Chapter 27
Parallel Algorithms

To analyze algorithms, we once laid out a basic assumption of a sequential execution model (Cf. Page 20 of the *Analysis of Simple Algorithm* chapter), i.e., algorithms run on a uniprocessor computer, where one instruction executes at one time, commonly known as the *von Neumann architecture.*

We now move over to a model for *parallel algorithms,* which can run on a multiprocessor computer that allows multiple instructions to execute concurrently.

In particular, we will discuss the *dynamic multithreaded algorithm* model, go through examples of parallel programming, introduce efficiency measurements, and carry out some simple algorithm analysis. We will end up with a parallel version of the merge sort algorithm.

Why parallel computers?

As we all know, after the creation of computers back in the late 1940's, there has been a great progress in terms of the computer speed, indeed a 20 million fold increase during a fifty year period.

This is done, mainly due to the fact that more and more transistors have been integrated into a silicon chip, from a few to tens (SSI), hundreds (MSI), thousands (LSI), and the billions (VLSI).

What has happened and what's next?

Moore's law

This dramatic speed-up phenomenon is nicely summarized via the Moore's law (1965): *The number of transistors that have been put onto a chip has been doubling every eighteen months.*

For example, Intel 8086, a processor chip made by Intel in 1978, contained 29,000 transistors, and ran at 5 MHz; and the Quad-core+GPU Core i7 Haswell, introduced by Intel in 2014, contained 1.4 billion transistors, running at the speed of up to 4.4 GHz.

Thus, during those 36 years, the number of transistors has gone up by 482,758 times, or doubled once every 22.9 months.

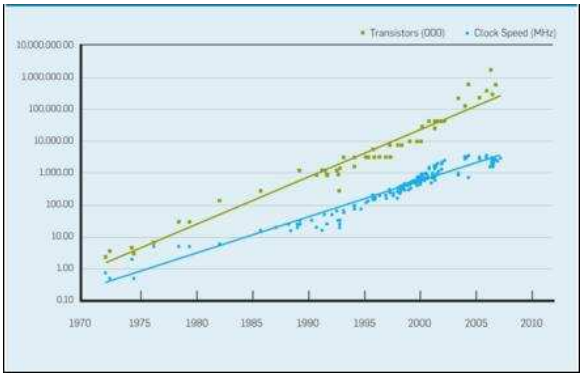
$$(36\times 12)/n = \log_2 482,758 \;\Rightarrow\; n \approx 22.9.$$

The most recent one, 2018, is i9 with up to 18 cores, running 36 threads.

Worth how many words?

Such an increase of the transistors on a chip directly leads to an increase of the computer speed.

The following chart shows the increase of the computer speed corresponding to that of the integration number.



In this case of going from 8086 to i7, the speed goes up by 880 times during this period, doubled every 44 months.

Not just the speed...

Besides processing speed, some of the other capabilities of many digital devices are also strongly connected to Moore's law: memory capacity, sensor usage, and even the number and size of pixels in digital cameras.

As a result, all of these technological developments have also been speeding up at this stunning, exponential, rates, as well.

Since Moore's law approximately describes a driving force of technological, and social, change in the past thirty or so years, it has been used to guide long term planning, and to set targets for research and development.

Check out the reading "The Exponential Nature of Moore's Law" on the course page.

How thin could it be?

Unfortunately, this era of steady and rapid growth of single-processor performance over 30 years could not last long.

By "doubling every eighteen months," , we have to make the wires $\sqrt{2}$ thinner every eighteen months. (Cf. Page 25 of Chapter 0) This has to come to an end at some point since we can't make the wires infinitely thin.

Incidentally, *Dennard's scaling law* refers to the property that the reduction of transistor size comes with a reduction of required power.

When the transistor size has reduced from 65 nanometer to about 10 nanometer (i7 is at 22, and i9 at 14), the ability to speed up processors for a constant power cost has stopped.

By the way, a sheet of paper is about 100,000 nanometers thick.

How hot could it be?

Although every transistor produces only a tiny bit of heat, when you put billions of them to a tiny space, the amount do add up. For this reason, the clock speed of a processor has been staying essentially the same, at about 4GHz, for the last ten years.

Although Moorel's law still lets us put more CPU cores on a chip, and let GPU continue to strengthen its power, it seems to be the case that transistor size cannot reduce much further, perhaps two or three more generations before seeing its end. ☹

Check out the link on the course page "Moore's law is on its way out..." on the course page.

What to do?

Fortunately, Moore's law is not completely out of the window yet.

This many transistors will no longer be used to construct a single processor, but will be used to increase the number of independent processors (cores) in a single chip.

We will then try to speed up the whole process of letting those independent processors work together on the data *in parallel*, thus saving time.

In the ancient time, we can only cook one thing at a time with our old fashioned stove.

Could we do this?

Nowadays, using a contemporary stove with multiple burners, we *could* cook many different dishes in parallel, or at the same time, which should save time.

By the same token, we could cut a big problem into many smaller ones (Divide and Conquer), and run them in parallel with multiple processors.

To speed up problem solving, we certainly should do it.

The challenge is how to coordinate those processors so that they can work together, often with balanced load, in parallel.

