Parallel Computing: Theory and Practice

COLLABORATORS			
	TITLE : Parallel Computing: Theory and Practice		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		April 6, 2016	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

List of Figures

Preface

The goal of this book is to cover the fundamental concepts of parallel computing, including models of computation, parallel algorithms, and techniques for implementing and evaluating parallel algorithms.

Our primany focus will be hardware-shared memory parallelism.

For implementations, we use a C++ library, called **PASL**, which we have been developing over the past 5 years. PASL stands for Parallel Algorithm Scheduling Library. It also sounds a bit like the French phrase "pas seul" (pronounced "pa-sole"), meaning "not alone".

The library provides several scheduling algorithms for executing parallel programs on modern multicores and provides a range of utilities for inspecting the empirical behavior of parallel programs. We expect that the instructions in this book to allow the reader 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, which might otherwise be necessary.

All code that discussed in this book can be found at the Github repository linked by the following URL:

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

This code-base includes the examples presented in the book, see file `minicourse/examples.hpp`.

Some of the material in this book is based on the course, 15-210, co-taught with Guy Blelloch at CMU. The interested reader can find more details on this material in this book.

Starting point for this book was this book on PASL.

v1.0 2016-01

1 Administrative Matters

Course combines theory and practice. We will try ask the following two questions.

1. Does it work in practice?
2. Does it work in theory?

As you know this is a graduate class. This means that I don't care about your grade. But if you are sitting here, you might as well work for an A.

Grading will be based on some assignments, one midterm exam, and one project. We shall make time so that you can devote a good chunk of your semester to the project. You will also be receive a participation score, which amounts to 20\% of the grade.

For each lecture, I will make some notes and we shall make them available for you to comment, perhaps via a github repository.

2 Preliminaries

2.1 Processors, Processes, and Threads

We assume a machine model that consists of a shared memory by a number of processors, usually written as P. The processors have access to a shared memory, which is readable and writable by all processors.

We assume that an operating system or a that allows us to create ***processes***. The kernel schedules processes on the available processors in a way that is mostly out of our control with one exception: the kernel allows us to create any number of processes and ***pin*** them on the available processors as long as no more than one process is pinned on a processor.

We define a ***thread*** to be a piece of sequential computation whose boundaries, i.e., its start and end points, are defined on a case by case basis, usually based on the programming model. In reality, there different notions of threads. For example, a system-level thread is created by a call to the kernel and scheduled by the kernel much like a process. A user-level thread is created by the application program and is scheduled by the application—user level threads are invisible to the kernel. Common property of all threads is that they perform a sequential computation. In this class, we will usually talk about user-level threads. In the literature, you will encounter many different terms for a user-level thread, such as "fiber", "sparc", "strand", etc.

For our purposes in this book an ***application***, a piece of runnable software, can only create threads but no processes. We will assume that we can assign to an application any number of processes to be used for execution. If an application is run all by itself (without any other application running at the same time) and if all of its processes are pinned, then we refer to such an execution as occurring in the ***dedicated mode***.

Note
For now, we leave the details of the memory consistency model unspecified.

2.2 C++ Background

This book 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.

2.2.1 Template metaprogramming

Templates are C++'s way of providing for parametric polymorphism, which allows using the same code at multiple types. For example, in modern functional languages such as the ML family or Haskell, you can write a function $\lambda\ x.x$ as an identity function that returns its argument for any type of x. You don't have to write the function at every type that you plan to apply. Since functional languages such as ML and Haskell rely on type inference and have powerful type systems, they can infer from your code the most general type (within the constraints of the type system). For example, the function $\lambda\ x.x$ can be given the type $\forall\alpha.\alpha \rightarrow \alpha$. This type says that the function works for any type α and given an argument of type α, it returns a value of type α.

C++ provides for polymorphism with ***templates***. In its most basic form, a template is a class declaration or a function declaration, which is explicitly stated to be polymorphic, by making explicit the type variable. Since C++ does not in general perform type inference (in a rigorous sense of the word), it requires some help from the programmer.

For example, the following code below defines an array class that is parametric in the type of its elements. The declaration `template <class T>` says that the declaration of `class array`, which follows is parameterized by the identifier `T`. The definition of `class array` then uses `T` as a type variable. For example, the array defines a pointer to element sequences of type `T`, and the `sub` function returns an element of type `T` etc.

```
template <class T>
class array {
  public:
   array (int size) {a = new T[size];}
   T sub (int i) { a[i];}

  private:
    *T a;
}
```

Note that the only part of the syntax `template <class T>` that is changeable is the identifier `T`. In other words, you should think of the syntax `template <class T>` as a binding form that allows you to pick an identifier (in this case `T`). You might ask why the type identifier/variable `T` is a `class`. This is a good question. The authors find it most helpful to not think much about such questions, especially in the context of the C++ language.

Once defined a template class can be initialized with different type variables by using the `< >` syntax. For examples, we can define different arrays such as the following.

```
array<int> myFavoriteNumbers(7);
array<char*> myFavoriteNames(7);
```

Again, since C++ does not perform type inference for class instances, the C++ compiler expects the programmer to eliminate explicitly parametricity by specifying the argument type.

It is also possible to define polymorphic or generic functions. For example, the following declarations defines a generic identity function.

```
template <class T>
T identity(T x) { return x;}
```

Once defined, this function can be used without explicitly specializing it at various types. In contrast to templated classes, C++ does provide some type inference for calls to templated functions. So generic functions can be specialized implicitly, as shown in the examples below.

```
i = identity (3)
s = identity ("template programming can be ugly")
```

This brief summary of templates should suffice for the purposes of the material covered in this book. Templates are covered in significant detail by many books, blogs, and discussions boards. We refer the interested reader to those sources for further information.

2.2.2 Lambda expressions

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

3 Introduction

This class is motivated by recent advances in architecture that put sequential hardware on the path to extinction. Due to fundamental architectural limitations, sequential performance of processors have not been increasing since 2005. In fact performance has been decreasing by some measures because hardware manufacturer's have been simplifying processorns by simplifying the handling of instruction level parallelism.

Moore's law, however, continues to be holding, at least for the time being, enabling hardware manufacturers to squeeze an increasing number of transistors into the same chip area. The result, not surprisingly, has been increased paralellelism, more processors that is. In particular, manufacturers have been producing multicore chips where each chip consists of a number of processors that fit snugly into a small area, making communication between them fast and efficient. You can read more about the history of modern multicore computers.

This simple change in hardware has led to dramatic changes in computing. Parallel computing, once a niche domain for computational scientists, is now an everyday reality. Essentially all computing media ranging from mobile phones to laptops and computers operate on parallel computers.

This change was anticipated by computers scientists. There was in fact much work on parallel algorithms in 80's and 90's. The models of computation assumed then turned out to be unrealistic. This makes it somewhat of a challenge to use the algorithms from that era. Some of the ideas, however, transcends those earlier models can still be used today to design and implement parallel algorithms on modern architectures.

The goal of this class is to give an introduction to the theory and the practice of parallol computing. Specifically, we will cover the following topics.

1. Multithreaded computation
2. Work and span
3. Offline scheduling
4. Structured or implicit parallel computation
    a. Fork-join, async-finish, nested parallelism.
    b. Futures.
    c. Parallelism versus concurrency
5. Parallel algorithms for sequences
6. Online scheduling: work-stealing algorithms and its analysis
7. Parallel algorithms for trees
8. Parallel algorithms for graphs
9. Concurrency

4 Chapter: Multithreading, Parallelism, and Concurrency

The term ***multithreading*** refers to computing with multiple threads of control. Once created, a thread performs a computation by executing a sequence of instructions, as specified by the program, until it terminates. A multithreaded computation starts by executing a ***main thread***, which is the thread at which the execution starts. A thread can create or ***spawn*** another thread and ***synchronize*** with other threads by using a variety of synchronization constructs such as locks, mutex's, synchronization variables, and semaphores.

4.1 DAG Representation

A multithreaded computation can be represented by a dag, a Directed Acyclic Graph, or written also more simply a ***dag***, of vertices. The figure below show an example multithreaded computation and its dag. Each vertex represents the execution of an ***instruction***, such as an addition, a multiplication, a memory operation, a (thread) spawn operation, or a synchronization operation. A vertex representing a spawn operation has outdegree two. A synchronization operation waits for an operation belonging to a thread to complete, and thus a vertex representing a synchronization operation has indegree two. Recall that a dag represents a partial order. Thus the dag of the computation represents the partial ordering of the dependencies between the instructions in the computation.

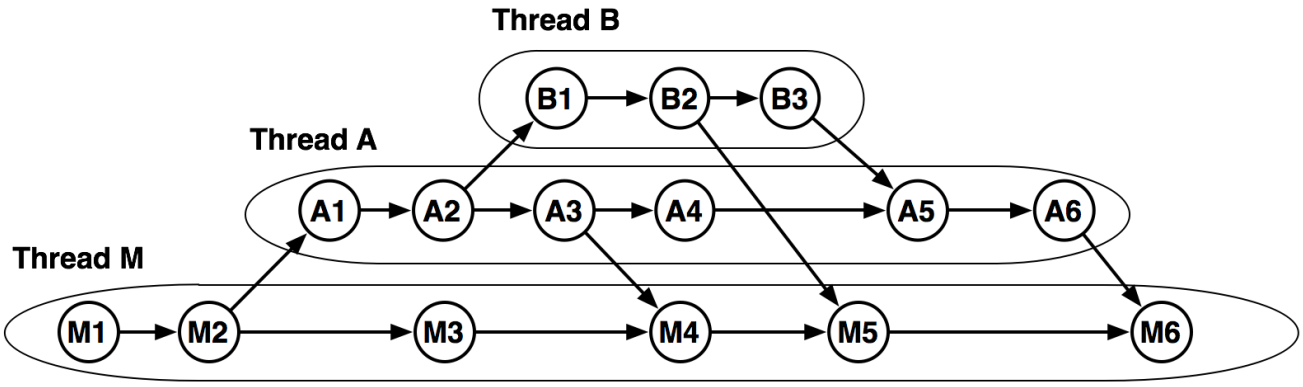


Figure 1: A multithreaded computation.

Throughout this book, we make two assumptions about the structure of the dag:

1. Each vertex has outdegree at most two.
2. The dag has exactly one ***root vertex*** with indegree zero and one ***final vertex*** vertex with outdegree zero. The root is the first instruction of the ***root thread***.

The outdegree assumption naturally follows by the fact that each vertex represents an instruction, which can create at most one thread.

4.2 Cost Model: Work and Span

For analyzing the efficiency and performance of multithreaded programs, we use several cost measures, the most important ones include work and span. We define the ***work*** of a computation as the number of vertices in the dag and the ***span*** as the length of the longest path in the dag. In the example dag above, work is 15 and span is 9.

4.3 Execution and Scheduling

The execution of a multithreaded computation executes the vertices in the dag of the computation in some partial order that is consistent with the partial order specified by the dag, that is, if vertices u, v are ordered then the execution orders them in the same way.

Multithreaded programs are executed by using a ***scheduler*** that assigns vertices of the dag to processes.

Definition: Execution Schedule

Given a dag G, an (execution) schedule for G is a function from processes and (time) ***steps*** to instructions of the dag such that

1. if a vertex u is ordered before another v in G, then v is not executed at a time step before u, and
2. each vertex in G is executed exactly once.

The ***length*** of a schedule is the number of steps in the schedule.

The first condition ensures that a schedule observes the dependencies in the dag. Specifically, for each arc (u, v) in the dag, the vertex u is executed before vertex v.

For any step in the execution, we call a vertex ***ready*** if all the ancestors of the vertex in the dag are executed prior to that step. Similarly, we say that a thread is ready if it contains a ready vertex. Note that a thread can contain only one ready vertex at any time.

Example 4.1 Schedule

An example schedule with 3 processes. The length of this schedule is 10

Time Step	Process 1	Process 2	Process 3
1	M1		
2	M2		
3	M3	A1	
4		A2	
5	B1	A3	
6	B2	A4	
7	B2		M4
8		A5	M5
9	A6		
10		M6	

Fact: Scheduling Invariant

Consider a computation dag G and consider an execution using any scheduling algorithm. At any time during the execution, color the vertices that are executed as blue and the others as red.

1. The blue vertices induce a blue sub-dag of G that is connected and that has the same root as G.
2. The red vertices incude a red sub-dag of G that is connected.
3. All the vertices of G are in the blue or the red sub-dag. In other words, the blue and red vertices partitions the dag into two sub-dags.

4.4 Scheduling Lower Bounds

Theorem: Lower bounds

Consider any multithreaded computation with work W and span S and P processes. The following lower bounds hold.

1. Every execution schedule has length at least $\frac{W}{P}$.
2. Every execution schedule has length at least S.

The first lower bound follows by the simple observation that a schedule can only execute P instructions at a time. Since all vertices must be executed, a schedule has length at least $\frac{W}{P}$. The second lower bound follows by the observation that the schedule cannot execute a vertex before its ancestors and thus the length of the schedule is at least as long as any path in the dag, which can be as large as the span S.

4.5 Offline Scheduling

Having established a lower bound, we now move on to establish an upper bound for the ***offline scheduling problem***, where we are given a dag and wish to find an execution schedule that minimizes the run time. It is known that the related decision problem in NP-complete but that 2-approximation is relatively easy. We shall consider two distinct schedulers: ***level-by-level scheduler*** and ***greedy scheduler***.

A ***level-by-level schedule*** is a schedule that executes the instructions in a given dag level order, where the ***level*** of a vertex is the longest distance from the root of the dag to the vertex. More specifically, the vertices in level 0 are executed first, followed by the vertices in level 1 and so on.

Theorem:[Offline level-by-level schedule]

For a dag with work W and span S, the length of a level-by-level schedule is $W/P+S$.

Proof

Let W_i denote the work of the instructions at level i. These instructions can be executed in $\lceil \frac{W_i}{P} \rceil$ steps. Thus the total time is

$$\sum_{i=1}^{S} \lceil \frac{W_i}{P} \rceil \leq \sum_{i=1}^{S} \lfloor \frac{W_i}{P} \rfloor + 1 \leq \lfloor \frac{W}{P} \rfloor + S$$

This theorem called Brent's theorem was proved by Brent in 1974. It shows that the lower bound can be approximated within a factor of 2.

Brent's theorem has later been generalized to all greedy schedules. A ***greedy schedule*** is a schedule that never leaves a process idle unless there are no ready vertices. In other words, greedy schedules keep processes as busy as possibly by greedily assigning ready vertices.

Theorem: Offline Greedy Schedule

Consider a dag G with work W and span S. Any greedy P-process schedule has length at most $\frac{W}{P} + S \cdot \frac{P-1}{P}$.

Proof

Consider any greedy schedule with length T.

For each step $1 \leq i \leq T$, and for each process that is scheduled at that step, collect a token. The token goes to the ***work bucket*** if the process executes a vertex in that step, otherwise the process is idle and the token goes to an ***idle bucket***.

Since each token in the work bucket corresponds to an executed vertex, there are exactly W tokens in that bucket.

We will now bound the tokens in the idle bucket by $S \cdot (P-1)$. Observe that at any step in the execution schedule, there is a ready vertex to be executed (because otherwise the execution is complete). This means that at each step, at most $P-1$ processes can contribute to the idle bucket. Furthermore at each step where there is at least one idle process, we know that the number of ready vertices is less than the number of available processes. Note now that at that step, all the ready vertices have no incoming edges in the red sub-dag consisting of the vertices that are not yet executed, and all the vertices that have no incoming edges in the red sub-dag are ready. Thus executing all the ready vertices at the step reduces the length of all the paths that originate at these vertices and end at the final vertex by one. This means that the span of the red sub-dag is reduced by one because all paths with length equal to span must originate in a ready vertex. Since the red-subdag is initially equal to the dag G, its span is S, and thus there are at most S steps at which a process is idle. As a result the total number of tokens in the idle bucket is at most $S \cdot (P-1)$.

Since we collect P tokens in each step, the bound thus follows.

Exercise

Show that the bounds for Brent's level-by-level scheduler and for any greedy scheduler is within a factor 2 of optimal.

4.6 Online Scheduling

In offline scheduling, we are given a dag and are interested in finding a schedule with minimal length. When executing multi-threaded program, however, we don't have full knowledge of the dag. Instead, the dag unfolds as we run the program. Furthermore, we are interested in not minimizing the length of the schedule but also the work and time it takes to compute the schedule. These two additional conditions define the ***online scheduling problem.***

An online scheduler or a simply a ***scheduler*** is an algorithm that solves the online scheduling problem by mapping threads to available processes. For example, if only one processor is available, a scheduler can map all threads to that one processor. If two processors are available, then 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 by ***load balancing***.

There are many different online-scheduling algorithms but these algorithms all operate similarly. We can outline a typical scheduling algorithm as follows.

Typical Online Scheduling Algorithm

The algorithm maintains a ***work pool*** of work, consisting of ready threads, and executes them. Execution starts with the root thread in the pool. It ends when the final vertex is executed. In order to minimize the cost of computing the schedule, the algorithm executes a thread until there is a need for synchronization with other threads.

To obtain work, a process removes a thread from the pool and executes its ready vertex. We refer to the thread executed by a process as the ***assigned thread***. When executed, the ready vertex can make the next vertex of the thread ready, which then also gets executed an so on until one of the following ***synchronization*** actions occur.

1. ***Die:*** The process executes last vertex of the thread, causing the thread to die. The process then obtains other work.
2. ***Block:*** The assigned vertex executes but the next vertex does not become ready. This blocks the thread and thus the process obtains other work.
3. ***Enable:*** The assigned vertex makes ready the continuation of the vertex and unblocks another previously blocked thread by making a vertex from that thread ready. In this case, the process inserts both (any) one thread into the work pool and continues to execute the other.
4. ***Spawn:*** The assigned vertex spaws another thread. As in the previous case, the process inserts one thread into the work pool and continues to execute the other.

These actions are not mutually exclusive. For example, a thread may spawn/enable a thread and die. In this case, the process performs the corresponding steps for each action.

Exercise: Scheduling Invariant

Convince yourself that the scheduling invariant holds in online scheduling.

For a given schedule generated by an online scheduling algorithm, we can define a tree of vertices, which tell us far a vertex, the vertex that enabled it.

Definition: Enabling Tree

Consider the execution of a dag. If the execution of a vertex u enables another vertex v, then we call the edge (u,v) an ***enabling edge*** and we call u the ***enabling parent*** of v. For simplicity, we simply use the term ***parent*** instead of enabling parent.

Note that any vertex other than the root vertex has one enabling parent. Thus the subgraph induced by the enabling edges is a rooted tree that we call the ***enabling tree***.

Example 4.2 Scheduler with a global thread queue.

We can give a simple greedy scheduler by using a queue of threads. At the start of the execution, the scheduler places the root thread into the queue and then repeats the following step until the queue becomes empty: for each idle process, take the thread at the front of the queue and assign it to the processor, let each processor run for one step, if at the end of the step, there are new ready threads, then insert them onto the tail of the queue.

The centralized scheduler with the global thread queue is a greedy scheduler that generates a greedy schedule under the assumption that the queue operations take zero time and that the dag is given. This algorithm, however, does not work well for online scheduling the operations on the queue take time. In fact, since the thread queue is global, the algorithm can only insert and remove one thread at a time. For this reason, centralized schedulers do not scale beyond a handful of processors.

Definition: Scheduling friction.

No matter how efficient a scheduler is there is real cost to creating threads, inserting and deleting them from queues, and to performing load balancing. We refer to these costs cumulatively as ***scheduling friction***, or simply as ***friction***.

There has been much research on the problem of reducing friction in scheduling. This research shows that distrubuted scheduling algorithms can work quite well. In a distributed algorithm, each processor has its own queue and primarily operates on its own queue. A load-balancing technique is then used to balance the load among the existing processors by redistributing threads, usually on a needs basis. This strategy ensures that processors can operate in parallel to obtain work from their queues.

A specific kind of distributed scheduling technique that can leads to schedules that are close to optimal is ***work stealing*** schedulers. In a work-stealing scheduler, processors work on their own queues as long as their is work in them, and if not, go "steal" work from other processors by removing the thread at the tail end of the queue. It has been proven that randomized work-stealing algorithm, where idle processors randomly select processors to steal from, deliver close to optimal schedules in expectation (in fact with high probability) and furthermore incur minimal friction. Randomized schedulers can also be implemented efficiently in practice. PASL uses an scheduling algorithm that is based on work stealing. We consider work-stealing in greater detail in a future chapter.

4.7 Writing Multithreaded Programs: Pthreads

Multithreaded programs can be written using a variety of language abstractions interfaces. One of the most widely used interfaces is the ***POSIX Threads* or *Pthreads*** interface, which specifies a programming interface for a standardized C language in the IEEE POSIX 1003.1c standard. Pthreads provide a rich interface that enable the programmer to create multiple threads of control that can synchronize by using the nearly the whole range of the synchronization facilities mentioned above.

Hello world with Pthreads An example Pthread program is shown below. The main thread (executing function `main`) creates 8 child threads and terminates. Each child in turn runs the function `helloWorld` and immediately terminates. Since the main thread does not wait for the children to terminate, it may terminate before the children does, depending on how threads are scheduled on the available processors.

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>

using namespace std;

#define NTHREADS 8

void *helloWorld(void *threadid)
{
   long tid;
   tid = (long)threadid;
   cout << "Hello world! It is me, 00" << tid << endl;
   pthread_exit(NULL);
}

int main ()
{
   pthread_t threads[NTHREADS];
   int rc;
   int i;
   for( i=0; i < NTHREADS; i++ ){
      cout << "main: creating thread 00" << i << endl;
      error = pthread_create(&threads[i], NULL, helloWorld, (void *) i);
      if (error) {
         cout << "Error: unable to create thread," << error << endl;
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

When executed this program may print the following.

```
main: creating thread 000
```

```
main: creating thread 001
main: creating thread 002
main: creating thread 003
main: creating thread 004
main: creating thread 005
main: creating thread 006
main: creating thread 007
Hello world! It is me, 000
Hello world! It is me, 001
Hello world! It is me, 002
Hello world! It is me, 003
Hello world! It is me, 004
Hello world! It is me, 005
Hello world! It is me, 006
Hello world! It is me, 007
```

But that would be unlikely, a more likely output would look like this:

```
main: creating thread 000
main: creating thread 001
main: creating thread 002
main: creating thread 003
main: creating thread 004
main: creating thread 005
main: creating thread 006
main: creating thread 007
Hello world! It is me, 000
Hello world! It is me, 001
Hello world! It is me, 006
Hello world! It is me, 003
Hello world! It is me, 002
Hello world! It is me, 005
Hello world! It is me, 004
Hello world! It is me, 007
```

And may even look like this

```
main: creating thread 000
main: creating thread 001
main: creating thread 002
main: creating thread 003
Hello world! It is me, 000
Hello world! It is me, 001
Hello world! It is me, 003
Hello world! It is me, 002
main: creating thread 004
main: creating thread 005
main: creating thread 006
main: creating thread 007
Hello world! It is me, 006
Hello world! It is me, 005
Hello world! It is me, 004
Hello world! It is me, 007
```

4.8 Writing Multithreaded Programs: Structured or Implicit Multithreading

Interface such as Pthreads enable the programmer to create a wide variety of multithreaded computations that can be structured in many different ways. Large classes of interesting multithreaded computations, however, can be expcessed using a more structured approach, where threads are restricted in the way that they synchronize with other threads. One such interesting class of computations is fork-join computations where a thread can spawn or "fork" another thread or "join" with another existing

thread. Joining a thread is the only mechanism through which threads synchronize. The figure below illustrates a fork-join computation. The main threads forks thread A, which then spaws thread B. Thread B then joins thread A, which then joins Thread M.

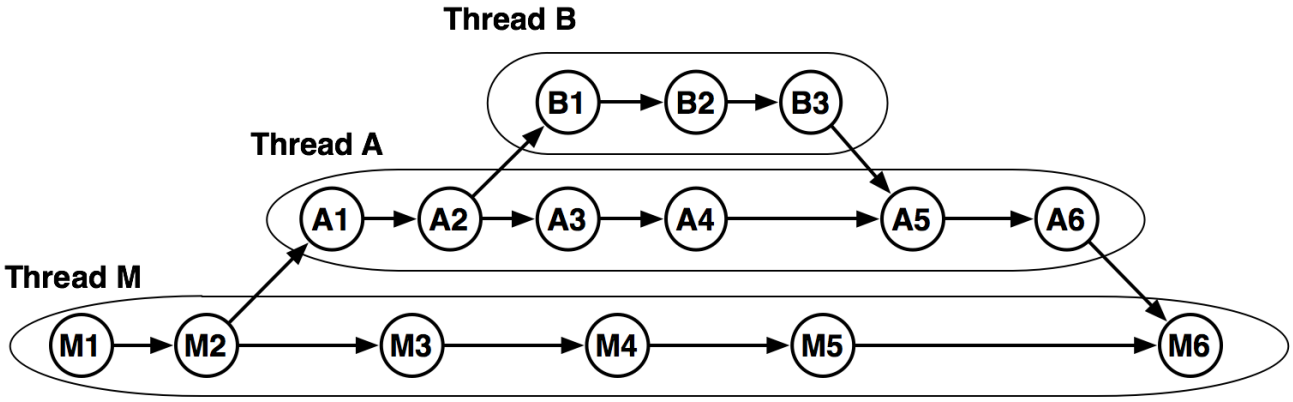


Figure 2: A multithreaded fork-join computation.

In addition to fork-join, there are other interfaces for structured multithreading such as async-finish, and futures. These interfaces are adopted in many programming languages: the Cilk language is primarily based on fork-join but also has some limited support for async-finish; X10 language is primarily based on async-finish but also supports futures; the Haskell language Haskell language provides support for fork-join and futures as well as others; Parallel ML language as implemented by the Manticore project is primarily based on fork-join parallelism. Such languages are sometimes called ***implicitly parallel***.

The class computations that can be expressed as fork-join and async-finish programs are sometimes called ***nested parallel***. The term "nested" refers to the fact that a parallel computation can be nested within another parallel computation. This is as opposed to ***flat parallelism*** where a parallel computation can only perform sequential computations in parallel. Flat parallelism used to be common technique in the past but becoming increasingly less prominent.

4.9 Parallelism versus concurrency

Structured multithreading offers important benefits both in terms of efficiency and expressiveness. Using programming constructs such as fork-join and futures, it is usually possible to write parallel programs such that the program accepts a "sequential semantics" but executes in parallel. The sequential semantics enables the programmer to treat the program as a serial program for the purposes of correctness. A run-time system then creates threads as necessary to execute the program in parallel. This approach offers is some ways the best of both worlds: the programmer can reason about correctness sequentially but the program executes in parallel. The benefit of structured multithreading in terms of efficiency stems from the fact that threads are restricted in the way that they communicate. This makes it possible to implement an efficient run-time system.

More precisely, consider some sequential language such as the untyped (pure) lambda calculus and its sequential dynamic semantics specified as a strict, small step transition relation. We can extend this language with the structured multithreading by enriching the syntax language with "fork-join" and "futures" constructs. We can now extend the dynamic semantics of the language in two ways: 1) trivially ignore these constructs and execute serially as usual, and 2) execute in parallel by creating parallel threads. We can then show that these two semantics are in fact identical, i.e., that they produce the same value for the same expressions. In other words, we can extend a rich programming language with fork-join and futures and still give the language a sequential semantics. This shows that structured multithreading is nothing but an efficiency and performance concern; it can be ignored from the perspective of correctness.

We use the term ***parallelism*** to refer to the idea of computing in parallel by using such structured multithreading constructs. As we shall see, we can write parallel algorithms for many interesting problems. While parallel algorithms or applications constitute a large class, they don't cover all applications. Specifically applications that can be expressed by using richer forms of multithreading such as the one offered by Pthreads do not always accept a sequential semantics. In such ***concurrent*** applications, threads can communicate and coordinate in complex ways to accomplish the intended result. A classic concurrency example is the "producer-consumer problem", where a consumer and a producer thread coordinate by using a fixed size buffer of items. The

producer fills the buffer with items and the consumer removes items from the buffer and they coordinate to make sure that the buffer is never filled more than it can take. We can use operating-system level processes instead of threads to implement similar concurrent applications.

In summary, parallelism is a property of the hardware or the software platform where the computation takes place, whereas concurrency is a property of the application. Pure parallelism can be ignored for the purposes of correctness; concurrency cannot be ignored for understanding the behavior of the program.

Parallelism and concurrency are orthogonal dimensions in the space of all applications. Some applications are concurrent, some are not. Many concurrent applications can benefit from parallelism. For example, a browser, which is a concurrent application itself as it may use a parallel algorithm to perform certain tasks. On the other hand, there is often no need to add concurrency to a parallel application, because this unnecessarily complicates software. It can, however, lead to improvements in efficiency.

The following quote from Dijkstra suggest pursuing the approach of making parallelism just a matter of execution (not one of semantics), which is the goal of the much of the work on the development of programming languages today. Note that in this particular quote, Dijkstra does not mention that parallel algorithm design requires thinking carefully about parallelism, which is one aspect where parallel and serial computations differ.

> From the past terms such as "sequential programming" and "parallel programming" are still with us, and we should try to get rid of them, for they are a great source of confusion. They date from the period that it was the purpose of our programs to instruct our machines, now it is the purpose of the machines to execute our programs. Whether the machine does so sequentially, one thing at a time, or with considerable amount of concurrency, is a matter of implementation, and should *not* be regarded as a property of the programming language.
>
> — Edsger W. Dijkstra *Selected Writings on Computing: A Personal Perspective (EWD 508)*

5 Chapter: Fork-join parallelism

Fork-join parallelism, a fundamental model in parallel computing, dates back to 1963 and has since been widely used in parallel computing. In fork join parallelism, 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.

As we will see, it is often possible to extend an existing language with support for fork-join parallelism by providing libraries or compiler extensions that support a few simple primitives. Such extensions to a language make it easy to derive a sequential program from a parallel program by syntactically substituting the parallelism annotations with corresponding serial annotations. This in turn enables reasoning about the semantics or the meaning of parallel programs by essentially "ignoring" parallelism.

PASL is a C++ library that enables writing implicitly parallel programs. 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 5.1 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 scheduling algorithms that the PASL uses to handle such load balancing and synchronization duties.

In fork-join programs, a thread is a sequence of instructions that do not contain calls to `fork2()`. A thread is essentially a piece of sequential computation. 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.

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 two branches of `fork2` are committed to memory before the join point is scheduled. The PASL runtime guarantees this property by using a local barrier. Such barriers are efficient, because they involve just a single dynamic synchronization point between at most two processors.

Example 5.2 Writes and the join statement

In the example below, both writes into `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.

5.1 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. This function for computing the Fibonacci numbers is inefficient because the algorithm takes exponential time, whereas there exist dynamic programming solutions that take linear time.

```
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;
}
```

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;
}
```

5.2 Incrementing an array, in parallel

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 5.3 Example: Using `map_incr`.

The code below illustrates an 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
```

This is not a good parallel algorithm but it is not difficult to give a parallel algorithm for incrementing an array. The code for such an algorithm is given below.

```
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);
    });
  }
}
```

It is easy to see that this algorithm has $O(n)$ work and $O(\log n)$ span.

5.3 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 and 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`.

The sequential elision is often useful for debugging and for optimization. It is useful for debugging because it is usually easier to find bugs in sequential runs of parallel code than in parallel runs of the same code. It is useful in optimization because the sequentialized code helps us to isolate the purely algorithmic overheads that are introduced by parallelism. By isolating these costs, we can more effectively pinpoint inefficiencies in our code.

5.4 Executing fork-join algorithms

We defined fork-join programs as a subclass case of multithreaded programs. Let's see more precisely how we can "map" a fork-join program to a multithreaded program. An our running example, let's use the `map_incr_rec`, whose code is reproduced below.

```
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);
    });
  }
}
```

Since, a fork-join program does not explicitly manipulate threads, it is not immediately clear what a thread refers to. To define threads, we can partition a fork-join computation into pieces of serial computations, each of which constitutes a thread. What we mean by a serial computation is a computation that runs serially and also that does not involve any synchronization with other threads except at the start and at the end. More specifically, for fork-join programs, we can define a piece of serial computation a ***thread***, if it executes without performing parallel operations (`fork2`) except perhaps as its last action. When partitioning the computation into threads, it is important for threads to be maximal; technically a thread can be as small as a single instruction.

Definition: Thread

A **thread** is a maximal computation consisting of a sequence of instructions that do not contain calls to `fork2()` except perhaps at the very end.

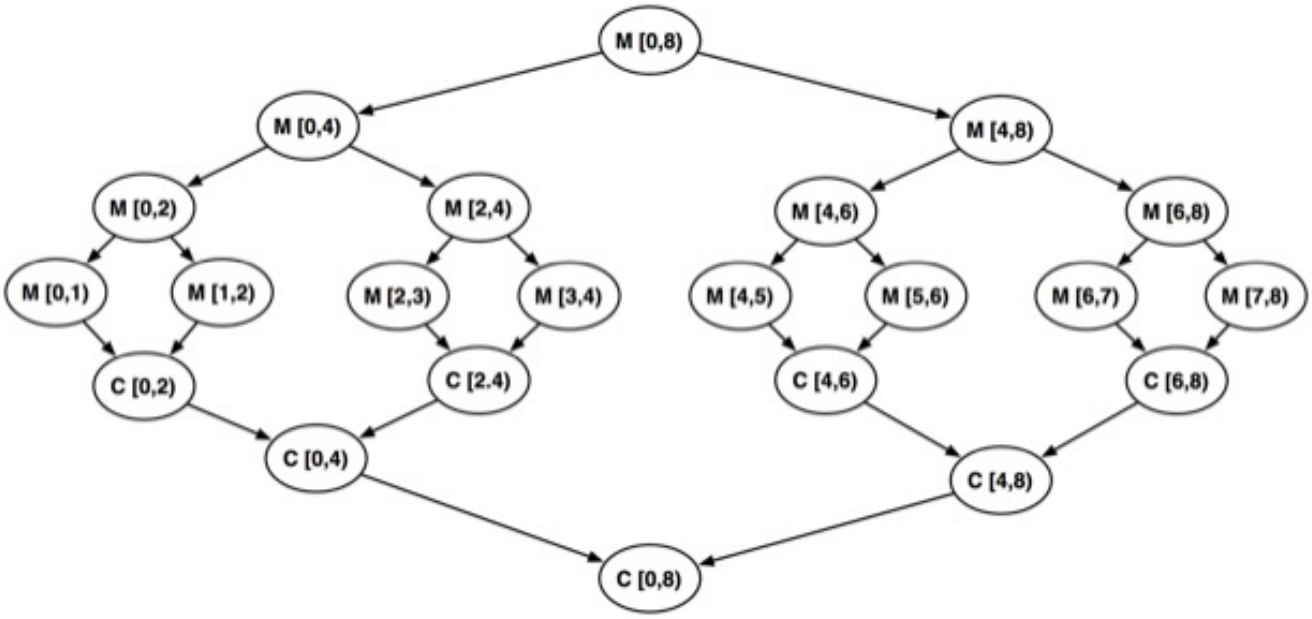


Figure 3: Dag for parallel increment on an array of 8: Each vertex corresponds a call to `map_inc_rec` excluding the `fork2` or the continuation of `fork2`, which is empty, an is annotated with the interval of the input array that it operates on (its argument).

The Figure above illustrates the dag for an execution of `map_incr_rec`. We partition each invocation of this function into two threads labeled by "M" and "C" respectively. The threads labeled by $M[i, j]$ corresponds to the part of the invocation of `map_incr_rec` with arguments `lo` and `hi` set to i and j respectively; this first part corresponds to the part of execution up and including the `fork2` or all of the function if this is a base case. The second corresponds to the "continuation" of the `fork2`, which is in this case includes no computation.

Based on this dag, we can create another dag, where each thread is replaced by the sequence of instructions that it represents. This would give us a picture similar to the dag we drew before for general multithreaded programs. Such a dag representation, where we represent each instruction by a vertex, gives us a direct way to calculate the work and span of the computation. If we want to calculate work and span ond the dag of threads, we can label each vertex with a weight that corresponds to the number of instruction in that thread.

Note

The term thread is very much overused in computer science. There are ***system threads***, which are threads that are known to the operating system and which can perform a variety of operations. For example, Pthreads library enables creating such system threads and programming with them. There are also many libraries for programming with ***user-level threads***, which are threads that exist at the application level but the operating system does not know about them. Then there are threads that are much more specific such as those that we have defined for the fork-join programs. Such threads can be mapped to system or user-level threads but since they are more specific, they are usually implemented in a custom fashion, usually in the user/application space. For this reason, some authors prefer to use a different term for such threads, e.g., ***spark***, ***strand***, ***task***.

Let's observe a few properties of fork-join computations and their dags.

1. The computation dag of a fork-join program applied to an input unfolds dynamically as the program executes. For example, when we run `map_inc_rec` with an input with n elements, the dag initially contains just the root vertex (thread) corresponding to the first call to `map_inc_rec` but it grows as the execution proceeds.

2. An execution of a fork-join program can generate a massive number of threads. For example, our 'map_inc_rec' function generates approximately $4n$ threads for an input with n elements.

3. The work/span of each thread can vary from a small amount to a very large amount depending on the algorithm. In our example, each thread performs either a conditional, sometimes an addition and a fork operation or performs no actual computation (continuation threads).

Suppose now we are given a computation dag and we wish to execute the dag by mapping each thread to one of the P processor that is available on the hardware. To do this, we can use an online scheduling algorithm.

