Introduction to Parallel Computing in C++ with PASL

COLLABORATORS			
	TITLE : Introduction to Parallel Computing in C++ with PASL		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		December 12, 2014	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

List of Figures

1 Authors and Credits

These mini-course notes are written by Umut A. Acar, Arthur Chargueraud, and Mike Rainey.

Some of the material is based on, and sometimes directly borrows from, an undergraduate course, 15-210, co-taught with Guy Blelloch at CMU. The interested reader can find more details on parallel algorithm design in the book developed for this course.

One important difference from the 15-210 material and the book is that we use a functional language in that course, whereas these notes are based an a lower-level C++ library for parallelism.

Using C++ allows us to discuss efficiency and performance concerns in parallel computing on modern multicore machines.

2 Book/Course Abstract

The goal of this mini-course is to introduce the participant to the following.

1. The basic concepts of parallel computing.
2. Some basic parallel algorithm design principles and techniques,
3. Real-world performance and efficiency concerns in writing parallel software and techniques for dealing with them, and
4. Parallel programming in C++.

For parallel programming in C++, we use a library, called **PASL**, that we have been developing over the past 5 years. The implementation of the library uses advanced scheduling techniques to run parallel programs efficiently on modern multicores and provides a range of utilities for understanding the behavior of parallel programs.

We chose the name Parallel Algorithm Scheduling Library because the corresponding acronym, namely PASL, sounds a little like the French phrase "pas seul", meaning "not alone".

By following the instructions in this minibook, we expect that the reader will be able to write performant parallel programs at a relatively high level (essentially at the same level of C++ code) without having to worry too much about lower level details such as machine specific optimizations that might be otherwise needed.

All code that is associated with this course can be found at the Github repository linked by the following URL:

https://github.com/deepsea-inria/pasl/tree/edu

Moreover, every code example that is presented in the text also also appears in executable code. In the repository, you can find this code in the file located at `minicourse/examples.hpp`.

3 C++ Background

The material is entirely based on C++ and a library for writing parallel programs in C++. We use recent features of C++ such as closures or lambda expressions and templates. A deep understanding of these topics is not necessary to follow the course notes, because we explain them at a high level as we go, but such prior knowledge might be helpful; some pointers are provided below.

3.1 Template metaprogramming

For a quick introduction, please see these slides.

3.2 Lambda expressions

The C++11 reference provides good documentation on lambda expressions.

4 Lecture 1: Fork-join parallelism

This course is based entirely on one, perhaps simplest from of parallelism: fork-join. Fork-join parallelism, a fundamental model in parallel computing, dates back to 1963 and has been widely used in parallel computing since. In fork join, computations create opportunities for parallelism by branching at certain points that are specified by annotations in the program text. Each branching point "forks" the control flow of the computation into two or more logical threads. When control reaches the branching point, the branches start running. When all branches complete, the control "joins" back to unify the flows from the branches. Results computed by the branches are typically read from memory and merged at the join point. Parallel regions can fork and join recursively in the same manner that divide and conquer programs split and join recursively. In this sense, fork join is the divide and conquer of parallel computing.

4.1 Fork join in PASL

In PASL, fork join is expressed by application of the `fork2()` function. The function expects two arguments: one for each of the two branches. Each branch is specified by one C++ lambda expression.

Example 4.1 Fork join

In the sample code below, the first branch writes the value 1 into the cell `b1` and the second 2 into `b2`; at the join point, the sum of the contents of `b1` and `b2` is written into the cell `j`.

```
long b1 = 0;
long b2 = 0;
long j  = 0;

fork2([&] {
  // first branch
  b1 = 1;
}, [&] {
  // second branch
  b2 = 2;
});
// join point
j = b1 + b2;

std::cout << "b1 = " << b1 << "; b2 = " << b2 << "; ";
std::cout << "j = " << j << ";" << std::endl;
```

Output:

```
b1 = 1; b2 = 2; j = 3;
```

When this code runs, the two branches of the fork join are both run to completion. The branches may or may not run in parallel (i.e., on different cores). In general, the choice of whether or not any two such branches are run in parallel is chosen by the PASL runtime system. The join point is scheduled to run by the PASL runtime only after both branches complete. Before both branches complete, the join point is effectively blocked. Later, we will explain in some more detail the algorithms that the PASL uses to handle such load balancing and synchronization duties.

We sometimes use the term **thread** to refer to an individual sequence of instructions that do not contain calls to `fork2()`. The two branches passed to `fork2()` in the example above correspond, for example, to two independent threads. Moreover, the statement following the join point (i.e., the continuation) is also a thread.

Note

If the syntax in the code above is unfamiliar, it might be a good idea to review the notes on lambda expressions in C++11. In a nutshell, the two branches of `fork2()` are provided as lambda-expressions where all free variables are passed by reference.

Note
Fork join of arbitrary arity is readily derived by repeated application of binary fork join. As such, binary fork join is universal because it is powerful enough to generalize to fork join of arbitrary arity. We will see how to derive arbitrary arity later in the course.

All writes performed by the branches of the binary fork join are guaranteed by the PASL runtime to commit all of the changes that they make to memory before the join statement runs. In terms of our code snippet, all writes performed by the bodies of the first two branch statements are committed to memory before the join point can be scheduled. The PASL runtime guarantees this property by using a local barrier, which is efficient because this specific kind of barrier involves just a single dynamic synchronization point between at most two processors.

Important
To avoid a thorny issue called ***race conditions***, throughout this course, we are going to use the following discipline. For any two branches and for any cell in memory, the following holds: if at least one of the two branches can write to the cell, then the other branch can neither read nor write in the same cell. We will reject any program that can violate this condition. At the end of the course, however, we will see one example where we break with this discipline and use "atomic memory operations" to avoid race conditions.

You can read more about race conditions in this section.

Example 4.2 Writes and the join statement

Both writes performed on `b1` and `b2` are guaranteed to be performed before the print statement.

```
long b1 = 0;
long b2 = 0;

fork2([&] {
  b1 = 1;
}, [&] {
  b2 = 2;
});

std::cout << "b1 = " << b1 << "; b2 = " << b2 << std::endl;
```

Output:

```
b1 = 1; b2 = 2
```

PASL provides no guarantee on the visibility of writes between any two parallel branches. In the code just above, for example, writes performed by the first branch (e.g., the write to `b1`) may or may not be visible to the second, and vice versa. This is why for the purposes of this course, we reject programs buggy that rely on assumptions about timing of parallel reads-writes to shared memory cells.

4.2 Example 1: Parallel Fibonacci

Now, we have all the tools we need to describe our first parallel code: the recursive Fibonacci function. Although useless as a program because of efficiency issues, this example is the "hello world" program of parallel computing.

Recall that the n^{th} Fibonnacci number is defined by the recurrence relation

$$F(n) \quad = \quad F(n-1)+F(n-2)$$

with base cases

$$F(0)=0, F(1)=1$$

Let us start by considering a sequential algorithm. Following the definition of Fibonacci numbers, we can write the code for (inefficiently) computing the n^{th} Fibonnacci number as follows.

```
long fib_seq (long n) {
  long result;
  if (n < 2) {
    result = n;
  } else {
    long a, b;
    a = fib_seq(n-1);
    b = fib_seq(n-2);
    result = a + b;
  }
  return result;
}
```

Note
We say that the above Fibonacci algorithm is inefficient because the algorithm takes exponential time, whereas there exist dynamic programming solutions that take linear time.

To write a parallel version, we remark that the two recursive calls are completely ***independent***: they do not depend on each other (neither uses a piece of data generated or written by another). We can therefore perform the recursive calls in parallel. In general, any two independent functions can be run in parallel. To indicate that two functions can be run in parallel, we use `fork2()`.

```
long fib_par(long n) {
  long result;
  if (n < 2) {
    result = n;
  } else {
    long a, b;
    fork2([&] {
      a = fib_par(n-1);
    }, [&] {
      b = fib_par(n-2);
    });
    result = a + b;
  }
  return result;
}
```

4.3 The sequential elision

In the Fibonacci example, we started with a sequential algorithm and derived a parallel algorithm by annotating independent functions. It is also possible to go the other way: derive a sequential algorithm from a parallel one. As you have probably guessed this direction is easier, because all we have to do is remove the calls to the `fork2` function. The sequential elision of our parallel Fibonacci code can be written by replacing the call to `fork2()` with a statement that performs the two calls (arguments of `fork2()`) sequentially as follows.

```
long fib_par(long n) {
  long result;
  if (n < 2) {
    result = n;
  } else {
    long a, b;
    ([&] {
      a = fib_par(n-1);
    })();
    ([&] {
      b = fib_par(n-2);
```

```
    })();
    result = a + b;
  }
  return result;
}
```

Note
Although this code is slightly different than the sequential version that we wrote, it is not too far away, because the only the difference is the creation and application of the lambda-expressions. An optimizing compiler for C++ can easily "inline" such computations. Indeed, After an optimizing compiler applies certain optimizations, the performance of this code the same as the performance of `fib_seq`.

Note
Programming languages and systems that support the ability to derive a sequential program from a parallel program by way of syntactic substitutions are sometimes called ***implicitly parallel***. PASL is one such system.

The sequential elision is often useful for performance tuning because the sequentialized code helps us to isolate purely algorithmic overheads that are introduced by parallel codes. By isolating these costs, we can more effectively pinpoint inefficiencies in our code. Debugging is another use of the sequential elision: it is much easier to find bugs in sequential runs of parallel code than in parallel runs of the same code.

4.4 Race conditions [Optional Material]

Although we already described the essentials of the of the memory model that is provided by PASL, we return for the moment to the subject of ***race conditions***. Special attention is crucial because race conditions introduce especially pernicious form error that can confound beginners and experts alike. In general, a race condition is any behavior in a program that is determined by some uncontrollable feature of the system, such as timing. In PASL, a race condition can occur whenever two parallel branches access the same location in memory and at least one of the two branches performs a write. When this situation occurs, the behavior of the program may be undefined, leading to (often) buggy behavior. We used the work "may" here because certain programs use race conditions in a careful way as a means to improve performance. However, such programs are rare and out of the scope of this course.

Race conditions can be highly problematic because, owing to their nondeterministic behavior, they are often extremely hard to debug. To make matters worse, it is also quite easy to create race conditions without even knowing it. Take a look at the following variant on our parallel Fibonacci function.

```
long fib_par_racy(long n) {
  long result = 0;
  if (n < 2) {
    result = n;
  } else {
    fork2([&] {
      result += fib_par_racy(n-1);
    },[&] {
      result += fib_par_racy(n-2);
    });
  }
  return result;
}
```

Question

Is this code correct when run on a single processor?

Question

Can you spot the race condition?

The race condition is easier to spot if we expand out the applications of the += operator.

```
long fib_par_racy(long n) {
  long result = 0;
  if (n < 2) {
    result = n;
  } else {
    fork2([&] {
      long a1 = fib_par_racy(n-1);
      long a2 = result;
      result = a1 + a2;
    },[&] {
      long b1 = fib_par_racy(n-2);
      long b2 = result;
      result = b1 + b2;
    });
  }
  return result;
}
```

When written in this way, it is clear that these two parallel threads are not independent: they both read `result` and write to `result`. Thus the outcome depends on the order in which these reads and writes are performed, as shown in the next example.

Example 4.3 Execution trace of a race condition

The following table takes us through one possible execution trace of the call `fib_par_racy(2)`. The number to the left of each instruction describes the time at which the instruction is executed. Note that since this is a parallel program, multiple instructions can be executed at the same time. The particular execution that we have in this example gives us a bogus result: the result is 0, not 1 as it should be.

	Thread 1		**Thread 2**
1	`a1 = fib_par_racy(1)`	1	`b2 = fib_par_racy(0)`
2	`a2 =result`	2	`b3 =result`
3	`result =a1 + a2`	4	`result =b1 + b2`

The reason we get a bogus result is that the second thread wins the race to write to the cell named `result`. The value 1 written by the first thread is effectively lost by being overwritten by the second thread.

Note

Research on the Cilk system pioneered the technology of "online race-condition detectection" for fork-join programs. The product of this research is an online race detection algorithm named Cilkscreen. Cilkscreen monitors every read and write issued by the program, enabling Cilkscreen to detect whenever a race occurs.

4.5 Parallel-Algorithm Design Techniques

The Fibonacci example that we considered happened to be defined in a way that makes it amenable to parallelization. This is not always the case. It can sometimes take more work to design a good parallel algorithm. We will now see that in fact, parallel algorithms require a different design strategy. At a very high level, this design strategy can be viewed as identifying dependencies in the computation and eliminating them as much as possible, leaving only those that are essential.

4.6 Example 2: incrementing an array, sequentially

Suppose that we wish to map an array to another by incrementing each element by one. We can write the code for a function `map_incr` that performs this computation serially.

```
void map_incr(const long* source, long* dest, long n) {
  for (long i = 0; i < n; i++)
    dest[i] = source[i] + 1;
}
```

Example 4.4 Example use of `map_incr`

```
const long n = 4;
long xs[n] = { 1, 2, 3, 4 };
long ys[n];
map_incr(xs, ys, n);
for (long i = 0; i < n; i++)
  std::cout << ys[i] << " ";
std::cout << std::endl;
```

Output:

```
2 3 4 5
```

Question

Is this a good parallel algorithm?

This algorithm for incrementing the array is not a good parallel algorithm because it exposes no parallelism. Specifically, there are no independent function calls that we can parallelize.

To see this more visually, we can use a ***Directed Acyclic Graph (DAG)***, shown in the Figure 1. In this DAG, each vertex represents an addition operation and each edge represents a dependency between its source and target. In this case, these dependencies are control dependencies that show that the target operation is executed after the source instruction. Since the algorithm is an inherently sequential algorithm, the DAG consists of a long chain of operations.

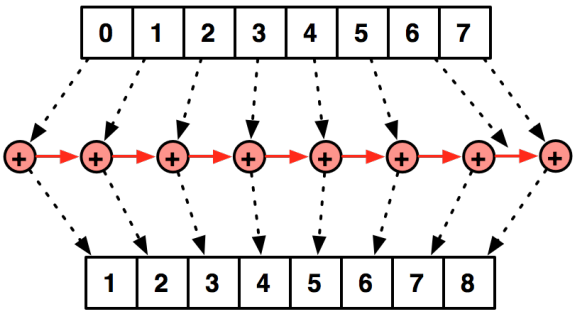


Figure 1: DAG for serial increment

Question

Can you think of a parallel algorithm for incrementing the elements of the array?

4.7 Example 3: Incrementing an array, in parallel

Note that there are no inherent data dependencies between the increment and write operations performed on the array: no addition operations reads the result of another. We can therefore increment the elements of the array in parallel. When designing a parallel algorithm, it is essential to realize such independent computations. To take advantage of this parallelism, we will write a divide-and-conquer algorithm.

Question

Can you think of a sequential divide-and-conquer algorithm for incrementing the elements of the array?

Such a divide-and-conquer algorithm is shown below. The idea behind the algorithm is to recursively divide the array into two halves and map each half to the result in parallel.

```
void map_incr(const long* source, long* dest, long n) {
  map_incr_rec(source, dest, 0, n);
}

void map_incr_rec(const long* source, long* dest, long lo, long hi) {
  long n = hi - lo;
  if (n == 0) {
    // do nothing
  } else if (n == 1) {
    dest[lo] = source[lo] + 1;
  } else {
    long mid = (lo + hi) / 2;
    map_incr_rec(source, dest, lo, mid);
    map_incr_rec(source, dest, mid, hi);
  }
}
```

Note

For those readers who are familiar with C++, the above solution may seem naive because the recursion goes all the way down to single items. Indeed, the costs involved are heavy relative to a solution which coarsens the base cases. We will return to this issue later, because in the world of parallel programming, coarsening is trickier than it may seem at first.

Now that we have a divide-and-conquer algorithm, we are quite close to parallelizing it.

Question

Can you see how to parallelize the divide-and-conquer algorithm?

Note that the two recursive calls are independent (they do not use data produced by the other) and they are devoid of race conditions (they do not write to the same memory cells). Thus, the recursive calls can be executed in parallel. We thus use our `fork2` function to indicate that these two functions are indeed parallel.

```
void map_incr_rec(const long* source, long* dest, long lo, long hi) {
  long n = hi - lo;
  if (n == 0) {
    // do nothing
  } else if (n == 1) {
    dest[lo] = source[lo] + 1;
  } else {
    long mid = (lo + hi) / 2;
    fork2([&] {
      map_incr_rec(source, dest, lo, mid);
    }, [&] {
```

```
      map_incr_rec(source, dest, mid, hi);
    });
  }
}
```

As before, we can use a DAG to visualize this computation. Such a DAG is shown in Figure 2. In this DAG, each vertex represents a call to function `map_incr_rec` or a join of two such parallel calls. The key point to notice here is that the ***depth*** of the DAG, defined as the longest directed path, is small compared to the total number of vertices in the DAG.

Question

If the input size is n, then what is the depth of the DAG?

On a large input, the depth of the DAG is logarithmic even when there are linearly many vertices in the DAG. We will make this notion precise by developing a precise cost model for reasoning about parallelism.

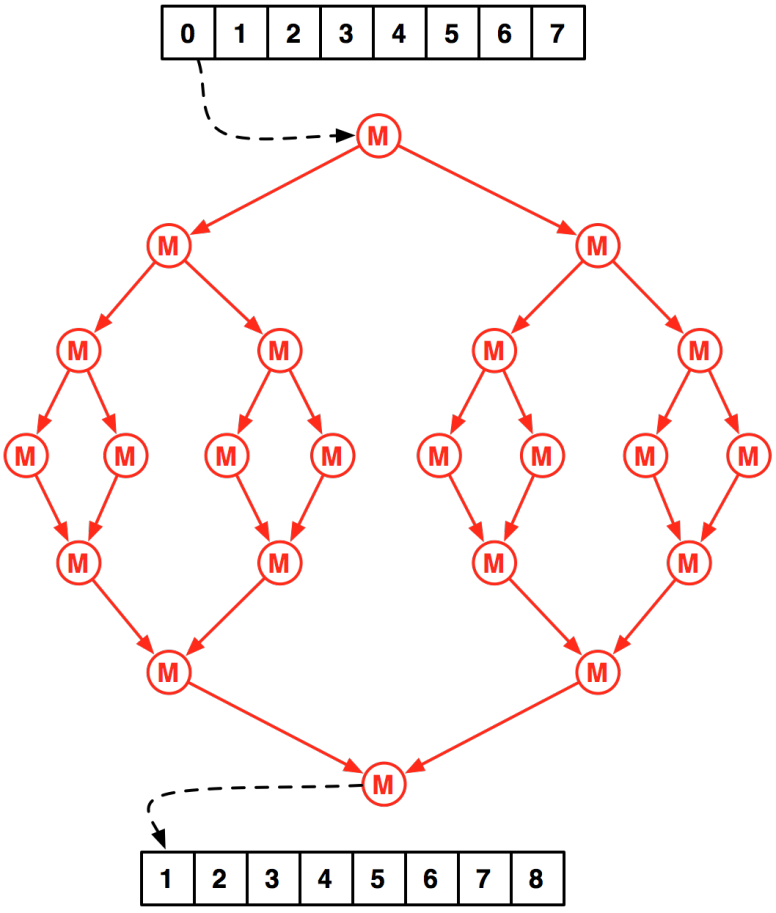


Figure 2: DAG for parallel increment

Note

The serial increment algorithm that we considered in the previous section created totally ordered the individual increment operations even though they could have been performed in parallel. This is common in many sequential algorithms, which often totally order computations that need not be ordered.

4.8 Summary of Lecture 1

In this lecture, we covered a simple parallel algorithm for computing Fibonacci numbers and another example, where the standard iterative algorithm does not yield a good parallel algorithm.

4.9 Administrivia

Announcement: Team Formation

Before we finish the lecture, we would like to make a short announcement. We would like you to form team's consisting of 2-4 students for doing some projects over the weekend. Each team should pick a name (names can be anything that you like) and let us know the name of their team. We don't need to know your actual identity, nor do your friends, though here is nothing wrong in making your team identity public. We will create an account for each team on a multicore machines.

5 Lecture 2: Asymptotic analysis for parallel algorithms

To design and implement good parallel algorithms, it is essential to reason about efficiency both at a high level and with high precision. Fortunately, thanks to the mature body of research on parallel computation, these two criteria are not mutually exclusive. A key result from this research is that we can achieve our goals by analyzing our algorithms with respect to a certain ***cost model***.

Our cost model is based on the idea of asymptotically measuring the work and span of a computation. In this model, we will assign to each instruction (or, more generally, to each block of instructions of at most some bounded size) a ***weight*** of "1" and calculate work and span as the total weight of a DAG and a heaviest path in the DAG. For example, considering the DAG for the serial increment function as shown in Figure 3, we can assign a weight of 1 to each vertex and compute the work and span of the computation.