Parallel computers

Parallel computers, those with multiple processing units, have become increasingly common, e.g., *this* desktop has four cores, and iPhone X has six.

Such an architecture contains a single *multi-core IC chip* that contains multiple processing cores, each a fully functional processor that can access a common, shared, memory.

Going a bit higher in the hierarchy, there could be a cluster built from individual computers, with a dedicated interconnection network.

At the very top, there are the so-called *supercomputers*, consisting of a combination of custom architectures, connected with an interconnection network, providing top performance in terms of MIPS. Examples include nCube, NASA Pleiades, and IBM Blue Gene.

Memory architectures

As we know, from *CS 2010 Computing Fundamentals* and/or *CS 2220 Computer Hardware,* the uniprocessor model is naturally connected with the RAM model in the von Neumann architecture. But, parallel computers have yet to agree on its memory architecture.

Some of them feature the *shared memory model,* where each processor can directly access any location of the memory. Other machines use *distributed memory,* where each processor possesses its own memory, and any message must be transmitted in between the processors in order for one processor to get data from another one, through an interconnection network.

With the emerging multi-core architecture, the trend seems to be in favor of the shared memory model. We will follow this model in our discussion.

Threads

I am sure that you are familiar with the ideas of *threads.* Essentially, when programming with a multi-core architecture, we split the program into a bunch of *static programming,* which provides an abstraction of "virtual processors".

Threads, i.e., all these independent, thus individually executable, units, share the same memory, each having a program counter, and a set of associated registers.

Each thread can be assigned to a processing core to run, and when either it is done, or its share is used, it is taken off the processor, and another one will be assigned to the processor.

We will talk a lot about threads, its scheduling and execution, in *CS 4310 Operating Systems.*

It is challenging....

They all sound good, but it is really challenging to program a shared-memory computer, using static threads.

In the cooking example, a good chef knows that she will not always be able to cook everything at the same time, no matter how many burners are at her disposal.

To cook the dish of, e.g., Pepper, Onions and Pork, I would fry the pepper, and the pork, first, which can be done at the same time; *then* fry the onion, which is then mixed with the partially fried pepper and the pork.

In the multiple lane case (Check out the toll-booth picture!), although the cars in different lanes can go forward in parallel, those in the same lane have to go forward in turn. ☹

MergeSort again....

It is the same idea to do computing in parallel. You have to figure out what parts can be done in parallel, and what can't be.

Take Mergesort as an example.

```
MERGE-SORT(A, p, r)
1. if (p<r)
2. then q<-(p+r)/2
3.      MERGE-SORT(A, p, q)
4.      MERGE-SORT(A, q+1, r)
5.      MERGE(A, p, q, r)
```

You cut a list into two halves, and sort them individually *in parallel* in Steps 3 and 4.

But, you have to wait for these steps to complete first, then, in Step 5, merge the two sorted sublists into one, which also has to be done sequentially.

Registration for courses

Have you signed up your Fall courses yet?

When adding somebody into a class, a program has to make sure, among other things, that the total number of students added into a class is no more than the cap of that class, 25 for *CS 3600 Database*.

If we run Course Add sequentially, we essentially do the following:

```
if the current number < 25
then add this student
```

Thus, we check the cap *before* adding in another student.

Is this sequential order necessary?

The parallel case

Since the above add operation consists of two steps: one check and another add, when we try to add multiple requests at the same time, we might get into trouble since we don't know in what order will the steps get mixed up: *No body knows which process, or thread, the OS scheduler will pick up first.*

For example, if there are 24 students signed up for this DB course, and two more students come to add into the course.

What is to happen, if we choose to do these two steps in parallel?

It might not work ☹

If we do the Course Add in parallel, and it happens that the arrangement of the two steps for the two add operations interleave the the following way, in one out of six $\left(= \frac{4!}{2!2!}\right)$ ways (?), i.e., with a probability of 16.7%, we would end up with the following situation.

Request 1	time	Request 2
-	$\downarrow$	-
Check the number	t_1	-
(Still 24)	$\downarrow$	-
-	t_2	Check the number
-	$\downarrow$	(Still 24)
Add in student	t_3	-
(Now 25)	$\downarrow$	-
-	t_4	Add in student
-	$\downarrow$	(Now 26)

Thus, as the above chart shows, we will add more students than what the cap allows.

What's wrong with it?

A parallel platform is intended to speed up the process, but it may not lead to a correct result. ☹

Indeed, *although we can come up with lots of cheap hardware parts, the challenge lies on the software part, especially on the communication and coordination of the participating threads.*

Only a small percentage of programmers can do it now. ☹ ☺

We will address lots of such issues in *CS4310 Operating Systems*.

If there were not parallel issues, we could finish that course in one week. ☺

Dynamic multi-threaded programming

Dynamic multi-threading is one important class of concurrency platforms. This model allows programmers to specify parallelism in applications without concerning about communication protocols, load balancing and other issues of multi-threaded programming, which will all be taken care of by a scheduler.

