Overview of parallel programming

Raymond J. Spiteri

Lecture Notes for CMPT 898:
Numerical Software

University of Saskatchewan

January 31, 2013

Objectives

- Overview of parallel programming
- Types of parallel decomposition
- Programming for shared / distributed memory
- Parallel program design

Introduction

The state of parallel programming is not at the same level that of parallel hardware.

There is plenty of commodity parallel hardware but relatively little commodity software.

As mentioned, we cannot rely only on hardware improvements for increased application performance.

Before discussing some of the broad issues involved in parallel programming, we establish some terminology.

When executing shared-memory programs, we start a single process, from which we fork multiple threads to carry out tasks.

When executing distributed-memory programs, we start multiple processes to carry out tasks.

It may be possible to interchange these words according to the context; i.e., processes in the context of a distributed-memory program correspond to threads in the context of a shared-memory program.

Introduction

Thus the concept of a process/thread is a fundamental building block in parallel programming.

A process/thread is an instance of a program (or sub-program) that is executing autonomously (more or less) on a physical processor.

A program is said to be parallel if at any time it consists of more than one process/thread.

In order to have useful parallel programs, there must be ways to create, specify, and destroy processes/threads as well as ways for processes/threads to interact.

We now give a brief overview of the three main parallel programming paradigms used in this course:

- shared-memory programming (OpenMP)
- message passing (MPI)
- hybrid model (Matlab)

SPMD

From this point on, we will only discuss the programming of MIMD systems.

Our focus will be on single-program, multiple-data (SPMD) programs.

Instead of running a different program on each core, we imagine one program run on each core but each core is made to do different things via conditional statements.

For example,

```
if (I am process/thread 0)
   do this;
else
   do that;
```

This is an example of so-called functional (or task) *decomposition*.

SPMD

On the other hand, the code

```
if (I am process/thread 0)
   work on first part of data
else /* I am process/thread 1 */
   work on second part of data
```

is an example of data parallelism (or domain decomposition).

SIMD is a special case of data parallelism.

These are the two main kinds of decompositions for the purposes of parallelization.

In practice, most parallel programs use both decompositions to achieve parallelism.

Typically the first step in designing a parallel algorithm is to decompose the problem into smaller problems that can be assigned to different processors to work on simultaneously.

Domain decomposition (or data parallelism)

A data-parallel algorithm is a sequence of elementary instructions applied to (different) data.

In other words, a new instruction is initiated only after the previous instruction terminates.

The advantage is that there is a single flow of control.

Data are divided into pieces of approximately the same size and mapped to different processors.

Each processor works only on the portion of the data that it is assigned.

Of course, the processes may need to communicate periodically in order to exchange data.

Domain decomposition (or data parallelism)

SPMD strategies are commonly employed in finite-difference / finite-element algorithms where processors can operate independently on large blocks of data and exchange only the much smaller shared border data at each iteration.

However, they are only really effective when working with regular structures such as dense matrices (or analogously, rectangular domains and Cartesian grids).

In other words, domain decomposition does not work as well (or is significantly more complicated) when dealing with unstructured (e.g., triangular) meshes or irregular geometries.

This may bias the choice of algorithms to solve a given problem towards those that use regular data structures.

Unless the data structure is static and highly regular (e.g., a dense matrix), the general problem of mapping data to processors is not trivial!

Co-ordinating processes/threads

On rare occasions, obtaining excellent parallel performance is easy.

For example, suppose we want to add two arrays:

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

To parallelize, we map the array elements to the processes/threads; e.g., if we have p processes/threads, we could assign elements 0 to $n/p-1$ to process/thread 0, elements n/p to $2n/p-1$ to process/thread 1, etc.

The two issues in general are only to ensure that each process/thread gets the same amount of work and the amount of communication is minimized.

Both of these issues are trivially addressed in this example, so much so that you may see the problem called embarrassingly parallel.

Co-ordinating processes/threads

The concept of data locality is critical in data-parallel programming in any environment where processor communication is not negligible.

In these situations, communication is much more expensive than computation, and the difference can be dramatic.