Definition: Work and Span

For a DAG, we define its work and span as follows.

- ***Work:*** The total weight of all the vertices.
- ***Span:*** The weight of the path with the heaviest total path weight.

A helpful intuition for span is to think of it as the time that we would need to complete a computation if we have an unlimited (infinite) supply of processors.

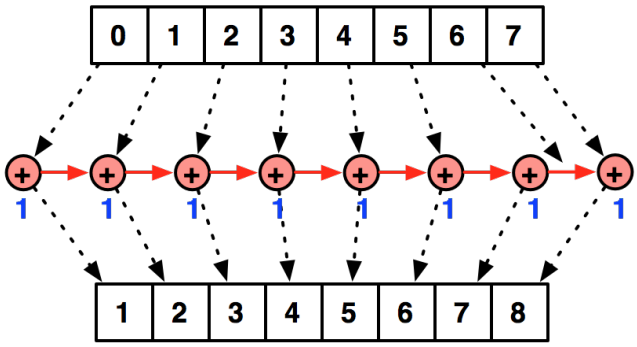


Figure 3: Costed DAG of serial increment

Question

What is the work and span for the serial array-increment algorithm?

The work and span are both 8. For an array with n elements, the work and span would both be asymptotically $\Theta(n)$. We often use the functions $W(\cdot)$ and $S(\cdot)$ to refer to work and span as function of the input size for the computation.

Question

We argued that the sequential array-increment function is not a good parallel algorithm. Can you justify why?

The algorithm is not a good parallel algorithm because it exposes no parallelism. We observe this by noting that the span and work of the algorithm are both the same, i.e., $W(n) = S(n) = \Theta(n)$. Good parallel algorithms have significantly smaller spans (typically poly-logarithmic) than total work.

Example 5.1 Work and Span of Parallel Increment

We can analyze the work and span of our parallel increment algorithm by assigning a cost to each vertex of its DAG, as shown in Figure 4. In the example, the span is 7 and the work is 22.

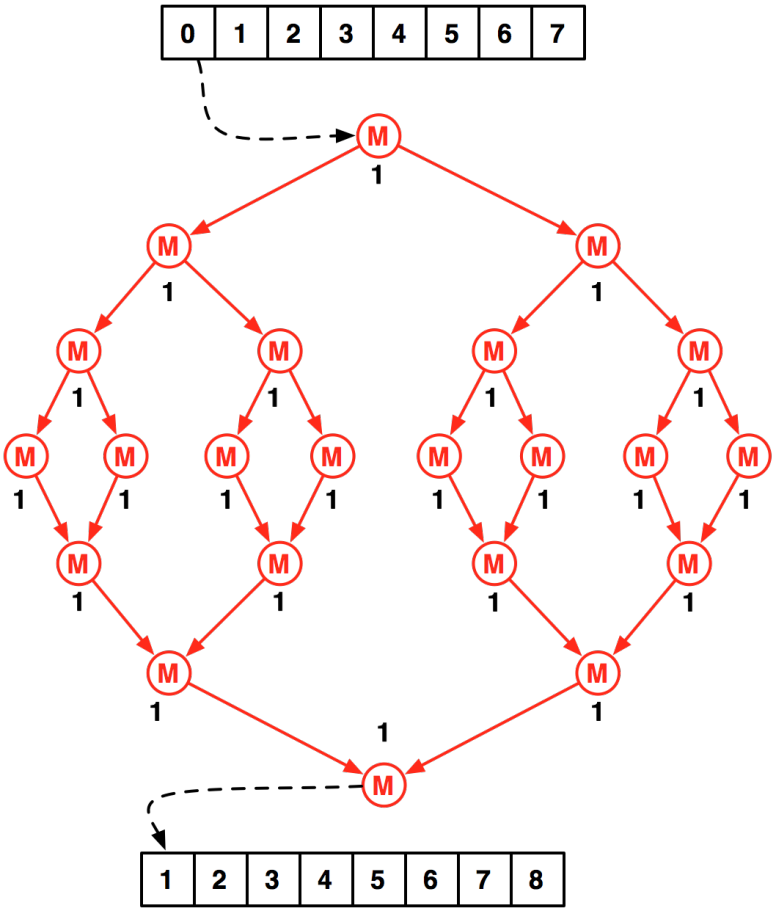


Figure 4: Costed DAG for parallel map.

More abstractly, we analyze work and span by setting up a recursion. For parallel increment the work and span can be written as follows:

$$W(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ W(n/2) + W(n/2) + 1 \end{cases}$$

$$S(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \max(W(n/2), W(n/2)) + 1 \end{cases}$$

EQUATION 5.1: Analyzing work and span, recursively

It is easy to solve both of these recursions, to obtain

1. $W(n) = \Theta(n)$.
2. $S(n) = \Theta(\log n)$.

Since our algorithm has logarithmic span, it is a good parallel algorithm. Indeed, it has an average parallelism of $\Theta(n/\log n)$, which grows quickly with input size.

Example 5.2 Work and Span of Parallel Fibonacci

$$W(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ W(n-1) + W(n-2) + 1 \end{cases}$$

$$S(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \max(W(n-1), W(n-2)) + 1 \end{cases}$$

EQUATION 5.2: Analyzing work and span, recursively

It is easy to solve both of these recursions, to obtain

1. $W(n) = \Theta(Fib(n))$.
2. $S(n) = \Theta(n)$.

Is this a good parallel algorithm? To see that it is, note that the n^{th} Fibonacci number is quite large and, in fact, is exponential in n. Thus, average parallelism is high: there is plenty of parallelism is the parallel algorithm for computing Fibonacci numbers.

To complete our cost model, we define one more term. The purpose of this term is to quantify the amount of parallelism in a given parallel computation.

Definition: Average Parallelism:

For a DAG with work $W(n)$ and span $S(n)$, ***average parallelism*** is defined as $\mathbb{P} = \frac{W(n)}{S(n)}$.

Work, Span, and Average Parallelism

Intuitively average parallelism indicates the average number of independent paths through the computation that can be pursued at the same time. In other words, if we view span as the "depth" of the computation we can view average parallelism as the "width" of the computation. Depth times width gives us the area, which we can think of as the work.

Maximum theoretical speedup

The average parallelism is the maximum theoretical speedup that can be achieved by a parallel algorithm, at least, according to our cost model. As such, if the average parallelism is $\mathbb{P}$, then using any more than $\mathbb{P}$ processors cannot speed up the computation.

5.1 Scheduling

By allowing us to reason about the parallelism available in an algorithm in terms of its work and span, the cost model that we presented enables us to design theoretically good parallel algorithms. However, the model has nothing to say about what kind of performance we might expect from a parallel execution with a given number of processors, say P.

Question

Can good parallel algorithms (i.e., algorithms whose span relative to their work are small) be executed efficiently in parallel?

This question turns out to be a highly non-trivial question, leading to many other algorithmic and engineering questions. In fact, it would be fair to say that there is a whole field called ***parallel-algorithm scheduling*** that studies this and related questions. At a high level, scheduling is challenging because of three reasons:

1. A parallel algorithm generates threads online as it runs, requiring the scheduling algorithm to make online decisions that can be expensive and imprecise.
2. A parallel algorithm can generate a massive number of threads, typically much more than the number of processors available when running, possibly leading to large overheads.
3. The running time of each thread can vary from a small period to a large period, with no a priori knowledge. This property makes it difficult to make online decisions about whether it would be worthwhile to move a thread or not for the purposes of balancing load between processors.

Definition: Scheduler

A scheduler is as algorithm that maps parallel threads to available processors. The scheduler works by assigning the parallel threads, which are generated dynamically as the algorithm evaluates, to processors.

For example, if there is only one processor is available, a scheduler would map all threads to that one processor. If two processors are available, the scheduler can divide the threads between the two processors as evenly as possible, in an attempt to keep the two processors as busy as possible.

Question

Can you think of a scheduling algorithm?

5.1.1 Greedy scheduling

We say that a scheduler is ***greedy*** if, whenever there is a processor available and a thread ready to be executed, then the scheduler assigns the thread to the processor and starts running the thread immediately. Greedy schedulers have a nice property that is summarized by the following:

Definition: Greedy Scheduling Principle

If a computation is run on p processors using a "perfect" greedy scheduler that incurs no costs in creating and moving threads, then the total time (clock cycles) for running the computation T_p is bounded by

$$T_p < \frac{W}{p} + S.$$

Here W is the work of the computation, and S is the span of the computation (both measured in units of clock cycles).

This simple statement is powerful. On the one hand, the time to execute the computation is at least $\frac{W}{p}$ because we have a total of W work. As such, the best possible execution strategy is to divide it evenly among the processors. On the other hand, execution time cannot be less than S since S represents the longest chain of sequential dependencies. Thus we have: $T_p \geq \max\left(\frac{W}{p}, S\right)$.

We therefore see that a greedy scheduler performs reasonably close to the best possible. In particular $\frac{W}{p} + S$ is never more than twice $\max(\frac{W}{p}, S)$. Furthermore, when $\frac{W}{p} \gg S$, the difference between the greedy scheduler and the optimal scheduler is very small. In fact, we can rewrite equation above in terms of the average parallelism $\mathbb{P} = W/S$ as follows:

$$\begin{aligned} T_p &< \frac{W}{p} + S \\ &= \frac{W}{p} + \frac{W}{\mathbb{P}} \\ &= \frac{W}{p}\left(1 + \frac{p}{\mathbb{P}}\right) \end{aligned}$$

Therefore as long as $\mathbb{P} \gg p$ (the parallelism is much greater than the number of processors) then we get near perfect speedup. (Speedup is W/T_p and perfect speedup would be p).

Greedy scheduling in practice

PASL implements a probabilistically greedy scheduling algorithm that is based on a technique called ***work stealing***. PASL's scheduling algorithm is not perfectly greedy, because there is overhead in scheduling the threads. Specifically, creating and moving threads requires additional work. In these notes, we will refer to such scheduling-related work as ***friction***. In many cases, however, the friction can be reduced to negligible amounts, allowing us to match greedy scheduling principle in practice.

5.2 Other approaches to parallelization [Optional Material]

Broadly speaking, the approach that we just studied, that is, the approach to parallel computation based on work-span analysis and a separate scheduling is sometimes referred to as ***dynamic parallelism***. It is named so because dynamic parallelism allows a parallel program to generate parallel threads as it runs. There has also been much work on what might be called ***static parallelism***. The idea behind this approach is to determine, statically (before inputs are available), the amount of parallelism to be created. Although relatively simple as a model of parallel computation, static scheduling has a number of key limitations.

Recall the array-map example that we studied in the previous chapter. The static schedule for this computation is one that divides the positions of the array up front into p blocks, where p is the number of processors, and assigns the blocks accordingly to be run by different processors. One major problem with this approach is that the run time of any given block is unpredictable, owing to platform-related effects that are out of our control. For example, paging or cache effects may very well cause one of the P blocks to finish well before the others. After finishing its assigned block, that processor is wasted because the processor is left idling.

An improvement would be to divide the array into KP blocks, for some fixed positive integer K and use a load-balancing algorithm to adaptively assign blocks to processors. Unfortunately, this solution suffers from a particular problem relating to nesting. That is, when parallel loops are nested i levels deep, the number of pieces of parallel work that are generated is $(KP)^i$. Even though K is typically a small constant, the cost of building $(KP)^i$ pieces of parallel work becomes significant for large P.

Of course, the programmer could try to manually sequentialize all of the interior loops in the program. However, this approach turns out to be problematic for modular software, because in this setting, it is not always possible to identify what code may be run from the body of any given loop.

Another possibility is to divide the computation up front into n/s blocks, where n is the size of the input array and s is some fixed constant, and then serially spawn one parallel thread for each block. The problem with this approach is that the serial work required to spawn the blocks may over sequentialize by introducing unnecessary control dependencies.

The approach that we are going to take consists of a few key ingredients. First, we use divide and conquer to expose up front all logically available parallelism in our algorithms. Second, we rely on smart load-balancing and granularity-control algorithms to make this strategy practical for the multicore systems that we have today. We already briefly described PASL's load-balancing algorithm. Later, we will get the complete picture when we describe the automatic granularity-control technique used by PASL.

Question

What is the work and span for the parallel versions of `map_incr` which serially generates:

- KP parallel blocks, and
- n/s parallel blocks?

5.3 Summary of Lecture 2

We covered the work-span model for designing parallel algorithms and scheduling, which enables the work-span model to remain relevant and realistic with respect to practical concerns.

6 Lecture 3 (Lab): Getting started with experiments

We are now going to study the practical performance of our parallel algorithms written with PASL on multicore computers.

To be concrete with our instructions, we assume that our username is `pasl` and that our home directory is `/home/pasl/`. You need to replace these settings with your own where appropriate.

But before we start the lecture, we wish to take a slight detour for an administrative matter.

6.1 Administrivia

Announcement: Team Formation

Each team: please write the name of your team on the sheet of paper that is being passed around. We will create an account for you to do a project.

6.2 Software Setup

You can skip this section if you are using a computer already setup by us or you have installed an image file containing our software.

6.2.1 Check for software dependencies

Currently, the software associated with this course supports Linux only. Any machine that is configured with a recent version of Linux and has access to at least two processors should be fine for the purposes of this course. Before we can get started, however, the following packages need to be installed on your system.

Software dependency	Version	Nature of dependency
`gcc`	>= 4.9.0	required to build PASL binaries
`php`	>= 5.3.10	required by PASL makefiles to build PASL binaries
`ocaml`	>= 4.0.0	required to build the benchmarking tools (i.e., pbench and pview)
`R`	>= 2.4.1	required by benchmarking tools to generate reports in bar plot and scatter plot form

Software dependency	Version	Nature of dependency
`latex`	recent	optional; required by benchmarking tools to generate reports in tabular form
`git`	recent	optional; can be used to access PASL source files
`tcmalloc`	>= 2.2	optional; may be useful to improve performance of PASL binaries
`hwloc`	recent	optional; might be useful to improve performance on large systems with NUMA (see below)

6.2.2 Fetch source files and configure

Let us change to our home directory: `/home/pasl`.

The PASL sources that we are going to use are part of a branch that we created specifically for this course. You can access the sources either via the tarball linked by the github webpage or, if you have `git`, via the command below.

```
$ git clone -b edu https://github.com/deepsea-inria/pasl.git
```

This rest of this section explains what are the optional software dependencies and how to configure PASL to use them. We are going to assume that all of these software dependencies have been installed in the folder `/home/pasl/Installs/`.

6.2.3 Use a custom parallel heap allocator

At the time of writing this document, the system-default implementations of `malloc` and `free` that are provided by Linux distributions do not scale well with even moderately large amounts of concurrent allocations. Fortunately, for this reason, organizations, such as Google and Facebook, have implemented their own scalable allocators that serve as drop-in replacements for `malloc` and `free`. We have observed the best results from Google's allocator, namely, tcmalloc. Using tcmalloc for your own experiements is easy. Just add to the `/home/pasl/pasl/minicourse` folder a file named `settings.sh` with the following contents.

Example 6.1 Configuration to select tcmalloc

We assume that the package that contains `tcmalloc`, namely `gperftools`, has been installed already in the folder `/home/pasl/Installs/gperftools-install/`. The following lines need to be in the `settings.sh` file in the `/home/pasl/pasl/minicourse` folder.

```
USE_ALLOCATOR=tcmalloc
TCMALLOC_PATH=/home/pasl/Installs/gperftools-install/lib/
```

Also, the environment linkder needs to be instructed where to find `tcmalloc`.

```
export LD_PRELOAD=/home/pasl/Installs/gperftools-install/lib/libtcmalloc.so
```

This assignment can be issued either at the command line or in the environment loader script, e.g., `~/.bashrc`.

Warning
Changes to the `settings.sh` file take effect only after recompiling the binaries.

6.2.4 Use `hwloc`

If your system has a non-uniform memory architecture (i.e., NUMA), then you may improve performance of PASL applications by using optional support for `hwloc`, which is a library that reports detailed information about the host system, such as NUMA

layout. Currently, PASL leverages `hwloc` to configure the NUMA allocation policy for the program. The particular policy that works best for our applications is round-robin NUMA page allocation. Do not worry if that term is unfamiliar: all it does is disable NUMA support, anyway!

Example 6.2 How to know whether my machine has NUMA

Run the following command.

```
$ dmesg | grep -i numa
```

If the output that you see is something like the following, then your machine has NUMA. Otherwise, it probably does not.

```
[    0.000000] NUMA: Initialized distance table, cnt=8
[    0.000000] NUMA: Node 4 [0,80000000) + [100000000,280000000) -> [0,280000000)
```

We are going to assume that `hwloc` has been installed already and is located at `/home/pasl/Installs/hwloc-install/`. To configure PASL to use `hwloc`, add the following lines to the `settings.sh` file in the `/home/pasl/pasl/minicourse` folder.

Example 6.3 Configuration to use `hwloc`

```
USE_HWLOC=1
HWLOC_PATH=/home/pasl/Installs/hwloc-install/
```

6.3 Fetch the benchmarking tools (pbench)

We are going to use two command-line tools to help us to run experiments and to analyze the data. These tools are part of a library that we developed, which is named pbench. The pbench sources are available via github.

```
$ cd /home/pasl
$ git clone https://github.com/deepsea-inria/pbench.git
```

The tarball of the sources can be downloaded from the github page.

6.3.1 Build the tools

The following command builds the tools, namely `prun` and `pplot`. The former handles the collection of data and the latter the human-readable output (e.g., plots, tables, etc.).

```
$ make -C /home/pasl/pbench/
```

Make sure that the build succeeded by checking the pbench directory for the files `prun` and `pplot`. If these files do not appear, then the build failed.

6.3.2 Create aliases

We recommend creating the following aliases.

```
$ alias prun='/home/pasl/pbench/prun'
$ alias pplot='/home/pasl/pbench/pplot'
```

It will be convenient for you to make these aliases persistent, so that next time you log in, the aliases will be set. If you are using the bash shell, add the following lines to the bottom of your `/home/pasl/.bashrc` file.

```
alias prun='/home/pasl/pbench/prun'
alias pplot='/home/pasl/pbench/pplot'
```

6.4 Visualizer Tool

When we are tuning our parallel algorithms, it can be helpful to visualize their processor utilization over time, just in case there are patterns that help to assign blame to certain regions of code. Later, we are going to use the utilization visualizer that comes packaged along with PASL. To build the tool, run the following make command.

```
$ make -C /home/pasl/pasl/tools/pview pview
```

Let us create an alias for the tool.

```
$ alias pview='/home/pasl/pasl/tools/pview/pview'
```

We recommend that you make this alias persistent, just like you did above for the pbench tools.

6.5 Using the Makefile

PASL comes equipped with a `Makefile` that can generate several different kinds of executables. These different kinds of executables and how they can be generated is described below for a benchmark program `pgm`.

- baseline: build the baseline with command `make pgm.baseline`
- elision: build the sequential elision with command `make pgm.elision`
- optimized: build the optimized binary with command `make pgm.opt`
- log: build the log binary with command `make pgm.log`
- debug: build the debug binary with the command `make pgm.dbg`

To speed up the build process, add to the `make` command the option `-j` (e.g., `make -j pgm.opt`). This option enables `make` to parallelize the build process. Note that, if the build fails, the error messages that are printed to the terminal may be somewhat garbled. As such, it is better to use `-j` only if after the debugging process is complete.

6.6 Task 1: Run the baseline Fibonacci program

We are going to start our experimentation with three different instances of the same program, namely `bench`. This program is what we will use as the entry point for every benchmark that we are going to examine. We are ready to build the benchmark program.

```
$ cd /home/pasl/pasl/minicourse
$ make bench.baseline
```

Warning
The command-line examples that we show here assume that you have `.` in your `$PATH`. If not, you may need to prefix command-line calls to binaries with `./` (e.g., `./bench.baseline`).

The file extension `.baseline` means that every benchmark in the binary uses the sequential-baseline version of the specified algorithm.

The `-bench` argument selects the benchmark to be run and the `-n` argument the input value for the Fibonacci function.

```
$ bench.baseline -bench fib -n 39
```

On our machine, the output of this run is the following.

```
exectime 0.556
utilization 1.0000
result 63245986
```

The three lines above provide useful information about the run.

- The `exectime` indicates the wall-clock time in seconds that is taken by the benchmark. In general, this time measures only the time taken by the benchmark under consideration. It does not include the time taken to generate the input data, for example.
- The `utilization` relates to the utilization of the processors available to the program. In the present case, for a single-processor run, the utilization is by definition 100%. We will return to this measure soon.
- The `result` field reports a value computed by the benchmark. In this case, the value is the 39^{th} Fibonacci number.