DM environment is still evolving, but two features are guaranteed: *nested parallelism* and *parallel loops.*

Nested parallelism allows a caller to spawn a subroutine, which is to proceed along with the caller (`fork()` in unix, in Lab 19 if you are taking CS 2470). And Parallel loop allows all the stuff within the loop body to execute concurrently (Could we really? Check out the example on Page 50).

Why are they important?

These two features form the basis of this model, where the programmer only needs to specify the *logic parallelism* (*what?*) within a computation, and a scheduler will taker care of the scheduling of these threads, balance the load among those threads and subtasks (*how!*).

It is quite similar to Window case. Who would (could) care about the 1,400 threads running inside the box, how and when they are picked and run. The scheduler does it all.

We will check out how to write parallel algorithms in terms of this model, and how the underlying concurrency platform will schedule computation efficiently.

Selling points

The dynamic multi-threading model is a simple extension of the sequential programming model with just three additional buzzwords: **parallel**, **spawn**, and **sync**. If we remove these three words from an algorithm, we get back a sequential algorithm for the same problem.

It also provides a clean theoretical way to quantify parallelism based on the notion of "work" and "span". (Efficiency issues)

The nested parallelism follows naturally from the traditional "divide-and-conquer" paradigm, and we can use recurrence to do the analysis.

This model is adopted by several influential parallel programming paradigms such as Chik, Chik++, OpenMP, TPL, TBB, and (Chapel (?))

Check out some of the links on the course page... .

Start with an example

Still remember the Fibonacci sequence? (1, 1, 2, 3, 5, 8, ...), which can be easily generated with the following recurrence:

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_i &= F_{i-1} + F_{i-2}, \text{ for } i \geq 2. \end{aligned}$$

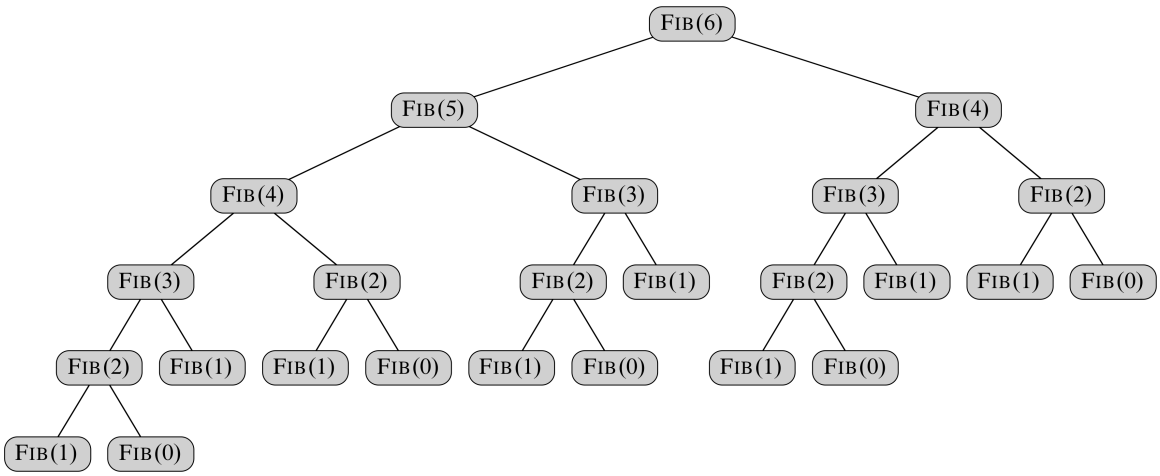
The following recursive algorithm of a sequential nature, $FIB(n)$, is thus natural.

1. if $n \leq 1$
2. return n
3. else $x = FIB(n-1)$
4. $y = FIB(n-2)$
5. return $x + y$

Did I tell you that recursion is not good to the computers? ☹

It is just a bad move....

Besides the extra cost of a hidden stack, a recursive procedure, in this case, does lots of redundant work, as the following chart shows:



It contains 25 computation steps to compute $FIB(6)$, where each box represents a computation step: a non-leaf node sums up the works done by its two children.

Question: How much does it do to compute $FIB(n)$?

Exponentially....

Let $T(n)$ stand for the time it takes to run $FIB(n)$, we have

$$T(n) = T(n-1) + T(n-2) + \Theta(1),$$

where $\Theta(1)$ represent the addition, condition testing, etc..

Assume that for all $k < n, T(k) \leq aF_k - b$, $a > 1, b > 0$, by inductive assumption,

$$\begin{aligned} T(n) &\leq (aF_{n-1} - b) + (aF_{n-2} - b) + \Theta(1) \\ &= a(F_{n-1} + F_{n-2}) - 2b + \Theta(1) \\ &= aF_n - b - (b - \Theta(1)) \\ &\leq aF_n - b. \end{aligned}$$

Hence, $T(n) = \Theta(F_n) = \Theta(\phi^n)$, where

$$\phi = \frac{1+\sqrt{5}}{2}.$$

We once gave a weaker one: $T(n) < \left(\frac{5}{3}\right)^n$. (Cf. Page 30 of the Math review notes)

What to do?

One way to overcome this inefficiency is to follow a *dynamic programming* approach: Let `F[0]=0`, `F[1]=1`, and for all $i \geq 2$, `F[i]=NIL`.