Thus, considerable research has gone into optimal data mapping, i.e., how to assign data to processors to minimize communication.

Of course the trivial (and likely optimally *inefficient*) solution is to map all the data to one node.

So the concept of load balancing is a competing goal to data locality: we would like all the processors to be fully utilized all the time.

Any effective mapping must take into account both data locality and load balancing.

Functional decomposition (or task parallelism)

When the data assigned to the different processors require greatly different lengths of time to process, domain decomposition will not be effective.

The performance of the code will be limited by the speed of the slowest process; i.e., this becomes a bottleneck for the computation.

Also, the remaining idle processes do no useful work.

In this case, functional decomposition (or task parallelism) may make more sense.

In task parallelism, the problem is decomposed into smaller tasks, and the tasks are assigned to the processors as they become available.

This way processors that finish quickly can be assigned more tasks.

Client-server paradigm

Task parallelism is typically implemented in a client-server paradigm.

A master process (the server) is in charge of allocating tasks to a number of worker processes (the clients).

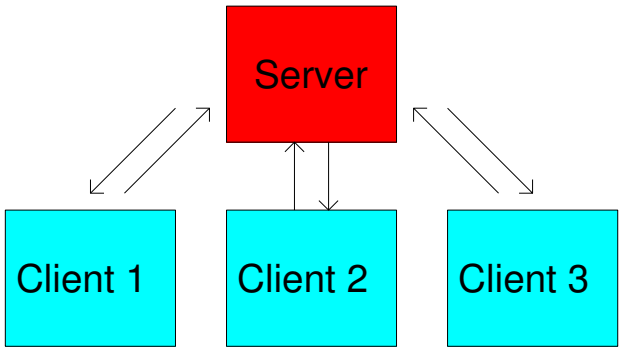

Figure 1: The client-server paradigm

The master process may also assign tasks to itself.

The client-server paradigm can be implemented at virtually any level in a program.

There are a few different ways for the server to assign (or schedule) jobs; some are more efficient than others depending on the problem.

Suppose there are J jobs and p processors.

If each job is (approximately) the same amount of work, then any (approximately) J/p jobs can be assigned to each server *a priori* (static scheduling), resulting in (approximately) optimal load balancing and communication.

If each job has a different but predictable amount of work, it may be possible to order the jobs differently and still achieve reasonable load balancing.

For example, if the work per job decreases as a function of job number, then jobs 1 to p can be assigned to processors 1 to p, but jobs $p+1$ to $2p$ can be assigned to processors p to 1, etc.

Static scheduling will not work well if the jobs have very different and relatively unpredictable workloads.

In dynamic scheduling, any p jobs are assigned to the p processors; then once a processor finishes its task, it can be assigned a new one until all J jobs are done.

It also helps if a client processor could become the server if it was assigned a particularly tough job.

Shared-memory programming

A shared-memory system is usually thought of as one in which the processors have (more or less) equal access to all the memory.

However, shared memory can be emulated with physically distributed memory if we have a global address space.

In such cases it is usually possible to program a distributed-memory system using shared-memory programming constructs.

Variables can be shared (accessed by any thread) or *private* (accessed by only one thread).

Communication is done implicitly via shared variables.

This is in contrast to distributed-memory programming, in which communication is done explicitly via message passing, i.e., sending and receiving messages.

Shared-memory programming

Shared-memory systems typically have both static as well as dynamic thread creation; i.e., threads can be created at the beginning of a program by an OS directive or they can be created during the execution.

With dynamic threads, the master thread forks worker threads as required; the worker thread carries out the task and joins the master thread upon completion.

This is efficient because only the resources required by a thread are used.

With static threads, all threads are forked at the beginning and run to completion, rejoining the master thread upon completion.

This may be less efficient because the resources of idle threads cannot be freed.

However, thread creation/destruction is expensive, so static threads can perform well, and it is closer to the mindset of distributed-memory programming.

Shared-memory programming

Co-ordination among processes in shared-memory programs is typically managed by three constructs:

- a way to specify that data can be accessed by all the processes
- a way to prevent processes from improperly accessing shared data
- a way to synchronize processes

We now illustrate these concepts by means of the following simple examples.

