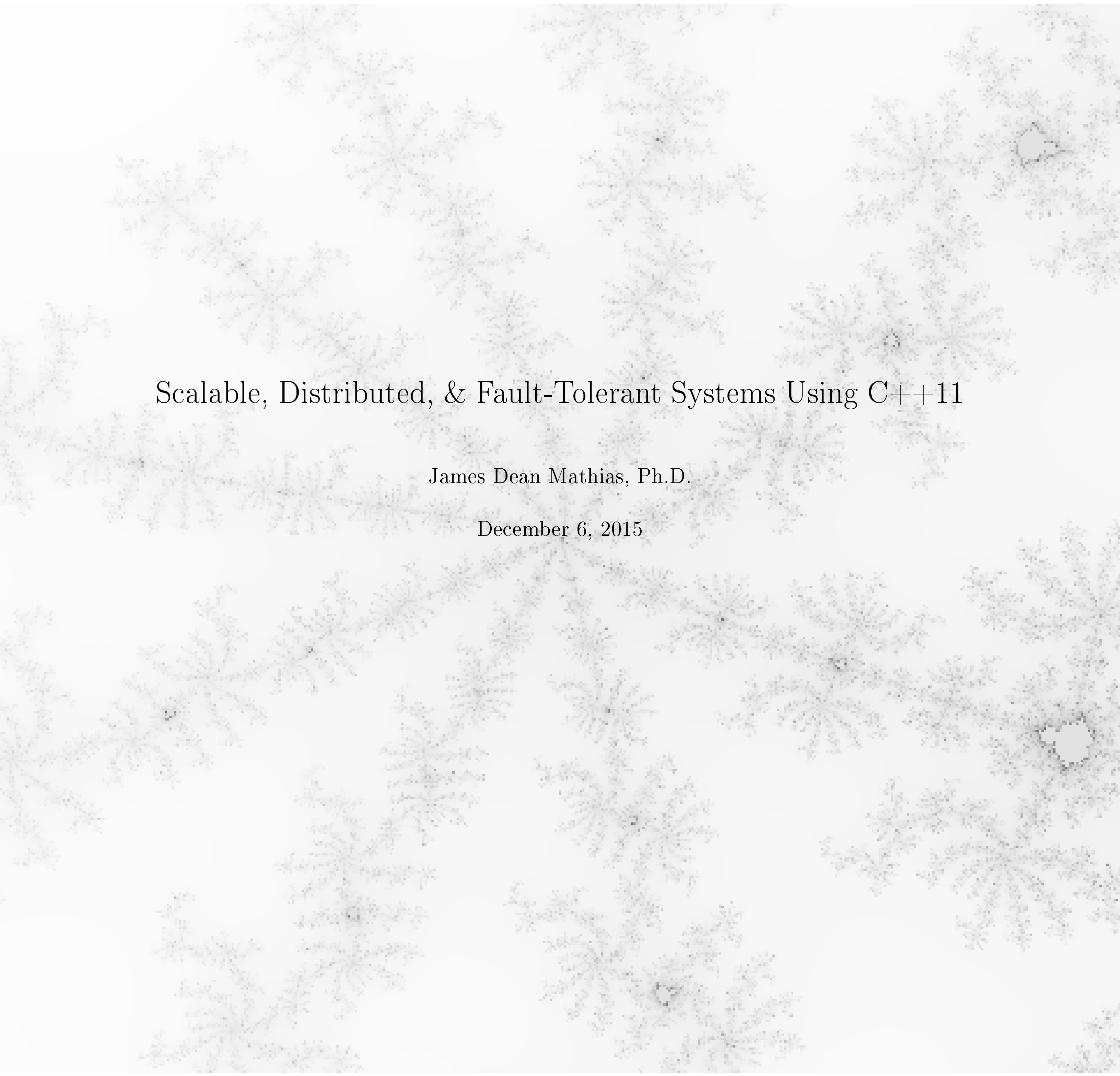

Scalable, Distributed, & Fault-Tolerant Systems Using C++11

James Dean Mathias, Ph.D.

December 6, 2015

Contents

1 Introduction

The purpose of this book is to act as a resource for those interested in building scalable, distributed, and fault-tolerant systems using C++11 and the Boost C++ libraries. Each of these topics is covered in detail, along with working code that can be used as the basis for developing or extending your applications. The combination of the C++11 language enhancements and standard library additions have made cross platform, distributed systems programming more accessible than ever before. This is a practical guide that shows how to develop these systems using C++11, with support from the Boost C++ libraries; this is a code heavy book.

This book assumes a background in C++ programming, familarity with multi-threading, synchronization, and network programming concepts, but not necessarily mastery or other expertise with them. Exposure to the new features introduced with C++11 language and standard library is not assumed, therefore an introduction to relevant topics are covered in Chapter 2. In particular, this includes lambda functions, along with the new threading and synchronization libraries. No background with Boost is assumed, an introduction to Boost and the relevant networking libraries is provided in Chapter 6. Enough discussion is given and code provided to be reasonably accessible to a wide range of developers, from the relatively inexperienced all the way up to senior developers.

The overarching approach presented in this book is that of task/data decomposition, presented in the context of scalability, distribution, and fault-tolerance. There are a number of tecchnical implementation details that are necessary which can work to obscure the general simplicity of the overall approach. It is important to maintain focus while working through these technical details that fundamentally, the building block is that of identifying and developing tasks that can be computed in parallel. This is true regardless if the application is intended to run on a single computer with any number of processors and CPU cores, or as a distributed application running across a wide range of heterogenous systems.

This book takes something of a *roll your own* approach to the construction of applications. It is recognized there are existing frameworks such as OpenMP[1], MPI[2], and HTCondor[3] that can be used as building blocks to achieve similar results, however, those frameworks are not appropriate for all applications; this book does not present development approaches using such frameworks. There are any number of good reasons to choose those for an application, similarly there are any number of good reasons not to choose them. The techniques presented in this book may be considered at a lower level than those because much of the infrastructure is built from the ground up, but also simply a *different* approach, with its own set of pros and cons, which are discussed throughout.

Existing applications benefit greatly from the approach presented in this book. It is likely easier to modify an application to incorporate the computing framework presented in this book versus changing the computing model over to something like MPI. Furthermore, not all computing environments are as controlled, or dedicated, as required for something like MPI. Your application may need to be delivered to dozens or thousands of customers, only some of which have the systems expertise or environment necessary to setup an MPI framework. The techniques presented in this book require

[1]http://en.wikipedia.org/wiki/OpenMP
[2]http://en.wikipedia.org/wiki/Message_Passing_Interface
[3]http://en.wikipedia.org/wiki/HTCondor

zero configuration for single systems and trival configuration (command line parameters) for distributed applications, making such applications much more reasonable to deploy and deliver to customers.

1.1 C++11 Features

C++11 introduced a host of new language and standard library features. While C++ is never going to be considered a syntactically *simple* language, many of the new language features help to reduce some of the syntactic complexity. Additionally, the standard library has adopted a number of libraries from Boost which support the writing of cross-platform and multi-threaded applications. A full chapter is dedicated to introducing the new capabilities that are specifically relevant to the construction of scalable, distributed, and fault-tolerant systems as presented in this book.

The following language and standard library features are discussed in Chapter 2:

- `auto` Keyword
- `decltype` operator
- `nullptr` Constant
- Range-based for Loop
- Smart Pointers
- Lambdas
- Threading & Synchronization

Each of these topics is discussed in detail and presented with working code samples to demonstrate their use.

1.2 Boost C++ Libraries

Chapter 6 provides an introduction to the Boost C++ libraries, which provides the basis for the networking framework used by much of the code presented in this book. Boost.Asio is the primary library that is covered. It provides a high-level approach to interprocess communication (i.e., networking), along with some key capabilities in support of threading, thread pools, and synchronization with respect to interprocess communication.

The following topics from the Boost.Asio library are covered in Chapter 6:

- Introduction to Boost.Asio
- The `io_service`
- The `io_service` Queue
- Thread Pools on an `io_service`
- Strands
- Buffers
- Socket Connections
- Socket Communication
- Multi-threading & Sockets

1.3 Scalable Systems

In the context of this book a *scalable* system is used to describe an application that scales to fully utilize the capabilities of a single computer or device. Generally speaking *scalable* applies to all kinds of applications, whether they run on a single computer or on a large distributed computing environment. Because this book builds towards full distributed and fault-tolerant systems, it is necessary to reduce the scope of the term to mean only a single computing device.

Three chapters are devoted to the topic of scalable systems, two for distributed systems, and only a single for fault-tolerance, with one more chapter extending the fault-tolerance framework capabilities. The reason for this is that scalability is the building block upon which the other systems are built. The next major building block is the distribured framework. Once the scalable and distributed infrastructure is in place, the most complex parts of the system are complete, allowing the fault-tolerant component to come along as a modest update.

Scalable systems begin with identifying computing tasks that can be computed in parallel. Chapter 3 starts the discussion by showing how to create an application that can take small computing tasks and spread them over any number of available CPU cores on a system. Often times applications have need for associating priority with tasks, some need to be done as soon as possible, others can tolerate being delayed. Chapter 4 takes on the challenge of creating a system that provides for application level control of task priority. Finally, it is often the case that some tasks must wait for others to complete before they can be computed, causing even more complexity in the underlying task distribution logic. Chapter 5 tackles the complex subject of providing for dependencies among computing tasks.

The work done in the initial chapter on scalable computing, Chapter 3, in building a framework that performs computations through a task-based approach is the most important. Everything else is just different ways of managing the order, distribution, and collection of the results from the tasks. Priority and dependencies affect the ordering of when and how the tasks are computed, whereas distribution changes the physical device to which computing resource a task is directed, and fault-tolerance decides how to handle tasks that are in process when their computing resource fails. Everything revolves around having a task-based computing framework, with that in place, the other capabilities become quite reasonable to provide.

1.4 Distributed Systems

This book uses the term *distributed system* to mean any two or more connected computing devices, which may be homogenous (same capabilities) or heterogeneous (different capabilities). When used in combination with *scalable* to describe an application, it means an application that scales to fully utilize the capabilities of all the connected systems. Furthermore, the intention is that the application scales across the distributed environment without requiring it to be rebuilt, it dynamically adapts to fully utilize the system's resources.

1.5 Fault-Tolerant Systems

An easily overlooked component of distributed systems is *fault-tolerance*. Fault-tolerance indicates an ability for an application to recover gracefully from a failure or error condition. The kind of fault-tolerance presented in this book is with respect to network and computing device failures. The goal is to build a system that continues correct operation in the face of an unreliable network or computing environment. Solving for one of these errors, solves both problems. As far as an application is concerned, the failure of either a network connection to a computing device, or the failure of the computing device itself looks the same and has the same result, the computing resource is no longer available. Therefore, a solution to one, is a solution to both.

Fault-tolerance only means *tolerant*, not *immune*, to failures. There are any number of different failure scenarios an application may face, both internal and external. The degree to which fault-tolerance is supported depends upon the application need, the complexity of providing that support, and ultimately the cost. A speadsheet has different fault-tolerant requirements versus a multi-player game versus the software running an inter-planetary probe, with each having different risks and costs.

A further benefit of building this kind of fault-tolerance is the ability for the application to dynamically scale, up or down, as computing resources come and go. Perhaps your application utilizes unused computing resources when employees leave for the day. By allowing for fault-tolerance an application can dynamically increase resource utilization when a computing resource becomes available, and similarly dynamically decrease resource utilization as appropriate, all while still providing correct operation.

One goal for this book is to show that it is relatively straightforward to add a fairly robust fault-tolerant capability to an application, when the proper underlying framework is provided. By building upon the task-based framework developed in the scalable and distributed chapters of the book, fault-tolerance readily fits into the model and is implemented with only modest code changes.

1.6 Performance

The framework algorithms in the code presented throughout the book are written to be approachable and generalized, with performance a secondary consideration. With respect to performance, they must be evaluated in the context of overall system performance. When the applications presented in this book are profiled, over 97% of the time is spent doing computational tasks, rather than framework or synchronization. While more performance improvements may be made, they provide no meaningful impact with respect to overall system performance.

1.7 Source Code

A large number of working demonstration programs are provided as part of this book. To a great extent, the chapter discussions are oriented around explaining sections of code from these demonstration programs. In particular, the core chapters on scalability, distributed systems, and fault-tolerant systems, are based around different implementations of an interactive Mandelbrot visualization application. The full source code for each revision is provided as part of this book, which includes each different building block framework. The frameworks from these programs can be used as the basis for developing a new application of your own, or for integration into an existing system.

The code presented within the pages of the book sometimes differs from that found in the downloaded source code. In order to fit code within the pages of the book, the formatting is often changed from what I do in a code editor; it is not how I prefer to format the code, but it is the limitation of the book format. Sometimes the code snippits are changed slightly to eliminate statements that create unnecessary noise as part of the discussion. Almost all comments are removed from the source listings, while the downloaded code is filled with comments. Finally, I expect to continue to make small improvements to the code after publication of the book.

1.8 Looking Forward

This book is only the first step in the evolution of an even more sophisticated system I am working towards. The following are topics that I intend to tackle in revisions to this book and another book that follows on from this one:

- Finish separation of framework from application code
- Google Protocol Buffers

- Authentication and encryption
- Reporting of task status for better fault detection
- Task cancellation
- Overcome single point of failure

The first change I'd like to make is to fully separate underlying framework from the application code. Currently there are bits of the framework in the application code, such as the setup of the networking and handling of the result messages. This needs to get cleaned up and fully separated from the application code.

The next change is to move away from the custom, ad-hoc, message encoding scheme and replace it with Google Protocol Buffers[4]. The technique presented in this revision of the book is well and good, but has some compromises. Part of the purpose in replacing it is that it will force me to create a better separation between task and messaging code. It also provides the benefit of a significantly more robust message definition scheme, one that can stand the test of time as an application grows.

An enhancement that goes along with replacing the messaging scheme is to introduce authentication and encryption. I have a long term eye towards the ability for this framework to be utilized over the Internet. Because of this, it makes sense to authenticate and encrypt all networking traffic to help prevent hacking.

Another area that needs to be addressed is a robust way to handle knowing when a server has failed and a task (or tasks) should be distributed elsewhere. The current mechanism is to define a timeout, which is difficult to do at coding time, given that different computers will execute the task at different rates. Therefore a new task computation status reporting scheme is required, one that has compute servers report at expected intervals the status of a task execution. This can then form the basis for better fault detection and recovery.

Related to task status and fault detection is the ability to cancel a task, or group of tasks, after having been sent for computation. There are several reasons for doing this. The first is that the operator of an application may decide some long-running task (or tasks) are no longer needed, and they should be immediately cancelled, making the resources they are currently consuming available for other tasks. Another reason is to allow for a more rapid shutdown. The current system requires that any tasks currently running at a compute server must complete before it will finish responding to a termination signal.

One of the biggest issues to overcome is the single point of failure that exists with having only one client. In the design presented throughout this book, if the client fails, the entire system goes down. My eventual design removes the problem of having a single client, allowing for any number of clients, and allowing for any (or all) of them to fail, but leaving the server framework intact and available as new clients connect.

1.9 Acknowledgements

I'd like to thank the following for helping with comments that have aided in improving the text: Chad Knight, Seth Humphries, Matt Burnham, Chris Heaps, and Scott Cannon.

[4]https://github.com/google/protobuf

2 Relevant C++11 Features

C++11 was adopted in August of 2011 and introduces a host of new language and standard library features. The language was enhanced with several features that simpify the way code is written, making it possible to write more concise code, at the same time, making it more understandable. With respect to new language features, of particular note are the *auto* keyword, the range-based for loop, and lambdas. The standard library was significantly enhanced by adopting a number of libraries from Boost (http://www.boost.org), along with adding a few new pieces. Specifically the threading and synchronization libraries enable a single C++ code-base that compiles and executes on a wide number of platforms. This chapter discusses the C++ features introduced with C++11 that are used throughout this book; it is not an exhaustive review of all new C++11 language and standard library features.

2.1 `auto` Keyword

The `auto` keyword is used to instruct the compiler to infer the type for a variable (or constant if you like). Even though the developer does not declare the type, the type is still known, by inference, to the compiler and while the program is compiled; it is not a dynamic or variant type.

A trivial usage of `auto` is for primitive types, such as `int`s and `float`s. Examples of these are shown in Listing 2.1.

Listing 2.1: Trivial `auto`

```
auto intValue = 10;
auto floatValue = 3.14f;
```

Another case for using the `auto` keyword is when using iterators. It is generally cumbersome and mistake-prone to write out the full type for an iterator, the `auto` keyword eases this burden. Inferring the type of an iterator is shown in Listing 2.2.

Listing 2.2: Inferred Iterator

```
std::unordered_map<std::string, std::uint_32> cityPopulation;
... // cityPopulation is initialized here

auto myHome = cityPopulation.find("My Home");
```

Without the `auto` keyword, the `.find` statment would have to look like Listing 2.3.

Listing 2.3: Iterator Type

```
std::unordered_map<std::string, std::uint32_t>::iterator
  myHome = cityPopulation.find("My Home");
```

There is no universal guiding principle for when to use `auto` versus writing the type (e.g., `uint16_t`); although Scott Meyers might disagree. You'll have to develop your own preference for the most appropriate usage; the code presented in the book demonstrates my preference at the time of writing.

