

MODULE – 2

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

INPUT AND OUTPUT

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.

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?

1. Speedup and efficiency

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 $T_{\text{parallel}} = T_{\text{serial}}/p$. 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 necessary “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.

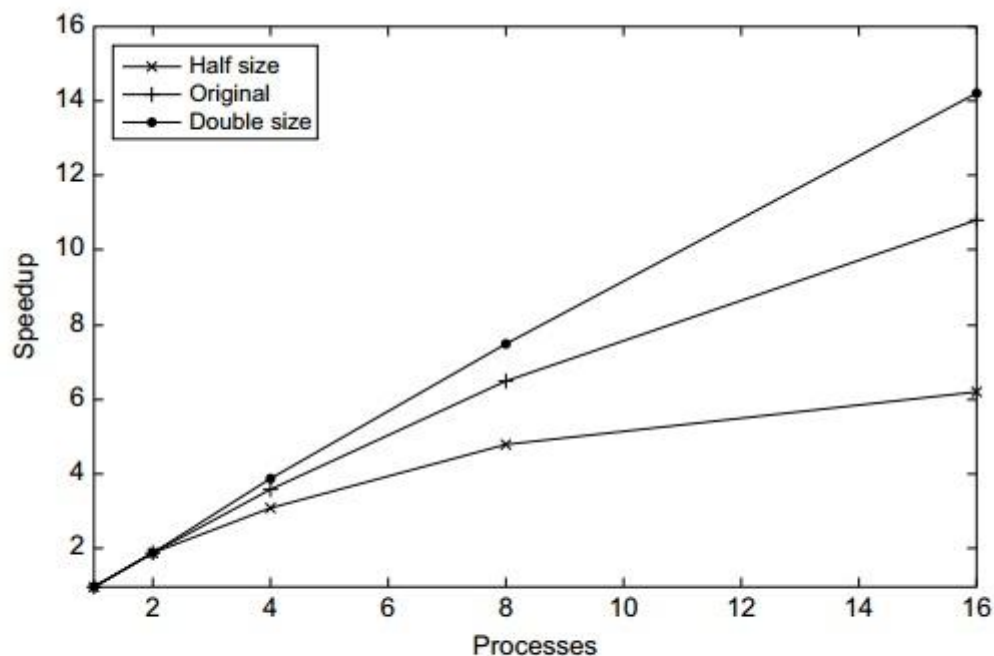
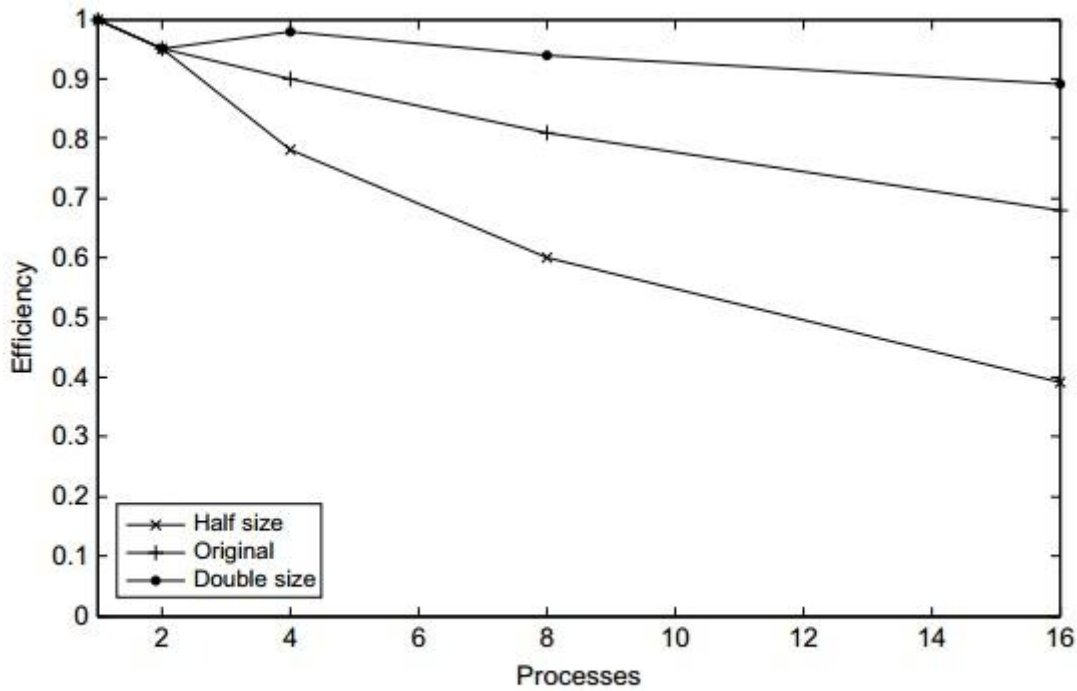


FIGURE 2.18

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 parallelized, 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 $T_{\text{serial}} = 20$ seconds, then the run-time of the parallelized part will be $0.9 \times T_{\text{serial}}/p = 18/p$ and the run-time of the “unparallelized” part will be $0.1 \times T_{\text{serial}} = 2$. 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, $0.9 \times T_{\text{serial}}/p = 18/p$ gets closer and closer to 0, so the total parallel run-time can't be smaller than $0.1 \times T_{\text{serial}} = 2$. 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

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

That is, $S \leq 10$. 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 consideration 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

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

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

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

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 program'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 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 deschedule 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.