6.7 Task 2: Run the sequential elision of the Fibonacci program

The `.elision` extension means that parallel algorithms (not sequential baseline algorithms) are compiled. However, all instances of `fork2()` are erased the fashion described in the previous chapter.

```
$ make bench.elision
$ bench.elision -bench fib -n 39
```

The run time of the sequential elision in this case is similar to the run time of the sequential baseline because the two are similar codes. However, for most other algorithms, the baseline will typically be at least a little faster.

```
exectime 0.553
utilization 1.0000
result 63245986
```

6.8 Task 3: Run the parallel Fibonacci program

The `.opt` extension means that the program is compiled with full support for parallel execution. Unless specified otherwise, however, the parallel binary uses just one processor.

```
$ make bench.opt
$ bench.opt -bench fib -n 39
```

The output of this program is similar to the output of the previous two programs.

```
exectime 0.553
utilization 1.0000
result 63245986
```

Because our machine has 40 processors, we can run the same application using all available processors.

```
$ bench.opt -bench fib -n 39 -proc 40
```

We see from the output of the 40-processor run that our program ran faster than the sequential runs. Moreover, the `utilization` field tells us that approximately 86% of the total time spent by the 40 processors was spent performing useful work, not idling.

```
exectime 0.019
utilization 0.8659
result 63245986
```

Warning
PASL allows the user to select the number of processors by the `-proc` key. The maximum value for this key is the number of processors that are available on the machine. PASL raises an error if the programmer asks for more processors than are available.

6.9 Measuring performance with "speedup"

We may ask at this point: What is the improvement that we just observed from the parallel run of our program? One common way to answer this question is to measure the "speedup".

Definition: P-processor speedup

The speedup on P processors is the ratio T_B/T_P, where the term T_B represents the run time of the sequential baseline program and the term T_P the time measured for the P-processor run.

The importance of selecting a good baseline

Note that speedup is defined with respect to a baseline program. How exactly should this baseline program be chosen? One option is to take the sequential elision as a baseline. The speedup curve with such a baseline can be helpful in determining the scalability of a parallel algorithm but it can also be misleading, especially if speedups are taken as a indicator of good performance, which they are not because they are only relative to a specific baseline. For speedups to be interpreted as an indicator of good performance, they must be measured against a baseline that is identical or close to the optimal sequential algorithm (for the same problem.)

The speedup at a given number of processors is a good starting point on the way to evaluating the scalability of the implementation of a parallel algorithm. The next step typically involves considering speedups taken from varying numbers of processors available to the program. The data collected from such a speedup experiment yields a ***speedup curve***, which is a curve that plots the trend of the speedup as the number of processors increases. The shape of the speedup curve provides valuable clues for performance and possibly for tuning: a flattening curve suggests lack of parallelism; a curve that arcs up and then downward suggests that processors may be wasting time by accessing a shared resource in an inefficient manner (e.g., false sharing); a speedup curve with a constant slope indicates at least some scaling.

Example 6.4 Speedup for our run of Fibonacci on 40 processors

The speedup T_B/T_{40} equals $0.556/0.019 = 29.26$x. Although not linear (i.e., 40x), this speedup is decent considering factors such as: the capabilities of our machine; the overheads relating to parallelism; and the small size of the problem compared to the massive computing power that our machine offers.

6.9.1 Generate a speedup plot

Let us see what a speedup curve can tell us about our parallel Fibonacci program. We need to first get some data. The following command performs a sequence of runs of the Fibonacci program for varying numbers of processors. You can now run the command yourself.

```
$ prun speedup -baseline "bench.baseline" -parallel "bench.opt -proc 1,10,20,30,40" -bench  ↩
    fib -n 39
```

Run the following command to generate the speedup plot.

```
$ pplot speedup
```

If successful, the command generates a file named `plots.pdf`. The output should look something like the plot in Figure 5.

```
Starting to generate 1 charts.
Produced file plots.pdf.
```

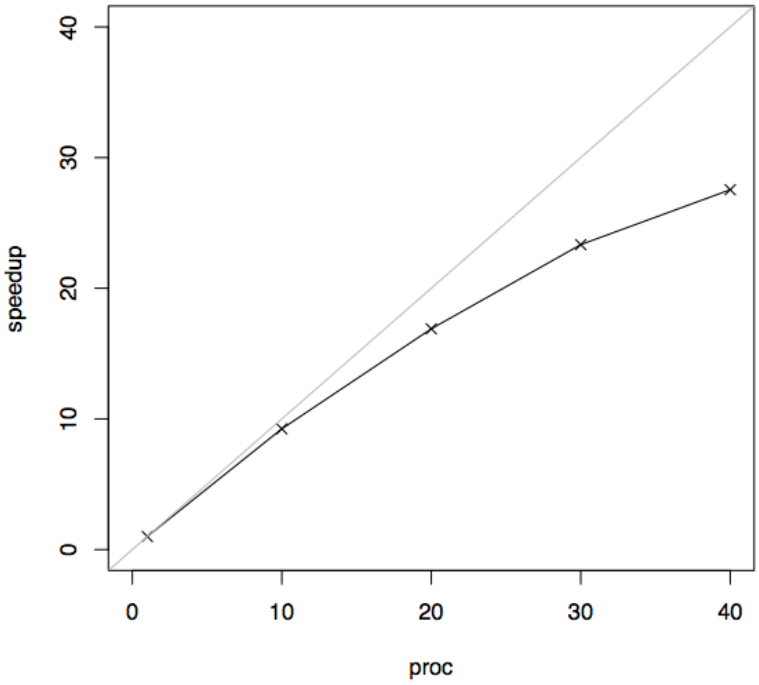


Figure 5: Speedup curve for the computation of the 39th Fibonacci number.

The plot shows that our Fibonacci application scales well, up to about twenty processors. As expected, at twenty processors, the curve dips downward somewhat. We know that the problem size is the primary factor leading to this dip. How much does the problem size matter? The speedup plot in Figure 6 shows clearly the trend. As our problem size grows, so does the speedup improve, until at the calculation of the 45^{th} Fibonacci number, the speedup curve is close to being linear.

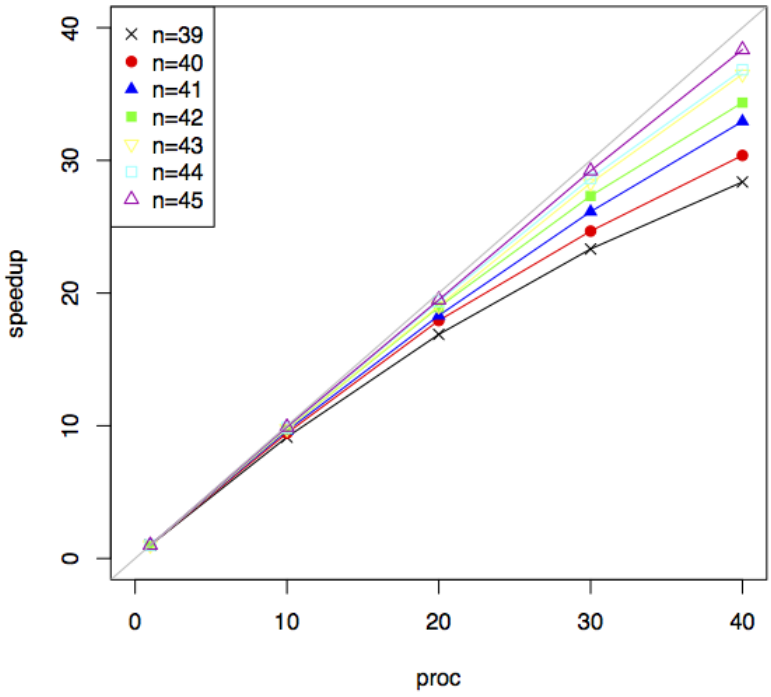


Figure 6: Speedup plot showing speedup curves at different problem sizes.

Note
The `prun` and `pplot` tools have many more features than those demonstrated here. For details, see the documentation provided with the tools in the file named `README.md`.

Noise in experiments
The run time that a given parallel program takes to solve the same problem can vary noticeably because of certain effects that are not under our control, such as OS scheduling, cache effects, paging, etc. We can consider such noise in our experiments random noise. Noise can be a problem for us because noise can lead us to make incorrect conclusions when, say, comparing the performance of two algorithms that perform roughly the same. To deal with randomness, we can perform multiple runs for each data point that we want to measure and consider the mean over these runs. The `prun` tool enables taking multiple runs via the `-runs` argument. Moreover, the `pplot` tool by default shows mean values for any given set of runs and optionally shows error bars. The documentation for these tools gives more detail on how to use the statistics-related features.

6.10 Superlinear speedup

Suppose that, on our 40-processor machine, the speedup that we observe is larger than 40x. It might sound improbable or even impossible. But it can happen. Ordinary circumstances should preclude such a **superlinear speedup**, because, after all, we have only forty processors helping to speed up the computation. Superlinear speedups often indicate that the sequential baseline program is suboptimal. This situation is easy to check: just compare its run time with that of the sequential elision. If the sequential elision is faster, then the baseline is suboptimal. Other factors can cause superlinear speedup: sometimes parallel programs running on multiple processors with private caches benefit from the larger cache capacity. These issues are, however, outside the scope of this course. As a rule of thumb, superlinear speedups should be regarded with suspicion and the cause should be investigated.

6.11 Visualize processor utilization (optional material)

The 29x speedup that we just calculated for our Fibonacci benchmark was a little dissapointing, and the 86% processor utilization of the run left 14% utilization for improvement. We should be suspicious that, although seemingly large, the problem size that we chose, that is, $n = 39$, was probably a little too small to yield enough work to keep all the processors well fed. To put this hunch to the test, let us examine the utilization of the processors in our system. We need to first build a binary that collects and outputs logging data.

```
$ make bench.log
```

We run the program with the new binary in the same fashion as before.

```
$ bench.log -bench fib -proc 40 -n 39
```

The output looks something like the following.

```
exectime 0.019
launch_duration 0.019
utilization     0.8639
thread_send     205
thread_exec     4258
thread_alloc    2838
utilization 0.8639
result 63245986
```

We need to explain what the new fields mean.

- The `thread_send` field tells us that 233 threads were exchaged between processors for the purpose of load balancing;
- the `thread_exec` field that 5179 threads were executed by the scheduler;
- the `thread_alloc` field that 3452 threads were freshly allocated.

Each of these fields can be useful for tracking down inefficiencies. The number of freshly allocated threads can be a strong indicator because in C++ thread allocation costs can sometimes add up to a significant cost. In the present case, however, none of the new values shown above are highly suspicious, considering that there are all at most in the thousands.

Since we have not yet found the problem, let us look at the visualization of the processor utilization using our `pview` tool. To get the necessary logging data, we need to run our program again, this time passing the argument `--pview`.

```
$ bench.log -bench fib -n 39 -proc 40 --pview
```

When the run completes, a binary log file named `LOG_BIN` should be generated in the current directory. Every time we run with `--pview` this binary file is overwritten. To see the visualization of the log data, we call the visualizer tool from the same directory.

```
$ pview
```

The output we see on our 40-processor machine is shown in Figure 7. The window shows one bar per processor. Time goes from left to right. Idle time is represented by red and time spent busy with work by grey. From the visualization, we can see that most of the time, particularly in the middle, all of the processors keep busy. However, there is a lot of idle time in the beginning and end of the run. This pattern suggests that there just is not enough parallelism in the early and late stages of our Fibonacci computation.

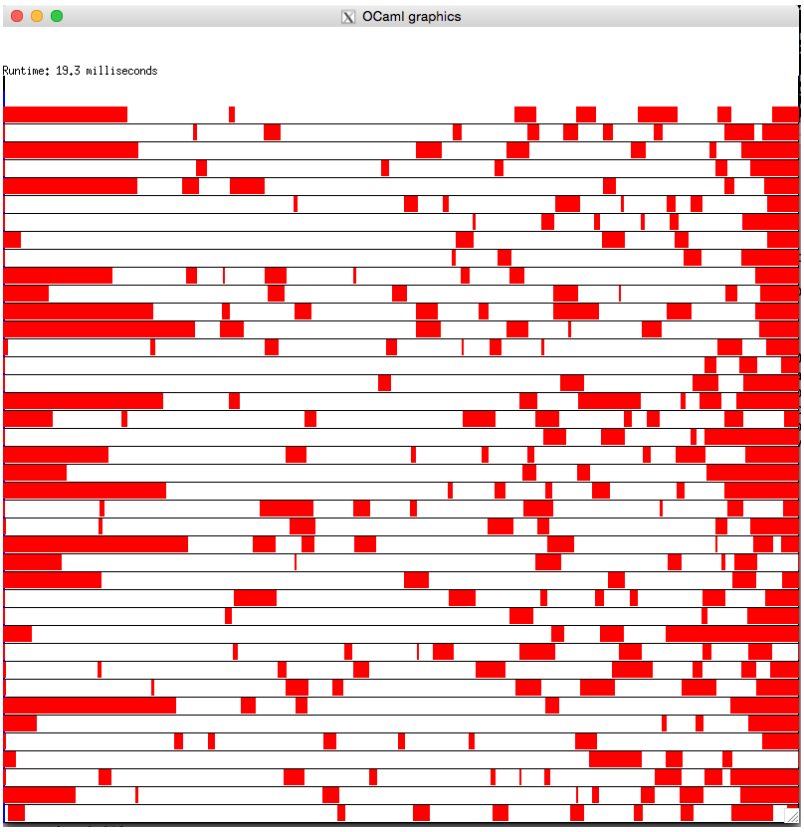


Figure 7: Utilization plot for computation of 39th Fibonacci number.

6.12 Strong versus weak scaling (optional material)

We are pretty sure that or Fibonacci program is not scaling as well is it could. But poor scaling on one particular input for n does not necessarily mean there is a problem with the scalability our parallel Fibonacci program in general. What is important is to know more precisely what it is that we want our Fibonacci program to achieve. To this end, let us consider a distinction that is important in high-performance computing: the distinction between strong and weak scaling. So far, we have been studying the strong-scaling profile of the computation of the 39^{th} Fibonacci number. In general, strong scaling concerns how the run time varies with the number of processors for a fixed problem size. Sometimes strong scaling is either too ambitious, owing to hardware limitations, or not necessary, because the programmer is happy to live with a looser notion of scaling, namely weak scaling. In weak scaling, the programmer considers a fixed-size problem per processor. We are going to consider something similar to weak scaling. In Figure 8, we have a plot showing how processor utilization varies with the input size. The situation dramatically improves from 12% idle time for the 39^{th} Fibonacci number down to 5% idle time for the 41^{st} and finally to 1% for the 45^{th}. At just 1% idle time, the utilization is excellent.

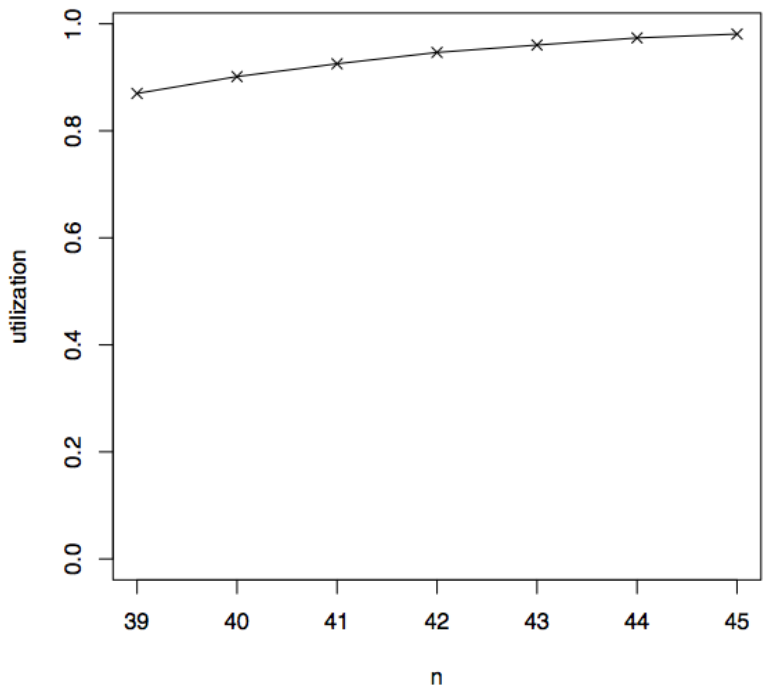


Figure 8: How processor utilization of Fibonacci computation varies with input size.

The scenario that we just observed is typical of multicore systems. For computations that perform relatively little work, such as the computation of the 39^{th} Fibonacci number, properties that are specific to the hardware, OS, and PASL load-balancing algorithm can noticeably limit processor utilization. For computations that perform lots of highly parallel work, such limitations are barely noticeable, because processors spend most of their time performing useful work. Let us return to the largest Fibonacci instance that we considered, namely the computation of the 45^{th} Fibonacci number, and consider its utilization plot.

```
$ bench.log -bench fib -n 45 -proc 40 --pview
$ pview
```

The utilization plot is shown in Figure 9. Compared the to utilization plot we saw in Figure 7, the red regions are much less prominent overall and the idle regions at the beginning and end are barely noticeable.

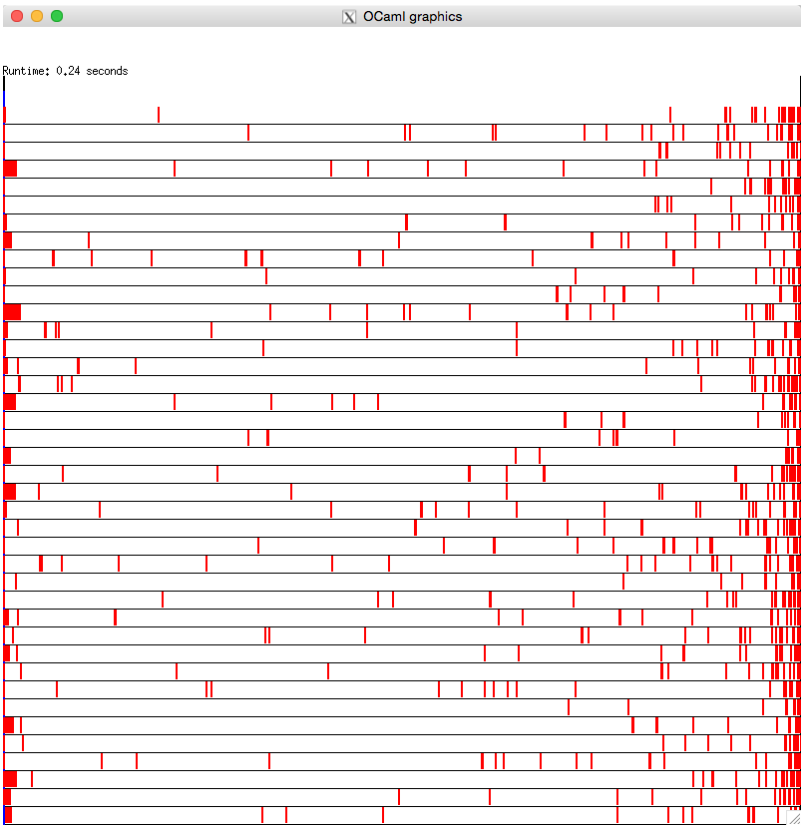


Figure 9: Utilization plot for computation of 45th Fibonacci number.

6.13 Summary of Lecture 3

We have seen in this lab how to build, run, and evaluate our parallel programs. Concepts that we have seen, such as speedup curves, are going to be useful for evaluating the scalability of our future solutions. Strong scaling is the gold standard for a parallel implementation. But as we have seen, weak scaling is a more realistic target in most cases.

7 Lecture 4: Work efficiency

In most cases, a parallel algorithm which solves a given problem performs more work than the fastest sequential algorithm that solves the same problem. This extra work deserves careful consideration because the amount of extra work imposes an upper limit on the speedup that can be achieved by the parallel algorithm for a given input and machine. As we are going to see, in order to implement observably efficient parallel algorithms for multicore platforms, we need to understand where the extra work comes from and, when possible, how to tame it.

When assessing a parallel algorithm, we are going to consider its "work efficiency", which is a measure of the extra work performed by the parallel algorithm. For precision, let us define two types of work efficiency: asymptotic and observed. The former relates to the asymptotic performance of a parallel algorithm relative to the fastest sequential algorithm. The latter relates to running time of a parallel algorithm relative to that of the fastest sequential algorithm.

Definition: asymptotic work efficiency

An algorithm is ***asymptotically work efficient*** if the work cost of the algorithm is the same as the work cost of the fastest known sequential program.