Shared-memory programming

Because the processors in MIMD systems execute autonomously, it is easy to experience non-determinism; i.e., the same program with the same inputs leads to different outputs.

This happens because the relative rates of how threads are executed differ from run to run.

Suppose we have two threads (with ranks 0 and 1), each storing a variable `my_x` with values 7 and 19, respectively, and execute the code

```
printf("Thread %d > my_val = %d \n", my_rank, my_x)'
```

Shared-memory programming

One time you might see

```
Thread 0 > my_val = 7
Thread 1 > my_val = 19
```

but another time it might be

```
Thread 1 > my_val = 19
Thread 0 > my_val = 7
```

or the two outputs may even be interspersed!

Non-determinism is often harmless, but not always!

Shared-memory programming

Suppose again that we have two threads (with ranks 0 and 1) and each thread has a private integer `iPrivate`.

We want to compute the sum of the private integers and print it.

One way to do this is to declare a shared variable `SUM`:

```
INTEGER :: SUM = 0
```

Now we can add up all the local values of `iPrivate`:

```
SUM = SUM + iPrivate
```

What's wrong with this idea?

Shared-memory programming

In reality, an add consists of something like the following sequence of machine instructions:

Fetch `SUM` into register A, `iPrivate` into register B.
Add contents of register A to contents of register B.
Copy contents of register A into `SUM`.

Suppose `iPrivate`$= 7$ on thread 0 and `iPrivate`$= 19$ on thread 1.

Then `SUM` could come out to 7, 19, or 26 depending on the way in which the threads executed.

This is non-determinism, but it is not harmless!

The cause of the error is the attempt to simultaneously write to the same variable.

This is called a race condition: the output depends on which process/thread wins the race.

For the code to work, we must ensure only one `iPrivate` is added to `SUM` before it is re-stored.

Shared-memory programming

This concept that only one process should execute a certain sequence of operations at one time is called mutual exclusion, and the sequence of statements is called a critical section.

The most commonly used mechanism for imposing mutual exclusion is called a mutual exclusion lock or mutex or simply *lock*.

Locks are supported in hardware.

Basically, each critical section is protected by a lock.

Before a thread can execute the code in a critical section, it must obtain the lock.

The thread is then said to own the lock.

When it is done, it unlocks the section.

Shared-memory programming

Modified code to implement this might look like:

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

Note that no order is imposed on the threads!

Also note, locks enforce serialization on the code.

Thus, to maximize parallel efficiency, critical sections should be made as few and as short as possible.

Shared-memory programming

Now to print the correct (final) value of SUM.

For this we use a barrier function.

Once a process/thread has called the barrier function, it will not proceed until all the other processes/threads have called it.

So if we put a barrier after the addition statement, a certain process/thread (usually 0) can then print the value of SUM with the confidence that it reflects the contributions of all the processes/threads.

```
CRITICAL
SUM = SUM + iPrivate
END CRITICAL
BARRIER
IF (RANK.EQ.0) PRINT*, "I AM PROCESS", PROC, "AND SUM=", SUM
```

This is an example of synchronization.

Synchronization is expensive because at some point all processes/threads are waiting for the last one to arrive.

Therefore its use should be limited!

Shared-memory programming

Almost always, parallel programs can call functions designed for serial execution without any problems.

The most important example in C is when functions use static local variables.

In ordinary C, local variables are private.

But a static variable declared in a function persists from one call to the next.

Thus static local variables are effectively shared variables and need to be treated as such!

A function with static local variables are said to **not** be *thread safe*.

We will return to this concept in more detail later.

Distributed-memory programming

The most common method of parallel programming (in particular for distributed-memory MIMD systems) is through the use of *message-passing libraries*.

These libraries manage data transfer between instances of a parallel program that is (usually) running on multiple processors.

The central construct is that one process packs up information into a message and sends it to another, which then receives and unpacks it.

Processes co-ordinate their activities by explicitly sending and receiving messages.

It should be noted that this paradigm works just fine on shared-memory hardware.

However, we now use the terminology of processes instead of threads.

There may in fact be no mechanism for forking worker threads on any node.