$FIB(n)$
1. if `F[`n`]`
2. return `F[`n`]`
3. else $x = FIB(n-1)$
4. $y = FIB(n-2)$
5. `F[`n`]` <- $x + y$

In other words, we *trade space with time.* To calculate `F(4)`, we need to get `F(3)` and `F(2)`. To get `F(3)`, we need to get `F(2)` and `F(1)`.

Finally, `F(2)` is the sum of `F[0]` and `F[1]`, which is used to fill `F[2]`.

When calculating `F(3)`, we just use `F[2]` and `F[1]`, *etc.*.

A parallel version

Another way is to turn the above to a parallel program, $PFIB(n)$, since these two recursive calls are independent of each other, thus they can be called in any order, even in parallel. *Now, we trade resource with time.*

1. if $n \leq 1$
2. return n
3. else $x = \mathbf{spawn}\ PFIB(n-1)$
4. $y = PFIB(n-2)$
5. **sync**
6. return $x + y$

As mentioned earlier, if we delete these buzzwords, we get back the original code.

Thus, the *serialization* of a multi-threaded algorithm is always the usual serial algorithm to crack the same problem. This makes part of the analysis straightforward as we will see.

What does **spawn** do?

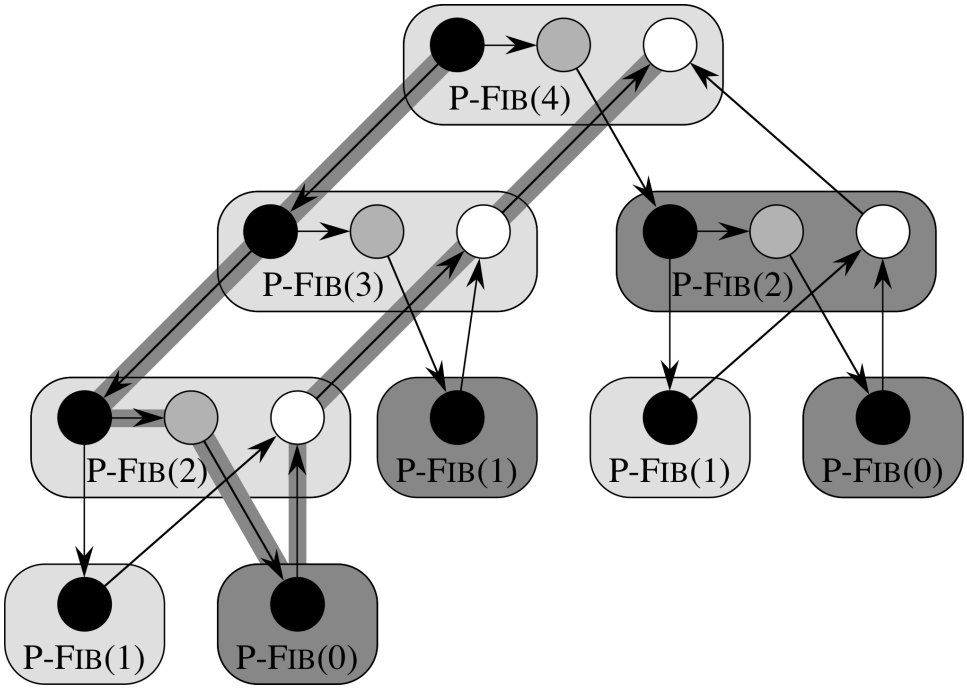
Nested parallelism occurs when the keyword **spawn** proceeds a procedure call.

It means that the *parent procedure instance* that executes that spawn *may* continue to execute in parallel with the spawned *child subroutine,* instead of waiting for the child to complete first, as what happens in a serial execution.

In this case, the parent process (thread) may continue with Line 4, while a child process (thread) will carry out Line 3. The parent process (?) has to wait for the child process to finish at Line 5, then returns the result with Line 6.

We will talk about the `fork()` mechanism in *CS 2470 System Programming,* which essentially does this spawn stuff.

How is $PFIB(4)$ played out?



Take the top node for example, to calculate $Fib(4)$, a process does three things: it spawns a black node (Line 3) to calculate $Fib(3)$, and at the same time, proceeds (Line 4), via a grey node, to find out $Fib(2)$, then wait for their completion (Line 5), and finally reports the result in a white node (Line 6).

It has to wait $Fib(3)$ to complete before it is done, thus the darkened bottleneck.

How fast could we do it?

It really depends... . Assume each node, a *strand,* takes one time unit, if you have just one processor, it will definitely takes 17 time units to complete.

In fact, no matter how many processors do you use, they all have get this much *work* done.

It also depends on the nature of the problem. For a problem of finishing a cake, the more persons you put in, the faster it gets finished.

Question: Could we get the above $Fib(4)$ calculation done in one unit if we have 17 processors?

Answer: No, no matter how many processors you throw in, it takes at least eight units of *span* to finish. (?)

We will further explore these measurement issue later on page 38.