Example 7.1 Asymptotic work efficiency

- A parallel algorithm that sorts n keys in span $O(\log^3 n)$ and work $O(n \log n)$ is asymptotically work efficient because the work cost is as fast as the best known sequential comparison-based sorting algorithm. However, a parallel sorting algorithm that takes $O(n^2)$ work is not asymptotically work efficient.
- The parallel array increment algorithm that we consider in the first chapter is asymptotically work efficient, because it performs linear work, which is optimal (any sequential algorithm must perform at least linear work).

When designing a good parallel algorithm, in addition to finding an algorithm with low span, we also need to take care to design an asymptotically work-efficient algorithm.

Question

Why is asymptotic work efficiency important?

To see why asymptotic work efficiency is important consider a parallel algorithm that perform $\Theta(n \log n)$ work instead of the sequential algorithm that perform $\Theta(n)$ work. If $n = 2^{20}$ (a million), a very small input size in many cases, the parallel algorithm is $20\times$ slower just to being with. That means: even if we can obtain perfect speedups, we would need 20 processors just to catch up with the sequential algorithm.

In addition to running time, there is one more concern: energy consumption. It turns out, energy consumption of an execution is strongly determined by the total work performed. Thus, in our example, the energy consumption of the parallel algorithm will likely be $20\times$ more than that of the sequential. This can be a disaster: imagine a data center, which can already consume massive amounts of energy, requiring $20\times$ as much; or that suddenly your laptop or phone battery draining that much faster.

Parallel algorithms incur additional practical overheads due to friction in scheduling. Even if an algorithm is asymptotically efficient, it might incur such large practical overheads that can easily undo the benefits of parallelization. For example, it is common for a parallel program to incur a $100\times$ overhead compared to sequential execution if friction is not controlled. To assess the practical effectiveness of a parallel algorithm, we define observed work efficiency, parameterized a value r.

Definition: r-observed work efficiency

A parallel algorithm that runs in time T_1 on a single processor is r-**observed work efficient** if $r = \frac{T_1}{T_{seq}}$, where T_{seq} is the time taken by the fastest known sequential algorithm.

Example 7.2 Observed work efficiency

- A parallel algorithm that runs $10\times$ slower on a single processor that the fastest sequential algorithm is 10-observed work efficient. We consider such algorithms unacceptable, as they are too slow and wasteful.
- A parallel algorithm that runs $1.1\times - 1.2\times$. slower on a single processor that the fastest sequential algorithm is 1.2-observed work efficient. We consider such algorithms to be acceptable.

Example 7.3 Observed work efficiency of parallel increment

To obtain this measure, we first run the baseline version of our parallel-increment algorithm.

```
$ bench.baseline -bench map_incr -n 100000000
exectime 0.884
utilization 1.0000
result 2
```

We then run the untuned version of the parallel algorithm, which is the same exact code that we presented in the first chapter for `map_incr_rec`. (We activate this code by using the special `opt fp` file extension. This special file extension forces parallelism to be exposed all the way down to the base cases. Later, we will see how to use this special binary mode for other purposes.)

```
$ make bench.optfp
$ bench.optfp -bench map_incr -n 100000000
exectime 45.967
utilization 1.0000
result 2
```

Our untuned algorithm is approximately $60\times$-observed work efficient. That is terrible! Such poor observed work efficiency suggests that we can expect no speedup on a machine with, say, 40 processors.

For practicioners, observed work efficiency is a major concern. First, the whole effort of parallel computing is wasted if the parallel program is consistently slower or not much faster than the best-known sequential program. In other words, in parallel programming, constant factors matter. After all, the number of processors in a modern machine is on the order of tens. If our parallel program is twice slower on a single processor than the sequential program, we expect that the best speedup the parallel program can achieve is a measly $\frac{P}{2}$ with P processors! Furthermore, observed work efficiency matters because energy/power consumption is increasingly an important factor. More work means more power consumed by the program.

Based on these discussions, we define a ***good parallel algorithm*** as follows.

Definition: good parallel algorithm

We say that a parallel algorithm is ***good*** if it has the following three characteristics:

1. It is asymptotically work efficient
2. It is observably work efficient
3. It has low span.

7.1 Tuning for work efficiency

Now that we have established some terminology, we can start considering an important question.

Question

Given that it is common for a parallel algorithm to be asymptotically and/or observably work inefficient, is the whole enterprise of parallel algorithms doomed?

Fortunately, the answer is no. There is a key tool in our disposal that can help us to reduce the impact of work efficiency: tuning. While tuning might sound as a horrible hack (it can be), as we will see, it can be done in a principled fashion. To see what we can do consider as an example a parallel algorithm that performs linear work and logarithmic span leads to average parallelism in the orders of $10,000$ with the small input size of one million. For such a small problem size, we would not need to employ tens of thousands of processors, just mere 10's would suffice.

Question

Given that many parallel algorithms create large, even perhaps, excessive, amounts of parallelism, do you have any ideas what we can do to improve work efficiency?

The idea simply is to reduce parallelism in the algorithm. Since the causes of both asymptotic and observed work inefficiency is parallelism, we should be able to improve work efficiency by reducing parallelism.

Question

Any ideas about how we can improve work efficiency of parallel algorithms by reducing parallelism?

In many parallel algorithms such as the algorithms based on divide-and-conquer, there is a simple way to achieve this goal: switch from parallel to sequential algorithm when the problem size falls below a certain threshold. This technique is sometimes called ***coarsening***.

But which code can we switch to: one idea is to simply switch to the sequential elision, which we always have available in PASL. If, however, the parallel algorithm is asymptotically work inefficient, this can be somewhat ineffective. In such cases, we can specify a separate sequential algorithm for small instances.

Example 7.4 Tuning our parallel array-increment function

We can apply coarsening our parallel code by switching to the sequential algorithm when the input falls below an established threshold.

```
long threshold = Some Number;

void map_incr_rec(const long* source, long* dest, long lo, long hi) {
  long n = hi - lo;
  if (n <= threshold) {
    for (long i = lo; i < hi; i++)
      dest[i] = source[i] + 1;
  } else {
    long mid = (lo + hi) / 2;
    fork2([&] {
      map_incr_rec(source, dest, lo, mid);
    }, [&] {
      map_incr_rec(source, dest, mid, hi);
    });
  }
}
```

Note

Even in sequential algorithms, it is not uncommon to revert to a different algorithm for small instances of the problem. For example, it is well known that insertion sort in very fast for inputs with 10-30 keys. Thus many carefully crafted sorting algorithm revert to insertion sort when the input size falls within that range.

Example 7.5 Observed work efficiency of tuned array increment

After some tuning, `map_incr` program is highly work efficient. In fact, there is barely a difference between the sequential program and the parallel one. The tuning is actually done automatically here by using a technique we describe in the section on automatic granularity control, the next section.

```
$ bench.baseline -bench map_incr -n 100000000
exectime 0.884
utilization 1.0000
result 2
$ bench.opt -bench map_incr -n 100000000
exectime 0.895
utilization 1.0000
result 2
```

In this case, we have $r = \frac{T_1}{T_{seq}} = \frac{0.895}{0.884} = 1.012$ observed work efficiency. Our parallel program on a single processor is one percent slower than the sequential baseline. Such work efficiency is excellent.

7.2 Determining the threshold

Now that we understand the basic idea behind tuning to achieve work efficiency, the question that remains is how to choose the threshold.

Question

Any ideas about how we can determine the right threshold?

Determining the threshold turns out to be difficult. On the one hand, if the threshold is set too small, the observed work efficiency would suffer because of the correspondingly small reduction in parallelism and thus larger overheads of scheduler friction. On the other, if the threshold is too high, then we can lose the benefits of parallelization because of over-sequentialization. Now, there is a middle ground of satisfactory values but the precise position of the "middle ground" depends on both algorithmic and machine-specific factors. To get a more visceral sense of the problem, let us perform a few runs of our array-increment benchmark using different values for the threshold.

Exercise

1. Repeat the speedup experiments above and observe the speedup obtained.
2. What happens when you change the threshold to different values such as 100, 1000, 10000?

As we describe in the next section, it is sometimes possible to determine the threshold completely automatically.

7.3 Automatic granularity control

The problem of finding a good middle ground on average for the threshold size is the granularity-control problem. Fortunately, we have at our disposal a technique to mitigate this problem: automatic granularity control.

7.3.1 Complexity functions

Our automatic granularity-control technique requires assistance from the application programmer: for each parallel region of the program, the programmer must annotate the region with a **complexity function**, which is simply a C++ function that returns the asymptotic work cost associated with running the associated region of code.

Example 7.6 Complexity function of `map_incr_rec`

Recall that the work cost taken by an application of our `map_incr_rec` function to a given range $[lo, hi)$ of a given array has work cost $cn = c(hi - lo)$ for some constant c. As such, the following lambda expression is one valid complexity function for our parallel `map_incr` program.

```
auto map_incr_rec_complexity_fct = [&] (long lo, long hi) {
  return hi - lo;
};
```

In general, the value returned by the complexity function need only be precise with respect to the asymptotic complexity class of the associated computation. As such, the following lambda expression is another valid complexity function for our parallel `map_incr`.

```
const long k = 123;
auto map_incr_rec_complexity_fct2 = [&] (long lo, long hi) {
  return k * (hi - lo);
};
```

However, the first complexity function is preferable because the first one is simpler and the second one delivers no useful information relative to the first one.

More generally, suppose that we know that a given algorithm takes time $W = n + \log n$. Although it would be fine to assign to this algorithm exactly W, we could just as well assign to the algorithm the cost n, because the second term is dominated by the first. In other words, complexity functions are like bit-oh notation: we care only about the most significant term in any given formula and we can disregard constant factors.

7.3.2 Controlled statements

In PASL, a ***controlled statement***, or `cstmt`, is an annotation in the program text that activates automatic granularity control for a specified region of code. In particular, a controlled statement behaves as a C++ statement that has the special ability to choose on the fly whether or not the computation rooted at the body of the statement spawns parallel threads. To support such automatic granularity control PASL uses a prediction algorithm to map the asymptotic work cost (as returned by the complexity function) to actual processor cycles. When the predicted processor cycles of a particular instance of the controlled statement falls below a threshold (determined automatically for the specific machine), then that instance is sequentialized, by turning off the ability to spawn parallel threads for the execution of that instance. If the predicted processor cycle count is higher than the threshold, then the statement instance is executed in parallel.

In other words, the reader can think of a controlled statement as a statement that executes in parallel when the benefits of parallel execution far outweighs its cost and that executes sequentially in a way similar to the sequential elision of the body of the controlled statement would if the cost of parallelism exceeds its benefits. We note that while the sequential execution is similar to the sequential execution, it is not exactly the same because every call to `fork2` must check whether it should create parallel threads or run sequentially. Thus the execution may differ from the sequential elision in terms of performance but not in terms of behavior or semantics.

Example 7.7 Array-increment function with automatic granularity control

The code below uses a controlled statement to automatically select, at run time, the threshold size for our parallel array-increment function.

```
controller_type map_incr_rec_contr("map_incr_rec");

void map_incr_rec(const long* source, long* dest, long lo, long hi) {
  long n = hi - lo;
  cstmt(map_incr_rec_contr, [&] { return hi - lo; }, [&] {
    if (n == 0) {
      // do nothing
    } else if (n == 1) {
      dest[lo] = source[lo] + 1;
    } else {
      long mid = (lo + hi) / 2;
      fork2([&] {
        map_incr_rec(source, dest, lo, mid);
      }, [&] {
        map_incr_rec(source, dest, mid, hi);
      });
    }
  });
}
```

The controlled statement takes three arguments, whose requirements are specified below, and returns nothing (i.e., `void`). The effectiveness of the granularity controller may be compromised if any of the requirements are not met.

- The first argument is a reference to the controller object. The controller object is used by the controlled statement to collect profiling data from the program as the program runs. Every controller object is initialized with a string label (in the code above `"map_incr_rec"`). The label must be unique to the particular controller. Moreover, the controller must be declared as a global variable.
- The second argument is the complexity function. The type of the return value should be `long`.
- The third argument is the body of the controlled statement. The return type of the controlled statement should be `void`.

When the controlled statement chooses sequential evaluation for its body the effect is similar to the effect where in the code above the input size falls below the threshold size: the body and the recursion tree rooted there is sequentialized. When the controlled statement chooses parallel evaluation, the calls to `fork2()` create parallel threads.

7.3.3 Granularity control with alternative sequential bodies

It is not unusual for a divide-and-conquer algorithm to switch to a different algorithm at the leaves of its recursion tree. For example, sorting algorithms, such as quicksort, may switch to insertion sort at small problem sizes. In the same way, it is not unusual for parallel algorithms to switch to different sequential algorithms for handling small problem sizes. Such switching can be beneficial especially when the parallel algorithm is not asymptotically work efficient.

To provide such algorithmic switching, PASL provides an alternative form of controlled statement that accepts a fourth argument: the ***alternative sequential body***. This alternative form of controlled statement behaves essentially the same way as the original described above, with the exception that when PASL run time decides to sequentialize a particular instance of the controlled statement, it falls through to the provided alternative sequential body instead of the "sequential elision."

Example 7.8 Array-increment function with automatic granularity control and sequential body

```
controller_type map_incr_rec_contr("map_incr_rec");

void map_incr_rec(const long* source, long* dest, long lo, long hi) {
  long n = hi - lo;
  cstmt(map_incr_rec_contr, [&] { return hi - lo; }, [&] {
    if (n == 0) {
      // do nothing
    } else if (n == 1) {
      dest[lo] = source[lo] + 1;
    } else {
      long mid = (lo + hi) / 2;
      fork2([&] {
        map_incr_rec(source, dest, lo, mid);
      }, [&] {
        map_incr_rec(source, dest, mid, hi);
      });
    }
  }, [&] {
    for (long i = lo; i < hi; i++)
      dest[i] = source[i] + 1;
  });
}
```

Even though the parallel and sequential array-increment algorithms are practically identical, excepting the calls to `fork2()`, there is still an advantage to using the alternative sequential body: the sequential code does not pay for certain overheads that are imposed by calls to `fork2()`. In specific, every call to `fork2()` must perform a conditional branch to check whether or not the context of the `fork2()` call is parallel or sequential. Because the cost of these conditional branches adds up, the version with the sequential body is going to be more work efficient.

Recommended style for programming with controlled statements
In general, we recommend that the code of the parallel body be written so as to be completely self contained, at least in the sense that the parallel body code contains the logic that is necessary to handle recursion all the way down to the base cases. The code for `map_incr_rec` honors this style by the fact that the parallel body handles the cases where `n` is zero or one (base cases) or is greater than one (recursive case). Put differently, it should be the case that, if the parallelism-specific annotations (including the alternative sequential body) are erased, the resulting program is a correct program.
We recommend this style because such parallel codes can be debugged, verified, and tuned, in isolation, without relying on alternative sequential codes.

7.3.4 Controlled parallel-for loops

Let us add one more component to our granularity-control toolkit: namely, the **parallel-for loop**. By using this loop form, we can avoid having to explicitly express log trees via recursion over and over again. For example, the following function performs the same computation as the example function we defined in the first lecture. Only, this function is much more compact and readable. Moreover, this code takes advantage of our automatic granularity control.

```
loop_controller_type map_incr_contr("map_incr");

void map_incr(const long* source, long* dest, long n) {
  parallel_for(map_incr_contr, (long)0, n, [&] (long i) {
    dest[i] = source[i] + 1;
  });
}
```

Underneath, the parallel-for loop uses a divide-and-conquer routine whose structure is similar to the structure of the divide-and-conquer routine of our `map_incr_rec`. Because the parallel-for loop generates the log-height recursion tree, the `map_incr` routine just above has the same span as the `map_incr` routine that we defined earlier: $\log n$, where n is the size of the input array.

Notice that the code above specifies no complexity function. The reason is that this particular instance of the parallel-for loop implicitly defines a complexity function. The implicit complexity function reports a linear-time cost for any given range of the iteration space of the loop. In other words, the implicit complexity function assumes that per iteration the body of the loop performs a constant amount of work. Of course, this assumption does not hold in general. If we want to specify explicitly the complexity function, we can use the form shown in the example below. The complexity function is passed to the parallel-for loop as the fourth argument. The complexity function takes as argument the range `[lo, hi)`. In this case, the complexity is linear in the number of iterations. The function simply returns the number of iterations as the complexity.

```
loop_controller_type map_incr_contr("map_incr");

void map_incr(const long* source, long* dest, long n) {
  auto linear_complexity_fct = [] (long lo, long hi) {
    return hi-lo;
  };
  parallel_for(map_incr_contr, linear_complexity_fct, (long)0, n, [&] (long i) {
    dest[i] = source[i] + 1;
  });
}
```

The following code snippet shows a more interesting case for the complexity function. In this case, we are performing a multiplication of a dense matrix by a dense vector. The outer loop iterates over the rows of the matrix. The complexity function in this case gives to each of these row-wise iterations a cost in proportion to the number of scalars in each column.

```
loop_controller_type dmdvmult_contr("dmdvmult");

// mtx: nxn dense matrix, vec: length n dense vector
// dest: length n dense vector
void dmdvmult(double* mtx, double* vec, double* dest, long n) {
  auto compl_fct = [&] (long lo, long hi) {
    return (hi-lo)*n;
  };
  parallel_for(dmdvmult_contr, compl_fct, (long)0, n, [&] (long i) {
    ddotprod(mtx, v, dest, i);
  });
  return dest;
}
```

Example 7.9 Speedup for matrix multiply

Matrix multiplication has been widely used as an example for parallel computing since the early days of the field. There are good reasons for this. First, matrix multiplication is a key operation that can be used to solve many interesting problems. Second, it is an expansive computation that is nearly cubic in the size of the input---it can thus can become very expensive even with modest inputs.

Fortunately, matrix multiplication can be parallelized relatively easily as shown above. The figure below shows the speedup for a sample run of this code. Observe that the speedup is rather good, achieving nearly excellent utilization.

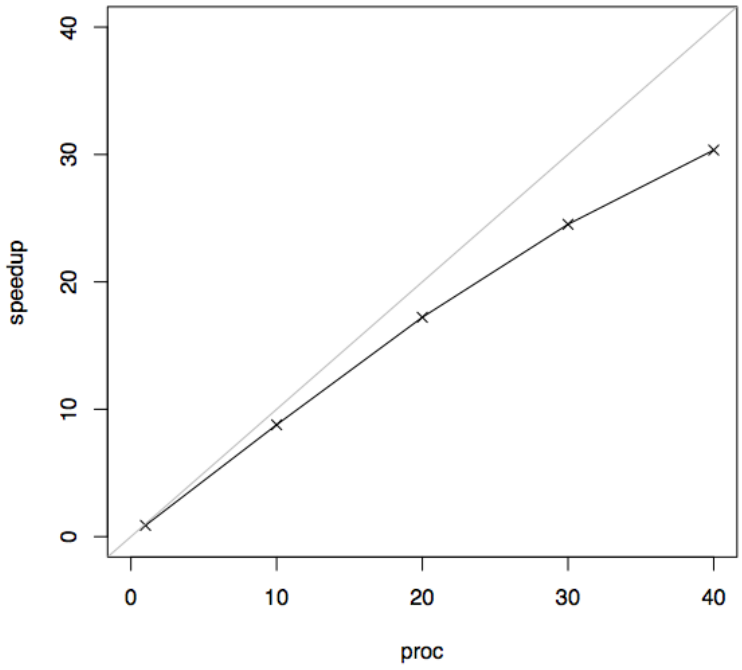


Figure 10: Speedup plot for matrix multiplication for 25000×25000 matrices.

While parallel matrix multiplication delivers excellent speedups, this is not common for many other algorithms on modern multicore machines where many computations can quickly become limited by the availability of bandwidth. Matrix multiplication does not suffer as much from the memory-bandwidth limitations because it performs significant work per memory operation: in touches $\Theta(n^2)$ memory cells, while performing nearly $\Theta(n^3)$ work.

8 Project 1

In this project, you will develop a parallel **reduce** function that performs computations, such as aggregation, efficiently in parallel. Before you do the project though, please the instructions below to warm up to the software system that you will be using.

For all the code that you write: all code should go in the header file named `exercises.hpp` that is in your `minicourse` folder.

We ask that you email your code and some results (speedup curve) to us by Sunday 9pm. Feel free to compete and collaborate with each other in a friendly manner. This is a competition but it is us against you. We will be doing the project and try to get the best speedups just as you are doing.

You can share your results on this web site by for example team name and maximum speedup on (don't forget to mention the total number of cores)

1. Repeat today's lectures 3 and 4 (the labs) on your own, making sure that you can produce results similar to those we showed.
2. Repeat today's lectures 3 and 4 (the labs) on your own, using the array increment function. See instructions below for testing and running your code.
3. Start out by writing a simple iterative algorithm for summing up the elements in an array. Your algorithm should be similar to the sequential increment algorithm discussed earlier.
4. Analyze the work and span of the algorithm. Is the algorithm a good parallel algorithm?