Message passing

A message-passing library must contain (at least) a send and a receive function.

Processes are identified by their rank.

For example, process 1 could send a message to process 0 as follows:

```
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);
}
```

Message passing

Note the arguments to `Send()` are: the message, the type of elements in the message, the number of elements in the message, and the rank of the destination process.

The arguments to `Receive()` are: the variable into which the message is to be received, the type of elements in the message, the number of elements assigned to store the message, and the rank of the sending process.

Process 0 also prints the message after it is received.

Message passing

Notes:

- This is SPMD: both processes execute the same program but do different things, in this case, depending on their ranks.

- The variable `message` refers to different blocks of memory on different processors. A better naming of the receiving variable might be `my_message` or `local_message`.

- We assume processor 0 can write to `stdout`. Normally, all processes have access to both `stdout` and `stderr`.

Message passing

Although this seems simple and intuitive, the details can become much more complicated. In particular,

- How are messages actually sent? (How are they buffered within the system?)
- Can a process do other useful work while sending or receiving messages?
- How can sends and receives be paired to ensure the proper transfer of information?

Blocking and non-blocking communication

There are usually various flavours of the `Send` and `Receive` functions.

The simplest for `Send` is the so-called blocking mode, in which the sending process does not return from the `Send` call until it finds out that the matching call to `Receive` (on the other processor) has started.

Alternatively, in the non-blocking mode, the sending processor may copy the contents of the message to one of its buffers returns from the `Send` call as soon as the copy has been completed.

The advantage is that the process can continue to do useful work instead of waiting for the return message.

The default behaviour for `Receive` is for the receiving process to block until the message has been completely received, i.e., stored somewhere in its memory.

Collective communication

There are also a number of additional functions to perform common tasks.

For example, one process can broadcast or the same data (or *scatter* certain data) to all processes.

(One might contemplate doing this from scratch via `Send` and `Receive` calls, but the result is more work and generally inferior to using built-in functions.)

Another common task is to reduce (or gather) data from all processes and combine them into one result on one process.

One-sided communication

So far, we have assumed inter-process communication requires the participation of two processes.

In one-sided communication, a single process either updates its local memory with a value from another process or it updates the (remote) memory on another process with a value from the local process.

For this reason, one-sided communication is also called remote memory access.

Clearly, communication is simplified and it can significantly reduce the overhead of synchronizing two processes as well as eliminating half the function calls.

But there is the expense of processes needing to know when it is safe to transfer data from one's memory to the other; this can be done by either synchronizing the processes or by polling to see when it is safe.

Plus, there is a certain amount of risk that can introduce bugs that are very hard to detect.

Programming hybrid systems

A note on programming hybrid systems:

Recall that hybrid systems can be thought of several small shared-memory systems connected into a distributed-memory system.

It is possible to finely tune code to essentially program the cores on a processor to take advantage of the shared-memory paradigm while at the same time program the overall system to take advantage of the distributed-memory paradigm.

This can absolutely maximize performance!

However, such programming is generally not for the faint of heart.

So normally such systems are programmed as if they were purely distributed-memory systems.

Input and Output

Parallel input and output (I/O) are by no means trivial subjects, but our interests do not entail serious I/O; we generally assume in this course that our programs do need read and write significant amounts of data.

In terms of C, we can get by with the standard functions `printf`, `fprintf`, `scanf`, and `fscanf`.

Nonetheless, these are functions designed for a serial language, so the behaviour when they are called by different processes is unpredictable.

For example, we might like the output from `printf` to appear on the console of a single system (the one on which we started the program).

This is what most systems do, but some may not!

It is possible for some specific process (or no process!) to have access to `stdout` or `stderr`.

Input and Output

What should happen with parallel calls to `scanf`?

The vast majority of systems only allow one process (usually process 0) to call `scanf`; some allow more; some none.

Threads that are forked by a single process do share `stdin`, `stdout`, and `stderr`.

In such cases, I/O is generally non-deterministic.

