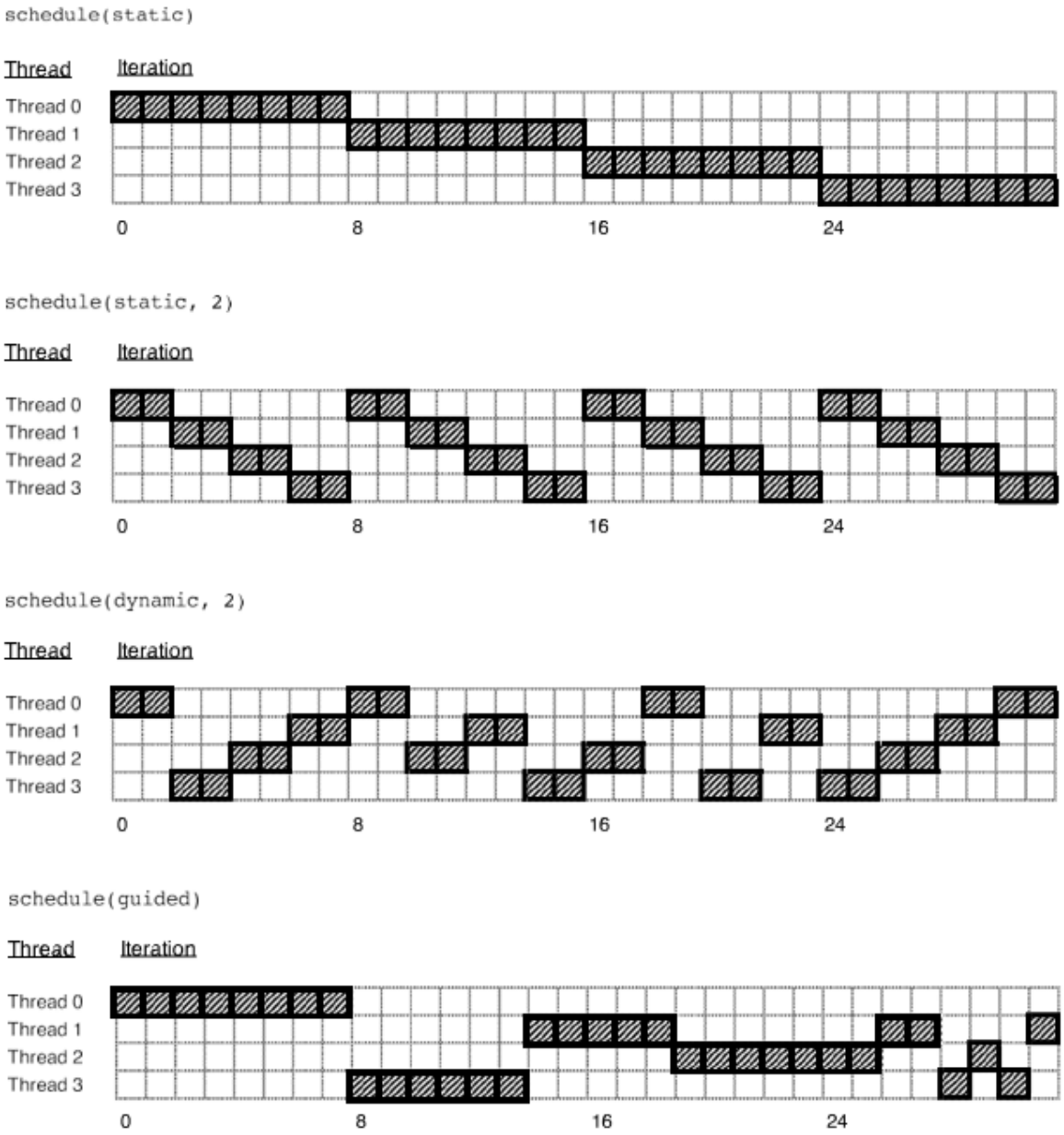


FIGURE 5.5

Scheduling visualization for the static, dynamic, and guidedschedule types with 4 threads and 32 iterations. The first static schedule uses the default chunksize, whereas the second uses a chunksizeof **2**. The exact distribution of work across threads will vary between different executions of the program for the dynamicand guidedschedule types, so this visualization shows one of many possible scheduling outcomes.

The chunksize is a positive integer. In OpenMP parlance, a **chunk** of iterations is a block of iterations that would be executed consecutively in the serial loop. The number of iterations in the block is the chunksize. Only static, dynamic, and guided schedules can have a chunksize. This determines the details of the schedule, but its exact interpretation depends on the type. Fig. 5.5 provides a visualization of how work is scheduled using the static, dynamic, and guided types.

5.7.2 The static schedule type

For a static schedule, the system assigns chunks of chunksize iterations to each thread in a round-robin fashion. As an example, suppose we have 12 iterations, 0, 1,…, 11, and three threads. Then if schedule(static , 1) is used, in the parallel for or for directive, we've already seen that the iterations will be assigned as

```
Thread 0 :  0, 3, 6, 9
Thread 1 :  1, 4, 7, 10
Thread 2 :  2, 5, 8, 11
```

If schedule(static , 2) is used, then the iterations will be assigned as

```
Thread 0 :  0, 1, 6, 7
Thread 1 :  2, 3, 8, 9
Thread 2 :  4, 5, 10, 11
```

If schedule(static , 4) is used, the iterations will be assigned as

```
Thread 0 :  0, 1, 2, 3
Thread 1 :  4, 5, 6, 7
Thread 2 :  8, 9, 10, 11
```

The default schedule is defined by your particular implementation of OpenMP, but in most cases it is equivalent to the clause

schedule(static , total_iterations/ thread_count)

It is also worth noting that the chunksize can be omitted. If omitted, the chunksize is approximately total_iterations / thread_count.

The static schedule is a good choice when each loop iteration takes roughly the same amount of time to compute. It also has the advantage that threads in subsequent loops with the same

number of iterations will be assigned to the same ranges; this can improve the speed of memory accesses, particularly on NUMA systems (see Chapter 2).

5.7.3 The dynamic and guided schedule types

In a dynamic schedule, the iterations are also broken up into chunks of chunksize consecutive iterations. Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system. This continues until all the iterations are completed. The chunksize can be omitted. When it is omitted, a chunksize of 1 is used.

The primary difference between static and dynamic schedules is that the dynamic schedule assigns ranges to threads on a first-come, first-served basis. This can be advantageous if loop iterations do not take a uniform amount of time to compute (some

Table 5.4 Assignment of trapezoidal rule iterations 1-9999 using a guided schedule with two threads.