Example 5.4 An example 2-processor schedule

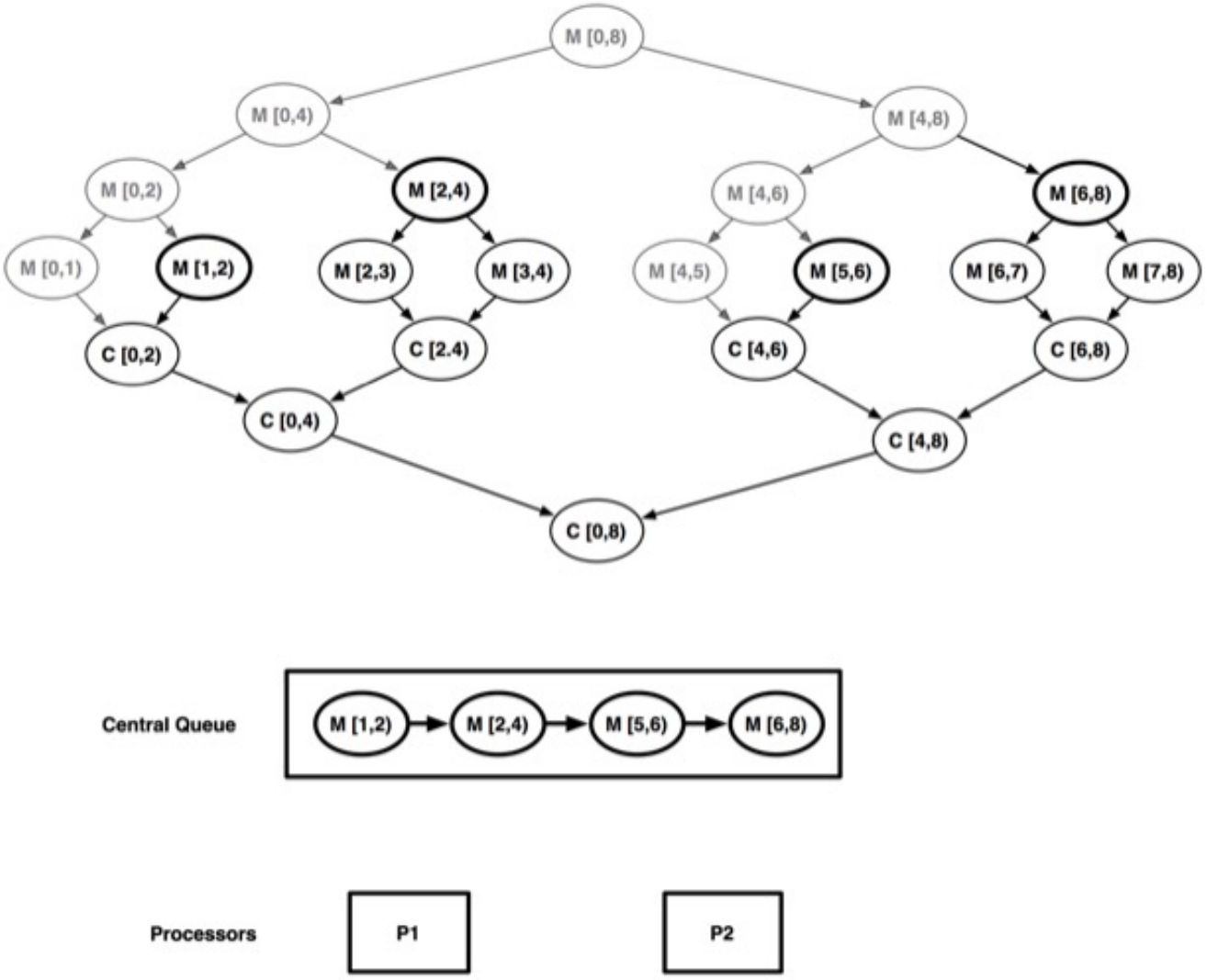
The following is a schedule for the dag shown in this Figure assuming that each thread takes unit time.

Time Step	Processor 1	Processor 2
1	M [0,8)	
2	M [0,4)	M [4,8)
3	M [0,2)	M [4,6)
4	M [0,1)	M [4,5)
5	M [1,2)	M [5,6)
6	C [0,2)	C [4,6)
7	M [2,4)	M [6,8)
8	M [2,3)	M [6,7)
9	M [3,4)	M [7,8)
10	C [2,4)	C [6,8)
11	C [0,4)	C [4,8)
12	C [0,8)	_

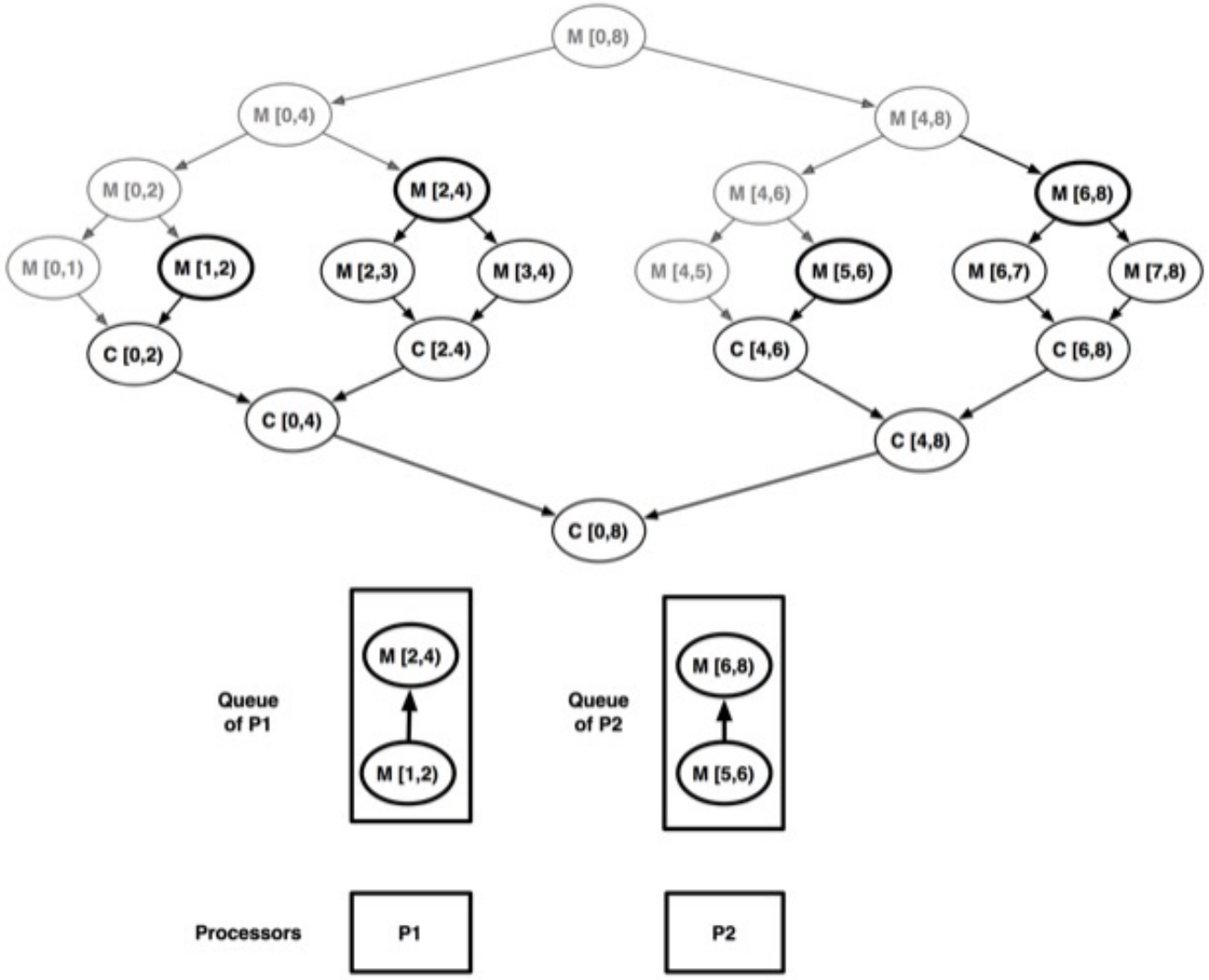
Exercise: Enabling Tree

Draw the enabling tree for the schedule above.

Example 5.5 Centralized scheduler illustrated: the state of the queue and the dag after step 4. Completed vertices are drawn in grey (shaded).



Example 5.6 Distributed scheduler illustrated: the state of the queue and the dag after step 4. Completed vertices are drawn in grey (shaded).



6 Chapter: Structured Parallelism with Async-Finish

The "async-finish" approach offers another mechanism for structured parallelism. It is similar to fork-join parallelism but is more flexible, because it allows essentially any number of parallel branches to synchronize at the same point called a "finish". Each "async" creates a separate parallel branch, which by construction must signal a "finish."

As with fork-join, it is often possible to extend an existing language with support for async-finish parallelism by providing libraries or compiler extensions that support a few simple primitives. Such extensions to a language make it easy to derive a sequential program from a parallel program by syntactically substituting the parallelism annotations with corresponding serial annotations. This in turn enables reasoning about the semantics or the meaning of parallel programs by essentially "ignoring" parallelism, e.g., via the sequential elision mechanism.

6.1 Parallel Fibonacci via Async-Finish

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

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

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 `async()`. It is a requirement that each `async` takes place in the context of a `finish`. All `async`'ed computations that takes place in the context of a `finish` synchronize at that `finish`, i.e., `finish` completes only when all the `async`'ed computations complete. The important point about a `finish` is that it can serve as the synchronization point for an arbitrary number of `async`'ed computations. This is not quite useful in Fibonacci because there are only two computations to synchronize at a time, but it will be useful in our next example.

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

6.2 Incrementing an array, in parallel

Recall our example for mapping 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;
}
```

Using async-finish, we can write to code parallel array increment by using only a single 'finish` block and `async`' ing all the parallel computations inside that `finish`.

```
void map_incr_rec_aux (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;
    async ([&] {
      map_incr_rec_aux(source, dest, lo, mid);
    };
    async [&] {
      map_incr_rec_aux(source, dest, mid, hi);
    };
  }
}


void map_incr_rec (const long* source, long* dest, long lo, long hi) {

  finish ([&] {
    map_inc_rec_aux (source, dest, lo, hi));
  };
}
```

It is helpful to compare this to fork-join, where we had to synchronize parallel branches in pairs. Here, we are able to synchronize them all at a single point.

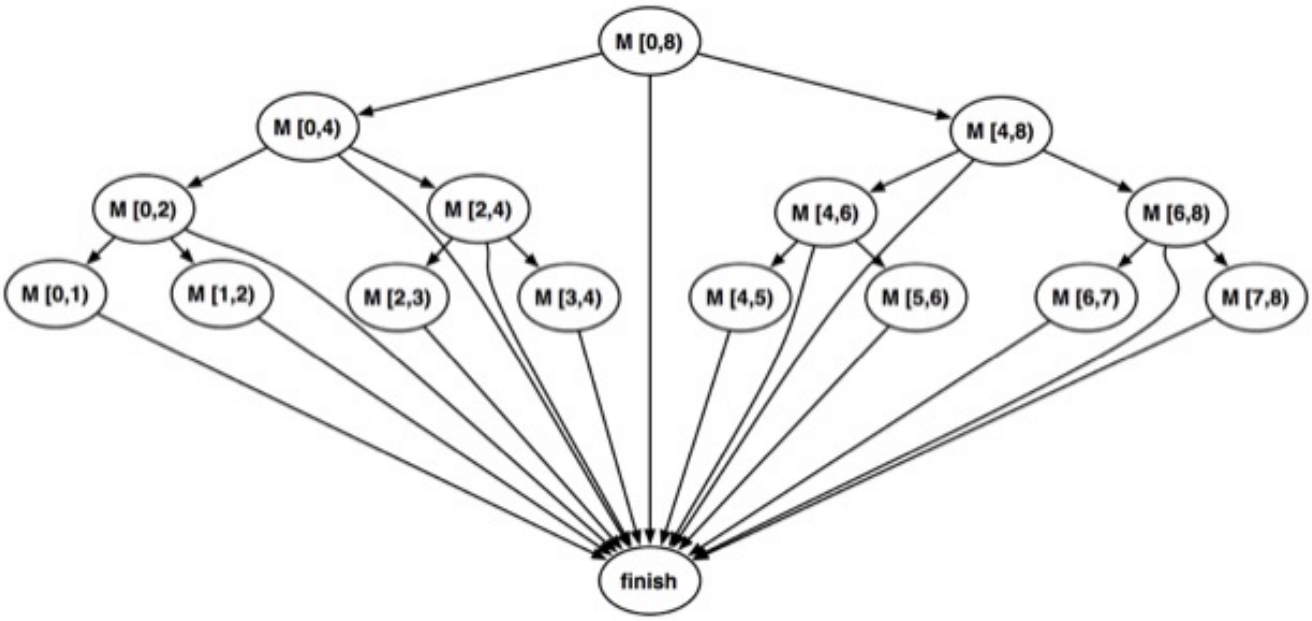


Figure 4: Dag for parallel increment on an array of 8 using async-finish: Each vertex corresponds a call to `map_inc_rec_aux` excluding the `async` or the continuation of `async`, which is empty, and is annotated with the interval of the input array that it operates on (its argument).

The Figure above illustrates the dag for an execution of `map_incr_rec`. Each invocation of this function corresponds to a thread labeled by "M". The threads labeled by $M[i, j]$ corresponds to the part of the invocation of `map_incr_rec_aux` with arguments `lo` and `hi` set to i and j respectively. Note that all threads synchronize at the single finish node.

Based on this dag, we can create another dag, where each thread is replaced by the sequence of instructions that it represents. This would give us a picture similar to the dag we drew before for general multithreaded programs. Such a dag representation, where we represent each instruction by a vertex, gives us a direct way to calculate the work and span of the computation. If we want to calculate work and span on the dag of threads, we can label each vertex with a weight that corresponds to the number of instruction in that thread.

Using async-finish does not alter the asymptotic work and span of the computation compared to fork-join, which remain as $O(n)$ work and $O(\log n)$ span. In practice, however, the async-finish version creates fewer threads (by about a factor of two), which can make a difference.

7 Chapter: Structured Parallelism with Futures

Futures were first used for expressing parallelism in the context of functional languages, because they permit a parallel computation to be a first-class value. Such a value can be treated just like any other value in the language. For example, it can be placed into a data structure, passed to other functions as arguments. Another crucial difference between futures and fork-join and async-finish parallelism is synchronization: in the latter synchronization is guided by "control dependencies", which are made apparent by the code. By inspecting the code we can identify the synchronization points for a piece of parallel computation, which correspond to `join` and `finish`. In futures, synchronization can be more complex because it is based on "data dependencies." When a parallel computation is needed, the programmer can demand computation to be completed. If the computation has not completed, then it will be executed to completion. If it has completed, its result will be used. Using futures, we can parallelize essentially any piece of computation, even if it depends on other parallel computations. Recall that when using fork-join and async-finish, we have had to ensure that the parallel computations being spawned are indeed independent. This is not necessary with futures.

To indicate a parallel computations, we shall use the `future` construct, which takes an expression as an argument and starts a parallel computation that will compute the value of that expression sometime in the future. The construct returns a "future" value representing the computation, which can be used just like any other value of the language. When the time comes to use the value of the future, we demand its completion by using the `force` construct.

7.1 Parallel Fibonacci via Futures

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

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

We can write a parallel version of Fibonacci using futures as follows.

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

Note that this code is very similar to the fork-join version. In fact, we can directly map any use of fork-join parallelism to futures.

Much the same way that we represent fork-join and async-finish computations with a dag of threads, we can also represent future-based computations with a dag. The idea is to spawn a subdag for each `future` and for each `force` add an edge from the terminus of that subdag to the vertex that corresponds to the `force`. For example, for the Fibonacci function, the dags with futures are identical to those with fork-join. As we shall see, however, we can create more interesting dags with futures.

7.2 Incrementing an array, in parallel

Recall our example for mapping 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;
}
```

We can write a parallel version of this code using futures by following the same approach that we did for Fibonacci. But let's consider something slightly different. Suppose that the array is given to us an array of future values. We can write a serial version of `map_incr` for such an array as follows.

```
void future_map_incr(const long^* source, long^* dest, long n) {
  for (long i = 0; i < n; i++)
    dest[i] = future [&] {force (source[i]) + 1};
}
```

The function `future_map_incr` takes an array of futures of long, written `long^` and returns an array of the same type. To compute each value of the array, the function `force`'s the corresponding element of the source inside a `future`.

7.3 Futures and Pipelining

Futures enable expressing parallelism at a very fine level of granularity, at the level of individual data dependencies. This in turn allows parallelizing computations at a similarly finer grain, enabling a technique called pipelining.

The idea behind ***pipelining*** is to decompose a task T into a sequence of smaller subtasks $T_0, \ldots, T_k$ to be performed in that order and overlap the computation of multiple tasks by starting another instance of the same kind of task as soon as the first subtask completes. As a result, if we have a collection of the same kinds of tasks that can be decomposed in the same way, we can have multiple of them "in flight" without having to wait for one to complete.

When computing sequentially, pipelining does not help performance because we can perform one computation at each time step. But when using parallel hardware, we can keep multiple processors busy by pipelining.

Suppose as an example, we have a sequence of tasks $T^0, T^1, T^2, T^3, T^4, T^5$ that we wish to compute. Suppose furthermore that these task depend on each other, that is a later task use the results of an earlier task. Thus it might seem that there is no parallelism that we can exploit. Upon further inspection, however, we may realize that we can partition each task T_i into subtasks $T^i_0, T^i_1, T^i_2, T^i_3, T^i_4, T^i_5$ such that the subtask j of task i is used by the subtask $j+1$ of task $i+1$. We can then execute these tasks in parallel as shown below. Thus in the "steady state" where we have a large supply of these tasks, we can increase performance by a factor 5, the number of subtasks that a task decomposes.

Processor	Step 0	Step 1	Step 2	Step 3	Step 4	Step 5
P_0	T^0_0	T^0_1	T^0_2	T^0_3	T^0_4	T^0_5
P_1		T^1_0	T^1_1	T^1_2	T^1_3	T^1_4
P_1			T^2_0	T^2_1	T^2_2	T^2_3
P_3				T^3_0	T^3_1	T^3_2
P_4					T^4_0	T^4_1
P_5						T^5_0

This idea of pipelining turns out to be quite important in some algorithms, leading sometimes to asymptotic improvements in run-time. It can, however, be painful to design the algorithm to take advantage of pipelining, especially if all we have available at our disposal are fork-join and async-finish parallelism, which require parallel computations to be independent. When using these constructs, we might therefore have to redesign the algorithm so that independent computations can be structurally separated and spawned in parallel. On the other hand, futures make it trivial to express pipelined algorithms because we can express data dependencies and ignore how exactly the individual computations may need to be executed in parallel. For example, in our hypothetical example, all we have to do is create a future for each sub-task, and force the relevant subtask as needed, leaving it to the scheduler to take advantage of the parallelism made available.

As a more concrete example, let's go back to our array increment function and generalize to array map, and assume that we have another function `mk_array` that populates the contents of the array.

```
long f (long i) {
  ...
}

long g (long i) {
  ...
}

void mk_array (long^* source, long n) {
  for (long i = 0; i < n; i++)
    source[i] = future ( [&] { f(i) } );
}

void future_map_g (const long^* source, long^* dest, long n) {
  for (long i = 0; i < n; i++)
    dest[i] = future ( [&] {g (force (source[i]))} );
}

main (long n) {
  long^* source = alloc (long, n);
```

```
  long^* dest = alloc (long, n);
  mk_array (source, n)
  future_map_g (source, dest, n)

}
```

In this example, the i^{th} element of the destination array depends only on the i^{th} element of the source, thus as soon as that element is available, the function `g` might be invoked. This allows us to pipeline the execution of the functions `mk_array` and `future_map_g`.

While we shall not discuss this in detail, it is possible to improve the asymptotic complexity of certain algorithms by using futures and pipelining.

An important research question regarding futures is their scheduling. One challenge is primarily contention. When using futures, it is easy to create dag vertices, whose out-degree is non-constant. The question is how can such dag nodes can be represented to make sure that they don't create a bottleneck, while also allowing the small-degree instance, which is the common case, to proceed efficiently. Another challenge is data locality. Async-finish and fork-join programs can be scheduled to exhibit good data locality, for example, using work stealing. With futures, this is more tricky. It can be shown for example, that even a single scheduling action can cause a large negative impact on the data locality of the computation.

8 Critical Sections and Mutual Exclusion

In a multithreaded program, a ***critical section*** is a part of the program that may not be executed by more than one thread at the same time. Critical sections typically contain code that alters shared objects, such as shared (e.g., global) variables. This means that the a critical section requires ***mutual exclusion***: only one thread can be inside the critical section at any time.

Since only one thread can be inside a critical section at a time, threads must coordinate to make sure that they don't enter the critical section at the same time. If threads do not coordinate and multiple threads enter the critical section at the same time, we say that a ***race condition*** occurs, because the outcome of the program depends on the relative timing of the threads, and thus can vary from one execution to another. Race conditions are sometimes benign but usually not so, because they can lead to incorrect behavior. Spectacular examples of race conditions' effects include the "Northeast Blackout" of 2003, which affected 45 million people in the US and 10 million people in Canada.

It can be extremely difficult to find a race condition, because of the non-determinacy of execution. A race condition may lead to an incorrect behavior only a tiny fraction of the time, making it extremely difficult to observe and reproduce it. For example, the software fault that lead to the Northeast blackout took software engineers "weeks of poring through millions of lines of code and data to find it" according to one of the companies involved.

The problem of designing algorithms or protocols for ensuring mutual exclusion is called the ***mutual exclusion problem*** or the ***critical section*** problem. There are many ways of solving instances of the mutual exclusion problem. But broadly, we can distinguish two categories: spin-locks and blocking-locks. The idea in ***spin locks*** is to busy wait until the critical section is clear of other threads. Solutions based on ***blocking locks*** is similar except that instead of waiting, threads simply block. When the critical section is clear, a blocked thread receives a signal that allows it to proceed. The term ***mutex***, short for "mutual exclusion" is sometimes used to refer to a lock.

Mutual exclusions problems have been studied extensively in the context of several areas of computer science. For example, in operating systems research, processes, which like threads are independent threads of control, belonging usually but not always to different programs, can share certain systems' resources. To enable such sharing safely and efficiently, researchers have proposed various forms of locks such as ***semaphores***, which accepts both a busy-waiting and blocking semantics. Another class of locks, called ***condition variables*** enable blocking synchronization by conditioning an the value of a variable.

8.1 Parallelism and Mutual Exclusion

In parallel programming, mutual exclusion problems do not have to arise. For example, if we program in a purely functional language extended with structured multithreading primitives such as fork-join and futures, programs remain purely functional and mutual-exclusion problems, and hence race conditions, do not arise. If we program in an imperative language, however, where memory is always a shared resource, even when it is not intended to be so, threads can easily share memory objects, even unintentionally, leading to race conditions. .Writing to the same location in parallel.

In the code below, both branches of `fork2` are writing into `b`. What should then the output of this program be?

```
long b = 0;

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

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

At the time of the print, the contents of `b` is determined by the last write. Thus depending on which of the two branches perform the write, we can see both possibilities:

Output:

```
b = 1
```

Output:

```
b = 2
```

Example 8.1 Fibonacci

Consider the following alternative implementation of the Fibonacci function. By "inlining" the plus operation in both branches, the programmer got rid of the addition operation after the `fork2`.

```
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;
}
```

This code is not correct because it has a race condition.

As in the example shows, separate threads are updating the value result but it might look like this is not a race condition because the update consists of an addition operation, which reads the value and then writes to i. The race condition might be easier to see 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 8.2 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.

Time step	Thread 1	Thread 2
1	a1 = fib_par_racy(1)	b2 = fib_par_racy(0)
2	a2 = result	b3 = result
3	result = a1 + a2	_
4	_	result = b1 + b2

The reason we get a bogus result is that both threads read the initial value of result at the same time and thus do not see each others write. In this example, the second thread "wins the race" and writes into `result`. The value 1 written by the first thread is effectively lost by being overwritten by the second thread.

8.2 Synchronization Hardware

Since mutual exclusion is a common problem in computer science, many hardware systems provide specific synchronization operations that can help solve instances of the problem. These operations may allow, for example, testing the contents of a (machine) word then modifying it, perhaps by swapping it with another word. Such operations are sometimes called atomic ***read-modify-write*** or ***RMW***, for short, operations.

A handful of different RMW operations have been proposed. They include operations such as ***load-link/store-conditional***, ***fetch-and-add***, and ***compare-and-swap***. They typically take the memory location `x`, and a value `v` and replace the value of stored at `x` with `f(x,v)`. For example, the fetch-and-add operation takes the location `x` and the increment-amount, and atomically increments the value at that location by the specified amount, i.e., `f(x,v) =*x + v`.

The compare-and-swap operation takes the location `x` and takes a pair of values `(a,b)` as the second argument, and stores `b` into `x` if the value in `x` is `a`, i.e., `f(x,(a,b)) =if *x =a then b else a`; the operation returns a Boolean indicating whether the operation successfully stored a new value in `x`. The operation "compare-and-swap" is a reasonably powerful synchronization operation: it can be used by arbitrarily many threads to agree (reach consensus) on a value. This instruction therefore is frequently provided by modern parallel architectures such as Intel's X86.

In C++, the `atomic` class can be used to perform synchronization. Objects of this type are guarantee to be free of race conditions; and in fact, in C++, they are the only objects that are guaranteed to be free from race conditions. The contents of an `atomic` object can be accessed by `load` operations, updated by `store` operation, and also updated by `compare_exchange_weak` and `compare_exchange_strong` operations, the latter of which implement the compare-and-swap operation.

Example 8.3 Accessing the contents of atomic memory cells

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

```
std::atomic<bool> flag;

flag.store(false);
std::cout << flag.load() << std::endl;
flag.store(true);
std::cout << flag.load() << std::endl;
```

Output:

```
0
1
```

The key operation that help with race conditions is the compare-and-exchange operation.

Definition: compare and swap

When executed with a ‘target` atomic object and an `expected` cell and a new value `new’ this operation performs the following steps, atomically:

1. Read the contents of `target`.
2. If the contents equals the contents of `expected`, then writes `new` into the `target` and returns `true`.
3. Otherwise, returns `false`.

Example 8.4 Reading and writing atomic objects

```
std::atomic<bool> flag;

flag.store(false);
bool expected = false;
bool was_successful = flag.compare_exchange_strong(expected, true);
std::cout << "was_successful = " << was_successful << "; flag = " << flag.load() << std:: ↩
    endl;
bool expected2 = false;
bool was_successful2 = flag.compare_exchange_strong(expected2, true);
std::cout << "was_successful2 = " << was_successful2 << "; flag = " <<
flag.load() << std::endl;
```

Output:

```
was_successful = 1; flag = 1
was_successful2 = 0; flag = 1
```

As another example use of the `atomic` class, recall our Fibonacci example with the race condition. In that example, race condition arises because of concurrent writes to the `result` variable. We can eliminate this kind of race condition by using different memory locations, or by using an atomic class and using a `compare_exchange_strong` operation.

Example 8.5 Fibonacci

The following implementation of Fibonacci is not safe because the variable `result` is shared and updated by multiple threads.

```
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;
}
```

We can solve this problem by declaring `result` to be an atomic type and using a standard busy-waiting protocol based on compare-and-swap.

```
long fib_par_atomic(long n) {
  atomic<long> result = 0;
  if (n < 2) {
    result.store(n);
```

```
  } else {
    fork2([&] {
      long r = fib_par_racy(n-1);
      // Atomically update result.
      while (true) {
        long exp = result.load();
        bool flag = result.compare_exchange_strong(exp,exp+r)
        if (flag) {break;}
      }
    },[&] {
      long r = fib_par_racy(n-2);
      // Atomically update result.
      while (true) {
        long exp = result.load();
        bool flag = result.compare_exchange_strong(exp,exp+r)
        if (flag) {break;}
      }
    });
  }
  return result;
}
```

The idea behind the solution is to load the current value of `result` and atomically update `result` only if it has not been modified (by another thread) since it was loaded. This guarantees that the `result` is always updated (read and modified) correctly without missing an update from another thread.

The example above illustrates a typical use of the compare-and-swap operation. In this particular example, we can probably prove our code is correct. But this is not always as easy due to a problem called the "ABA problem."

8.3 ABA problem

While reasonably powerful, compare-and-swap suffers from the so-called ***ABA*** problem. To see this consider the following scenario where a shared variable `result` is update by multiple threads in parallel: a thread, say T, reads the `result` and stores its current value, say `2`, in `current`. In the mean time some other thread also reads `result` and performs some operations on it, setting it back to `2` after it is done. Now, thread T takes its turn again and attempts to store a new value into `result` by using `2` as the old value and being successful in doing so, because the value stored in `result` appears to have not changed. The trouble is that the value has actually changed and has been changed back to the value that it used to be. Thus, compare-and-swap was not able to detect this change because it only relies on a simple shallow notion of equality. If for example, the value stored in `result` was a pointer, the fact that the pointer remains the same does not mean that values accessible from the pointer has not been modified; if for example, the pointer led to a tree structure, an update deep in the tree could leave the pointer unchanged, even though the tree has changed.

This problem is called the ***ABA*** problem, because it involves cycling the atomic memory between the three values A, B, and again A). The ABA problem is an important limitation of compare-and-swap: the operation itself is not atomic but is able to behave as if it is atomic if it can be ensured that the equality test of the subject memory cell suffices for correctness.

In the example below, ABA problem may happen (if the counter is incremented and decremented again in between a load and a store) but it is impossible to observe because it is harmless. If however, the compare-and-swap was on a memory object with references, the ABA problem could have had observable effects.

The ***ABA*** problem can be exploited to give seemingly correct implementations that are in fact incorrect. To reduce the changes of bugs due to the ABA problem, memory objects subject to compare-and-swap are usually tagged with an additional field that counts the number of updates. This solves the basic problem but only up to a point because the counter itself can also wrap around. The load-link/store-conditional operation solves this problem by performing the write only if the memory location has not been updated since the last read (load) but its practical implementations are hard to come by.

9 Chapter: Experimenting with PASL

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.

9.1 Obtain source files

Let's start by downloading the PASL sources. 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.

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

9.2 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. To skip this part and use installed binaries, see the heading "Starting with installed binaries", below.

9.2.1 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
`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)

The 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/`.

9.2.2 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 9.1 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.

9.2.3 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 9.2 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 9.3 Configuration to use `hwloc`

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

9.3 Starting with installed binaries

At this point, you have either installed all the necessary software to work with PASL or these are installed for you. In either case, make sure that your `PATH` variable makes the software visible. For setting up your `PATH` variable on andrew.cmu domain, see below.

9.3.1 Specific set up for the andrew.cmu domain

We have installed much of the needed software on andrew.cmu.edu. So you need to go through a relatively minimal set up.

First set up your `PATH` variable to refer to the right directories. Using cshell

```
setenv PATH  /opt/rh/devtoolset-3/root/usr/bin:/usr/lib64/qt-3.3/bin:/usr/lib64/ccache:/usr ↩
    /local/bin:/bin:/usr/bin:./
```

The part added to the default PATH on andrew is

```
/opt/rh/devtoolset-3/root/usr/bin
```

It is important that this is at the beginning of the `PATH` variable. To make interaction easier, we also added the relative path `./` to the `PATH` variable.

9.3.2 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.

9.3.3 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.

9.3.4 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. Add the commands above to your shell configuration file.

9.3.5 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 by putting it into your shell configuration file (as you did above for the pbench tools).

9.4 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.

9.5 Task 1: Run the baseline Fibonacci

We are going to start our experimentation with three different instances of the same program, namely `bench`. This program serves as a "driver" for the benchmarks that we have implemented. These implementations are good parallel codes that we expect to deliver good performance. We first build the baseline version.

```
$ 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.

We can now run the baseline for one of our benchmarks, say Fibonacci by using the `-bench` argument to specify the benchmark and the `-n` argument to specify 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.

9.6 Task 2: Run the sequential elision of Fibonacci

The `.elision` extension means that parallel algorithms (not sequential baseline algorithms) are compiled. However, all instances of `fork2()` are erased as described in an earlier 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
```

9.7 Task 3: Run parallel Fibonacci

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. Before running this command, please adjust the `-proc` option to match the number of cores that your machine has. Note that you can use any number of cores up to the number you have available. You can use `nproc` or `lscpu` to determine the number of cores your machine has.

```
$ 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.

9.8 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 a valid indication of good performance, they must be calculated against an optimized implementation of the best serial 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 9.4 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 computing power that our machine offers.

9.8.1 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
```

Here is another example on a 24-core machine.

```
$ prun speedup -baseline "bench.baseline" -parallel "bench.opt -proc 1,4,8,16,24" -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 speedup plot below.

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

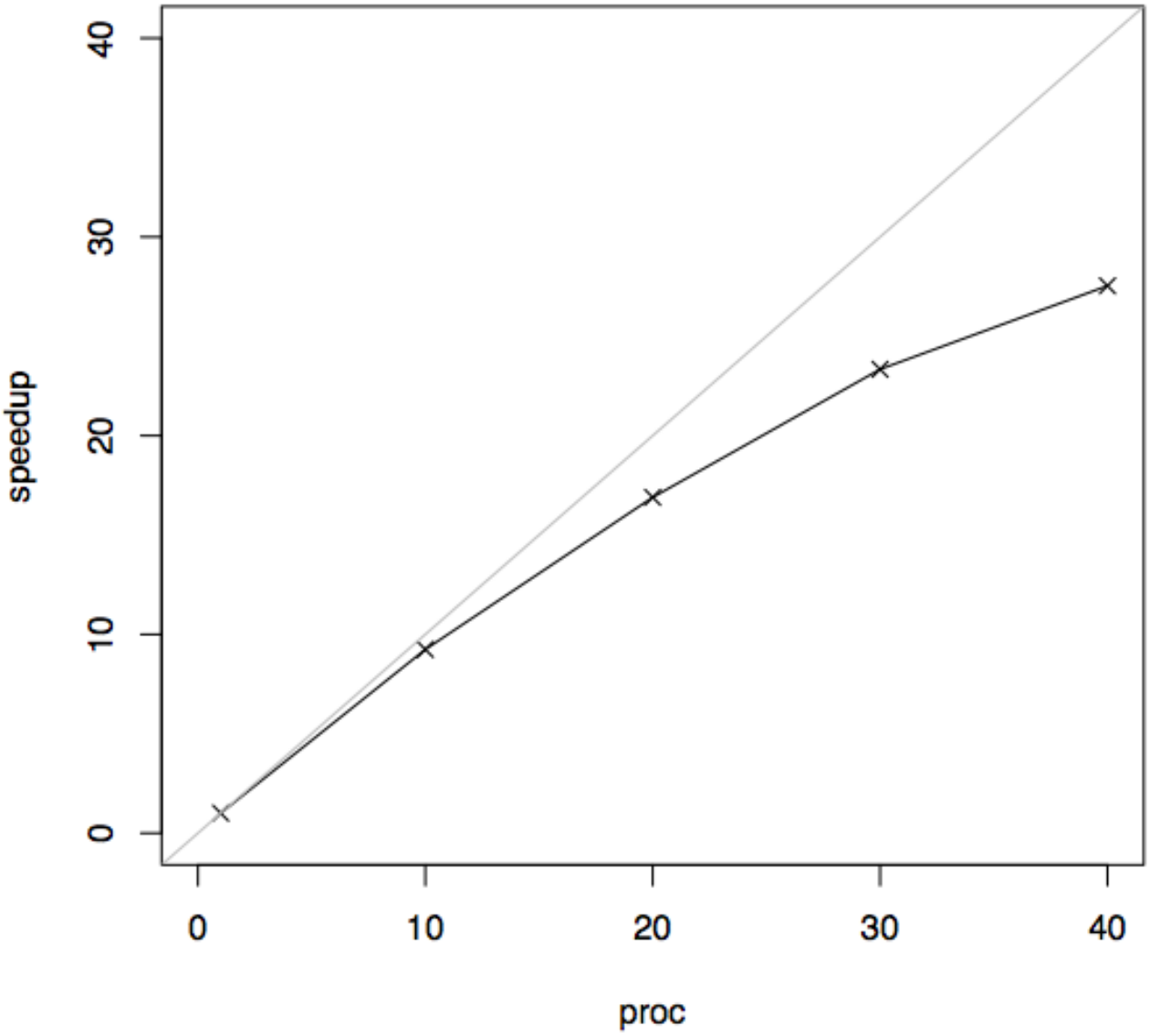


Figure 5: Speedup curve for the computation of the 39th Fibonacci number.

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 the Figure below 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.