To address I/O issues, we assume henceforth that

- Only process/thread 0 can access `stdin`.
- All processes/threads can access `stdout`, `stderr`, but (except for debugging) only process/thread 0 will output.
- Only a single process/thread can access a given file.
- Debug output gives the generating process/thread.

Performance

The fundamental point of parallel programming is to increase performance; so we need ways to measure it.

We begin with some comments on serial performance.

Ideally, we would like to have a function $T(n)$ that gives the execution time of a program in terms of the input (or problem size) n.

In theory, $T(n)$ can be constructed by counting statements and summing their individual costs.

Serial Program Performance

Of course, $T(n)$ will depend (perhaps strongly) on

1. the hardware being used
2. the programming language and compiler
3. other details of the input besides size

Because these details are so specific, we usually use asymptotic estimates of $T(n)$; e.g., $T(n) = \mathcal{O}(n^3)$.

Strictly speaking, this means $T(n) \sim Cn^3$ for n sufficiently large.

However, perhaps a more useful interpretation is that the program run with input size $2n$ will take roughly $2^3 = 8$ times as long as the program run with input size n, provided n is sufficiently large.

Serial Program Performance

For serial programs in scientific computing, the standard metric is floating-point operations.

In a serial world, minimizing floating-point operations essentially implies minimizing execution time.

(The significant exception we are ignoring relates to programs that are better able to take advantage of a given computer's memory hierarchy when accessing their data. In such cases, better memory access patterns can trump fewer floating-point operations.)

In a parallel world, the goal remains to minimize execution time, but the correspondence with minimizing operation count no longer holds.

Here, the mitigating factor (in some sense analogous to what happens in a serial world but much more common) is the cost of communication: better communication patterns easily trump fewer floating-point operations.

Parallel Program Performance

An obvious difference between serial and parallel performance estimation is that the run time of a parallel program depends on the number of processes p as well as the input size n.

So we will now denote program execution time by $T(n, p)$, where p is the number of processes, with $T(n, 1)$ representing the parallel program run in serial.

One of the obvious issues that arises when assessing parallel program performance is how does the parallel program compare to the serial one.

The two most common measures of comparison are speedup and efficiency.

Parallel Program Performance

The speedup $S(n,p)$ of a parallel program is defined to be

$$S(n,p) = \frac{T(n,1)}{T(n,p)};$$

i.e., it is the ratio of the serial and parallel run times.

There is a modicum of ambiguity in this definition.

Some authors take $T(n,1)$ to be of the fastest serial program known; some take it to be the parallel program run with one process.

(We will use the second approach in this course.)

There is even room to argue whether the codes are run on the same machine!

What one uses for $T(n,1)$ is context-dependent.

For example, it may make more sense to compare with the "fastest" serial program if the serial algorithm is different than the parallel one.

Parallel Program Performance

For fixed p, we normally have

$$0 < S(n, p) \leq p.$$

Ideally, $S = p$, and we have linear speedup.

This is the best we can hope for: equally dividing the work among the processes/threads while introducing no additional work.

In practice of course this is not realistic because there will be time lost to, e.g., critical sections (shared memory) and communication (distributed memory).

Sadly, a common occurrence is slowdown; i.e., $S < 1$.

In distributed-memory programs, this is generally the result of excessive communication among the processors; a different communication pattern (perhaps involving a different underlying solution algorithm) may be the only remedy.

Parallel Program Performance

An alternative to the speedup metric is efficiency.

The efficiency $E(n, p)$ of a parallel program is defined to be

$$E(n, p) = \frac{S(n, p)}{p};$$

i.e., it is the relative process utilization as compared to the serial program.

Because $0 < S(n, p) \leq p$, we have

$$0 < E(n, p) \leq 1.$$

$E(n, p) = 1$ corresponds to linear speedup, whereas $E(n, p) < 1/p$ corresponds to slowdown.

Parallel Program Performance

E generally decreases as p increases and increases as n increases.

This is because normally

$$T(n,p) = \frac{T(n,1)}{p} + T_O,$$

where T_O is the time required for overhead.

T_O generally increases much more slowly as n increases than $T(n,1)$ does.

Amdahl's law

Consider a single instance of a program running in serial; i.e., a certain program with a given input running with one process/thread on one processor.