2.2 `decltype` Operator

The `decltype` keyword is an operator that extracts the type of a variable or expression. This operator instructs the compiler to take the type of the variable or expression and use it as the type for the specified identifier. Listing 2.4 shows an example of a trival usage of `decltype`.

Listing 2.4: Trivial `decltype`

```
int source = 10;
decltype(source) scaled = source * 4;
```

The sample code provided in Listing 2.5 shows its utility with template types. The class `EightBitArray` is templated on the data type of an array that has a fixed size off 256 entries (8 bits). The variable `source` requires a type declaration, whereas `destination`'s type is based upon the type of the variable `source`.

Listing 2.5: Templates & `decltype`

```
template <typename T>
class EightBitArray
{
public:
  T& operator[](uint8_t index) { return m_array[index]; }

private:
  std::array<T, 256> m_array;
};

int main()
{
  EightBitArray<double> source;
  decltype(source) destination;

  ... Do something interesting with the arrays ...

  return 0;
}
```

Without the context of a larger application it is difficult to demonstrate `decltype`'s effective use, but there are good uses that help clean up some otherwise clumsy syntax. The way that I've found to use `decltype` is to ensure a set of related types are all the same. The initial type is specified, then all other variables, class members, or parameters are defined using `decltype`. In this way, when the original type is changed, all the other types are guaranteed to change at the same time, without having to go through all the code, potentially making a mistake.

2.3 `nullptr` Constant

C++11 introduces a `nullptr` constant that is intended for use with pointer types, versus the old C and C++ techniques of `NULL` and `0`. Having the `nullptr` constant prevents ambiguity with the integral values of `NULL` and `0`. Its use is quite simple, as demonstrated in Listing 2.6.

Listing 2.6: nullptr Example

```
uint8_t* myValue = nullptr;

if (myValue == nullptr)
{
  myValue = new uint8_t(6);
}

delete myValue;
```

It is important to note that I **do not** recommend allocation of raw pointers, apart from a few exceptional cases. Section 2.5 discusses the use of smart pointers, the preferred approach to C++ memory management. All of the code examples in this book make use of smart pointers.

2.4 Range-based for Loop

The range-based for loop simplifies the syntax in expressing a for loop, in addition to preventing issues such as *off by one* errors in counted loops. The concise syntax improves readability without imposing any penalties. The new `for` loop can be used with C/C++ style arrays, and any type that has iterators accessed through `.begin()` and `.end()` methods; the standard library containers all work with this form of the `for` loop.

The general form of the range-based for loop is shown in Listing 2.7.

Listing 2.7: Ranged For Loop Expression

```
for (type name : expression)
  statement;
```

The `type` is the type of the element returned by the expression, which can be inferred through the use of the `auto` keyword. The `name` is the identifier by which the value is called and used within the loop. The `expression` is any valid sequence, most commonly C/C++ arrays and standard library containers.

Consider the pre-C++11 code shown in Listing 2.8 that uses the standard library and iterators.

Listing 2.8: Iterated For Loop

```
uint32_t myAccumulate(const std::vector<uint8_t>& source)
{
  uint32_t total = 0;
  for (std::vector<uint8_t>::const_iterator itr =
      source.begin();
      itr != source.end();
      ++itr)
  {
    total += *itr;
  }
  return total;
}
```

In this code the iterator must be fully specified, end of sequence test written, manually update the iterator, and finally, dereference the iterator to get at the value.

The code in Listing 2.9 shows the same function, but this time using the `auto` keyword to infer the type of the iterator.

Listing 2.9: Iterated For Loop With Inferred Iterator

```
uint32_t myAccumulate(const std::vector<uint8_t>& source)
{
  uint32_t total = 0;
  for (auto itr = source.begin(); itr != source.end(); ++itr)
  {
    total += *itr;
  }
  return total;
}
```

This code is reasonably simplified only by using the `auto` keyword to infer the type. However, the end of sequence test, manual update of the iterator, and dereferencing of the iterator are all still necessary; leaving room to accidentally make a mistake, in addition to the visual complexity overhead.

The code in Listing 2.10 demonstrates the use of the ranged-based for loop in combination with the `auto` keyword.

Listing 2.10: Ranged For Loop

```
uint32_t myAccumulate(const std::vector<uint8_t>& source)
{
  uint32_t total = 0;
  for (auto value : source)
  {
    total += value;
  }
  return total;
}
```

The first thing to notice is how concise the code is and syntactically simplified from the first example. The next thing to notice is that instead of working with an iterator, `value` is the value! Our mind is not cluttered with thinking about iterators, instead we are now more focused on solving the problem, summing integers from a vector.

The `auto` keyword can be decorated with a reference, among others (e.g., `const`), which allows the value to be modified. This is demonstrated in Listing 2.11.

Listing 2.11: Ranged For With Reference Type

```
void myScale(std::vector<uint8_t>& source, uint32_t scalar)
{
  for (auto& value : source)
  {
      value *= scalar;
  }
}
```

This code tells the compiler to infer the type, but also to make sure the inferred type is a reference. By doing this, the `value` is allowed to be modified, which obviates the need to create a counted for loop to index the vector and update the value in the old way.

2.5 Smart Pointers

Smart pointers are not a new C++11 standard library feature, but are unfamiliar to a large enough audience that it is important to review as part of this chapter. While they are not new, there is one new addition to the family, `std::unique_ptr`, improvements to both `std::shared_ptr` and `std::weak_ptr`, with `std::auto_ptr` now deprecated. This section discusses the concept of smart pointers and provides specific discussion of the `std::shared_ptr`. The other smart pointers types are useful, but are not essential to the core content of this book.

2.5.1 Automatic Resource Management

In simple terms, a smart pointer is a class that provides a wrapper around a raw pointer. The class provides automatic memory management of the underlying raw pointer, while also providing a natural C++ syntactical interface for working with the pointer. Interestingly, smart pointers can be used to manage other kinds of resources, such as file handles, database connections, or network connections; these are not covered or demonstrated in this book.

The foremost problem solved by shared pointers is to prevent most situations of *memory leaks*. This is possible through the use of the concept of *reference counting* combined with the shared pointer being a class and having a constructor and destructor. A shared pointer contains a reference count that indicates how many other shared pointers refer to the same raw pointer. In the constructor the reference count is incremented and in the destructor it is decremented. During the destructor, if the reference count goes to 0, the raw pointer is deleted by the shared pointer, thereby automatically freeing the memory.

C++ smart pointers are implemented through the use of templates, allowing any type to be contained. Consider the code in Listing 2.12 that demonstrates a couple uses of a `std::shared_ptr`.

Listing 2.12: Smart Pointer Introduction

```
{
  std::shared_ptr<uint32_t> myValue(new uint32_t(6));
  std::cout << *myValue << std::endl;
}
...
{
  auto myValue = std::make_shared<uint32_t>(6);
  std::cout << *myValue << std::endl;
}
```

There are several things of note in Listing 2.12. The first is because the shared pointer is a template, the type must be supplied; in this example an `uint32_t`. The second is the two different ways the `std::shared_ptr` is allocated. The first code segment uses the `new` keyword to allocate the memory, whereas the second uses `std::make_shared`. Both compile and execute correctly, why the difference and does it make any difference? The difference is in memory allocation efficiency. In the first example, a memory allocation is made for the value, and another is made for the underlying reference count memory; two different memory allocations. In the second example a single allocation is used to allocate memory for the value and the underlying reference count memory. The second example makes use of the `auto` keyword to deduce the shared pointer type, which is not possible with the first example. The final item of note in this code is the lack of any code that deallocates the memory. Instead of having to write the code that does this, it happens automatically when `myValue` goes out of scope. As it goes out of scope the reference count is decremented to 0 in the destructor and because it is 0, the raw pointer is deallocated; automatic memory management.

2.5.2 Copying Shared Pointers

Shared pointers can be copied, as they are copied, the reference count is incremented, allowing multiple shared pointers to refer to the same raw pointer, and when the last one goes out of scope the raw pointer is deleted. Consider the code in Listing 2.13.

Listing 2.13: Shared Pointer Copy

```
{
  std::shared_ptr<uint32_t> outer = nullptr;
  {
    std::shared_ptr<uint32_t> inner = std::make_shared<uint32_t>(6);
    std::cout << *inner << std::endl;
    outer = inner;
  }
  std::cout << *outer << std::endl;
}
```

The first shared pointer, `outer` is not initialized, it is a valid `std::shared_ptr` that doesn't point to anything. Below its declaration is a small scoped section of code where `inner` is initialized to refer to a value. The value is sent to `std::out`, and then `inner` is copied to `outer`; at this moment, the reference count to the raw pointer is incremented to 2, because two different `std::shared_ptr`s refer to it. As `inner` goes out of scope, the reference count to the raw pointer is decremented from 2 to 1, the raw pointer is not deallocated. When `outer` goes out of scope, the reference count is decremented to 0 and the raw pointer is then deallocated.

2.5.3 Preventing The Copy Penalty

There is a small performance penalty every time a `shared_ptr` is copied. One of the most common times this occurs is when one is passed as a parameter to a function. Unless otherwise specified, C++ makes copies of function parameters (pass by value), `shared_ptr`s are no different. To avoid the cost associated with making a copy of a parameter, it can be marked as a reference type. Additionally, if the value of the parameter (the `std::shared_ptr`) does not need to change, it is read only, it can be marked as `const`. The function in Listing 2.14 illustrates passing a `shared_ptr` by const reference.

Listing 2.14: Const Shared Pointer Reference

```
void reportValue(const std::shared_ptr<uint32_t>& value)
{
  std::cout << "The value is: " << *value << std::endl;
}
```

This approach avoids the cost of making a copy of the `shared_ptr`, prevents an accidental coding error that might change the value of the pointer, while still providing the benefit of having a `smart_ptr`.

While the previous use of `const` combined with making the parameter a reference type avoids the cost of copying the `shared_ptr` along with preventing the pointer from being changed, it does not prevent the value being pointed to from being changed; which might be desired in many cases. In order to prevent the value that is being pointed to from being changed, the type being pointed to needs to be identified as constant. The code segment in Listing 2.15 demonstrates how to do this.

Listing 2.15: Fully Const Shared Pointer

```
void reportValue(const std::shared_ptr<uint32_t const>& value)
{
  std::cout << "The value is: " << value << std::endl;
}
```

This code adds a `const` decorator to the `uint32_t`, which now tells the compiler to not allow any code that attempts to change the value.

2.6 Function Polymorphism

The ability to create and manage pointers to functions has been a part of C++ since before C++, it was possible back in the original C language. However, the syntax to use them is difficult to read, and the ability to create and effectively use pointers to class members is even more difficult. In part, to overcome this burdensome syntax, C++11 has adopted the Boost `function` and `bind` libraries, making them a native part of the language. Additionally, as you'll see in Section 2.7, there are other compelling reasons for adopting those libraries. In this section of the chapter the use of `std::function` is reviewed.

Consider the polymorphic code in Listing 2.16. What makes the code polymorphic is that it takes as its first parameter a `std::function`. A `std::function` is simply a function wrapper, a wrapper that can refer (or point) to any so-called *callable* expression, a regular C++ function is *callable*.

Listing 2.16: Polymorphic Function

```
void report(
    std::function<std::string(std::string)> update,
    std::string message)
{
    std::cout << "Your message: " << update(message) << std::endl;
}
```

The expected signature of the function parameter in Listing 2.16 is one that returns a `std::string` and accepts a `std::string` parameter. The type that follows the `std::function<` is the return type, and the type inside the parenthesis is the type of the parameter the function accepts. As usual, the name of the function, `update`, follows the `std::function` type itself. In the case of this code, the `std::function` accepts only a single parameter, but it could be any number of parameters.

In order to demonstrate function level polymorphism, we'll use the two functions in Listing 2.17 and 2.18, along with the method in Listing 2.19. The two functions and the method will be passed as parameters into the `report` function from Listing 2.16.

Listing 2.17: Polymorphic Function

```
std::string leetE(std::string message)
{
  std::replace(message.begin(), message.end(), 'e', '3');
  std::replace(message.begin(), message.end(), 'E', '3');
  return message;
}
```

Listing 2.18: Polymorphic Function

```
std::string leetT(std::string message)
{
  std::replace(message.begin(), message.end(), 't', '7');
  std::replace(message.begin(), message.end(), 'T', '7');
  return message;
}
```

Listing 2.19: Polymorphic Function

```
class Replace
{
public:
  std::string leetL(std::string message)
  {
    std::replace(message.begin(), message.end(), 'l', '1');
    std::replace(message.begin(), message.end(), 'L', '1');
    return message;
  }
};
```

In C++, the name of a function is also a pointer to that function. Therefore, for the first two functions, it is as simple as using the function name as the first parameter to the `report` function. The code shown in Listing 2.20 demonstrates doing this.

Listing 2.20: Polymorphic Function

```
std::string myMessage = "This is a leet test";
report(leetE, myMessage);
report(leetT, myMessage);
```

2.7 Lambdas

Lambda functions or expressions are one of the most interesting, and welcome, additions to the C++ language. The terms *lambda function* and *lambda expression* are considered interchangable; this book uses the term *lambda function* or simply *lambda*. A lambda function is the ability to define a function object that has no name, a so-called anonymous function object. This section introduces lambdas to a level that will make reading the code samples provided with this book comfortable to understand. This section provides a solid foundation from which to build a more sophisticated understanding; a comprehensive discussion of lambdas and their full exploitation is beyond the scope of this book.

The best place to start is with a simple lambda function, as shown in Listing 2.21.

Listing 2.21: Simple Lambda

```
void simpleLambda()
{
  auto myLambda =
    [] ()
    {
      std::cout << "My first lambda!" << std::endl;
    };
  myLambda();
}
```

The first thing you'll likely notice in Listing 2.21 is the unusual syntax[1], which is detailed in Section 2.7.2. Also notice the use of the `auto` keyword to have the compiler infer the type. The signature is known, but the type is a little more subtle and difficult to express, therefore the `auto` essentially becomes a necessity, not only a convenience. For now, it is enough to know that the variable `myLambda` is now a function object that can be invoked using the normal function syntax. When executed, the code in this example will output '`My first lambda!`' to the console. Clearly this example does not illustrate the benefits of lambdas, that is developed through the remainder of this section and chapter.

2.7.1 Functor Review

Before diving into the syntax and use of lambdas, it is worthwhile to review *functors*, also known as function objects. Functors were possible well before C++11, making functors a good place to start.

Functors make use of the ability to overload the `()` operator, which provides another convenient method for dynamic, or run-time, binding of code. A common use of functors is in combination with standard library containers. The `std::transform` algorithm takes a source iterator range, a

[1]The formatting of this lambda is to make it fit nicely within the page margin. Often a simple lambda like this will be placed on a single line.

destination start iterator, and *transforms* the objects in that range by the function it is given. The code in Listing 2.22 demonstrates its use.

Listing 2.22: Functor Example

```
uint32_t initialize(uint32_t value)
{
  static uint32_t nextValue = 1;
  return nextValue++;
}

class Scale
{
public:
  Scale(uint8_t factor) : m_factor(factor) {}

  uint32_t operator()(uint32_t& value) const
  {
    return value * m_factor;
  }

private:
  uint8_t m_factor;
};

void functorExample()
{
  std::vector<uint32_t> myVector(10);

  std::transform(
    myVector.begin(), myVector.end(), myVector.begin(), initialize);
  std::transform(
    myVector.begin(), myVector.end(), myVector.begin(), Scale(4));
}
```

The first `std::transform` in the `functorExample` function utilizes the `initialize` function to assign a set of increasing values to the vector. In this case, the clear use of a function is being utilized.

The second `std::transform` creates an instance of the `Scale` class as the last parameter, passing a `4` as the constructor value, which becomes the scaling factor for the `Scale` instance. During the execution of the `std::transform` algorithm, it calls the `()` operator on the `Scale` instance, which accepts the original value from the vector, scales it by `m_factor` and returns the result, with the result being stored back into the vector. The `Scale` class in this example is a *functor*.

With the advent of lambdas in C++11, they can be used in place of a functor, while providing some additional capabilities not easily handled by functors.

2.7.2 Lambda Syntax

The basic lambda syntax is shown in Listing 2.23.

Listing 2.23: Lambda Syntax

```
[ capture ] ( parameters ) mutable-specification exception-specification
  -><return-type>
  { body }
```

There are six sections that compose a lambda function: capture, parameters, mutable specification, exception specification, return type, and the body. Of these, only the *capture* and the *body* are required, the others are optional. Each of these is briefly described below.

capture Used to indicate which variables declared outside the scope of the lambda are accessible within the lambda. Because this is a lengthy topic, it is more fully explored in Section 2.7.3.

parameters Lambdas are used in the context of some code calling into them, passing parameters into the lambda, just like passing parameters into a function. This section identifies the parameter types and names.

mutable-specification The `mutable` keyword can be used here to inform the lambda to call non-const member functions of captured (by copy) parameters.

exception-specification A `throw()` keyword can be used here to indicate the lambda does not throw an exception. If this specification is used and the code within the lambda has code that generates an exception, a compiler warning results.

return-type The return type for the lambda, in the same sense that a function returns a type. Generally this is not necessary as the compiler is able to infer the return type. If there is more than one return statement within the lambda, the return type becomes necessary [2].

body This is where the statements that perform the desired function are placed.

