MODULE-2

GPU programming, Programming hybrid systems, MIMD systems, GPUs, Performance —Speedup and efficiency in MIMD systems, Amdahl's law, Scalability in MIMD systems, Taking timings of MIMD programs, GPU performance.

2.1 GPU Programming

2.1 GPU Programming

GPU programming involves writing code for both the **CPU host** and the **GPU device**. Unlike general-purpose CPUs, **GPUs are not standalone processors**—they typically **don't run an operating system** or provide services like **direct access to secondary storage**. Hence, the **host CPU** manages the **allocation and initialization** of memory on both itself and the GPU, launches the **GPU kernel**, and collects the **resulting output**. This style of programming is called **heterogeneous programming** because it involves **two types of processors** working together.

Each **GPU processor** can run **hundreds to thousands of threads**, and they share a **large block of memory**. Additionally, every GPU processor has a **smaller, faster memory block**, accessible only to its own threads—serving as a **programmer-managed cache**.

Threads are organized into **groups**, where threads **within a group follow the SIMD model**. Threads **across groups** can run **independently**, but **within a group**, **no thread can proceed to the next instruction until all threads complete the current one**. This impacts performance especially when **branching occurs within a SIMD group**.

Here is an example snippet from GPU thread execution:

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

In this example, **threads with rank_in_gp < 16** will execute the my_x += 1 instruction, while the **rest are idle**. Then, **threads with rank_in_gp ≥ 16** execute my_x -= 1, while the others idle. This **two-phase execution** causes **resource underutilization**, and it's up to the programmer to **minimize such branching** for efficiency.

Another distinct feature of GPUs is their **hardware-based thread scheduler**. Unlike CPUs that use **software to schedule tasks**, the **GPU scheduler** incurs **very low overhead**. It **executes a SIMD group only when all its threads are ready**, thus requiring **careful data preparation**, such as **registering rank_in_gp in advance**.

To optimize throughput, programmers often **create multiple SIMD groups**. When one group is **waiting for memory or dependent operations**, another group can be

scheduled immediately, which helps to **hide memory latency** and **maximize GPU utilization**.

2.1.2 Programming Hybrid Systems

In modern parallel computing, it is possible to **program hybrid systems**, such as **clusters of multicore processors**, by using a **combination of APIs**. Typically, a **shared-memory API** (such as OpenMP) is used **within each node**, while a **distributed-memory API** (like MPI) is used for **communication between nodes**. This approach allows the system to **exploit both shared and distributed memory architectures** simultaneously.

However, this **hybrid programming model** is generally adopted only in cases where the **highest possible performance** is required. The reason is that using **two different APIs** in the same program significantly **increases development complexity**. The programmer must manage both **intra-node synchronization** and **inter-node message passing**, which makes the software harder to design, debug, and maintain.

As a result, many such systems are programmed using only a **single distributed-memory API**, like MPI, to handle **both inter-node and intra-node communication**. While this may not be the most efficient approach in terms of raw performance, it offers **simplicity** and **portability**, which are often more valuable in practical application development.

2.2 Input and Output

When it comes to **parallel computing**, **input and output (I/O)** present unique challenges. In most basic programs, **I/O operations are minimal** and handled using standard **C I/O functions** such as printf, fprintf, scanf, and fscanf. However, when these functions are used in **parallel programs**, issues can arise due to their **sequential nature** and lack of built-in support for concurrent access.

2.2.1 MIMD systems

In **MIMD systems**, each **process may attempt to access stdin, stdout, or stderr**. Since these standard streams were designed for serial usage, when **multiple processes or threads** try to read from or write to them, the behavior becomes **nondeterministic**. For example, output might appear in a **different order** each time the program is executed, or worse, the **output from one process might interleave** with the output from another.

In practical parallel systems, it is common for **only one process (usually process 0)** to read input via scanf, and typically only that process will **access stdin**. All processes may be allowed to **write to stdout**, but this often leads to **interleaved or jumbled output**, especially during debugging.

To ensure clarity and correctness, the following **guidelines are often followed** in I/O handling:

- In **distributed-memory programs**, only **process 0 accesses stdin**.
- In **shared-memory programs**, **thread 0 or the master thread** reads from stdin.

- **All processes or threads can write to stdout and stderr**, but usually only **one** is designated to do so, to preserve output order.
- Each process or thread should use a **private file** for reading or writing to avoid file conflicts.
- **Debugging output** should always include the **process or thread ID** to identify the source.

For example:

printf("Process %d: value = %d\n", rank, value);

This helps identify **which process** produced the output.

2.2.2 GPUs and I/O

In **GPU programming**, I/O is typically performed by the **host CPU code**. The **GPU device** itself does not have access to **operating system services**, so it cannot interact directly with **secondary storage**, stdin, or stderr.

