or, at best, unpredictable performance, since each time a running process accessed memory, it might access local memory—that is, memory belonging to the core on which it was executing—or remote memory, memory belonging to another core. Accessing remote memory can take hundreds or even thousands of times longer than accessing local memory. As an example, consider the following pseudocode for a shared-memory vector addition:

```
   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 ];
```

We first declare two shared arrays. Then, on the basis of the process's rank, we determine which elements of the array "belong" to which process. After initializing the arrays, each process adds its assigned elements. If the assigned elements of `x` and `y` have been allocated so that the elements assigned to each process are in the memory attached to the core the process is running on, then this code should be very fast. However, if, for example, all of `x` is assigned to core 0 and all of `y` is assigned to core 1, then the performance is likely to be terrible, since each time the assignment `x[i] += y[i]` is executed, the process will need to refer to remote memory.

**Partitioned global address space**, or PGAS, languages provide some of the mechanisms of shared-memory programs. However, they provide the programmer with tools to avoid the problem we just discussed. Private variables are allocated in the local memory of the core on which the process is executing, and the distribution of the data in shared data structures is controlled by the programmer. So, for example, she knows which elements of a shared array are in which process's local memory.

There are several projects currently working on the development of PGAS languages. See, for example, [8,54].

### 2.4.5 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.



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 programmermanaged 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 ≥ 16 are idle. After the threads with rank < 16 are done, the roles will be reversed: 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.

Another issue in GPU programming that's different from CPU programming is how the threads are scheduled to execute. GPUs use a hardware scheduler (unlike CPUs, which use software to schedule threads and processes), and this hardware scheduler uses very little overhead. However, the scheduler will choose to execute an instruction when all the threads in the SIMD group are ready. In the preceding example, before executing the test, we would want the variable `rank_in_gp` stored in a register by each thread. So, to maximize use of the hardware, we usually create a large number of SIMD groups. When this is the case, groups that aren't ready to execute (e.g., they're waiting for data from memory, or waiting for the completion of a previous instruction) can be idled, and the scheduler can choose a SIMD group that is ready.

### 2.4.6 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.

---

## 2.5 Input and output

### 2.5.1 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, [38]. 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/threads, we, as developers, usually want 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, with processes, 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/threads is a little less obvious. Should the input be divided among the processes/threads? Or should only a single process/thread be allowed to call `scanf`? The vast majority of systems allow at least one process to call `scanf`—usually process 0—while most allow multiple threads to call `scanf`. Once again, there are some systems that don't allow any processes to call `scanf`.

When multiple processes/threads *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/thread may be broken up by the output of another process/thread. For input, the data read by each process/thread may be different on each run, even if the same input is used.

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 sharedmemory 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.

### 2.5.2 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 debugging 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.

---

## 2.6 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.

### 2.6.1 Speedup and efficiency in MIMD systems

Usually the best our parallel program can do is to divide the work equally 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 runs on a single core of the same design. If we call the serial run-time $T_{serial}$ and our parallel run-time $T_{parallel}$, then it's usually the case that the best possible run-time

**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 |

of our parallel program is $T_{parallel} = T_{serial}/p$. When this happens, we say that our parallel program has **linear speedup**.

In practice, we usually don't get perfect 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 the 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 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. For example, more threads will probably mean more threads need to access a critical section, and 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 $S = p$. Furthermore, since as p increases we expect the parallel overhead to increase, we also 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.[1] 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

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

If the serial run-time has been taken on the same type of core that the parallel system is using, we can think of efficiency as the average utilization of the parallel

**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 |

cores on solving the problem. That is, the efficiency can be thought of as the fraction of the parallel run-time that's spent, on average, by each core working on solving the original problem. The remainder of the parallel run-time is the parallel overhead. This can be seen by simply multiplying the efficiency and the parallel run-time:

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

For example, suppose we have $T_{\text{serial}} = 24$ ms, $p = 8$, and $T_{\text{parallel}} = 4$ ms. Then

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

and, on average, each process/thread spends 3 $\cdot 4 = /4$ 3 ms on solving the original problem, and $4 - 3 = 1$ ms in parallel overhead.

Many parallel programs are developed by explicitly dividing the work of the serial program among the processes/threads and adding in the necessary "parallel

[1] These data are taken from Chapter 3. See Tables 3.6 and 3.7.

overhead," such as mutual exclusion or communication. Therefore if $T_{overhead}$ denotes this parallel overhead, it's often the case that

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

When this formula applies, it's clear that the parallel efficiency is just the fraction of the parallel run-time spent by the parallel program in the original problem, because the formula divides the parallel run-time into the time on the original problem, $T_{serial}/p$, and the time spent in parallel overhead, $T_{overhead}$.

We've already seen 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 Fig. 2.18, and the efficiencies are plotted in Fig. 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.