With knowledge of the full lambda syntax, the previous functor example can be re-written using Lambdas. This is demonstrated in Listing 2.24.

Listing 2.24: Lambda Example

```
void lambdaExample()
{
  std::vector<uint32_t> myVector(10);

  std::transform(myVector.begin(), myVector.end(), myVector.begin(),
    [](uint32_t value)
    {
      static uint32_t nextValue = 1;
      return nextValue++;
    });

  std::transform(myVector.begin(), myVector.end(), myVector.begin(),
    [](uint32_t value)
    {
      return value * 4;
    });
}
```

Through the use of lambdas we are able to create the same functionality with a much more concise piece of code. The development effort is reduced by not having to write free functions or functor classes that provide the needed functionality, instead, two short lambda functions are coded in-place.

2.7.3 Lambda Capture

A lambda is an object that captures external state at the time it is instantiated. Looking back at the function example in Section 2.7.1 we see the functor accepted a `scale` parameter as part of its constructor. The capture clause of a lambda is very much like the constructor parameters of a functor, the difference being a much simpler syntax. Consider each of the following capture examples:

[2] I recommend following the concept of having a single point of return for all code where reasonable.

`[]` Nothing is captured.

`[=]` Capture all used variables by value (copy).

`[&]` Capture all used variables by reference.

`[data]` Capture the variable `data` by value.

`[&data]` Capture the variable `data` by reference.

`[=, &data]` Capture all used variables by value, except for `data` which is captured by reference.

`[&, data]` Capture all used variables by reference, except for `data` which is captured by value.

Because a lambda is an object with a lifetime that may exist longer than the scope in which it was declared, some caution is necessary in their use. For example, a lambda may be returned from a function, with the lambda having attempted to capture a local function variable by reference. Later on, when the lambda function is invoked and it attempts to access the reference capture, it will either use an invalid value or cause an exception because the scope for the referenced local variable no longer exists. This is illustrated in Listing 2.25.

Listing 2.25: Lambda Capture

```
std::function<uint32_t ()> makeLambda(uint32_t value)
{
  uint32_t local = value;
  return [&] () { return local; };
}

void badLambdaCapture()
{
  auto badLambda = makeLambda(8);
  uint32_t result = badLambda();
  std::cout << result << std::endl;
}
```

The lambda function captures the local variable `local` by reference, with lambda then returned from the function. When this code is executed some large number gets displayed, on my computer `3435973836`, rather than seeing the value of `8` written to the console. The program did not throw an exception, it returned a value stored at a no longer valid memory location. Because it didn't throw an exception, you can imagine the potential for a subtle bug to be introduced. If the capture had been by value, `return [=] () { return local; };`, the problem would not have happened.

Notice that it is possible to define a `std::function` type that matches the lambda signature as the `makeLambda` function return type. In this example, the lambda accepts no parameters and has a return type of `uint32_t`, implied in the lambda function. Given these two pieces, a `std::function<uint32_t ()>` is defined as the return type for the function. While the type of a lambda is considered *ineffible* (too great to express), it is possible to use `std::function` as a means for expressing the type of a lambda.

2.8 Threading & Synchronization

Threading and synchronization are one of the biggest features added to the C++11 language and standard library, also extremely welcome. These capabilities come from the Boost libraries with only a few modest changes; if you are familiar with Boost threading and synchronizaton, you know 90% of the C++11 features. This section covers creating threads, joining threads, several related thread utilities, and the use of mutexes for synchronization.

2.8.1 Thread Creation

Let's start with a simple example of creating a thread as shown in Listing 2.26.

Listing 2.26: Simple Thread

```
void simpleThread()
{
  auto report = []()
  {
    std::cout << "This executed from thread: ";
    std::cout << std::this_thread::get_id();
    std::cout << std::endl;
  };

  std::thread myThread(report);
  myThread.join();

  report();
}
```

This example starts by defining a lambda that reports the id of the thread from which it executed. The lambda uses `std::this_thread::get_id()` to obtain the id of the thread. Next, a thread is created with the `std::thread myThread(report);` statement. `myThread` is the name of the thread instance. A `std::thread` accepts a function as its constructor parameter, this function is called as soon as the thread is created, the thread is terminated when the function exits. The `myThread.join();` statement causes the main application thread to wait for the termination of the thread referenced by `myThread`. Finally, to demonstrate there really are two different threads, the `report` lambda is called by the main application thread to report its id. A run of this function will look like Figure 2.27.

Figure 2.27: Simple Thread Output

```
This executed from thread: 3552
This executed from thread: 9468
```

2.8.2 Thread Utilities

There are several thread utilities that are useful to know. The previous section showed `std::this_thread::get_id()`, there are also utilities to sleep, discover how many CPU cores are on the system, and perform timing. Some of these are directly part of the `<thread>` library, others are supported through the `<chrono>` library.

Thread Sleep

`std::thread` provides a `sleep_for` function that accepts a duration. The duration is a `std::chrono::duration` type. At a simple level, `std::chrono::duration` can be defined in terms of hours, minutes, seconds, milliseconds, or nanoseconds. It is also possible to define which kind of system clock is used to measure the time, for most purposes the default clock is sufficient. The code in Listing 2.28 demonstrates how to use `sleep_for`, measuring the time to sleep in millisecond using a `std::chrono::duration`.

Listing 2.28: Thread Sleep

```
void sleepingThread()
{
  std::thread myThread(
    []()
    {
      std::cout << "Going to sleep for two seconds...";
      std::this_thread::sleep_for(std::chrono::milliseconds(2000));
      std::cout << "awakened" << std::endl;
    });

  myThread.join();
}
```

When executed, the `Going to sleep for 2 seconds...` message is displayed, then two seconds later, `awakened` appears.

CPU Cores

When developing scalable applications it is critical to be able to determine at runtime the number of available CPU cores on a system. This capability is now provided by `std::thread` through the `hardware_concurrency` function. This is a static function that returns the number of CPU cores, the total of all system cores, including hyper-threaded or logical cores. The documentation for this function notes this is only an estimate and may not be able to be determined. In the case it can not be determined, a 0 is returned. The code in Listing 2.29 demonstrates how to use the `hardware_concurrency` function.

Listing 2.29: Number of CPU Cores

```
void howManyCores()
{
  auto coreCount = std::thread::hardware_concurrency();

  std::cout << "This system has " << coreCount;
  std::cout << " CPU cores" << std::endl;
}
```

When executed on my computer this code segment reported, `This system has 12 CPU cores`. The computer on which this was executed has 6 CPU cores, each of which is also hyper-threaded. There isn't a way using the C++11 standard library to determine which of the 12 are the hyper-threaded cores, or even if any of them are hyper-threaded. For the purposes of this book, it doesn't matter, the scalability techniques work well with asymmetric CPU capabilities, and that is really the point of the book, *scalability*.

2.8.3 Timing

Another tool that is useful in building fault-tolerant systems is the ability to time how long something has taken. Three capabilities are necessary to support this. The ability to capture the current system time, a data type to represent the time, and the ability to represent the difference between two times. These are all provided by the `<chrono>` library. The `system_clock` allows the current system time to be taken, the `time_point` class represents a point in time, and the `duration` class represents the difference between two points in time.

In order to take a snapshot of the current system time the `now` function is called from the `system_clock`. The type returned from a call to `now` is `time_point`. A call to capture the system time and store the result looks like Listing 2.30.

Listing 2.30: Time Point

```
std::chrono::system_clock::time_point now = std::chrono::system_clock::now();
```

Alternatively, using `auto` looks like Listing 2.31.

Listing 2.31: Time Point Inferred

```
auto now = std::chrono::system_clock::now();
```

The `time_point` class overloads the minus operator, returning a `duration` type. The `duration` type takes a little more work to use, but is simplified through the use of some pre-defined helper types. The `duration` class is templated on two types. The first is a signed integral type used to store the duration *tick count*, and the second is a *period* that represents the number of seconds between ticks. At first glance this sounds confusing, but begins to make sense after only a few uses.

The definition of a duration looks like Listing 2.32.

Listing 2.32: Duration Type

```
template<class S, T>
class duration;
```

Where `S` is the storage type, and `T` is the number of seconds between ticks. Listing 2.33 shows an example of how to define a duration that stores milliseconds.

Listing 2.33: Duration Example

```
std::chrono::duration<int64_t, std::ratio<1, 1000>> millisecs;
```

In order to get the number of seconds between ticks `std::ratio<1, 1000>` is used, rather than a floating point type, this allows exact integral steps to be defined. This can be further simplified by using one of the pre-defined `std::ratio` types, for this example, `std::milli`. The `<ratio>` header file defines a number of commonly used ratio types. The following is a list of some of these helper types:

- `std::chrono::nanoseconds`
- `std::chrono::microseconds`
- `std::chrono::milliseconds`
- `std::chrono::seconds`
- `std::chrono::minutes`
- `std::chrono::hours`

Putting all of the pieces from this chapter together, Listing 2.34 shows how to time the computation of a series of *Fibonacci* numbers.

Listing 2.34: Timed Fibonacci

```
void timeFibonacci()
{
  std::function<uint32_t (uint16_t )> fib;
  fib = [&fib](uint16_t n)->uint32_t
  {
    if (n == 0) return 1;
    if (n == 1) return 1;

    return fib(n - 1) + fib(n - 2);
  };

  auto start = std::chrono::system_clock::now();
  for (auto n: IRange<uint16_t>(0, 10))
  {
    std::cout << "Fibonacci of " << n << " is ";
    std::cout << fib(n) << std::endl;
  }

  auto end = std::chrono::system_clock::now();

  std::chrono::nanoseconds difference = end - start;

  std::cout << "Time to compute is: " << difference.count();
  std::cout << " nanoseconds" << std::endl;
}
```

The function starts by defining a lambda function that is stored into the `fib` variable. The reason for declaring and then assigning is that the compiler won't allow `fib` to be captured in the same statement as it is declared. Pay careful attention to the way the capture of the `fib` function is captured into the lambda and used. The `fib` function is captured and then called inside the lambda into which it is captured, we are recursively calling into a lambda! The next step is to take a snapshot of the system clock, before entering the loop that computes some fibonacci numbers. The `IRange` class creates an iterated range over which the range-based for loop can operate[3]. Once the loop completes another snapshot of the current time is taken, the difference is computed, and finally the duration is reported to the console.

It is relevant to point out that the `system_clock` is not really capable of measuring individual nanoseconds, but is good enough for general use. The most accurate clock provided by `std::chrono` is the `high_resolution_clock`, but note that it is often only an alias for the `system_clock`.

2.8.4 Mutexes

Mutexes are one tool used to synchronize operations among threads. C++11 includes a new library, `<mutex>`. This library comes directly from Boost and if you are familiar with its usage from Boost, you already know everything you need. There are two parts to effective use of C++11 mutexes. The first is the `mutex` class that represents the mutex, the second is the `lock_guard` wrapper which provides RAII[4] style usage; each of these are detailed next.

The `mutex` class is what you expect in a mutex. It provides three member functions for locking and unlocking: `.lock`, `.try_lock`, and `.unlock`.

`.lock` If the mutex is available the lock is taken, otherwise it blocks until the lock becomes available.

`.try_lock` If the mutex is available the lock is taken and returns `true`, otherwise it returns `false`.

`.unlock` Unlocks the mutex.

The code in Listing 2.35 shows a sample use of locking and unlocking of a mutex.

Listing 2.35: Simple Mutex

```
void synchronize(std::mutex& mutex)
{
  myMutex.lock();
  //
  // Perform some operation that requires synchronization
  //
  myMutex.unlock();
}
```

[3]The code for `IRange` is described in Appendix A
[4]http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization

Instead of using explicit locking and unlocking I strongly recommend the use of `std::lock_guard` which, as noted earlier, provides RAII style mutex usage. When a `std::lock_guard` object is created it takes ownership of the mutex and releases it upon destruction. In other words, on its constructor `std::lock_guard` calls `.lock` and its destructor calls `.unlock`. A rewrite of Listing 2.35 to use `std::lock_guard` is shown in Listing 2.38.

Listing 2.36: `lock_guard` Mutex

```
void synchronize(std::mutex& mutex)
{
  std::lock_guard<std::mutex> lock(myMutex);
  //
  // Perform some operation that requires synchronization
  //
}
```

The code in Listing 2.38 is a little bit simpler, but that is not the primary reason for this approach. The reason it is more robust is that the lock isn't released until the `std::lock_guard` object goes out of scope. Consider the code shown in Listing 2.37.

Listing 2.37: `lock_guard` Mutex

```
uint32_t synchronize(std::mutex& mutex, uint32_t& param)
{
  std::lock_guard<std::mutex> lock(myMutex);
  param++;

  return param;
}
```

In this example the function receives a parameter by reference, increments its value and uses it as a return value. Writing this same code using explicit locking and unlocking is more cumbersome and potentially error prone. The RAII approach is correct, more elegant, and easier to read.

A complete example of synchronizing threads to ensure correct updating of a resource is shown in Listing 2.38.

Listing 2.38: Full RAII Mutex Example

```
void demoThreadMutex()
{
  uint16_t resource(0);
  std::mutex mutex;

  auto report = [](uint16_t value)
  {
    std::cout << "The value is: " << value << std::endl;
  };

  auto reportValues =
    [&resource, &mutex, report](bool reportEvens)
    {
      while (resource < 10)
      {
        auto isEven = (resource % 2 == 0);
        if (isEven == reportEvens)
        {
          std::lock_guard<std::mutex> lock(mutex);
```

```
                resource++;
                report(resource);
            }
        }
    };

    std::thread thread1(threadFunction, true);
    std::thread thread2(threadFunction, false);

    thread1.join();
    thread2.join();
}
```

The example begins by defining a shared resource and mutex. This is followed by defining reporting and thread function lambdas. The thread lambda uses the resource, mutex, and `report` lambda to synchronize, update, and report the value of the resource. Two threads are then created, both using the same thread function/lambda, with one updating even values, the other updating odd values. Finally the two threads are joined, causing the function to only exit when both threads have completed. Not only does this code demonstrate the use of lambdas, threading, and mutexes, it also shows how concise C++11 code can be.

The `std::thread` constructor takes as the first parameter the name of a a start function (or lambda) to begin execution. The remaining constructor parameters become parameters to the start function. In this example, the `threadFunction` lamba expects a `bool` parameter.

2.8.5 Atomic Operations

In addition to support for mutexes, which can be used to create atomic operations, C++11 provides specific support for atomic operations through the `<atomic>` library. This library defines a template class `std::atomic` that can be used on any type that is `TriviallyCopyable`[5]. Atomic operations are already defined for primitive C++ types, such as `bool`, `uint8_t`, `uint16_t`, and so on. The use of `std::atomic` on custom types is beyond the scope of this book, the focus here is on integral types.

There are a large number of operations that can be performed on the atomic types, of particular interest for our purposes is the ability to set, increment, decrement, and read values. The code shown in Listing 2.39 shows how to define and use an atomic type.

Listing 2.39: Simple Atomic

```
std::atomic<uint16_t> value(0);

value++;
std::cout << value << std::endl;
```

From the example in Listing 2.39 it is possible to see how easy the types are to declare and use. Other than the declaration, the type is used as expected, and guarantees operations on it are performed atomically. Instead of using the template syntax, the `<atomic>` library includes definitions for the standard integral types. The code in Listing 2.39 could have declared the type using `std::atomic_uint16_t`. My personal preference is to use the template syntax for no reason other than that is just what I prefer.

To demonstrate the use of atomics in a multi-threaded context, the example from Listing 2.38 is rewritten to use `std::atomic` for the resource instead of using mutex operations, the revised code is shown in Listing 2.40.

Listing 2.40: Complete Atomic Example

[5]http://en.cppreference.com/w/cpp/concept/TriviallyCopyable

```
void demoAtomic()
{
  std::atomic<uint16_t> resource(0);

  auto report = [](uint16_t value)
  {
    std::cout << "The value is: " << value << std::endl;
  };

  auto threadFunction =
    [&resource, report](bool reportEvens)
    {
      while (resource < 10)
      {
        auto isEven = (resource % 2 == 0);
        if (isEven == reportEvens)
        {
          resource++;
          report(resource);
        }
      }
    };

  std::thread thread1(threadFunction, true);
  std::thread thread2(threadFunction, false);

  thread1.join();
  thread2.join();
}
```

While the use of `std::atomic` cleans up the code by eliminating the need to manually use mutexes each time the resoure is accessed, that is not the most important reason for using the library. The most important reason is for performance. As with all things, context matters, but for the kind of uses presented in this book, `std::atomic` can range anywhere from three to ten times faster than `std::mutex`.

2.8.6 Condition Variables

Condition variables are used to notify, or signal, a thread or threads that some *condition* has taken place and that it is okay to continue based upon the notification. In C++11 this is done through a `std::condition_variable`. Specifically, a `std::condition_variable` blocks until it is signaled. It takes more discussion and code to explain and demonstrate the use of condition variables, therefore this section is longer than most of the others in this chapter. While it is a little longer, it is also a useful way to finish this chapter because most of the new C++11 features presented in this chapter are utilized. Take the time to read through the associated example and fully understand how condition variables work, they are a key building block for creating scalable, efficient, multi-threaded, and distributed applications.