Could we do this? *Maybe...*

In such a chart, a horizontal *continuation edge* (u, v') connects u, a black node, and v', a grey one, in the same function call (e.g., $PFIB(4)$), while a downward edge (u, v) can either connect u, a black node, with a **spawned** child v, also black (e.g., the one from $PFIB(4)$ to $PFIB(3)$); or indicate the usual subroutine call, a grey one to a black one, e.g., the one in the $PFIB(4)$ box, calling $PFIB(2)$.

If v is **spawned** from u, u also connects to another strand u' in the same procedure instance with a continuation edge, and u' and v *may* execute in parallel. For example, $PFIB(3)$ and $PFIB(2)$ *may* execute by two processors in parallel.

It is up to the scheduler to decide whether this "*may*" will materialize.

The case of $Fib(4)$

Question: How does the previous discussion about $Fib(4)$ fall into this general description?

Answer: In this case, while the spawned child is computing $PFIB(3)$, the parent may continue to compute $PFIB(2)$ in line 4 *in parallel* with the spawned child.

Furthermore, since the $PFIB$ procedure is recursive, these two subroutine calls create nested parallelism, as do their descendants, thus leading to a *computation tree,* with some nodes executing in parallel.

I feel dizzy....

Logic parallelism

Notice that the keyword **spawn** does *not* enforce the parent procedure to run in parallel with its spawned child. It only says "may".

Even though the other lane is there, you might still want to follow a car in the same lane.

Thus, this keyword merely represents a *logical parallelism* of the computation, indicating which part *may* run in parallel; but *may not* in real due to, e.g., a scheduling or a resource issue. ☺

We definitely want them to run in parallel, but there might not be a processor available at the moment. ☹

It is then up to a *scheduler* to decide, during the run time, which subprocesses will *actually* run in parallel by assigning them to separate processors. We will not be concerned about this mess until later... (Page 47 of the notes).

What about **sync**?

A **sync** statement guarantees that a procedure must wait for all its spawned children to complete their respective tasks before continuing to the part after **sync**, sort of a sequence enforcer.

For the $PFIB$ procedure, a **sync** is needed (Line 5) so that before x is added to y in Line 6, we have to make sure that x is returned by the spawned child, and the value in x is stable.

Besides such explicit **sync**, every procedure executes an implicit **sync** before it returns, thus making sure all its children are completed before it does.

It is actually a nice practice to add an explicit **sync** by the end as we will see.

I want to see....

As we just saw, we can demo the process of multi-threaded computation, a collection of instructions executed by processors on behalf of a multi-threaded program, in terms of a directed graph $G(V, E)$, called a *computation dag.* (We will talk about this *dag* stuff, standing for *directed acyclic graph,* when we get to the graph part (Page 34 of Chapter 22: *Basic Stuffs of Graph Algorithms*.)

The set V stands for a collection of instructions, and E gives the dependency between the instructions, in the sense that if $(u, v) \in E$, then u has to be executed *before* v can be.

For example, you have to take *MA 2250* and *CS 2370* before taking *CS 3600*.

A parallel program

Below is the $PFIB(n)$ procedure (Page. 26), a parallel program implementing the *Fibonacci* function.

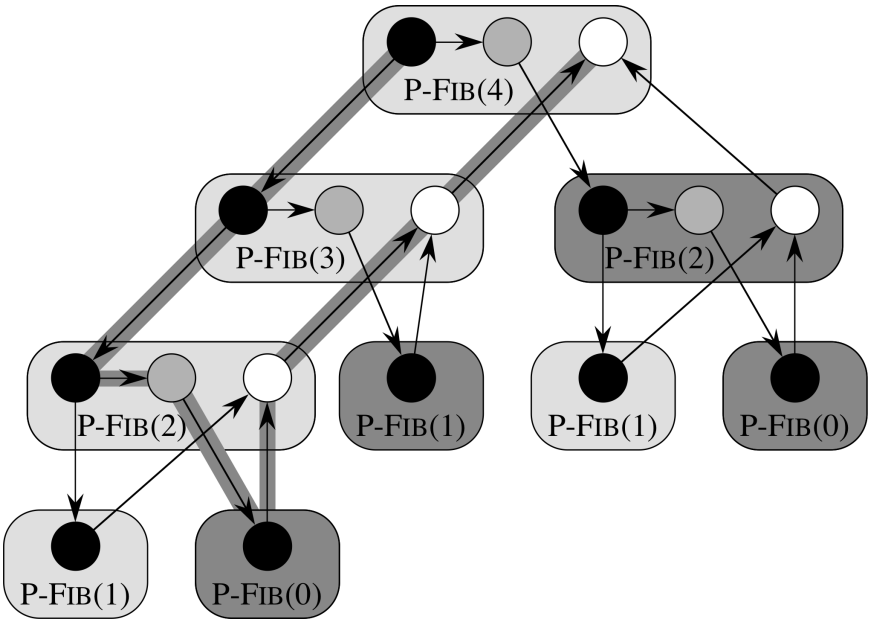
1. if $n \leq 1$
2. return n
3. else $x = \mathbf{spawn}\ PFIB(n-1)$
4. $y = PFIB(n-2)$
5. **sync**
6. return $x + y$

As mentioned earlier, if we delete these buzzwords such as **spawn**, and **sync**, we get back the original sequential code $FIB(n)$ (Page. 22).