5. Now, determine if there is parallelism in this problem. Can you identify pieces of computation that are independent?
6. Based on the intuition above, design an divide-and-conquer algorithm for summing up the elements of an array.
7. Can you parallelize your divide and conquer algorithm? Your parallel algorithm should be very similar to the parallel increment algorithm.
8. Can you analyze the work and span of your algorithm?
9. Mentally repeat the same process above for computing the minimum or the maximum element in an array. Do you see a difference? What property of the "+" and "max" operations are you exploiting in designing your parallel algorithm?
10. Implement your parallel sum algorithm and derive a speepdup curve with a large input, for example in the range 100 million to 250 million.
11. Optimize your algorithm by tuning it, for example, by using automatic granularity control to deliver the best speedup.
12. Generate your best speedup curve with the maximum number of cores available on your machine. Use random data generation as automatically supported by the benchmarking tools. Use either cadmium (48 cores) or teraram (40 cores). You will need to coordinate with other teams to make sure that they are not using the machine at the same time as you are making measuruments for your speedup curve.
13. This might require some research: can you generalize your parallel array sum to a **higher order reduce** function that takes another function that defines a binary operator such as, "+", "max", and applies it to the array to aggreate the values using this operator.
14. Can you implement array sum, and array max in terms of your higher order reduce function?
15. Email your code (as a single file) and your best speedup curve along with a one paragraph description of your results by 10pm on Sunday. Our email addresses are umut@cs.cmu.edu, and mrainey@inria.fr.

8.1 How to test your solutions

Once you have completed a solution, you are ready to start testing and debugging. The first step to take is to write a few sample applications of your functions. Follow the directions provided in the file located at `minicourse/examples.hpp`. The place to start is the comment:

```
// Add an option for your example code here:
```

After you get your sample applications to generate the correct output, you are ready for the next step: unit testing. The next thing to do is build the binary for our unit testing program. Notice that we are building a debugging binary. It is important to use the debugging binary at this stage, because the debugging binary enables useful features, such as array-bounds checks.

```
$ make check.dbg
```

Now, we are ready to run some unit tests. There is one test for each exercise that you are assigned. Let us start by testing `map_incr_ex`.

```
$ check.dbg -check map_incr_ex
```

If your solution is correct, the following message should appear. Otherwise, the program will report which input caused the test to fail.

```
* Checking that solution to map_incr exercise is correct...
OK, passed 500 tests.

exectime 0.013
```

It is possible to get a larger coverage by running more tests.

```
$ check.dbg -check map_incr_ex -nb_tests 5000
```

The names of the checks are the following.

- `map_incr_ex`
- `max_ex`
- `sum_ex`
- `reduce_ex`

If your program is crashing, it may be helpful to use `valgrind` to investigate the cause of the crash. This program runs our unit test binary in a special mode that monitors for common errors, such as reading from uninitialized memory. Valgrind currently supports only single-processor runs of PASL programs.

```
$ valgrind check.dbg -check map_incr_ex
```

If you need to debug multi-processor runs, then `gdb` is one option.

```
$ gdb --args check.dbg -check map_incr_ex -proc 40
```

Once you have a correct program, you should proceed to benchmarking.

8.2 How to benchmark your solutions

If you have note already, build the benchmarking binary.

```
$ make bench.opt
```

To benchmark our `map_incr` solution, we use the following command. The `-n` argument specifies that the input array being passed to `map_incr` has 10^7 items.

```
$ bench.opt -bench map_incr_ex -n 10000000
```

You are now ready to generate speedup curves and other plots as needed. All the instructions that you need appear in the previous chapters.

9 Lecture 5: Simple Parallel Arrays

Arrays are a fundamental data structure in sequential and parallel computing. In fact, when computing sequentially, arrays can often be replaced by linked lists, because linked lists are more flexible. Linked lists are deadly for parallelism, however, because linked lists do not leave room for much parallelism at all, making arrays all the more important in parallel computing. For example, from this point on, every program that we consider in this course involves arrays.

Unfortunately, it is difficult to find a good treatment of parallel arrays in C++: the various array implementations provided by C++ have been designed primarily for sequential computing. Each one has various pitfalls for parallel use.

Example 9.1 C++ arrays

By default, C++ arrays that are created by the `new[]` operator are initialized sequentially. Therefore, the work and span cost of the call `new[n]` is `n`. But we have seen already that we can initialize an array more efficiently by using parallelism to achieve logarithmic span in the number of items.

Example 9.2 STL vectors

The "vector" data structure that is provided by the Standard Template Library has a similar issue. The STL vector implements a dynamically resizable array that provides push, pop, and indexing operations. The push and pop operations take amortized linear time and the indexing operation constant time. The STL vector also provides the method `resize(n)` which changes the size of the array to be `n`. The resize operation takes, in the worst case, linear work and span in proportion to the new size, `n`. In other words, the resize function uses a sequential algorithm to fill the cells in the vector. The resize operation is therefore inefficient for the same reason as for the default C++ arrays. You might be tempted to think that there is an efficient parallel algorithm to initialize an STL vector, but unfortunately, the sequential bottleneck imposed by the resize operation defeats efficient parallel solutions.

Such sequential loops that exist behind the wall of abstraction of a data structure can harm parallel performance by introducing implicit sequential dependencies. Moreover, finding the source of such sequential bottlenecks can be time consuming, because they are hidden behind the abstraction boundary of the native array abstraction that is provided by the programming language.

We can easily avoid such pitfalls by carefully designing our own array data structure. Because array implementations are quite subtle, we consider our own implementation of parallel arrays, which makes explicit the cost of array operation, allowing us to control them quite carefully. Specifically, we disallow implicit copy operations on arrays, because copy operations can greatly harm performance (their asymptotic work cost is linear).

9.1 Interface and cost model

The key components of our array data structure, `sparray`, are shown by the code snippet below. For simplicity, in this course, our arrays can store 64-bit words only; in particular, our arrays are monomorphic and fixed to values of type `long`.

```
using value_type = long;

class sparray {
public:

  // n: size to give to the array; by default 0
  sparray(unsigned long n = 0);

  // constructor from list
  sparray(std::initializer_list<value_type> xs);

  // indexing operator
  value_type& operator[](unsigned long i);

  // size of the array
  unsigned long size() const;

};
```

The class `sparray` provides two constructors. The first one takes in the size of the array (set to 0 by default) and allocates an unitialized array of the specified size (`nullptr` if size is 0). The second constructor takes in a list specified by curly braces and allocates an array with the same size. Since the argument to this constructor must be specified explicitly in the program, its size it constant by definition.

Note

Generalizing our parallel arrays to store values of any given type can be achieved by use of C++ templates. However, it is not entirely straightforward to implement such polymorphic parallel arrays in an efficient manner. Because this issue is outside the scope of this course, we leave this feature for students to consider independently.

The cost model guaranteed by our implementation of parallel array is as follows:

- **Constructors/Allocation:** The work and span of simply allocating an array on the heap, without initialization, is constant. The second constructor performs initialization, of constant size, and thus also has constant work and span.

- **Array indexing:** Each array-indexing operation, that is the operation which accesses an individual cell, requires constant work and constant span.
- **Size operation:** The work and the span of accessing the size of the array is constant.
- **Destructors/Deallocation:** Not shown, the class includes a destructor that frees the array. Combined with the "move assignment operator" that C++ allows us to define, destructors can be used to deallocate arrays when they are out of scope. The destructor takes constant time because the contents of the array are just bits that do not need to be destructed individually.
- **Move assignment operator:** Not shown, the class includes a move-assignment operator that gets fired when an array is assigned to a variable. This operator moves the contents of the right-hand side of the assigned array into that of the left-hand side. This operation takes constant time.

Note

The constructors of our array class do not perform initializations that involve non-constant work. If desired, the programmer can write an initializer that performs linear work and logarithmic span (if the values used for initialization have non-constant time cost, these bounds may need to be scaled accordingly).

Example 9.3 Simple use of arrays

This program below shows a basic use `sparray`'s. The first line allocates and initializes the contents of the array to be three numbers. The second uses the familiar indexing operator to access the item at the second position in the array. The third line extracts the size of the array. The fourth line assigns to the second cell the value 5. The fifth prints the contents of the cell.

```
sparray xs = { 1, 2, 3 };
std::cout << "xs[1] = " << xs[1] << std::endl;
std::cout << "xs.size() = " << xs.size() << std::endl;
xs[2] = 5;
std::cout << "xs[2] = " << xs[2] << std::endl;
```

Output:

```
xs[1] = 2
xs.size() = 3
xs[2] = 5
```

9.2 Allocation and deallocation

Our array type supports allocation as follows.

Example 9.4 Allocation and deallocation

```
sparray zero_length = sparray();
sparray another_zero_length = sparray(0);
sparray yet_another_zero_length;
sparray length_five = sparray(5);    // contents uninitialized
std::cout << "|zero_length| = " << zero_length.size() << std::endl;
std::cout << "|another_zero_length| = " << another_zero_length.size() << std::endl;
std::cout << "|yet_another_zero_length| = " << yet_another_zero_length.size() << std::endl;
std::cout << "|length_five| = " << length_five.size() << std::endl;
```

Output:

```
|zero_length| = 0
|another_zero_length| = 0
|yet_another_zero_length| = 0
|length_five| = 5
```

Just after creation, the array contents consist of uninitialized bits. We use this convention because the programmer needs flexibility to decide the parallelization strategy to initialize the contents. Internally, the `sparray` class consists of a size field and a pointer to the first item in the array. The contents of the array are heap allocated (automatically) by constructor of the `sparray` class. Deallocation occurs when the array's destructor is called. The destructor can be called by the programmer or by run-time system (of C++) if an object storing the array is destructed. Since C++ destructs (stack allocated) variables that go out of scope when a function returns, we can combine the stack discipline with heap-allocated arrays to manage the deallocation of arrays mostly automatically. We give several examples of this automatic deallocation scheme below.

Example 9.5 Automatic deallocation of arrays upon return

In the function below, the `sparray` object that is allocated on the frame of `foo` is deallocated just before `foo` returns, because the variable `xs` containing it goes out of scope.

```
void foo() {
  sparray xs = sparray(10);
  // array deallocated just before foo() returns
}
```

Example 9.6 Dangling pointers in arrays

Care must be taken when managing arrays, because nothing prevents the programmer from returning a dangling pointer.

```
value_type* foo() {
  sparray xs = sparray(10);
  ...
  // array deallocated just before foo() returns
  return &xs[0]
}

...

std::cout << "contents of deallocated memory: " << *foo() << std::endl;
```

Output:

```
contents of deallocated memory: .... (undefined)
```

It is safe, however, to take a pointer to a cell in the array, when the array itself is still in scope. For example, in the code below, the contents of the array are used strictly when the array is in scope.

```
void foo() {
  sparray xs = sparray(10);
  xs[0] = 34;
  bar(&xs[0]);
  ...
  // array deallocated just before foo() returns
}

void bar(value_type* p) {
  std::cout << "xs[0] = " << *p << std::endl;
}
```

Output:

```
xs[0] = 34
```

We are going to see that we can rely on cleaner conventions for passing to functions references on arrays.

9.3 Passing to and returning from functions

If you are familiar with C++ container libraries, such as STL, this aspect of our array implementation, namely the calling conventions, may be unfamiliar: our arrays cannot be passed by value. We forbid passing by value because passing by value implies creating a fresh copy for each array being passed to or returned by a function. Of course, sometimes we really need to copy an array. In this case, we choose to copy the array explicitly, so that it is obvious where in our code we are paying a linear-time cost for copying out the contents. We will return to the issue of copying later.

Example 9.7 Incorrect use of copy constructor

What then happens if the program tries to pass an array to a function? The program will be rejected by the compiler. The code below does **not** compile, because we have disabled the copy constructor of our `sparray` class.

```
value_type foo(sparray xs) {
  return xs[0];
}

void bar() {
  sparray xs = { 1, 2 };
  foo(xs);
}
```

Example 9.8 Correctly passing an array by reference

The following code does compile, because in this case we pass the array `xs` to `foo` by reference.

```
value_type foo(const sparray& xs) {
  return xs[0];
}

void bar() {
  sparray xs = { 1, 2 };
  foo(xs);
}
```

Returning an array is straightforward: we take advantage of a feature of modern C++11 which automatically detects when it is safe to move a structure by a constant-time pointer swap. Code of the following form is perfectly legal, even though we disabled the copy constructor of `sparray`, because the compiler is able to transfer ownership of the array to the caller of the function. Moreover, the transfer is guaranteed to take constant time — not linear like a copy would take. The return is fast because internally what is happening is only a couple words are being exchanged and nothing more is happening. Such "move on return" is achieved by the "move-assignment operator" of `sparray` class.

Example 9.9 Create and initialize an array (sequentially)

```
sparray fill_seq(long n, value_type x) {
  sparray tmp = sparray(n);
  for (long i = 0; i < n; i++)
    tmp[i] = x;
  return tmp;
}

void bar() {
  sparray xs = fill_seq(4, 1234);
  std::cout << "xs = " << xs << std::endl;
}
```

Output after calling `bar()`:

```
xs = { 1234, 1234, 1234, 1234 }
```

9.4 Ownership-passing semantics

Although it is perfectly fine to assign to an array variable the contents of a given array, what happens may be surprising to those who know the usual conventions of C++11 container libraries. Consider the following program.

Example 9.10 Ownership-passing semantics

```
sparray xs = fill_seq(4, 1234);
sparray ys = fill_seq(3, 333);
ys = std::move(xs);
std::cout << "xs = " << xs << std::endl;
std::cout << "ys = " << ys << std::endl;
```

The assignment from `xs` to `ys` simultaneously destroys the contents of `ys` (by calling its destructor, which nulls it out), namely the array `{ 333, 333, 333 }`, and moves the contents of `xs` to `ys. The result is the following.

```
xs = { }
ys = { 1234, 1234, 1234, 1234 }
```

The reason we use this semantics for assignment is that the assignment takes constant time. Later, we are going to see that we can efficiently copy items out of an array. But for reasons we already discussed, the copy operation is going to be explicit.

9.5 Lab exercise: duplicating items in parallel

The aim of this exercise is to combine our knowledge of parallelism and arrays. To this end, your assignment is to implement two functions. The first, namely `duplicate`, is to return a new array in which each item appearing in the given array `xs` appears twice.

```
sparray duplicate(const sparray& xs) {
  // fill in
}
```

For example:

```
sparray xs = { 1, 2, 3 };
std::cout << "xs = " << duplicate(xs) << std::endl;
```

Expected output:

```
xs = { 1, 1, 2, 2, 3, 3 }
```

The second function is a generalization of the first: the value returned by `ktimes` should be an array in which each item `x` that is in the given array `xs` is replaced by `k` duplicate items.

```
sparray ktimes(const sparray& xs, long k) {
  // fill in
}
```

For example:

```
sparray xs = { 5, 7 };
std::cout << "xs = " << ktimes(xs, 3) << std::endl;
```

Expected output:

```
xs = { 5, 5, 5, 7, 7, 7 }
```

Notice that the `k` parameter of `ktimes` is not bounded. Your solution to this problem should be highly parallel not only in the number of items in the input array, `xs`, but also in the duplication-degree parameter, `k`.

1. What is the work and span complexity of your solution?
2. Does your solution expose ample parallelism? How much, precisely?
3. What is the speedup do you observe in practice on various input sizes?

10 Lecture 6: Fundamental "data parallel" array operations

Advances in memory technology give us unprecedented access to large amounts of data. We have the power to organize, mine, and manipulate massive amounts of data with just our own personal computers. Along with huge memory, our machines provide massive processing power. However, in many cases, just to get close to harnessing the total processing power, we must take advantage of parallelism that exists among simultaneous operations on many small pieces of data. Data parallelism is a classic paradigm in programming that addresses this issue.

In this section, we take the approach to data parallelism that was pioneered by functional languages, such as Nesl and Data-Parallel Haskell, and later adopted by frameworks, such as Google's MapReduce. The characteristic of this approach is the emphasis on the use of combinators, such as map, reduce, scan, etc., that collectively provide a vocabulary for writing parallel algorithms. The other important characteristic is the emphasis on composition as the tool for taming the complexity of algorithm design. The goal of the following sections is to show the reader the advantages and pitfalls of this approach. By the final chapter, in which we study graph algorithms, the authors hope that the reader is convinced that the price they pay for learning the combinators and the composition rules pays for itself by bringing organization and clarity to the design of non-trivial algorithms.

10.1 Tabulation

A **tabulation** is a data-parallel operation which creates a new array of a given size and initializes the contents according to a given "generator function". In our implementation, the call `tabulate(g, n)` allocates an array of length `n` and assigns to each valid index in the array `i` the value returned by `g(i)`.

```
template <class Generator>
sparray tabulate(Generator g, long n);
```

Tabulations can be used to generate sequences according to a specified formula.

Example 10.1 Sequences of even numbers

```
sparray evens = tabulate([&] (long i) { return 2*i; }, 5);
std::cout << "evens = " << evens << std::endl;
```

Output:

```
evens = { 0, 2, 4, 6, 8 }
```

Copying an array can be expressed as a tabulation.

Example 10.2 Parallel array copy function using tabulation

```
sparray mycopy(const sparray& xs) {
  return tabulate([&] (long i) { return xs[i]; }, xs.size());
}
```

Exercise

Solve the `duplicate` and `ktimes` problems that were given in homework, this time using tabulations.

Solutions appear below.

Example 10.3 Solution to `duplicate` and `ktimes` exercises

```
sparray ktimes(const sparray& xs, long k) {
  long m = xs.size() * k;
  return tabulate([&] (long i) { return xs[i/k]; }, m);
}

sparray duplicate(const sparray& xs) {
  return ktimes(xs, 2);
}
```

The implementation of tabulate is a straightforward application of the parallel-for loop. What is interesting about this code is the work and span costs. By just looking at this code, can you tell what they are? If not, where do you get stuck?

```
loop_controller_type tabulate_contr("tabulate");

template <class Generator>
sparray tabulate(Generator g, long n) {
  sparray tmp = sparray(n);
  parallel_for(tabulate_contr, (long)0, n, [&] (long i) {
    tmp[i] = g(i);
  });
  return tmp;
}
```

The answer to "What is the work and span cost of the tabulation?" is: it depends. In particular, it depends on the work and span cost of the generator function that is passed to the tabulation. Let us first analyze for the simple case, where the generator function takes constant work (and hence, constant span). In this case, it should be clear that a tabulation should take work linear in the size of the array. The reason is that the only work performed by the body of the loop is performed by the constant-time generator function. Since the loop itself performs as many iterations as positions in the array, the work cost is indeed linear in the size of the array. The span cost of the tabulation is the sum of two quantitles: the span taken by the loop and the maximum value of the spans taken by the applications of the generator function. Recall that we saw before that the span cost of a parallel-for loop with n iterations is $\log n$. The maximum of the spans of the generator applications is a constant. Therefore, we can conclude that, in this case, the span cost is logarithmic in the size of the array.

The story is only a little more complicated when we generalize to consider non-constant time generator functions. Let $W(\mathrm{g}(i))$ denote the work performed by an application of the generator function to a specified value i. Similarly, let $S(\mathrm{g}(i))$ denote the span. Then the tabulation takes work

$$\sum_{i=0}^{n} W(\mathrm{g}(i))$$

and span

$$\log n + \max_{i=0}^{n} S(\mathrm{g}(i))$$

10.1.1 Advanced topic: higher-order granularity controllers

We just observed that each application of our tabulate operation can have different work and span cost depending on the selection of the generator function. Pause for a moment and consider how this observation could impact our granularity-control scheme. Consider, in particular, the way that the tabulate function uses its granularity-controller object, `tabulate_contr`. This one controller object is shared by every call site of `tabulate()`.

The problem is that all of the profiling data that the granularity controller collects at run time is lumped together, even though each generator function that is passed to the tabulate function can have completely different performance characteristics. The threshold that is best for one generator function is not necessarily good for another generator function. For this reason, there must be one distinct granularity-control object for each generator function that is passed to `tabulate`. For this reason, we refine our solution from the one above to the one below, which relies on C++ template programming to effectively key accesses to the granularity controller object by the type of the generator function.

```
template <class Generator>
class tabulate_controller {
public:
  static loop_controller_type contr;
};

template <class Generator>
loop_controller_type
  tabulate_controller<Generator>::contr("tabulate"+std::string(typeid(Generator).name()));

template <class Generator>
sparray tabulate(Generator g, long n) {
  sparray tmp = sparray(n);
  parallel_for(tabulate_controller<Generator>::contr, (long)0, n, [&] (long i) {
    tmp[i] = g(i);
  });
  return tmp;
}
```

To be more precise, the use of the template parameter in the class `tabulate_controller` ensures that each generator function in the program that is passed to `tabulate()` gets its own unique instance of the controller object. The rules of the template system regarding static class members that appear in templated classes ensure this behavior. Although it is not essential for our purposes to have a precise understanding of the template system, it is useful to know that the template system provides us with the exact mechanism that we need to properly separate granularity controllers of distinct instances of higher-order functions, such as tabulation.