A typical paradigm for the use of `std::condition_variable`s is in a *producer-consumer* application. One or more producers create resources that are consumed by one or more consumers; the producers place the resources into a fixed sized container. If the container is full, producers should efficiently wait until there is room in the container. Similarly, consumers remove resources from the container and should efficiently wait if the container is empty. It is through the use of `std::condition_variable`s that it is possible to have the producers and consumers efficiently wait or work based upon the state of the resource container. The way this is done is to define a *signal*, a `std::condition_variable`, that indicates when the container has room to place new resources, along with another that indicates when there are resources available to consume. The next series of code segments illustrate the definition of such a resource container in combination with a single producer and multiple consumers.

The Resource Container

The resource container is the most crucial component of the design. It provides the space where resources are placed and consumed, along with coordinating access among the various producers and consumers. The code in Listing 2.41 shows a portion of the `Resource` class declaration. It is a template class, allowing any data type to be the resource. It defines a `std::array` for storing resources, along with integers that describe attributes of the storage (it is a circular buffer). Additionally, a `std::mutex` is defined to coordinate safe multi-threaded access, along with two `std::condition_variable`s used to signal when there is room to place something into the container, and when there is a resource that can be consumed.

Listing 2.41: Container Declaration

```
template <typename T>
class Resource
{
public:
  Resource() : m_front(0), m_back(0), m_count(0) {}

  void add(T value) {...}
  T remove() {...}
...
private:
  std::array<T, 10> m_data;
  std::uint8_t m_front;
  std::uint8_t m_back;
  std::uint8_t m_count;

  std::mutex m_mutex;
  std::condition_variable m_isRoom;
  std::condition_variable m_isData;
};
```

The next piece of the resource container is the code that implements the `add` method. The `add` method must wait until there is space available to place a new resource, and do so efficiently.

Listing 2.42: `add` Method

```
void add(T value)
{
  std::unique_lock<std::mutex> lock(m_mutex);
  // Have to wait until m_data is has room
  m_isRoom.wait(lock,
    [this]()
    {
      // Verify there really is room to place
      // something...could be spurious wakeup.
      return m_count != m_data.size();
    });
  m_data[m_back] = value;
  m_back = (m_back + 1) % m_data.size();
  m_count++;
  // Signal the has data condition
  {
        std::unique_lock<std::mutex> lockNotify(m_mutex);
        m_isData.notify_one();
  }
}
```

Notice the use of a `std::unique_lock` instead of a `std::lock_guard`; this is very important! The thread obtaining a `std::unique_lock` does not have to own the mutex, ownership can be transferred. If (when) the lock is taken in the constructor and then is used with a `std::condition_variable`, the thread invoking the `std::condition_variable` takes ownership and decides whether or not the lock is necessary. If the lock isn't necessary (e.g., when the condition isn't signaled) the lock is given up, which allows another thread's `std::unique_lock` or `std::condition_variable` to take ownership; yes, this is complex stuff. This will become a little more clear when the code for the `remove` method is shown next.

Getting back to the `add` method: A lock on the mutex is taken, then a wait is performed on the `m_isRoom` condition. The `.wait` method of a `std::condition_variable` takes a predicate that indicates when the condition is true. In the case of the resource container, the condition is true as long as the count of items in the storage is not equal to the max storage capacity. Until this condition becomes true, the `add` method is blocked from continuing. Once the condition is signaled and the predicate is true, the lock on the mutex is obtained and the method continues. The last statement in the method uses the `.notify_one` method to signal the other condition, indicating there is now a resource that can be consumed. While the C++ standard doesn't appear to require the acquisition of a lock on the mutex before signaling the condition, in practice (on Linux, not on Windows), I have found it necessary to ensure correct signaling; this is most likely due the the use of the PThreads library as the underlying implementation.

Pay attention to the comment in the `add` method regarding a *suprious wakeup*. It turns out the underlying implementation of `std::condition_variable`s can result in the wait state waking up even when the condition has not been signaled (the reasons this can happen are beyond the scope of this book). There is an overloaded version of the `.wait` method that does not require a predicate, but still has the same spurious wakeup issue. The solution in that case is to use a loop around the `wait`. By using the overload that requires a predicate, the condition won't fall through until the predicate indicates it is okay to move forward; eliminating the need to write a looping structure. I recommend only the use of the predicate overload, it is much easeir to write correct code and understand later when the time comes to return to the code.

The `remove` method for the resource container is shown in Listing 2.43. This code is fundamentally similar, with respect to coding techniques, to the `add` method, except written from the perspective of removing an item from the container. The same lock, then conditional wait is used, followed by removing the next value, and then finishing by signaling the `m_isRoom` condition that notifies any pending producers new space is available.

Listing 2.43: `remove` Method

```
T remove()
{
  std::unique_lock<std::mutex> lock(m_mutex);
  // Have to wait until m_data contains something
  m_isData.wait(lock,
    [this]()
    {
      // Verify there really is something to
      // work on...could be spurious wakeup.
      return m_count > 0;
    });
  T value = m_data[m_front];
  m_front = (m_front + 1) % m_data.size();
  m_count--;
  // Signal the has room condition
  {
        std::unique_lock<std::mutex> lockNotify(m_mutex);
    m_isRoom.notify_one();
  }

  return value;
}
```

The `add` and `remove` methods take care of the heavy lifting, making writing producer and consumer code almost trivial to write.

Producers and Consumers

Producers add items to the resource container, while consumers remove items. Lambdas for producers and consumers are found in Listing 2.44.

Listing 2.44: The Producer

```
Resource<uint32_t> resources;

auto producer =
  [&resources](uint8_t howMany)
  {
    for (auto item : IRange<uint8_t>(1, howMany))
    {
      resources.add(item);
      std::this_thread::sleep_for(std::chrono::milliseconds(50));
    }
  };

auto consumer =
  [&resources](uint8_t howMany)
  {
    for (auto item : IRange<uint8_t>(1, howMany))
    {
      std::cout << "Thread: " << std::this_thread::get_id();
      std::cout << " Consumed: " << resources.remove();
      std::cout << std::endl;
      std::this_thread::sleep_for(std::chrono::milliseconds(200));
    }
  };
```

The only parts of these two lambdas that relate to the use of the resource buffer are the `resources.add(item)` and `resources.remove()` method calls. The purpose of the `sleep_for` statements is to help demonstrate a single producer that can produce faster than any one consumer, which creates the need to have more than one consumer in order to keep up with a single producer.

The final piece of this demonstration is to put it all together by creating a single producer along with multiple consumers, Listing 2.45 shows this. It is as simple as instantiating the five different threads, one producer and four consumers, and then calling join on each of them to allow all to complete their work.

Listing 2.45: The Producer

```
std::thread producer1(producer, 100);
std::thread consumer1(consumer, 25);
```

```
std::thread consumer2(consumer, 25);
std::thread consumer3(consumer, 25);
std::thread consumer4(consumer, 25);

producer1.join();
consumer1.join();
consumer2.join();
consumer3.join();
consumer4.join();
```

2.9 Summary

C++11 is a significant step forward for the language and the associated development community. The updates to the language and the standard library work together to provide two things of interest for the techniques presented in this book, concise code and better cross-platform support. Additions like the range-based for loop and lambdas work to enable more concise and readable code. The use of smart pointers relieves a great burden from the developer by providing automatic lifetime management of dynamically allocated objects. The adoption of threading and synchronization capabilities are a huge benefit for writing cross-platform code. The benefits of these new additions are well demonstrated throughout the remainder of this book.

3 | Scalability - Task-Based

This chapter begins the discussion of designing and constructing scalable, distributed, and fault-tolerant applications. The first concept is that of taking a global application view of thread management. This is followed by introducing the computational task building block, and then the concept and role of a Thread Pool. The background discussion is then followed up with a detailed walkthrough of the implementation of the concepts along with an application that utilizes these building blocks.

This chapter introduces the demonstration application used throughout the remainder of the book to help illustrate the topics presented. The application is an interactive Mandelbrot viewer, which allows the user to interactively zoom and pan over the Mandelbrot set. The reason for choosing it is that it has the kinds of computational features that make for a good demonstration. In particular, the computational complexity for each pixel varies widely over an image, helping provide a good demonstration for how breaking a large computational task into smaller ones creates the kind of automatic load balancing necessary for a scalable application. Additionally, the amount of computing is easy to configure by changing the size of the display or the maximum number of iterations used in the equation. Finally, the rendering of the Mandelbrot set is visually pleasing, helping engage the interest of the user. For more details regarding the Mandelbrot set, refer to Appendix C.

The fundamental scalability concepts are presented in Section 3.1. Following the concept presentation, implementations of the concepts is presented in Section 3.2. Finally, the application walkthrough begins in Section 3.3.

3.1 Concepts

Designing and building scalable applications is not that difficult conceptually, the hard work is in the details of the specific application domain and how to apply the concepts to the domain. Therefore, the appropriate starting point is to understand the conceptual scalable building blocks; it is only from this basis that a specific application can be constructed. Any scalable application begins with these concepts, building upon them to provide the computational services for an application purpose.

3.1.1 Thread Management

Effective thread management is essential to any scalable application. Most developers know how to create threads well enough, but what isn't as well understood is how to utilize threads in a way that dynamically scales to fully utilize the system's capabilities. In order to achieve effective scalability through thread utilization, an application must be designed with a *global view* of thread management versus a *local view*. What does that mean?

During the design of an application using a local view, a developer identifies a problem that is solved through the use of thread-level parallelism. The developer creates the necessary threads for that problem and moves on. Over time, as an application is built using this approach, various sub-systems, when taken in isolation (locally), all solve a problem using some number of threads. However, when viewed together (globally), these sub-systems result in a large number of system threads, all of which are competing for limited CPU resources, resulting in thread contention and lowering overall system utilization as threads compete, rather than cooperate, for limited system resources.

An application developed using a global view identifies the CPU as an application resource to share among all sub-systems. This shared resource is commonly exposed and managed through a *Thread Pool*; Section 3.1.3 discusses Thread Pools in detail. As a developer identifies problems that can take advantage of thread-level parallelism, the Thread Pool is used. Over time, as this application is built and different sub-systems request threads from the Thread Pool, they are effectively managed without overwhelming the system's resources, resulting in an efficient and scalable system.

The system built using a local view may result in hundreds of threads, all competing for the same CPU and other hardware resources, and little chance for true scalability. The system built using a global view will result in a system that coordinates the threads over all the application sub-systems through the use of a Thread Pool. The Thread Pool dynamically matches the number of threads to the available system resources, allowing the whole application to scale and fully utilize the available hardware resources.

3.1.2 Tasks

The fundamental building block of a scalable application is a *task*. A task is a combination of code and data, where the code operates over the data. Furthermore, each task must be able to be computed in parallel with other tasks.

An application must subdivide its operation into tasks, with different types of tasks having different code and data. These tasks are then distributed to available threads for computation. As soon as a thread is available, it is matched with a task, executes the code over the data, returns a result, is then matched with the next task, etc. Because each task is computed independently, the number of tasks that can be computed in parallel scales with the number of available CPU cores. It is this concept and capability that forms the basis for all scalable computation presented in this chapter and throughout the book.

The data associated with a task ideally should not be shared by another task which is being computed in parallel, for reasons of correctness and scalability. With respect to correctness, if the same memory location is being read and written concurrently, this can lead to a race condition. To avoid these race conditions, the data must be synchronized via mutexes, locks, critical sections, or other similar techniques. With respect to scalability, if the same data is being accessed in parallel with proper synchronization, this may lead to linearizing task execution, resulting in a system that will not scale. Careful design and planning must be taken in order to construct tasks that will compute in parallel and scale as more computing resources become available.

This chapter handles dependencies between tasks in a simple manner, all tasks are computed in the order they are generated. Chapter 4 explores tasks in the context of varying priorities among the tasks, and Chapter 5 explores the topic of dependencies existing between tasks, enforcing that some tasks must complete before others can begin.

3.1.3 Thread Pool

A *Thread Pool* is a resource where a managed set of threads are matched with incoming tasks. The purpose of the thread pool is to manage the number of threads created by an application, in order to properly scale and fully utilize the system, while not overwhelming the system resources. In addition to the benefit of being able to manage threads for scalability, it is more efficient to reuse existing threads rather than creating new threads, thereby offering an important thread utilization optimization.

Thread pools usually start with some small fixed number of threads created at application startup, and then dynamically adjust the number of threads, up to some specified maximum. The number of threads in the thread pool can increase or decrease based upon application utilization. The thread pool can monitor the wait times of the incoming tasks, along with the wait time of threads. If the wait time of tasks is increasing, additional threads, up to the maximum allowed, are created. If the wait time of available threads is decreasing, threads can be destroyed, down to a minimum threshold.

Two data structures are used to compose a thread pool, one for the threads and another for tasks. Threads are placed into a *set* or similar container, simply as a way to track them for the lifetime of the application. Each thread in the pool waits in an efficient state until it is signaled that a new task is available to execute. Incoming tasks are placed into a first in first out (FIFO), and matched up with the a waiting thread, with the task being removed from the FIFO queueu upon being matched with a thread. As a thread complete work on a task, it first checks to see if another task is available. If another task is available, it is immediately taken and executed, otherwise the thread returns to an efficient waiting state, again, waiting to be signaled another task has become available. In the case there are tasks in the task queue, but no available threads, the task queue continues to fill, until threads become available for matching with tasks.

There is no single, or simple, answer to the question of how many threads at a minimum and maximum to create. The short answer is that it depends upon the application. A longer answer is to remember that a computing system is composed of more than just a CPU, there are many hardware components that can work in parallel. These components include the CPU, GPU, network devices, and storage devices; all of these devices can and should be utilized to work in parallel.

At a minimum the thread pool should have at least as many threads as CPU cores. The maximum number is a little more complex, and not that easy to decide. Depending upon the mix of tasks an application generates helps guide this decision. An application that is almost completely computational doesn't need many more threads than the number of CPU cores. On the other hand, an application that has a mix of computational and IO tasks will benefit from having a larger upper limit. The final answer is that the only way to know is to try different levels and measure the performance of the application.

An application has a single thread pool, it provides the global view of thread management for the application. It is assumed the application using the thread pool is the only one running on the system that is trying to maximize utilization. In the case this isn't true, that there are several applications wanting to share system resources, the applications should look to the underlying operating system (e.g. Windows) for a shared thread pool resource.

3.2 Building Blocks

This section of the chapter walks through the implementation of the the building block concepts of the Task and Thread Pool. The next section, Section 3.3 demonstrates the use of these building blocks in the context of a real application, an interactive Mandelbrot visualization application.

3.2.1 Task

As described in Section 3.1.2, a task is the fundamental building block for any computational component. Looking forward just a bit towards the thread pool, it is necessary to allow any type of task to be handled by the thread pool. With this in mind, an abstraction of a task is necessary, one that allows any task to be accepted by the thread pool and when ready, have its computation performed. In C++ terms, this means defining an abstract base class from which all concrete tasks are derived; the name for this class is `Task`.

The abstract base class `Task` defines the behavior that all derived tasks must implement. The first behavior is the computational task for which the task is defined, the method name for this behavior is `execute`. Because the `Task` class can not possibly know anything about the computation, it is a pure virtual method, which causes the class to be abstract. The next behavior necessary is to provide a means through which some other application component can be informed when the task computation is complete. This is accomplished through a combination of taking a `std::function` through the `Task` constructor and invoking that function in the `complete` method. The `Task` class is shown in Listing 3.1.

Listing 3.1: Abstract Task Class

```
class Task
{
public:
  Task(std::function<void ()> onComplete) :
    m_onComplete(onComplete)
  {
  }
  virtual ~Task() {}

  virtual void execute() = 0;
  void complete() { if (m_onComplete) { m_onComplete(); } }

private:
  std::function<void ()> m_onComplete;
};
```

At first glance it may seem odd that the `complete` method isn't virtual or abstract, why not allow the derived task classes to have custom behavior when the task is complete? The reason is that we want to allow code other than the task itself to be informed of the completion, which is why the `Task` constructor accepts an `onComplete` function. With that said it is often meaningful for the task itself to perform some additional work following the completion of the `execute` function. Chapter 8 discusses this and demonstrates an approach to effectively solving this problem.

It is worth noting the destructor for the `Task` class is declared as `virtual`. This is to ensure that derived class destructors are correctly called.

3.2.2 Thread Pool

The thread pool, discussed in Section 3.1.3, is the heart of the scalability framework. Conceptually a thread pool is composed of a set of available threads and a queue of available tasks. Surprisingly, the amount of code necessary to build this core system is relatively modest. The implementation associated with this chapter is less than 300 lines of code, with more than half of those comments and other boilerplate code such as `#includes`. This is possible, in part, due to the concise code possible using C++11.

Because the thread pool is a resource that needs to be available throughout an application, it is implemented as a Singleton[1]. The first time the `ThreadPool` class `instance` member is invoked, the initial pool of threads is created. This implementation creates four plus the number of available CPU cores, that number does not change throughout the lifetime of the thread pool. The reason for the four extra threads is to ensure computational work can be performed while dealing with the various bits of overhead naturally incurred during signaling threads and matching them with tasks.