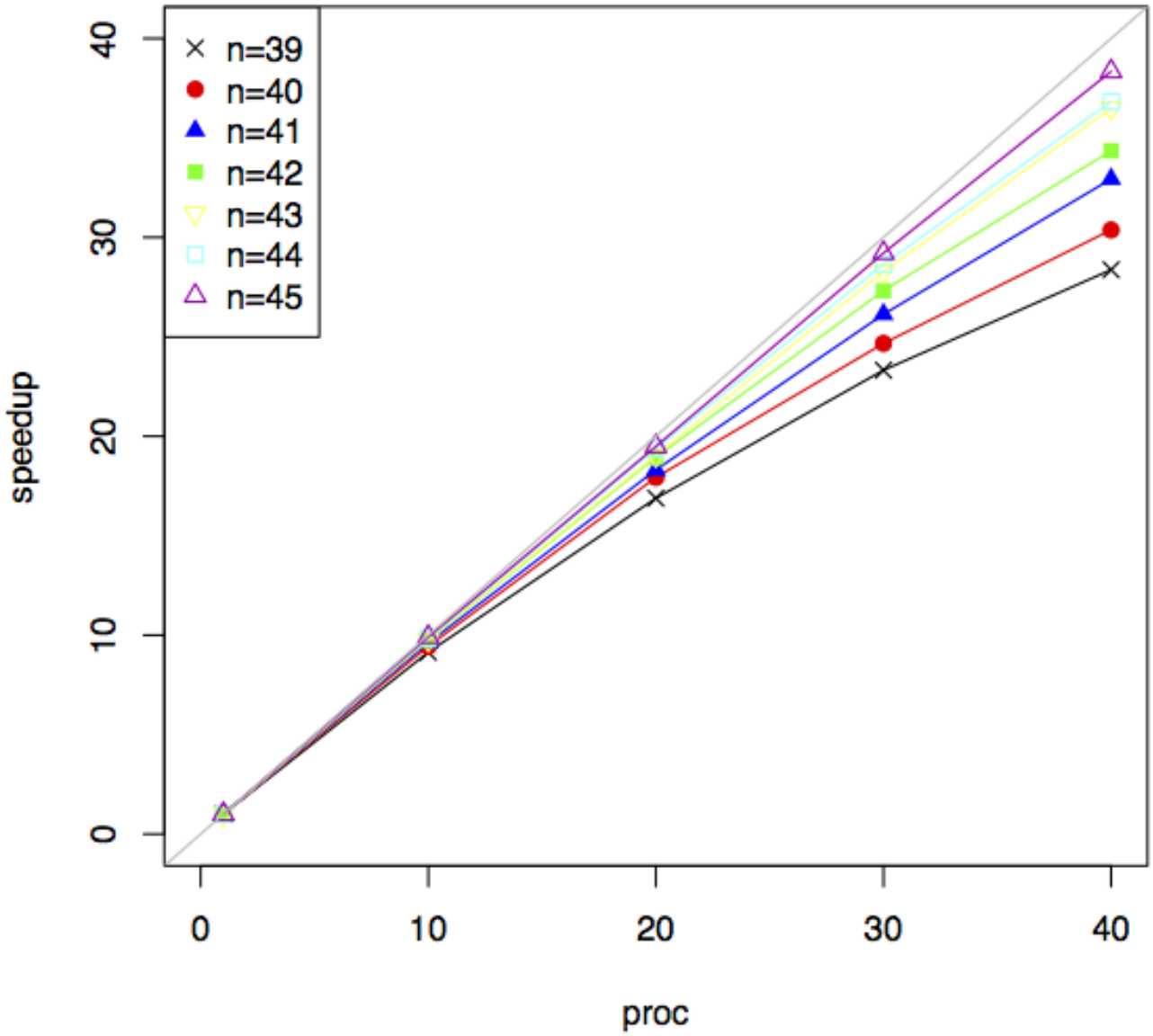


Figure 6: Speedup plot showing speedup curves at different problem sizes.

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.

9.8.2 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.

9.9 Visualize processor utilization

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 the Figure below. 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. You can zoom in any part of the plot by clicking on the region with the mouse. To reset to the original plot, press the space bar. 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.

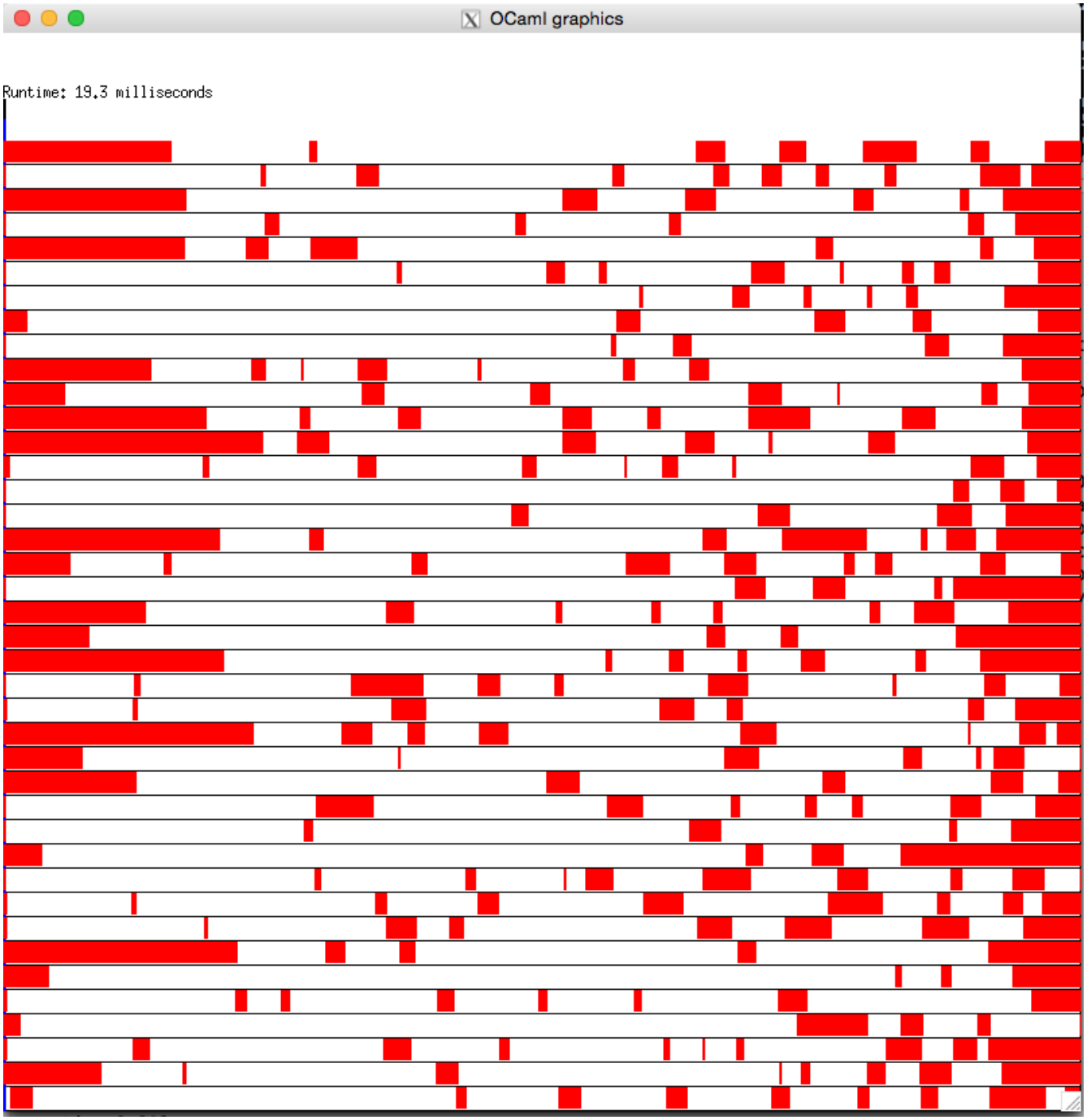


Figure 7: Utilization plot for computation of 39th Fibonacci number.

9.10 Strong versus weak scaling

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 the Figure below, 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.

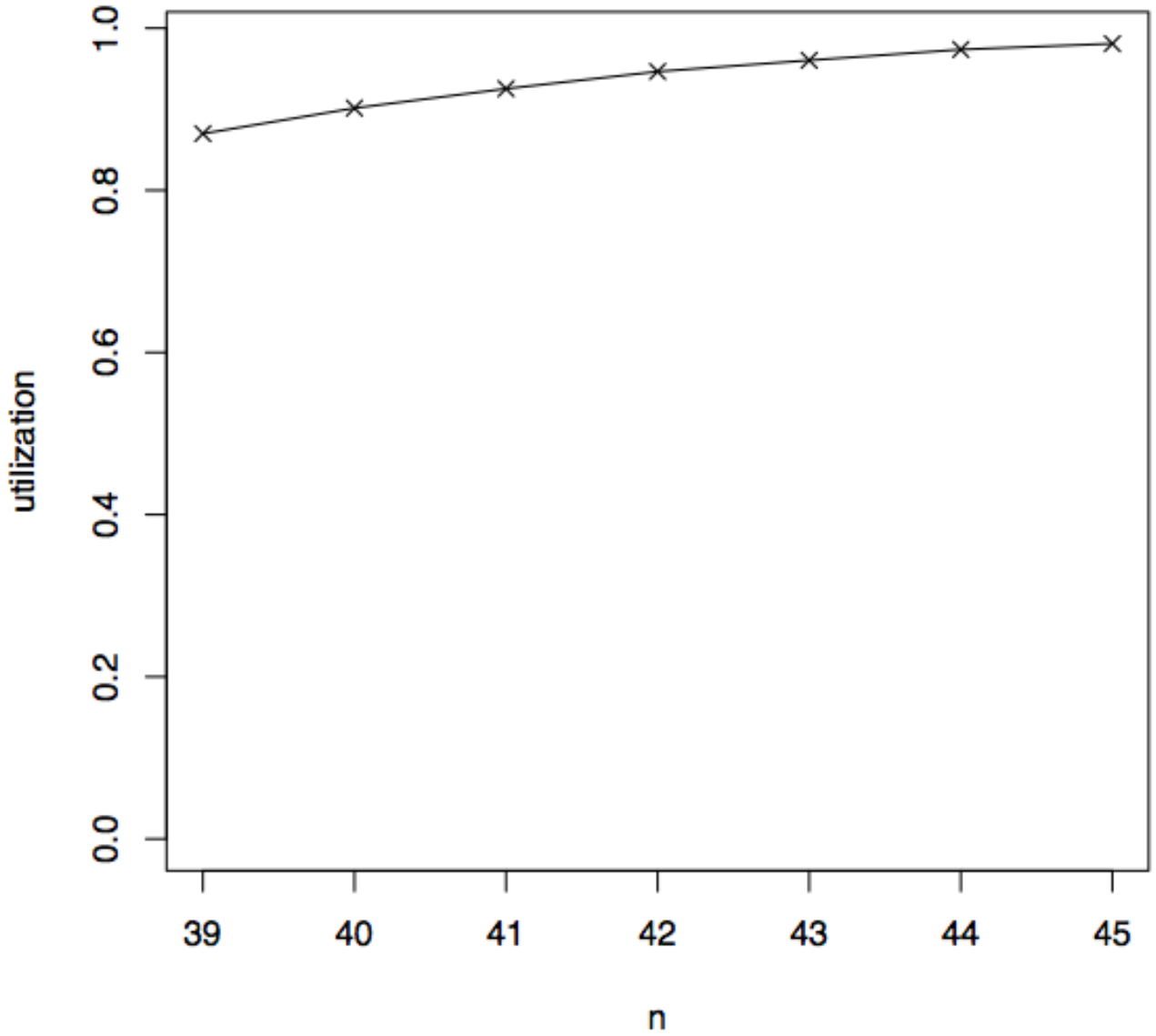


Figure 8: How processor utilization of Fibonacci computation varies with input size.

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 the Figure below. Compared the to utilization plot we saw in the Figure above for n=39, the red regions are much less prominent overall and the idle regions at the beginning and end are barely noticeable.

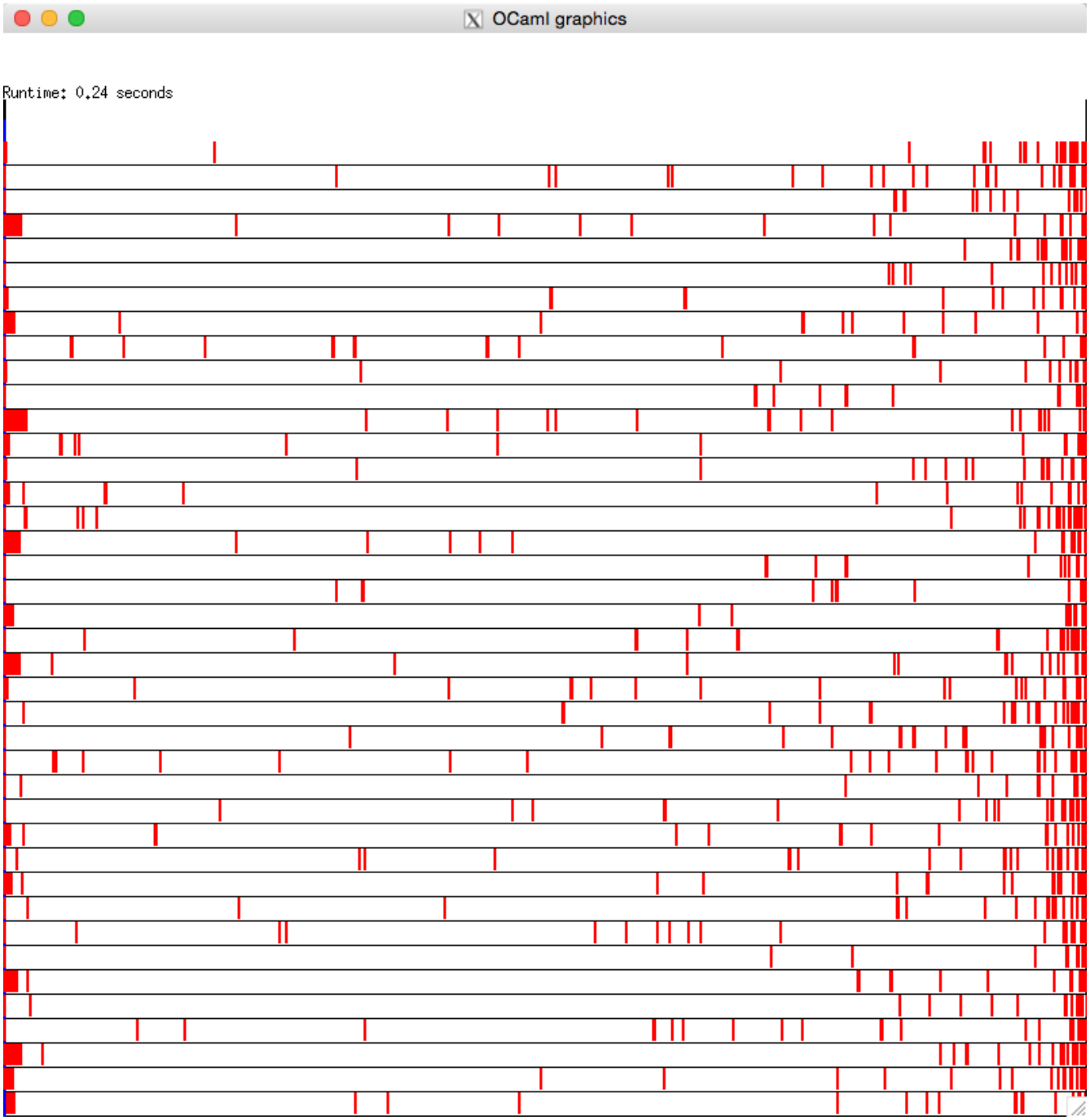


Figure 9: Utilization plot for computation of 45th Fibonacci number.

9.11 Chapter Summary

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.

10 Chapter: Work efficiency

In many 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 for several reasons. First, since it performs additional work with respect to the serial algorithm, a parallel algorithm will generally require more resources such as time and energy. By using more processors, it may be possible to reduce the time penalty, but only by using more hardware resources.

For example, if an algorithm performs $O(\log n)$-factor more work than the serial algorithm, then, assuming that the constant factor hidden by the asymptotic notation is 1, when $n = 2^{20}$, it will perform 20-times more actual work, consuming 20 times more energy consumption than the serial algorithm. Assuming perfect scaling, we can reduce the time penalty by using more processors. For example, with 40 processors, the algorithm may require half the time of the serial algorithm.

Sometimes, a parallel algorithm has the same asymptotic complexity of the best serial algorithm for the problem but it has larger constant factors. This is generally true because scheduling friction, especially the cost of creating threads, can be significant. In addition to friction, parallel algorithms can incur more communication overhead than serial algorithms because data and processors may be placed far away in hardware. For example, it is not unusual for a parallel algorithm to incur a $10-100\times$ overhead over a similar serial algorithm because of scheduling friction and communication.

These considerations motivate considering "work efficiency" of parallel algorithm. Work efficiency is a measure of the extra work performed by the parallel algorithm with respect to the serial algorithm. We define two types of work efficiency: ***asymptotic work efficiency*** and ***observed work efficiency***. 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 of the algorithm is the same as the work of the best known serial algorithm.

Example 10.1 Asymptotic work efficiency

- A parallel algorithm that comparison-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 $O(n \log^2 n)$ work is not asymptotically work efficient.
- The parallel array increment algorithm that we consider in an earlier Chapter is asymptotically work efficient, because it performs linear work, which is optimal (any sequential algorithm must perform at least linear work).

To assess the practical effectiveness of a parallel algorithm, we define observed work efficiency, parameterized a value r.

Definition: observed work efficiency

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

Example 10.2 Observed work efficiency

- A parallel algorithm that runs $10\times$ slower on a single processor that the fastest sequential algorithm has an observed work efficiency factor of 10. 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 has observed work efficiency factor of 1.2. We consider such algorithms to be acceptable.

Example 10.3 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 parallel algorithm, which is the same exact code as `map_incr_rec`. We build this code by using the special `optfp` "force parallel" 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 algorithm has an observed work efficiency factor of $60\times$. Such poor observed work efficiency suggests that the parallel algorithm would require more than an order of magnitude more energy and that it would not run faster than the serial algorithm even when using less than 60 processors.

In practice, observed work efficiency is a major concern. First, the whole effort of parallel computing is wasted if parallel algorithms consistently require more work than the best sequential algorithms. In other words, in parallel computing, both asymptotic complexity and constant factors matter.

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.

10.1 Improving work efficiency with granularity control

It is common for a parallel algorithm to be asymptotically and/or observably work inefficient but it is often possible to improve work efficiency by observing that work efficiency increases with parallelism and can thus be controlled by limiting it.

For example, a parallel algorithm that performs linear work and has logarithmic span leads to average parallelism in the orders of thousands with the small input size of one million. For such a small problem size, we usually would not need to employ thousands of processors. It would be sufficient to limit the parallelism so as to feed tens of processors and as a result reduce impact of excess parallelism on work efficiency.

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*** or ***granularity control***.

But which code should 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 would be ineffective. In such cases, we can specify a separate sequential algorithm for small instances.

Optimizing the practical efficiency of a parallel algorithm by controlling its parallelism is sometimes called ***optimization***, sometimes it is called ***performance engineering***, and sometimes ***performance tuning*** or simply ***tuning***. In the rest of this document, we use the term "tuning."

Example 10.4 Tuning the parallel array-increment function

We can apply coarsening to `map_inc_rec` 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 is faster than other sorting algorithms for very small inputs containing 30 keys or less. Many optimize sorting algorithm therefore revert to insertion sort when the input size falls within that range.

Example 10.5 Observed work efficiency of tuned array increment

As can be seen below, after some tuning, `map_incr_rec` program becomes highly work efficient. In fact, there is barely a difference between the serial and the parallel runs. The tuning is actually done automatically here by using an automatic-granularity-control technique described in the 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.

10.2 Determining the threshold

The basic idea behind coarsening or granularity control is to revert to a fast serial algorithm when the input size falls below a certain threshold. To determine the optimal threshold, we can simply perform a search by running the code with different threshold settings.

While this approach can help find the right threshold on the particular machine that we performed the search, there is no guarantee that the same threshold would work on another machine. In fact, there are examples in the literature that show that such optimizations are not ***portable***, i.e., a piece of code optimized for a particular architecture may behave poorly on another.

In the general case, determining the right threshold is even more difficult. To see the difficulty consider a generic (polymorphic), higher-order function such as `map` that takes a sequence and a function and applies the function to the sequence. The problem is that the threshold depends both on the type of the elements of the sequence and the function to be mapped over the sequence. For example, if each element itself is a sequence (the sequence is nested), the threshold can be relatively small. If, however, the elements are integers, then the threshold will likely need to be relatively large. This makes it difficult to determine the threshold because it depends on arguments that are unknown at compile time. Essentially the same argument applies to the function being mapped over the sequence: if the function is expensive, then the threshold can be relatively small, but otherwise it will need to be relatively large.

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

11 Chapter: Automatic granularity control

There has been significant research into determining the right threshold for a particular algorithm. This problem, known as the ***granularity-control problem***, turns out to be a rather difficult one, especially if we wish to find a technique that can ensure close-to-optimal performance across different architectures. In this section, we present a technique for automatically controlling granularity by using asymptotic cost functions.

11.1 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 11.1 Complexity function of `map_incr_rec`

An application of our `map_incr_rec` function on a given range $[lo, hi)$ of an 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_rec` 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. The following lambda expression is another valid complexity function for function `map_incr_rec`. The complexity function above is preferable, however, because it is simpler.

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

More generally, suppose that we know that a given algorithm has work cost of $W = n + \log n$. 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, when expressing work costs, we only need to be precise up to the asymptotic complexity class of the work.

11.2 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 outweigh 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 exection is similar to a sequential elision, 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 11.2 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 n; }, [&] {
    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.

11.2.1 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 11.3 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 n; }, [&] {
    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 algorithmically identical, except for the calls to `fork2()`, there is still an advantage to using the alternative sequential body: the sequential code does not pay for the parallelism overheads due to `fork2()`. Even when eliding `fork2()`, the run-time-system has to 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. Another reason for why a sequential body may be more efficient is that it can be written more simply, as for example using a for-loop instead of recursion, which will be faster in practice.

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.

11.3 Controlled parallel-for loops

Let us add one more component to our granularity-control toolkit: the ***parallel-for*** from. By using this loop construct, we can avoid having to explicitly express recursion-trees 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, also by replacing the parallel-for with a serial-for.

```
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 11.4 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.

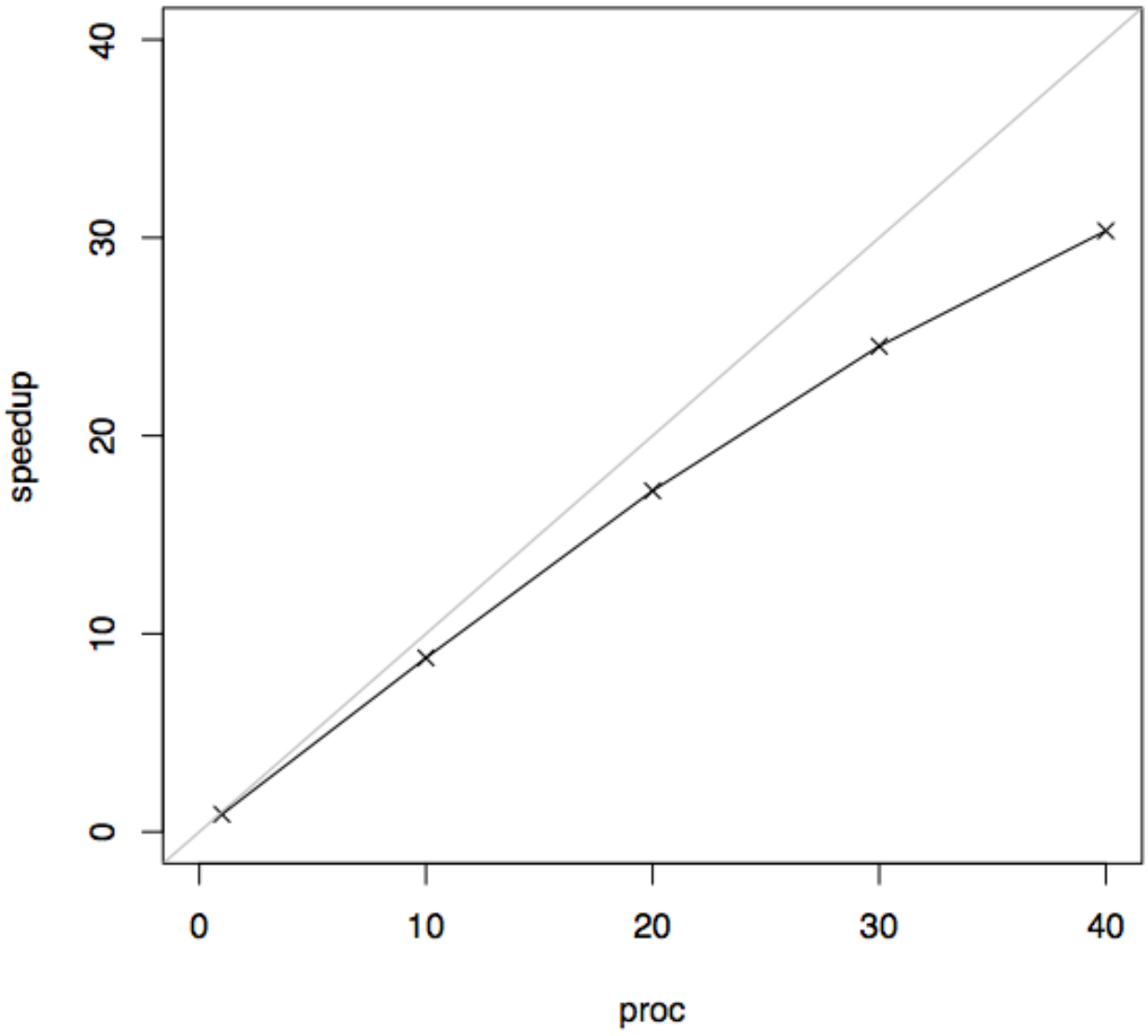


Figure 10: Speedup plot for matrix multiplication for 25000×25000 matrices.

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: it touches $\Theta(n^2)$ memory cells, while performing nearly $\Theta(n^3)$ work.

12 Simple Parallel Arrays

Arrays are a fundamental data structure in sequential and parallel computing. When computing sequentially, arrays can sometimes be replaced by linked lists, especially because linked lists are more flexible. Unfortunately, linked lists are deadly for parallelism, because they require serial traversals to find elements; this makes arrays all the more important in parallel computing.

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 12.1 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 can initialize an array in logarithmic span in the number of items.

Example 12.2 STL vectors

The "vector" data structure that is provided by the Standard Template Library (STL) has similar issues. The STL vector implements a dynamically resizable array that provides push, pop, and indexing operations. The push and pop operations take amortized constant time and the indexing operation constant time. As with C++ arrays, initialization of vectors can require linear work and span. 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 not parallel for the same reason as for the default C++ arrays.

Such sequential computations that exist behind the wall of abstraction of a language or library can harm parallelism by introducing implicit sequential dependencies. 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 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 carefully control initialization and disallow implicit copy operations on arrays, because copy operations can harm observable work efficiency (their asymptotic work cost is linear).

12.1 Interface and cost model

The key components of our array data structure, `sparray`, are shown by the code snippet below. An `sparray` can store 64-bit words only; in particular, they are monomorphic and fixed to values of type `long`. Generalizing `sparray` to store values of arbitrary types can be achieved by use of C++ templates. We stick to monomorphic arrays here to simplify the presentation.

```
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 is constant by definition.

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, based on constant-size lists, 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.
- **Copy constructor:** The copy constructor of `sparray` is disabled. This prohibits copying an array unintentionally, for example, by passing the array by value to a function.

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 12.3 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
```

12.2 Allocation and deallocation

Arrays can be allocated by specifying the size of the array.

Example 12.4 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 12.5 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 12.6 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 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.

12.3 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 12.7 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 12.8 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 be constant work — not linear like a copy would take. The return is fast, because internally all that happens is that a couple words are being exchanged. Such "move on return" is achieved by the "move-assignment operator" of `sparray` class.

Example 12.9 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 }
```

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 12.10 Move constructor

```
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 }`, moves the contents of `xs` to `ys`, and empties out the contents of `xs`. This behavior is defined as part of the move operator of `sparray`. 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.

Exercise: duplicating items in parallel

The aim of this exercise is to combine our knowledge of parallelism and arrays. To this end, the exercise 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?

12.4 Tabulation

A **tabulation** is a parallel operation which creates a new array of a given size and initializes the contents according to a given "generator function". 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 12.11 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 12.12 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 12.13 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.

```
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;
}
```

Note that the work and span of the generator function depends on the generator function passed as an argument 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

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

12.5 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.

Note
Above, we still assume constant-work generator functions.

12.6 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 12.14 Addition monoid

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

Example 12.15 Multiplication monoid

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

Example 12.16 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 b, value_type id, const sparray& xs);
```

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

Example 12.17 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 12.18 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.

12.7 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 12.19 Inclusive scan

```
  scan_incl(b, 0, sparray({ 2, 1, 8, 3 }))
= { reduce(b, id, { 2 }),       reduce(b, id, { 2, 1 }),
    reduce(b, id, { 2, 1, 8 }), reduce(b, 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 12.20 Exclusive scan

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

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

  scan_excl(b, 0, { 2, 1, 8, 3 }).total
= reduce(b, 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.

12.8 Derived 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.

12.8.1 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 12.21 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.

12.8.2 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 12.22 Creating an array of all 3s

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

Output:

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

12.8.3 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 12.23 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 }
```

12.8.4 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 12.24 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 }
```

12.8.5 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 12.25 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 }
```

12.8.6 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 12.26 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
```

12.8.7 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 12.27 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 12.28 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?

12.8.8 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 12.29 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.

12.9 Summary of operations

12.9.1 Tabulate

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

The call `tabulate(g, n)` returns the length-`n` array where the `i` th element is given by `g(i)`.

Let $W(\mathtt{g}(i))$ denote the work performed by an application of the generator function to a specified value i. Similarly, let $S(\mathtt{g}(i))$ denote the span. Then the tabulation takes work

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

and span

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

12.9.2 Reduce

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

The call `reduce(b, id, xs)` is logically equal to `id` if `xs.size() ==0`, `xs[0]` if `xs.size() ==1`, and

```
b(reduce(b, id, slice(xs, 0, n/2)),
  reduce(b, id, slice(xs, n/2, n)))
```

otherwise where `n ==xs.size()`.

The work and span cost are $O(n)$ and $O(\log n)$ respectively, where n denotes the size of the input sequence `xs`. This cost assumes that the work and span of `b` are constant. If it's not the case, then refer directly to the implementation of `reduce`.

12.9.3 Scan

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

For an associative function `b` and corresponding identity `id`, the return result of the call `scan_incl(b, id, xs)` is logically equivalent to

```
tabulate([&] (long i) { return reduce(b, id, slice(xs, 0, i+1)); }, xs.size())
```

```
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);
```

For an associative function `b` and corresponding identity `id`, the call `scan_excl(b, id, xs)` returns the object `res`, such that `res.partials` is logically equivalent to

```
tabulate([&] (long i) { return reduce(b, id, slice(xs, 0, i)); }, xs.size())
```

and `res.total` is logically equivalent to

```
reduce(b, id, xs)
```

The work and span cost are $O(n)$ and $O(\log n)$ respectively, where n denotes the size of the input sequence `xs`. This cost assumes that the work and span of `b` are constant. If it's not the case, then refer directly to the implementation of `scan_incl` and `scan_excl`.

12.9.4 Map

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

Let $W(\texttt{f}(x))$ denote the work performed by an application of the function `f` to a specified value x. Similarly, let $S(\texttt{f}(x))$ denote the span. Then the map takes work

$$\sum_{x \in xs} W(\texttt{f}(x))$$

and span

$$\log \texttt{xs.size}() + \max_{x \in xs} S(\texttt{f}(x))$$

12.9.5 Fill

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

Returns a length-`n` array with all cells initialized to `v`.

Work and span are linear and logarithmic, respectively.

12.9.6 Copy

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

Returns a fresh copy of `xs`.

Work and span are linear and logarithmic, respectively.

12.9.7 Slice

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

The call `slice(xs, lo, hi)` returns the array `{xs[lo], xs[lo+1], ..., xs[hi-1]}`.

Work and span are linear and logarithmic, respectively.

12.9.8 Concat

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

Concatenate the two sequences.

Work and span are linear and logarithmic, respectively.

12.9.9 Prefix sums

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

The result of the call `prefix_sums_incl(xs)` is logically equivalent to the result of the call

```
scan_incl([&] (value_type x, value_type y) { return x+y; }, 0, xs)
```

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

The result of the call `prefix_sums_excl(xs)` is logically equivalent to the result of the call