10.2 Reduction

In the homework assigned in the previous lecture, you designed a few divide-and-conquer algorithms to take the sum and the max of a given array. As we hinted, there is a commonality between each of these divide-and-conquer algorithms. That is, each one is a particular instance of a more general pattern: the reduction. A **reduction** is an operation which combines a given set of values according to a specified **identity element** and a specified **associative combining operator**. Let S denote a set. Recall from algebra that an associative combining operator is any binary operator $\oplus$ such that, for any three items $x, y, z \in S$, the following holds.

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

An element $\mathbf{I} \in S$ is an identity element if for any $x \in S$ the following holds.

$$(x \oplus \mathbf{I}) = (\mathbf{I} \oplus x) = x$$

This algebraic structure consisting of $(S, \oplus, \mathbf{I})$ is called a **monoid** and is particularly worth knowing because this structure is a common pattern in parallel computing.

Example 10.4 Addition monoid

- S = the set of all 64-bit unsigned integers; $\oplus$ = addition modulo 2^{64}; $\mathbf{I} = 0$

Example 10.5 Multiplication monoid

- S = the set of all 64-bit unsigned integers; $\oplus$ = multiplication modulo 2^{64}; $\mathbf{I} = 1$

Example 10.6 Max monoid

- S = the set of all 64-bit unsigned integers; $\oplus$ = max function; $\mathbf{I} = 0$

The identity element is important because we are working with sequences: having a base element is essential for dealing with empty sequences. For example, what should the sum of the empty sequence? More interestingly, what should be the maximum (or minimum) element of an empty sequence? The identity element specifies this behavior.

What about the associativity of $\oplus$? Why does associativity matter? Suppose we are given the sequence $[a_0, a_1, \ldots, a_n]$. The serial reduction of this sequence always computes the expression $(a_0 \oplus a_1 \oplus \ldots \oplus a_n)$. However, when the reduction is performed in parallel, the expression computed by the reduction could be $((a_0 \oplus a_1 \oplus a_2 \oplus a_3) \oplus (a_4 \oplus a_5) \oplus \ldots \oplus (a_{n-1} \oplus a_n))$ or $((a_0 \oplus a_1 \oplus a_2) \oplus (a_3 \oplus a_4 \oplus a_5) \oplus \ldots \oplus (a_{n-1} \oplus a_n))$. In general, the exact placement of the parentheses in the parallel computation depends on the way that the parallel algorithm decomposes the problem. Associativity gives the parallel algorithm the flexibility to choose an efficient order of evaluation and still get the same result in the end. The flexibility to choose the decomposition of the problem is exploited by efficient parallel algorithms, for reasons that should be clear by now. In summary, associativity is a key building block to the solution of many problems in parallel algorithms.

Now that we have monoids for describing a generic method for combining two items, we can consider a generic method for combining many items in parallel. Once we have this ability, we will see that we can solve the remaining problems from last homework by simply plugging the appropriate monoids into our generic operator, `reduce`. The interface of this operator in our framework is specified below. The first parameter corresponds to $\oplus$, the second to the identity element, and the third to the sequence to be processed.

```
template <class Assoc_binop>
value_type reduce(Assoc_binop plus_fct, value_type id, const sparray& xs);
```

We can solve our first problem by plugging integer plus as $\oplus$ and 0 as $\mathbf{I}$.

Example 10.7 Solution to homework exercise: summing elements of array

```
auto plus_fct = [&] (value_type x, value_type y) {
  return x+y;
};

sparray xs = { 1, 2, 3 };
std::cout << "sum_xs = " << reduce(plus_fct, 0, xs) << std::endl;
```

Output:

```
reduce(plus_fct, 0, xs) = 6
```

We can solve our second problem in a similar fashion. Note that in this case, since we know that the input sequence is nonempty, we can pass the first item of the sequence as the identity element. What could we do if we instead wanted a solution that can deal with zero-length sequences? What identity element might make sense in that case? Why?

Example 10.8 Solution to homework exercise: taking max of elements of array

Let us start by solving a special case: the one where the input sequence is nonempty.

```
auto max_fct = [&] (value_type x, value_type y) {
  return std::max(x, y);
};

sparray xs = { -3, 1, 634, 2, 3 };
std::cout << "reduce(max_fct, xs[0], xs) = " << reduce(max_fct, xs[0], xs) << std::endl;
```

Output:

```
reduce(max_fct, xs[0], xs) = 634
```

Observe that in order to seed the reduction we selected the provisional maximum value to be the item at the first position of the input sequence. Now let us handle the general case by seeding with the smallest possible value of type `long`.

```
long max(const sparray& xs) {
  return reduce(max_fct, LONG_MIN, xs);
}
```

The value of `LONG_MIN` is defined by `<limits.h>`.

Like the tabulate function, reduce is a higher-order function. Just like any other higher-order function, the work and span costs have to account for the cost of the client-supplied function, which is in this case, the associative combining operator. The analysis is straightforward in the case where the associative combining operator takes constant time. In this case, the reduction takes linear work and logarithmic span in the size of the array. It should easy to convince yourself that these costs are the correct costs for the `reduce` function below.

```
template <class Assoc_binop>
value_type reduce(Assoc_binop b, value_type id, const sparray& xs) {
  return reduce_rec(b, id, xs, 0, xs.size());
}

//-----------------------
// Granularity controller

template <class Assoc_binop>
class reduce_controller {
public:
  static controller_type contr;
};

template <class Assoc_binop>
controller_type
  reduce_controller::contr("reduce"+std::string(typeid(Assoc_binop).name()));

//-----------------
// Recursive reduce

template <class Assoc_binop>
value_type reduce_rec(Assoc_binop b, value_type id, const sparray& xs,
                      long lo, long hi) {
  value_type result = id;
  long n = hi-lo;
  cstmt(reduce_controller<Assoc_binop>::contr, [&] { return n; }, [&] {
    if (n == 0) {
      return id;
    } else if (n == 1) {
      return xs[lo];
    } else {
      long mid = (lo+hi)/2;
      value_type x,y;
      fork2([&] {
        x = reduce_rec(b, id, xs, lo, mid);
      }, [&] {
        y = reduce_rec(b, id, xs, mid, hi);
      });
      result = b(x, y);
    }
  }, [&] {
    for (long i = lo; i < hi; i++)
      result = b(result, xs[i]);
  });
  return result;
}
```

The situation is more complicated when the work cost of the associative combining operator is not constant. Because the reduce

operations that we are going to consider in this course all take constant time, we leave this line of investigation to students who are interested to study independently.

10.3 Scan

A **scan** is an iterated reduction that is typically expressed in one of two forms: inclusive and exclusive. The inclusive form maps a given sequence $[x_0, x_1, x_2, \ldots, x_{n-1}]$ to $[x_0, x_0 \oplus x_1, x_0 \oplus x_1 \oplus x_2, \ldots, x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1}]$.

```
template <class Assoc_binop>
sparray scan_incl(Assoc_binop b, value_type id, const sparray& xs);
```

Example 10.9 Inclusive scan

```
  scan_incl(plus_fct, 0, sparray({ 2, 1, 8, 3 }))
= { reduce(plus_fct, id, { 2 }),       reduce(plus_fct, id, { 2, 1 }),
    reduce(plus_fct, id, { 2, 1, 8 }), reduce(plus_fct, id, { 2, 1, 8, 3}) }
= { 0+2, 0+2+1, 0+2+1+8, 0+2+1+8+3 }
= { 2, 3, 11, 14 }
```

The exclusive form maps a given sequence $[x_0, x_1, x_2, \ldots, x_{n-1}]$ to $[\mathbf{I}, x_0, x_0 \oplus x_1, x_0 \oplus x_1 \oplus x_2, \ldots, x_0 \oplus x_1 \oplus \ldots \oplus x_{n-2}]$. For convenience, we extend the result of the exclusive form with the total $x_0 \oplus \ldots \oplus x_{n-1}$.

```
class scan_excl_result {
public:
  sparray partials;
  value_type total;
};

template <class Assoc_binop>
scan_excl_result scan_excl(Assoc_binop b, value_type id, const sparray& xs);
```

Example 10.10 Exclusive scan

The example below represents the logical behavior of `scan`, but actually says nothing about the way scan is implemented.

```
  scan_excl(plus_fct, 0, { 2, 1, 8, 3 }).partials
= { reduce(plus_fct, 0, { }),      reduce(plus_fct, 0, { 2 }),
    reduce(plus_fct, 0, { 2, 1 }), reduce(plus_fct, 0, { 2, 1, 8}) }
= { 0, 0+2, 0+2+1, 0+2+1+8 }
= { 0, 2, 3, 11 }

  scan_excl(plus_fct, 0, { 2, 1, 8, 3 }).total
= reduce(plus_fct, 0, { 2, 1, 8, 3})
= { 0+2+1+8+3 }
= 14
```

Scan has applications in many parallel algorithms. To name just a few, scan has been used to implement radix sort, search for regular expressions, dynamically allocate processors, evaluate polynomials, etc. Suffice to say, scan is important and worth knowing about because scan is a key component of so many efficient parallel algorithms. In this course, we are going to study a few more applications not in this list.

The expansions shown above suggest the following sequential algorithm.

```
template <class Assoc_binop>
scan_excl_result scan_excl_seq(Assoc_binop b, value_type id, const sparray& xs) {
  long n = xs.size();
  sparray r = array(n);
  value_type x = id;
  for (long i = 0; i < n; i++) {
```

```
    r[i] = x;
    x = b(x, xs[i]);
  }
  return make_scan_result(r, x);
}
```

If we just blindly follow the specification above, we might be tempted to try the solution below.

```
loop_controller_type scan_contr("scan");

template <class Assoc_binop>
sparray scan_excl(Assoc_binop b, value_type id, const sparray& xs) {
  long n = xs.size();
  sparray result = array(n);
  result[0] = id;
  parallel_for(scan_contr, 1l, n, [&] (long i) {
    result[i] = reduce(b, id, slice(xs, 0, i-1));
  });
  return result;
}
```

Question

Although it is highly parallel, this solution has a major problem. What is it?

Consider that our sequential algorithm takes linear time in the size of the input array. As such, finding a work-efficient parallel solution means finding a solution that also takes linear work in the size of the input array. The problem is that our parallel algorithm takes quadratic work: it is not even asymptotically work efficient! Even worse, the algorithm performs a lot of redundant work. Can we do better? Yes, in fact, there exist solutions that take, in the size of the input, both linear time and logarithmic span, assuming that the given associative operator takes constant time. It might be worth pausing for a moment to consider this fact, because the specification of scan may at first look like it would resist a solution that is both highly parallel and work efficient. Unfortunately, we do not have time to consider solutions to the parallel `scan` problem in this course.

10.4 Derived data-parallel operations

The remaining operations that we are going to consider are useful for writing more succinct code and for expressing special cases where certain optimizations are possible. All of the the operations that are presented in this section are derived forms of tabulate, reduce, and scan.

10.4.1 Map

The `map(f, xs)` operation applies `f` to each item in `xs` returning the array of results. It is straightforward to implement as a kind of tabulation, as we have at our disposal efficient indexing.

```
template <class Func>
sparray map(Func f, sparray xs) {
  return tabulate([&] (long i) { return f(xs[i]); }, xs.size());
}
```

The array-increment operation that we defined on the first day of lecture is simple to express via `map`.

Example 10.11 Incrementing via `map`

```
sparray map_incr(sparray xs) {
  return map([&] (value_type x) { return x+1; }, xs);
}
```

The work and span costs of `map` are similar to those of tabulate. Granularity control is handled similarly as well. However, that the granularity controller object corresponding to `map` is instantiated properly is not obvious. It turns out that, for no extra effort, the behavior that we want is indeed preserved: each distinct function that is passed to `map` is assigned a distinct granularity controller. Although it is outside the scope of this course, the reason that this scheme works in our current design owes to specifics of the C++ template system.

10.4.2 Fill

The call `fill(v, n)` creates an array that is initialized with a specified number of items of the same value. Although just another special case for tabulation, this function is worth having around because internally the `fill` operation can take advantage of special hardware optimizations, such as SIMD instructions, that increase parallelism.

```
sparray fill(value_type v, long n);
```

Example 10.12 Creating an array of all 3s

```
sparray threes = fill(3, 5);
std::cout << "threes = " << threes << std::endl;
```

Output:

```
threes = { 3, 3, 3, 3, 3 }
```

10.4.3 Copy

Just like `fill`, the `copy` operation can take advantage of special hardware optimizations that accellerate memory traffic. For the same reason, the `copy` operation is a good choice when a full copy is needed.

```
sparray copy(const sparray& xs);
```

Example 10.13 Copying an array

```
sparray xs = { 3, 2, 1 };
sparray ys = copy(xs);
std::cout << "xs = " << xs << std::endl;
std::cout << "ys = " << ys << std::endl;
```

Output:

```
xs = { 3, 2, 1 }
ys = { 3, 2, 1 }
```

10.4.4 Slice

We now consider a slight generalization on the copy operator: with the `slice` operation we can copy out a range of positions from a given array rather than the entire array.

```
sparray slice(const sparray& xs, long lo, long hi);
```

The `slice` operation takes a source array and a range to copy out and returns a fresh array that contains copies of the items in the given range.

Example 10.14 Slicing an array

```
sparray xs = { 1, 2, 3, 4, 5 };
std::cout << "slice(xs, 1, 3) = " << slice(xs, 1, 3) << std::endl;
std::cout << "slice(xs, 0, 4) = " << slice(xs, 0, 4) << std::endl;
```

Output:

```
{ 2, 3 }
{ 1, 2, 3, 4 }
```

10.4.5 Concat

In contrast to `slice`, the `concat` operation lets us "copy in" to a fresh array.

```
sparray concat(const sparray& xs, const sparray& ys);
```

Example 10.15 Concatenating two arrays

```
sparray xs = { 1, 2, 3 };
sparray ys = { 4, 5 };
std::cout << "concat(xs, ys) = " << concat(xs, ys) << std::endl;
```

Output:

```
{ 1, 2, 3, 4, 5 }
```

10.4.6 Prefix sums

The prefix sums problem is a special case of the scan problem. We have defined two solutions for two variants of the problem: one for the exclusive prefix sums and one for the inclusive case.

```
sparray prefix_sums_incl(const sparray& xs);
scan_excl_result prefix_sums_excl(const sparray& xs);
```

Example 10.16 Inclusive and exclusive prefix sums

```
sparray xs = { 2, 1, 8, 3 };
sparray incl = prefix_sums_incl(xs);
scan_excl_result excl = prefix_sums_excl(xs);
std::cout << "incl = " << incl << std::endl;
std::cout << "excl.partials = " << excl.partials << "; excl.total = " << excl.total << std ↩
    ::endl;
```

Output:

```
incl = { 2, 3, 11, 14 }
excl.partials = { 0, 2, 3, 11 }; excl.total = 147
```

10.4.7 Filter

The last data-parallel operation that we are going to consider is the operation that copies out items from a given array based on a given predicate function.

```
template <class Predicate>
sparray filter(Predicate pred, const sparray& xs);
```

For our purposes, a predicate function is any function that takes a value of type `long` (i.e., `value_type`) and returns a value of type `bool`.

Example 10.17 Extracting even numbers

The following function copies out the even numbers it receives in the array of its argument.

```
bool is_even(value_type x) {
  return (x%2) == 0;
}

sparray extract_evens(const sparray& xs) {
  return filter([&] (value_type x) { return is_even(x); }, xs);
}

sparray xs = { 3, 5, 8, 12, 2, 13, 0 };
std::cout << "extract_evens(xs) = " << extract_evens(xs) << std::endl;
```

Output:

```
extract_evens(xs) = { 8, 12, 2, 0 }
```

Example 10.18 Solution to the sequential-filter problem

The particular instance of the filter problem that we are considering is a little tricky because we are working with fixed-size arrays. In particular, what requires care is the method that we use to copy the selected items out of the input array to the output array. We need to first run a pass over the input array, applying the predicate function to the items, to determine which items are to be written to the output array. Furthermore, we need to track how many items are to be written so that we know how much space to allocate for the output array.

```
template <class Predicate>
sparray filter(Predicate pred, const sparray& xs) {
  long n = xs.size();
  long m = 0;
  sparray flags = array(n);
  for (long i = 0; i < n; i++)
    if (pred(xs[i])) {
      flags[i] = true;
      m++;
    }
  sparray result = array(m);
  long k = 0;
  for (long i = 0; i < n; i++)
    if (flags[i])
      result[k++] = xs[i];
  return result;
}
```

Question

In the sequential solution above, it appears that there are two particular obstacles to parallelization. What are they?

Hint: the obstacles relate to the use of variables `m` and `k`.

Question

Under one particular assumption regarding the predicate, this sequential solution takes linear time in the size of the input, using two passes. What is the assumption?

10.4.8 Lab exercise: solve the parallel-filter problem

The starting point for our solution is the following code.

```
template <class Predicate>
sparray filter(Predicate p, const sparray& xs) {
  sparray flags = map(p, xs);
  return pack(flags, xs);
}
```

The challenge of this exercise is to solve the following problem: given two arrays of the same size, the first consisting of boolean valued fields and the second containing the values, return the array that contains (in the same relative order as the items from the input) the values selected by the flags. Your solution should take linear work and logarithmic span in the size of the input.

```
sparray pack(const sparray& flags, const sparray& xs);
```

Example 10.19 The allocation problem

```
sparray flags = { true, false, false, true, false, true, true };
sparray xs    = {   34,    13,     5,    1,    41,   11,   10 };
std::cout << "pack(flags, xs) = " << pack(flags, xs) << std::endl;
```

Output:

```
pack(flags, xs) = { 34, 1, 11, 10 }
```

Tip
You can use scans to implement `pack`.

Note
Even though our arrays can store only 64-bit values of type `long`, we can nevertheless store values of type `bool`, as we have done just above with the `flags` array. The compiler automatically promotes boolean values to long values without causing us any problems, at least with respect to the correctness of our solutions. However, if we want to be more space efficient, we need to use arrays that are capable of packing values of type `bool` more efficiently, e.g., into single- or eight-bit fields. It should easy to convince yourself that achieving such specialized arrays is not difficult, especially given that the template system makes it easy to write polymorphic containers.

11 Lectures 7 and 8: Parallel Sorting

In this chapter, we are going to study parallel implementations of quicksort and mergesort.

11.1 Quicksort

The quicksort algorithm for sorting an array (sequence) of elements is known to be a very efficient sequential sorting algorithm. In fact, many efficient sorting algorithms are variants of quicksort. A natural question thus is whether quicksort is similarly effective as a parallel algorithm?

Let us first convince ourselves, at least informally, that quicksort is actually a good parallel algorithm. But first, what do we mean by "parallel quicksort." Chances are that you think of quicksort as an algorithm that, given an array, starts by reorganizing the elements in the array around a randomly selected pivot by using an in-place *partitioning* algorithm, and then sorts the two parts of the array to the left and the right of the array recursively.

While this implementation of the quicksort algorithm is not immediately parallel, it can be parallelized.

Question

Can you think of a way to parallelize the algorithm?

Note that the recursive calls are naturally independent. So we really ought to focus on the partitioning algorithm. There is a rather simple way to do such a partition in parallel by performing three filter operations on the input array, one for picking the elements less that the pivot, one for picking the elements equal to the pivot, and another one for picking the elements greater than the pivot. This algorithm can be described as follows.

Algorithm: parallel quicksort

1. Pick from the input sequence a pivot item.
2. Based on the pivot item, create a three-way partition of the input sequence:
    a. the sequence of items that are less than the pivot item,
    b. those that are equal to the pivot item, and
    c. those that are greater than the pivot item.
3. Recursively sort the "less-than" and "greater-than" parts
4. Concatenate the sorted arrays.

Now that we have a parallel algorithm, we can check whether it is a good algorithm or not. Recall that a good parallel algorithm is one that has the following three characteristics

1. It is asymptotically work efficient
2. It is observably work efficient
3. It is highly parallel, i.e., has low span.

Let us first convince ourselves that Quicksort is a highly parallel algorithm. Observe that

1. the dividing process is highly parallel because no dependencies exist among the intermediate steps involved in creating the three-way partition,
2. two recursive calls are parallel, and
3. concatenations are themselves highly parallel.