Thus, the *serialization* of a multi-threaded algorithm is always the same usual serial algorithm to crack the same problem.

Look at $PFIB(4)$ again

If a segment of instructions contains none of the three keywords, we group them together into a *strand* (node), represented as a bubble, which is a basic computing unit. Then, the parallel relationship between strands can be represented by the following *dag* structure:



If a strand has two successors, one of them must be a **spawned** child process, and if one strandhas two predecessors, this strand must be a **sync**.

An ideal environment

We now make a few assumptions for a platform to run our parallel programs.

An ideal parallel computer consists of a bunch of processors and a *sequentially consistent shared memory.* Such a memory, although doing lots of loads and stores at the same time, produces the same result, as if all the involved instructions are done in a sequential way. (This is the 'I' piece of the *ACID* requirement that we went through in *CS 3600 Database*)

The deal is that, although the order of the instructions in each parallel execution might change, the above linear order as imposed by the computation structure has to stay the same to achieve a correct result. Again, *correctness precedes efficiency.*

Each processor in such an ideal machine has the same computing power and it ignores the above scheduling cost.

Measurements

When measuring the time efficiency of parallel programs, we use two metrics: *work* and *span.*

Work refers to *the total amount of work done by all the processors.* If we assume each strand costs a unit of time, once we draw a computation dag, the work done by the associated computation is simply the number of strands in that dag.

Span refers to *the longest time to execute all the strands along any path in such a dag,* namely, the number of nodes along the longest, or the *critical,* path in such a dag. Instructions along such a path can't be executed in parallel, even if additional processors are available.

In the previous dag for $PFIB(4)$, it is easy to see that the work is 17, and the span is 8.

Yet another factor

Work tells us we have this much to do, and span tells us that it takes at least this much time *no matter how many processors you throw in.*

The actual running time also depends on two other issues: how many processors can you use, and how the scheduler is to assign them to the strands. (*Logic parallelism vs practical reality*)

We use T_P to represent the running time of a multi-threaded computation with P processors.

Thus, the work is simply T_1, as if we have only one processor to finish all the strands.

On the other hand, span counts the time when each strand runs at its own processor, as if we have as many processors as it wants. Hence, span turns out to be T_∞.

Implication

Work and span actually set up lower bounds for T_P.

Sine one processor can complete one strand at one time unit, P processors, in one unit of time, can complete at most P strands. Thus PT_P in T_P time.

But, we have to get all the work, T_1, done, hence, $PT_P \geq T_1$. This leads to the *work law:*

$$T_P \geq T_1/P.$$

On the other hand, a computation with P processors cannot run faster than a one with unlimited processors, thus, the *span law:*

$$T_P \geq T_\infty.$$

Combined together, $T_P \geq \max\{T_1/P, T_\infty\}$

Speed up

The only reason we kick in more resource is to get a better result, e.g., to get it done faster.

Thus, we define the *speedup* of a computation on P processors by the ratio of T_1/T_P, i.e., *how many times faster is the computation on P processors than on one?*

By the work law, we have $T_1/T_P \leq P$. When this ratio hits linear, i.e., when $T_1/T_P = \Theta(P)$, this computation shows *linear speedup,* when we are pretty happy, and we could not be happier when $T_1/T_P = P$, hitting a *perfect linear speedup.*

For example, assume it takes one minute to go through a toll booth, if we have just one lane, it takes four cars T_1 $(= 4)$ minutes to go through. With P $(= 4)$ lanes, it would take at least T_4 $(= 1)$ minute. Thus, the speedup is at most $(T_1/T_4 = 4)$ in this case.

Parallelism

The notion of *parallelism* is defined as the ratio of work over span, i.e., T_1/T_∞, i.e., given a computation, the ratio of time with just one processor over that with unlimited resource.

It actually tells the average amount of work (T_1) that can be done in parallel for each step along the critical path (T_∞ is the length of such a path.).

Because of the span law, $T_P \geq T_\infty$, $T_1/T_P \leq T_1/T_\infty$. And by the work law, $T_1/T_P \leq P$, hence

$$T_1/T_P \leq \min\{P, T_1/T_\infty\}.$$

As a result, once the number of processors exceeds parallelism, we are simply throwing money away... . ☹

Why is that?

Let's assume $P > T_1/T_\infty$, then

$$\frac{T_1}{T_P} \leq \min\{P, T_1/T_\infty\} = \frac{T_1}{T_\infty} < P,$$

i.e., the speedup is strictly less than P. ☺

Moreover, if P is much larger than parallelism, then the speedup will be much less than P. ☹

For example, in the `P-FIB`(4) case, since work is 17 and span is 8, its associated parallelism is just $17/8 = 2\frac{1}{8}$.

Thus, it is impossible to achieve much more than doubling the speedup, no matter how many processors are thrown in. ☹

Homework: Exercises 27.1-2 and 27.1-5.

How fast could it be?

The natural expectation for the speedup from parallelization would be linear: If you put in a two lane highway, then two cars can do through the toll both at the same time, and if you put in a four lane, then four cars can pay tolls in parallel.