Thus, in GPU programs:

- **All I/O operations** (input, output, file access) are handled by the **host**.
- **Only one process/thread** runs on the host, so standard C I/O behaves **as expected**.
- During **GPU kernel debugging**, threads may **write to stdout**, but the output will again be **nondeterministic**.
- **GPU threads do not have access to stderr, stdin, or files**.

This I/O design in GPU systems ensures that data is loaded and output by the **host** and not the device itself. During debugging, you can enable selective printf statements inside the kernel, but the output should not be relied upon for sequence or consistency across runs.

2.3 Performance

The main motivation for writing **parallel programs** is to achieve better **performance**. To evaluate this, we examine **speedup** and **efficiency**, especially in **homogeneous MIMD systems**, where all cores share the same architecture.

2.3.1 Speedup and Efficiency in MIMD Systems

In the ideal case, a program using p cores would run p times faster than its serial counterpart, assuming perfect workload division and no additional overhead.

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

Let:

- T_serial = Time taken by the serial program
- T_parallel = Time taken by the parallel version

Then:

Speedup:

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

If $S = p$, we have **linear speedup**.

Efficiency:

$$E = \frac{S}{p} = \frac{T_{serial}}{p \cdot T_{parallel}}$$

This gives the **average utilization** of each core. Efficiency decreases as **parallel overhead** increases with more threads or processes.

Calculating Efficiency

Suppose T_serial = 24 ms, T_parallel = 4 ms, and p = 8:

$$E = \frac{24}{8 \cdot 4} = \frac{3}{4} = 0.75$$

Each core works for 3 ms on actual problem solving and 1 ms is spent in overhead.

Common overheads include:

- In **shared-memory** systems: **mutexes** used in critical sections slow down execution.
- In **distributed-memory** systems: **communication delays** across the network reduce performance.

Thus, as p increases:

- Speedup increases **sub-linearly**
- Efficiency **decreases**

Problem Size and Its Impact

Speedup and efficiency also depend on **problem size**. If the problem is:

- **Smaller** → less work per core → higher overhead proportion
- **Larger** → more work per core → better efficiency

Reference: Table 2.5

Table 2.5 Speedups and efficiencies of 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

Figure 2.18 shows the speedup curve.

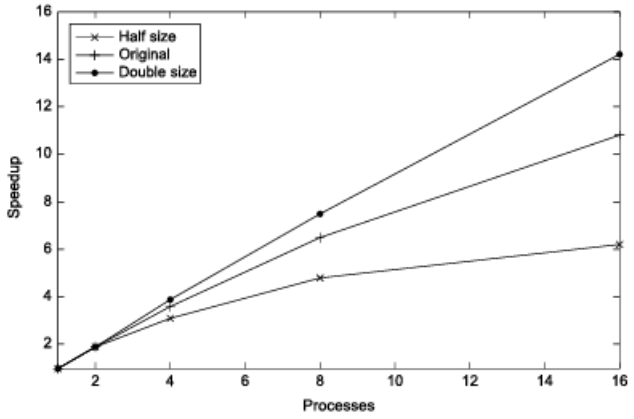

FIGURE 2.18

Speedups of parallel program on different problem sizes.

Figure 2.19 shows the efficiency trend.

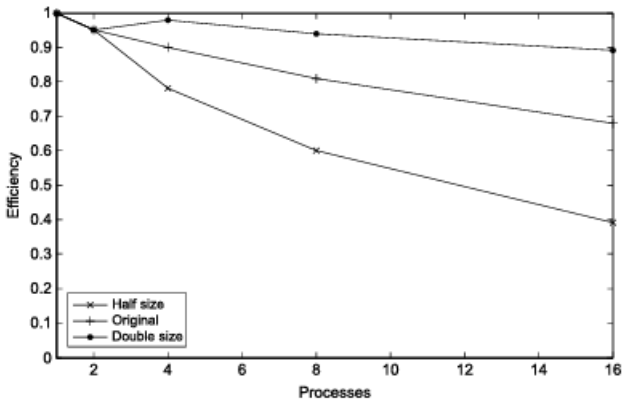

FIGURE 2.19

Efficiencies of parallel program on different problem sizes.

As seen in both figures, increasing problem size improves both **speedup** and **efficiency**, because **T_serial** grows faster than overhead **Toverhead**.

Overhead Model

In many programs:

Tparallel=Tserialp+ToverheadT_{parallel} = \frac{T_{serial}}{p} + T_{overhead}Tparallel =pTserial+Toverhead

The total runtime is the sum of time spent solving the actual problem and the time spent in synchronization, communication, etc.

