

PARALLEL COMPUTING		Semester	VII
Course Code	BCS702	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	SEE Marks	50
Total Hours of Pedagogy	40 hours Theory + 8-10 Lab slots	Total Marks	100
Credits	04	Exam Hours	03
Examination nature (SEE)	Theory/Practical		

Course outcomes (Course Skill Set):

At the end of the course, the student will be able to:

- Explain the need for parallel programming
- Demonstrate parallelism in MIMD system.
- Apply MPI library to parallelize the code to solve the given problem.
- Apply OpenMP pragma and directives to parallelize the code to solve the given problem
- Design a CUDA program for the given problem.

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.

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, programming 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.

```
// 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 ≥ 16 are idle.

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.

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.

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

1. Speedup and efficiency in MIMD systems

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

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

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

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

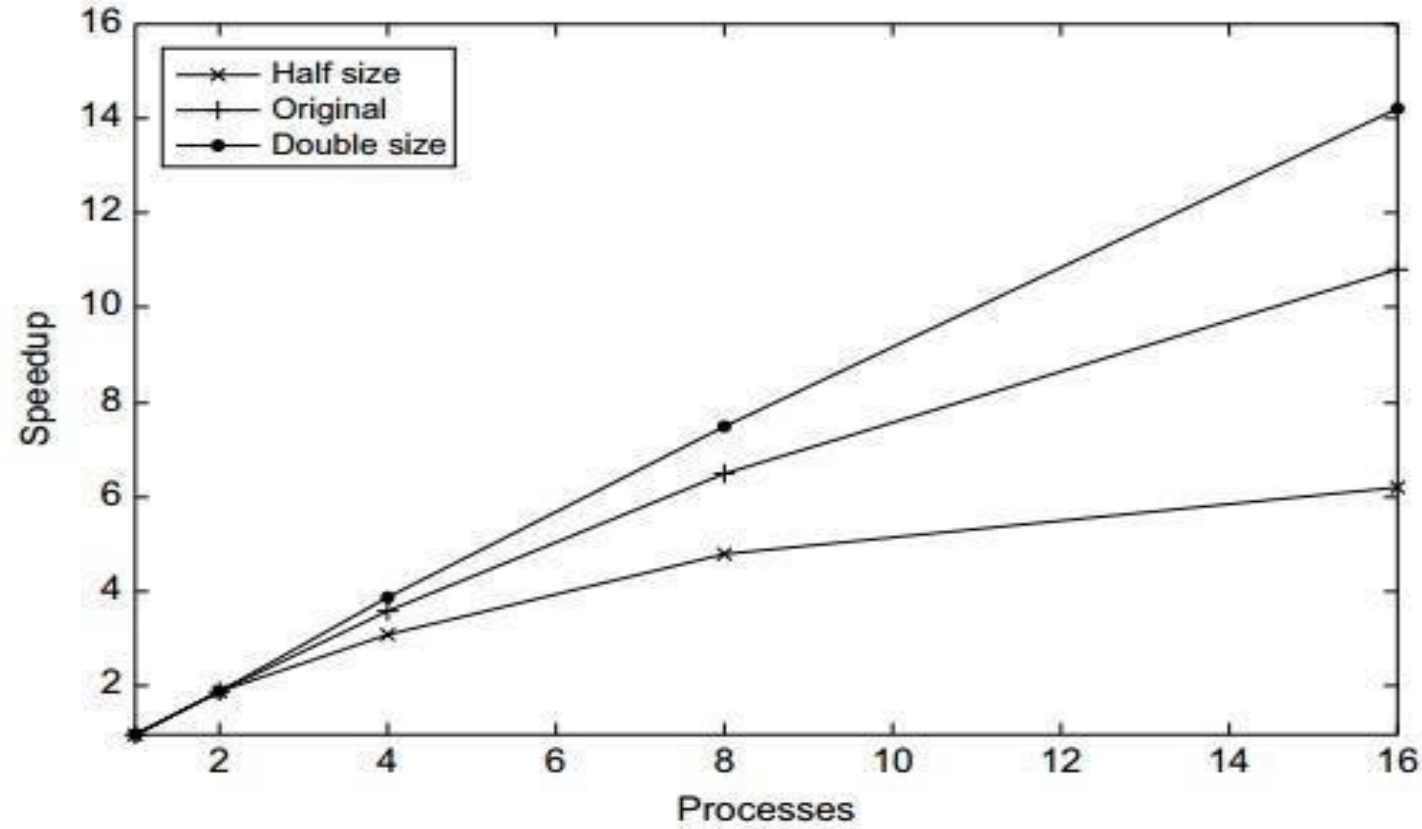


FIGURE 2.18

Speedups of parallel program on different problem sizes

2. Amdahl's law

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

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

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

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

and the speedup will be

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

3. Scalability in MIMD systems

As an example, suppose that $T_{\text{serial}} = n$, where the units of T_{serial} are in microseconds, and n is also the problem size. Also suppose that $T_{\text{parallel}} = n/p + 1$. Then

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

4. Taking timings of MIMD programs

The first thing to note is that there are at least two different reasons for taking timings. During program development we may take timings in order to determine if the program is behaving as we intend.

Second, we're usually *not* interested in the time that elapses between the program's start and the program's finish.

Third, we're usually *not* interested in "CPU time." This is the time reported by the standard C function clock.

5 GPU performance

It's quite common to see reported speedups of GPU programs over serial programs or parallel MIMD programs.

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.

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:

Thank you