The declaration of the `ThreadPool` class is provided in Listing 3.2. The singleton implementation is visible by the static `instance` accessor method, the protected constructor, and the `m_instance` member.

Listing 3.2: `ThreadPool` Declaration

```
class ThreadPool
{
public:
  static std::shared_ptr<ThreadPool> instance();
```

[1]http://en.wikipedia.org/wiki/Singleton_pattern

```
  static void terminate();

  void enqueueTask(std::shared_ptr<Task> task);

protected:
  ThreadPool(uint16_t sizeInitial);

private:
  static std::shared_ptr<ThreadPool> m_instance;

  std::set<std::shared_ptr<WorkerThread>> m_threads;

  ConcurrentQueue<std::shared_ptr<Task>> m_taskQueue;
  std::condition_variable m_eventQueue;
};
```

The thread pool is a simple `std::set` of `WorkerThread` pointers. The container is a `std::set` because threads are only added at startup and then nothing is removed or added during the remaining lifetime of the thread pool. The reason threads don't come and go in this container is that the `WorkerThread`s all receive a reference to the task queue and pull tasks from that queue as they become available.

The task queue is given by the `m_taskQueue` member, a thread safe queue of pointers to tasks. The thread safe queue is provided by a class called `ConcurrentQueue`; Section 3.2.2 details the implementation of this queue.

Associated with this queue is a `std::condition_variable` named `m_eventQueue`. This condition variable is signaled each time a task is added to the task queue. All worker threads are given a reference to this condition variable. When a task is added, the condition variable is signaled using the `.notify_one` method. This ensures that only one waiting (if available) thread receives the signal, falls through, and attempts to obtain a task from the shared task queue. This is the preferred behavior, ensuring only a single worker thread unblocks on the event, rather than having all waiting threads unblock and create unnecessary contention on the task queue as they all attempt to grab the newly available task.

The only behavior exposed by the `ThreadPool` class is the ability to add a task, through the `enqueueTask` method. The code for enqueuing a task is very simple, as shown in Listing 3.3. The task is added to the private task queue and the associated condition variable `m_eventTaskQueue` is signaled.

Listing 3.3: Enqueuing a Task

```
void ThreadPool::enqueueTask(std::shared_ptr<Task> task)
{
  m_taskQueue.enqueue(task);
  m_eventTaskQueue.notify_one();
}
```

Although this code is simple, more is happening than is seen at first glance. The member `m_taskQueue` is thread safe, ensuring that only one thread at a time is adding a task. Secondly, when `m_eventTaskQueue` is signaled, it (potentially) causes a waiting thread to fall through and begin work on the newly added task.

The final part of the `ThreadPool` singleton is a static `terminate` method; the code for this method is shown in Listing 3.4. The purpose of this method is to perform a graceful shutdown of the worker threads. The first step is to ask each of the worker threads to voluntarily terminate as soon as they are finished with their current task. Not all threads may be working on a task, they may be waiting on the task notification event. For those threads the task queue condition variable is signaled with a `notify_all`, causing all threads waiting on that condition variable to unblock and find out they need to terminate. Finally, the method waits to return until all worker threads have completed by performing a `join` on them.

Listing 3.4: Thread Pool Terminate

```
void ThreadPool::terminate()
{
  if (m_instance != nullptr)
  {
    for (auto thread : m_instance->m_threads)
    {
      thread->terminate();
    }

    m_instance->m_eventTaskQueue.notify_all();

    for (auto thread : m_instance->m_threads)
    {
      thread->join();
    }

    m_instance.reset();
    m_instance = nullptr;
  }
}
```

Task Queue

The queue used to hold the tasks is a lightweight wrapper around the `std::queue`. The name of the class is `ConcurrentQueue` and provides synchronized access through the `enqueue` and `dequeue` class members. `ConcurrentQueue` is a template class, allowing any data type to be used; a `Task` in the case of the Mandelbrot application. The declaration of the class is found in Listing 3.5. The implementations of the `enqueue` and `dequeue` show in Listing 3.6 and Listing 3.7 respectively.

Listing 3.5: `ConcurrentQueue` Declaration

```
template <typename T>
class ConcurrentQueue
{
public:
  void enqueue(const T& val) { ... }
  bool dequeue(T& item) { ... }

private:
  std::queue<T> m_queue;
  std::mutex m_mutex;
};
```

The `ConcurrentQueue` is composed of a `std::queue`, which is the container for the tasks, and a `std::mutex`, which is used to support synchornized access. Rather than presenting the same interface as `std::queue`, this class provides only the ability to add a new item to the queue or remove an item from the queue. These two methods provide all the capability required by the scalable application.

The code for the `enqueue` method is found in Listing 3.6. The first line of code locks on the mutex. If the mutex is not available, the calling thread is blocked until the mutex becomes available. Once the lock is obtained, the item is added to the queue. When the method goes out of scope, the mutex lock is released. Using a lock in this way is the correct RAII approach, versus making specific calls to `lock` and `unlock` on the mutex.

Listing 3.6: Add Item to `ConcurrentQueue`

```
void enqueue(const T& item)
{
  std::lock_guard<std::mutex> lock(m_mutex);
  m_queue.push(item);
}
```

The code for the `dequeue` method is found in Listing 3.7. In the same way as the `enqueue` method, this method blocks until the calling thread is able obtain a lock on the mutex. The interesting approach in this method is that it returns a boolean `true` or `false` depending upon whether or not an item was returned in the reference parameter. The reason for this is to allow a calling thread do something different based upon whether or not there is any work to do.

Listing 3.7: Remove Item from `ConcurrentQueue`

```
bool dequeue(T& item)
{
  std::lock_guard<std::mutex> lock(m_mutex);

  bool success = false;
  if (!m_queue.empty())
  {
    item = m_queue.front();
    m_queue.pop();
    success = true;
  }

  return success;
}
```

Worker Thread

The threads associated with the thread pool require a specific implementation. The core logic of these threads is to watch the task queue and grab the next available task. Clearly, because there is more than one thread, the threads must coordinate retrieving tasks from the task queue. This is performed in two ways. The most obvious way is through standard synchronization techniques, i.e. locking while accessing the shared task queue. The second way is through the use of a condition variable. All threads go into an efficient wait state, waiting to be signaled by the condition variable. When a new task is added to the task queue, only one thread is signaled. Having only one thread signaled reduces contention on the task queue by preventing unnecessary locking by a large number of threads, only one of which is going to obtain the next task.

The declaration for the `WorkerThread` class is shown in Listing 3.8. The constructor accepts references to the task queue and the task notification condition variable. Internally, the thread class maintains a pointer to its own C++11 thread object, along with a boolean `m_done` that indicate whether or not the thread should voluntarily terminate. The `m_mutexEventTaskQueue` is defined as a `static` to ensure only a single mutex instance is used across all `WorkerThreads`. The reason for this is that all `condition_variables` must use the same mutex in order to be correctly signaled when one of their `notify` methods is called.

Listing 3.8: `WorkerThread` Declaration

```
class WorkerThread
```

```
{
public:
  WorkerThread(
    ConcurrentQueue<std::shared_ptr<Task>>& taskQueue,
      std::condition_variable& eventTaskQueue);

  void terminate();
  void join();

private:
  std::unique_ptr<std::thread> m_thread;
  bool m_done;

  ConcurrentQueue<std::shared_ptr<Task>>& m_taskQueue;
  std::condition_variable& m_eventTaskQueue;
  static std::mutex m_mutexEventTaskQueue;

  void run();
};
```

As soon as an instance of the `WorkerThread` class is created, it begins execution of a thread, beginning with its private `run` method. The constructor consists of a single line of code that creates and kicks off the execution of the thread, as seen in Listing 3.9. The reason a `unique_ptr` is used is because the class itself is the only code that will ever have a pointer to the thread, therefore no need to use a `shared_ptr`.

Listing 3.9: `WorkerThread` Constructor

```
WorkerThread::WorkerThread(
    ConcurrentQueue<std::shared_ptr<Task>>& taskQueue,
    std::condition_variable& eventTaskQueue) :
  m_taskQueue(taskQueue),
  m_eventTaskQueue(eventTaskQueue),
  m_done(false),
  m_thread(nullptr)
{
  m_thread = std::unique_ptr<std::thread>(
    new std::thread(&WorkerThread::run, this));
}
```

The internal thread code, shown in Listing 3.10, executes a fairly simple loop. The body of the loop checks the task queue for an available task. If one is available, its two methods `execute` and `complete` are performed in sequence. After completion of these two methods, the task queue is checked again. When no task is available, the thread goes into an efficient wait state, waiting until the condition variable `m_eventTaskQueue` associated with the task queue is signaled. The `run` method stays in its loop until asked to voluntarily terminate, which occurs when `m_done` is set to `true`.

Listing 3.10: `WorkerThread::run` Method

```
void WorkerThread::run()
{
  while (!m_done)
  {
    std::shared_ptr<Task> task;
    if (m_taskQueue.dequeue(task))
    {
      task->execute();
```

```
            task->complete();
        }
        else
        {
            std::unique_lock<std::mutex> lock(m_mutexEventTaskQueue);
            m_eventTaskQueue.wait(lock);
        }
    }
}
```

Remember that the `execute` method of the task is the code from the derived task. On the other hand, while the code for the `complete` method is common to all tasks, it also executes in the context of the thread. Therefore, the function called by the `complete` method should be relatively short, otherwise the task thread will be working on non-computational code.

The public interface to the `WorkerThread` consists of two simple methods, `terminate` and `join`. The purpose of the `terminate` method is to signal to the thread it should voluntarily terminate as soon as possible. The join method is only a pass-through to the underlying thread `join` method. The code for these methods is found in Listing 3.11 and Listing 3.12 respectively.

Listing 3.11: `ConcurrentQueue::terminate` Method

```
void WorkerThread::terminate()
{
  m_done = true;
}
```

Listing 3.12: `ConcurrentQueue::join` Method

```
void WorkerThread::join()
{
  m_thread->join();
}
```

3.3 Scalable Mandelbrot Viewer

The fairly simple pieces of code presented in the first part of this chapter work to create an efficient, loading balancing, and scalable framework, but require an application to take on meaning. The remainder of this chapter shows how to take the scalable framework and utilize it in the context of an interactive Mandelbrot viewer application. To further demonstrate the generic nature and scalability of the underlying framework, the application also computes prime numbers while also computing the Mandelbrot set. As the application changes to match the framework throughout the rest of the book, the prime number computatation is carried forward, showing how the evolution of the framework continues to work naturally with different kinds of tasks.

As noted in the first part of this chapter, the Mandelbrot set works as a good demonstration because of the asymmetric nature of the comptational complexity of each pixel in the resulting image. In other words, the number of instructions it takes to compute the result for each pixel typically varies widely. Appendix C provides a more in-depth discussion of the mathematics, computational code, and visualization. If you are unfamiliar with the Mandelbrot set, please review the appendix before continuing with this chapter.

Upon startup the application shows an overview of the Mandelbrot set, initially a mostly blue region outlined by some red hints. The blue region contains the points considered to be inside the set, anything else is outside. The remaining points are represented as colors other than blue, based upon how many iterations it took to decide they are outside of the set. At this point, you can interact with the viewer by panning and zooming throughout the set. Take the time to explore the set, if you've not experienced it before, it is possible to find any number of interesting and beautiful regions. The panning and zooming controls are:

pan up : w or up arrow

pan down : s or down arrow

pan left : a or left arrow

pan right : d or right arrow

zoom in : q or numpad +

zoom out : e or numpad -

Behind the Mandelbrot visualization is a console window showing a series of increasing numbers scrolling by, these are prime numbers. In addition to the Mandelbrot set computation, the application is also computing prime numbers. The purpose for this is to demonstrate the general nature of the scalable framework, the ability to mix different tasks at the same time, along with showing how easy it is to define different kinds of computational tasks.

3.3.1 Application Source

There are three parts that compose the application source. The first is the Windows specific startup and message loop, followed by the Mandelbrot (and prime number) framework and computational logic, the last part is the task-based scalable framework. The following identifies the source files associated with each of these sections:

Windows Support
`WinMain.cpp`

Mandelbrot Framework
`IRange.hpp`
`Mandelbrot.hpp`
`Mandelbrot.cpp`
`MandelImageTask.hpp`
`MandelImageTask.cpp`
`MandelPartTask.hpp`
`MandelPartTask.cpp`
`NextPrimeTask.hpp`
`NextPrimeTask.cpp`
`ScalabilityApp.hpp`
`ScalabilityApp.cpp`

Scalable Framework
`ConcurrentQueue.hpp`
`Task.hpp`
`ThreadPool.hpp`
`ThreadPool.cpp`
`WorkerThread.hpp`
`WorkerThread.cpp`

The file `WinMain.cpp` provides the Windows specific application framework. This includes creating the Mandelbrot viewing window, the prime number console window, and the core message loop. This file contains the code to capture keyboard input and feed that back into the Mandelbrot viewing controls. Finally, this file creates and holds a shared pointer to a `ScalabilityApp` class.

The core application logic is contained within the `ScalabilityApp` class and the supporting `Mandelbrot`, `MandelImageTask`, `MandelPartTask`, and `NextPrimeTask` classes. These are all detailed in Section 3.3.2. As already described in the first part of this chapter, the underlying scalable framework is provided by the `ConcurrentQueue`, `Task`, `ThreadPool`, and `WorkerThread` classes.

3.3.2 Application Logic

The Mandelbrot application is based around the Windows message loop. The message loop runs by processing operating system messages as fast as possible. Each time through this loop the Mandelbrot viewer is updated by calling the `pulse` method of the `ScalabilityApp` class. Keyboard input is captured during the message loop through the `WinMessageHandler` function.

Startup

The `ScalabilityApp` constructor sets the `m_updateRequired` variable to `true`, indicating the next time through the `pulse` method a new view of the Mandelbrot set needs computed, and sets `m_inUpdate` to `false`, indicating the appplication is not currently computing a Mandelbrot set view. Additionally, the constructor sets the `m_currentPrime` variable to `1` and calls the `startNextPrime` method to kick off the background prime number generation.

The `startNextPrime` method creates a `NextPrimeTask` and places it on the `ThreadPool` task queue. A small lambda is passed in as the function to call on completion of the prime number computation task. This lamba calls the `startNextPrime` method, which causes the next prime number to be computed, creating an infinite prime number computation loop.

Keyboard Input

Each time an application supported key is pressed, a call is made into the `Mandelbrot` that adjusts the viewing parameters for the next time the Mandelbrot set is computed. An example of one of these methods is shown in Listing 3.13.

Listing 3.13: `ScalableApp::moveLeft` Method

```
void Mandelbrot::moveLeft()
{
  double distance = (m_mandelRight - m_mandelLeft) * MOVEMENT_RATE;
  m_mandelLeft -= distance;
  m_mandelRight -= distance;
  m_updateRequired = true;
}
```

The first part of the method updates the viewing parameters, while the last statement, `m_updateRequired = true;` works as a signal to indicate that a new viewing frame needs to be computed. This ensures the application only computes a new Mandelbrot set when the viewing parameters have changed. Furthermore, because the user may press multiple keys while the current frame is being rendered, this allows multiple view adjustments to be accepted and combined into the next frame computation.

3.3.3 Task Decomposition

As noted in Section 3.1.2, tasks are the fundamental building block for a scalable application. Defining these tasks is known as *Task Decomposition*. The demonstration application easily decomposes into two tasks, one for computing the Mandelbrot set and another to compute prime numbers. While the prime number computation makes sense as a small unit of work, computing an entire Mandelbrot set as a single task is too much work, more importantly it provides no ability to scale over more than one processor. Therefore, a closer look at the Mandelbrot task decomposition is in order.

In order to decide how to decompose the Mandelbrot image into sub-tasks, the nature of the Mandelbrot computation and understanding of how to optimize are necessary; a general truism for any task decomposition effort. With respect to the Mandelbrot set, each pixel computation is independent of every other pixel. Therefore, it is possible to decompose the tasks down to the individual pixel level. Additionally, the computational complexity of each pixel (potentially) differs widely. One pixel may take 1 iteration, another may take the application defined maximum (e.g., 1000 iterations). With respect to system optimization, batching many operations together is typically more efficient. This is especially true when considering the overhead involved in creating an instance of a task, adding it to the thread pool, matching it with an available thread, executing, and then completing the task.

Taking into consideration the nature of the Mandelbrot pixel computations and thinking about system optimization, it makes sense to define the small unit of computation as multiple pixels per task. Specifically, I have chosen to define each task as one row of pixels in the image. This selection was made from a combination of trying out different numbers of pixels, along with thinking about creating a large enough number of tasks to ensure the comptuation will scale well with an increasing number of CPU cores. Too few tasks, such as 6, won't scale beyond 6 CPU cores, whereas too many tasks, such as 10,000, incurs too much overhead from the tasking framework.

One issue that needs to be solved is ensuring the application doesn't try to start a new Mandelbrot image computation until the previous one has completed. The reason this is desired is to prevent cueuing up multiple images, each caused by a single keypress. Instead, it is better to allow multiple keypresses to take place while an image is being computed, accumulate all those changes, then request a new image only when the current one has completed. The framework presented in this chapter isn't capable defining dependencies between tasks, that comes in Chapter 5. Therefore, some other technique is necessary.