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999	1	0

algorithms are more compute-intensive in later iterations, for instance). However, since the ranges are not allocated ahead of time, there is some overhead associated with assigning them dynamically at run-time. Increasing the chunk size strikes a balance between the performance characteristics of static and dynamic scheduling; with larger chunk sizes, fewer dynamic assignments will be made.

The guided schedule is similar to dynamic in that each thread also executes a chunk and requests another one when it's finished. However, in a guided schedule, as chunks are completed, the size of the new chunks decreases. For example, on one of our systems, if we run the trapezoidal rule program with the parallel **for** directive and a schedule(guided) clause, then when $n = 10{,}000$ and

thread_count 2, the iterations are assigned as shown in Table 5.4. We see that the size of the chunk is approximately the number of iterations remaining divided by the number of threads. The first chunk has size 9999/2 5000, since there are 9999 unassigned iterations. The second chunk has size 4999/2 2500, and so on.

In a guided schedule, if no chunksize is specified, the size of the chunks decreases down to 1. If chunksize is specified, it decreases down to chunksize, with the exception that the very last chunk can be smaller than chunksize. The guided schedule can improve the balance of load across threads when later iterations are more compute- intensive.

5.7.4 The runtime schedule type

To understand schedule(runtime), we need to digress for a moment and talk about **environment variables**. As the name suggests, environment variables are named values that can be accessed by a running program. That is, they're available in the program's environment. Some commonly used environment variables are PATH, HOME, and SHELL. The PATH variable specifies which directories the shell should search when it's look- ing for an executable and is usually defined in both Unix and Windows. The HOME variable specifies the location of the user's home directory, and the SHELL variable specifies the location of the executable for the user's shell. These are usually defined in Unix systems. In both Unix-like systems (e.g., Linux and macOS) and Windows, environment variables can be examined and specified on the command line. In Unix- like systems, you can use the shell's command line. In Windows systems, you can use the command line in an integrated development environment.

As an example, if we're using the bash shell (one of the most common Unix shells), we can examine the value of an environment variable by typing:

```
$ echo $PATH
```

and we can use the exportcommand to set the value of an environment variable:

```
$ export TEST_VAR="hello"
```

These commands also work on ksh, sh, and zsh. For details about how to examine and set environment variables for your particular system, check the man pages for your shell, or consult with your system administrator or local expert.

When schedule(runtime) is specified, the system uses the environment variable OMP_SCHEDULE to determine at run-time how to schedule the loop. The OMP_SCHEDULE environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule. For example, suppose we have a parallel **for** directive in a program and it has been modified by schedule(runtime). Then if we use the bash shell, we can get a cyclic assignment of iterations to threads by executing the command

```
$ export OMP_SCHEDULE="static,1"
```

Now, when we start executing our program, the system will schedule the iterations of the **for** loop as if we had the clause schedule(static,1) modifying the parallel **for** directive. This can be very useful for testing a variety of scheduling configurations.

The following bash shell script demonstrates how one might take advantage of this environment variable to test a range of schedules and chunk sizes. It runs a matrix-vector multiplication program that has a parallel **for** directive with the schedule(runtime) clause.

```
#!/usr/bin/env bash

declare -a schedules=("static" "dynamic" "guided")
declare -a chunk_sizes=("" 1000 100 10 1)

for schedule in "${schedules[@]}"; do
    echo "Schedule: ${schedule}"
    for chunk_size in "${chunk_sizes[@]}"; do
        echo "Chunk Size: ${chunk_size}"
        sched_param="${schedule}"

        if [[ "${chunk_size}" != "" ]]; then
            # A blank string indicates we want
            # the default chunk size
            sched_param="${schedule},${chunk_size}"
        fi

        # Run the program with OMP_SCHEDULE set:
        OMP_SCHEDULE="${sched_param}" ./omp_mat_vect 4 2500 2500
    done
    echo
done
```

5.7.5 Which schedule?

If we have a **for** loop that we're able to parallelize, how do we decide which type of schedule we should use and what the chunksize should be? As you may have guessed, there *is* some overhead associated with the use of a schedule clause. Furthermore, the overhead is greater for dynamic schedules than static schedules, and the overhead associated with guided schedules is the greatest of the three. Thus if we're getting satisfactory performance without a schedule clause, we should go no further. How- ever, if we suspect that the performance of the default schedule can be substantially improved, we should probably experiment with some different schedules.

In the example at the beginning of this section, when we switched from the default schedule to schedule(**static** ,1) , the speedup of the two-threaded execution of the pro- gram increased from 1.33 to 1.99. Since it's extremely unlikely that we'll get speedups that are significantly better than 1.99, we can just stop here, at least if we're only going to use two threads with 10,000 iterations. If we're going to be using varying numbers of threads and varying numbers of iterations, we need to do more experimentation, and it's entirely possible that we'll find that the optimal schedule depends on both the number of threads and the number of iterations.

It can also happen that we'll decide that the performance of the default schedule isn't very good, and we'll proceed to search through a large array of schedules and iteration counts only to conclude that our loop doesn't parallelize very well and *no* schedule is going to give us much improved performance. For an example of this, see Programming Assignment 5.4.

There are some situations in which it's a good idea to explore some schedules before others:

- If each iteration of the loop requires roughly the same amount of computation, then it's likely that the default distribution will give the best performance.
- If the cost of the iterations decreases (or increases) linearly as the loop executes, then a static schedule with small chunksizes will probably give the best perfor- mance.
- If the cost of each iteration can't be determined in advance, then it may make sense to explore a variety of scheduling options. The schedule(runtime) clause can be used here, and the different options can be explored by running the program with different assignments to the environment variable OMP_SCHEDULE.

5.8 Producers and consumers

Let's take a look at a parallel problem that isn't amenable to parallelization using a

parallel **for** or **for** directive.

5.8.1 Queues

A **queue** is a list abstract datatype in which new elements are inserted at the "rear" of the queue and elements are removed from the "front" of the queue. A queue can thus be viewed as an abstraction of a line of customers waiting to pay for their groceries in a supermarket. The elements of the list are the customers. New customers go to the end or "rear" of the line, and the next customer to check out is the customer standing at the "front" of the line.

When a new entry is added to the rear of a queue, we sometimes say that the entry has been "enqueued," and when an entry is removed from the front of a queue, we sometimes say that the entry has been "dequeued."