Choosing T_serial for Comparison

There are two approaches:

- Use the fastest serial algorithm on the fastest system (**idealistic**)
- Use the **serial version of the parallel program** running on one processor of the **same system** (**practical and common**)

The second method better reflects the **realistic utilization** of available cores and is used in most analyses.

2.3.2 Amdahl's Law

Amdahl's Law, formulated by **Gene Amdahl** in the 1960s, highlights a significant limitation in parallel computing. It states that unless **virtually all of a serial program is parallelized**, the overall **speedup** achievable will be very limited, no matter how many cores are available. Suppose 90% of a serial program is successfully parallelized. Assuming **perfect parallelization**, the **parallel portion** of the program will scale with the number of cores, p, and the speedup for that part will be p.

If the **serial run-time** is T_{serial}=20 seconds, the **parallelized part** would run in $0.9 \times T_{\text{serial}}/p = \frac{18}{p}$, while the **remaining serial part** would take2 $0.1 \times T_{\text{serial}} = 2$ seconds. . Therefore, the total **parallel run-time** becomes

$$T_{\text{parallel}} = \frac{18}{p} + 2$$

and the **speedup** is

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

As p becomes very large, the term $\frac{18}{p}$ becomes smaller, approaching 0. The total **parallel run-time** can never be smaller than 2 seconds, which means the speedup is limited to

$$S \leq \frac{20}{2} = 10$$

Thus, **even with perfect parallelization** of 90% of the program and an **infinite number of cores**, the maximum speedup will not exceed **10**.

In general, if a fraction r of the program remains **unparallelized**, **Amdahl's Law** asserts that the speedup cannot exceed

$$S \leq \frac{1}{r}$$

In the example $r = 1 - 0.9 = 0.1$, hence $S \leq 10$. If r is as small as 1/100, the maximum speedup is still limited to 100, regardless of the number of processors. This result emphasizes that **even a small serial portion can severely limit the total performance gain**.

Although this may seem discouraging, there are reasons not to be overly concerned. The law does not take **problem size** into account. As the problem size increases, the fraction of the code that is inherently serial often becomes smaller. A more mathematical perspective of this is expressed in **Gustafson's Law**. Additionally, many programs used in **scientific and engineering domains** routinely achieve substantial speedups on large distributed-memory systems. In practice, a speedup of **5 or 10** may be sufficient, especially when the **effort to parallelize** the program is not significant.

2.3.3 Scalability in MIMD Systems

The term **"scalable"** is commonly used in informal contexts to describe how well a system or program handles growth. In parallel computing, **scalability** refers to whether a program continues to demonstrate performance gains as the **computing power increases**, for example, by increasing the **number of cores**. In a more formal context, especially for **MIMD (Multiple Instruction, Multiple Data)** systems, a program is said to be **scalable** if increasing the number of processes or threads and the problem size proportionally results in **unchanged efficiency**.

Let us consider a scenario where a parallel program is run with a fixed number of threads/processes and a fixed input size, yielding an efficiency EEE. If we increase the number of processes/threads and also increase the problem size such that the efficiency remains constant, then the program is **scalable**.

Suppose the **serial run-time** is given by

$$T_{\text{serial}} = n$$

where n represents both time (in microseconds) and problem size. Also, suppose the **parallel run-time** is given by

$$T_{\text{parallel}} = \frac{n}{p} + 1$$

The **efficiency** is then

$$E = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}} = \frac{n}{n+p}$$

To test **scalability**, let us scale the number of processes/threads by a factor of k, resulting in kp threads. We now want to find the scaling factor x such that the **problem size** becomes xn, and the efficiency remains the same.

The new efficiency becomes

$$E = \frac{xn}{xn + kp}$$

We want this to equal the original efficiency:

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

If we let $x = k$, then

$$\frac{xn}{xn + kp} = \frac{kn}{kn + kp} = \frac{n}{n+p}$$

This confirms that **increasing the problem size by the same factor as the number of processes/threads** maintains constant efficiency, thus making the program **scalable**.

There are two specific cases of scalability:

- If the **efficiency remains constant without increasing the problem size**, even when the number of threads increases, the program is called **strongly scalable**.
- If the **efficiency remains constant only when the problem size increases at the same rate as the number of threads**, the program is called **weakly scalable**.

In the above example, the program would be classified as **weakly scalable** since it requires proportional scaling of both threads and problem size to maintain efficiency.

2.3.4 Taking Timings of MIMD Programs

To evaluate the **performance of parallel programs**, it's essential to determine the values of $\mathbf{T_{serial}}$ and $\mathbf{T_{parallel}}$ Depending on the phase of development, we may take timings for different purposes. During **program development**, detailed timings help identify performance issues like **message waiting time** in distributed-memory programs. However, when evaluating final performance, we often focus on **overall execution time**, reporting just a **single elapsed value**.