The approach taken in this chapter is to define a master Mandelbrot image task (`MandelImageTask`), and have that task spawn sub-image for each of the rows in the image. The application creates the `MandelImageTask` and places it on the task queue, and also sets the `m_inUpdate` flag to `true`, which prevents any additional `MandelImageTask`s from being created. `MandelImageTask` creates tasks for each of the rows and places those on the task queue. Rather than having `MandelImageTask` complete after placing the sub-image tasks on the task queue, it waits for all of those tasks to complete first. Once all of the sub-image tasks are complete, `MandelImageTask` finishes, which results in the application being notified of this completion. When the `MandelImageTask` completes, `m_inUpdate` is set back to `false`, which allows a new `MandelImageTask` to be created. In this way, it is possible to create a dependency between tasks, a master task doesn't complete until child tasks it spawns complete.

Prime Number Task

The `NextPrimeTask` is a good place to start, as it is about the simplest kind of task to create. Listing 3.14 shows the declaration of the class. The task is derived from the abstract base class `Task`, providing a custom constructor and implementing the pure virtual `execute` method.

Listing 3.14: `NextPrimeTask` Declaration

```
class NextPrimeTask : public Task
{
public:
  NextPrimeTask(
    uint32_t lastPrime,
    uint32_t& nextPrime,
    std::function<void ()> onComplete);

  virtual void execute();
private:
  uint32_t m_lastPrime;
  uint32_t& m_nextPrime;

  bool isPrime(uint32_t value);
};
```

The `NextPrimeTask` constructor, show in Listing 3.15, calls into the base `Task` constructor, passing the `onComplete` function into it. Additionally, the `lastPrime` and `nextPrime` parameters are assigned to their corresponding member variables. The `nextPrime` parameter is passed by reference, allowing the task to set the value during the `execute` method. When the task is complete, the source variable is updated with the newly computed prime number. Doing this requires a bit of careful thought, as it doesn't seem like the right thing to do in a multi-threaded application. What makes this work out correctly is that there can only be one `NextPrimeTask` ever executing at one time. Because of this, no reason to worry about synchronizing the `nextPrime` value between multiple threads, it can never happen.

Listing 3.15: `NextPrimeTask` Constructor

```
NextPrimeTask::NextPrimeTask(
      uint32_t lastPrime,
      uint32_t& nextPrime,
      std::function<void ()> onComplete) :
  Task(onComplete),
  m_lastPrime(lastPrime),
  m_nextPrime(nextPrime)
{
}
```

The `execute` method for the task is shown in Listing 3.16. This method simply computes the next prime number in the series. The purpose of this method isn't a fast technique, it is just a way to compute prime numbers for demonstration.

Listing 3.16: `NextPrimeTask::execute` Method

```
void NextPrimeTask::execute()
{
  m_nextPrime += 2;
  while (!isPrime(m_nextPrime))
  {
    m_nextPrime += 2;
  }
}
```

Mandelbrot Image Task

The `MandelImageTask` is the task that takes on the responsibility for generating a Mandelbrot image. As described earlier, this is done by creating a large number of sub-tasks (one for each line in the image) and waiting for them to complete before finishing. The essential parts of the declaration for the task is shown in Listing 3.17.

Listing 3.17: `MandelImageTask` Declaration

```
class MandelImageTask : public Task
{
```

```
public:
  MandelImageTask(
    double startX, double endX,
    double startY, double endY,
    uint16_t maxIterations,
    uint16_t sizeX, uint16_t sizeY,
    uint8_t* pixels, uint16_t stride,
    std::function<void ()> onComplete);

  virtual void execute();

private:
  std::atomic<uint16_t> m_partsFinished;
  std::condition_variable m_imageFinished;
  std::mutex m_mutexImageFinished;

  std::unique_ptr<uint16_t[]> m_image;

  void prepareColors();
  void copyToPixels();
  void completePart();
};
```

The constructor receives the Mandelbrot viewing parameters, along with details regarding the physical location and layout of the memory for where to place the final image. Additionally, the constructor allocates private memory to place the reults of the image sub-tasks. The code for this allocation is shown in Listing 3.18. The reason for using `std::unique_ptr` is because this pointer should only ever be held by the task itself. There is also another benefit to using `std::unique_ptr` is that its deleter correctly deletes arrays, where `std::shared_ptr` does not. The reason for using `new` to perform the allocation rather than `std::make_unique` is because the C++11 committee made a mistake in not providing `std::make_unique` for arrays (it is coming in C++14).

Listing 3.18: `MandelImageTask` Constructor

```
m_image = std::unique_ptr<uint16_t[]>(new uint16_t[sizeX * sizeY]);
```

The heart of this task is contained in the `execute` method. The first part of the method creates the sub-tasks, the second part waits for the sub-tasks to complete, and the final part is a call to place the results into the viewing image memory. The part of the method that creates the sub-tasks is shown in Listing 3.19. In short, each line of the image is created as a new task and placed on the thread pool queue.

Listing 3.19: `MandelImageTask` Create Sub-Tasks

```
for (auto row : IRange<decltype(m_sizeY)>(0, m_sizeY - 1))
{
  auto task = std::shared_ptr<MandelPartTask>(
    new MandelPartTask(
      m_image.get() + row * m_sizeX,
      m_sizeX, m_maxIterations,
      m_startY + row * deltaY,
      m_startX, m_endX,
      std::bind(&MandelImageTask::completePart, this)));
  ThreadPool::instance()->enqueueTask(task);
}
```

The interesting part of the sub-tasks is that the `completePart` private method is passed in as the completion function to call. The code for this method is found in Listing 3.20. The `MandelImageTask` maintains an atomic private member `m_partsFinished` that tracks how many of the sub-tasks have finished. As each sub-task finishes, this method is invoked, causing `m_partsFinished` to increment. After incrementing, a check is made to see if all tasks have completed, if they have, then the condition variable `m_imageFinished` is signaled, causing the `execute` method to stop waiting and continue executing.

Listing 3.20: `MandelImageTask::completePart`

```
void MandelImageTask::completePart()
{
  m_partsFinished++;
  if (m_partsFinished == m_sizeY)
  {
    m_imageFinished.notify_one();
  }
}
```

After creating the sub-tasks, the `execute` method goes into an efficient waiting state, waiting for all of the sub-tasks to complete. The code for this is shown in Listing 3.21. This segment of code performs a wait on the `m_imageFinished` condition variable. Because condition variables can wake without having been signaled, I have chosen to use the `wait` overload that accepts a test function. For the test function parameter, I have written a simple lambda that validates that all of the sub-image tasks have actually completed. If they haven't, the wait returns to an efficient state, if they have, the wait completes and execution continues.

Listing 3.21: `MandelImageTask` Wait for Sub-Tasks

```
std::unique_lock<std::mutex> lock(m_mutexImageFinished);
m_imageFinished.wait(lock,
  [this]()
  {
    return m_partsFinished == m_sizeY;
  });
```

Once all of the sub-tasks have completed their work, a call to `copyToPixels` is made, the code for this method is shown in Listing 3.22. The purpose of this method is to take the local task image results and copy them into the buffer being used to display the image. The `get` method of `std::unique_ptr` is used to obtain the underlying raw memory pointer, this is desired for performance reasons. This code moves over the pixels in the image, copying from the newly computed image into the destination image pointer passed in through the constructor.

Listing 3.22: `MandelImageTask::copyToPixels`

```
void MandelImageTask::copyToPixels()
{
  uint16_t* source = m_image.get();
  uint8_t* destination = m_pixels;
  for (auto row : IRange<decltype(m_sizeY)>(0, m_sizeY - 1))
  {
    uint8_t* destination = m_pixels + row * m_stride;
    for (auto column : IRange<decltype(m_sizeX)>(0, m_sizeX - 1))
    {
      uint16_t color = *(source++);
      *(destination++) = m_colors[color].r;
      *(destination++) = m_colors[color].g;
      *(destination++) = m_colors[color].b;
      *(destination++) = 0;
    }
  }
}
```

An interesting note about this being done as part of the task itself, rather than returning a result back to the `ScalabilityApp` class and having it perform the action; similar to how this is done with the `NextPrimeTask`. As currently written, two threads are (potentially) accessing the display memory at the same time, the task thread writing a new result, and the main application thread reading it for display. For a demonstration application like this it is fine, the image rendering is hardly affected. However, for something other than a demonstration, a double buffering technique is clearly the approach that should be taken. It is possible to create a double buffered solution, but at the expense of additional code complexity that that takes away from the focus of this chapter, which is scalability through task decomposition.

Mandelbrot Image Section Task

The `MandelPartTask` class performs the heavy computational lifting for the Mandelbrot image. The `MandelImageTask` spawns a large number of these tasks, but it is the `MandelPartTask` that computes the actual image data. This task is quite similar to the `NextPrimeTask` class with respect to its simplicity. It takes a few parameters that tell it what part of the image to compute and where to place the results, and that is all. The declaration for the class is found in Listing 3.23.

Listing 3.23: `MandelPartTask` Declaration

```
class MandelPartTask : public Task
{
public:
  MandelPartTask(
    uint16_t* imageRow,
    uint16_t resolution, uint16_t maxIterations,
    double y, double startX, double endX,
    std::function<void ()> onComplete);

  virtual void execute();

private:
  uint16_t* m_imageRow;
  uint16_t m_resolution;
  uint16_t m_maxIterations;
  double m_y;
  double m_startX;
  double m_endX;

  uint16_t computePoint(double x0, double y0);
};
```

The constructor receives a raw pointer that tells the task where to place its results. With C++11, I'd normally not make use of raw pointers, but there are always exceptions, this is one of those exceptions. The purpose of this task is to compute a row of pixels in a Mandelbrot image and place the results into a memory region owned by another part of the code. Because this task does not ever have ownership of the memory, there really isn't a reason to need any kind of shared pointer. The second reason is that of performance. Because the computation of the pixels is performance sensitive, where reasonable small performance opportunities should be taken. Finally, and maybe most importantly, when the parent task is creating these sub-tasks, shared pointers don't exist to each of the image rows, therefore, it would require creating shared pointers to even give to the `MandelPartTask` code.

The `execute` method is shown in Listing 3.24. This method loops through each of the pixels in a row, computes the number of iterations, and then uses a smooth coloring algorithm to determine the pixel color. Because the task is writing pixel values directly into the parent task image result, there is nothing else to do after the computation is done, the results are already in place.

Listing 3.24: `MandelPartTask::execute` Method

```
void MandelPartTask::execute()
{
  double deltaX = (m_endX - m_startX) / m_resolution;
  for (auto x : IRange<decltype(m_resolution)>(0, m_resolution - 1))
  {
    auto iterations = computePoint(m_startX + x * deltaX, m_y);

    double colorIndex = iterations - log2MaxIterations;
    colorIndex = (colorIndex / m_maxIterations) * 768;
    colorIndex = std::min(colorIndex, 767.0);
    colorIndex = std::max(colorIndex, 0.0);

    m_imageRow[x] = static_cast<uint16_t>(colorIndex);
  }
}
```

Additional details of the Mandelbrot comptuation, along with a discussion of the `computePoint` method are found in Appendix C.

3.4 Summary

This chapter introduced the concept of scalability through task-based computation. Only a few simple building blocks are necessary to achieve effective scalability. The first is the basic building block of a task. A task is a unit of computation, with the intention that many (or all) tasks can be computed in parallel. The second is an application wide available thread pool through which threads and tasks are matched and dispatched for computation. The final part is an application designed and constructed around these building blocks.

The application presented in this chapter shows how each of the building blocks are relatively simple to build and utilize. It also shows how to take a computationally intensive task, the computation of a Mandelbrot set, break it down into sub-tasks that can be computed in parallel, results collected, and then presented to the user. Additionally, the application shows that different kinds of tasks can flow through the scalable framework, allowing any number of different kinds of computational activities to take place at the same time. Finally, the application demonstrates scalability through improved performance on systems with larger numbers of CPU cores.

4 | Scalability - Priority

For some applications, not all tasks are equal, some require higher priority over others. This chapter introduces the concept of priority to the task framework, which offers a developer greater control over the ordering of application tasks. The impact on the task framework is relatively modest, and nearly trivial in the application code. The code presented in this chapter is modifed from Chapter 3, extended to support tasks with priority.

There are a variety of approaches for a priority based system, keeping in mind there are possibly many worker threads running on a large number of CPU cores. One approach is to have all threads select the highest priority task available. This requires an aging scheme to ensure low priority tasks are not starved. Another approach is to have most worker threads select the highest priority task available, and a smaller number of worker threads that select tasks from other (lower) specified priorities. With this approach, no aging scheme is necessary, simplifying the code complexity. Furthermore, this approach allows low priority tasks to always be worked on, even in the presence of a constant stream of higher priority tasks. It also gives the application developer a finer grained control of how tasks flow through a system. The second approach is presented in this chapter and implemented in the Mandelbrot visualization application.

This chapter adds priority to the fundamental scalability concepts, building upon those presented in Chapter 3. With this concept now introduced, Section 4.1 discusses the same building blocks presented in Chapter 3 with respect to the changes necessary to support task priority. Finally, the Mandelbrot application is revisited and the changes necessary to support task priority are presented in Section 4.2.

4.1 Building Blocks

This section of the chapter walks through the implementation, highlighting the changes needed to transform the framework code from Chapter 3 into one that supports task priority. In general, the changes are fairly modest, with the exception of a new priority queue.

4.1.1 Priority Queue

The most interesting part of the changes introduced in this chapter is with the choice of priority queue implementation. The first-thought choice is to use the standard library `std::priority_queue`, and place it in a lightweight synchronization wrapper, similar to what was done in the previous chapter with `ConcurrentQueue` through the use of `std:queue`. However, we have a requirement that worker threads can request tasks with priority other than the highest. Specifically, that worker threads can select from specific priorities. The `std:priority_queue` is designed to return only the highest priority item; realistically another approach is necessary.

Thankfully, there is another standard library container that meets this need, although it is not an obvious choice at first glance; the `std::multiset` container. While it is a set by definition, it allows multiple entries of items with the same key, with those items in the set of the same key removed in the order they were added. It is this feature, along with the ability to specify an item comparison operator that makes it possible to utilize the `std::multiset` for the framework priority queue.

The `std::multiset` is wrapped into a new type named `ConcurrentMultiqueue` that provides lightweight synchronization along with methods to add and remove tasks. The remainder of this section details the implementation.

Task Comparison

Because tasks are an Abstract Data Type (ADT), the `std::multiset` needs to be given a way to compare one `Task` with another. The `std::multiset` takes as one of its template parameters a binary predicate that performs the comparison between the set items. The binary predicate must return a `bool` result of `true` if the first element should be ordered before the second. For this framework I have chosen to implement a `std::binary_function` that compares tasks based upon their priority. The code for this function is shown in Listing 4.1.

Listing 4.1: `TaskCompare` Predicate

```
class TaskCompare : public std::binary_function<
  std::shared_ptr<Task>,
  std::shared_ptr<Task>, bool>
{
public:
  bool operator()(
    const std::shared_ptr<const Task>& lhs,
    const std::shared_ptr<const Task>& rhs) const
  {
    return (lhs->m_priority < rhs->m_priority);
  }
};
```

Remember that tasks are managed through `std::shared_ptr`s, therefore the `std::binary_function` types are shared pointers to tasks. The comparision operator inside the class is quite simple, it returns `true`/`false` based upon the priority of the two tasks.

`ConcurrentMultiqueue` Types

The `ConcurrentMultiqueue` is a template class that accepts three different template parameters. The class type declaration is show in Listing 4.2.

Listing 4.2: `ConcurrentMultiqueue` Declaration

```
template <typename T, typename P, typename C>
class ConcurrentMultiqueue
```

Type `T` is the data type represented in the container. In this application `T` is a shared pointer to a task, `std::shared_ptr<Task>`. The second type `P` is the data type used to represent priority. For this application `P` is an inner `enum` class of type `Task::Priority`. Finally, `C` is the boolean operator that provides the comparision between the types. As described in the previous section, this is a binary function named `TaskCompare`. Even though `TaskCompare` is always the operator, I decided to leave it as a template parameter in a effort to keep the ConcurrentMultiqueue class as generic as possible, even though there is one piece in the class that is tied to the implementation of the `Task` class.

Enqueuing Items

The code for enqueuing items into the container is fairly simple, nothing more than a simple synchronization wrapper around the `enqueue` method provided by the `std::multiset`. The code for the `enqueue` method is shown in Listing 4.3.

Listing 4.3: `enqueue` Implementation

```
void enqueue(const T& item)
{
  std::lock_guard<std::mutex> lock(m_mutex);
  m_queue.insert(item);
}
```

Dequeue Highest Priority

This `dequeue` method returns `true`/`false` depending upon whether or not an item is returned through the reference parameter. In other words, this method does not block if there is nothing in the queue, it will return with `false` if nothing was available when it was called. The `begin` method of the `std::multiset` returns an iterator to the highest priority item in the set. As a result, the code to dequeue the highest priority item is relatively simple, as shown in Listing 4.4.