Queues occur frequently in computer science. For example, if we have a number of processes, each of which wants to store some data on a hard drive, then a natural way to ensure that only one process writes to the disk at a time is to have the processes form a queue, that is, the first process that wants to write gets access to the drive first, the second process gets access to the drive next, and so on.

A queue is also a natural data structure to use in many multithreaded applications. For example, suppose we have several "producer" threads and several "consumer" threads. The producer threads might "produce" requests for data from a server— for example, current stock prices—while the consumer threads might "consume" the request by finding or generating the requested data—the current stock prices. The producer threads could enqueue the requested prices, and the consumer threads could dequeue them. In this example, the process wouldn't be completed until the consumer threads had given the requested data to the producer threads.

5.8.2 Message-passing

Another natural application would be implementing message-passing on a shared- memory system. Each thread could have a shared-message queue, and when one thread wanted to "send a message" to another thread, it could enqueue the message in the destination thread's queue. A thread could receive a message by dequeuing the message at the head of its message queue.

Let's implement a relatively simple message-passing program, in which each thread generates random integer "messages" and random destinations for the messages. After creating the message, the thread enqueues the message in the appropriate message queue. After sending a message, a thread checks its queue to see if it has re- ceived a message. If it has, it dequeues the first message in its queue and prints it out. Each thread alternates between sending and trying to receive messages. We'll let the user specify the number of messages each thread should send. When a thread is done sending messages, it receives messages until all the threads are done, at which point all the threads quit. Pseudocode for each thread might look something like this:

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {
    Send_msg();
    Try_receive();
}

while (! Done())
    Try_receive();
```

5.8.3 Sending messages

Note that accessing a message queue to enqueue a message is probably a critical section. Although we haven't looked into the details of the implementation of the message queue, it seems likely that we'll want to have a variable that keeps track of the rear of the queue. For example, if we use a singly linked list with the tail of the list corresponding to the rear of the queue, then, to efficiently enqueue, we would want to store a pointer to the rear. When we enqueue a new message, we'll need to check and update the rear pointer. If two threads try to

do this simultaneously, we may lose a message that has been enqueued by one of the threads. (It might help to draw a picture!) The results of the two operations will conflict, and hence enqueuing a message will form a critical section.

Pseudocode for the Send_msg() function might look something like this:

```
    mesg = random();
    dest = random() % thread_count;
#   pragma omp critical
    Enqueue(queue, dest, my_rank, mesg);
```

Note that this allows a thread to send a message to itself.

5.8.4 Receiving messages

The synchronization issues for receiving a message are a little different. Only the owner of the queue (that is, the destination thread) will dequeue from a given message queue. As long as we dequeue one message at a time, if there are at least two messages in the queue, a call to Dequeue can't possibly conflict with any calls to Enqueue. So if we keep track of the size of the queue, we can avoid any synchronization (for example, critical directives), as long as there are at least two messages.

Now you may be thinking, "What about the variable storing the size of the queue?" This would be a problem if we simply store the size of the queue. However, if we store two variables, enqueued and dequeued, then the number of messages in the queue is

$$\text{queue\_size} = \text{enqueued} - \text{dequeued}$$

and the only thread that will update dequeued is the owner of the queue. Observe that one thread can update enqueued at the same time that another thread is using it to compute queue_size. To see this, let's suppose thread *q* is computing queue_size. It will either get the old value of enqueued or the new value. It *may* therefore compute a queue_size of 0 or 1 when queue_size should actually be 1 or 2, respectively, but in our program this will only cause a modest delay. Thread *q* will try again later if queue_size is 0 when it should be 1, and it will execute the critical section directive unnecessarily if queue_size is 1 when it should be 2.

Thus we can implement Try_receive as follows:

```
        queue_size = enqueued - dequeued;
        if (queue_size == 0) return;
        else if (queue_size == 1)
#           pragma omp critical
            Dequeue(queue, &src, &mesg);
        else
            Dequeue(queue, &src, &mesg);
        Print_message(src, mesg);
```

5.8.4 Termination detection

We also need to think about implementation of the Done function. First note that the following "obvious" implementation will have problems:

```
queue_size = enqueued - dequeued; if (queue_size== 0)
return TRUE; else
return FALSE;
```

If thread *u* executes this code, it's entirely possible that some thread—call it thread *v*—will send a message to thread *u after u* has computed queue_size = 0. Of course, after thread *u* computes queue_size = 0, it will terminate and the message sent by thread *v* will never be received.

However, in our program, after each thread has completed the **for** loop, it won't send any new messages. Thus if we add a counter done_sending, and each thread increments this after completing its **for** loop, then we *can* implement Done as follows:

```
queue_size = enqueued - dequeued;
if (queue_size == 0 && done_sending == thread_count) return TRUE;
else
return FALSE;
```

5.8.5 Startup

When the program begins execution, a single thread, the master thread, will get command-line arguments and allocate an array of message queues, one for each thread. This array needs to be shared among the threads, since any thread can send to any other thread, and hence any thread

can enqueue a message in any of the queues. Given that a message queue will (at a minimum) store

- a list of messages,
- a pointer or index to the rear of the queue,
- a pointer or index to the front of the queue,
- a count of messages enqueued, and
- a count of messages dequeued,

it makes sense to store the queue in a struct, and to reduce the amount of copying when passing arguments, it also makes sense to make the message queue an array of pointers to structs. Thus once the array of queues is allocated by the master thread, we can start the threads using a parallel directive, and each thread can allocate storage for its individual queue.

An important point here is that one or more threads may finish allocating their queues before some other threads. If this happens, the threads that finish first could start trying to enqueue messages in a queue that hasn't been allocated and cause the program to crash. We therefore need to make sure that none of the threads starts sending messages until all the queues are allocated. Recall that we've seen that sev- eral OpenMP directives provide implicit barriers when they're completed, that is, no thread will proceed past the end of the block until all the threads in the team have completed the block. In this case, though, we'll be in the middle of a parallel block, so we can't rely on an implicit barrier from some other OpenMP construct—we need an *explicit* barrier. Fortunately, OpenMP provides one:

```
# pragma omp barrier
```

When a thread encounters the barrier, it blocks until all the threads in the team have reached the barrier. After all the threads have reached the barrier, all the threads in the team can proceed.

5.8.7 The atomic directive

After completing its sends, each thread increments done_sending before proceeding to its final loop of receives. Clearly, incrementing done_sending is a critical section, and we could protect it with a critical directive. However, OpenMP provides a potentially higher performance directive: the atomic directive[5]:

```
# pragma omp atomic
```

Unlike the critical directive, it can only protect critical sections that consist of a single C assignment statement. Further, the statement must have one of the following forms:

```
x <op>= <expression>;
x++;
++x;
x--;
--x;
```

Here <op> can be one of the binary operators

+, *, -, /, &, ^, |, <-<, or >>.

It's also important to remember that <expression> must not reference x.

It should be noted that only the load and store of xare guaranteed to be protected. For example, in the code

```
#       pragma omp atomic
        x += y++;
```

a thread's update to x will be completed before any other thread can begin updating x. However, the update to ymay be unprotected and the results may be unpredictable.

The idea behind the atomic directive is that many processors provide a special load-modify-store instruction, and a critical section that only does a load-modify- store can be protected much more efficiently by using this special instruction rather than the constructs that are used to protect more general critical sections.

5.8.8 Critical sections and locks

OpenMP's specification of the critical directive. In our earlier exam- ples, our programs had at most one critical section, and the critical directive forced mutually exclusive access to the section by all the threads. In this program, however, the use of critical sections is more complex. If we simply look at the source code, we'll see three blocks of code preceded by a critical or an atomic directive:

- done_sending++;
- Enqueue(q_p, my_rank, mesg);
- Dequeue(q_p, &src, &mesg);

However, we don't need to enforce exclusive access across all three of these blocks of code. We don't even need to enforce completely exclusive access within Enqueue and Dequeue. For example, it would be fine for, say, thread 0 to enqueue a message in thread 1's queue at the same time that thread 1 is enqueuing a message in thread 2's queue. But for the second and third blocks—the blocks protected by critical directives—this is exactly what OpenMP does. From OpenMP's point of view our program has two distinct critical sections: the critical section protected by the atomic directive, (done_sending++), and the "composite" critical section in which we enqueue and dequeue messages.

Since enforcing mutual exclusion among threads serializes execution, this default behavior of OpenMP—treating all critical blocks as part of one composite critical section—can be highly detrimental to our program's performance. OpenMP *does* provide the option of adding a name to a critical directive:

```
# pragma omp critical(name)
```

When we do this, two blocks protected with critical directives with different names *can* be executed simultaneously. However, the names are set during compilation, and we want a different critical section for each thread's queue. Therefore we need to set the names at run-time, and in our setting, when we want to allow simultaneous access to the same block of code by threads accessing different queues, the named critical directive isn't sufficient.

The alternative is to use **locks**.[6] A lock consists of a data structure and functions that allow the programmer to explicitly enforce mutual exclusion in a critical section. The use of a lock can be roughly described by the following pseudocode:

```
/* Executed by one thread */
Initialize the lock data structure ;
. . .
/* Executed by multiple threads */
Attempt to lock or set the lock data structure Critical section ; ;
Unlock or unset the lock data structure;
. . .
/* Executed by one thread */
Destroy the lock data structure ;
```

The lock data structure is shared among the threads that will execute the critical section. One of the threads (e.g., the master thread) will initialize the lock, and when all the threads are done using the lock, one of the threads should destroy it.

Before a thread enters the critical section, it attempts to *set* the lock by calling the lock function. If no other thread is executing code in the critical section, it *acquires* the lock and proceeds into the critical section past the call to the lock function. When the thread finishes the code in the critical section, it calls an unlock function, which *releases* or *unsets* the lock and allows another thread to acquire the lock.

While a thread owns the lock, no other thread can enter the critical section. If another thread attempts to enter the critical section, it will *block* when it calls the lock function. If multiple threads are blocked in a call to the lock function, then when the thread in the critical section releases the lock, one of the blocked threads returns from the call to the lock, and the others remain blocked.

OpenMP has two types of locks: **simple** locks and **nested** locks. A simple lock can only be set once before it is unset, while a nested lock can be set multiple times by the same thread before it is unset. The type of an OpenMP simple lock is omp_lock_t, and the simple lock functions that we'll be using are

```
void omp_init_lock(omp_lock_t*      lock_p /* out */);
void omp_set_lock(omp_lock_t*       lock_p /* in/out */);
void omp_unset_lock(omp_lock_t*     lock_p /* in/out */);
void omp_destroy_lock(omp_lock_t*   lock_p /* in/out */);
```

The type and the functions are specified in omp.h. The first function initializes the lock so that it's unlocked, that is, no thread owns the lock. The second function attempts to set the lock. If it succeeds, the calling thread proceeds; if it fails, the calling thread blocks until the lock becomes available. The third function unsets the lock so another thread can acquire it. The fourth function makes the lock uninitialized. We'll only use simple locks. For information about nested locks, see [9], [10], or [47].

5.8.9 Using locks in the message-passing program

In our earlier discussion of the limitations of the critical directive, we saw that in the message-passing program, we wanted to ensure mutual exclusion in each individual message queue, not in a particular block of source code. Locks allow us to do this. If we include a data member with type omp_lock_t in our queue struct, we can simply call omp_set_lock each time we want to ensure exclusive access to a message queue.

So the code

```
#  pragma omp critical
   /* q_p = msg_queues[dest] */
   Enqueue(q_p, my_rank, mesg);
```

can be replaced with

```
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, mesg);
omp_unset_lock(&q_p->lock);
```

Similarly, the code

```
#  pragma omp critical
   /* q_p = msg_queues[my_rank] */
   Dequeue(q_p, &src, &mesg);
```

can be replaced with

```
/* q_p = msg_queues[my_rank] */
omp_set_lock(&q_p->lock);
Dequeue(q_p, &src, &mesg);
omp_unset_lock(&q_p->lock);
```

Now when a thread tries to send or receive a message, it can only be blocked by a thread attempting to access the same message queue, since different message queues have different locks. In our original implementation, only one thread could send at a time, regardless of the destination.

Note that it would also be possible to put the calls to the lock functions in the queue functions Enqueue and Dequeue. However, to preserve the performance of Dequeue, we would also need to move the code that determines the size of the queue (enqueued – dequeued) to Dequeue. Without it, the Dequeue function will lock the queue every time it is called by Try_receive. In the interest of preserving the structure of the code we've already written, we'll leave the calls to omp_set_lock and omp_unset_lock in the Send and Try_receive functions.

Since we're now including the lock associated with a queue in the queue struct, we can add initialization of the lock to the function that initializes an empty queue. Destruction of the lock can be done by the thread that owns the queue before it frees the queue.

5.8.10 Critical directives, atomic directives, or locks?

Now that we have three mechanisms for enforcing mutual exclusion in a critical sec- tion, it's natural to wonder when one method is preferable to another. In general, the atomic directive has the potential to be the fastest method of obtaining mutual ex- clusion. Thus if your critical section consists of an assignment statement having the required form, it will probably perform at least as well with the atomic directive as the other methods. However, the OpenMP specification [47] allows the atomic directive to enforce mutual exclusion across *all* atomic directives in the program—this is the way the unnamed critical directive behaves. If this might be a problem—for example, you have multiple different critical sections protected by atomic directives—you should use named critical directives or locks. For example, suppose we have a pro- gram in which it's possible that one thread will execute the code on the left while another executes the code on the right.