In performance studies, we're generally not concerned with the **total program runtime** (from launch to exit), but only the **runtime of critical code sections**, such as the sorting

phase in a sorting algorithm. Thus, general-purpose tools like the time command in Unix are often unsuitable.

Moreover, we rarely use **CPU time** (e.g., reported by clock() in C), which only accounts for time actively spent executing code. It **excludes idle time**, such as when a process is blocked waiting for communication. For example, if a process is waiting in a receive() call, it might be sleeping and not using the CPU, but it still affects the **real-time performance** of the application.

Hence, we typically use **wall clock time**, which measures the **actual elapsed time** from start to end of a code segment. It's equivalent to manually starting and stopping a stopwatch. The timing code would look like:

```
double start, finish;
start = Get_current_time();
/* Code to be timed */
finish = Get_current_time();
printf("Elapsed time = %e seconds\n", finish - start);
```

Here, Get_current_time() is a placeholder. In **MPI**, this could be MPI_Wtime(), while in **OpenMP**, we might use omp_get_wtime(). Both return wall clock time.

The **resolution** of the timing function (smallest measurable time unit) is critical. If a timer only has **millisecond resolution**, very fast code (e.g., nanoseconds per instruction) may appear to take zero time. Some APIs provide functions to **query the timer resolution**, while others guarantee a specific resolution. As developers, we must **verify timer accuracy**.

For **parallel programs**, we often care about **global elapsed time**—the time from when the **first thread/process starts** until the **last one finishes**. This differs from each process measuring its own time. A typical strategy is:

```
shared double global_elapsed;
private double my_start, my_finish, my_elapsed;
Barrier();
my_start = Get_current_time();
/* Code to be timed */
my_finish = Get_current_time();
my_elapsed = my_finish - my_start;
global_elapsed = Global_max(my_elapsed);
if (my_rank == 0)
   printf("Elapsed time = %e seconds\n", global_elapsed);
```

Here, a **barrier** ensures all processes begin approximately together. After timing, a **global maximum** of elapsed times is computed to represent the total parallel runtime.

It's also important to consider **variability in timing results**. Even with the same input and system, runtime often fluctuates due to system noise. Instead of reporting **mean or median**, we generally report the **minimum runtime**, assuming it's the best approximation of true performance.

Running **more than one thread per core** increases variability and overhead due to thread scheduling. Hence, it is best to run **one thread per core** for consistent and optimal timing.

Finally, since **I/O is not the focus**, we **exclude file or console input/output** from performance timings.

2.3.5 GPU Performance

When evaluating the **performance of parallel programs**, we often compare them against equivalent **serial programs**. This approach works well for **MIMD systems**, where we assume that both the serial and parallel versions are executed on **identical types of cores**. However, this assumption doesn't hold for **GPU programs**, because **GPU cores are fundamentally different** from traditional **CPU cores**. As a result, using concepts like **efficiency** or **linear speedup**—common in MIMD performance evaluation—does **not make sense for GPU programs**.

Even though we cannot compute **efficiency** in the traditional sense, it's common to report **speedups** of GPU programs over either serial CPU programs or **parallel MIMD programs**. However, since GPU cores are **highly parallel in nature**, comparing them to serial CPUs using standard metrics like efficiency is generally **inapplicable**.

The same applies to **scalability**. While the **formal definition of scalability** used for MIMD systems doesn't apply, we often use an **informal definition**: a **GPU program is considered scalable** if it can run faster on a **larger GPU** compared to a **smaller one** when the problem size increases accordingly.

There is a case where **Amdahl's Law** becomes relevant to GPU performance. If the **serial portion** of a program is executed on a **conventional CPU** and the **parallel portion** is offloaded to the GPU, then **Amdahl's Law** provides a bound on the maximum achievable speedup. Specifically, if **a fraction r** of the total execution remains serial, the **maximum possible speedup** is limited to 1/r, even when using a GPU.

Just like in MIMD programs, the limitations of Amdahl's Law still apply: the **serial portion might shrink** as the **problem size increases**, and in practice, **GPU programs often achieve significant speedups**. Moreover, even a **modest speedup** may be **sufficient**, depending on the application.

In terms of **timing GPU programs**, we can use the **CPU's timer** if we are measuring the entire duration of the GPU task. This involves starting the timer **before launching the GPU kernel** and stopping it **after the GPU computation completes**. This method is sufficient for most cases where the entire program's performance is being measured. However, if we need to time **specific portions of GPU code**, we must use the **timing facilities provided by the GPU's API**.