Listing 4.4: Dequeue Highest Priority

```
bool dequeue(T& item)
{
  std::lock_guard<std::mutex> lock(m_mutex);
  bool success = false;

  auto itr = m_queue.begin();
  if (itr != m_queue.end())
  {
    item = *itr;
    m_queue.erase(itr);
    success = true;
  }

  return success;
}
```

This method begins by locking on the mutex to ensure correct thread synchronization. Next, an iterator to the first item in the set is grabbed through the `begin` method. Because there might not be an item in the set, the iterator is checked to see if it came from an empty set or is an actual item. If an actual item was retrieved, the iterator is dereferenced and the value stored into the `item` parameter passed by reference. Once a copy of the item is made into the reference parameter, the item is removed from the container and the `success` indicator is set to `true`. A quick note about the performance of this copy. The items stored in the container are `std::shared_ptr`s. Therefore, the copy is only that of a shared pointer, which is quite small.

Dequeue Selected Priority

The `ConcurrentMultiqueue` provides the ability to remove an item of a selected priority, rather than the highest priority item overall. This is provided through the overloaded `dequeue` method shown in Listing 4.5.

Listing 4.5: Dequeue Selected Priority

```
bool dequeue(P priority, T& item)
{
  std::lock_guard<std::mutex> lock(m_mutex);
  bool success = false;

  auto itr = std::find_if(m_queue.begin(), m_queue.end(),
    [priority](T value)
    {
      return value->getPriority() == priority;
    });

  if (itr != m_queue.end())
  {
    item = *itr;
    m_queue.erase(itr);
    success = true;
  }

  return success;
}
```

As compared to the other `dequeue` method, the only difference is in how the item is selected. To get the next item of the specified priority the standard algorithm `std::find_if` is used to look through the set. Even though it is a template class, the lambda used in support of the `find_if` knows that it is working with `Task` types and uses the `getPriority` method. This keeps the `ConcurrentQueue` from being truly generic, but this isn't a problem as it is used only for this application. Once the `find_if` algorithm returns, the remainder of the code is the same as the previously described `dequeue` method.

4.1.2 Task

The `Task` is much the same as before, but updated to include the concept of priority. In fact, it is the `Task` class that provides the source for priority throughout the scalable framework. A priority type is added to the framework through the use of an `enum` class named `Priority`. This is shown in Listing 4.6. This enumeration spells out the three different priority levels used in the framework.

Listing 4.6: `Priority` Definition

```
enum class Priority : uint8_t
{
  One = 1,
  Two = 2,
  Three = 3
};
```

The remainder of the `Task` class declaration is shown in Listing 4.7. The constructor is modified to include a new `Priority` parameter, which is then assigned to the private `m_priority` attribute. In addition to the new `m_priority` attribute, a `getPriority` member is added which returns the task priority.

Listing 4.7: `Task` Declaration

```
class Task
{
public:
  Task(std::function<void ()> onComplete, Priority priority = Priority::One) :
    m_onComplete(onComplete),
    m_priority(priority)
  {
  }
  virtual ~Task() {}

  virtual void execute() = 0;
  void complete()  { if (m_onComplete) { m_onComplete(); } }
  Priority getPriority()  { return m_priority; }

private:
  Priority m_priority;
  std::function<void ()> m_onComplete;
};
```

4.1.3 Thread Pool

The public interface to the `ThreadPool` class remains the same, however the implementation is updated to support worker threads dedicated to different levels of prioirty. Structurally the `ThreadPool` remains essentially the same. The most obvious change is the addition of notification events for when tasks of different priorities are added. Listing 4.8 shows the revised `ThreadPool` class declaration.

Listing 4.8: `ThreadPool` Declaration

```
class ThreadPool
{
public:
  static std::shared_ptr<ThreadPool> instance();
  static void terminate();

  void enqueueTask(std::shared_ptr<Task> task);

protected:
  ThreadPool(uint16_t sizeInitial);

private:
  static std::shared_ptr<ThreadPool> m_instance;

  std::vector<std::shared_ptr<WorkerThread>> m_threadQueue;

  ConcurrentMultiqueue<
    std::shared_ptr<Task>,
    Task::Priority,
    TaskCompare> m_taskQueue;

  std::condition_variable m_eventPriorityOne;
  std::condition_variable m_eventPriorityTwo;
  std::condition_variable m_eventPriorityThree;
};
```

The container for the tasks is changed to utilize the `ConcurrentMultiqueue`, which handles the prioritization of the tasks. The next change is to have individual `std::condition_variable` variables for each of the priority levels. These are used to signal worker threads when tasks of the corresponding priority are added.

As noted at the start of this chapter, the approach taken in this book is to allow for the possibility of worker threads dedicated to working on lower priority tasks concurrently with worker threads working on the highest available priority tasks. The ability to do this is as simple as creating a `WorkerThread` and passing in the highest priority level of tasks it can accept. Revisions to the `WorkerThread` are discussed in the Section 4.1.4. Creating the different worker threads is demonstrated in the `ThreadPool` constructor, shown in Listing 4.9.

Listing 4.9: `ThreadPool` Constructor

```
ThreadPool::ThreadPool(uint16_t sizeInitial)
{
  // Creating worker threads that take tasks of the highest available priority.
  for (auto thread : IRange<uint16_t>(1, sizeInitial + 4))
  {
    auto worker = std::make_shared<WorkerThread>(
      Task::Priority::One,
      m_taskQueue,
      m_eventPriorityOne);
    m_threadQueue.push_back(worker);
  }
  // Create a worker thread that takes only tasks of Priority::Three.
  auto worker =std::make_shared<WorkerThread>(
    Task::Priority::Three,
    m_taskQueue,
    m_eventPriorityThree);
  m_threadQueue.push_back(worker);
}
```

The `enqueueTask` is updated to signal one or more condition variables, depending upon the priority of the task added. The code for the revised method is shown in Listing 4.10. As with Chapter 3, the task is added to the queue, which in this case is now a prioirty queue. Following this, the priority of the task is checked and one or more of the condition variables is signaled.

Listing 4.10: Enqueuing a Task

```
void ThreadPool::enqueueTask(std::shared_ptr<Task> task)
{
  m_taskQueue.enqueue(task);
  switch (task->getPriority())
  {
    case Task::Priority::Three:
      m_eventPriorityThree.notify_one();
    case Task::Priority::Two:
      m_eventPriorityTwo.notify_one();
    case Task::Priority::One:
      m_eventPriorityOne.notify_one();
  }
}
```

The code works by allowing the case statement to fall though from the lowest priority to the highest. This ensures all worker threads checking tasks of higher priority are signaled of a low priority task in the event they aren't already busy. If a task of `Priority::One` is added, only the `m_eventPriorityOne` `condition_variable` is signaled. If a task of `Priority::Three` is added, one of each working thread priority levels is signaled.

4.1.4 Worker Thread

Worker threads have the same fundamental purpose, as used in Chapter 3, watch the task queue and grab the next available task. Now, however, the tasks come from a priority queue rather than a first in, first out queue. The primary change to the `WorkerThread` is to now search for tasks based upon a highest level of priority assigned through its constructor. The revised constructor is shown in Listing 4.11.

Listing 4.11: `WorkerThread` Constructor

```
WorkerThread::WorkerThread(
    Task::Priority priority,
    ConcurrentMultiqueue<
      std::shared_ptr<Task>,
      Task::Priority,
      TaskCompare>& taskQueue,
    std::condition_variable& taskQueueEvent) :
  m_priority(priority),
  m_taskQueue(taskQueue),
  m_eventTaskQueue(taskQueueEvent),
  m_done(false),
  m_thread(nullptr)
{
  m_thread = std::unique_ptr<std::thread>(
    new std::thread(&WorkerThread::run, this));
}
```

The `run` method has changed quite a bit in order to handle the new priority concept. The revised method is shown in Listing 4.12. This method works by staying inside of an inner loop checking to see if it can get a task of any priority level, starting with the highest priority to which it has been assigned. When no more tasks are available, it goes into an efficient waiting state, waiting on the `condition_variable` associated with its priority to be signaled.

Listing 4.12: `WorkerThread::run` Method

```
void WorkerThread::run()
{
  while (!m_done)
  {
    Task::Priority current = m_priority;
    bool donePriority = false;
    while (!m_done && !donePriority)
    {
      bool executed = false;
      std::shared_ptr<Task> task;
      if (m_taskQueue.dequeue(current, task))
      {
        task->execute();
        task->complete();
        executed = true;
      }
      donePriority = updatePriority(executed, current);
    }
    if (!m_done)
    {
      std::unique_lock<std::mutex> lock(m_mutexEventTaskQueue);
      m_eventTaskQueue.wait(lock);
    }
  }
}
```

A closer look at the inner loop of the `run` method is warranted. Before entering the loop, the `current` variable is set to the highest priority type of tasks it should execute. Upon entering the loop, an attempt is made to retrieve a task of `current` priority. If one is found, it is executed. If one is not found, the `updatePriority` method is called, which updates `current` to the next lowest priority, and the top of the loop is started again. If a task has not previously executed and the priority can not be lowered further, `updatePriority` returns `false` and the inner loop ends. In simple terms, the inner loop grabs the next availabe task with the highest remaining priority the worker thread is assigned.

It is also worth taking a look at the `updatePriority` method, shown in Listing 4.13. The simple purpose of this method is to update the priority of tasks the worker should execute. The first test in the method is to check if the worker had immediately executed a task. If it had, the current priority is restored (if it had been changed) to the highest priority for the worker. If no task had immediately been executed, the priority is lowered and `true` is returned. If no lower priority is possible, `false` is returned, which will end up causing the inner loop of the `run` method to terminate and return the worker to an efficient waiting state.

Listing 4.13: `WorkerThread::updatePriority` Method

```
bool WorkerThread::updatePriority(
  bool executed,
  Task::Priority& currentPriority)
{
  bool donePriority = false;
  if (!executed)
  {
    switch (currentPriority)
    {
      case Task::Priority::One:
        currentPriority = Task::Priority::Two;
        break;
      case Task::Priority::Two:
        currentPriority = Task::Priority::Three;
        break;
      default:
        donePriority = true;
        break;
    }
  }
  else
  {
    currentPriority = m_priority;
  }
  return donePriority;
}
```

4.2 Application Priority

In addition to revising the scalability framework to include the concept of task priority, the interactive Mandelbrot viewing application is also revised to support task priority. This is done by having the tasks associated with computing the next Mandelbrot image set to the highest priority, and the task associated with computing the next prime number set to the lowest.

The changes to the application are trivial, only the addition of a single new parameter to the task constructors, that of priority. The code for the revised `NextPrimeTask` is shown in Listing 4.14. The `priority` parameter is added to the end of the list, and also passed into the base `Task` class along with the `onComplete` function. The `MandelImageTask` and `MandelPartTask` are changed similarly.

Listing 4.14: `NextPrimeTask` Constructor

```
NextPrimeTask::NextPrimeTask(
    uint32_t lastPrime,
    uint32_t& nextPrime,
    std::function<void ()> onComplete,
    Priority priority) :
  Task(onComplete, priority),
  m_lastPrime(lastPrime),
  m_nextPrime(nextPrime)
{
}
```

The application runs as before, however, as the computational load increases, the `MandelPartTask` tasks take priority over the `NextPrimeTask` tasks, causing the next prime number generation to slow down. As a side note, prime numbers continue to be computed even at full or near full CPU load. The reason this happens is that the application has a small bit of time between the completion of one Mandelbrot image and the next as the new image pixels are copied over. During this time worker threads are available, which can work on `NextPrimeTasks`.

The way the `ThreadPool` is initialized in this demonstration results in many `Priority::One` worker threads and one `Priority::Three` worker thread. This means that `Priority::Three` threads can be worked on, even while there are `Priority::One` tasks available. For some applications, this may not be desirable. For those applications, only create worker threads with `Priority::One` as their starting priority.

The Mandelbrot viewer application can be modified by removing the `Priority::Three` worker. Upon doing so, the application will no longer consume any resources working on `NextPrimeTask` tasks while a Mandelbrot image is being computed. Prime numbers will still be computed, but never while any Mandelbrot image computation tasks are available.

4.3 Summary

This chapter introduced the concept of tasks having different priorities. This enables the developer to have a greater degree of control over an application with respect to the ordering of tasks with respect to each other through priority. This capability was primarily achieved by replacing the simple first in, first out queue from Chapter 3 with a new priority-based queue. Additionally, the thread pool was revised to utilize the new priority concept added to the tasks, ensuring that worker threads are given tasks appropriate for the level of priority they are assigned.

5 | Scalability - Task Dependencies

For many applications, some tasks must wait to be executed until other tasks have completed their work. This chapter introduces the concept of dependencies among tasks, that one or more tasks must complete before one or more other tasks can begin. As with the previous chapter using priority, this capability provides a developer another, very powerful, tool to aid in scheduling of computational tasks. Similarly, the impact on the task framework is relatively modest, and reasonably straightfoward in the application code. The biggest change is with the addition of a complex data structure to support the task dependencies. The bulk of this chapter is devoted to describing the data stucture, algorithms, and code that provide the task dependency capability. The code presented in this chapter is modified from Chapter 3, extended to support dependencies between tasks.

A two-fold goal with adding a capability to have task dependencies is to expose a powerful capability for application developers, while keeping the interface for doing so as simple as possbile. There is a lot of complexity in the underlying data structure and algorithms, and care must be taken to not expose unnecessary complexity to application code. Towards this goal, the code presented in this chapter only slightly modifies the interface for adding a task without dependencies, and adds a straightfoward interface for adding tasks with dependencies.

5.1 Concepts

There are two major components to creating the task dependency capability. The first is the data structure, which describes and maintains the dependencies among tasks. The second is an algorithm, or set of algorithms, used to operate on the data structure to determine the next available task, if any. Both of these are detailed in the next two sections.

5.1.1 Directed Acyclic Graph

A Directed Acyclic Graph (DAG)[1] is a directed graph that has no cycles. Figure 5.1 shows an example DAG. The term *directed* means that the edges from one node in the graph to other nodes have direction, and only one direction. The term *acyclic* means that it isn't possible to start at one node, follow a sequence of edges, and have that sequence result in returning to the starting node. In the context of task dependencies, these features mean it is possible to describe tasks (nodes), with dependencies (edges) to other tasks, and ensure that no tasks are either directly or indirectly dependent upon themselves.

As an application generates tasks it can be the case that one group of tasks is dependent upon another group, a third group of tasks is dependent upon a single task, and there may also be many tasks with no dependencies. Strictly speaking, a DAG is a single connected graph, which is not sufficient to represent this scenario. What is necessary, instead, is a data structure that has the ability to contain many locally connected graphs, and individual nodes, something that might be called a multi-DAG. An example of such a DAG is shown in Figure 5.2. For the purpose of communication ease, the term

[1] http://en.wikipedia.org/wiki/Directed_acyclic_graph

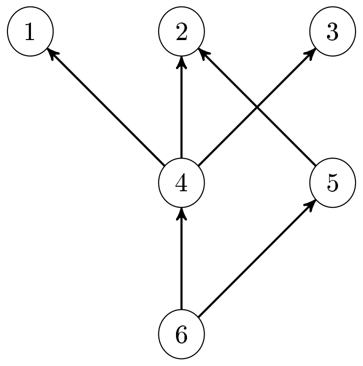

Figure 5.1: Directed Acyclic Graph

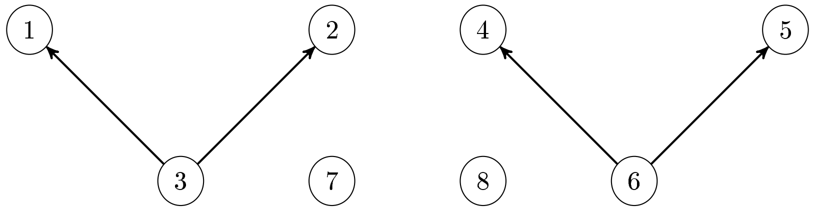

Figure 5.2: Multi-DAG

DAG is used throughout this chapter and the book, with the intention that it means a multi-DAG, or a collection of DAGs.

5.1.2 Scheduling Algorithm

The DAG is used to represent the task dependencies and an algorithm is needed to traverse the DAG to determine the next available task. The standard first-thought answer to this part of the problem is to use a topological sort[2] to determine the order; given a DAG, a topological sort will provide a valid ordering of the tasks. For any given DAG, there may be multiple correct orderings, any one of which is valid with respect to scheduling. For example, the DAG in Figure 5.1 has a possible ordering of: 1, 2, 3, 4, 5, 6. It also has an alternative, and just as correct, ordering of: 3, 1, 2, 5, 4, 6.

For a single processor system, this is easy enough, simply compute the ordering and feed in that order. However, we are concerned with scalability over a number of processing cores. Therefore, additional insight is necessary to not only build a valid framework mechansim, but also consider application level concerns for task dependencies.

Consider a system with three CPU cores, and also assume equal computational effort for each of the tasks. For Figure 5.1 the ordering of the tasks looks like: (1, 2, 3), (4, 5, x), and (6, x, x). In the first step, all three CPU cores are busy, in the second step, only two of the CPU cores are busy because task 6 has to wait for tasks 4 and 5 to complete. Finally, in the third step, only one CPU core is busy. Again, this is important to understand not only for the framework in how it responds to worker thread requests for something from the work queue, but also for application level design, ensuring there aren't too many dependencies between tasks, resulting in low system utilization.