```
scan_excl([&] (value_type x, value_type y) { return x+y; }, 0, xs)
```

Work and span are linear and logarithmic, respectively.

12.9.10 Filter

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

The call `filter(p, xs)` returns the subsequence of `xs` which contains each `xs[i]` for which `p(xs[i])` returns `true`.

Warning
Depending on the implementation, the predicate function `p` may be called multiple times by the `filter` function.

Work and span are linear and logarithmic, respectively.

13 Chapter: Parallel Sorting

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

13.1 Quicksort

The quicksort algorithm for sorting an array (sequence) of elements is known to be a very efficient sequential sorting algorithm. 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. 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` calls 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.

13.1.1 Asymptotic Work Efficiency and Parallelism

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. Here, 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 the analysis, let's assume a version of quicksort that compares the pivot p to each key in the input once (instead of 3 times. The figure below illustrates the structure of an execution of quicksort by using a tree. Each node corresponds to a call to the quicksort function and is labeled with the key at that call. Note that the tree is a binary search tree.

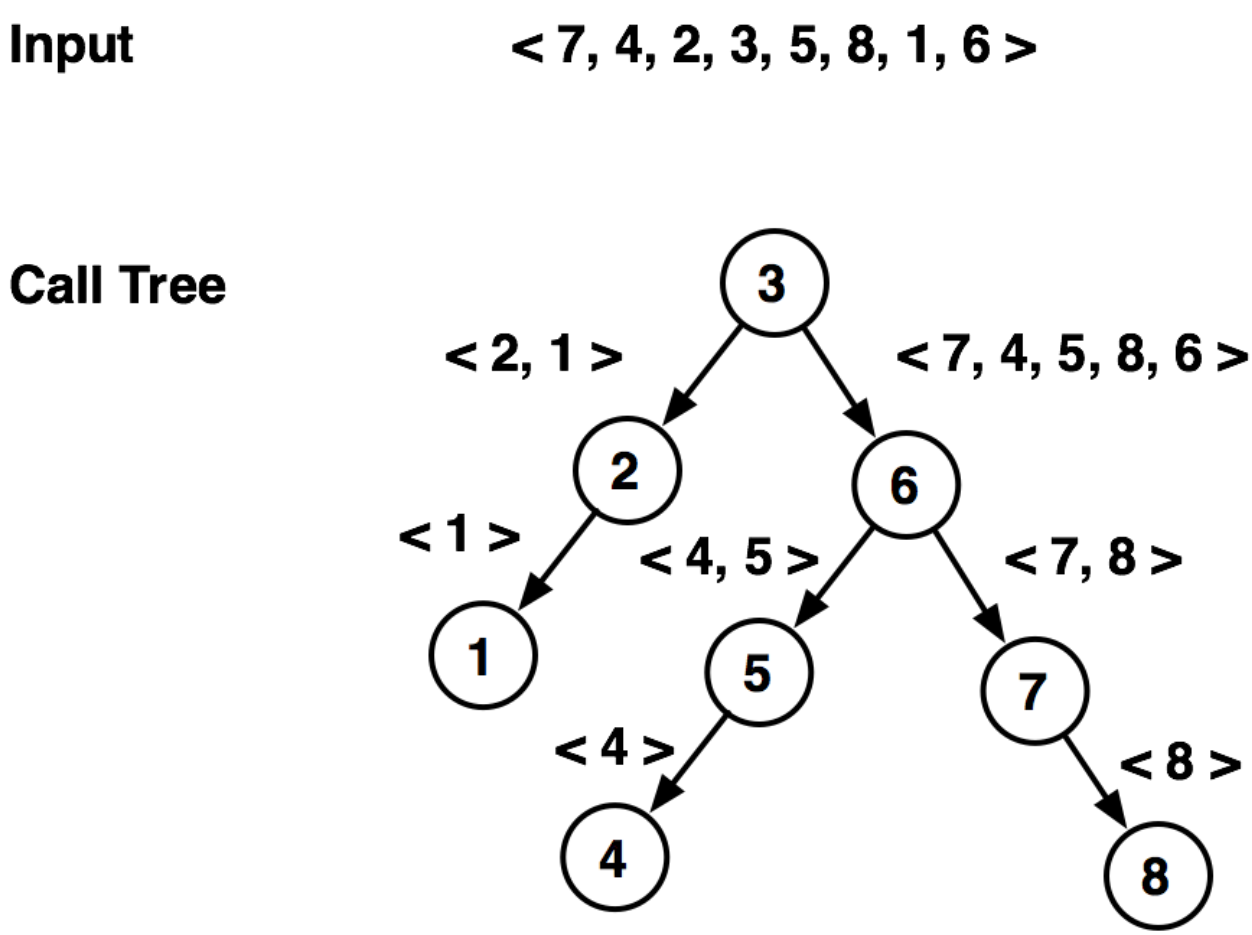


Figure 11: Quicksort call tree.

Let's observe some properties of quicksort.

1. In quicksort, a comparison always involves a pivot and another key. Since, the pivot is never sent to a recursive call, a key is selected as a pivot exactly once, and is not involved in further comparisons (after is becomes a pivot). Before a key is selected as a pivot, it may be compared to other pivots, once per pivot, and thus two keys are never compared more than once.
2. When the algorithm selects a key y as a pivot and if y is between two other keys x,z such that $x < y < z$, it sends the two keys latexmath:[y,z to two separate subtrees. The two keys x and z separated in this way are never compared again.
3. We can sum up the two observations: a key is compared with all its ancestors in the call tree and all its descendants in the call tree, and with no other keys.

Since a pair of key are never compared more than once, the total number of comparisons performed by quicksort can be expressed as the sum over all pairs of keys.

Let X_n be the random variable denoting the total number of comparisons performed in an execution of quicksort with a randomly permuted input of size latexmath:[n.

We want to bound the expectation of X_n, $E[X_n]$.

For the analysis, let's consider the final sorted order of the keys T, which corresponds to the output. Consider two positions $i, j \in \{1,\ldots,n\}$ in the sequence T. We define following random variable:

$$A_{ij} = \begin{cases} 1 & \text{if } T_i \text{ and } T_j \text{are compared} \\ 0 & \text{otherwise} \end{cases}$$

We can write X_n by summing over all A_{ij}'s:

$$X_n \leq \sum_{i=1}^{n} \sum_{j=i+1}^{n} A_{ij}$$

By linearity of expectation, we have

$$E[X_n] \leq \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[A_{ij}]$$

Furthermore, since each A_{ij} is an indicator random variable, $E[A_{ij}] = P(A_{ij}) = 1$. Our task therefore comes down to computing the probability that T_i and T_j are compared, i.e., $P(A_{ij} = 1)$, and working out the sum.

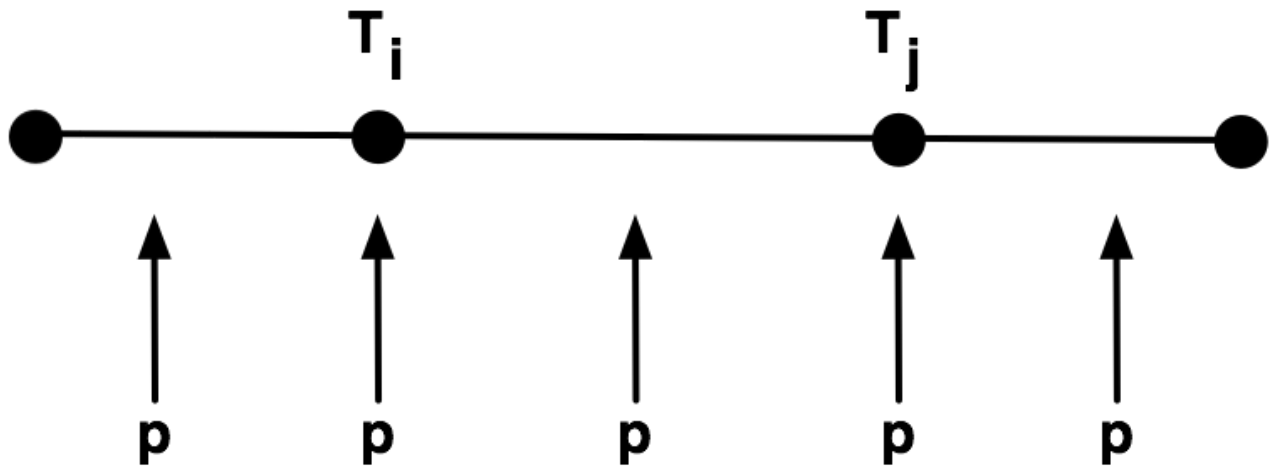


Figure 12: Relationship between the pivot and other keys.

To compute this probability, note that each call takes the pivot p and splits the sequence into two parts, one with keys larger than p and the other with keys smaller than p. For any one call to quicksort there are three possibilities as illustrated in the figure above.

1. The pivot is (equal to) either T_i or T_j, in which case T_i and T_j are compared and $A_{ij} = 1$. Since there are $j - i + 1$ keys in the interval $T_i \ldots T_j$ and since each one is equally likely to be the first in the randomly permuted input, the $P(A_{ij} = 1) = \frac{2}{j-i+1}$.
2. The pivot is a key between T_i and T_j, and T_i and T_j will never be compared; thus $A_{ij} = 0$.
3. The pivot is less than T_i or greater than T_j. Then latexmath:[T_i and T_j are sent to the same recursive call. Whether T_i and T_j are compared will be determined in some later call.

$$\begin{aligned} E[X_n] &\leq \textstyle\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[A_{ij}] \\ &= \textstyle\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= \textstyle\sum_{i=1}^{n-1} n \sum_{k=2}^{n-i+1} \frac{2}{k} \\ &\leq 2 \textstyle\sum_{i=1}^{n-1} H_n \\ &= 2nH_n \in O(n \log n) \end{aligned}$$

The last step follows by the fact that $H_n = \ln n + O(1)$.

Having completed work, let's analyze the span of quicksort. Recall that each call to quicksort partitions the input sequence of length n into three subsequences L, E, and R, consisting of the elements less than, equal to, and greater than the pivot.

Let's first bound the size of the left sequence L.

$$\begin{aligned} E[|L|] &= \textstyle\sum_{i=1}^{n-1} \frac{i-1}{n} \\ &\leq \frac{n}{2}. \end{aligned}$$

By symmetry, $E[|R|] \leq \frac{n}{2}$. This reasoning applies at any level of the quicksort call tree. In other words, the size of the input decreases by $1/2$ in expectation at each level in the call tree.

Since pivot choice at each call is independent of the other calls. The expected size of the input at level i is $E[Y_i] = \frac{n}{2^i}$.

Let's calculate the expected size of the input at depth $i = 5\lg n$. By basic arithmetic, we obtain

$$E[Y_{5\lg n}] = n\frac{1}{2^{5\lg n}} = nn^{-5\lg 2} = n^{-4}.$$

Since Y_i's are always non-negative, we can use Markov's inequality, to turn expectations into probabilities as follows.

$$P(Y_{5\lg n} \geq 1) \leq \frac{E[Y_{5\lg n}]}{1} = n^{-4}.$$

In other words, the probability that a given path in the quicksort call tree has a depth that exceeds $5\lg n$ is tiny.

From this bound, we can calculute a bound on the depth of the whole tree. Note first that the tree has exactly $n+1$ leaves because it has n internal nodes. Thus we have at most $n+1$ to consider. The probability that a given path expeeds a depth of $5\lg n$ is n^{-4}. Thus, by union bound the probability that any one of the paths exceed the depeth of $5\lg n$ is $(n+1)\cdot n^{-4} \leq n^{-2}$. Thus the probability that the depth of the tree is greater than $5\lg n$ is n^{-2}.

By using Total Expectation Theorem or the Law of total expectation, we can now calculate expected span by dividing the sample space into mutually exclusive and exhaustive space as follows.

$$\begin{aligned} E[S] &= E[S \mid \text{Depth is no greater than } 5\lg n] \cdot P(\text{Depth is no greater than } 5\lg n) + E[S \mid \text{Depth is greater than } 5\lg n] \cdot P(\text{Depth is greate} \\ &\leq \lg^2 n \cdot (1 - 1/n^2) + n^2 \cdot 1/n^2 \\ &= O(lg^2 n) \end{aligned}$$

In thes bound, we used the fact that each call to quicksort has a span of $O(\lg n)$ because this is the span of `filter`.

Here is an alternative analysis.

Let $M_n = \max\{|L|, |R|\}$, which is the size of larger subsequence. The span of quicksort is determined by the sizes of these larger subsequences. For ease of analysis, we will assume that $|E| = 0$, as more equal elements will only decrease the span. As the partition step uses `filter` we have the following recurrence for span:

$$S(n) = S(M_n) + O(\log n)$$

To develop some intuition for the span analysis, let's consider the probability that we split the input sequence more or less evenly. If we select a pivot that is greater than $T_{n/4}$ and less than $T_{3n/4}$ then M_n is at most $3n/4$. Since all keys are equally likely to be selected as a pivot this probability is $\frac{3n/4 - n/4}{n} = 1/2$. The figure below illustrates this.

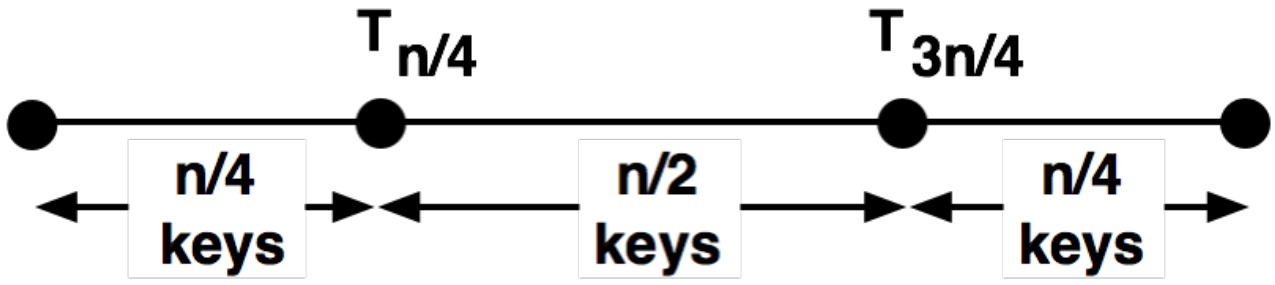


Figure 13: Quicksort span intuition.

This observations implies that at each level of the call tree (every time a new pivot is selected), the size of the input to both calls decrease by a constant fraction (of $4/3$). At every two levels, the probability that the input size decreases by latexmath:[4/3 is the probability that it decreases at either step, which is at least $3/4$, etc. Thus at a small costant number of steps, the probability

that we observe a $4/3$ factor decrease in the size of the input approches 1 quickly. This suggest that at some after $c \log n$ levels quicksort should complete. We now make this intuition more precise.

For the analysis, we use the conditioning technique for computing expectations as suggested by the total expectation theorem. Let X be a random variable and let A_i be disjoint events that form a a partition of the sample space such that $P(A_i) > 0$. The Total Expectation Theorem or the Law of total expectation states that

$$E[X] = \sum_{i=1}^{n} P(A_i) \cdot E[X|A_i].$$

Note first that $P(X_n \leq 3n/4) = 1/2$, since half of the randomly chosen pivots results in the larger partition to be at most $3n/4$ elements: any pivot in the range $T_{n/4}$ to $T_{3n/4}$ will do, where T is the sorted input sequence.

By conditioning S_n on the random variable M_n, we write,

$$E[S_n] = \sum_{m=n/2}^{n} P(M_n = m) \cdot E[S_n|(M_n = m)].$$

We can re-write this

$$E[S_n] = \sum_{m=n/2}^{n} P(M_n = m) \cdot E[S_m]$$

The rest is algebra

$$\begin{aligned} E[S_n] &= \textstyle\sum_{m=n/2}^{n} P(M_n = m) \cdot E[S_m] \\ &\leq P(M_n \leq \tfrac{3n}{4}) \cdot E[S_{\frac{3n}{4}}] + P(M_n > \tfrac{3n}{4}) \cdot E[S_n] + c \cdot \log n \\ &\leq \tfrac{1}{2} E[S_{\frac{3n}{4}}] + \tfrac{1}{2} E[S_n] \\ &\implies E[S_n] \leq E[S_{\frac{3n}{4}}] + 2c \log n. \end{aligned}$$

This is a recursion in $E[S(\cdot)]$ and solves easily to $E[S(n)] = O(\log^2 n)$.

13.1.2 Observable Work Efficency and Scalability

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 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 beyond the scope of this book. The interested reader can find more details in another book.

Fact

The randomized quicksort algorithm above has expected work of $O(n \log n)$ and expected span of $O(\log^2 n)$, 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! In other words, our quicksort appears to have "6-observed work efficiency". That means we need at least six processors working on the problem to see even a small improvement compared to a good sequential baseline.

```
[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. This plot 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
```

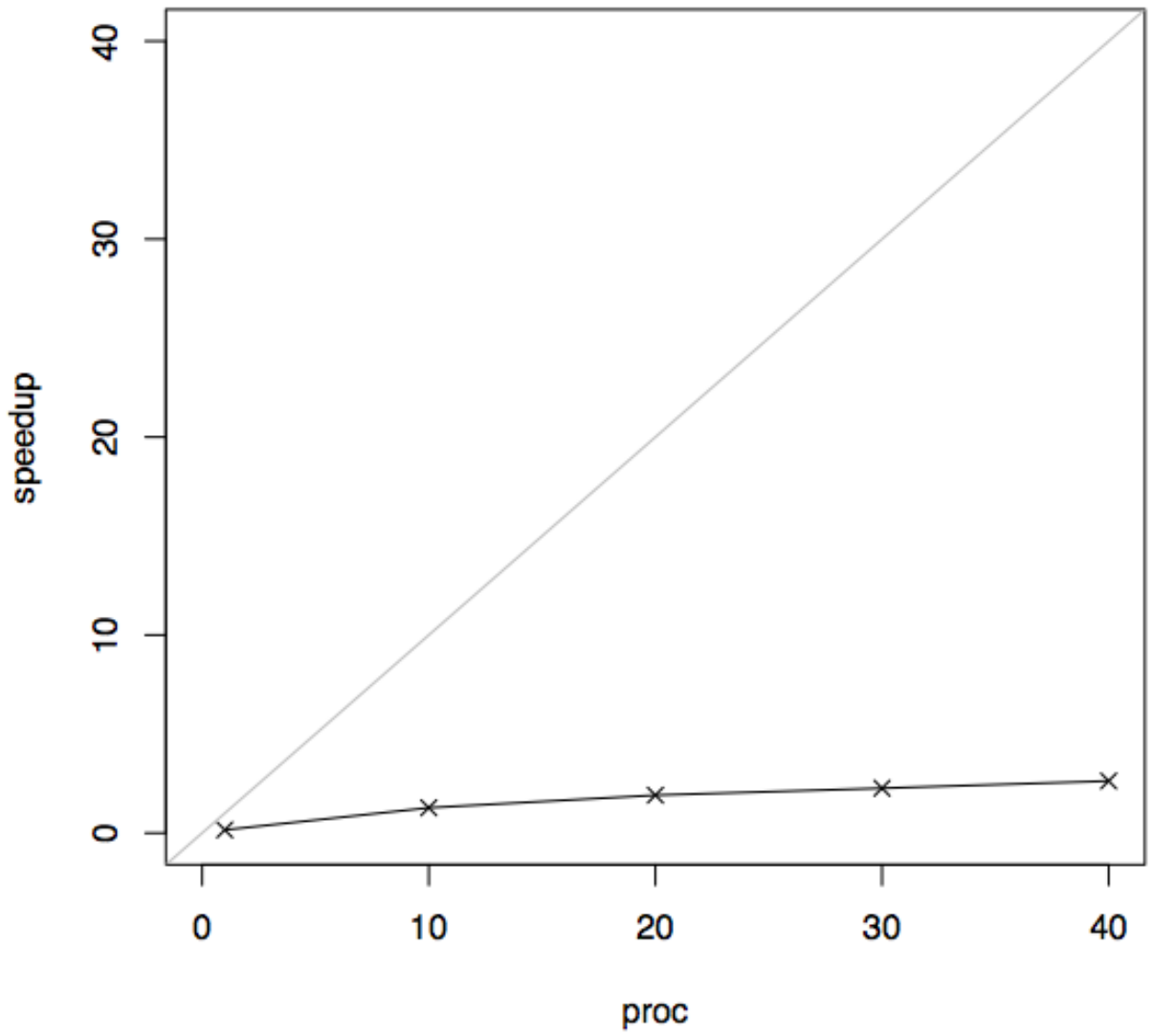


Figure 14: Speedup plot for quicksort with 100000000 elements.

Our analysis suggests that we have a good parallel algorithm for quicksort, yet our observations suggest that, at least on our test machine, our implementation is rather slow relative to our baseline program. In particular, we noticed that our parallel quicksort started out being 6x slower than the baseline algorithm. What could be to blame?

In fact, there are many implementation details that could be to blame. The problem we face is that identifying those causes experimentally could take a lot of time and effort. Fortunately, our quicksort code contains a few clues that will guide us in a good direction.

It should be clear that our quicksort is copying a lot of data and, moreover, that much of the copying could be avoided. The copying operations that could be avoided, in particular, are the array copies that are performed by each of the the three calls to filter and the one call to concat. Each of these operations has to touch each item in the input array.

Let us now consider a (mostly) in-place version of quicksort. This code is mostly in place because the algorithm copies out the input array in the beginning, but otherwise sorts in place on the result array. The code for this algorithm appears just below.

```
sparray in_place_quicksort(const sparray& xs) {
  sparray result = copy(xs);
  long n = xs.size();
```

```
  if (n == 0) {
    return result;
  }
  in_place_quicksort_rec(&result[0], n);
  return result;
}
```

```
controller_type in_place_quicksort_contr("in_place_quicksort");

void in_place_quicksort_rec(value_type* A, long n) {
  if (n < 2) {
    return;
  }
  cstmt(in_place_quicksort_contr, [&] { return nlogn(n); }, [&] {
    value_type p = A[0];
    value_type* L = A;   // below L are less than pivot
    value_type* M = A;   // between L and M are equal to pivot
    value_type* R = A+n-1; // above R are greater than pivot
    while (true) {
      while (! (p < *M)) {
        if (*M < p) std::swap(*M,*(L++));
        if (M >= R) break;
        M++;
      }
      while (p < *R) R--;
      if (M >= R) break;
      std::swap(*M,*R--);
      if (*M < p) std::swap(*M,*(L++));
      M++;
    }
    fork2([&] {
      in_place_quicksort_rec(A, L-A);
    }, [&] {
      in_place_quicksort_rec(M, A+n-M); // Exclude all elts that equal pivot
    });
  }, [&] {
    std::sort(A, A+n);
  });
}
```

We have good reason to believe that this code is, at least, going to be more work efficient than our original solution. First, it avoids the allocation and copying of intermediate arrays. And, second, it performs the partitioning phase in a single pass. There is a catch, however: in order to work mostly in place, our second quicksort code sacrified on parallelism. In specific, observe that the partitioning phase is now sequential. The span of this second quicksort is therefore linear in the size of the input and its average parallelism is therefore logarithmic in the size of the input.

Exercise

Verify that the span of our second quicksort has linear span and that the average parallelism is logarithmic.

So, we expect that the second quicksort is more work efficient but should scale poorly. To test the first hypothesis, let us run the second quicksort on a single processor.

```
$ bench.opt -bench in_place_quicksort -n 100000000 -proc 1
```

Indeed, the running time of this code is essentially same as what we observed for our baseline program.

```
exectime 12.500
total_idle_time 0.000
utilization 1.0000
result 1048575
```

Now, let us see how well the second quicksort scales by performing another speedup experiment.

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

```
[1/10]
bench.baseline -bench quicksort -n 100000000
exectime 12.031
[2/10]
bench.opt -bench quicksort -n 100000000 -proc 1
exectime 68.998
[3/10]
bench.opt -bench quicksort -n 100000000 -proc 20
exectime 5.968
[4/10]
bench.opt -bench quicksort -n 100000000 -proc 30
exectime 5.115
[5/10]
bench.opt -bench quicksort -n 100000000 -proc 40
exectime 4.871
[6/10]
bench.baseline -bench in_place_quicksort -n 100000000
exectime 12.028
[7/10]
bench.opt -bench in_place_quicksort -n 100000000 -proc 1
exectime 12.578
[8/10]
bench.opt -bench in_place_quicksort -n 100000000 -proc 20
exectime 1.731
[9/10]
bench.opt -bench in_place_quicksort -n 100000000 -proc 30
exectime 1.697
[10/10]
bench.opt -bench in_place_quicksort -n 100000000 -proc 40
exectime 1.661
Benchmark successful.
Results written to results.txt.
```