```
# pragma omp atomic
x++;
```

```
# pragma omp atomic
y++;
```

Even if x and y are unrelated memory locations, it's possible that if one thread is exe- cuting x++, then no thread can simultaneously execute y++. It's important to note that the standard doesn't require this behavior. If two statements are protected by atomic directives and the two statements modify different variables, then there are implemen- tations that treat the two statements as different critical sections. (See Exercise 5.10.) On the other hand, different statements that modify the same variable *will* be treated as if they belong to the same critical section, regardless of the implementation.

We've already seen some limitations to the use of critical directives. However, both named and unnamed critical directives are very easy to use. Furthermore, in the implementations of OpenMP that we've used there doesn't seem to be a very large difference between the performance of critical sections protected by a critical directive, and critical sections protected by locks, so if you can't use an atomic directive, but you can use a critical directive, you probably should. Thus the use of locks should probably be reserved for situations in which mutual exclusion is needed for a data structure rather than a block of code.

5.8.11 Some caveats

You should exercise caution when using the mutual exclusion techniques we've dis- cussed. They can definitely cause serious programming problems. Here are a few things to be aware of:

1. You shouldn't mix the different types of mutual exclusion for a single critical section. For example, suppose a program contains the following two segments:

```
# pragma omp atomic
    x += f(y);
```

```
# pragma omp critical
    x = g(x);
```

The update to x on the right doesn't have the form required by the atomic directive, so the programmer used a critical directive. However, the critical directive won't exclude the action executed by the atomic block, and it's possible that the results will be incorrect. The programmer needs to either rewrite the function *g* so that its use can have the form required by the atomic directive or to protect both blocks with a critical directive.

2.There is no guarantee of **fairness** in mutual exclusion constructs. This means that it's possible that a thread can be blocked forever in waiting for access to a critical section. For example, in the code

```
while (1) {
    . . .
# pragma omp critical
    x = g(my_rank);
    . . .
}
```

it's possible that, for example, thread 1 can block forever waiting to execute x = g(my_rank) while the other threads repeatedly execute the assignment. Of course, this wouldn't be an issue if the loop terminated.

3.It can be dangerous to "nest" mutual exclusion constructs. As an example, suppose a program contains the following two segments:

```
#   pragma omp critical
    y = f(x);
    . . .
    double f(double x) {
#       pragma omp critical
        z = g(x);       /* z is shared */
        . . .
    }
```

This is guaranteed to **deadlock**. When a thread attempts to enter the second critical section, it will block forever. If thread *u* is executing code in the first critical block, no thread can execute code in the second block. In particular, thread *u* can't execute this code. However, if thread *u* is blocked waiting to enter the second critical block, then it will never leave the first, and it will stay blocked forever.

In this example, we can solve the problem by using named critical sections. That is, we could rewrite the code as

```
#   pragma omp critical (one)
    y = f(x);
    . . .
    double f(double x) {
#       pragma omp critical (two)
        z = g(x);       /* z is global */
        . . .
    }
```

However, it's not difficult to come up with examples when naming won't help. For example, if a program has two named critical sections—say one and two—and threads can attempt to enter the critical sections in different orders, then deadlock can occur. For example, suppose thread *u* enters one at the same time that thread *v* enters two and *u* then attempts to enter two while *v* attempts to enter one:

Time	Thread *u*	Thread *v*
0	Enter crit. sect. one	Enter crit. sect. two
1	Attempt to enter two	Attempt to enter one
2	Block	Block

Then both *u* and *v* will block forever waiting to enter the critical sections. So it's not enough to just use different names for the critical sections—the programmer must ensure that different critical sections are always entered in the same order.

5.9 Caches, cache coherence, and false sharing

Recall that for a number of years now, computers have been able to execute operations involving only the processor much faster than they can access data in main memory. If a processor must read data from main memory for each operation, it will spend most of its time simply waiting for the data from memory to arrive. Also recall that to address this problem, chip designers have added blocks of relatively fast memory to processors. This faster memory is called **cache memory**.

Table 5.5 Memory and cache accesses.

Time	Memory	Th 0	Th 0 cache	Th 1	Th 1 cache
0	x = 5	Load x	—	Load x	—
1	x = 5	—	x = 5	—	x = 5
2	x = 5	x++	x = 5	—	x = 5
3	???	—	x = 6	my_z = x	???

The design of cache memory takes into consideration the principles of **temporal and spatial locality**: if a processor accesses main memory location *x* at time *t* , then it is likely that at times close to *t* it will access main memory locations close to *x*. Thus if a processor needs to access main memory location *x*, rather than transferring only the contents of *x* to/from main memory, a block of memory containing *x* is transferred from/to the processor's cache. Such a block of memory is called a **cache line** or **cache block**.

In Section 2.3.5, we saw that the use of cache memory can have a huge impact on shared memory. Let's recall why. First, consider the following situation: Suppose x is a shared variable with the value five, and both thread 0 and thread 1 read x from memory into their (separate) caches, because both want to execute the statement

```
my_y = x;
```

Here, my_y is a private variable defined by both threads. Now suppose thread 0 exe- cutes the statement

```
x++;
```

Finally, suppose that thread 1 now executes

```
my_z = x;
```

where my_z is another private variable. Table 5.5 illustrates the sequence of accesses. What's the value in my_z? Is it five? Or is it six? The problem is that there are (at least) three copies of x: the one in main memory, the one in thread 0's cache, and the one in thread 1's cache. When thread 0 executed x++, what happened to the values in main memory and thread 1's cache? This is the **cache coherence** problem, which we discussed in Chapter 2. We saw there that most systems insist that the caches be made aware that changes have been made to data they are caching. The line in the cache of thread 1 would have been marked *invalid* when thread 0 executed x++, and before assigning my_z = x, the core running thread 1 would see that its value of x was out of date. Thus the core running thread 0 would have to update the copy of x in main memory (either now or earlier), and the core running thread 1 could get the line with the updated value of x from main memory. For further details, see Chapter 2.

The use of cache coherence can have a dramatic effect on the performance of shared-memory systems. To illustrate this, let's take a look at matrix-vector multiplication. Recall that if $A = (a_{ij})$ is an $m \times n$ matrix and $\mathbf{x}$ is a vector with n components, then their product $\mathbf{y} = A\mathbf{x}$ is a vector with m components, and its ith component y_i is

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1} \end{bmatrix}$$

FIGURE 5.6
Matrix-vector multiplication.

found by forming the dot product of the ith row of A with $\mathbf{x}$:

$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}.$$

See Fig. 5.6.

So if we store A as a two-dimensional array and $\mathbf{x}$ and $\mathbf{y}$ as one-dimensional arrays, we can implement serial matrix-vector multiplication with the following code:

```
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][i] * x[i];
}
```

There are no loop-carried dependences in the outer loop, since A and x are never updated and iteration *i* only updates y[i]. Thus we can parallelize this by dividing the iterations in the outer loop among the threads:

```
1  #  pragma omp parallel for num_threads(thread_count)  \
2        default(none) private(i, j) shared(A, x, y, m, n)
3     for (i = 0; i < m; i++) {
4        y[i] = 0.0;
5        for (j = 0; j < n; j++)
6           y[i] += A[i][i] x[i];
7     }
```

If T_{serial} is the run-time of the serial program, and $T_{parallel}$ is the run-time of the parallel program, recall that the *efficiency* E of the parallel program is the speedup S divided by the number of threads:

$$E = \frac{S}{t} = \frac{\frac{T_{serial}}{T_{parallel}}}{t} = \frac{T_{serial}}{t \times T_{parallel}}.$$

Table 5.6 Run-times and efficiencies of matrix-vector multiplication (times are in seconds).

	Matrix Dimension					
	8,000,000 × 8		8000 × 8000		8 × 8,000,000	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Since $S \leq t$, $E \leq 1$. Table 5.6 shows the run-times and efficiencies of our matrix- vector multiplication with different sets of data and differing numbers of threads. In each case, the total number of floating point additions and multiplications is 64,000,000. An analysis that only considers arithmetic operations would predict that a single thread running the code would take the same amount of time for all three in- puts. However, it's clear that this is *not* the case. The 8,000,000 × 8 system requires about 22% more time than the 8000 × 8000 system, and the 8

8,000,000 system re- quires about 26% more time than the 8000 × 8000 system. Both of these differences are at least partially attributable to cache performance.

Recall that a *write-miss* occurs when a core tries to update a variable that's not in cache, and it has to access main memory. A cache profiler (such as Valgrind [51]) shows that when the program is run with the 8,000,000 × 8 input, it has far more cache write-misses than either of the other inputs. The bulk of these occur in Line 4. Since the number of elements in the vector y is far greater in this case (8,000,000 vs. 8000 or 8), and each element must be initialized, it's not surprising that this line slows down the execution of the program with the 8,000,000 × 8 input.

Also recall that a *read-miss* occurs when a core tries to read a variable that's not in the cache, and it has to access main memory. A cache profiler shows that when the program is run with the 8 × 8,000,000 input, it has far more cache read-misses than either of the other inputs. These occur in Line 6, and a careful study of this program (see Exercise 5.12) shows that the main source of the differences is due to the reads of x. Once again, this isn't surprising, since for this input, x has 8,000,000 elements, versus only 8000 or 8 for the other inputs.

It should be noted that there may be other factors that are affecting the relative performance of the single-threaded program with the differing inputs. For example, we haven't taken into consideration whether virtual memory (see Subsection 2.2.4) has affected the performance of the program with the different inputs. How frequently does the CPU need to access the page table in main memory?

Of more interest to us, though, are the differences in efficiency as the number of threads is increased. The two-thread efficiency of the program with the 8 × 8,000,000 input is more than 20% less than the efficiency of the program with the 8,000,000 × 8 and the 8000 × 8000 inputs. The four-thread efficiency of the program with the 8 × 8,000,000 input is more than 50% less than the program's efficiency with the 8,000,000 × 8 and the 8000 × 8000 inputs. Why, then, is the multithreaded performance of the program so much worse with the 8 × 8,000,000 input?

In this case, once again, the answer has to do with cache. Let's take a look at the program when we run it with four threads. With the 8,000,000 × 8 input, y has 8,000,000 components, so each thread is assigned 2,000,000 components. With the 8000 × 8000 input, each thread is assigned 2000 components of y, and with the 8 × 8,000,000 input, each thread is assigned two components. On the system we used, a cache line is 64 bytes. Since the type of y is **double**, and a **double** is 8 bytes, a single cache line will store eight **double**s.

Cache coherence is enforced at the "cache-line level." That is, each time any value in a cache line is written, if the line is also stored in another core's cache, the entire *line* will be invalidated—not just the value that was written. The system we're using has two dual-core processors and each processor has its own cache. Suppose for the moment that threads 0 and 1 are assigned to one of the processors and threads 2 and 3 are assigned to the other. Also suppose that for the 8 × 8,000,000 problem all of y is stored in a single cache line. Then every write to some element of

will invalidate the line in the other processor's cache. For example, each time thread 0 updates y[0] in the statement

```
y[i] += A[i][j]*x[j];
```

if thread 2 or 3 is executing this code, it will have to reload y. Each thread will update each of its components 8,000,000 times. We see that with this assignment of threads to processors and components of y to cache lines, all the threads will have to reload y many times. This is going to happen in spite of the fact that only one thread accesses any one component of y—for example, only thread 0 accesses y[0].

Each thread will update its assigned components of y a total of 16,000,000 times. It appears that many if not most of these updates are forcing the threads to access main memory. This is called **false sharing**. Suppose two threads with separate caches access different variables that belong to the same cache line. Further suppose at least one of the threads updates its variable. Then even though neither thread has written to a shared variable, the cache controller invalidates the entire cache line and forces the other threads to get the values of the variables from main memory. The threads aren't sharing anything (except a cache line), but the behavior of the threads with respect to memory access is the same as if they were sharing a variable, hence the name false sharing.

Why is false sharing not a problem with the other inputs? Let's look at what happens with the 8000 × 8000 input. Suppose thread 2 is assigned to one of the processors and thread 3 is assigned to another. (We don't actually know which threads are assigned to which processors, but it turns out—see Exercise 5.13—that it doesn't matter.) Thread 2 is responsible for computing

```
y[4000], y[4001], . . . , y[5999],
```

and thread 3 is responsible for computing