That is why we often put in multiple lanes in the highway (Check out the DoT construction project in I-93S near the state border.)

Unfortunately, this does not happen to the parallel computing: *Very few parallel algorithms achieve linear speed-up.* ☹

Most of them have a near-linear speed-up for small number of processing elements, but degrades to a constant value for large numbers of processing elements.

Here is the limit

The potential speedup of a parallel algorithm on a parallel computer is given by Gene Amdahl in 1960s.

When we cut a big problem to a bunch of smaller ones, some of them can run in parallel, while the others cannot, it is the latter that will decide overall speed-up available from parallelization.

Below is the *Amdahl's law*:

$$S = \frac{1}{s + \frac{1-s}{P}},$$

where S is the speedup of a parallel program, s the fraction that has to be run in sequence, and P is the number of processors.

Question: Where will we have linear speedup?

Answer: $s = 0$, when everything can be done in parallel.

An example

If we cut the problem into ten pieces, nine of them can run in parallel, while one piece can't, with 10 processors.

We have that $s = 10\%$, and $P = 10$, then, the Amdahl's law tells us that

$$S = \frac{1}{0.1 + 0.9/10} = \frac{1}{0.19} = 5.26.$$

In other words, when 90% of the work can be done in parallel, with ten processors, we can speed it up only 5.26 times. Thus, those four extra processors are definitely wasted. ☹

This result thus puts an *upper limit* on the usefulness of adding more parallel execution units.

One way to put it: "The bearing of a child takes nine months, no matter how many people are assigned."

Scheduling

As we saw earlier, performance of parallel programming depends not only on minimizing works and spans, but also on a scheduler that maps strands to processors.

With the multi-threaded model, we rely on a scheduler to map those dynamically generated strands to individual processors. Such a scheduler has to do its job *on line (in real time),* when those strands got created, or spawned off. Moreover, such a scheduler has to balance the load among a bunch of processors.

Provably good on-line, load balancing schedulers do exist, but it is hard to analyze them.

We will instead look at a centralized scheduler kept in a server, which knows a *global state* of the computation, i.e., status of all the processors, at any time.

A centralized scheduler

A *greedy* scheduler (ring a bell?) assigns as many strands to processors as possible at each time step. With a computer with P processors, a step is called a *complete step* if there are at least P strands available for assignment at that step, when a greedy scheduler will select and assign any P strands to processors.

Where fewer than P strands are ready for execution at a step, we call it an *incomplete step,* where a greedy scheduler will assign each *ready* strand to its own processor.

By the Work law and the Span law, we may conclude (Cf. Page 40) that

$$T_P \geq \min\{T_1/P, T_\infty\}.$$

"Greedy is good."

We can prove that a greedy scheduler achieves the sum of the above two lower bounds as an upper bound.

Theorem: On an ideal parallel computer (Cf. Page 37) with P processors, a greedy scheduler executes a multi-threaded computation with work T_1 and span T_∞ in time

$$T_P \leq T_1/P + T_\infty.$$

Moreover, a greedy scheduler always performs well in the sense that it performs as well as an optimal one within a factor of 2. (Cf. Page 14 of the Greedy algorithm notes.)

Finally, if P is significantly less than the parallelism, then the greedy algorithm could achieve a linear speedup.

Race condition

A multi-threaded algorithm is *deterministic* if it always does the same thing on the same input, no matter how the scheduling is done; otherwise, we call it non-deterministic. (Cf. Chapter 5 on the non-deterministic Hiring program)

An intended deterministic algorithm often fails to have this property if it contains a "determinacy race", since no one knows how a scheduler will allocate strands to processors.

Such race condition related bugs actually could lead to serious consequences.

Homework: What is wrong with `MAT-VET-WRONG` as shown in page 791?

A concrete example

A *determinacy race* occurs when two logically parallel instructions access the same memory location and at least one of them performs a write.

The following program contains the third buzzword **parallel**:

1. $x = 0$
2. **parallel for** $i = 1$ to 2
3. $\quad x = x + 1$
4. print x

Question: What should the sequential version print out? 2

Question: What would the parallel version print out?

Answer: It depends on a *race* between these two children processes.

What is happening here?

When a processor increments a memory location x, although this operation is atomic, it is involved with a sequence of three operations:

1. Read the value of x into a register.

2. Increment the value of that register.

3. Set the value from the register back to the location.

$r_1 = x;$	$r_2 = x;$
$r_1 + +;$	$r_2 + +;$
$x = r_1;$	$x = r_2;$

Question: With $\frac{6!}{3!3!}$ $(= 20)$ interleaving patterns, what could happen?

Answer: Who knows?

Anything is possible...

For the following "sequential" pattern, running on two cores

		x	r_1	r_2
x=0;		0	0	0
$r_1 = x$;		0	0	0
$r_1 + +$;		0	1	0
$x = r_1$;		1	1	0
	$r_2 = x$;	1	1	1
	$r_2 + +$;	1	1	2
	$x = r_2$;	2	1	2

at the end, $x \equiv 2$. But, r_1 gets lost with the following one.

		x	r_1	r_2
x=0;		0	0	0
$r_1 = x$;		0	0	0
	$r_2 = x$;	0	0	0
$r_1 + +$;		0	1	0
	$r_2 + +$;	0	1	1
$x = r_1$;		1	1	1
	$x = r_2$;	1	1	1

Algorithm analysis (again?)