Let us now turn our attention to asymptotic and observed work efficiency. Recall first that quicksort can exhibit a quadratic-work worst-case behavior on certain inputs if we select the pivot deterministically. To avoid this, we can pick a random element as a pivot by using a random-number generator, but then we need a parallel random number generator. For the purposes of this course, we are going to side-step this issue by assuming that the input is randomly permuted in advance. Under this assumption, we can simply pick the pivot to be the first item of the sequence. With this assumption, our algorithm performs asymptotically the same work as sequential quicksort implementations that perform $\Theta(n \log n)$ in expectation.

For an implementation to be observably work efficient, we know that we must control granularity by switching to a fast sequential sorting algorithm when the input is small. This is easy to achieve using our granularity control technique by using `seqsort()`, a fast sequential algorithm provided in the code base; `seqsort()` is really a call to STL's sort function. Of course, we have to assess observable work efficiency experimentally after specifying the implementation.

The code for quicksort is shown below. Note that we use our array class `sparray` to store the input and output. To partition the input, we use our parallel filter function from the previous lecture to efficiently parallelize the partitioning phase. Similarly, we use our parallel concatenation function to constructed the sorted output.

```
controller_type quicksort_contr("quicksort");

sparray quicksort(const sparray& xs) {
  long n = xs.size();
  sparray result = { };
  cstmt(quicksort_contr, [&] { return n * std::log2(n); }, [&] {
    if (n == 0) {
      result = { };
    } else if (n == 1) {
      result = { xs[0] };
    } else {
      value_type p = xs[0];
      sparray less = filter([&] (value_type x) { return x < p; }, xs);
      sparray equal = filter([&] (value_type x) { return x == p; }, xs);
      sparray greater = filter([&] (value_type x) { return x > p; }, xs);
      sparray left = { };
      sparray right = { };
      fork2([&] {
        left = quicksort(less);
      }, [&] {
        right = quicksort(greater);
      });
      result = concat(left, equal, right);
    }
  }, [&] {
    result = seqsort(xs);
  });
  return result;
}
```

By using randomized-analysis techniques, it is possible to analyze the work and span of this algorithm. The techniques needed to do so are well beyond the scope of this course. The interested reader can find more detail in the Principles of Parallel Algorithms book.

Exercise

Show that the algorithm above takes expected work $O(n \log n)$ and expected span $O(\log^2 n)$, where n is the number of items in the input sequence.

One consequence of the work and span bounds that we have stated above is that our quicksort algorithm is highly parallel: its average parallelism is $\frac{O(n \log n)}{O(\log^2 n)} = \frac{n}{\log n}$. When the input is large, there should be ample parallelism to keep many processors well fed with work. For instance, when $n = 100$ million items, the average parallelism is $\frac{10^8}{\log 10^8} \approx \frac{10^8}{40} \approx 3.7$ million. Since 3.7 million is much larger than the number of processors in our machine, that is, forty, we have a ample parallelism.

Unfortunately, the code that we wrote leaves much to be desired in terms of observable work efficiency. Consider the following benchmarking runs that we performed on our 40-processor machine.

```
$ prun speedup -baseline "bench.baseline" -parallel "bench.opt -proc 1,10,20,30,40" -bench ↩
    quicksort -n 100000000
```

The first two runs show that, on a single processor, our parallel algorithm is roughly 6x slower than the sequential algorithm that we are using as baseline! To relate with the material we presented early in the course, our quicksort has "6-observed work efficiency", according to the definition that we gave for work efficiency. That means we need at least six processors working on the problem to even see an improvement.

```
[1/6]
bench.baseline -bench quicksort -n 100000000
exectime 12.518
[2/6]
```

```
bench.opt -bench quicksort -n 100000000 -proc 1
exectime 78.960
```

The rest of the results confirm that it takes about ten processors to see a little improvement and forty processors to see approximately a 2.5x speedup. Figure 11 shows the speedup plot for this program. Clearly, it does not look good.

```
[3/6]
bench.opt -bench quicksort -n 100000000 -proc 10
exectime 9.807
[4/6]
bench.opt -bench quicksort -n 100000000 -proc 20
exectime 6.546
[5/6]
bench.opt -bench quicksort -n 100000000 -proc 30
exectime 5.531
[6/6]
bench.opt -bench quicksort -n 100000000 -proc 40
exectime 4.761
```

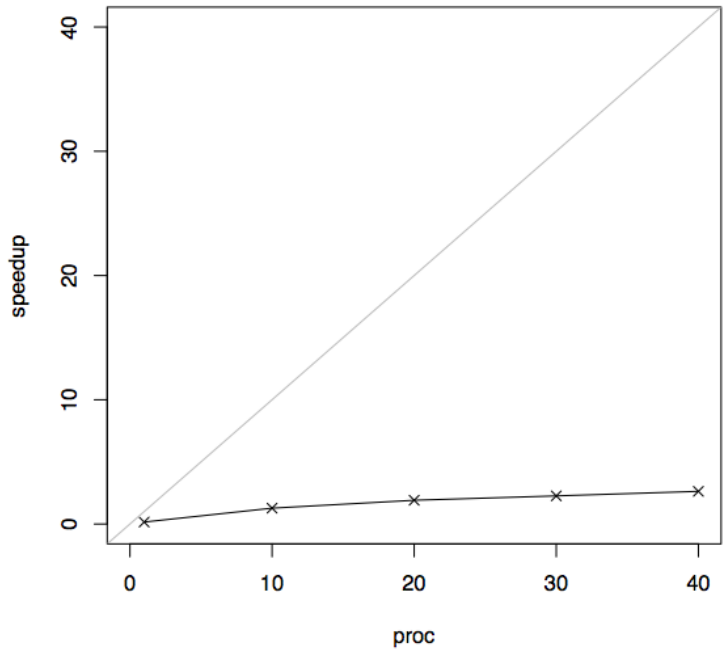


Figure 11: Speedup plot for quicksort with 100000000 elements.

We are pretty sure that we have a good algorithm, but the code that we wrote may be less efficient that it could be.

Question

What can we do to improve our code?

Tip

Eliminate unecessary copying and array allocations.

Tip
Eliminate redundant work by building the partition in one pass instead of three.

We encourage students to look for improvements to quicksort independently. For now, we are going to consider parallel mergesort. This time, we are going to focus more on achieving better speedups.

11.2 Mergesort

As a divide-and-conquer algorithm, the mergesort algorithm, is a good candidate for parallelization, because the two recursive calls for sorting the two halves of the input can be independent. The final merge operation, however, is typically performed sequentially. It turns out to be not too difficult to parallelize the merge operation to obtain good work and span bounds for parallel mergesort. The resulting algorithm turns out to be a good parallel algorithm, delivering asymptotic, and observably work efficiency, as well as low span.

Mergesort algorithm

1. Divide the (unsorted) items in the input array into two equally sized subrange.
2. Recursively and in parallel sort each subrange.
3. Merge the sorted subranges.

This process requires a "merge" routine which merges the contents of two specified subranges of a given array. The merge routine assumes that the two given subarrays are in ascending order. The result is the combined contents of the items of the subranges, in ascending order.

The precise signature of the merge routine appears below and its description follows. In mergesort, every pair of ranges that are merged are adjacent in memory. This observation enables us to write the following function. The function merges two ranges of source array `xs`: `[lo, mid)` and `[mid, hi)`. A temporary array `tmp` is used as scratch space by the merge operation. The function writes the result from the temporary array back into the original range of the source array: `[lo, hi)`.

```
void merge(sparray& xs, sparray& tmp, long lo, long mid, long hi);
```

Example 11.1 Use of merge function

```
sparray xs = {
  // first range: [0, 4)
  5, 10, 13, 14,
  // second range: [4, 9)
  1, 8,  10, 100, 101 };

merge(xs, sparray(xs.size()), (long)0, 4, 9);

std::cout << "xs = " << xs << std::endl;
```

Output:

```
xs = { 1, 5, 8, 10, 10, 13, 14, 100, 101 }
```

To see why sequential merging does not work, let us implement the merge function by using one provided by STL: `std::merge()`. This merge implementation performs linear work and span in the number of items being merged (i.e., `hi -lo`). In our code, we use this STL implementation underneath the `merge()` interface that we described just above.

Now, we can assess our parallel mergesort with a sequential merge, as implemented by the code below. The code uses the traditional divide-and-conquer approach that we have seen several times already.

Question

Is the implementation asymptotically work efficient?

The code is asymptotically work efficient, because nothing significant has changed between this parallel code and the serial code: just erase the parallel annotations and we have a textbook sequential mergesort!

```
sparray mergesort(const sparray& xs) {
  long n = xs.size();
  sparray result = copy(xs);
  mergesort_rec(result, sparray(n), (long)0, n);
  return result;
}

controller_type mergesort_contr("mergesort");

void mergesort_rec(sparray& xs, sparray& tmp, long lo, long hi) {
  long n = hi - lo;
  cstmt(mergesort_contr, [&] { return n * std::log2(n); }, [&] {
    if (n == 0) {
      // nothing to do
    } else if (n == 1) {
      tmp[lo] = xs[lo];
    } else {
      long mid = (lo + hi) / 2;
      fork2([&] {
        mergesort_rec(xs, tmp, lo, mid);
      }, [&] {
        mergesort_rec(xs, tmp, mid, hi);
      });
      merge(xs, tmp, lo, mid, hi);
    }
  }, [&] {
    if (hi-lo < 2)
      return;
    std::sort(&xs[lo], &xs[hi-1]+1);
  });
}
```

Question

How well does our "parallel" mergesort scale to multiple processors, i.e., does it have a low span?

Unfortunately, this implementation has a large span: it is linear, owing to the sequential merge operations after each pair of parallel calls. More precisely, we can write the work and span of this implementation as follows:

$$W(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ W(n/2) + W(n/2) + n \end{cases}$$

$$S(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \max(W(n/2), W(n/2)) + n \end{cases}$$

EQUATION 11.1: Analyzing work and span of mergesort

It is not difficult to show that these recursive equation solve to $W(n) = \Theta(n \log n)$ and $S(n) = \Theta(n)$.

With these work and span costs, the average parallelism of our solution is $\frac{cn \log n}{2cn} = \frac{\log n}{2}$. Consider the implication: if $n = 2^{30}$, then the average parallelism is $\frac{\log 2^{30}}{2} = 15$. That is terrible, because it means that the greatest speedup we can ever hope to achieve is 15x!

The analysis above suggests that, with sequential merging, our parallel mergesort does not expose ample parallelism. Let us put that prediction to the test. The following experiment considers this algorithm on our 40-processor test machine. We are going to sort a random sequence of 100 million items. The baseline sorting algorithm is the same sequential sorting algorithm that we used for our quicksort experiments: `std::sort()`.

```
$ prun speedup -baseline "bench.baseline" -parallel "bench.opt -proc 1,10,20,30,40" -bench  ↩
    mergesort_seqmerge -n 100000000
```

The first two runs suggest that our mergesort has better observable work efficiency than our quicksort. The single-processor run of parallel mergesort is roughly 50% slower than that of the sequential baseline algorithm. Compare that to the 6x-slower running time for single-processor parallel quicksort! We have a good start.

```
[1/6]
bench.baseline -bench mergesort_seqmerge -n 100000000
exectime 12.483
[2/6]
bench.opt -bench mergesort_seqmerge -n 100000000 -proc 1
exectime 19.407
```

The parallel runs are encouraging: we get 5x speedup with 40 processors.

```
[3/6]
bench.opt -bench mergesort_seqmerge -n 100000000 -proc 10
exectime 3.627
[4/6]
bench.opt -bench mergesort_seqmerge -n 100000000 -proc 20
exectime 2.840
[5/6]
bench.opt -bench mergesort_seqmerge -n 100000000 -proc 30
exectime 2.587
[6/6]
bench.opt -bench mergesort_seqmerge -n 100000000 -proc 40
exectime 2.436
```

But we can do better by using a parallel merge instead of a sequential one: the speedup plot in Figure 12 shows three speedup curves, one for each of three mergesort algorithms. The `mergesort()` algorithm is the same mergesort routine that we have seen here, except that we have replaced the sequential merge step by our own parallel merge algorithm. The `cilksort()` algorithm is the carefully optimized algorithm taken from the Cilk benchmark suite. What this plot shows is, first, that the parallel merge significantly improves performance, by at least a factor of two. The second thing we can see is that the optimized Cilk algorithm is just a little faster than the one we presented here. That's pretty good, considering the simplicity of the code that we had to write.

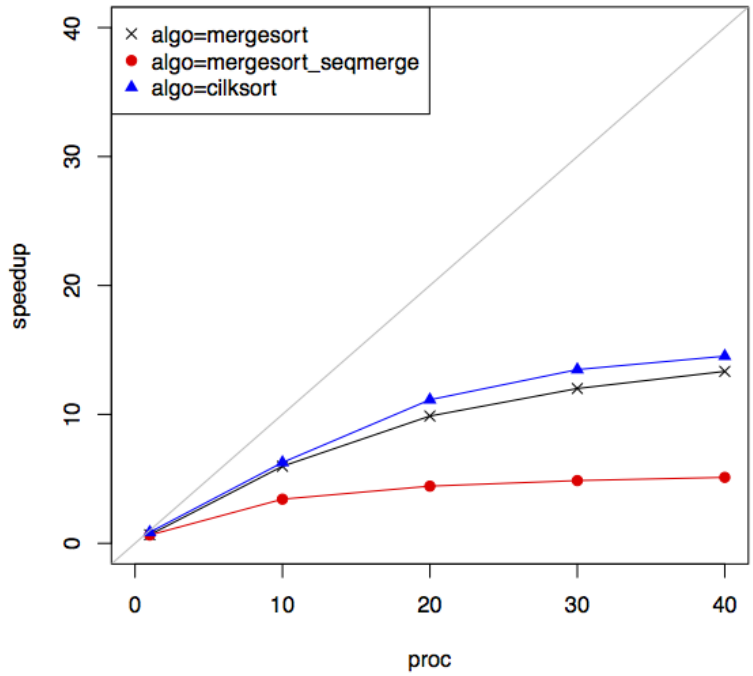


Figure 12: Speedup plot for three different implementations of mergesort using 100 million items.

It turns out that we can do better by simply changing some of the variables in our experiment. The plot shown in Figure 13 shows the speedup plot that we get when we change two variables: the input size and the sizes of the items. In particular, we are selecting a larger number of items, namely 250 million instead of 100 million, in order to increase the amount of parallelism. And, we are selecting a smaller type for the items, namely 32 bits instead of 64 bits per item. The speedups in this new plot get closer to linear, topping out at approximately 20x.

Practically speaking, the mergesort algorithm is memory bound because the amount of memory used by mergesort and the amount of work performed by mergesort are both approximately roughly linear. It is an unfortunate reality of current multicore machines that the main limiting factor for memory-bound algorithms is amount of parallelism that can be achieved by the memory bus. The memory bus in our test machine simply lacks the parallelism needed to match the parallelism of the cores. The effect is clear after just a little experimentation with mergesort. You can see this effect yourself, if you are interested to change in the source code the type aliased by `value_type`. For a sufficiently large input array, you should observe a significant performance improvement by changing just the representation of `value_type` from 64 to 32 bits, owing to the fact that with 32-bit items is a greater amount of computation relative to the number of memory transfers.

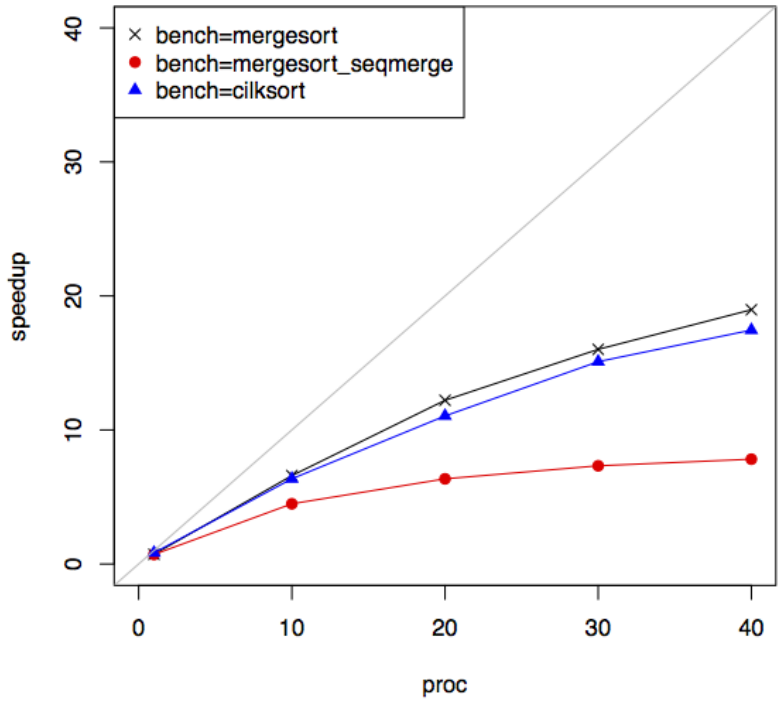


Figure 13: Speedup plot for three different implementations of mergesort using 100 million items.

12 Project 2: parallel merge sort

Your task is to design and implement a parallel merge-sort algorithm that takes work $O(n \log n)$ and span $O(\log^3 n)$.

The key component of such a parallel merge-sort algorithm is a parallel merge algorithm. We recommend that, for your parallel-merge solution, you use divide and conquer. As with all efficient divide-and-conquer solutions, it is crucial to find a way to achieve an even (or approximately even) division of the input. You may be asking, how can I do that? Indeed, it is not obvious, because all that we know for sure is that we are given two sorted sequences. Moreover, there is no relation between the sizes of the two sequences: one can be much larger than the other, for example. So, where in the two sequences is there a cut that evenly divides the work into two roughly balanced pieces?

The big hint that we are going to give is the following. Such a balanced cut exists and can be found by two steps. The first step is to take the median item $x_{n/2}$ from the larger of the two sequences, $[x_0, x_1, \ldots, x_{n/2}, \ldots x_{n-1}]$. The second is to search for an item y_p from the smaller of the two sequences $[y_0, y_1, \ldots, y_p, \ldots y_{m-1}]$, such that y_p is the smallest item in the smaller sequence where $x_{n/2} < y_p$. Thanks to the fact that the input sequences are guaranteed to be sorted, the item y_p can be found in logarithmic time by using a binary search. Fortunately, STL provides a binary search routine, namely `std::lower_bound()`. An important property of the sequential merge-sort algorithm is that it is stable: it can be written in such a way that it preserves the relative order of equal elements in the input. Is the parallel merge-sort algorithm that you designed stable? If not, then can you find a way to make it stable?

13 Lecture 9-12: Graph processing

In just the past few years, a great deal of interest has grown for frameworks that can process very large graphs. Interest is coming from a diverse collection of fields. To name a few: physicists are using graph frameworks to simulate emergent properties from large networks of particles; Google to mine the web for the purpose of web search; social scientists to test theories regarding the origins of social trends; marketers to identify individuals in social networks who are highly influential.

In response, many graph-processing frameworks have been implemented by researchers and industrial laboratories. Such frameworks offer to client programs a particular application programming interface. The purpose of the interface is to give the client

programmer a high-level view of the basic operations of graph processing. Internally, at a lower level of abstraction, the framework provides key algorithms to perform basic functions, such as one or more functions that "drive" the traversal of a given graph.

The exact interface and the underlying algorithms vary from one graph-processing framework to another. One commonality among the frameworks is that it is crucial to harness parallelism, because interesting graphs are often huge and performance can mean the difference between minutes and hours, hours and days, etc.

One dimension on which the frameworks differ is whether or not the framework assumes shared or distributed memory. In this chapter, we focus on shared memory for two reasons. First, distributed memory is out of the scope of this course. Second, nowadays, many interesting graphs fit comfortably in the main memory of a well-provisioned personal computer.