Suppose the time it takes for the program to complete is $T(n, 1)$.

Suppose further that some fraction r of its statements $(0 \leq r \leq 1)$ are perfectly parallelizable; i.e., this part of the code has linear speedup, independent of p.

Hence the run time for this part of the code with p processes/threads is $rT(n, 1)/p$.

If the remaining $(1 - r)$ part of the code is inherently serial, then its run time will be $(1-r)T(n)$, independent of p.

Amdahl's law

With these assumptions, the speedup of the parallelized program running with p processes is

$$\begin{aligned} S(N,p) &= \frac{T(N,1)}{(1-r)T(N,1) + \frac{rT(N,1)}{p}} \\ &= \frac{1}{(1-r) + \frac{r}{p}}. \end{aligned}$$

Now as $p \to \infty$,

$$S(N,p) \to \frac{1}{1-r}^{-}.$$

That is, $S(N,p)$ is bounded above by $\frac{1}{1-r}$ and it approaches this limit from below as $p \to \infty$.

This observation was first made by Gene Amdahl in the 1960s.

Amdahl's law

This is generally bad news.

For example, if $r = 0.5$, $S(n, p) \leq 2$.

But even if $r = 0.99$, $S(n, p) \leq 100$, independent of the number of processes p!

If $p = 100,000$, $E(n, p) = 0.01$.

This paints a bleak picture because it implies there is no hope for massive parallelism.

Indeed from 1967 to 1988, Amdahl's law dampened the enthusiasm for parallel computing generally.

That was until 1988 when researchers at Sandia National Labs reported speedups of over 1000 on an nCUBE computer with 1024 processors.

This led to a re-examination of what Amdahl's law really says.

Amdahl's law

Given the assumptions, there is nothing wrong with the analysis.

And for a given instance of a problem and program for solving it, it is reasonable to assume some fraction of the statements in that program cannot be parallelized.

It is unrealistic to assume that part of the program is perfectly parallelizable, but one can imagine this is probably not a bad approximation in some cases.

What may not be reasonable is the assumption of using an infinite number of processes to solve a problem of a given size.

If we have a problem that required only say 500 calculations, then clearly it would be wasteful to use more than 500 processors to solve it!

Amdahl's law

The key idea is to use more processes to solve larger instances of the problem.

In this way, the serial fraction of the program becomes less significant, allowing us to make more optimistic predictions than those afforded by Amdahl's law.

An inherently serial program segment can be parallelized in essentially two ways:

1. One processor executes the serial statements while the other processes remain idle.

2. All processors execute the serial statements.

Either way, these solutions are sources of parallel overhead, which can be defined as the amount of work by the parallel program that is not done by the serial program.

Sources of overhead

There are 3 main sources of overhead:

1. communication,

2. idle time,

3. extra computation.

The ordering given is in general order of impact as well; i.e., communication time tends to dominate idle time as a source of overhead, which in turn dominates overhead due to extra computation.

In other words, this is the roughly order in which you need to worry about introducing inefficiencies into your parallel program.

In practice, only the effect of overhead on increased computation time (or decreased efficiency) is relevant; i.e., the fact that more computation is done by a parallel program matters less if it is faster than serial.

Sources of overhead

Clearly, serial programs do not communicate, so any communication costs directly make a program take longer to complete.

Idle time may or may not make a program take longer, e.g., a parallel program could be run as if it were serial, but efficiency generally decreases with increased idling.

Extra computation again may or may not make a program take longer to complete, e.g., an algorithm that lends itself well to parallelization may do more work (e.g., floating-point operations) than another algorithm that does not, but it may nonetheless complete in less time.

So, the main point is that all overhead is not created equal, and it is relatively easy to give too much importance to certain kinds of overhead and be distracted from the true goal of minimizing run time.

Scalability

Amdahl's law gives us a definite upper bound on the attainable speedup of a parallel program.

If a fraction r of a parallel program is perfectly parallelizable, then

$$S(n,p) \leq \frac{1}{1-r}.$$

In terms of efficiency,

$$E(n,p) \leq \frac{1}{p(1-r)+r}.$$

So, the equivalent statement of Amdahl's law in terms of efficiency is that the efficiency of a parallel program on a given instance of a problem approaches 0 as the number of processes goes to infinity.