Yet another line to estimate the speedup of a parallel algorithm is to analyze its parallelism (Cf. Page 42 of this set of notes), i.e., T_1/T_∞. It is relatively easy to count the *work*, T_1. (?)

Span, T_∞, is a bit more challenging. Let's use the $PFIB$ algorithm as an example. The original FIB procedure is essentially the serialization of $PFIB$. Thus, by the result shown on Page 24,

$$T_1(n) = T(n) = \Theta(\phi^n).$$

The span of two parallel computation is simply the maximum of the two spans (The longer of the two maximum paths). Thus

$$\begin{aligned} T_\infty(n) &= \max\{T_\infty(n-1), T_\infty(n-2)\} + \Theta(1) \\ &= T_\infty(n-1) + \Theta(1) = \Theta(n). \end{aligned}$$

The parallelism for $PFIB(n)$ is $T_1(n)/T_\infty(n)$ $= \Theta(\phi^n/n)$, which grows very fast as n grows, which implies a potentially large speedup.

Homework: Exercise 27.1-1.

Multi-threaded mergesort

Still remember this piece that we went through early in the course? Since it is already done in divide-and-conquer, we can easily convert it to a parallel version, $MERGESORT'(A, p, r)$ as follows;

1. if$p < r$
2. $q = (p + r)/2$
3. **spawn** $MERGESORT'(A, p, q)$
4. $MERGESORT'(A, q + 1, r)$
5. **sync**
6. $MERGE(A, p, q, r)$

Like its serial counterpart, with the **sync**, after two recursive calls, both the first and the second halves are sorted, then the same $MERGE$ wraps it up.

A bit analysis

The work of $MERGESORT'$ has been done earlier in the course (Cf. Page 46-48 in the Chapter 2 notes):

$$\begin{aligned} MS'_1(n) &= 2MS'(n/2) + \Theta(n) \\ &= \Theta(n \lg n). \end{aligned}$$

For the span, since the two sub calls run in parallel,

$$\begin{aligned} MS'_\infty(n) &= MS'_\infty(n/2) + \Theta(n) \\ &\stackrel{?}{=} \Theta(n). \end{aligned}$$

Therefore, its parallelism is $\Theta(\lg n)$, implying little speedup for *this version* of the parallel mergesort.

When sorting 10 million items, at most, we can get a speedup of about 20, no matter how many processors we will throw in. ☹

It could be better....

The bottleneck is certainly the serial merge procedure, which we can try to reshape into a parallel procedure. For details, you can look at §27.3 of the textbook. *It is thus important as how to convert a sequential algorithm into a parallel one.*

Indeed, the parallelism of this improved version of parallel mergesort is $\Theta(n/\lg^2 n)$, which is much better than the original $\Theta(\lg n)$.

For example, when sorting a million items, the most speedup we can hope for is thus about 2,500, instead of 20. ☺

We have talked a lot ☹, shall we walk a little? ☺

An example in *Chapel*

The following calculates the value of π by finding out the area of half of a circle with radius being 1.

```
[zshen@cs4310 sync]$ more syncExec.chpl
const numRect = 10000000;
const width = 2.0 / numRect; // rectangle width
// number of cores the computers processor has
const numThreads = here.maxTaskPar;
var globalSum: real = 0.0;

proc calculateArea(init) {
  var partialSum: real = 0.0;
  var x: real;
  var i: int = init;
  do {
    x = -1 + ( i + 0.5) * width;
    partialSum += sqrt(1.0 - x*x) * width;
    i += numThreads;
  } while (i < numRect-1);

globalSum += partialSum;

writeln("Thread: ", init, " globalSum: ", globalSum);
}

//The following sync will guarantee that this parallel loop will be
//completed first before the last writeln executes. --zs

sync coforall i in 1..numThreads {
  begin calculateArea(i);
}

writeln("This code estimates pi as ", globalSum*2);
```

The process and the result

The sixteen threads will start and finish randomly, and it looks like Thread 8 is the one that will finish last, and gives back the final answer.

```
[zshen@cs4310 sync]$ chpl syncExec.chpl
[zshen@cs4310 sync]$ ./syncExec
Thread: 1 globalSum: 0.0981748
Thread: 16 globalSum: 0.19635
Thread: 2 globalSum: 0.294524
Thread: 3 globalSum: 0.392699
Thread: 12 globalSum: 0.490874
Thread: 6 globalSum: 0.589049
Thread: 5 globalSum: 0.687223
Thread: 13 globalSum: 0.785398
Thread: 7 globalSum: 0.883573
Thread: 10 globalSum: 0.981748
Thread: 11 globalSum: 1.07992
Thread: 15 globalSum: 1.1781
Thread: 4 globalSum: 1.27627
Thread: 14 globalSum: 1.37445
Thread: 9 globalSum: 1.47262
Thread: 8 globalSum: 1.5708
This code estimates pi as 3.14159
```

There are lots of other stuff here, but we have to call a break... . ☺