```
$ pplot speedup -series bench
```

The plot below shows one speedup curve for each of our two quicksort implementations. The in-place quicksort is always faster. However, the in-place quicksort starts slowing down a lot at 20 cores and stops after 30 cores. So, we have one solution that is observably not work efficient and one that is, and another that is the opposite. The question now is whether we can find a happy middle ground.

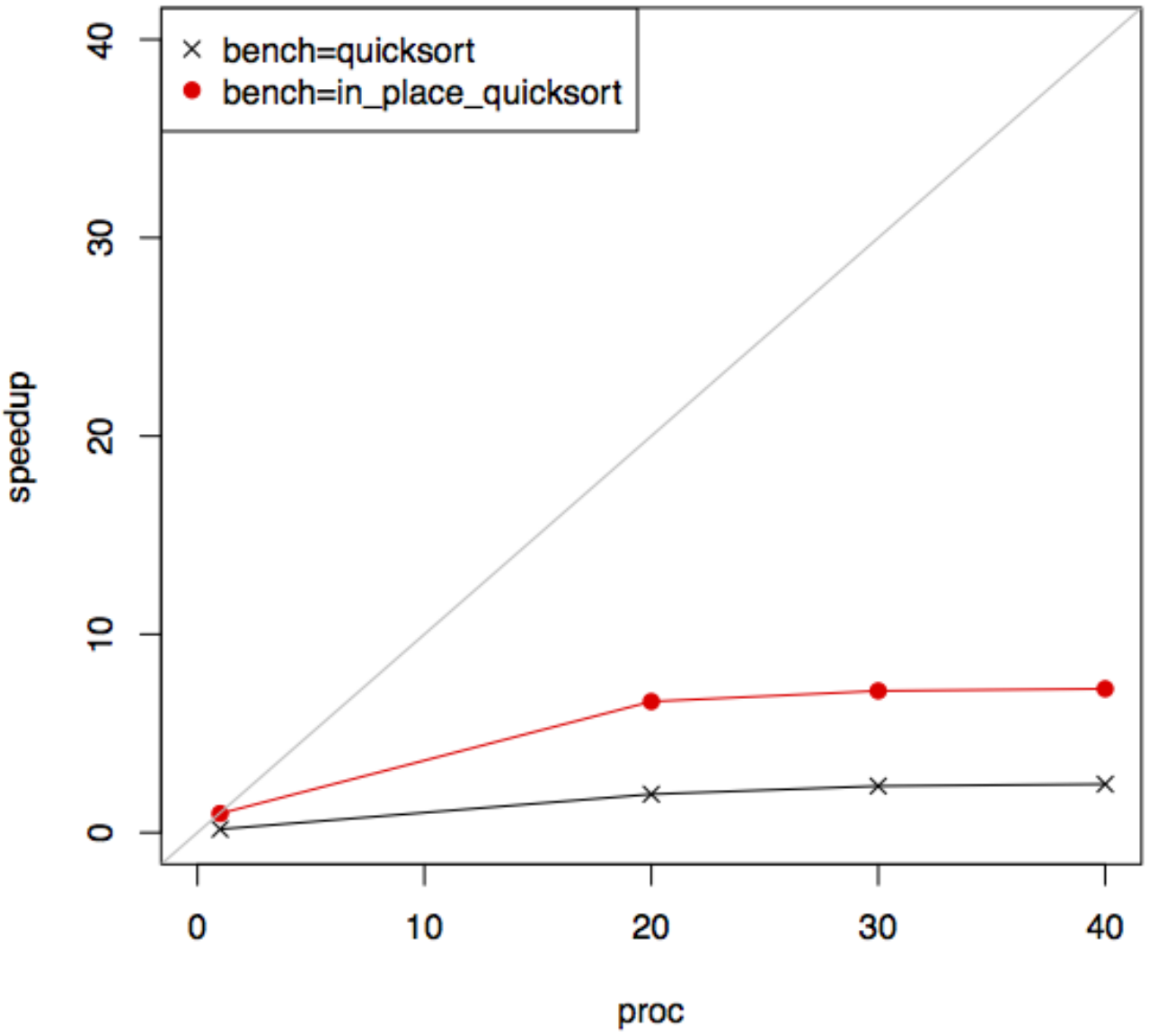


Figure 15: Speedup plot showing our quicksort and the in-place quicksort side by side. As before, we used 100000000 elements.

Question

What can we do to write a better quicksort?

Tip

Eliminate unecessary copying and array allocations.

Tip

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

Tip
Find a solution that has the same span as our first quicksort code.

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.

13.2 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 13.1 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 13.1: 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 13 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.

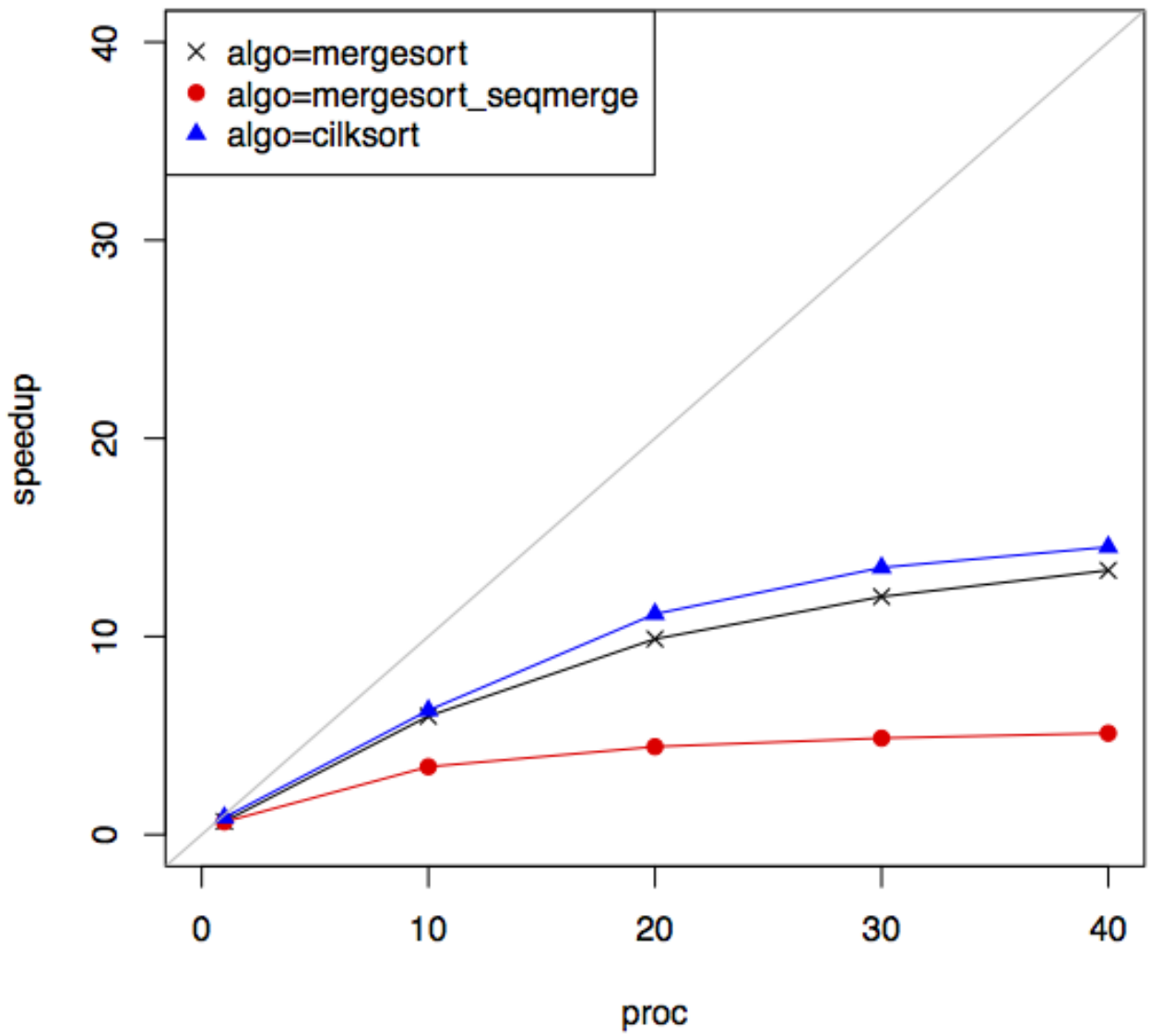


Figure 16: Speedup plot for three different implementations of mergesort using 100 million items.

It turns out that we can do better by simply changing some of the variables in our experiment. The plot shown in Figure 14 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.

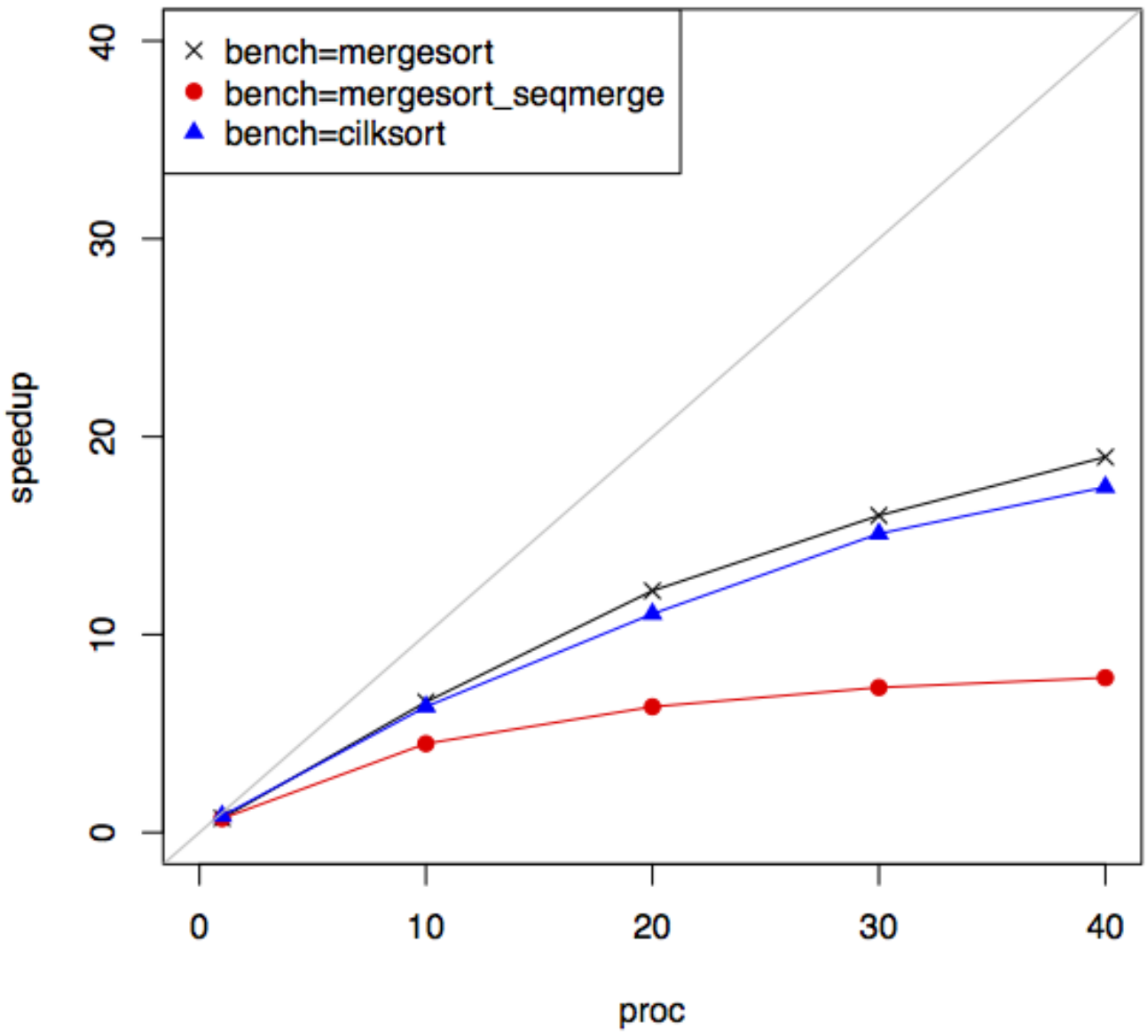


Figure 17: Speedup plot for three different implementations of mergesort using 250 million items.

Question: Stable Mergesort

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?

14 Graph Theory

This chapter is a brief overview of some of the graph terminology used in this book. For the definitions, we assume undirected graphs but the definitions generalize straightforwardly for the case of directed graphs.

14.1 Walks, Trails, Paths

A ***walk*** in a graph is a sequence of alternating vertex-edge pairs, $v_0, e_1, v_1, \ldots, v_{n-1}, e_n, v_n$, such that $e_i = (v_{i-1}, v_i) \in E$. Note that vertices and edges can be repeated. A walk corresponds to the intuitive notion of "taking a walk in the graph."

In a simple graph, where there are no parallel or multiple edges, we can specify a walk as a sequence of vertices (which imply) the edges.

A ***trail*** in a graph is a walk where all edges are distinct, that is no edge is traversed more than once. Note that there can be repeated vertices in a trail. A trail corresponds to the intuitive notion of the "trail" of a walk.

A ***path*** is a graph is a walk where all edges and vertices are distinct.

We say that a walk or a trail is ***closed*** if the initial and terminal vertices are the same.

14.2 Euler Tours

Given an undirected graph $G = (V, E)$, an ***Euler trail*** is a trail that uses each edge exactly once. An ***Euler tour*** is a trail that is closed. In other words, it starts and terminates at the same vertex. We say that a graph is ***Eulerian*** if it has an Euler tour.

It can be proven that a connected graph $G = (V, E)$ is Eulerian if and only if every vertex $v \in V$ has even degree.

14.3 Trees

A ***tree*** is an undirected graph in which any two of vertices are connected by exactly one path.

A ***forest*** is a set of disjoint trees.

A ***rooted tree*** is a tree with a designated root.

A rooted tree may be directed by pointing all edges towards the root or away from the root. Both are acceptable.

For a rooted tree, we can define the a parent-child relationship between nodes. If a node u is the first node on the path that connects another node v to the root, then u is the parent of v and v is the child of u.

An ***ordered tree*** is a tree where the children of each node are totally ordered.

14.3.1 Binary Trees

A ***binary tree*** is a rooted, directed, ordered tree where each node has has at most two children, called the ***left*** and the ***right child***, corresponding to the first and the second respectively.

For a binary tree, we can define couple of different kinds of traverses. An ***in-order traversal*** traverses the binary tree by performing an in-order traversal of the left subtree, visiting the root, and then performing an in-order traversal of the right subtree. An ***post-order traversal*** traverses the binary tree by performing an post-order traversal of the right subtree, visiting the root, and then performing an post-order traversal of the right subtree. An ***pre-order traversal*** traverses the binary tree by visiting the root, performing an pre-order traversal of the left subtree, and then performing an pre-order traversal of the right subtree.

A ***full binary tree*** is a binary tree, where each non-leaf node has degree exatly two.

A ***complete binary tree*** is a full binary tree, where all the levels except possibly the last are completely full.

A ***perfect binary tree*** is a full binary tree where all the leaves are at the same level.

15 Chapter: Tree Computations

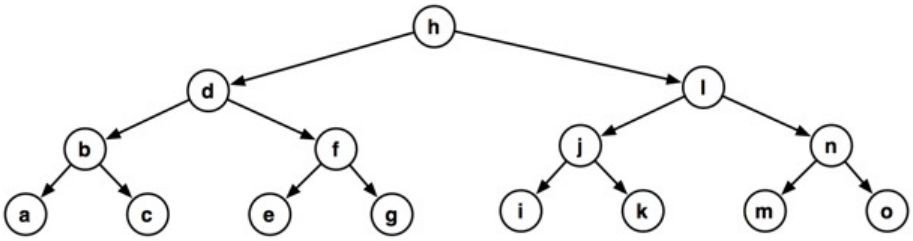
Trees are a basic structure for representing relations. It is thus natural to ask whether we can compute various properties of trees in parallel.

15.1 Computing In-Order Traversals with Divide-and-Conquer and Contraction

As our first motivating example, let's consider the following question: we are given a binary tree T and asked to determine the ***in-order rank*** of each node in the tree, defined as the order of the node in an in-order traversal of the tree. We shall use this example to develop several key ideas.

As two important special cases of this problem, let's consider a complete binary tree, where each internal node has exactly two children and a chain, where each internal node has exactly one child. A complete binary tree is a balanced tree whereas a chain is an unbalanced tree.

Example 15.1 In-Order Traversal



An in-order traversal of the complete binary tree above traverses the nodes in the following order: **a,b,c,d,e,f,g,h,i,j,k,l,m,n,o**.

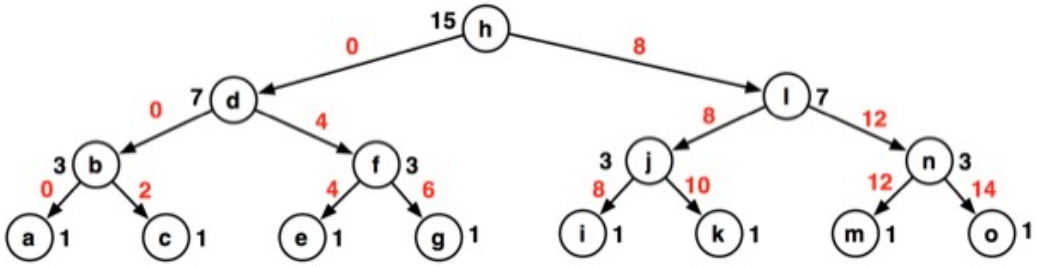
Let's apply two algorithm design techniques to this problem: divide and conquer and contraction.

For divide and conquer, we can use a two-pass algorithm:

1. First pass: compute the size of each subtree in the tree.
2. Second pass: Compute the in-order ranks by traversing the tree from root to leaves as we propagate to each subtree in-order rank of its root, which can be calculated based on the sizes of the left and the right subtrees.

The first phase of the divide-and-conquer algorithm proceeds by computing the recursively the sizes of each subtree in parallel, and the computing the size for the tree by adding the sizes and adding one for the root. Similarly, the second phase of the algorithm computes the rank of the root as the size of the left-subtree (assuming we are counting rank's from zero) plus an offset (initially zero), and recurses on the left and the right subtrees. For the left subtree, the offset is the same as the offset of the root and for the right subtree, the offset is the calculated by adding the size of the left subtree plus one.

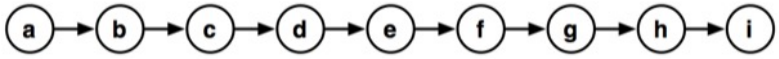
Example 15.2 Two-Pass Algorithm for Computing Ranks



In the example tree above, each node is labeled with the size of its subtree computed in the first pass and each edge is labeled with the offset computed in the second pass.

If the tree is balanced, as for example in our example, we can show that this algorithm performs $O(n)$ work in $O(\lg n)$ span. If however, the tree is not balanced the algorithm can do poorly, requiring as much as $O(n)$ span.

Example 15.3 Divide and Conquer on Chains



An in-order traversal of a chain of nodes yields **a,b,c,d,e,f,g,h,i** assuming that the left child of each node is "empty." The divide-and-conquer algorithm would require linear span on this example.

Another technique we have seen for parallel algorithm design is contraction. Appling the idea behind this technique, we want to "contract" the tree into a smaller tree, solve the problem for the smaller tree, and "expand" the solution for the smaller tree to compute the solution for the original tree.

There are several ways to contract a tree. One way is to "fold" or "rake" the leaves to generate a smaller tree. Another way is to "compress" long branches of the tree removing some of the nodes on such longe branches.

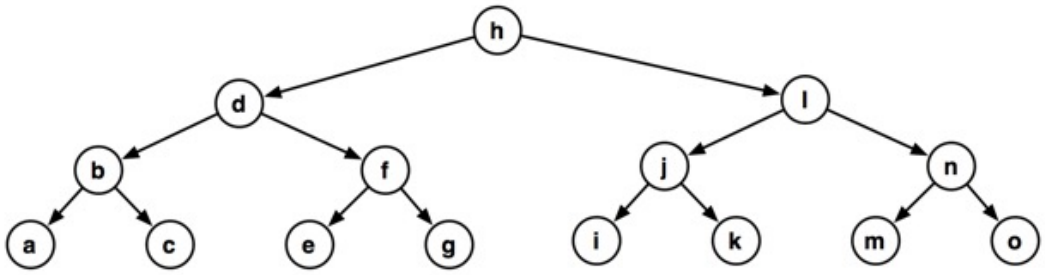
15.1.1 Rake Operation

Lets define a ***rake*** as an operation that when applied to a leaf deletes the leaf and stores its size in its parent. With some care, we can rake all the leaves in parallel: we just need to have a place for each leaf, sometimes called a ***cluster***, to store their size at their parent so that the rakes can be performed in parallel without interfence.

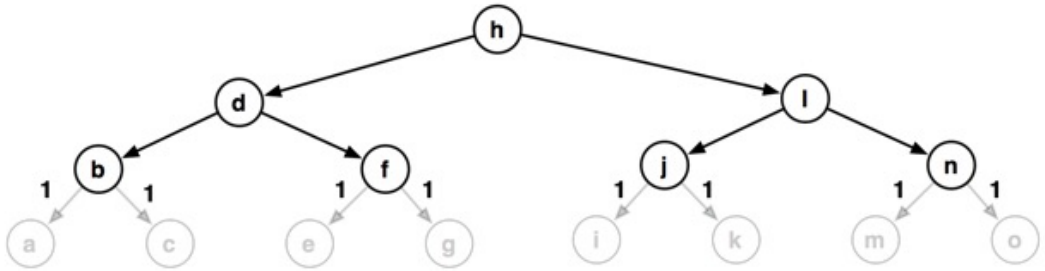
Using the rake operation, we can give an algorithm for computing the in-order traversal of a tree:

1. **Base case:** The tree has only one node, compute the result.
2. **Contraction step:** Rake all the leaves to contract the tree.
3. **Recursive step:** Solve the problem for the contracted tree.
4. **Expansion step:** "Reinsert" the raked leaves to compute the result for the input tree.

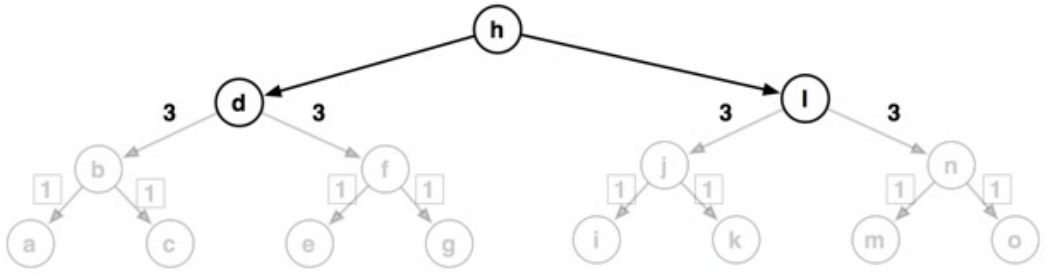
Example 15.4 Contraction with Rake



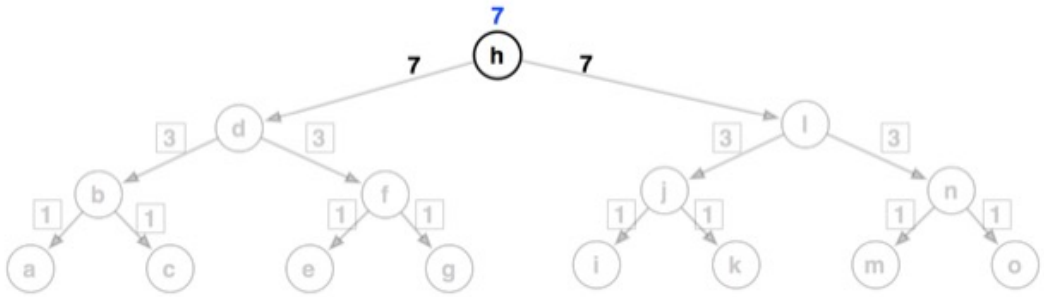
Given the example tree above, we rake all the leaves and store their size in clustered stored in the parents. For the drawings we draw the clusters on the edges.



We can rake the leaves again.

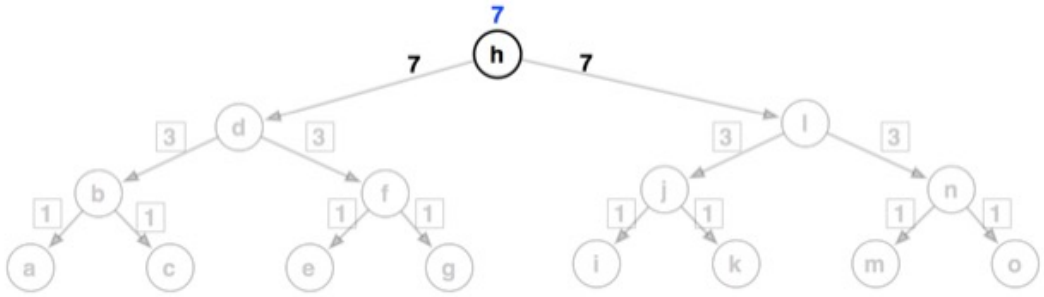


And again, until we reach a tree with a single node. We can then compute the rank of the node as the size of its subtree.

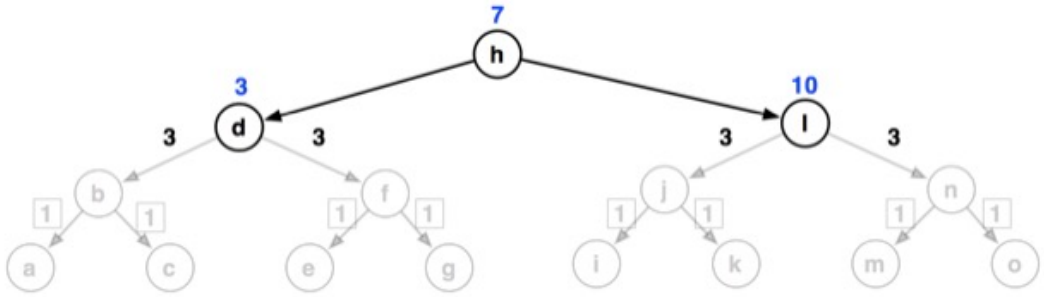


Example 15.5 Expansion with Rake

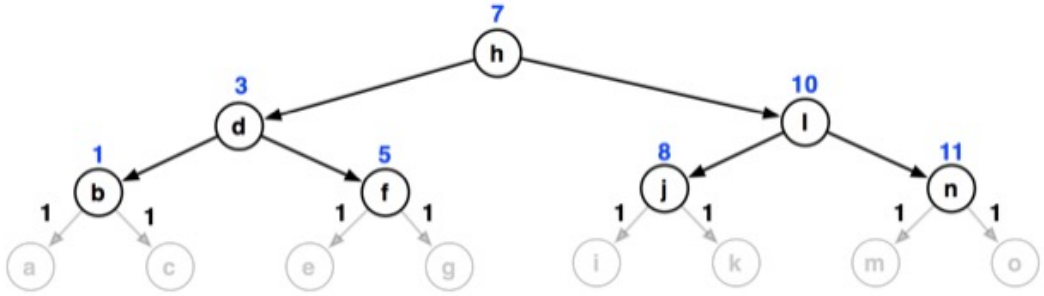
Having reached the base case by contractions, we then perform expansion steps to compute the result for the original tree.



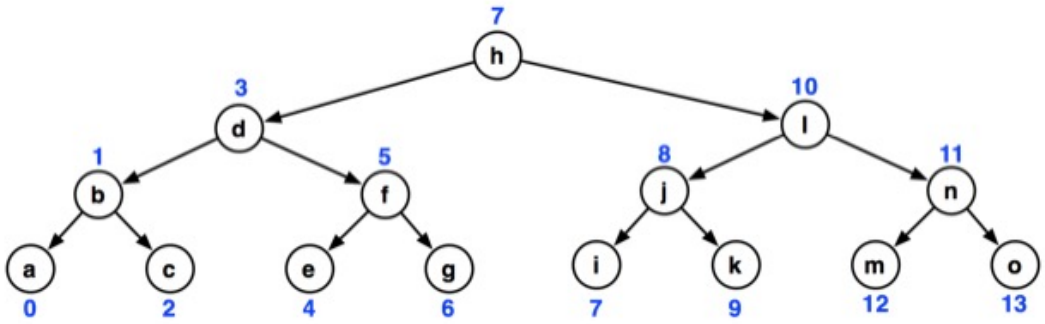
As we expand, we compute the in order traversal for the tree constructed by "expanding" the tree out.



Each expansion corresponds to the a contraction step.



The expansion step completes when all the contractions are reversed.



Let's calculate the work and span of this algorithm assuming that we have a complete binary tree, which is a tree where all internal nodes have exactly two children and all the leaves are collected at the last two levels of the tree. Since a complete binary tree is a full binary tree, raking all the leaves removes half of the nodes. Thus we can write the work recursion as $W(n) = W(n/2) + n$. This solves to $W(n) = O(n)$. Since complete trees are balanced, for the span we have $S(n) = S(n/2) + \log n$. This solves to $S(n) = O(\lg^2 n)$.

15.1.2 Compress Operation

The contraction algorithm based rake operations performs well for complete binary trees but on unbalanced trees, the algorithm can do verp poorly. In fact, the contraction algorithm requires $O(n^2)$ work and $O(n^2)$ span on a chain, which is a degenerate tree.

Let's consider the worst-case example of the chain and design a different contraction algorithm that works well just on chains.

Define ***compress*** as an operation that when applied to a node u with a single child v and parent w deletes u and the edges incident on it and inserts an edge from the parent to the child, i.e., (w,v). To incorporate into the computation the contribution of the compressed vertex, we can construct a ***cluster***, which for example, can be attached to the newly inserted edge. For the in-order traversal example, this cluster will simply be a weight corresponding to the size of the deleted nodes. Initially all edges have weight 0.

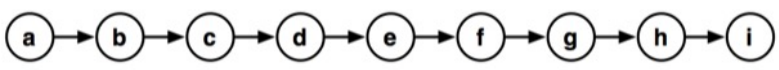
Using compress operation, we wish to be able to contract a tree to a smaller tree in parallel. Since a compress operation updates the two neighbors of a compressed node, we need to be careful about how we apply these operations. One way do this is to select in each round an independent set of nodes (nodes with no edges in between) and compress them.

Based on this idea, we can give an algorithm for computing the in-order traversal of a chain:

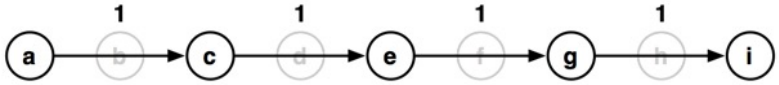
1. **Base case:** The chain consists of a single edge, compute the result.
2. **Contraction step:** Compress an independent set of internal nodes to obtain a contracted chain.
3. **Recursive step:** Solve the problem for the contracted chain.
4. **Expansion step:** "Reinsert" the compressed nodes to compute the result for the input chain.

To maximize the amount of contraction at each contraction step, we want to select a maximal independent set and do so in parallel. There are many ways to do this, we can use a deterministic algorithm, or a randomized one. Here, shall use randamization. The idea is to flip for each node a coin and select a vertex if it flipped heads and its child flipped tails. This idea of using randomization to make parallel decisions is sometimes called ***symmetry breaking***.

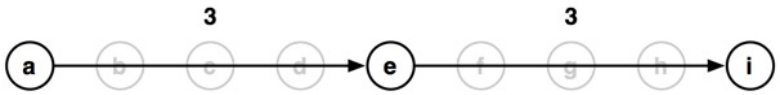
Example 15.6 Contraction with Compress



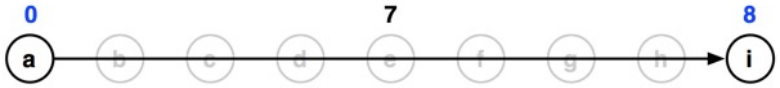
In the example chain above, we compress every other internal node to obtain the new chain below.



We then repeat this process.

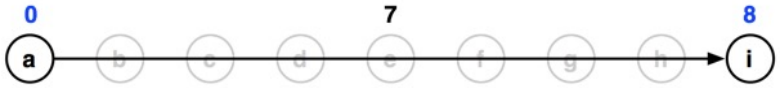


And repeat it again until we are down to a single edge at which point we can solve the problem.

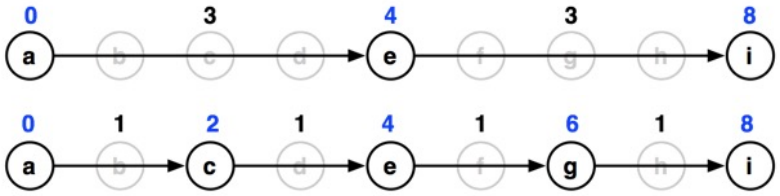


Example 15.7 Expansion with Compress

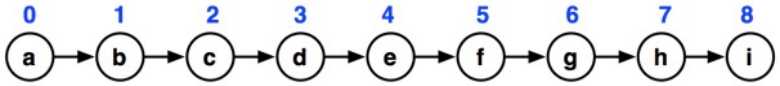
Having contracted the chain down to a single edge, we start the expansion phase, where we "re-insert" the deleted nodes to compute the result for the expanded tree.



At each expansion step, to compute the result for the re-inserted nodes, we use the pre-computed result for the clusters adjacent to them.



Expansion completes when we expand back the input tree.



Since the algorithm is randomized its analysis requires some care.

For the analysis, we shall make two assumptions:

1. all rounds are independent: the source of randomness in each round is fully independent of the other rounds, and
2. coin flips are pairwise independent: the outcome of one coin flip does not affect that of any other coin flip.

Let F_i be a random variable denoting the fraction of internal nodes of the chain that are not compressed after round i. We know that the probability that a node is compressed is $1/4$. We thus know that $E[F_i] = \frac{m-m/4}{m} = 3/4$ for all i.

Let X_i be a random variable denoting the size of the tree at the beginning of round i.

We have

$$X_i = \left(\Pi_{j=0}^{i-1} F_i\right) \cdot n.$$

Since all rounds are independent, F_i's are independent and thus we can write:

$$E[X_i] = \left(\Pi_{j=0}^{i-1} E[F_i]\right) = \left(\frac{3}{4}\right)^{i-1} \cdot n.$$

Since we perform constant work per node in each round, we can bound the work as follows.

$$\begin{array}{lcl} W(n) & \leq & \sum_{i=1}^{\infty} X_i \\ E[W(n)] & \leq & \sum_{i=1}^{\infty} \frac{3^{i-1}}{4^{i-1}} \cdot n \\ E[W(n)] & \leq & O(n). \end{array}$$

Note that for this bound, we made the conservative assumption that the computation continues infinitely. This still gives us a tight bound because we the size of the input decreases geometrically. We thus conclude that the algorithm is work efficient.

To bound the span, we need a high-probability bound. Let's consider the probability that the number of nodes is non-trivial after $9\lg n$ rounds, that is at the beginning of round $r = 9\lg n + 1$. By Markov's inequality we have

$$\begin{aligned} P[X_r \geq 1] &\leq E[X_r]/1 = \\ &\leq \left(\tfrac{3}{4}\right)^{9\lg n} \\ &\leq n^{-3}. \end{aligned}$$

This means that after $9\log n$ rounds, the probability that the tree constains more than a few nodes is tiny. We can thus state with high probability that the algorithm terminates after $O(\log n)$ rounds. If it does not, we know that the span is no more than linear in expectation, because the algorithm does expected linear work. Let R a random variable denoting the span of the algorithm, we can thus write span as follows.

$$S(n) = \begin{cases} R & \text{if } R \leq 9\lg n \\ W(n) & \text{otherwise} \end{cases}$$

By conditioning on a random variable R denoting the number of rounds we can bound span as follows.

$$\begin{aligned} E[S(n)] &\leq E[S(n)|R \leq 9\lg n] \cdot P[R \leq 9\lg n] + E[S(n)|R > 9\lg n] \cdot P[R > 9\lg n] \\ &\leq 9\lg n \cdot (1 - 1/n^3) + n \cdot 1/n^3. \\ &\leq O(\lg n). \end{aligned}$$

15.2 Tree Contraction

In this chapter thus far, we have seen that we can compute the in-order rank a complete binary tree, which is a perfectly balanced tree, by using a contraction algorithm that rakes the leaves of the tree until the tree reduces to a single vertex. We have also seen that we can compute in-order ranks of nodes in a worst-case unbalanced tree, a chain, by using a contraction algorithm that compresses nodes with single child's until the tree reduces to a single edge.

We will now see that we can in fact compute in-order ranks for any tree, balanced or unbalanced, by simultaneously applying the same two operations recursively in a number of rounds. Each round of application rakes the leaves and selects an independent set of nodes to compress until the tree contracts down to a single node. After the contraction phase completes, the expansion phase starts, proceeding in rounds, each of which reverses the corresponding contraction round by reinserting the compressed and raked nodes and computing the result for the corresponding tree.

Algorithm: Tree Contraction

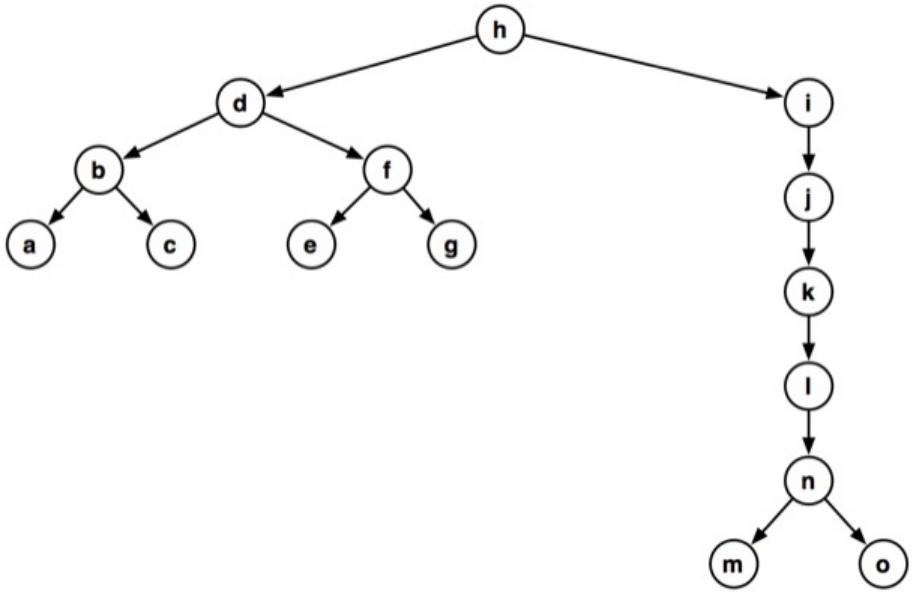
```
tree_contract (T)
if |T| = 1
  solve/expand T
else
    T' = rake the leaves and an independent set of nodes with a single, non-leaf child ↩
        in T
    tree_contract T'
    expand to T
```

To select the independent set of single-child nodes to compress, we use the same randomization technique as in chains: each node flips a coin and a single-child node is selected if it flips heads and its child flips tails.

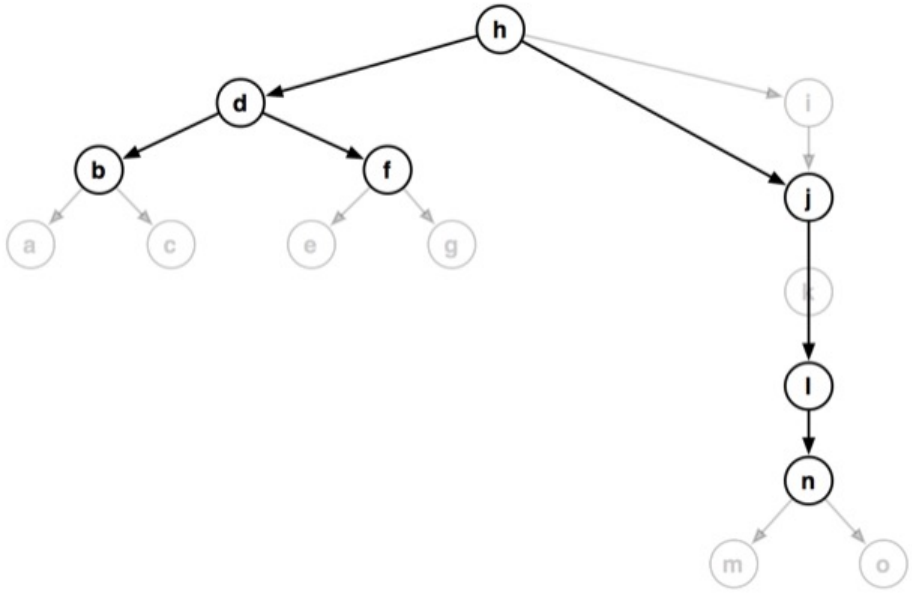
Since expansion is symmetric to contraction and since we have already discussed expansion in some detail, in the rest of this chapter, we shall focus on contraction.

Example 15.8 Tree Contraction

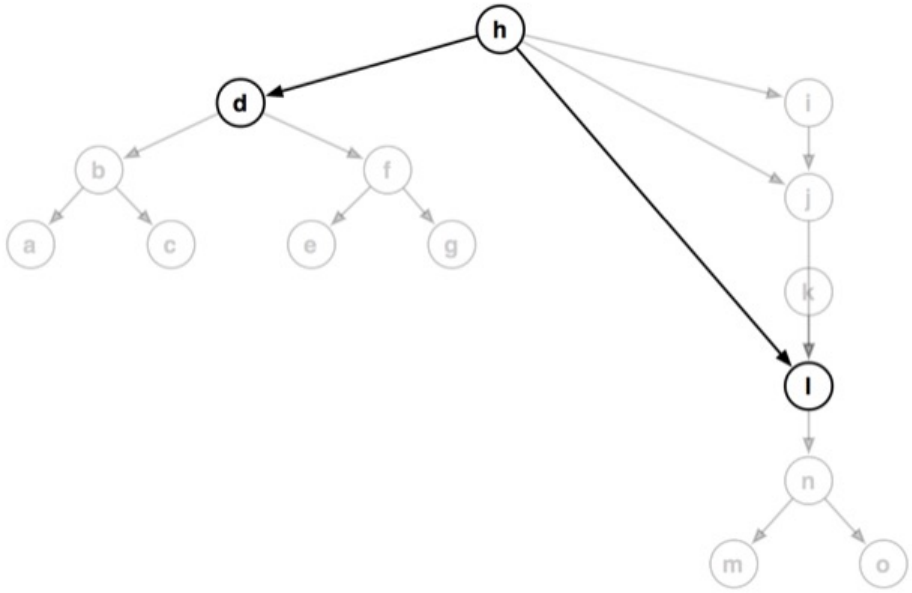
An example tree contraction illustrated on the input tree below. Random coin flips are not illustrated.



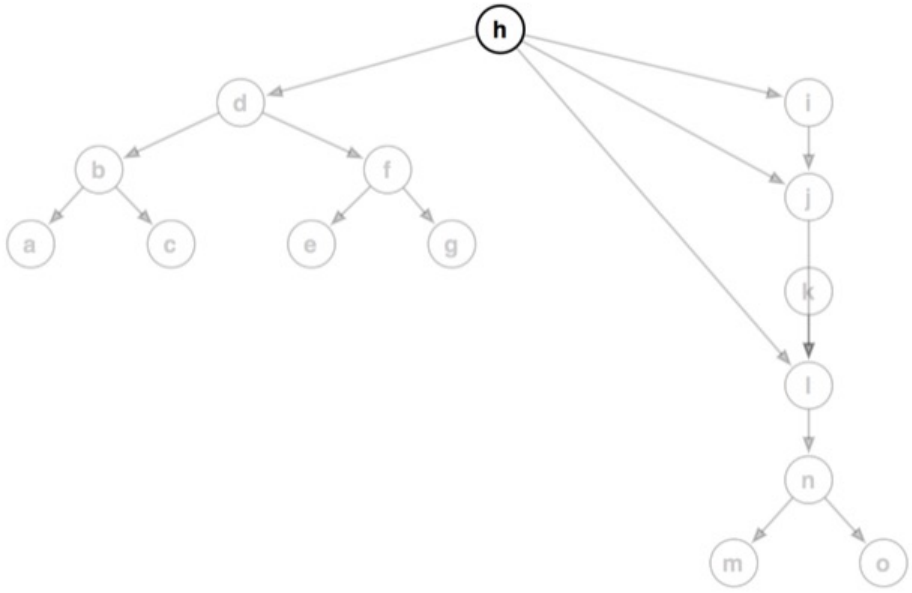
The first round rakes the nodes **a, c, e, g, m, o** and compresses **i, k**, resulting in the tree below.



The second round rakes the nodes **b, f, n** and compresses **j**, resulting in the tree below.



The third round rakes the nodes **d, l** and compresses no nodes, resulting in a tree with a single node, thus completing the contraction phase.



To analyze the tree-contraction algorithm, we first establish a basic facts about binary trees.

Lemma[Branches and Leaves]

Consider any binary tree and let ℓ denote the number of leaves in the tree and let b denote the number of branches or internal nodes with two chilren. We have $\ell = b + 2$

Proof

The proof is by induction. In the base case, the tree consists of a single node and thus $\ell = 1, b = 0$ and the lemma holds. Assume that the lemma holds for all tree up to and including $n > 1$ and consider a tree with $n+1$ nodes.

We have two cases to consider. In the first case, the root has a single child. In this case, we know by induction that for the subtree rooted at the child of the root, $\ell' = b' + 1$. Since $\ell = \ell' = b' + 1 = b + 1$. In the second case, we know that $\ell_1 = b_1 + 1$ for the left subtree and know that $\ell_2 = b_2 + 1$ for the right subtree. Thus, we knaw that know that $\ell = \ell_1 + \ell_2 = b_1 + b_2 + 2$. Since $b = b_1 + b_2 + 1$, we have $\ell = b + 1$.

Lemma [Contraction Ratio]

Consider a tree with n nodes, one round of tree contraction yields a tree which has no more than $3n/4$ nodes in expectation.

Proof

Partition the nodes in T into four disjoint sets

1. R: singleton set containing the root.
2. V_0: set of nodes with 0 children, i.e., the leaves.
3. V_1: set of nodes with 1 child.
4. V_2: set of nodes with 2 children.

Let's further partition V_1 into three disjoint sets:

1. V_{10}: set of nodes with 1 child, where the child has no other children.
2. V_{11}: set of nodes with 1 child, where the child has one child.
3. V_{12}: set of nodes with 1 child, where the child has two children.

By construction, we know the set of nodes V of the tree can be written as

$$V = R \cup V_0 \cup V_{10} \cup V_{11} \cup V_{12} \cup V_2.$$

Let's consider the nodes in the set $V_{11} \cup V_{12}$. These are exactly the nodes an independent subset of which we compress. What fraction of them are compressed, i.e., deleted? The probability that a particular node is selected as part of the independent set is $1/4$. Thus $3/4$ fraction of these vertices remain live at the end of this round.

Let's now consider the remaining nodes, i.e., $R \cup V_0 \cup V_{10} \cup V_2$.

We have the following relationships between these sets.

1. $|V_{10}| \leq |V_0|$.
2. $|V_2| + 1 = |V_0|$ (by the Branches and Leaves Lemma.

Since R is a singleton, we have $|R \cup V_0 \cup V_{10} \cup V_2| \leq 3|V_0|$.

Since all the leaves are raked, a fraction $2/3$ of these vertices remain live at the end of this round.

We have thus partitioned the set of all nodes into two subsets and showed that a fraction of at most $3/4$ (expected) and $2/3$ remain live at the end of the round. We thus conclude that in expectation $3/4$ of the nodes remain live.

Exercise

Improve the constant factor to $2/3$ by changing the way compress warks.

Theorem[Work and Span of Tree Contraction]

Assuming that all rake, compress, and expansion operations complete in $O(1)$ work, tree contraction performs $O(n)$ expected work in $O(\lg n)$ expected span.

Proof

The proof of this theorem is essentially the same as the proof for chains given above. For brevity, we don't repeat the proof.

15.3 Applications of Tree Contraction

In order to apply tree contraction to solve a particular problem, we need to determine how various operation in tree contraction manipulate the application data, specifically the following:

1. the computation performed by a rake operation,
2. the computation performed by a compress operation,
3. the computation performed for expanding singleton tree,
4. the computation performed for expanding raked nodes, and
5. the computation performed for expanding compressed nodes.

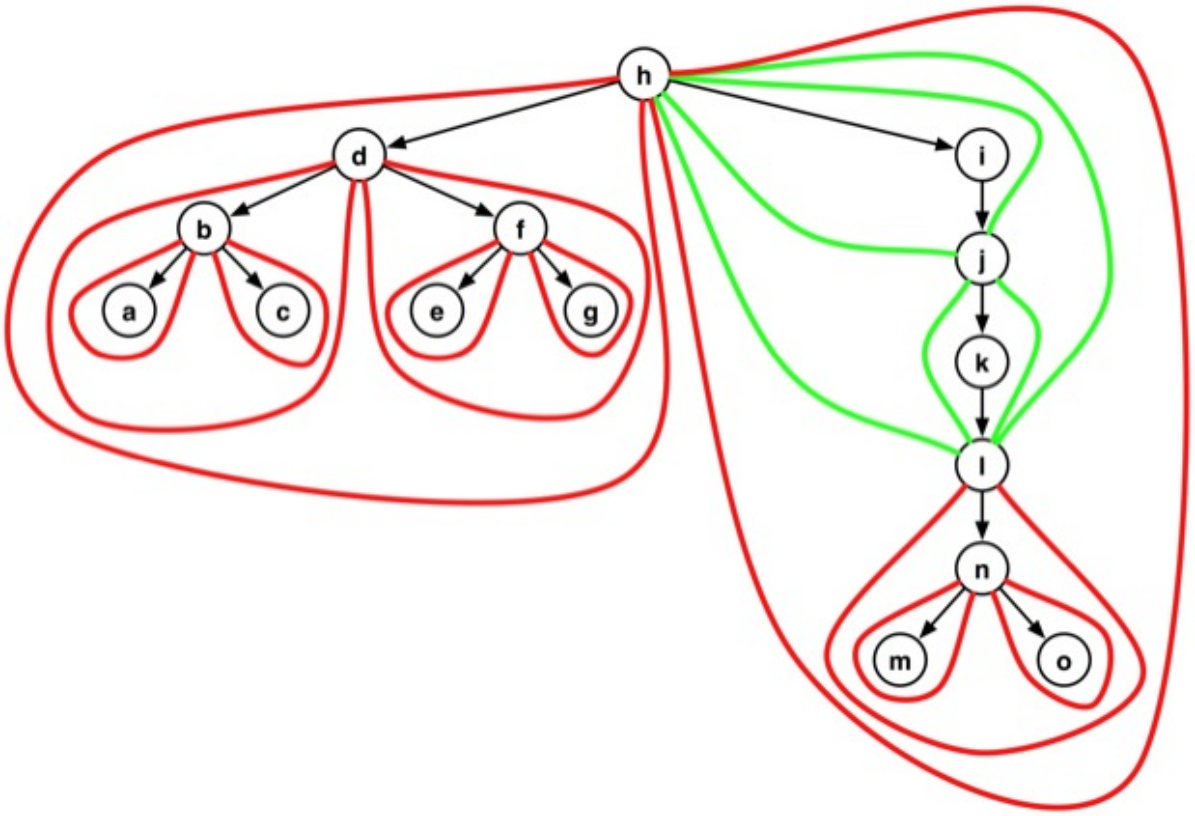
To specify these it is often helpful to view the tree contraction algorithm as a hierarchical clustering of the input tree T.

1. We view a rake operation as constructing a ***unary cluster***, which correspond to a subtree rooted at a node u of the tree T along with the edge between u and its parent; we refer to u as the boundary of the unary cluster. The simplest unary cluster consists of a leaf in the tree and the edge from the parent.
2. We view the compress operation as construction a ***binary cluster***, which corresponds to a subtree of the tree T that is defined by two nodes u and v where v is a proper descendant of u. Such a binary cluster consists of the subtree induced by the all the proper descendants of u except for the descendants of v and the edge that connects v to the parent and the edge that connects u to its child.

Based on this view, for a tree where each node has at most d children, we can specify a rake operation as constructing a unary cluster by combining $d-1$ unary clusters and one binary cluster by also taking into account the label of the node being raked. Similarly, we can specify a compress operation as constructing a binary cluster by combining $d-1$ unary clusters with two binary clusters by also taking into account the label of the node being compressed.

Example 15.9 Hierarchical Clustering via Tree Contraction

The figure below illustrates a hierarchical clustering of the example tree from the example above. Clusters constructed during earlier rounds are nested inside those constructed in later rounds. Each edge of the tree represents a binary cluster and each node represents a unary cluster.



This clustering can be represented as a balanced tree of clusters. We can thus say that tree contraction maps an arbitrary possibly unbalanced trees to balanced trees (of clusters).

Exercise

Draw the tree representing the hierarchical clustering. Why is this cluster tree balanced?

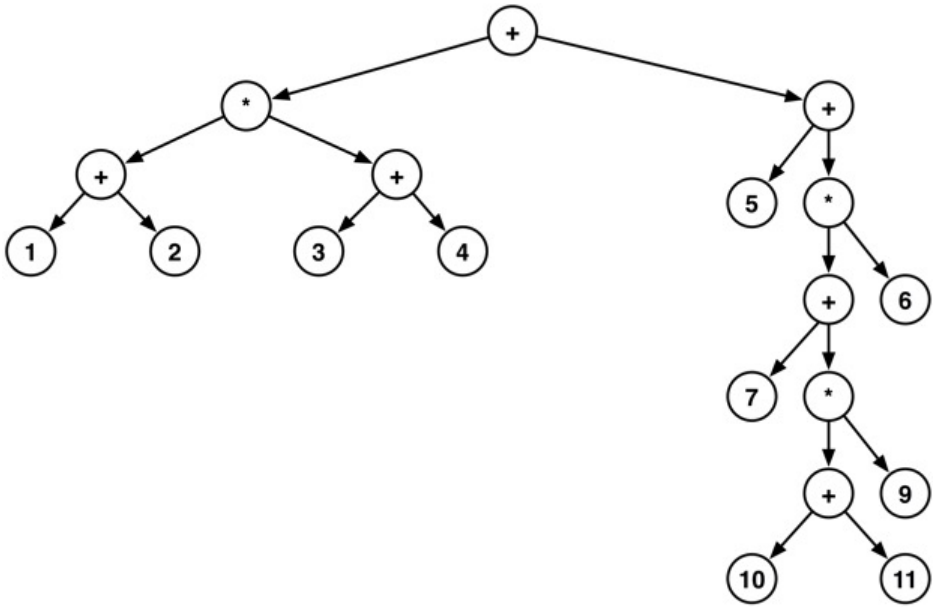
A classic application of tree contraction is the expression trees problem. In this problem, we use a tree to represent a mathematical expression and are asked to compute the value the expression. To solve this problem, we can use tree contraction with rake and compress operations. To this end, we first need to determine the definitions for the unary and binary clusters. Let's assume for this example that the expression is the addition and multiplication of a number of integers as can be represented by a tree where leaves are integers and internal nodes are plus or multiplication operations as shown in the example below.

Example 15.10 Tree Contraction for an expression tree

Consider the expression

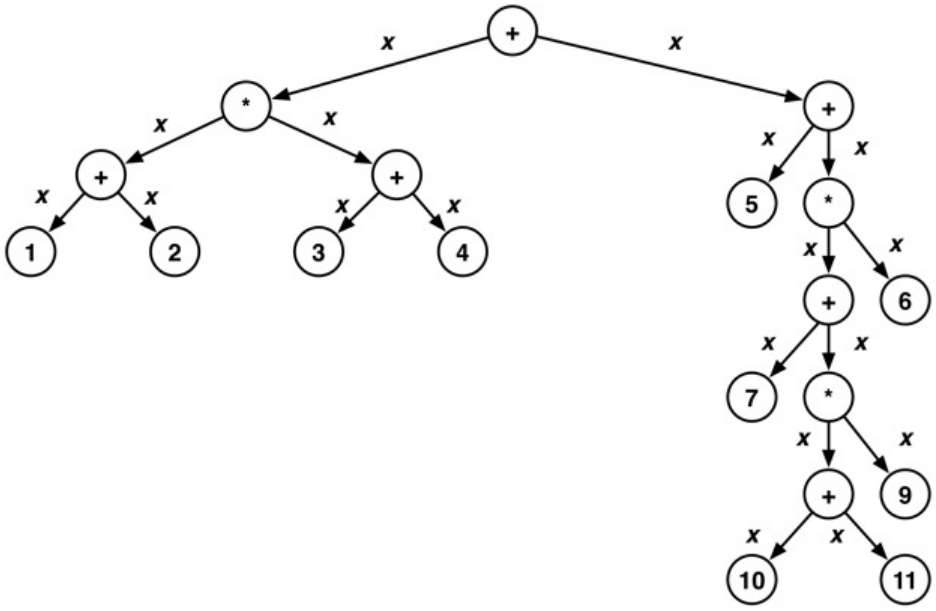
$$(1+2)\cdot(3+4)+((5+(7+((10+11)\cdot 9))\cdot 6)\,.$$

The expression tree shown below represents this expression

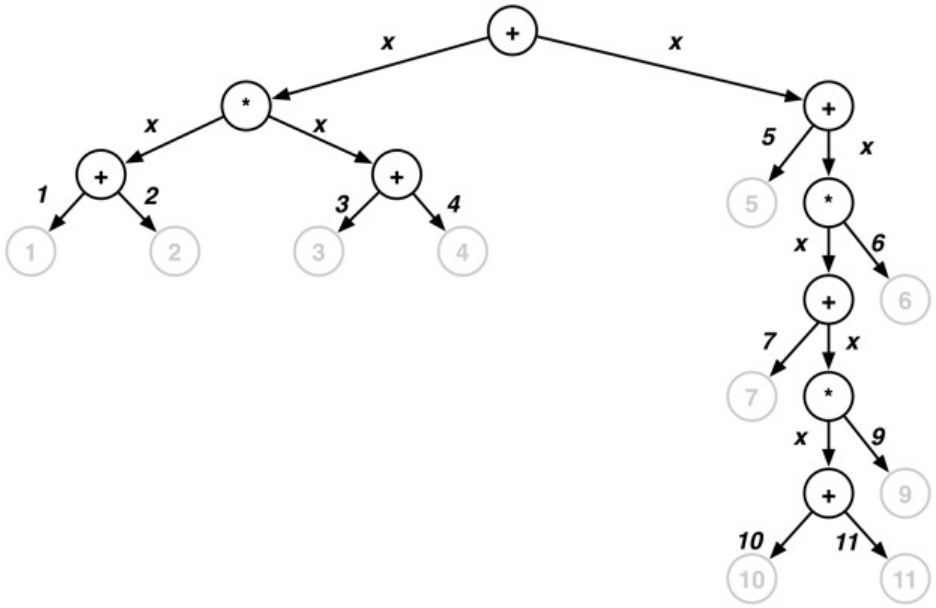


By remembering that unary clusters represent subtrees of the tree rooted at a node, we can see that a natural definition for a unary cluster is the value of the corresponding subexpression. Determining the value for a binary cluster is a bit more tricky. Recall that a binary cluster is a sub-tree induced by a set of nodes between two nodes in the tree. What should such a structure reduce to? After some reflection, we can see that such a subtree "transforms" the value at its lower boundary node by applying a linear transformation of the form $a \cdot x + b$, where a, b are integers.

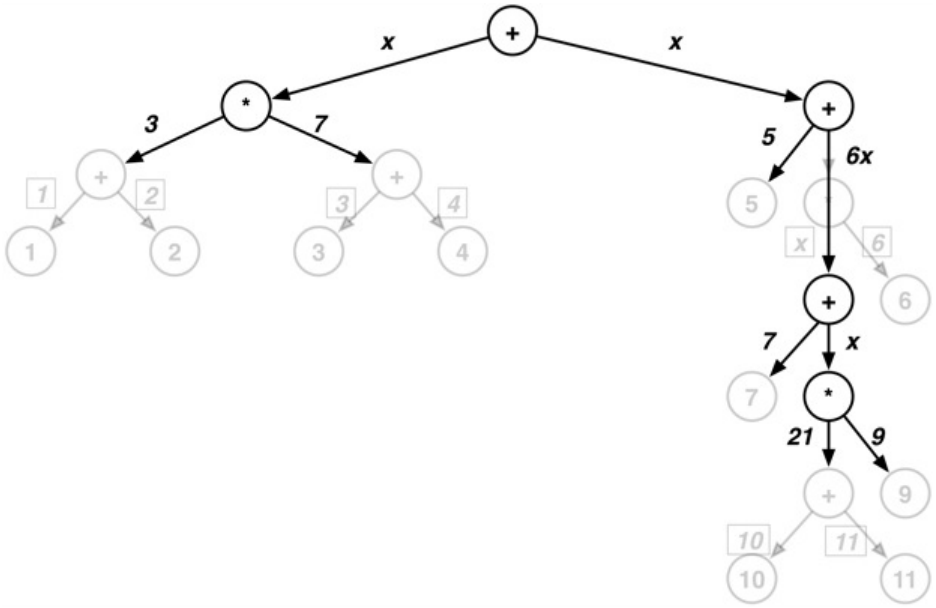
To apply tree contraction, we first annotate the initial values for each binary cluster, which is the identity functions written as x. The initial values for unary clusters can be defined as the value stored at the leaf.



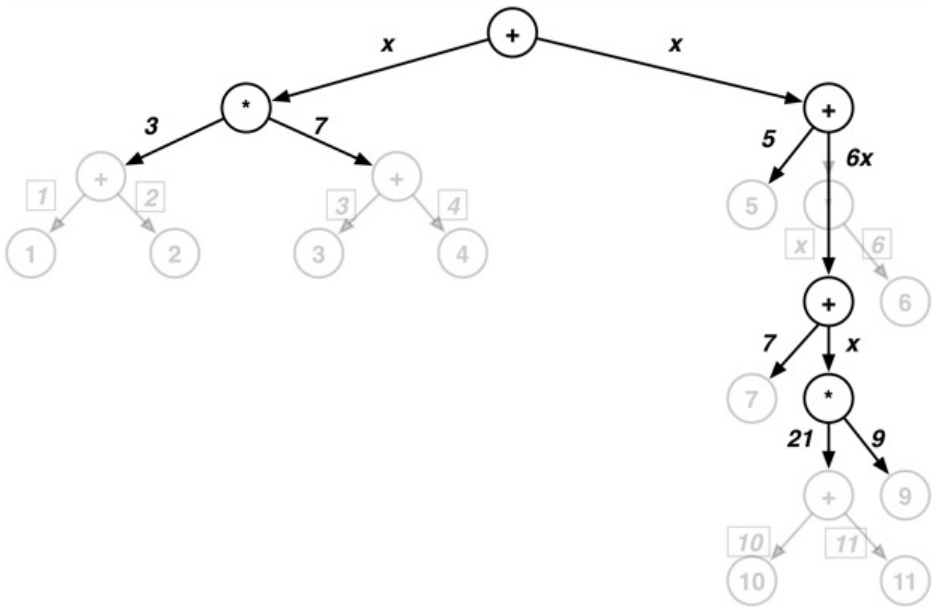
In the first round, we perform a number of rakes and update the corresponding clusters.



In the second round, we perform a number of rakes and one compress and update the corresponding clusters.



In the third round, we again perform a number of rakes and one compress and update the corresponding clusters.



It is important the expressions on the binary clusters have the form $ax+b$ where a and b are constants. As a result, as tree contraction progresses, expressions do not grow into larger expressions, rather they can be simplified into a simple form. This is crucial is making sure that rake and compress operations require constant work.

Another broad class of tree computations include ***treefix computations***, which generalize the "prefix sum" or the "scan" example for sequences to rooted trees by separately considering the two possible directions:

1. from root to leaf, which are called rootfix computations, and
2. from leaf to root, which are called leaffix computations.

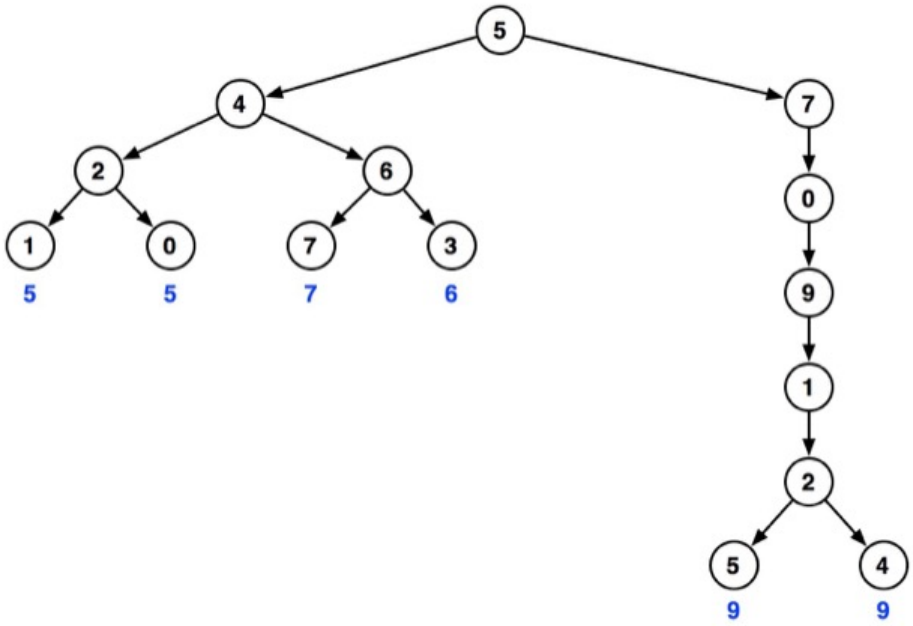
For the rest of this section, we shall assume binary trees.

15.3.1 Rootfix computations

Let $\oplus$ be an associative binary operation. A ***rootfix computation*** on a binary tree T computes the $\oplus$-sum for all the (simple) paths from the root of T to all the leaves of T.

Example 15.11 Maximum on root-to-leaf path

Suppose that we are given a binary tree where each node is labeled with an integer number. We wish to compute for each leaf in the tree the maximum node from the leaf to the root. An example tree and the result to be computed are shown below. This computation is a rootfix computation with "max" operation on integers.



We can use tree contraction to perform a rootfix computation. To this end, we define each binary cluster to represent the maximum node on the path between the boundary nodes of the cluster. Initially each edge is a binary cluster with an empty path and thus has the initial value of $-\infty$. We don't define any specific value for unary cluster. We then define the rake and compress operations as follows.

1. Compress operation: take the maximum of the deleted clusters and the deleted node and assign that value to the new binary cluster created.
2. Rake operation: no specific computation on unary clusters is needed.

At the expansion steps, we compute the correct values for raked and compressed operations by considering parent of the deleted vertex and the weight of the incoming edge and taking the maximum.

Exercise

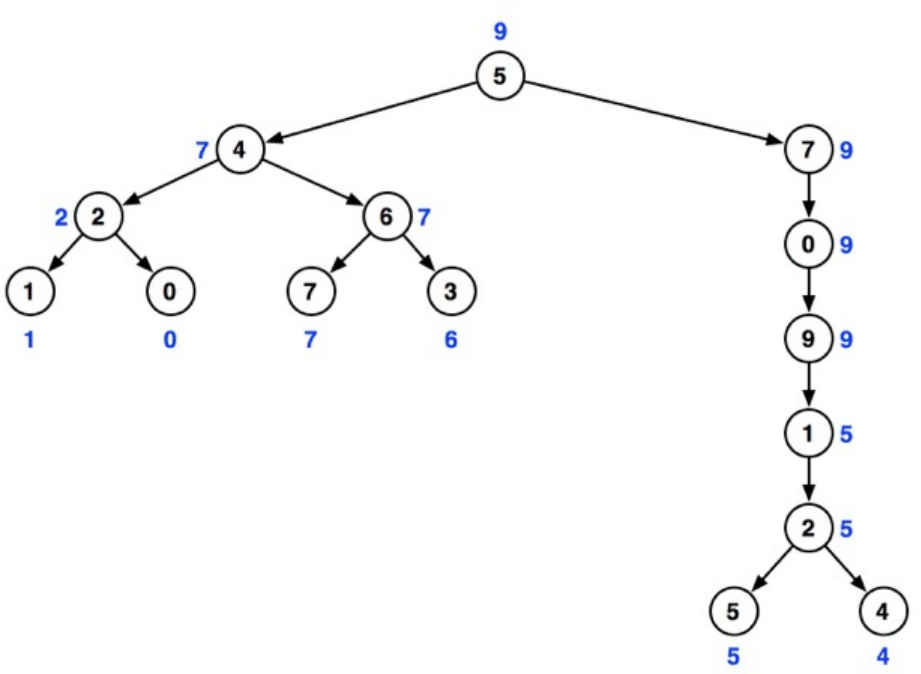
Complete the example given above to compute the maximums for the root to leaf paths using tree contraction.

15.3.2 Leaffix computations

Let $\oplus$ be an associative binary operation. A ***leaf computation*** on a binary tree T computes for each node in T, the $\oplus$-sum of the in-order traversal of the subtree rooted at that node.

Example 15.12 Maximum in a subtree

Suppose that we are given a binary tree where each node is labeled with an integer number. We wish to compute for each subtree rooted at a node in the tree the maximum node in that subtree. An example tree and the result to be computed are shown below. This computation is a rootfix computation with "max" operation on integers.



We can use tree contraction to perform a leaffix computation.

Exercise

Specify the unary and binary clusters for performing the leaffix-max computation. Describe how rake and compress operations should behave and how expansion works.

Exercise

Specify the necessary operations to compute the in-order rank of the nodes in a tree.

1. What information does a unary cluster contain?
2. What information does binary cluster contain?
3. What computation should rake and compress perform?
4. How should expansion work?

16 Models of Parallel Computation

> Recent advances in microelectronics have brought closer to feasibility the construction of computers containing thousands (or more) processing elements.
>
> — James Christopher Wyllie *Ph.D. Thesis Cornell University 1979*

Parallel computing has fascinated computer scientists from the early days of computing, even before parallel computers have become easily available starting in 2000's. The architecture of early parallel computers varied greatly.

1. Cray-1 was the first vectorized parallel machine that can perform operations on sequences of data called vectors.
2. ILLIAC IV had 64 processors laid out on a rectangular grid. Each processor had its own memory but could communicate with its four neighbors on the grid and thus request data from them. ILLIAC was a synchronous machine where each processor would execute the same instruction in each step, operating on its own memory.

3. CM (Connection Machine) could have many (tens of thousands) processors arranged into clusters, which are in turn arranged into superclusters, and communication taking place through buses connecting processors with each level of the cluster hierarchy. Each processor had its own memory and access the memory of others via the communication bus. The machine operated asynchronously allowing each processor to perform instructions independently of the others.

This diversity continues to exist today. For example, graphics processors (GPUs), multicore computers, large data centers consisting of many clusters of computers have characteristics of these earlier designs.

It is thus natural to consider the question of how one might design algorithms for these machines. This question may be viewed as especially relevant because serial algorithms are traditionally designed for the RAM (Random Access Memory) machine of computation, which is equivalent to a Turing Machine and thus to Lambda Calculus. In 1979, James C. Wyllie proposed the PRAM model as a RAM-like model for parallel computing. Wyllie viewed asynchronous computation as inappropriate for the purposes of worst-case complexity analysis and thus proposed a synchronous model of computation that combines the synchronous computation model of ILLIAC-IV with the hierarchical memory model of the Connection Machine. As mentioned by Wyllie, the PRAM model was used by many authors before it was proposed by Wyllie, probably because it is a relatively natural generalization of the sequential RAM model.

16.1 PRAM Model

16.1.1 The Machine

A PRAM consists of

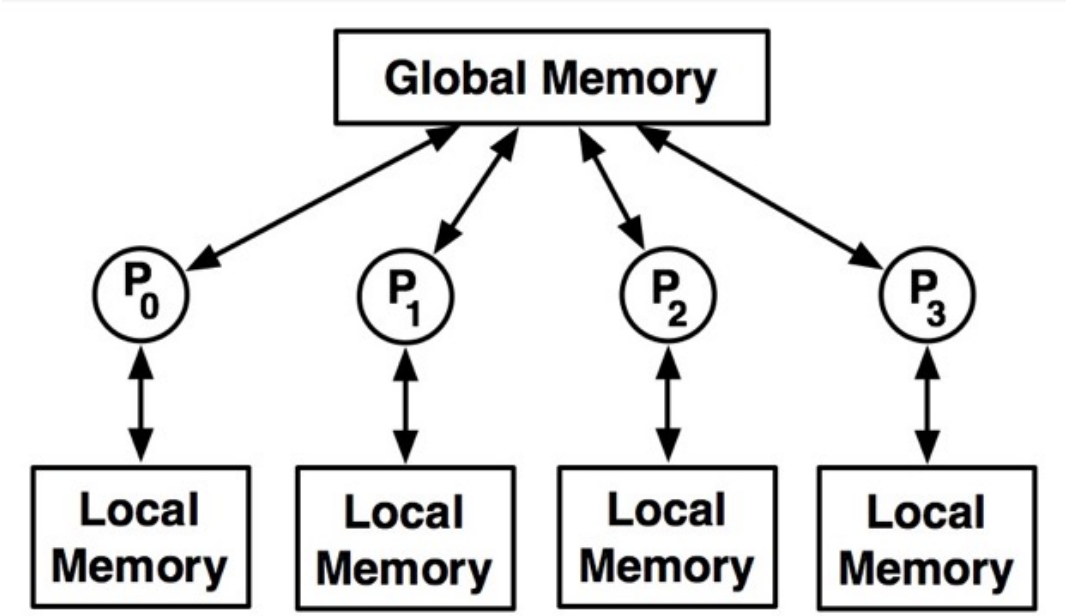
1. an unbounded number of processors, $P_0, \ldots,$
2. an unbounded global memory,
3. a finite program.

Each processor in turn has

1. a unique processor id,
2. a program counter,
3. an unbounded local memory,
4. a flag indicating whether or not a processor is running or active.

Example 16.1 A 4-Processor PRAM

The drawing below illustrates a 4-processor PRAM.



The PRAM model abstracts over various connection patterns used by parallel architectures by supplying a global memory that is accessible to all processors.

A PRAM program is a synchronous program that specifies the computation performed by each processor at each step. Execution of a PRAM program proceeds in step. In each step all active processors execute the instruction pointed by their program counter. The instruction may use the id of the processor, which can be thought as being stored in the local memory. For example, each processor i can read the i^{th} cell of an array stored in global memory. Each processor can access its own local memory or the global memory but not the local memory of another processor. A processor may choose not to participate in a step; such a processor would be inactive on that step. An active processor may activate an inactive processor and direct it to a certain instruction by setting its program counter.

In his formulation of the PRAM model Wyllie did not permit multiple processors to write into the same (global) memory cell. Many different variations of this model, however, been later proposed that allow different degrees of "concurrency." Some notable variants include the following.

1. ***EREW (Exclusive-Read-Exclusive-Write) PRAM:*** concurrent reads from or writes into the same global memory cell are disallowed.
2. ***CREW (Concurrent-Read-Exclusive-Write) PRAM:*** concurrent reads from global memory cells are permitted but concurrent writes into the same global memory cell are disallowed.
3. ***CRCW (Concurrent-Read-Concurrent-Write) PRAM:*** concurrent reads from and concurrent writes into the same global memory cells are permitted. It is possible to distinguish further between different CRCW PRAMs.
    a. ***Common CRCW:*** concurrent writes must all write the same value.
    b. ***Arbitrary CRCW:*** concurrent writes can write different values in a step, but only one arbitrary write succeeds.
    c. **Priority CRCW:* concurrent rites can write different values in a step, but only the processor with the highest priority defined as the processor with the minimum id succeeds.

In terms of computational power, these different models turn out to be similar.

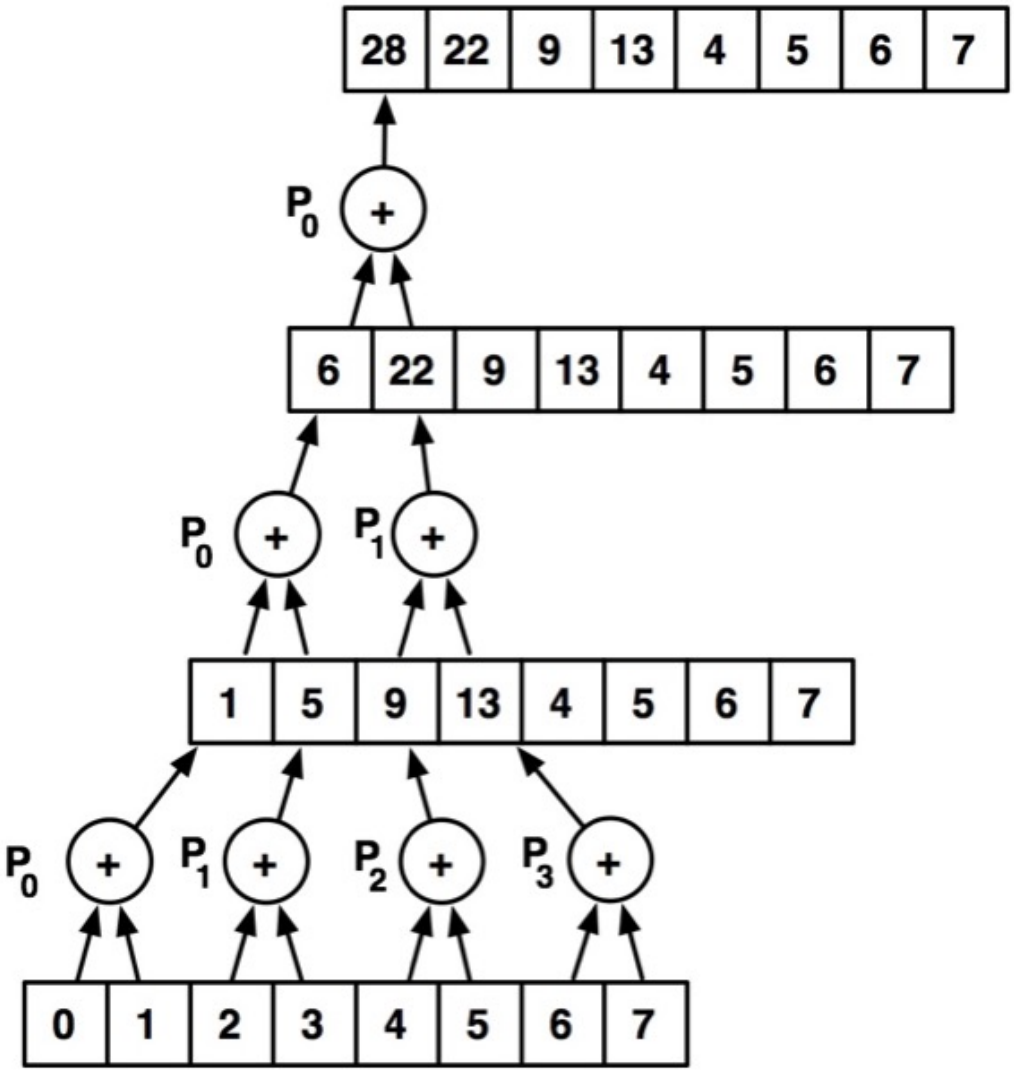
16.1.2 An Example: Array Sum

Suppose that we are given an array of elements stored in global memory and wish to compute the sum of the elements. We can write a PRAM program to find the sum. Suppose that the array contains n elements and we wish to use n processors. Let's assume for simplicity that n is a power of 2.

We can proceed in rounds. In the first round, each processor P_i, where $0 \leq i \leq n/2$, adds up the pair of elements at position $2i$ and $2i+1$ and places it in position i. In the second round, we repeat the same computation but consider only the first $n/2$ of the elements, and so on. After $\lg n$ rounds, the sum is placed in the first position of the input array. The drawing below illustrates this algorithm for an input of size 8.

Example 16.2 Array sum in PRAM

The drawing below illustrates the PRAM array sum algorithm for an array with 8 elements.



We can write the code for this algorithm as follows. In the code the local variable `i` denotes the id of the processor. The computation starts by each processor executing this piece of code.

PRAM code for array sum