Neither of these is surprising after some thought.

Scalability

However, another way to express efficiency is

$$\begin{aligned} E(n,p) &= \frac{T(n,1)}{pT(n,p)} \\ &= \frac{T(n,1)}{p\,T_O(n,p) + T(n,1)} \\ &= \frac{1}{\frac{p\,T_O(n,p)}{T(n,1)} + 1}, \end{aligned}$$

where $T_O(p)$ is the extra compute time incurred by the overhead of parallelization.

So, the behaviour of $E(n,p)$ as $p \to \infty$ is completely determined by $T_O(n,p)$!

If we fix p and let $n \to \infty$, $E(n,p) \to 1$.

Again, another unsurprising result after some thought.

Scalability

So, in some sense, the real insight comes in the situation when n and p are increased simultaneously.

A parallel program is said to be scalable if there exist rates of increases of problem size n and processes p such that $E(n, p)$ is constant.

Note this does not say n must increase like p; so it allows the possibility of different degrees of scalability.

If $E(n, p)$ remains fixed for increasing p (but not n), then we say the program is strongly scalable.

If $E(n, p)$ remains fixed when p and n increase at the same rate, then we say the program is weakly scalable.

Scalability

As an example, suppose we run a program on a problem of size n using p processes/threads.

Let $T(n, 1) = n$ and $T(n, p) = n/p + 1$.

Then

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

To test for scalability, we let $p \to \lambda p$ and $n \to \mu n$ and solve for μ under the assumption of constant efficiency; i.e.,

$$\frac{\mu n}{\mu n + \lambda p} = \frac{n}{n + p}.$$

Solving for μ leads to

$$\mu = \lambda.$$

That is, our example program is weakly scalable.

Timing is everything

We have discussed the importance of timing programs to test their performance and scalability.

However, during development we may also want to time programs to see if their behaviour is as expected, e.g., how much time is spent waiting for messages, etc.

The second kind of timing is known as profiling.

This will be discussed further when we develop our own programs.

In this case, we are interested in detailed measurements of how long a specific part of a program takes to run; so the Unix shell command `time`, which measures the time taken to run an entire program, is not useful.

Timing is everything

To assess performance of parallel programs we are also generally *not* interested in "CPU time", the time spent executing code, as measured, e.g., by the standard C function `clock`.

This is mainly because idle time, e.g., waiting for communication, is a crucial aspect of assessing performance of parallel programs but is not included in CPU time.

What we are interested in reporting is normally "wall clock" time, i.e., the amount of (real) time that elapsed while the code was running.

Source code that would enable such a timing might be

```
double start, finish;
...
start = Get_current_time();
/* Code to be timed */
...
finish = Get_current_time();
printf("Time elapsed = %e s.\n", finish-start);
```

Timing is everything

A subtle issue that cannot be safely ignored is the resolution of the timer function, i.e., the shortest event that will give a non-zero measurement.

This information is usually given with the timer function used; e.g., the resolution may be 1 ms.

It is important for the programmer to know the resolution of any timer functions used to account for any difference in scale between it and the things that are to be measured.

For example, if instructions are executed on the order of ns but the resolution is ms, only groups on the order of a million instructions will be measurable.

Or conversely, measurements of on the order of less than a million instructions will give 0.

Timing is everything

When timing parallel codes we are usually interested in the elapsed time from when the first process/thread executed its first instruction to the time when the last processor/thread executed its last instruction.

This is usually not measurable because there is no correspondence between processor clocks.

In practice, we settle for a compromise like this:

```
shared double global_elapsed;
private double local_start, local_finish, local_elapsed;
...
/* synchronize
Barrier();
local_start = Get_current_time();
/* Code to be timed */
...
local_finish = Get_current_time();
local_elapsed = local_finish - local_start;

/* Find max elapsed time over all processes/threads */
global_elapsed = Global_max(local_elapsed);
if (local_rank == 0)
   printf("Time elapsed = %e s.\n", global_elapsed);
```

Timing is everything

We first execute a barrier function to (approximately) synchronize all the processes/threads.