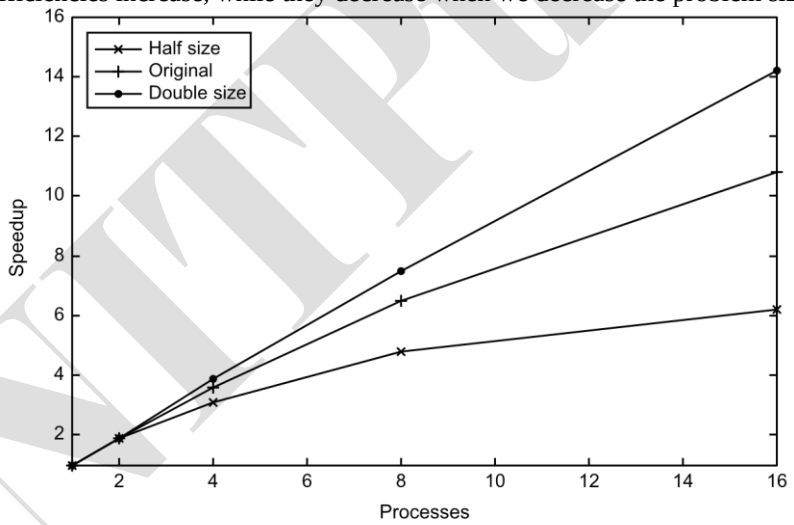


**FIGURE 2.18**

Speedups of parallel program on different problem sizes.

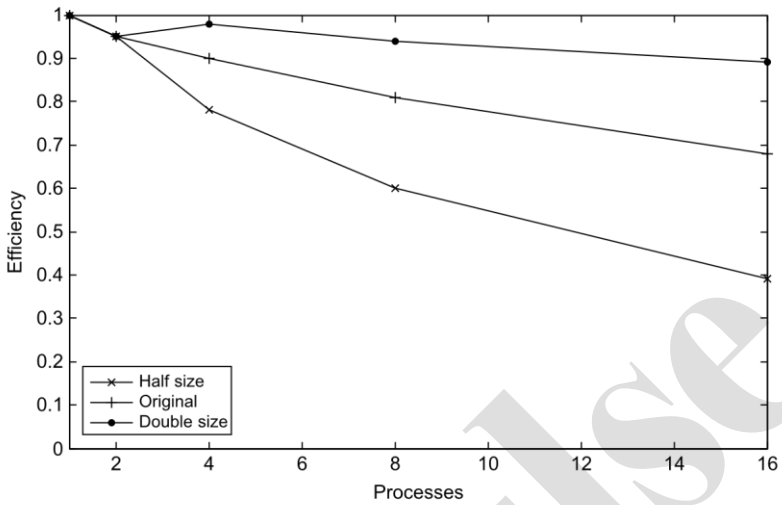


**FIGURE 2.19**

Efficiencies of parallel program on different problem sizes.

This behavior is quite common, because in many parallel programs, as the problem size is increased but the number of processes/threads is fixed, the parallel overhead grows much more slowly than the time spent in solving the original problem. That is, if we think of $T_{serial}$ and $T_{overhead}$ as functions of the problem size, $T_{serial}$ grows much faster as the problem size is increased. Exercise 2.15 goes into more detail.

A final issue to consider is what values of $T_{serial}$ should be used when reporting speedups and efficiencies. Some authors say that $T_{serial}$ should be the run-time of the fastest program on the fastest processor available. However, since it's often the case that we think of efficiency as the utilization of the cores on the parallel system, 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 parallel 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.

### 2.6.2 Amdahl's law

Back in the 1960s, Gene Amdahl made an observation [3] 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. Furthermore, 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 $0.9 \times T_{serial}/p = 18/p$ and the run-time of the "unparallelized" part will be

$0.1 \times T_{serial} = 2$. The overall parallel run-time will be

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

and the speedup will be

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

T

S.

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

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

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.

### 2.6.3 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 program is scalable if, by increasing the power of the system it's run on (e.g., increasing the number of cores), we can obtain speedups over the program when it's run on a less powerful system (e.g., a system with fewer cores). However, in discussions of MIMD 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; 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.

### 2.6.4 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 timings. During program development, we may take timings 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 2.21 for a brief discussion of one issue 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. So 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 ) ;
i f    ( my_rank == 0)
    printf ( "The elapsed time = %e seconds\n" , 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 processes/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 processes/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 highperformance I/O, we'll usually not include I/O in our reported run-times.

### 2.6.5 GPU performance

In our discussion of MIMD performance, there is an assumption that we can evaluate a parallel program by comparing its performance to the performance of a serial program. We can, of course, 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.

However, as we noted in our discussion of efficiency of MIMD programs, we often assume that the serial program is run on the same type of core that's used by the parallel computer. Since GPUs use cores that are inherently parallel, this type of comparison usually doesn't make sense for GPUs. So efficiency is ordinarily not used in discussions of the performance of GPUs.

Similarly, 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.

Note that 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.