```
y[6000], y[6001], . . . , y[7999].
```

If a cache line contains eight consecutive **double**s, the only possibility for false sharing is on the interface between their assigned elements. If, for example, a single cache line contains

```
y[5996], y[5997], y[5998], y[5999],
y[6000], y[6001], y[6002], y[6003],
```

then it's conceivable that there might be false sharing of this cache line. However, thread 2 will access

```
y[5996], y[5997], y[5998], y[5999]
```

at the *end* of its **for** i loop, while thread 3 will access

```
y[6000], y[6001], y[6002], y[6003]
```

at the *beginning* of its iterations. So it's very likely that when thread 2 accesses, say,

y[5996], thread 3 will be long done with all four of

y[6000], y[6001], y[6002], y[6003].

Similarly, when thread 3 accesses, say, y[6003], it's very likely that thread 2 won't be anywhere near starting to access

y[5996], y[5997], y[5998], y[5999].

It's therefore unlikely that false sharing of the elements of y will be a significant problem with the 8000 × 8000 input. Similar reasoning suggests that false sharing of y is unlikely to be a problem with the 8,000,000 × 8 input. Also note that we don't need to worry about false sharing of A or x, since their values are never updated by the matrix-vector multiplication code.

This brings up the question of how we might avoid false sharing in our matrix- vector multiplication program. One possible solution is to "pad" the y vector with dummy elements to ensure that any update by one thread won't affect another thread's cache line. Another alternative is to have each thread use its own private storage during the multiplication loop, and then update the shared storage when they're done. (See Exercise 5.15.)

5.10 Tasking

While many problems are straightforward to parallelize with OpenMP, they generally have a fixed or predetermined number of parallel blocks and loop iterations to sched- ule across participating threads. When this is not the case, the constructs we've seen thus far make it difficult (or even impossible) to effectively parallelize the problem at hand. Consider, for instance, parallelizing a web server; HTTP requests may arrive at irregular times, and the server itself should ideally be able to respond to a potentially infinite number of requests. This is easy to conceptualize using a **while** loop, but recall our discussion in Section 5.5.1: **while** and **do while** loops cannot be parallelized with OpenMP, nor can **for** loops that have an unbounded number of iterations. This poses potential issues for dynamic problems, including recursive algorithms, such as graph traversals, or producer-consumer style programs like web servers. To address these issues, OpenMP 3.0 introduced *Tasking* functionality [47]. Tasking has been success- fully applied to a number of problems that were previously difficult to parallelize with OpenMP [1].

Tasking allows developers to specify independent units of computation with the task directive:

#pragma omp task

When a thread reaches a block of code with this directive, a new task is generated by the OpenMP run-time that will be scheduled for execution. It is important to note that the task will not necessarily be executed immediately, since there may be other tasks already pending execution. Task blocks behave similarly to a standard parallel region, but can launch an arbitrary number of tasks instead of only num_threads. In fact, tasks must be launched from within a parallel region but generally by only one of the threads in the team. Therefore a majority of programs that use Tasking functionality will contain an outer region that looks somewhat like:

```
# pragma omp parallel
# pragma omp single
  {
    ...
#   pragma omp task
    ...
  }
```

where the parallel directive creates a team of threads and the single directive in- structs the runtime to only launch tasks from a single thread. If the single directive is omitted, subsequent task instances will be launched multiple times, one for each thread in the team.

To demonstrate OpenMP tasking functionality, recall our discussion on parallelizing the calculation of the first *n* Fibonacci numbers in Section 5.5.2. Due to the loop-carried dependence, results were unpredictable and, more importantly, often in- correct. However, we *can* parallelize this algorithm with the task directive. First, let's take a look at a recursive serial implementation that stores the sequence in a global array called fibs:

```
int fib(int n) {
  int i = 0;
  int j = 0;
if (n <= 1) {
  // fibs is a global variable
  // It needs storage for n+1 ints
  fibs[n] = n;
```

```
        return n;
    }

    j = fib(n - 1);
    j = fib(n - 2);
    fibs[n] = i + j;
    return fibs[n];
}
```

This chain of recursive calls will be time-consuming, so let's execute each as a separate task that can run in parallel. We can do this by adding a parallel and a single directive before the initial (nonrecursive) call that starts fib, and adding **#pragma** omp task before each of the two recursive calls in fib. However, after we make this change, the results are incorrect—more specifically, except for fib[1], the sequence is all zeroes. This is because the default data scope for variables in tasks is private. So after completing each of the tasks

```
# pragma omp task i =
fib(n - 1);
# pragma omp task j =
fib(n - 2);
```

the results in iand jare lost: iand jretain their values from the initializations

```
int i = 0;
int j = 0;
```

at the beginning of the function. In other words, the memory locations that are as- signed the results of fib(n−1) and fib(n−2) are not the same as the memory locations declared at the beginning of the function. So the values that are used to update fibs[n] are the zeroes assigned at the beginning of the function.

We can adjust the scope of i and j by declaring the variables to be shared in the tasks that execute the recursive call. However executing the program now will produce unpredictable results similar to our original attempt at parallelization. The problem here is that the order in which the various tasks execute isn't specified. In other words, our recursive function calls, fib(n − 1) and fib(n − 2) will be run even- tually, but the thread executing the task that makes the recursive calls can continue to run and simply **return** the current value of fibs[n] early. We

need to force this task to wait for its subtasks to complete with the taskwait directive, which operates as a barrier for tasks. We've put this all together in Program 5.6.

```
1   int fib(int n) {
2       int i = 0;
3       int j = 0;
4
5       if (n <= 1) {
6               fibs[n] = n;
7           return n;
8       }
9
10  #   pragma omp task shared(i)
11      i = fib(n - 1);
12
13  #   pragma omp task shared(j)
14      j = fib(n - 2);
15
16  #   pragma omp taskwait
17          fibs[n] = i + j;
18        return fibs[n];
19  }
```

Program 5.6: Computing the Fibonacci numbers using OpenMP tasks.

Our parallel Fibonacci program will now produce the correct results, but you may notice significant slowdowns with larger values of n; in fact, there is a good chance that the serial version of the program executes much faster! To gain an intuition as to why this occurs, recall our discussion of the overhead associated with forking and joining threads. Similarly, each task requires its own data environment to be generated upon creation, which takes time. There are a few options we can use to help reduce task creation overhead. The first option is to only create tasks in situations where n is large enough. We can do this with the **if** directive:

```
#pragma omp task shared(i) if (n > 20)
```

which in this case will restrict task creation to only occur when n is larger than 20 (chosen arbitrarily in this case based on some experimentation). Reviewing fib again, we can see that there will be a task executing fib itself, another executing fib(n − 1), and a third executing fib(n − 2) for each recursive call. This is inefficient, because the parent task executing fib only launches two subtasks and then simply waits for their results. We can eliminate a task by having the parent thread perform one of the recursive calls to fib instead before doing the final calculation after the taskwait directive. On our 64-core testbed, these two changes halved the execution time of the program with $n = 35$.

=

While using the Tasking API requires a bit more planning and care to use— especially with data scoping and limiting runaway task creation—it allows a much broader set of problems to be parallelized by OpenMP.

5.11 Thread-Safety

Let's look at another potential problem that occurs in shared-memory programming: thread-safety. A block of code is **thread-safe** if it can be simultaneously executed by multiple threads without causing problems.

As an example, suppose we want to use multiple threads to "tokenize" a file. Let's suppose that the file consists of ordinary English text, and that the tokens are just contiguous sequences of characters separated from the rest of the text by white space—a space, a tab, or a newline. A simple approach to this problem is to divide the input file into lines of text and assign the lines to the threads in a round-robin fashion: the first line goes to thread 0, the second goes to thread 1, …, the t th goes to thread t, the t 1st goes to thread 0, and so on.

We'll read the text into an array of strings, with one line of text per string. Then we can use a parallel **for** directive with a schedule(**static** ,1) clause to divide the lines among the threads.

One way to tokenize a line is to use the strtok function in string.h. It has the following prototype:

```
char * strtok(
          char *        string       /* in/out */,
          const char *  separators   /* in     */);
```

Its usage is a little unusual: the first time it's called the string argument should be the text to be tokenized, so in our example it should be the line of input. For subsequent calls, the first argument should be NULL. The idea is that in the first call, strtok caches a pointer to string, and for subsequent calls it returns successive tokens taken from the cached copy. The characters that delimit tokens should be passed in separators, so we should pass in the string " \t\n" as the separators argument.

Given these assumptions, we can write the Tokenize function shown in Pro- gram 5.7. The main function has initialized the array lines so that it contains the input text, and line_count is the number of strings stored in lines. Although for our purposes, we only need the lines argument to be an input argument, the strtok function modifies its input. Thus when Tokenize returns, lines will be modified. When we run the program with a single thread, it correctly tokenizes the input stream. The first time we run it with two threads and the input

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

the output is also correct. However, the second time we run it with this input, we get the following output:

```
1  void Tokenize(
2          char *  lines[]          /* in/out */,
3          int     line_count       /* in     */,
4          int     thread_count     /* in     */) {
5      int my_rank, i, j;
6      char *my_token;
7
8  #   pragma omp parallel num_threads(thread_count) \
9          default(none) private(my_rank, i, j, my_token)          \
10         shared(lines, line_count)
11     {
12         my_rank = omp_get_thread_num();
13 #       pragma omp for schedule(static, 1)
14         for (i = 0; i < line_count; i++) {
15             printf("Thread %d > line %d = %s",
16                    my_rank, i, lines[i]);
17             j = 0;
18             my_token = strtok(lines[i], " \t\n");
19             while ( my_token != NULL ) {
20                 printf("Thread %d > token %d = %s\n",
21                        my_rank, j, my_token);
22                 my_token = strtok(NULL, " \t\n");
23                 j++;
24             }
25         } /* for i */
26     }  /* omp parallel */
27
28 }  /* Tokenize */
```

Program 5.7: A first attempt at a multi-threaded tokenizer.

```
Thread   0  > line 0 = Pease porridge hot.
Thread   1  > line 1 = Pease porridge cold.
Thread   0  > token 0 = Pease
Thread   1  > token 0 = Pease
Thread   0  > token 1 = porridge
Thread   1  > token 1 = cold.
Thread   0  > line 2 = Pease porridge in the pot
Thread   1  > line 3 = Nine days old.
Thread   0  > token 0 = Pease
Thread   1  > token 0 = Nine
Thread   0  > token 1 = days
Thread   1  > token 1 = old.
```

What happened? Recall that strtok caches the input line. It does this by declaring a variable to have static storage class. This causes the value stored in this variable to persist from one call to the next. Unfortunately for us, this cached string is shared, not private. Thus it appears that thread 1's call to strtok with the second line has apparently overwritten the contents of thread 0's call with the first line. Even worse, thread 0 has found a token ("days") that should be in thread 1's output.

The strtok function is therefore *not* thread-safe: if multiple threads call it simul- taneously, the output it produces may not be correct. Regrettably, it's not uncommon for C library functions to fail to be thread-safe. For example, neither the random num- ber generator rand in stdlib.h nor the time conversion function localtime in time.h is guaranteed to be thread-safe. In some cases, the C standard specifies an alternate, thread-safe version of a function. In fact, there is a thread-safe version of strtok:

```
char * strtok_r(
       char *          string      /* in/out */,
       const char *    separators  /* in     */,
       char **         saveptr_p   /* in/out */);
```

The "_r" is supposed to suggest that the function is *re-entrant*, which is sometimes used as a synonym for thread-safe.[9] The first two arguments have the same purpose as the arguments to strtok. The saveptr argument is used by strtok_r for keeping track of where the function is in the input string; it serves the purpose of the cached pointer in strtok. We can correct our original

Tokenize function by replacing the calls to strtok with calls to strtok_r. We simply need to declare a **char** variable to pass in for the third argument, and replace the calls in line 18 and line 22 with the following calls:

```
my_token = strtok_r(lines[i], " \t\n", &saveptr);
. . .
my_token = strtok_r(NULL, " \t\n", &saveptr);
```

respectively.

5.11.1 Incorrect programs can produce correct output

Notice that our original version of the tokenizer program shows an especially insid- ious form of program error: The first time we ran it with two threads, the program produced correct output. It wasn't until a later run that we saw an error. This, un- fortunately, is not a rare occurrence in parallel programs. It's especially common in shared-memory programs. Since, for the most part, the threads are running independently of each other, as we noted back at the beginning of the chapter, the exact sequence of statements executed is nondeterministic. For example, we can't say when thread 1 will first call strtok. If its first call takes place after thread 0 has tokenized its first line, then the tokens identified for the first line should be correct. However, if thread 1 calls strtok before thread 0 has finished tokenizing its first line, it's entirely possible that thread 0 may not identify all the tokens in the first line, so it's especially important in developing shared-memory programs to resist the temptation to assume that since a program produces correct output, it must be correct. We always need to be wary of race conditions.