13.1 Adjacency-list format

The format that we are going to consider for representing graphs is the **adjacency list**. In particular, we are going to use adjacency lists to represent directed graphs. An adjacency list is a graph representation that consists of a collection of unordered lists. Each vertex in the graph $G = (V,E)$ is represented by an identifier $v \in \{0,\ldots,n-1\}$, where $n = |V|$ is the number of vertices in the graph. For each vertex v in the graph, the adjacency list stores one list. Each such list, namely $out\_edges\_of[v]$, describes the set of neighbors of v. For our purposes, it suffices for us to store in $out\_edges\_of[v]$ only lists of outgoing edges, although sometimes it is useful to store just incoming edges, and other times useful to store both. The primary reason that we use the adjacency list is because finding the neighbors of a specified vertex is a fast constant-time operation. The graph-traversal techniques that we are going to consider rely on this property to traverse the graph efficiently. In specific, we are going to see that what our graph traversals need and what we have is that all of the operations provided by the `adjlist` implementation take constant time.

```
using vtxid_type = value_type;
using neighbor_list = const value_type*;

class adjlist {
public:
  long get_nb_vertices() const;
  long get_nb_edges() const;
  long get_out_degree_of(vtxid_type v) const;
  neighbor_list get_out_edges_of(vtxid_type v) const;
};
```

Space use is a major concern because graphs that people are interested to analyze nowadays can have tens of billions of edges or more. The Facebook social network graph (including just the network and no metadata) uses 100 billion edges, for example, and as such could fit snugly into a machine with 2TB of memory. Such a large graph is a greater than the capacity of the RAM available on current personal computers. But it is not that far off, and there are many other interesting graphs that easily fit into just a few gigabytes. Our adjacency list consumes a total of $n+m$ vertex-id cells in memory, where $n = |V|$ and $m = |E|$. For simplicity, we always use 64 bits to represent vertex identifiers but note that a practical library would support 32 bit representations as well. Although the format that we use is reasonably space efficient for storing large graphs, we should point out that there are other representations that may offer more compact graphs. In fact, graph-compression techniques are an active area of research at present.

Example 13.1 Graph creation

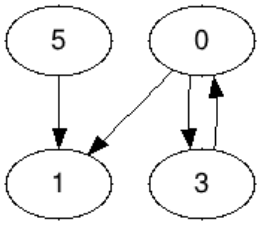
Specifying a (small) graph in textual format is as easy as specifying an edge list. Moreover, getting a textual representation of the graph is as easy as printing the graph by `cout`.

```
adjlist graph = { mk_edge(0, 1), mk_edge(0, 3), mk_edge(5, 1), mk_edge(3, 0) };
std::cout << graph << std::endl;
```

Output:

```
digraph {
0 -> 1;
0 -> 3;
3 -> 0;
5 -> 1;
}
```

Note
The output above is an instance of the "dot" format. This format is used by a well-known graph-visualization tool called graphviz. The diagram below shows the visualization of our example graph that is output by the graphviz tool. You can easily generate such visualizations for your graphs by using online tools, such is Click this one.



Example 13.2 Adjacency-list interface

```
adjlist graph = { mk_edge(0, 1), mk_edge(0, 3), mk_edge(5, 1), mk_edge(3, 0),
                  mk_edge(3, 5), mk_edge(3, 2), mk_edge(5, 3) };
std::cout << "nb_vertices = " << graph.get_nb_vertices() << std::endl;
std::cout << "nb_edges = " << graph.get_nb_edges() << std::endl;
std::cout << "neighbors of vertex 3:" << std::endl;
neighbor_list neighbors_of_3 = graph.get_out_edges_of(3);
for (long i = 0; i < graph.get_out_degree_of(3); i++)
  std::cout << " " << neighbors_of_3[i];
std::cout << std::endl;
```

Output:

```
nb_vertices = 6
nb_edges = 7
neighbors of vertex 3:
 0 5 2
```

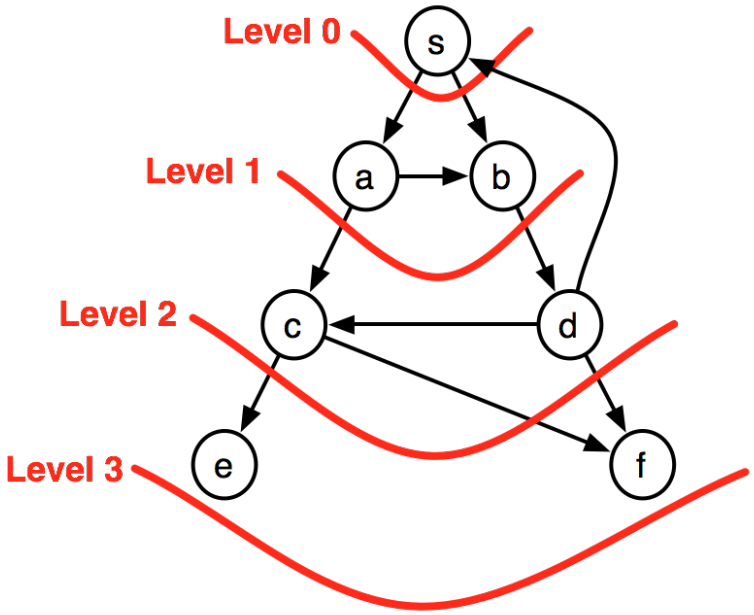
Next, we are going to study a version of breadth-first search that is useful for searching in large in-memory graphs in parallel. After seeing the basic pattern of BFS, we are going to generalize a little to consider general-purpose graph-traversal techniques that are useful for implementing a large class of parallel graph algorithms.

13.2 Breadth-first search

The breadth-first algorithm is a particular graph-search algorithm that can be applied to solve a variety of problems such as finding all the vertices reachable from a given vertex, finding if an undirected graph is connected, finding (in an unweighted graph) the shortest path from a given vertex to all other vertices, determining if a graph is bipartite, bounding the diameter of an undirected graph, partitioning graphs, and as a subroutine for finding the maximum flow in a flow network (using Ford-Fulkerson's algorithm). As with the other graph searches, BFS can be applied to both directed and undirected graphs.

The idea of ***breadth first search***, or ***BFS*** for short, is to start at a *source* vertex s and explore the graph outward in all directions level by level, first visiting all vertices that are the (out-)neighbors of s (i.e. have distance 1 from s), then vertices that have distance two from s, then distance three, etc. More precisely, suppose that we are given a graph G and a source s. We define the ***level*** of a vertex v as the shortest distance from s to v, that is the number of edges on the shortest path connecting s to v.

Example 13.3 BFS Levels
A graph and its levels.



13.2.1 Sequential BFS

Many variations of BFS have been proposed over the years. The one that may be most widely known is the classic sequential BFS that uses a FIFO queue to buffer vertices that are waiting to be visited. The FIFO-based approach is a poor approach for parallelization because accesses to the FIFO queue are by definition serialized.

13.2.2 Parallel BFS

A better place to start for a parallel algorithm is to start with the level-order traversal of the graph. We can implement such a level-order traversal by maintaining a *frontier* as a set of vertices that have not yet been visited but will be visited next, and visiting the vertices in the frontier (which represents all the vertices is a level) all together in parallel. The pseudo-code for this is shown below.

Set-based pseudocode for parallel BFS

```
frontier = { source }
visited = {}
while frontier not empty
  start level
    next = {}
    foreach vertex v in current frontier
      visit v
    visited = visited set-union frontier

    foreach v in frontier
      next = next set-union (neighbors of v)
    frontier = next set-difference visited
  end level
```

Exercise

Convince yourself that this algorithm does indeed perform a BFS by performing a level-by-level traversal.

Assuming that we have a parallel set data structure, let us parallelize this algorithm. First, note that we can visit all the vertices in the frontier in parallel. That is, we can parallelize the first `foreach` loop. Second, we can also compute the next set in

parallel by performing a reduce with the set-union operation. Finally, we can compute the next frontier by taking a set-difference operation.

In addition, it turns out that there are many different ways to implement the BFS algorithm by considering two factors:

1. The specific set data structure that we wish to use.
2. Whether we wish to use atomic read-modify-write operations such as compare-and-swap operations available in modern multicore computers

In this course, we will implement this algorithm by using atomic ***read-modify-write*** operations such as **compare-and-swap** and an array-based implementation of sets. By using this representation, we are able to present a simple implementation that also performs reasonably well by reducing the different sets that must be computed as well as the total number of set operations.

To this end, we will change the notion of the frontier slightly. Instead of holding the vertices that we are will visit next, the frontier will hold the vertices we just visited. At each level, we will visit the neighbors of the vertices in the frontier, but only if they have not yet been visited. Why is this guard necessary? It is necessary because two vertices in the frontier can both have a vertex as their neighbor. In fact, without such a guard, the algorithm can may fail to terminate if the graph has a cycle. After we visited all the neighbors of the vertices in the frontier at this level, we assign the frontier to be the vertices visited. The pseudocode for this algorithm is shown below.

Pseudocode for parallel BFS

```
visit source
frontier = { source }
while frontier not empty
  start level
    foreach vertex in current frontier
      foreach neighbor of vertex
        if neighbor not visited
          visit neighbor v
    frontier = neighbors visited at this level
   end level
```

Now, let us turn our attention to parallelism. From the pseudocode, we see that there are at least two clear opportunities for parallelism. The first is the foreach loop that processes the frontier and the second the foreach loop that processes the neighbors of the vertex that is currently being visited. These two loops should expose a lot of parallelism, at least for certain classes of graphs. The outer loop exposes a lot of parallelism when the frontier gets to be large. The inner loop exposes a lot of parallelism when the traversal reaches a vertex that has a high out degree.

Question

The while loop cannot be parallelized. Why not?

Parallelizing the two foreach loops requires some extra care, because parallelizing in a naive fashion would enable a race condition. To see why, we need to consider how an implementation of BFS keeps track of which vertices have been visited already and which have not. Suppose that we use an array of booleans $visited[v]$ of size n that is keyed by the vertex identifier. If $visited[v] = \text{true}$, then vertex v has been visited already and has not otherwise. Suppose now that two processors, namely A and B, concurrently attempt to gain access to the same vertex v (via two different neighbors of v). If A and B both read $visited[v]$ at the same time, then both consider that they have gained access to v. Both processors then mark v as visited and then proceed to visit the neighbors of v. As such, v will be visited twice and subsequently have its outgoing neighbors processed twice.

Consider now the implication: owing to the race condition, such an implementation of parallel BFS cannot in general guarantee that each reachable vertex is visited once and only once. Of course, the race does not necessarily happen every time a vertex is visited. But even if it happens only rarely, there is a real chance that a huge, in fact unbounded, amount of redundant work is performed by the BFS: when the same vertex is visited twice, all of its neighbors are processed twice. The amount of redundant work that is performed is high when the outdegree of the vertex is high. In other words, a racy parallel BFS is not even an asymptotically work-efficient BFS due to the unbounded amount of redundant work that it could perform in any given round.

Question

Clearly, the race conditions on the visited array that we described above can cause BFS to visit any given vertex twice.

- Could such race conditions cause the BFS to visit some vertex that is not reachable? Why or why not?
- Could such race conditions cause the BFS to not visit some vertex that is reachable? Why or why not?
- Could such race conditions trigger infinite loops? Why or why not?

13.2.3 Atomic memory

The issues relating to the race conditon leads us to consider lightweight atomic memory. We can use lightweight atomic memory to both eliminate race conditions and avoid having to sacrifice a lot of performance. The basic idea is to guard each cell in our "visited" array by an atomic type.

Example 13.4 Accessing the contents of atomic memory cells

Access to the contents of any given cell is achieved by the `load()` and `store()` methods.

```
const long n = 3;
std::atomic<bool> visited[n];
long v = 2;
visited[v].store(false);
std::cout << visited[v].load() << std::endl;
visited[v].store(true);
std::cout << visited[v].load() << std::endl;
```

Output:

```
0
1
```

The key operation that enables us to eliminate the race condition is the **compare and exchange** operation.

Example 13.5 Compare and Exchange

This operation performs the following steps, atomically:

1. Read the contents of the target cell in the visited array.
2. If the contents is false (i.e., equals the contents of `orig`), then write `true` into the cell and return `true`.
3. Otherwise, just return `false`.

```
const long n = 3;
std::atomic<bool> visited[n];
long v = 2;
visited[v].store(false);
bool orig = false;
bool was_successful = visited[v].compare_exchange_strong(orig, true);
std::cout << "was_successful = " << was_successful << "; visited[v] = " << visited[v].load ↩
    () << std::endl;
bool orig2 = false;
bool was_successful2 = visited[v].compare_exchange_strong(orig2, true);
std::cout << "was_successful2 = " << was_successful2 << "; visited[v] = " << visited[v]. ↩
    load() << std::endl;
```

Output:

```
was_successful = 1; visited[v] = 1
was_successful2 = 0; visited[v] = 1
```

Excercise

Convince yourself that the compare and exchange eliminates the race conditions.

13.3 Implementation of parallel BFS

So far, we have seen pseudocode that describes at a high level the idea behind the parallel BFS. We have seen that special care is required to eliminate problematic race conditions. We still have a way to go before we can implement an efficient parallel BFS in C++. In this section, we get there and a little further.

The following function signature is the signature for our parallel BFS implementation. The function takes as parameters a graph and the identifier of a source vertex and returns an array of boolean flags.

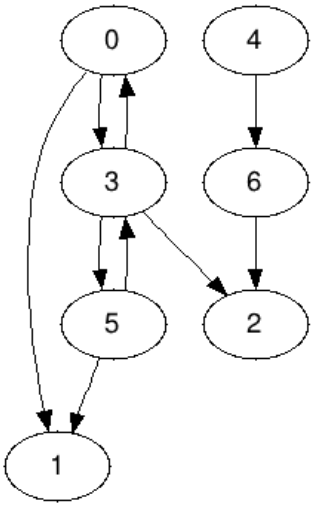
```
sparray bfs(const adjlist& graph, vtxid_type source);
```

The flags array is a length $|V|$ array that specifies the set of vertices in the graph which are reachable from the source vertex: a vertex with identifier v is reachable from the given source vertex if and only if there is a `true` value in the v^{th} position of the flags array that is returned by `bfs`.

Example 13.6 Parallel BFS

```
adjlist graph = { mk_edge(0, 1), mk_edge(0, 3), mk_edge(5, 1), mk_edge(3, 0),
                  mk_edge(3, 5), mk_edge(3, 2), mk_edge(5, 3),
                  mk_edge(4, 6), mk_edge(6, 2) };
std::cout << graph << std::endl;
sparray reachable_from_0 = bfs(graph, 0);
std::cout << "reachable from 0: " << reachable_from_0 << std::endl;
sparray reachable_from_4 = bfs(graph, 4);
std::cout << "reachable from 4: " << reachable_from_4 << std::endl;
```

The following diagram shows the structure represented by `graph`.



Output:

```
digraph {
0 -> 1;
0 -> 3;
3 -> 0;
3 -> 5;
3 -> 2;
4 -> 6;
```

```
5 -> 1;
5 -> 3;
6 -> 2;
}
reachable from 0: { 1, 1, 1, 1, 0, 1, 0 }
reachable from 4: { 0, 0, 1, 0, 1, 0, 1 }
```

There is one helper function that we need to describe before we get to the implementation of our `bfs` function. The following signature specifies the "edge map" operation. This operation takes as parameters a graph, an array of atomic flag values, and a frontier and returns a new frontier.

```
sparray edge_map(const adjlist& graph, std::atomic<bool>* visited, const sparray&  ↩
    in_frontier);
```

Example 13.7 Edge map

```
adjlist graph = // same graph as shown in the previous example
const long n = graph.get_nb_vertices();
std::atomic<bool> visited[n];
for (long i = 0; i < n; i++)
  visited[i] = false;
visited[0].store(true);
visited[1].store(true);
visited[3].store(true);
sparray in_frontier = { 3 };
sparray out_frontier = edge_map(graph, visited, in_frontier);
std::cout << out_frontier << std::endl;
sparray out_frontier2 = edge_map(graph, visited, in_frontier);
std::cout << out_frontier2 << std::endl;
```

Output:

```
{ 5, 2 }
{  }
```

From the perspective of BFS, the edge-map function is the function that advances one level ahead in the level-by-level traversal of the graph. More concretely, this function takes as argument the frontier at level i in the BFS traversal and returns the frontier at level $i+1$.

13.3.1 BFS main loop

The main loop of BFS is shown below. The algorithm uses the edge-map function to advance level by level through the graph. The traversal stops when the frontier is empty.

```
loop_controller_type bfs_init_contr("bfs_init");

sparray bfs(const adjlist& graph, vtxid_type source) {
  long n = graph.get_nb_vertices();
  std::atomic<bool>* visited = my_malloc<std::atomic<bool>>(n);
  parallel_for(bfs_init_contr, 0l, n, [&] (long i) {
    visited[i].store(false);
  });
  visited[source].store(true);
  sparray cur_frontier = { source };
  while (cur_frontier.size() > 0)
    cur_frontier = edge_map(graph, visited, cur_frontier);
  sparray result = tabulate([&] (value_type i) { return visited[i].load(); }, n);
  free(visited);
  return result;
}
```

One minor technical complication relates to the result value: our algorithm performs extra work to copy out the values from the visited array. Although it could be avoided, we choose to copy out the values because it is more convenient for us to program with ordinary `sparray`'s.

13.3.2 Edge map

Now, let us finish up by considering the internals of the edge-map operation. In this implementation, we are going to use the following sentinel value to represent empty cells in sparse arrays of vertex identifiers.

```
const vtxid_type not_a_vertexid = -1l;
```

Example 13.8 Sparse-array representation of a set of vertex identifiers

The following array represents a set of three valid vertex identifiers, with two positions in the array being empty.

```
{ 3, not_a_vertexid, 0, 1, not_a_vertexid }
```

Let us define two helper functions. The first one takes a sparse array of vertex identifiers and copies out the valid vertex identifiers.

```
sparray just_vertexids(const sparray& vs) {
  return filter([&] (vtxid_type v) { return v != not_a_vertexid; }, vs);
}
```

The other function takes a graph and an array of vertex identifiers and returns the array of the degrees of the vertex identifiers.

```
sparray get_out_degrees_of(const adjlist& graph, const sparray& vs) {
  return map([&] (vtxid_type v) { return graph.get_out_degree_of(v); }, vs);
}
```

At a high level, our solution is the following. First, we construct a sparse-array representation of the set of vertex ids that are to be returned by the edge map. Second, we construct a compact representation of the sparse the array and return the compacted result array. Aside from this sparse-array detail, the algorithm implemented here corresponds exactly to the high level description that we presented in the previous section. That is, the outer loop processes the vertex ids from the frontier of the previous level and the inner loop processes the neighbors of each vertex in the frontier of the previous level, adding newly visited neighbors to the result set.

```
loop_controller_type process_out_edges_contr("process_out_edges");
loop_controller_type edge_map_contr("edge_map");

sparray edge_map(const adjlist& graph, std::atomic<bool>* visited, const sparray&  ↩
    in_frontier) {
  scan_excl_result offsets = prefix_sums_excl(get_out_degrees_of(graph, in_frontier));
  long m = in_frontier.size();
  long n = offsets.total;
  auto weight = [&] (long lo, long hi) {
    long u = (hi == m) ? n : offsets.partials[hi];
    return u - offsets.partials[lo];
  };
  sparray out_frontier = sparray(n);
  parallel_for(edge_map_contr, weight, 0l, m, [&] (long i) {
    vtxid_type src = in_frontier[i];
    long offset = offsets.partials[i];
    neighbor_list out_edges = graph.get_out_edges_of(src);
    long degree = graph.get_out_degree_of(src);
    parallel_for(process_out_edges_contr, 0l, degree, [&] (long j) {
      long dst = out_edges[j];
      bool orig = false;
      if (visited[dst].load() || ! visited[dst].compare_exchange_strong(orig, true))
        dst = not_a_vertexid;
      out_frontier[offset+j] = dst;
```

```
    });
  });
  return just_vertexids(out_frontier);
}
```

The complexity function used by the outer loop in the edge map is interesting because the complexity function treats the vertices in the frontier as weighted items. In particular, each vertex is weighted by its out degree in the graph. The reason that we use such weighting is because the amount of work involved in processing that vertex is proportional to its out degree. We cannot treat the out degree as a constant, unfortunately, because the out degree of any given vertex is unbounded, in general. As such, it should be clear why we need to account for the out degrees explicitly in the complexity function of the outer loop.

Question

What changes you need to make to BFS to have BFS annotate each vertex v by the length of the shortest path between v and the source vertex?

13.3.3 Performance analysis

Our parallel BFS is asymptotically work efficient: the BFS takes work $O(n+m)$. To establish this bound, we need to assume that the compare-and-exchange operation takes constant time. After that, confirming the bound is only a matter of inspecting the code line by line. On the other hand, the span is more interesting.

Question

What is the span of our parallel BFS?

Tip

In order to answer this question, we need to know first about the graph **diameter**. The diameter of a graph is the length of the shortest path between the two most distant vertices. It should be clear that the number of iterations performed by the while loop of the BFS is at most the same as the diameter.

13.3.4 Beyond BFS

Using the primitives that are similar to the ones that we presented here, in their paper on the Ligra library, Shun and Blelloch implemented several other graph algorithms, such as:

- Page Rank
- Connected components
- Betweenness centrality
- Radii estimation
- Bellman-Ford

Question

- After looking at Shun and Blelloch's paper, do you see a way to extend our code to implement the algorithms that they present in their paper.
- Can you think of other graph algorithms that can be parallelized in a similar fashion, using the Ligra primitives?