```
array_sum (A, n) =
  for j = 1 to lg n  {
    active_procs  = n/2^j
    if (i < active_procs) {
      x = global_read A[2i]
      y = global_read A[2i+1]
      z = x + y
      global_write z into A[i]
    }
```

Note the role that the synchronous execution of PRAM programs plays in the example. If the processors did not execute each step synchronously at the same time, then the execution can mix up results from different rounds and obtain an incorrect result.

In this algorithm, no (global) memory cell is read by more than one processor at the same step. Similarly, no (global) memory cell is written my more than one processor at the same step. This algorithm is thus a EREW PRAM algorithm.

16.1.3 PRAM in Practice

Several assumptions of the PRAM model make it unlikely that the human kind will ever be able to build an actual PRAM.

1. Constant memory access: in PRAM, all processors can access memory in constant time independent of the number of processors. This is currently impossible because an arbitrary number of processors and memory cannot be packed into the same 3-dimensional space. Assuming that memory access speed is bounded by the speed of light, there will thus be a dependence between the number of processors and the time for memory access.
2. Concurrent memory reads and writes: all known memory hardware can serve a constant number of reads and writes in the same time step but in a PRAM with p processors, there can be p concurrent reads from and writes into the same memory location.

Another problem with the PRAM model is that the PRAM algorithms do not translate to practice well.

1. The synchrony assumption, that all processors execute program instructions in lock step, is nearly impossible to guarantee. In practice, parallel programs are executed by a system that maps many processes on the same processors, swapping processes in and out as needed. Furthermore, the programmers write their programs using higher level languages which then translate to many individual machine instructions. Even if these instruction might be executed on the hardware in lock step, the programmer does not have control over how their high-level parallel programs are translated into low-level instructions.
2. PRAM programs specify instructions executed by each processor at each time step. In other words, they must specify the algorithm and the schedule for the algorithm. This is very tedious and extremely difficult to do in practice because for example, we may not know the number of processors available, or worse the number of processors may change during execution.

For these reasons, the value of a PRAM algorithm from a practical perspective is limited to the ideas of the algorithm. Such ideas can still be valuable but new ideas may be needed to implement a PRAM algorithm in practice.

16.2 Work-Time Framework

Since a PRAM program must specify the action that each processor must take at each step, it can be very tedious to use. Parallel algorithms therefore are usually expressed in the ***work-time (WT) framework***. In this approach, we describe the parallel algorithm in terms of a number of steps, where each step may contain any number of operations that can all be executed in parallel. To express such a step, all we need is a "parallel for" construct, written `pfor`. Once the algorithm is described at this high level, we then use a general scheduling principle to map the WT algorithm to a PRAM algorithm. The basic idea is to distribute the total work in each step among the available processors as evenly as possible. Since this second transformation step is usually routine, and is often omitted.

Example 16.3 Array sum in work-time framework

The array-sum algorithm can be written as follows.

```
array_sum (A, n) =
  for j = 1 to lg n  {
    pfor i = 0 to n/(2^j) {
      A[i] = A[2i] +  A[2i+1]
    }
  }
```

Having specified the algorithm, we can then derive a PRAM algorithm by mapping each parallel step to a sequence of PRAM steps by distributing the work among the available processors. For example, if we wish to use n processors, then we can schedule `pfor` by assigning each iteration to the first $\frac{n}{2^j}$ processor. Doing so yields the PRAM algorithm described earlier in the section.

16.2.1 Work-Time Framework versus Work-Span Model

In this course, we primarily used the work-span model instead of the work-time framework. The two are similar but the work-span model is somewhat more expressive: we don't have to use just parallel for's. We can use any number of available parallelism constructs such as fork-join, async-finish, and futures. In other words, the work-span model is more amenable as a language-level cost model. In the work-span model we also do not care to specify the PRAM algorithm at all, because we expect that the algorithm will be executed with a scheduler that is able to keep the processors busy without us specifying how to do so. As we have seen, this is a realistic assumption, because there are scheduling algorithms such as work stealing that can schedule computations efficiently.

17 Chapter: Graphs

In just the past few years, a great deal of interest has grown for frameworks that can process very large graphs. Interest comes from a diverse collection of fields. To name a few: physicists use graph frameworks to simulate emergent properties from large networks of particles; companies such as Google mine the web for the purpose of web search; social scientists test theories regarding the origins of social trends.

In response, many graph-processing frameworks have been implemented both in academia and in the industry. 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, making it practically infeasible to perform sequentially interesting computations.

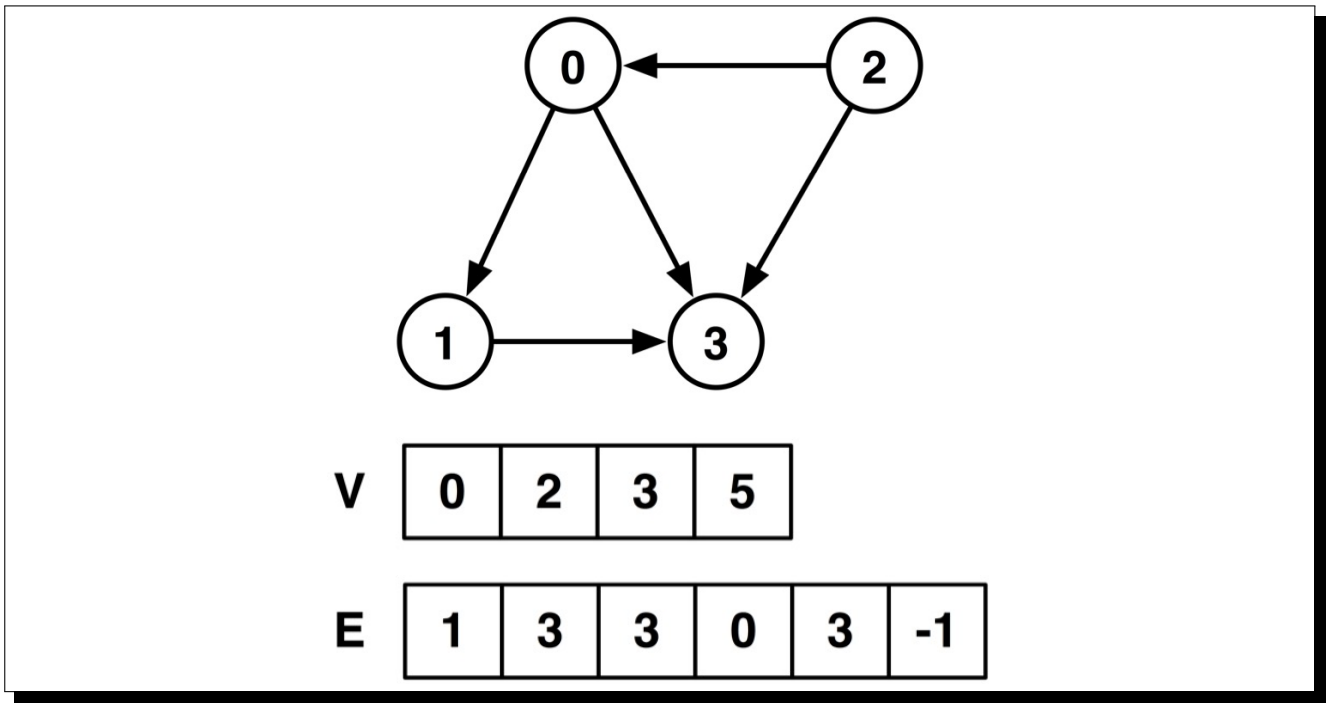
17.1 Graph representation

We will use an adjacency lists representation based on ***compressed arrays*** to represent directed graphs. In this representation, a graph is stored as a compact array containing the neighbors of each vertex. Each vertex in the graph $G = (V, E)$ is assigned an integer identifier $v \in \{0, \ldots, n-1\}$, where $n = |V|$ is the number of vertices in the graph. The representation then consists of two array.

1. The ***edge array*** contains the adjacency lists of all vertices ordered by the vertex ids.
2. The ***vertex array*** stores an index for each vertex that indicates the starting position of the adjacency list for that vertex in the edge array. This array implements the

A graph (top) and its compressed-array representation (bottom)

consisting of the vertex and the edge arrays. The sentinel value "-1" is used to indicate a non-vertex id.



The compressed-array representation requires a total of $n+m$ vertex-id cells in memory, where $n = |V|$ and $m = |E|$. Furthermore, it involves very little indirection, making it possible to perform many interesting graph operations efficiently. For example, we can determine the out-neighbors af a given vertex with constant work. Similarly, we can determine the out-degree of a given vertex with constant work.

Exercise

Give a constant-work algorithm for computing the out-degree of a vertex.

Space use is a major concern because graphs 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. For simplicity, we always use 64 bits to represent vertex identifiers; for small graphs 32-bit representation can work just as well.

We implemented the adjacency-list representation based on compressed arrays with a class called `adjlist`.

```
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;
};
```

Example 17.1 Graph creation

Sometimes it is useful for testing and debugging purposes to create a graph from a handwritten example. For this purpose, we define a type to express an edge. The type is a pair type where the first component of the pair represents the source and the second the destination vertex, respectively.

```
using edge_type = std::pair<vtxid_type, vtxid_type>;
```

In order to create an edge, we use the following function, which takes a source and a destination vertex and returns the corresponding edge.

```
edge_type mk_edge(vtxid_type source, vtxid_type dest) {
  return std::make_pair(source, dest);
}
```

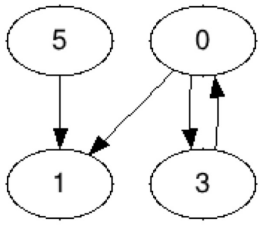
Now, specifying a (small) graph in textual format is as easy as specifying an edge list. Moreover, getting a textual representation of the graph is as easy as printing the graph by `cout`.

```
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 17.2 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.

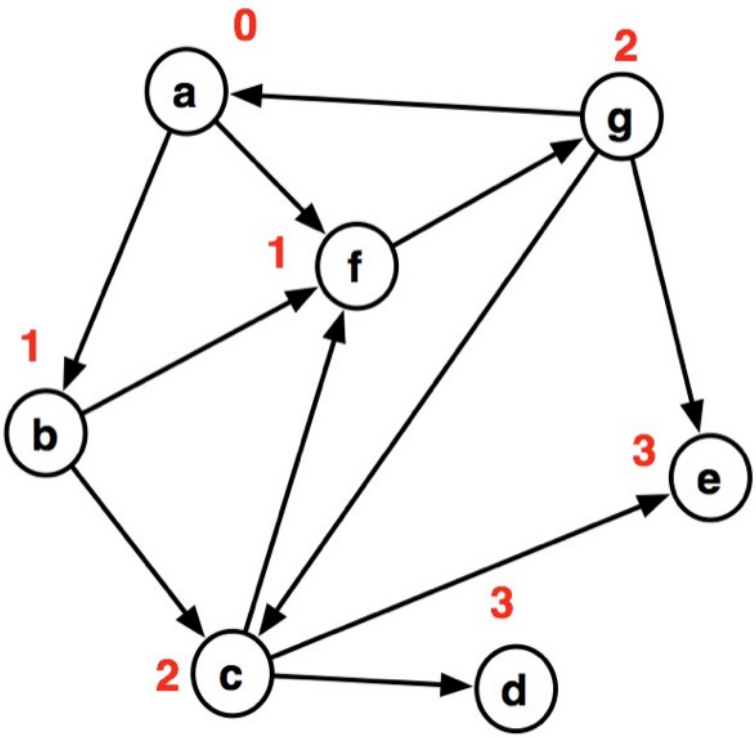
17.2 Breadth-first search

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 17.3 BFS Levels

A graph, where each vertex is labeled with its level.



At a high level, BFS algorithm maintains a set of vertices called ***visited***, which contain the vertices that have been visited, and a set of vertices called ***frontier***, which contain the vertices that are not visited but that are adjacent to a visited vertex. It then visits a vertex in the frontier and adds its out-neighbors to the frontier.

17.2.1 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 represent the `frontier`. The `visited` set can be represented as some array data structure, or can be represented implicitly by keeping a flat at each vertex that indicating whether the vertex is visited or not.

Pseudocode for serial BFS

```
sequential_bfs (G = (V,E), s) =

  frontier = <s>
  visited = {}
```

```
  while frontier is not <> do
    let <v,frontier_n> be frontier
    if v is not in visited then
      visit v
      foreach out-neighbor u of v do
        frontier_n = append frontier_n <u>
      frontier = frontier_n

  return visited
```

17.2.2 Parallel BFS

Our goal is to design and implement a parallel algorithm for BFS that is observably work efficient and has plenty of parallelism. There is natural parallelism in BFS because the vertices in each level can actually be visited in parallel, as shown in the pseudo-code below.

Pseudo-code for parallel BFS

```
parallel_bfs (G=(V,E), source) =

  frontier = {source}
  visited = {}
  level = 0
  while frontier is not {} do
    next = {}
    let {v_1, ..., v_m} be frontier
    parallel for i = 1 to m do
      visit v_i
    visited = visited set-union frontier

    next = out_neighbors(v_1) set-union ... set-union  out_neighbors(v_m)
    frontier = next set-difference visited
    level = level + 1

  return visited
```

Exercise

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

Note that we can also compute the next set (frontier) in parallel by performing a reduce with the set-union operation, and then by taking a set-difference operation.

Our goal is to implement an observably work-efficient version of this algorithm on a hardware-shared memory parallel machine such as a modern multicore computer. The key challenge is implementing the set operations on the `visited`, `frontier`, and `next` sets. Apart from maintaining a visited map to prevent a vertex from being visited more than once, the serial algorithm does not have to perform these operations.

To achieve work efficiently, we will use atomic read-modify-write operations, specifically compare-and-swap, to mark visited vertices and use an array representation for the frontier. To achieve observable work efficiency, 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. This guard is necessary, because two vertices in the frontier can both have the same vertex v as their neighbor, causing v to be visited multiple times. After we visit 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

```
parallel_bfs (G=(V,E), source)

  level = 0
  parallel for i = 1 to n do
    visited[i] = false

  visit source
  visited[source] = true
  frontier = {source}

  while frontier is not {} do
     level = level + 1
     let {v_1, ..., v_m} be frontier
     parallel for i = 1 to m do
       let {u_1, ..., u_l} be out-neighbors(v_i)
       parallel for j = 1 to l do
         if compare_and_swap (&visited[u_j], false, true) succeeds then
           visit u_j
       frontier = vertices visited at this level

  return visited
```

As show in the pseudocode, there are at least two clear opportunities for parallelism. The first is the parallel loop that processes the frontier and the second the parallel for loop that processes the neighbors of a vertex. These two loops should expose a lot of parallelism, at least for certain classes of graphs. The outer loop exposes parallelism when the frontier gets to be large. The inner loop exposes parallelism when the traversal reaches a vertex that has a high out degree.