We would like for all the processes to return from the call to the barrier function simultaneously; i.e., they all start executing the code to be timed at the same time.

Normally, however, barriers can only guarantee that all the processes/threads have entered the call when the first process/thread is allowed to exit.

Each process/thread then measures its own (local) execution time.

The maximum of these times is then determined, and process 0 prints out the result.

This is the value taken to represent the execution time of the entire program.

Timing is everything

Finally, we must also be aware of variability in timings.

Even in serial, executing a program with a given input on the same system does not produce identical timings.

It might seem natural to therefore report a mean or median run time.

This may be a good measure of what a user might experience; but we rarely watch our programs run.

Outside events are not likely to make our programs run faster; so we so we report the minimum run time.

This is closer to what is theoretically possible and more representative of where our interests lie.

Executing more than one thread per core can increase the timing variability due to the overhead of scheduling cores; so we assume executing only one thread per core.

Our programs are not designed for high-performance I/O; so I/O will not be included in timings.

Parallel Program Design

Often the single most important decision in parallel programming is knowing when to parallelize.

Not every program needs to be parallelized in order to provide an acceptable solution.

There are only two scenarios in which parallelization makes sense as a way to help solve a problem:

1. The problem is too big to fit into the memory of one processor.

2. The problem takes too long to run on one processor.

The goal in both of these scenarios can be described as reducing the amount of (real) time it takes to get the solution.

Both reasons often apply!

That is, programs that require a lot of memory take a long time to run.

"Long time" may be a subjective term, but often there are real constraints that determine what is an acceptable time for a solution to be returned[1].

Memory requirements are more objective. Assuming good use is being made of memory allocations (and no leaks, etc.), when a program's memory requirements exceed the amount available, it will either produce an "out of memory" error or effectively grind to a halt as memory is swapped.

[1] Many times this is "as soon as possible".

Parallel Program Design

The starting point for designing a parallel program is often a serial program.

At present, there is no completely automated and universal process for parallelizing a serial program.

However, Ian Foster outlines 4 steps to do this in his on-line book Designing and Building Parallel Programs.

1. Partitioning. Logically divide computation and data into smaller units.

2. Communication. Determine communication needs.

3. Aggregation. Re-combine tasks, data, and communication (if advantageous).

4. Mapping. Assign tasks to processes/threads minimizing communication and balancing loads.

This is sometimes called Foster's methodology.

Parallel Program Design

Before looking at an example, we make a few comments on writing and running parallel programs.

The old-fashioned way of developing programs was to use a text editor like `vi` or `emacs`; the program was compiled and run (and debugged) from command line.

Today there are integrated development environments (IDEs) like `Eclipse`.

Our focus is on homogeneous MIMD systems, i.e., ones for which all the nodes have the same architecture[2].

Programs run as SPMD on identical cores with at most one process per core.

We generally use static processes/threads; i.e., processes/threads will be created/started at more or less the same time, run to completion, and terminated more or less at the same time.

[2] even though strictly speaking `socrates` is not!

Parallel Program Design

Some APIs for parallel programming define new programming languages.

Most extend existing languages, either through function libraries (e.g., functions to send messages) or with compiler extensions for the serial language.

We focus on this second approach via parallel extensions to C and at times Fortran and MATLAB.

We mainly use the `gcc` compiler or some extension (e.g., `mpicc`).

`socrates` also has the Intel compilers `icc` and `ifort`.

Parallel Program Design

Compiling and running the famous "hello, world" program can be accomplished using

```
gcc -g -Wall -o hello hello.c
./hello
```

with output

```
hello, world
```

The compiler options used are

- `-g`. Create information for use with debugger.
- `-Wall`. Print all warnings.
- `-o <outfile>`. Name the executable `outfile`.

When timing programs, we usually optimize the code using the `-O2` option.

Summary

- Parallel programming paradigms
- Domain and functional decomposition (data and task parallelism)
- Shared-memory programming; client-server paradigm
- Message passing
- Input and output
- Parallel program design