To keep track of the visited vertices, the pseudo-code use an array of booleans `visited[v]` of size n that is keyed by the vertex identifier. If `visited[v] ==true`, then vertex `v` has been visited already and has not otherwise. We used an atomic compare-and-swap operation to update the visited array because otherwise vertices can be visited multiple times. To see this suppose now that two processors, namely A and B, concurrently attempt to visit the same vertex `v` (via two different neighbors of `v`) but without the use of atomic operations. If A and B both read `visited[v]` at the same time, then both consider that they can visit `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.

Exercise

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?

In the rest of this section, we describe more precisely how to implement the parallel BFS algorithm in C++.

In C++, we can we can use lightweight atomic memory, as described in this chapter to eliminate race conditions. The basic idea is to guard each cell in our "visited" array by an atomic type.

Example 17.4 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. 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
```

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. Let's now put these ideas together to complete and implementation. 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.

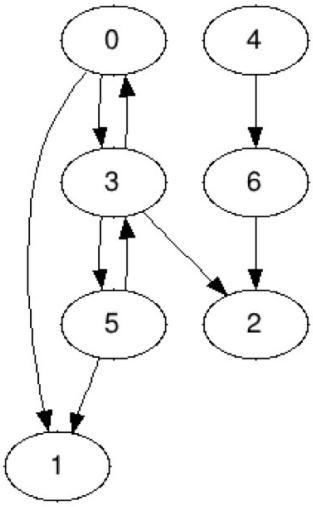
```
sparray bfs(const adjlist& graph, vtxid_type source);
```

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 17.5 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 }
```

The next challenge is the implementation of the frontiers. To obtain an observably work efficient algorithm, we shall represent frontiers as arrays. Let's assume that we have a function called `edge_map` with the following signature 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);
```

Using this function, we can implement the main loop of BFS as shown below. The algorithm uses the `edge-map` 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. Here is an example describing the behavior of the `edge_map` function.

Example 17.6 A run of `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, out_frontier);
std::cout << out_frontier2 << std::endl;
```

Output:

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

There are several ways to implement `edge_map`. One way is to allocate an array that is large enough to hold the next frontier and then allow the next frontier to be computed in parallel. Since we don't know the exact size of the next frontier ahead of time, we will bound it by using the total number of out-going edges originating at the vertices in the frontier. To mark unused vertices in this array, we can use a sentinel value.

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

Example 17.7 Array representation of the next frontier

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 an 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);
}
```

We can implement edge-map as follows. We first allocate the array for the next frontier by upper bounding its size; each element of the array is initially set to `not_a_vertexid`. We then, in parallel, visit each vertex in the frontier and attempt, for each neighbor, to insert the neighbor into the next frontier by using an atomic compare-and-swap operations. If we succeed, then we write the vertex into the next frontier. If not, we skip. Once all neigbors are visited, we pack the next frontier by removing non-vertices. This packed array becomes our next frontier.

```
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) {
```

```
  // temporarily removed.
}
```

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?

17.2.3 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.

Exercise

By using sentinel values, it might be possible to implement BFS to eliminate the compaction used by `edge_map`. Describe and implement such an algorithm. Does in perform better?

17.2.4 Direction Optimization

If the graph has reverse edges for each vertex, we can use an alternative "bottom-up" implementation for `edge_map` instead of the "top-down" approach described above: instead of scanning the vertices in the frontier and visiting their neighbors, we can check, for any unvisited vertex, whether that vertex has a parent that is in the current frontier. If so, then we add the vertex to the next frontier. If the frontier is large, this approach can reduce the total number of edges visited because the vertices in the next frontier will quickly find a parent. Furthermore, we don't need use compare-and-swap operations. The disadvantage of the bottom-up approach is that it requires linear work in the number of vertices. However, if the frontier is already large, the top-down approach requires lineal work too. Thus if, the frontier is large, e.g., a constant fraction of the vertices, then the bottom-up approach can be beneficial. On current multicore machines, it turns out that if the frontier is larger than 5% of the total number of vertices, then the bottom-up approach tends to outperform the top-down approach.

An optimized "hybrid" implementation of PBFS can select between the top-down and the bottom-up approaches proposed based on the size of the current frontier. This is called "direction-optimizing" BFS.

Exercise:

Work out the details of the bottom-up algorithm for parallel BFS and the hybrid algorithm. Can you implement and observe the expected improvements?

17.3 Depth First Search

In Depth First Seach (DFS), we visit vertices of a graph in "depth first" order by visiting the most recently seen vertex. We can write the pseudocode for DFS as follows.

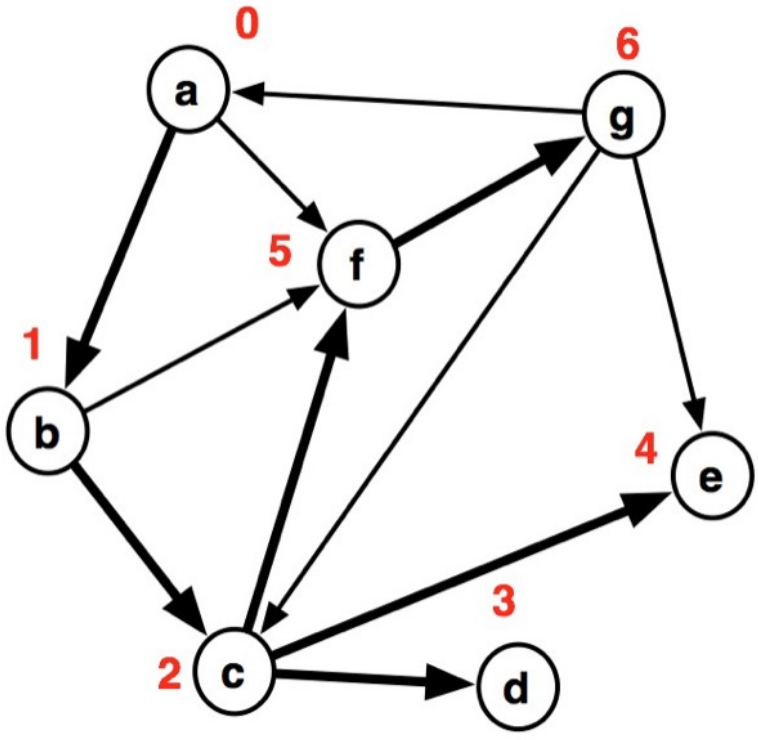
Pseudocode for DFS

```
dfs_rec (G, visited, v) =
  if v in visited
    return visited
  else
    visit v
    visited = visited set-union {v}
    let <v_1, ..., v_m> be out_neighbors(G,v)
    for i = 1 to m do
      visited = dfs_rec (G, visited, v_i)
    return visited

dfs (G, source) =
  visited = dfs_rec (G, {}, source)
  return visited
```

Example 17.8 DFS

DFS on a graph illustrated. Each vertex is labeled with the order that it was visited starting the source vertex **"a"**. We assume that the out-edges of each vertex are ordered counterclockwise starting from left. The edges that lead to a successful visit are highlighted.



An important property of the DFS algorithm is that it recursively explores the out-neighbors of a vertex in order. This "lexicographical" ordering bestows DFS with special powers. It can be used to solve many interesting graph problems, exactly because it visits vertices in a specific order. The DFS problem, however, is a P-complete problem. This suggests that DFS is difficult to parallelize. There are parallel algorithm for special graphs, such as planar graphs, but work-efficient general-purpose algorithms remain unknown.

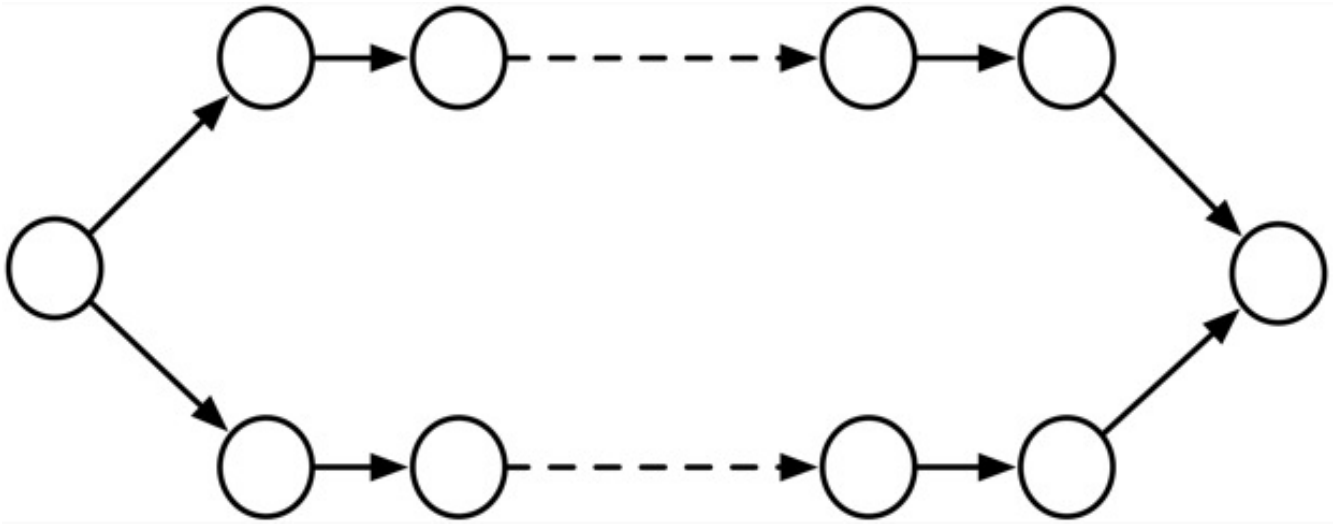
The lexicographical ordering followed by DFS is not useful in some applications. For example the reachability problem, which requires finding the vertices that are reachable from a given vertex, does not require the vertices to be visited in a particular order. An interesting question therefore is whether an ***unordered DFS*** or ***pseudo DFS***, can be performed in parallel.

Before we answer this question, let's convince ourselves that it is worth our while. Considering that we already have a parallel graph traversal algorithm, parallel BFS, why is there a need? The main reason is that parallel BFS requires global synchronization on every level of the graph. This can lead to unnecessary synchronization, which can reduce parallelism.

We define a ***parallel pseudo DFS*** or ***PDFS*** for short as a parallel algorithm that explores in parallel the reachable vertices from the current set of vertices in depth-first order but without observing any ordering constraints.

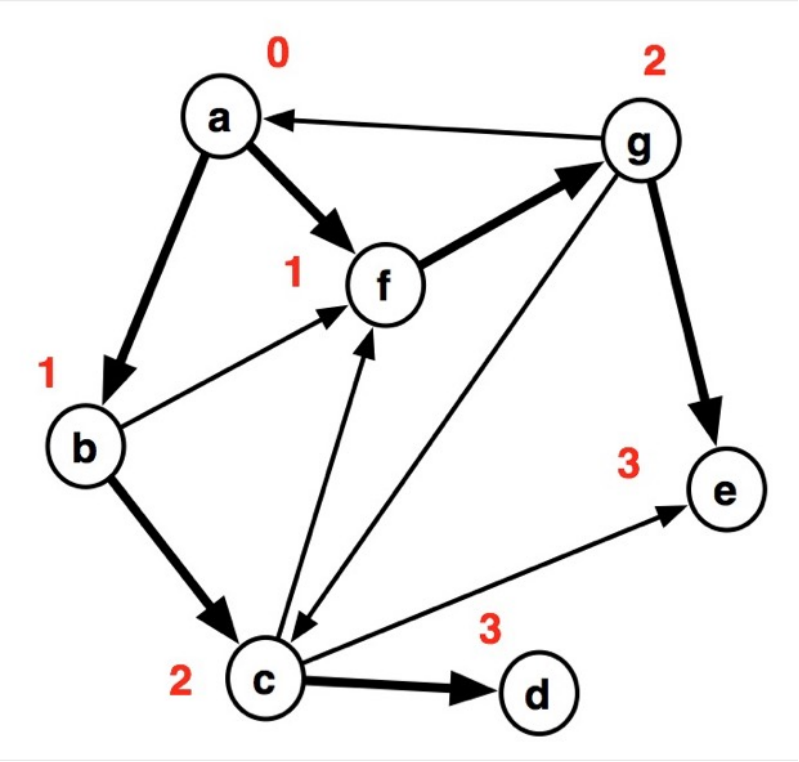
Example 17.9 BFS versus PDFS in a deep graph

Consider the following graph that consists of two deep parallel branches, the dotted lines indicate a long chain of vertices. A BFS on this graph visits each level in order, synchronizing after each. Since there are only two vertices in each level, there is no practical benefit to parallelizing the visits inside a level. Thus, for all practical purposes, there is no parallelism in this graph when using BFS. In fact, even a graph that has hundreds of such parallel chains may offer too little parallelism. Using PDFS, however, the parallel chains can all be traversed in parallel effectively, because there is no need for synchronization.



Example 17.10 PDFS

A graph, where each vertex is labeled with the time at which it is visited assuming that no more that two edges are visited at a time.



Note that there are many different ways in which the vertices could have been visited.

We can write the pseudocode for PDFS as follows. In order to search the graph in parallel, we rely on a global `visited` array that keeps a flag for each vertex visited.

Pseudocode for PDFS

```
pdfs_rec (G, visited, frontier) =
  if frontier = {} then
    return ()
  else if {v} = frontier then
    if compare_and_swap (&visited[v], false, true) succeeds then
        visit v
        N = out_neighbors (G, v) in
        pdfs_rec (G, visited, N)
  else
    let {v_1, ..., v_m} be frontier
    mm = m/2
    frontier_1 = {v_1, ..., v_mm}
    frontier_2 = frontier set-minus frontier_1

    async pdfs_rec (G, visited, frontier_1)
    async pdfs_rec (G, visited, frontier_2)

pdfs (G, source) =
  let frontier = {source}

  parallel for i = 1 to n do
      visited[i] = false

  finish  {
    pdfs_rec (G, visited, frontier)
  }

 return visited
```

The function `pdfs` takes as argument the graph and the source vertex and calls. It starts by setting up a `visited` array where all vertices are marked as unvisited and then calls, nested inside a `finish` block, the function `pdfs_rec` with a frontier consisting only of the source vertex and the graph. The `finish` block waits for all the parallel computations spawned off by the function `pdfs_rec` to complete. The function `pdfs_rec` takes as argument the input graph, aset `visited` of vertices that are already visited, and a set `frontier` of vertices that are to be visited next. Based on the size of the frontier, the algorithm performs the following actions.

1. If the frontier is empty, then it returns.
2. If the frontier contains only one vertex, then the algorithm uses a compare-and-swap to make sure that the vertex is visited exactly once. If the compare-and-swap succeeds then the algorithm visits the vertex and performs a PDFS on the out-neighbors of the vertex. Note that at this point, there are no other unvisited vertices in the frontier, and thus the out-neighbors of the vertex can be used as the new frontier.
3. If the frontier contains more than one vertex, then the algorithm splits the frontier into two frontiers `frontier_1` and `frontier_2` and performs in parallel a PDFS on each frontier. To perform the two searches in parallel, the algorithm uses the `async` function, which spawns off a parallel computation to perform the given argument. In order to achieve a low span, it is important that the `split` operations splits the frontier evenly.

Note that all the parallel computations created by a run of `pdfs_rec` synchronize exactly at one point, which is the `finish` block at the end of the computation.

17.3.1 Work Efficient PDFS

The algorithm presented above is neither asymptotically and nor observably work efficient, because of two issues.

1. The algorithm relies on a frontier data structure that the serial DFS algorithm does not use. In fact, the serial DFS algorithm uses no auxiliary data structures, except perhaps a simple stack.
2. The algorithm creates tiny threads by recursing down to single element frontiers.

To solve these problems, we shall design a ***frontier data structure*** specifically geared towards supporting the efficient operations needed by parallel BFS. This data structure should support various operations such as insertion and deletion of vertices, splitting of the frontier into two even halves, and unioning of two frontiers. To solve the second problem, we might be tempted to change change the base case of the algorithm so that it considers larger frontiers for sequential processing. The pseudo-code for such an algorithm is shown below. The algorithm stops when it encounters a small frontier consisting of `K` or fewer vertices and visits `K` vertices until it generates parallelism. In the presentation for the algorithm, we treat the frontier data structure, `frontier`, as imperative data structure, which gets updated by the operations performed on it.

Pseudocode for granularity-controlled PDFS, Attempt 1

```
pdfs_rec (G, visited, frontier) =
  if frontier.size = 0 then
    return ()
  else if frontier.size <= K then
    repeat
      v =  frontier.remove ()
      if compare_and_swap (&visited[v], false, true) succeeds then
        (* v is not visited. *)
        visit v
        f = mkFrontierFromSequnce (out_neighbors (G,v))
        frontier.union f
    until
      frontier.size = 0

      (* PDFS recursively on the remaining frontier *)
      pdfs_rec (G, visited, frontier)
  else
    (frontier_1, frontier_2) = frontier.split
    async pdfs_rec (G, visited, frontier_1)
    async pdfs_rec (G, visited, frontier_2)
```

This algorithm, however, is not quite what we want, because it can explore a graph with a small frontier entirely serially. In other words, the algorithm above serializes too aggressively. What we would like to do instead is to generate parallelism but amortize the cost of doing so by performing a commensurate amount of serial work. The pseudo-code for an algorithm that follows this technique is shown below. Note that the algorithm performs, at each recursive call, a fair amount of serial work by visiting K vertices serially. But after it does so, the algorithm splits the frontier and explores the two frontiers in parallel. Note that the algorithm avoids splitting a singleton frontiers.

Pseudocode for granularity-controlled PDFS

```
pdfs_rec (G, visited, frontier) =
  repeat
    if frontier.size = 0 then
      return ()

    v = frontier.remove ()
    if compare_and_swap (&visited[v], false, true) succeeds then
      (* v is not visited. *)
      visit v
      f = mkFrontierFromSequence (out_neighbors (G,v))
      frontier.union f
  until
    at least K vertices are visited
    and
    frontier.size > 1

  (frontier_1, frontier_2) = frontier.split () in
  async pdfs_rec (G, visited, frontier_1)
  async pdfs_rec (G, visited, frontier_2)
```

Since the serial DFS algorithm does not use any data structures to represent the frontier, making PDFS work efficient requires designing an efficient, ideally constant-time, frontier data structure. This data structure needs to support (at least) the following operations.

1. `mkEmpty`: return an empty frontier.
2. `mkFrontierFromSequence`: constructs a frontier from a sequence.
3. `insert`: insert a vertex into given frontier.
4. `remove`: remove a vertex from a given frontier.
5. `split`: split a given frontier into two frontiers, preferably evenly.
6. `union`: union two frontiers.

For our purposes, an imperative data structure, where the operations `insert`, `remove`, `split`, and `union` destructively update the frontier suffices. Such imperative data structures may consume their argument frontier in order to produce their output. Note also that the operation `mkFrontierFromSequence` can be implemented by starting with an empty frontier and inserting the elements of the sequence one by one into it. A more direct implementation can be more efficient, however.

Since the frontier data structure does not have to enforce any ordering on the vertices and since a vertex can be inserted into the frontier many times (once for each incoming edge), we can think of the frontier data structure as implementing a bag, which is a set that allows multiplicity. In fact, the interface presented above is a fairly general interface for bags, and a data structure implementing these operations efficiently can be very useful for many parallel algorithms, not just PDFS.

In what follows, we first consider a data structure that supports all bag operations in logarithmic work and span. We then introduce a "chunking" mechanism for improving the constant factors, which is important for observable work efficiency. It is also possible to refine the data structure to reduce the work of `insert` and `delete` operations to amortized constant but we shall not discuss this here.

17.3.2 Bags

The basic idea behind our bag data structure is to represent the contents as a list of complete trees that mimic the binary representation of the number elements in the bag. Recall that a complete tree is a tree where all internal nodes have exactly two children and all the leaves are at the same level. For example, we shall represent a tree with 3 elements, with two complete trees of size 1 and 2, because the binary representation of 3 is 11; we shall represent a tree with 13 elements with three complete trees of size 8, 4, and 1, because the binary representation of 13 is 1011. The number 0 will be represented with an empty list.

The complete SML code for this bag data structure is shown below. The operations insert, remove, union, and split follow the arithmetic operations increment, decrement, add, and divide on binary numbers.

The SML code for bags.

```
structure Bag =
struct

  (* A tree is either a leaf holding a value
     or it is a node with a size/weight field and two subtrees.
     The size/weight field is not necessary. *)

  datatype 'a tree =
    Leaf of 'a
  | Node of int * 'a tree * 'a tree

  datatype 'a digit =
    Zero
  | One of 'a tree

  type 'a bag = 'a digit list

  exception EmptyBag
  exception SingletonTree

  (* empty bag *)
  fun mkEmpty () =
    nil

  (* size of a tree, constant work *)
  fun sizeTree t =
    case t of
      Leaf x => 1
    | Node (w, l, r) =>  w

  (** Utilities **)

  (* link two trees, constant work *)
  fun link (l, r) =
    Node (sizeTree l + sizeTree r, l, r)

  (* unlink two trees, constant work *)
  fun unlink t =
    case t of
      Leaf _ => raise SingletonTree
    | Node (_, l, r) => (l,r)

  (* insert a tree into a bag. *)
  fun insertTree (t, b) =
    case b of
      nil => [One t]
    | Zero::b' => (One t)::b'
    | One t'::b' =>
    let
      val tt' = link (t,t')
```

```
    val b'' = insertTree (tt', b')
  in
    Zero::b''
  end

(* borrow a tree from a bag. *)
fun borrowTree b =
  case b of
    nil => raise EmptyBag
  | (One t)::nil => (t, nil)
  | (One t)::b' => (t, Zero::b')
  | Zero::b' =>
    let
      val (t', b'') = borrowTree b'
      val Node(_, l, r) = t'
    in
      (l, (One r)::b'')
    end

(** Mainline **)

(* insert element into a bag *)
fun insert (x, b) =
  insertTree (Leaf x, b)

(* remove an element from a bag *)
fun remove b =
  let
    val (Leaf x, b') = borrowTree b
  in
    (x, b')
  end

(* union two bags. *)
fun union (b, c) =
  case (b,c) of
    (_, nil) => b
  | (nil, _) => c
  | (d::b', Zero::c') => d::union(b',c')
  | (Zero::b', d::c') => d::union(b',c')
  | ((One tb)::b', (One tc)::c') =>
    let
      val t = link (tb, tc)
      val bc' = union (b',c')
    in
      Zero::(insertTree (t, bc'))
    end

fun split b =
  let
    (* even number of elements, split all trees *)
    fun split_even b =
      case b of
        nil => (nil, nil)
      | Zero::b' =>
        let
          val (c,d) = split_even b'
        in
          (Zero::c, Zero::d)
        end
      | (One t)::b' =>
        let
```

```
            val (l,r) = unlink t
            val (c,d) = split_even b'
          in
            ((One l)::c, (One r)::d)
          end
    in
      case b of
        nil => (nil, nil)
      | Zero::b' =>
          (* Even number of elements *)
          split_even b'
      | (One t)::b' =>
        (* Odd number of elements *)
        let
          val (c,d) = split_even b'
        in
          (insertTree (t,c), d)
        end
    end

end
```

USE BLACKBOARD TO DISCUSS THE DATA STRUCTURE

Logarithmic-work data structures such as trees are reasonably fast asymptotic data structures but not when compared to constant-time array based data structures that does not require memory allocation and deallocation. On current architectures, such array based implementations are on average one-order of magnitude faster in practice. We shall now see a technique that we call ***chunking*** that can significantly reduce the cost of some of these operations. Specifically, in the case of the bag data structure we shall reduce the amortized cost of insertions and deletions to $O(\lg n/K)$, where K is the chunk size. This means that if we set the chunk size to be a value such as 512, the cost is no more than $10 \cdot 64/512 = 1.25$, which is 25% slower than the array based data structure, an acceptable overhead, when the size of the data structure is no more than 2^{64}. Note that the size assumption is reasonable because on 64-bit architectures we can't store more than 2^{64} items in the main memory. Via chunking we shall also reduce the cost of union and split operations to $O(\lg(n/K))$, which is not nearly as significant an improvement as with the insert and delete operations.

The basic idea behind the chunking technique is to store chunks of data items instead of individual data items at the leaves of the trees. Each chunk is a size-K array.

17.3.2.1 Amortization with a K-Size Buffer

Let's start with an approach that is quite common in practice but does not actually yield a provable improvement. The idea is to extend the bag data structure with a ***buffer*** that sits in front of the list of trees. This buffer is implemented as a chunk that can hold K items.

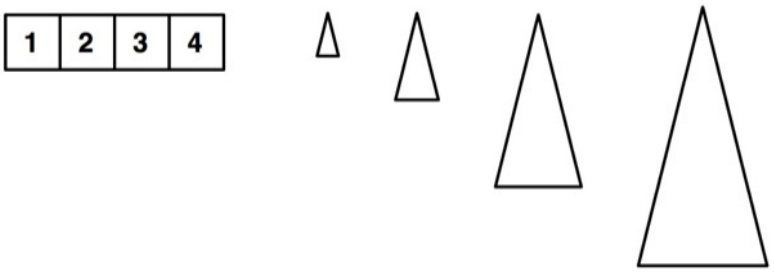


Figure 18: A bag data structure with a size K buffer.

To insert an element into the bag, we follow the following algorithm.

1. Check if there is space in the buffer. If so, insert the element into the buffer.
2. If there is no space in the buffer, then insert the buffer into the tree list. Allocate a new buffer and insert the element into the buffer.

To remove an element from the bag, we mirror the insertion operation.

1. Check if the buffer is not empty. If not, then remove an element from the buffer.
2. If the buffer is empty, then remove a chunk from the tree list, and set it as the buffer. Remove an element from the new buffer.

The `union` and `split` operations on chunked bags follow essentially the same algorithm as bags, but operate on chunks instead of individual items. In addition, we take into account the buffers. In the `union` operation, we combine the two buffers. If the combined number of items exceed the capacity of the buffer, we insert a new chunk into the tree list. In the `split` operation, we split the buffer into two equally sized buffers.

By using this approach, we make sure that the tree list contains only full chunks. By inserting and removing elements from the buffer first, we also reduce the cost of insertions and deletions.

In the worst case, however, the cost of insertions and deletions can still be large. For example, consider the case of an insertion that inserts the buffer into the tree list. Now follow this insertion with two deletions, the latter of which remove a chunk from the tree list. Another insertion will insert the chunk again into the tree list. Thus, we can force a chunk insertion or deletion nearly every other operation. This is far more frequent than what we were aiming for and does not lead to a factor $1/K$ reduction in cost.

REMARK: Even though this algorithm is not provably efficient, variants of it are used in practice. For example, at the time of writing, Standard Template Library's doubly ended queue (deque) data structure uses this approach to amortize the cost of insertion and deletion and does exhibit the aforementioned worst-case behavior.

17.3.3 Amortization with $2K$-Size Buffers

The problem with the amortization approach using a K-size buffer is that it is unable to ensure that K operations are performed between operations on the tree list. The solution for this actually quite simple. What need is to use a larger capacity buffer so that a removal from the tree list does not lead to an insertion into the tree list after a small number of operations. Since we remove K elements from the tree list each time, we need additional space for K more items, and thus a buffer that can hold $2K$ items.

To insert an element into the bag, we follow the following algorithm.

1. Check if there is space in the buffer. If so, insert the element into the buffer.
2. If there is no space in the buffer, then take K items from the buffer and make a chunk. Insert the chunk into the tree list and the item into the buffer.

To remove an element from the bag, we mirror the insertion operation.

1. Check if the buffer is not empty. If not, then remove an item from the buffer.
2. If the buffer is empty, then remove a chunk from the tree list, and place the items into the buffer. Remove an item from the new buffer.

The `union` and `split` operations follow a similar procedure.

The algorithm amortizes the cost of an insertion because the cost of every insertion into the tree list requires the buffer to be full: the K items that we insert into the chunk can thus pay for the cost of the tree operation. Similarly, a deletion from the tree list must be followed by at least K deletions from the buffer to delete the elements that are removed from the tree list.

We can prove that this algorithm achieves the desired factor $1/K$ reduction in the cost of insertions and deletions. There are several ways we can go about this proof. One option is to realize that each remove operation can be paired with an insert operation and thus we can assign remove operations an amortized cost of 0. To bound the amortized cost for insertions, we can use various techniques. One technique is to assign to the data structure that contains n items a potential function that is of the form $\min\left(0, (\text{size of the buffer} - K) \cdot \frac{\lg n}{K}\right)$. Since this potential function is always non-negative, it is a valid potential function.

We can now prove that the amortized cost of insertion is $O(\frac{\lg n}{K})$. There are two cases to consider

1. There is space in the buffer. The actual cost of insertion is $O(1)$. The amortized cost is actual cost plus change in the potential $O(\frac{1}{\lg n}$.
2. There is no space in the buffer and we insert K items into the middle list. The actual cost is $O(\lg n)$. The amortized cost is $O(1)$ because the potential decreases by $O(\lg n)$.

We can also account for deletions separately, without charging them to insertions but this requires a bit more care.

Exercise

Show that, using a $2K$ size buffer, insertian and deletions operations require $O(\frac{\lg n}{K}$ amortized work/span.

17.4 Chapter Notes

The bag data structure presented in this chapter is based on the random-access list data structure of Chris Okasaki [Chris Okasaki. Purely Functional Data Structures. PhD Thesis, Carnegie Mellon University]. The version presented here extends Okasaki's presentation with union and split functions that can be used to union two bags and to split a bag evenly into two bags.

18 Chapter: Work Stealing in Dedicated Environments

The work-stealing algorithm is a solution to the ***online scheduling problem***, where we are given a P processes and a computation dag that unfolds dynamically during execution, and asked to construct an execution schedule with minimal length, while spending as little work and time for scheduling itself.

We consider multithreaded computations represented as dags as described in an earlier chapter. To streamline the analysis, we assume without loss of generality that the root vertex has a single child. This assumption eliminates a few corner cases from the analysis.

18.1 Work-Stealing Algorithm

In work stealing, each process maintains a ***deque***, doubly ended queue, of vertices. Each process tries to work on its local deque as much as possible. Execution starts with the root vertex in the deque of one of the processes. It ends when the final vertex is executed.

A work stealing scheduler operates as described by our generic scheduling algorithm but instead of operating on threads, it operates on vertices of the dag. To obtain work, a process pops the vertex at the bottom of its deque and executes it. We refer to the vertex being executed by a process as the ***assigned vertex***. When executed, the ready vertex can make the other vertices ready, which are then pushed onto the bottom end of the deque in an arbitrary order. If a process finds its deque empty, then the process becomes a ***thief***. A thief picks a ***victim*** process at random and attempts to steals a thread from another it by popping a thread off the top of the victim's deque. The thief performs steal attempts until it successfully steals a thread, at which point, the thief goes back to work and the stolen thread becomes its assigned thread.

The pseudo-code for the algorithm is shown below. All deques are initially empty and the root vertex is assigned to the process zero. The algorithm operates in ***rounds***. In each round, a process executes the assigned vertex if any, pushes the newly enabled vertices to its deque, and obtains a new assigned vertex from its deque. If the round starts with no assigned vertex then the process becomes a thief performs a steal attempt. Note that a steal attempt start and completes in the same round.

Such a ***steal attempt*** can fail if

1. the victim's deque is empty, or
2. contention between processes occurs and the vertex targeted by the thief is executed by the process that own the dequer or stolen by another thief.

For the analysis of the algorithm, we shall assume that each instruction and each deque operations executes in a single step to execute. As a result, each iteration of the loop, a round, completes in constant steps.

```
// Assign root to process zero.
assignedVertex = NULL
if (self == ProcessZero) {
  assignedVertex = rootVertex
}


// Run scheduling loop.
while (computationDone == false) {

  // Execute assigned vertex.
  if (assignedVertex <> NULL) {
    (nChildren, child1, child2) = execute (assignedVertex)

    if (nChildren == 1) {
      self.pushBottom child1
    }
    else {
      self.pushBottom child1
      self.pushBottom child2
    }
    assignedVertex = self.popBottom ()
  }
  else {
    // Make steal attempt.
    victim = randomProcess ()
    assignedVertex = victim.popTop ()
  }
}
```

Note

Note that when a vertex enables two vertices they are both pushed onto the bottom of the deque in an order that is unspecified. The analysis holds for any such order. However, realistic implementations will operate at the granularity of threads as defined in for example an earlier chapter and by the generic scheduling algorithm. To make this algorithm consistent with such implementations, we would push the vertex that is next in the thread last, so that it is executed next. Pushing and popping the vertex in of course unnecessary and should be avoided in an implementation. For the purposes of the analysis, this adds a small constant factor that we don't care to account for.

18.1.1 Deque Specification

The deque supports three methods:

1. `pushBottom`, which pushes a vertex at the bottom end of the deque.
2. `popBottom`, which returns the vertex at the bottom end of the deque if any, or returns `NULL` otherwise.
3. `popTop`, returns the vertex at the top end of the deque, if any, or returns `NULL` if the deque is empty.

For the analysis, we assume that these operations take place atomically. We can think of them as starting by taking a lock for the deque, performing the desired operation, and releasing the lock.

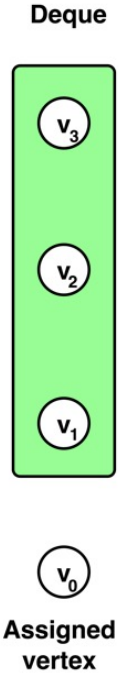
For the analysis, we shall assume that all these operations take constant time and in fact complete in one step. This assumption is somewhat unrealistic, because it is not known whether `popTop` can be implemented in constant time. But a relaxed version of `popTop`, which allows `popTop` to return `NULL` if another concurrent operation removes the top vertex in the deque, accepts a constant-time implementation. This relaxed version suffices for our purposes.

18.1.2 Work sequence of a process

Consider the execution of the work stealing algorithm and let q be any process. We define the ***work sequence*** of q as the sequence of vertices defined by the assigned vertex of q followed by the vertices in its deque ordered from bottom to top. If a vertex is in the work sequence of a process, then we say that it ***belongs*** to that process.

Example 18.1 Work sequence

Consider the deque for a process shows below along with the assigned vertex. The work sequence for this process is $\langle v_0, v_1, v_2, v_3 \rangle$.



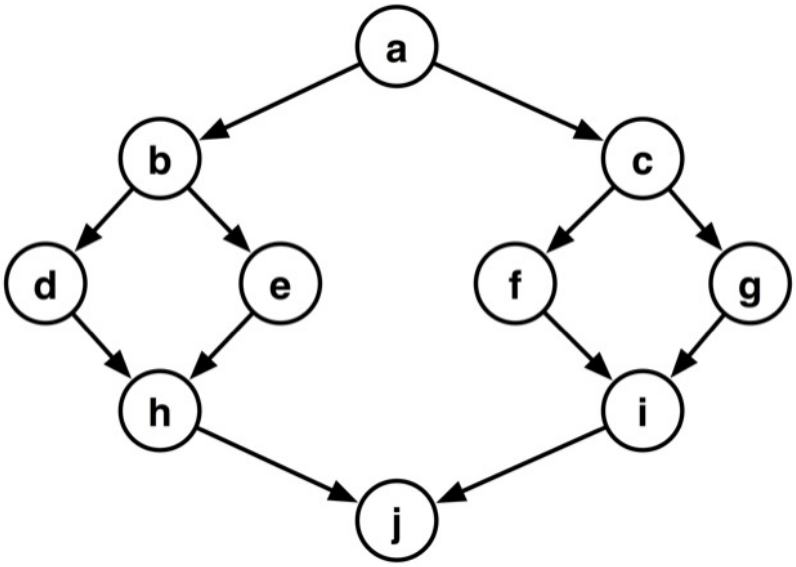
Now, if the process completes executing v_0, which enables no new children, then the work sequence consist of the vertices in the deque, i.e., $\langle v_1, v_2, v_3 \rangle$. If the process, later removes v_1 from its deque and starts working on it, i.e., v_1 becomes the assigned vertex, then the work sequence remains the same, i.e., $\langle v_1, v_2, v_3 \rangle$.

18.1.3 Enabling Tree

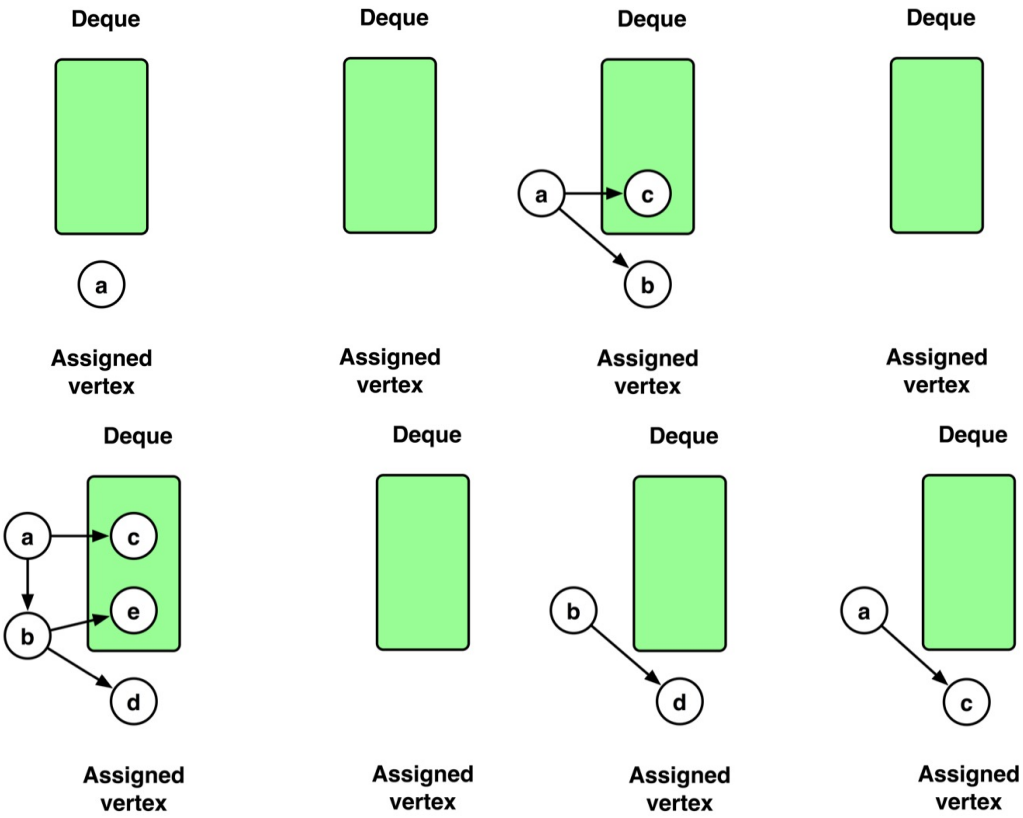
Let's recall first the notion of an enabling three. If execution of u enables v, then we call the edge (u, v) an ***enabling edge*** and call u the ***designated parent*** of v. Every vertex except for the root has a designated parent. Therefore the subgraph of the dag consisting of the enabling edges form a rooted tree, called the ***enabling tree***. Note each execution can have a different enabling tree.

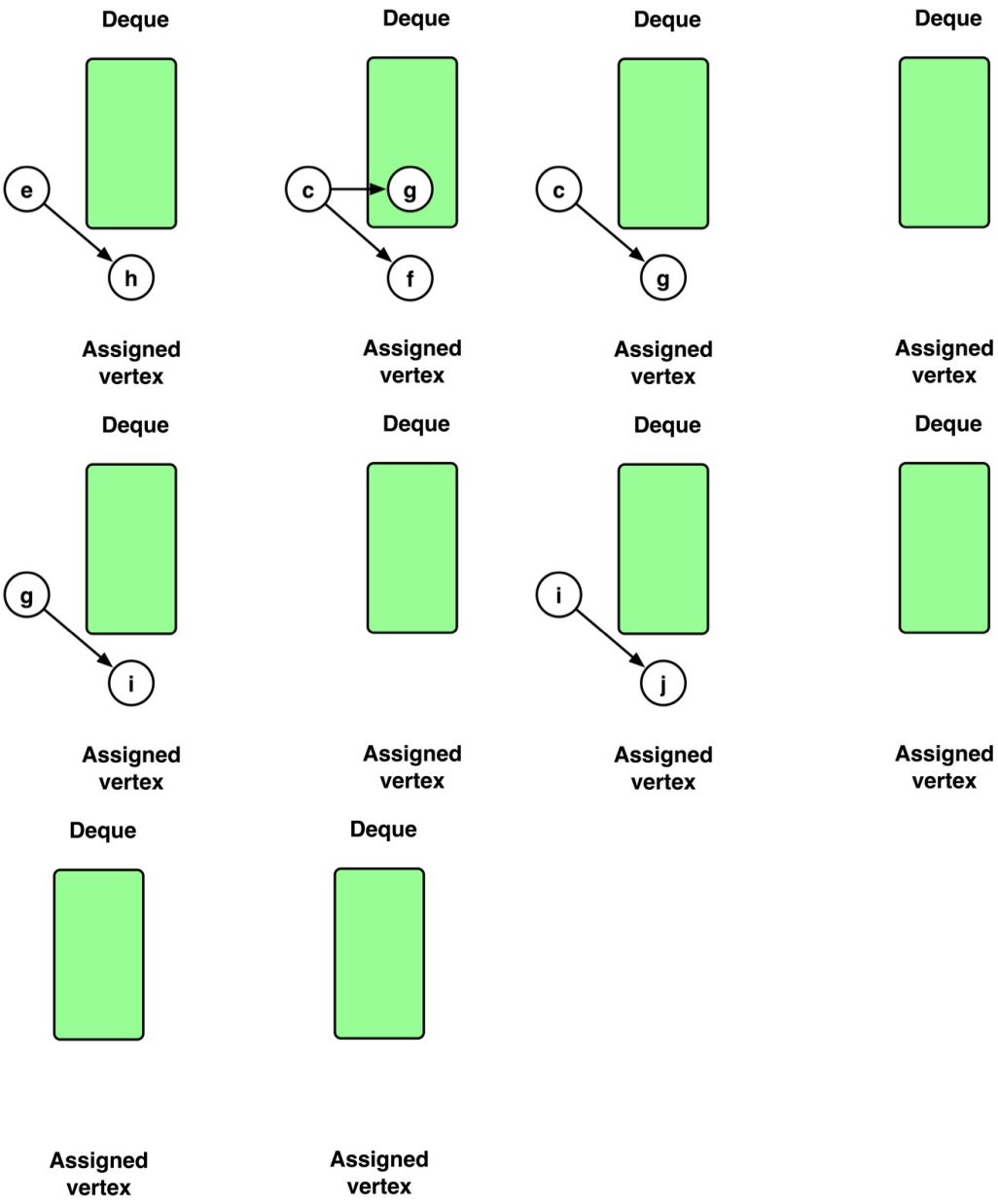
18.1.4 An Example Run with Work Stealing

Consider the following computation dag.

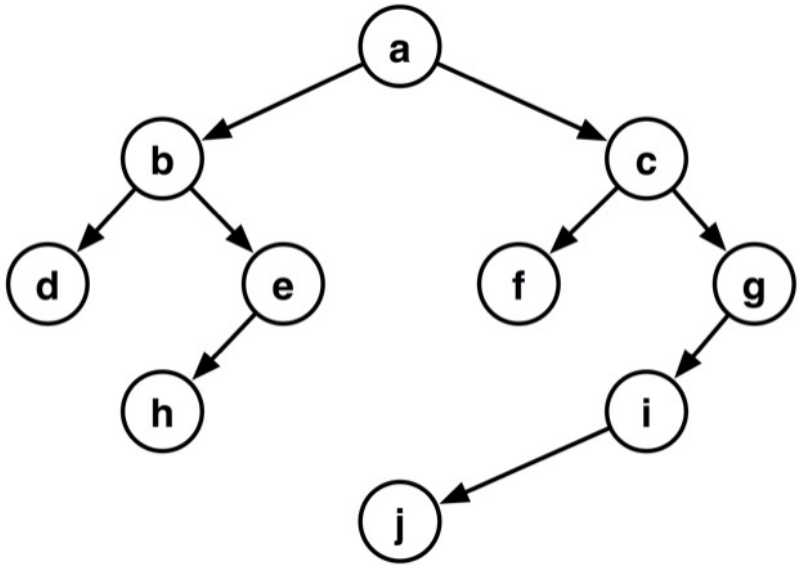


The following drawings illustrate a 2-process execution of this dag using work stealing. Time flows left to right and top to bottom.





The enabling tree for this execution is shown below.

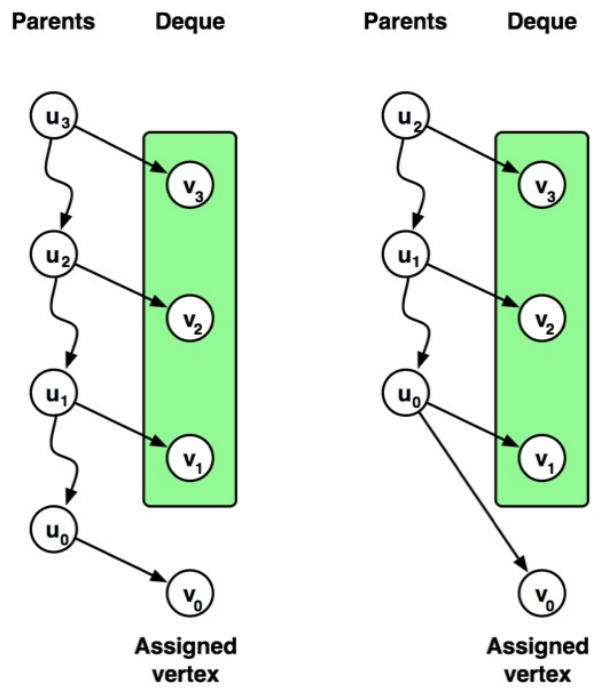


18.1.5 Structural Lemma

Lemma[Structural Lemma]

Consider any time in an execution of the work-stealing algorithm after the execution of the root vertex. Let $v_0, v_1, \ldots, v_k$ denote the work sequence of any process. Let $u_0, u_1, \ldots, u_k$ be the sequence consisting of the designated parents of the vertices in the work sequence in the same order. Then u_i is an ancestor of u_{i-1} in the enabling tree. Moreover, we may have $u_0 = u_1$ but for any $2 \leq i \leq k$ $u_{i-1} \neq u_i$, that is the ancestor relationship is proper.

Structural lemma illustrated.



The proof of the structural lemma is by induction on the number of rounds of the execution.

Consider the first round. At the initialization and before the beginning of the first round, all deques are empty, root vertex is assigned to process zero but has not been executed. The root vertex is then executed. By assumption, the root vertex has a single child v, which becomes enabled and pushed onto the deque and popped again becoming the assigned vertex at the beginning of the second round. At any of point in time after the execution of the root, process zero's work sequence consist of v. The designated parent of v is the root vertex and the lemma holds trivially.

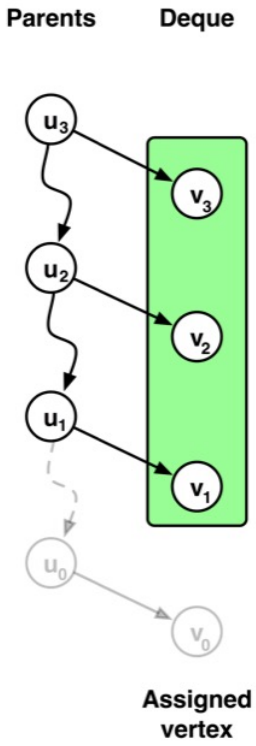
For the inductive case, assume that the lemma holds up to beginning of some later round. We will show that it holds at any point during the round and also after the completion of the round.

Consider any process and its deque. We have two cases to consider.

Case 1: There is an assigned vertex, v_0, which is executed.

By the definition of work sequences, we know that $v_1, \ldots, v_k$ are the vertices in the deque. Let $u_1, \ldots, u_k$ be their designated parents. By induction, we know that u_i is an ancestor of u_{i-1} in the enabling tree and the ancestor relationship is proper except for $i = 1$, where it is possible that $u_0 = u_1$. Immediately after the execution of the assigned node, the work sequence of the process consists of all the vertices in the deque and the lemma follows.

Structural lemma illustrated after the assigned vertex is executed.



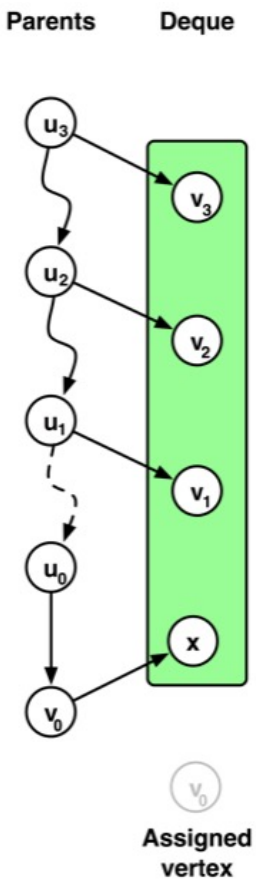
After the execution of the assigned vertex v_0, we have several sub-cases to consider.

Case 1.1: execution of v_0 enables no children.

Since the deque remains the same, the lemma holds trivially.

Case 1.2: execution of v_0 enables one child x, which is pushed onto the bottom of the deque. In this case, v_0 becomes the parent of x. The lemma holds.

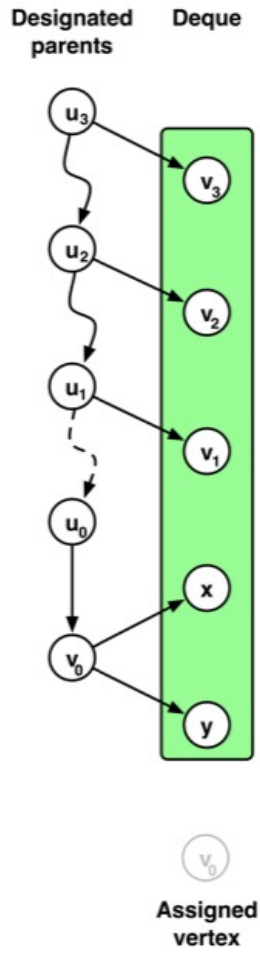
Structural lemma illustrated after the assigned vertex enables one child.



Case 1.2: execution of v_0 enables two children x, y, which are pushed to the bottom of the deque in an arbitrary order.

In this case, v_0 becomes the parent of x and y. We need to show that $v_0 \neq u_1$. This holds because $v_0 \neq u_0$ and $v_0 \neq u_1$. The lemma holds.

Structural lemma illustrated after the assigned vertex enables two children.



After the execution of the assigned vertex completes and the children are pushed, the process pops the vertex at the bottom of the deque. There are two cases to consider.

1. If the deque is empty, then the process finds no vertex in its deque and there is no assigned vertex at the end of the round, thus the work sequence is empty and the lemma holds trivially.
2. If the deque is not empty, then the vertex at the bottom of the deque becomes the new assigned vertex. The lemma holds trivially because making the bottom vertex the assigned vertex has no impact on the work sequence of the process and thus the correctness of the lemma.

Case 2: A successful steal takes place and removes the top vertex in the deque. In this case, the victim process loses its top vertex, which becomes the assigned vertex of the thief. The work sequence of the victim loses its rightmost element. It is easy to see that the lemma continues to hold for the victim. When the stolen vertex is assigned, the work sequence of the thief consist of just the stolen vertex and the lemma holds for the thief.

18.2 Analysis

The strategy analysis is to assign a potential to each vertex and show that the potential decreases geometrically as the execution proceeds.

18.2.1 Weights

If $d(u)$ is the depth of a vertex u in the enabling tree, then we define the weight of u, written $w(u)$ as follows $w(u) = S - d(u)$. The root has weight S. Intuitively, the weight is equal to the distance of a vertex from the completion.

A crucial corollary of the structural lemma is that the weights of the vertices decrease from top to bottom.

Corrollary

Consider the work sequence of a process $v_0, v_1, v_2 \ldots v_k$. We have $w(v_0) \leq w(v_1) < w(v_2) \ldots w(k-1) < w(v_k)$.

18.2.2 Balls and Bins Game

One crucial fact behind the analysis is a probabilistic lemma, called the Balls and Bins lemma. This lemma proves something relatively intuitive: if you throw as many ball as there are bins, chances are good that you will have a ball in at least a constant fraction of the bins, because chances of all balls landing in a small number of bins is low.

Lemma[Balls and Bins]

Suppose that P balls are thrown uniformly and randomly into P bins, where bin $1 \leq i \leq P$ has weight $W_i \geq 0$, and $W = \sum_{i=1}^{P} W_i$. For each bin, define the random variable

$$X_i = \begin{cases} W_i & \text{if a ball lands in bin } i \\ 0 & \text{otherwise} \end{cases}$$

If $X = \sum_{i=1}^{P} X_i$, then for any $\beta, 0 < \beta < 1$, we have $P[X \geq \beta W] > 1 - \frac{1}{(1-\beta)e}$.

Proof

The proof of the lemma is a relatively straightforward application of Markov's inequality. Consider the random variable $W_i - X_i$. This random variable takes on the value 0 if a ball lands in bin i and W_i otherwise. Thus, we have

$$\begin{aligned} E[W_i - X_i] &= W_i \cdot (1 - 1/P)^P, \text{and} \\ E[W - X] &= W \cdot (1 - 1/P)^P. \end{aligned}$$

We know that $\lim_{P \to \infty} (1 - 1/P)^P = 1/e$ and furthermore that the derivative of this function is non-negative (the function is non-decreasing). Thus we know that $(1 - 1/P)^P \leq 1/e$.

It follows that $E[W - X] \leq W/e$.

Since the random variable $W - X$ is a non-negative random variable, we can apply Markov's inequality:

$$P[W - X > (1 - \beta)W] \leq \frac{E[W - X]}{(1 - \beta)W}.$$

It follows that

$$P[X < \beta W] < \frac{1}{(1 - \beta)e},$$

and thus

$$P[X \geq \beta W] > 1 - \frac{1}{(1 - \beta)e}.$$

Example 18.2 An application of the lemma

Let's calculate the probability for $\beta = 1/2$. By the lemma, we know that if P balls are thrown into P bins, then the probability that the total weight of the bins that have a ball in them is at least half the total weight is $P[X \geq \frac{W}{2}] 1 - \frac{1}{0.5e}$. Since $e > 2.71$, this quantity can be calculated as at least 0.25. We thus conclude that we using the Ball and Bins lemma, we can "collect" at least half of the weight with probability at least 0.25

18.2.3 Bound in terms of Work and Steal Attempts

Lemma[Work and Steal Bound]

Consider any multithreaded computation with work W. The P-process execution time is $O(W/P + A/P)$ steps where A is the number of steal attempts.

Proof

Consider the execution in terms of rounds

1. If a vertex is executed in that round, then the processor places a token into the work bucket.
2. If a steal attempt takes place in that round, then the process places a token into the steal-attempt bucket.

There are exactly A tokens in the steal-attempt bucket and exactly W tokens is the work bucket. Thus the total number of tokens is at most $W + A$. Since in each round a process either executes a vertex or attempts a steal, and since each round is a constant number of steps, the P-process execution time is $T_P = O(\frac{W+A}{P})$.

Note

The proof assumes that each instructions including deque operations takes a (fixed) constant number of steps, because it assumes that each round contributes to the work or to the steal bucket. If this assumption is not valid, then we might need to change the notion of rounds so that they are large enough for steals to complete.

18.2.4 Bounding the Number of Steal Attempts

Our analysis will use a potential-function based method. We shall divide the computation into phases each of which decreases the potential by a constant factor.

We start with a few definitions. Consider some round i.

1. Let R_i denote the set of ***ready vertices*** in the beginning of that round.
2. A vertex is R_i is either assigned to a process or is in a deque. Let $R_i(q)$ denote the set of ***ready vertices belonging to a process*** q at the beginning of round i; these are exactly the vertices in the work sequence of that process.
3. For each vertex $v \in R_i$, we define its ***potential*** as
   a. $\phi_i(v) = 3^{2w(v)-1}$, if v is assigned, or
   b. $\phi_i(v) = 3^{2w(v)}$, otherwise.

Note that potential of a vertex is always a natural number.

1. The ***potential of process*** q is $\Phi_i(q) = \sum_{v \in R_i(q)} \phi_i(v)$.
2. We write H_i (mnemonic for "Hood") for the set of processes whose deques are empty at the beginning of round i. We write $\Phi_i(H_i)$ for ***the total potential of the processes*** H_i, $\Phi_i(H_i) = \sum_{q \in H_i} \Phi_i(q)$.
3. We write D_i for the set of other processes. We write $\Phi_i(D_i)$ for ***the total potential of the processes*** D_i, $\Phi_i(D_i) = \sum_{q \in D_i} \Phi_i(q)$.
4. Define the ***total potential at round*** i, written Φ_i as $\Phi_i = \sum_{v \in R_i} \phi_i(v)$. We can write the total potential in round i as follows $\Phi_i = \Phi_i(H_i) + \Phi_i(D_i)$.

Definition: Beginning and termination potential

At the beginning of the computation, the only ready vertex is the root, which has a weight of S, because it is also the root of the enabling tree. Therefore the potential in the beginning of the computation is 3^{2S-1}. At the end of the computation, there are no ready vertices and thus the potential is zero.

Lemma[Top-Heavy Deques]

Consider any round i and any process $q \in D_i$. The topmost vertex in q's deque contributes at least $3/4$ of the potential of process q.

Proof

This lemma follows directly from the structural lemma. The case in which the topmost vertex v contributes the least of the process is when the assigned vertex and v have the same parent. In this case, both vertices have the same depth and thus we have

$$\Phi_i(q) = \phi_i(v) + \phi_i(x) = 3^{2w(v)} + 3^{2w(x)-1} = 3^{2w(v)} + 3^{2w(v)-1} = (4/3)\phi_i(v).$$

Lemma[Vertex Assignment]

Consider any round i and let v be a vertex that is ready but not assigned at the beginning of that round. Suppose that the scheduler assigns v to a process in that round. In this case, the potential decreases by $2/3 \cdot \phi_i(v)$.

Proof

This is a simple consequence of the definition of the potential function: $\phi_i(v) - \phi_{i+1}(v) = 3^{2w(v)} - 3^{2w(v)-1} = 2/3\phi_i(v)$.

Lemma[Decreasing Total Potential]

Total potential does not increase from one round to the next, i.e., $\Phi_{i+1} \leq \Phi_i$.

Proof

Let's now consider how the potential changes during each round. There are two cases to consider based on scheduler actions.

Case 1: Suppose that the scheduler assign a vertex v to a process. By the Vertex Assignment Lemma, we know that the potential decreases by $2/3 \cdot \phi_i(v)$. Since $\phi_i(v)$ is positive, the potential decreases.

Note that this calculation holds regardless of where in the deque the vertex v is. Specifically, it could have been the bottom or the top vertex.

Case 2: suppose that the scheduler executes a vertex v and pushes its children onto the deque. There are two sub-cases to consider.

Case 2.1: suppose that the scheduler pushes onto the deque the only child x of a vertex v. Assuming that the child stays in the deque until the beginning of the next round, the potential decreases by

$$\phi_i(v) - \phi_{i+1}(x) = 3^{2w(v)-1} - 3^{2w(v)-2} = 3^{2w(v)-1}(1-1/3) = 2/3 \cdot \phi_i(v).$$

Case 2.2: suppose that the scheduler pushes onto the deque two children x, y of a vertex v. Assuming that the children remain in the deque until the beginning of the next round, then potential decreases by

$$\begin{aligned} \phi_i(v) - \phi_{i+1}(x) - \phi_{i+1}(y) &= 3^{2w(v)} - 3^{2w(v)-2} - 3^{2w(v)-2} \\ &= 3^{2w(v)-1}(1-1/3-1/3) \\ &= 1/3 \cdot \phi_i(v). \end{aligned}$$

Since $\phi_i(v)$ is positive, the potential decreases in both cases. Note that, it is safe to assume that the children remain in the deque until the next round, because assignment of a vertex decreases the potential further.

In each round each process performs none, one, or both of these actions. Thus the potential never increases.

We have thus ustablished that the potential decreases but this by itself does not suffice. We also need to show that it decreases by some significant amount. This is our next step in the analysis. We show that after P steal attempts the total potential decreases with constant probability.

Lemma[P Steal Attempts]

Consider any round i and any later round j such that at least P steal attempts occur at rounds from i (inclusive) to j (exclusive). Then, we have

$$Pr[\Phi_i - \Phi_j \geq \frac{1}{4}\Phi_i(D_i)] > \frac{1}{4}.$$

Proof:

First we use the Top-Heavy Deques Lemma to establish the following. If a steal attempt targets a process with a nonempty deque as its victim, then the potential decreases by at least of a half of the potential of the victim.

Consider any process q in D_i and let v denote the vertex at the top of its deque at round i. By Top-Heavy Deques Lemma, we have $\phi_i(v) \geq \frac{3}{4}\Phi_i(q)$.

Consider any steal attempt that occurs at round $k \geq i$.

1. Suppose that this steal attempt is sucessful with `popTop` returning a vertex. The two subcases both follow by the Vertex Assignment Lemma.
   a. If the returned vertex is v, then after round k, vertex v has been assigned and possibly executed. Thus, the potential has decreased by at least $\frac{2}{3}\phi_i(u)$.
   b. If the returned vertex is not v, then v is already assigned and possibly executed. Thus, the potential has decreased by at least $\frac{2}{3}\phi_i(v)$.
2. Suppose that the steal attempt is not succesful, and `popTop` returns `NULL`. In this case, we know that q's deque was empty during `popTop`, or some other `popTop` or `popBottom` operation returned v. In all cases, vertex v has been assigned or possibly executed by the end of round k and thus, potential decreases by $\frac{2}{3}\phi_i(v)$.

Thus, we conclude that if a thief targets a process $q \in D_i$ as victim at round $k \geq i$, then the potential decreases by at least

$$\frac{2}{3}\phi_i(u) \geq \frac{2}{3}\frac{3}{4}\Phi_i(q) = \frac{1}{2}\Phi_i(q).$$

Second, we use Ball and Bins Lemma to establish the total decrease in potential.

Consider now all P processes and P steal attempts that occur at or after round i. For each process q in D_i, assign a weight of $\frac{1}{2}\Phi_i(q)$ and for each process in H_i, assign a weight of 0. The total weight is thus $\frac{1}{2}\Phi_i(D_i)$. Using the Balls-and-Bins Lemma with $\beta = 1/2$, we conclude that the potential decreases by at least $\beta W = \frac{1}{4}\Phi_i(D_i)$ with probability greater that $1 - \frac{1}{(1-\beta)e} > \frac{1}{4}$.

Important
For this lemma to hold, it is crucial that a steal attempt does not fail unless the deque is empty or the vertex being targeted at the time is popped from the deque is some other way. This is why, we required the `popTop` operation called by a process to fail only if the top vertex is removed from the deque by another process.

Theorem[Run-Time Bound]

Consider any multithreaded computation with work W and span S and execute it with non-blocking work stealing with P processes in a dedicated environment. The exection time is

1. $O(W/P + S)$ in expectation, and
2. $O(W/P + S + \lg(1/\varepsilon))$ with probability at least $1 - \varepsilon$ for any $\varepsilon > 0$.

Proof

The Steal Attempt-Bound Lemma bounds the execution time in terms of steal attempts. We shall prove bounds on the number of steal attempts.

We break execution into ***phases*** of $\Theta(P)$ steal attempts. We show that with constant probability, a phase causes the potential to drop by a constant factor, and since the potential starts at $\Phi_0 = 3^{2S-1}$ and ends at zero, and is always an natural number, we can bound the number of phases.

The first phase begins at round 1 and ends at the first round k, where at least P throws occur during the interval $[1,k]$. The second phase begins at round $k+1$ and so on.

Consider a phase $[i,j)$, where the next phase begins at round j. We show that $\Pr[\Phi_j \leq \frac{3}{4}\Phi_i] < \frac{1}{4}$.

Recall that the potential can be partitioned as $\Phi_i = \Phi_i(H_i) + \Phi_i(D_i)$.

1. Since the phase contains at least P steal attempts, by P Steal Attempts Lemma, we know that $P[\Phi_i - \Phi_j \geq \frac{1}{4}\Phi_i(D_i)] > \frac{1}{4}$.
2. We need to show that the potential also drops by a constant fraction of $\Phi_i(H_i)$. Consider some process q in H_i. If q does not have an assigned vertex, then $\Phi_i(q) = 0$. If q has an assigned vertex v, then $\Phi_i(q) = \phi_i(v)$. In this case, process q executes v at round i and the potential decreases by at least $\frac{1}{3}\phi_i(u)$ by an argument included in the proof of Decreasing Potential Lemma.

Summing over all processes in H_i, we have $\Phi_i - \Phi_j \geq \frac{1}{3}\Phi_i(H_i)$.

Thus we conclude that $P[\Phi_i - \Phi_j \geq \frac{1}{4}\Phi_i] > \frac{1}{4}$. In other words, we established that in any phase, potential drops by a quarter with some probability $\frac{1}{4}$.

Define a phase to be ***successful*** if it causes the potential do decrease by at least a quarter fraction. We just established that phase is successful with probability at least 0.25. Since the start potential $\Phi_0 = 3^{2S-1}$ and ends at zero and is always an integer, the number of successful phases it as most $(2S-1)\log_{4/3} 3 < 8S$.

The expected number of phases to obtain a single successful phase is distributed geometrically with expectation 4. Therefore, the total expected number of phases is $32S$, i.e., $O(S)$. Since each phase contains $O(P)$ steal attempts, the expected number of steal attempts is $O(SP)$.

We now establish the high-probability bound.

Suppose that the execution takes $n = 32S + m$ phases. Each phase succeds with probability at least $p = \frac{1}{4}$, so the expected number of successes is at least $np = 8S + m/4$. Let X the number of successes. By Chernoff bound

$$P[X < np - a] < e^{-a^2/2np},$$

with $a = m/4$. Thus if we choose $m = 32S + 16\ln 1/\varepsilon$, then we have

$$P[X < 8S] < e^{-(m/4)^2/(16S+m/2)} \leq e^{-(m/4)^2/(m/2+m/2)} = e^{-m/16} \leq e^{-16\ln 1/\varepsilon/16} = \varepsilon.$$

Thus the probabilty that the execution takes $64S + 18\ln 1/\varepsilon)$ phases or more is less than ε.

We conclude that the number of steal attempts is $O(S + \lg 1/\varepsilon P)$ with probability at least $1-\varepsilon$.

18.3 Chapter Notes.

The material presented here is a adapted from the paper:

N. S. Arora, R. D. Blufome, and C. Greg Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. Theory of Computing Systems, 34(2), 2001.

In this chapter, we consider dedicated environments, and simplify and streamline the proof for this case. The original paper considers multiprogrammed environments.

19 Chapter: Parallelism and Concurrency

We distinguish between parallelism and concurrency.

We use the term ***parallel*** to refer to an algorithm or application that performs multiple computations at the same time for the purposes of improving the completion or run time of the algorithm or application. The input-output behavior or the extrinsic semantics of a parallel computation can be defined purely sequentially, even though the computation itself executes in parallel.

We use the term ***concurrent*** to refer to a computation that involves independent agents, which can be implemented with processes or threads, that communicate and coordinate to accomplish the intended result. The input-output behavior or the extrinsic semantics of a concurrent application cannot be defined purely sequentially; we must consider the interaction between different threads.

In other words, parallelism is a property of the platform (software and/or hardware) on which the computation takes place, whereas concurrency is a property of the application. A concurrent program can be executed serially or in parallel.

For example, the sorting algorithms that we covered earlier are parallel but they are not concurrent, because the algorithms do not involve communication between different, independent agents; an operating system on the other hand is a concurrent application, where many processes (agents) may run at the same time and communicate to accomplish a task. For example, when you request these notes to be downloaded to your laptop, the kernel, the file system, and your browser communicate to accomplish the task. Web browsers are typically written as concurrent applications because they may include multiple processes to communicate with the operating system, the network, and the user, all of which coordinate as needed.

Parallelism and concurrency are orthogonal dimensions in the space of all applications. Some applications are concurrent, some are not. Many concurrent applications can benefit from parallelism. For example, a browser, which is a concurrent application itself as it may use a parallel algorithm to perform certain tasks. On the other hand, there is often no need to add concurrency to a parallel application, because this unnecessarily complicates software. It can, however, lead to improvements in efficiency.

The following quote from Dijkstra suggest pursuing the approach of making parallelism just a matter of execution (not one of semantics), which is the goal of the much of the work on the development of programming languages for parallel programs today.

> From the past terms such as "sequential programming" and "parallel programming" are still with us, and we should try to get rid of them, for they are a great source of confusion. They date from the period that it was the purpose of our programs to instruct our machines, now it is the purpose of the machines to execute our programs. Whether the machine does so sequentially, one thing at a time, or with considerable amount of concurrency, is a matter of implementation, and should *not* be regarded as a property of the programming language.
>
> — Edsger W. Dijkstra *Selected Writings on Computing: A Personal Perspective (EWD 508)*

19.1 Race conditions and concurrency

Even though concurrent behavior is not necessary in parallel computing, it can arise as a result of shared state. When shared variables are used for communication, for example by reading and writing the variable, between threads, we may have to treat the computation as a concurrent program to understand its behavior.

Consider a simple example function, which increments a counter in parallel for a specified number of times n. The code for such a function is shown below. The function `concurrent_counter` takes a number `n` and uses a parallel-for loop to increment the variable `counter`. When called with a large `n`, this function will create many threads that increment `counter` in parallel.

```
loop_controller_type concurrent_counter_contr ("parallel for");

void concurrent_counter(long n) {
  long counter = 0;

  auto incr = [&] () {
    counter = counter + 1;
  };

  par::parallel_for(concurrent_counter_contr, 0l, n, [&] (long i) {
      incr();
  });
```

```
  std::cout << "Concurrent-counter: " << "n = " << n << " result = " << counter << std:: ↩
      endl;
}
```

Since it is implemented by using our parallelism constructs, we may feel that the function `concurrent_counter` is parallel. To define its behavior, we may thus conclude that we can ignore the parallelism constructs, because such constructs only impact the way that the function is executed but not its semantics. So we may describe the semantics (input-output behavior) of `concurrent_counter` as taking a value `n`, incrementing it `n` times, and since `counter` starts out at `0`, printing the value `n` in the end. But this is hardly the case, as shown below by several runs of the function.

```
$ make example.opt
...

// problem size = 1K, counter = 1K
$ ./example.opt -example concurrent_counter -n 1000 -proc 16
Concurrent-counter:n = 1000 result = 1000
exectime 0.000
total_idle_time 0.000
utilization 0.9790

// problem size = 100M, counter < 10M.
$ ./example.opt -example concurrent_counter -n 100000000 -proc 16
Concurrent-counter:n = 100000000 result = 6552587
exectime 0.016
total_idle_time 0.046
utilization 0.8236

// problem size = 100M, counter < 10M.
$ ./example.opt -example concurrent_counter -n 100000000 -proc 24
Concurrent-counter:n = 100000000 result = 5711624
exectime 0.015
total_idle_time 0.046
utilization 0.8743

// problem size = 100M, counter ~ 10M.
$ ./example.opt -example concurrent_counter -n 100000000 -proc 24
Concurrent-counter:n = 100000000 result = 10785626
exectime 0.020
total_idle_time 0.128
utilization 0.7338
```

What has gone wrong is that the function `concurrent_counter` is in fact concurrent—not parallel—because the counter is a variable that is shared by many parallel threads, each of which read the value and update it by using multiple instructions. In fact, as we will see, the parallel updates to the counter lead to a race condition. More generally, a program written with our parallelism constructs is only truly parallel if and only if parallel computations are ***disentangled***, i.e., they don't interact by reading and writing memory, as for example in purely functional programs. Note that this restriction holds only for parts of the computation that execute in parallel, those that are not parallel can read and write from memory.

To understand more specifically what goes wrong, we need to rewrite our example to reflect the code, shown below, as executed by our machines. The only difference from the previous example is the `incr` function, which now uses separate operations to read the counter, to calculate the next value, and to write into it. The point is that the computation of reading-a-value-and-updating-it is not an ***atomic*** operation that executes in one step but a composite operation that takes place in three steps:

1. read `counter` into local, non-shared variable `current`,
2. increment `current`, and
3. write the updated value of `current` into shared variable `counter`.

```
loop_controller_type concurrent_counter_contr ("parallel for");

void concurrent_counter(long n) {
  long counter = 0;

  auto incr = [&] () {
    long current = counter;
    current = current + 1;
    counter = current;
  };

  par::parallel_for(concurrent_counter_contr, 0l, n, [&] (long i) {
      incr();
  });

  std::cout << "Concurrent-counter: " << "n = " << n << " result = " << counter << std:: ↩
      endl;
}
```

Since in a parallel execution, multiple threads can execute the function `incr` in parallel, the shared variable `counter` may not be updated as expected. To see this, consider just two parallel executions of `incr` that both start out the current value of `counter` let`s say `i`, locally calculate `i+1`, and write it back. The resulting value of `counter` is thus `i+1` even though `incr` has been applied twice. This is the reason for missed increments illustrated by the runs above.

19.2 Eliminating race conditions by synchronization

As discussed in an earlier chapter, we can use synchronization operations provided by modern hardware to eliminate race codnitions. In particular, we can use compare-and-swap in our example. Since compare-and-swap allows us to update the value stored at a location atomically, we can use it to implement our increment function as shown below. To this end, we declare `counter` as a atomic long type; the syntax `std::atomic<long>` declares an atomic `long`. In the function `incr`, we load the value of the counter into a local variable `current` and use compare-and-swap to replace the value of the `counter` with `current+1` by calling `compare_exchange_strong`, which implements the compare-and-swap operation. Since compare-and-swap can fail, because of another thread succeeding to update at the same time, function `incr` tries this update cycle until the operation succeeds.

```
loop_controller_type concurrent_counter_atomic_contr ("parallel for");

void concurrent_counter_atomic(long n) {
  std::atomic<long> counter;
  counter.store(0);

  auto incr = [&] () {
    while (true) {
      long current = counter.load();
      if (counter.compare_exchange_strong (current,current+1)) {
        break;
      }
    }
  };

  par::parallel_for(concurrent_counter_atomic_contr, 0l, n, [&] (long i) {
      incr();
  });

  std::cout << "Concurrent-counter-atomic: " << "n = " << n << " result = " << counter <<  ↩
      std::endl;
}
```

Here are some example runs of this code with varying number of processors. Note that the counter is always updated correctly, but the execution takes significantly longer time (approximately 10 times slower, accounting for the loss of parallelism due to the atomic operation) than the incorrect implementation with the race condition. This shows how expensive synchronization operations can be. Intuitively, synchronization operations are expensive, because they can require atomic access to memory and because they disable certain compiler optimizations. The reality is significantly more complex, because modern multicore computers have memory models that are only weakly consistent and because they rely on several levels of caches, which are nevertheless kept consistent by using "cache coherency protocols".

```
$ ./example.opt -example concurrent_counter_atomic -n 1000 -proc 16
Concurrent-counter-atomic:n = 1000 result = 1000
exectime 0.000
total_idle_time 0.000
utilization 0.9796

$ ./example.opt -example concurrent_counter_atomic -n 100000000 -proc 16
Concurrent-counter-atomic:n = 100000000 result = 100000000
exectime 14.718
total_idle_time 0.082
utilization 0.9997

$ ./example.opt -example concurrent_counter_atomic -n 100000000 -proc 24
Concurrent-counter-atomic:n = 100000000 result = 100000000
exectime 7.570
total_idle_time 0.388
utilization 0.9979
```