There is still an additional level of complexity that exists in an operational system. In an interactive (and most others) application, the DAG is dynamic, with tasks coming and going throughout the application lifetime, in addition to tasks having differing levels of computational complexity. For example, going back to the example from Figure 5.1, let's say that two more tasks, task 7 and 8, show up while tasks 1, 2, and 3 are being computed. The overall ordering would then look like: (1, 2, 3), (4,

[2]http://en.wikipedia.org/wiki/Topological_sorting

5, 7), and (6, 8, x). Notice that task 7 is computed before task 6, even though it arrived afterwards. A mix of dependent and independent tasks offer the best opportunity for scalability, while also increasing the complexity of the scheduling framework.

The standard topological sorting will correctly handle the ordering of the queue at any time we wish to take a snapshot for a full connected DAG. However, it doesn't handle operational situations with tasks coming and going, some issues with respect to concurrency, or multi-DAGs. What is needed is an algorithm that provides topological-like sorting, but in the context of a dynamic multi-DAG.

The answer to the problem involves two modifications. The first is to keep track of tasks that have been removed from the queue, but have not yet completed execution. This is required to ensure tasks that depend upon removed but not completed tasks are not selected for execution. The second is to take the next available item from the DAG, rather than computing the ordering of all items in the queue. This requires the algorithm check to ensure the task has no dependencies on other tasks awaiting execution, but also that it has no dependencies on tasks that have been removed but have not yet completed execution. The result is still a topological-like sorting, but also correctly manages the additional complexity of an operational context.

5.2 Building Blocks

This section of the chapter walks through the implementation, highlighting the changes needed to transform the framework code from Chapter 3 into one that supports task dependencies. The changes to the task scheduling framework are moderate, and involve changing the interface through which tasks are added. The changes to the application are similarly moderate, involving changes to the Mandelbrot tasks, along with changing how the application adds tasks for execution. The updated interactive Mandelbrot viewer is detailed in Section 5.3.

5.2.1 Directed Acyclic Graph

As with the introduction of priority to the framework, the most interesting part of the changes introduced with this chapter is the implementation of the DAG; noting the implementation is that of the multi-DAG described earlier. There is no ready-made standard library data structure available, instead it is necessary to build a custom data structure. Rather than starting from scratch, and keeping with the theme of this book in taking advantage of the C++ language and framework, a data structure is composed using existing standard library components. Specifically, the `std::unordered_map` and `std::unordered_set` are used to perform a lot of heavy lifting, which signficantly reduces the amount of custom code. The name of the new data structure is `ConcurrentDAG`. Like the data structures introduced in the previous two chapters, as its name suggests, it is thread-safe. The remainder of this section details the implementation.

Task Identification & Ordering

With the ability to define dependencies between tasks, some way is needed to uniquely identify a task. Additionally, the scheduling system needs to have a way of knowing the time-based ordering of tasks. In order to achieve this, I have chosen to add an `uint32_t` identifier to the `Task` class. The revised `Task` declaration and constructor implementation are found in Listing 5.1.

Listing 5.1: Revised `Task` Class

```
class Task
{
public:
  explicit Task(std::function<void()> onComplete);
  virtual ~Task() {}
  uint32_t getId() const { return m_id; }
  virtual void execute() = 0;
  void complete()        { if (m_onComplete) { m_onComplete(); } }

protected:
  uint32_t m_id;

private:
  std::function<void ()> m_onComplete;
};

Task::Task(std::function<void ()> onComplete) :
  m_onComplete(onComplete)
{
  static uint32_t currentId = 1;
  static std::mutex myMutex;

  std::lock_guard<std::mutex> lock(myMutex);

  m_id = currentId++;

  if (currentId == std::numeric_limits<uint32_t>::max())
  {
    currentId = 1;
  }
}
```

The key part of this class is the technique used to assign the unique id. A `static currentId` is intially assigned a value of `1`. The first `Task` instance is assigned this value, then the `currentId` is incremented. This serves two purposes. The first is that each task is assigned a unique identifier. Secondly, because of the linearly increasing values, an implicit time-based ordering of the tasks is provided. Notice that a `std::mutex` is used to protect the assignment and update of the `currentId`; this ensures the code is thread-safe, meaning that mutliple threads can safely create instances of this class and expect that unique ids are generated.

Even though the `ConcurrentDAG` class is written as a template type, for the purposes of this book, the type is only ever expected to be a `std::shared_pointer<Task>`. Specifically the `ConcurrentDAG` expects the template type `T` to have a `getId()` method. This method is expected to return a unique identifier for each instance stored in the data structure.

Data Members

Several pieces of data have to be tracked in order to construct the DAG functionality. The declaration for these data members is found in Listing 5.2. The first, `m_nodes`, is a set of all tasks either waiting for execution or in execution but pending completion. The `m_nodes` member is a `std::map` in order to return items in sorted (i.e. time-based) order as they are dequeued; all other `maps` are `std::unordered_maps` for best possible performance. Next, `m_inUse`, is the set of tasks that have been selected for execution, but have not yet completed; the tasks in this container are still references in the `m_nodes` container. A task may have other tasks depending upon it for completion before they can execute. A container for tracking these dependencies is needed; this is sometimes known as an adjacency list. These dependencies are identified in the `m_adjacent` container. Finally, when looking for the next task available for computation another container, `m_reference`, is needed to help speed up the search for tasks that have no references in the `m_adjacent` container. How these data structures are used is highlighted as the class methods are discussed later in the chapter.

Listing 5.2: `ConcurrentDAG` Data Members

```
std::map<uint32_t, Node> m_nodes;
std::unordered_set<uint32_t> m_inUse;
std::unordered_map<uint32_t, std::unordered_set<uint32_t>> m_adjacent;
std::unordered_map<uint32_t, std::unordered_set<uint32_t>> m_reference;
```

Adding Tasks

There are two contexts in which tasks are added to the DAG. The first is adding a task that has no dependencies, the second is adding a group of tasks that have dependencies within that group. Although it is easily argued that adding a single task is a subset of adding a group, in practice it turns out to be better to write code to handle these cases independently. Another important consideration is that adding a group of tasks with dependencies must be handled as an atomic operation; no other tasks can be added or removed while the group is added.

Because of the need to ensure all tasks in a group are added atomically, the concept of a group operation is included as part of the `ConcurrentDAG`. This is provided through two simple methods that must be used to wrap the adding of the tasks. The names of these methods are `beginGroup` and `endGroup`, their implemention is shown in Listing 5.3.

Listing 5.3: Group Operations

```
void beginGroup()     { m_mutex.lock(); }
void endGroup()       { m_mutex.unlock(); }
```

The implementation for these methods is trivial, but important. They obtain a lock on a mutex; noting all other operations on the `ConcurrentDAG` require obtaining a lock on this mutex. The type of mutex used is a `std::recursive_mutex`. This type of mutex allows a thread that has already obtained a lock on a mutex to pass through when it tries to obtain it again. In other words, a thread can call `beginGroup`, which obtains the mutex lock, then make another call that requires the mutex lock and pass right through, while other threads are blocked on the same methods. The use of the mutex not only ensures thread safety, it also ensures the thread that obtained the lock is the only one able to add new tasks.

Therefore, when adding a group of related tasks, the `beginGroup` method is called first, the tasks are added, then `endGroup` is called. Between these two calls, tasks are added through the `addEdge` method. The code for this method is shown in Listing 5.4. The method is named `addEdge` because it is not only adding tasks to the DAG, but is also adding an edge (a dependency) between the tasks. The `source` parameter identifies the task that must be executed before the task identified by the `dependent` parameter.

Listing 5.4: Adding Tasks with Dependencies

```
void addEdge(T source, T dependent)
{
  std::lock_guard<std::recursive_mutex> lock(m_mutex);

  m_nodes[source->getId()] = source;
  m_nodes[dependent->getId()] = dependent;

  if (m_adjacent.find(source->getId()) == m_adjacent.end())
  {
    m_adjacent[source->getId()] = std::unordered_set<uint32_t>();
  }
  m_adjacent[source->getId()].insert(dependent->getId());

  if (m_reference.find(dependent->getId()) == m_reference.end())
  {
    m_reference[dependent->getId()] = std::unordered_set<uint32_t>();
  }
  m_reference[dependent->getId()].insert(source->getId());
}
```

The first step in the method is to grab the recursive mutex. In the case of a group operation, the thread will already own the lock from a call to `beginGroup` and pass through. Next, both the `source` and `dependent` tasks are added to the container that tracks all tasks currently in the DAG. Because `m_nodes` is a `std::map`, if the task is already in the container it isn't duplicated, it simply overwrites itself; this is faster than searching and then adding if it doesn't exist.

The next step in `addEdge` is to add the `dependent` task as an *adjacent* task to the `source`. Adjacent tasks are those that depend upon the `source` to complete before they can be dequeued for use. Finally, the last step is to add the `source` task to a `m_reference` container. This container allows for quick lookup, when dequeing the next available task, to determine if the candidate task is *referenced* as a dependent to another task. By having each `dependent` task maintain a set of `source` tasks upon which it depends, it makes the search trivial, rather than having to look for the task in all other `source` tasks.

Adding a single task to the DAG is trivial, as compared to adding two. The first difference is that it is not necessary to invoke the `beginGroup` and `endGroup` methods. The second, is the code for adding a single node is requires only that the node is added to the master list of tasks and setting its adjaceny list to an empty set. The code for the `addNode` method is shown in Listing 5.5.

Listing 5.5: Adding a Single Task

```
void addNode(T one)
{
  std::lock_guard<std::recursive_mutex> lock(m_mutex);

  m_nodes[one->getId()] = one;
  m_adjacent[one->getId()] = std::unordered_set<uint32_t>();
}
```

Dequeueing & Removing Tasks

A task goes through a two-step dequeue and removal process before it is completely eliminated from the DAG. The first step is to dequeue the task for execution. At this point, the task is identified as *in use* (i.e., being executed), but still considered part of the DAG so as to prevent dependent tasks from being dequeued for execution. The second step is the final removal of the task from the DAG upon completion. In this step the task is fully removed from the DAG, allowing dependent tasks to be dequeued.

The code for dequeuing a task from the DAG is shown in Listing 5.6. As with all other methods, the first step in `dequeue` is to grab a lock on the mutex. Once the lock is obtained, the list of all tasks is searched for a task that is not dependent upon any other tasks or any tasks currently in use. The search of all tasks isn't a random search, it is in order by the oldest tasks. This happens by design through the use of a `std::map`, which is a binary search tree (`std::unordered_map` is an unordered hash table). The ordering of the tasks in `m_nodes` is based upon the `Task::m_id` member, which is an `uint32_t` that is assigned an increasing value based upon when it was created.

Listing 5.6: Dequeing a Task

```
T dequeue()
{
  std::lock_guard<std::recursive_mutex> lock(m_mutex);
  T item = nullptr;
  bool found = false;
  for (const auto& candidate : m_nodes)
  {
    auto itr = m_reference.find(candidate.first);
    if ((itr == m_reference.end() ||
      itr->second.size() == 0) &&
      m_inUse.find(candidate.first) == m_inUse.end())
    {
      found = true;
      item = candidate.second;
      m_inUse.insert(candidate.first);
      break;
    }
  }

  return item;
}
```

It is easy to determine if a task is dependent upon another task by checking the `m_reference` data structure. If the task is not found in the data structure, or is found but has a size of 0, then it isn't reliant upon a task not yet dequeued. However, it may still be dependent upon an in use task. This is also easily determined by looking for a reference to it in the `m_inUse` set. If the `candidate` task is found to not be dependent upon any other tasks, it is assigned to the local `item` variable, added to the `m_inUse` set, `found` set to `true`, and the search loop ended. Because the `Task` information is no longer needed, only the `Task::m_id` (stored as the key in the `candidate.first` member) is stored in the `m_inUse` set. In the case no task is found that meets the criteria, or none exist at all, the `item` variable remains set to the `nullptr` and the method returns.

When a task has completed execution, its needs to be completely removed from the DAG. The method for doing this is called `finalize` and is show in Listing 5.7. This method first performs a quick check to see if the task being finalized actually exists in the `m_inUse` set. This is only done as an `assert` so that program logic errors can be discovered, release builds will not have this check. Next, the code goes through all of its adjacent tasks and removes references to itself from those tasks. This removes the dependency from this task to any other task in the DAG. The code then cleans up the `m_adjacent` set by erasing it. Finally, the task itself is removed from both the master set of tasks and the in use tasks. When this method exits, the task is no longer a part of the DAG, and any tasks previously dependent upon its completion are now free to be dequeued.

Listing 5.7: Finalizing a Task

```
void finalize(const T& node)
{
  std::lock_guard<std::recursive_mutex> lock(m_mutex);

  assert(m_inUse.find(node->getId()) != m_inUse.end());

  for (auto id : m_adjacent[node->getId()])
  {
    m_reference[id].erase(node->getId());
  }
  m_adjacent.erase(node->getId());
  m_nodes.erase(node->getId());
  m_inUse.erase(node->getId());
}
```

5.2.2 Thread Pool

The public interface to the `ThreadPool` is changed from both the previous chapters. The changes to the interface are made to support the ability to add tasks that depend upon other tasks. The interface exposed by the `ThreadPool` matches that of the `ConcurrentDAG` as detailed in Section 5.2.1. Listing 5.8 shows the revised `ThreadPool` class declaration.

Listing 5.8: `ThreadPool` Declaration

```
class ThreadPool
{
public:
  static std::shared_ptr<ThreadPool> instance();
  static void terminate();

  void beginGroup()     { m_taskDAG.beginGroup(); }
  void endGroup()       { m_taskDAG.endGroup(); }
  void enqueueTask(std::shared_ptr<Task> source);
  void enqueueTask(
    std::shared_ptr<Task> source,
    std::shared_ptr<Task> dependent);

protected:
  ThreadPool(uint16_t sizeInitial);

private:
  static std::shared_ptr<ThreadPool> m_instance;
  friend WorkerThread;

  std::vector<std::shared_ptr<WorkerThread>> m_threadQueue;
  ConcurrentDAG<std::shared_ptr<Task>> m_taskDAG;
  std::condition_variable m_eventDAG;

  void taskComplete();
};
```

The application interface to the `ThreadPool` is nearly identical to the `ConcurrentDAG`, because it is essentially a pass-through wrapper around the `ConcurrentDAG`. The `beginGroup` and `endGroup` methods simply invoke the underlying `m_taskDAG` methods. The two `enqueueTask` methods invoke the `addNode` and `addEdge` methods of the `m_taskDAG`. Additionally, these `enqueueTask` methods signal the `m_eventDAG` to ensure a worker thread is released so the task can potentially be dequeued from the DAG. The code for these two methods is found in Listing 5.9.

Listing 5.9: `ThreadPool::enqueueTask` Methods

```
void ThreadPool::enqueueTask(std::shared_ptr<Task> source)
{
  m_taskDAG.addNode(source);
  m_eventDAG.notify_one();
}

void ThreadPool::enqueueTask(
  std::shared_ptr<Task> source,
  std::shared_ptr<Task> dependent)
{
  m_taskDAG.addEdge(source, dependent);
  m_eventDAG.notify_one();
}
```

The one new structural addition to the `ThreadPool` is the `taskComplete` method. The implementation of this method is found in Listing 5.10. The purpose of this method is to allow `WorkerThreads` to inform the `ThreadPool` when a task has completed execution. On completion of a task, more than one dependent task may become available for computation. Therefore, multiple worker threads must be signaled to that any available tasks are grabbed for computation.

Listing 5.10: `ThreadPool::taskComplete` Method

```
void ThreadPool::taskComplete()
{
  m_eventQueue.notify_all();
}
```

5.2.3 Worker Thread

Worker threads continue to have the same fundamental purpose, as used in Chapter 3, watch the task queue and grab the next available task. The change this time is that the tasks come from a DAG, which may describe complex inter-relationships between the tasks, which affect the order in which they are removed.

The revised constructor is shown in Listing 5.11. The constructor returns to looking much like the `WorkerThread` from Chapter 3. The constructor accepts parameters that provide references to the `ConcurrentDAG` which contains the tasks, along with the `std::condition_variable` used to signal when tasks are added to the DAG. The body of the constructor creates the thread associated with the worker.

Listing 5.11: `WorkerThread` Constructor

```
WorkerThread::WorkerThread(
    ConcurrentDAG<std::shared_ptr<Task>>& taskDAG,
    std::condition_variable& taskDAGEvent) :
  m_taskDAG(taskDAG),
  m_eventTaskDAG(taskDAGEvent),
  m_done(false),
  m_thread(nullptr)
{
  m_thread = std::unique_ptr<std::thread>(
    new std::thread(&WorkerThread::run, this));
}
```

The `run` method has changed somewhat to support the way tasks are represented in the DAG. The revised method is shown in Listing 5.12. The only difference in this method versus the one in Chapter 3 is the addition of two statments. The first is `m_taskDAG.finalize(task)`. This statement tells the DAG to remove any reference of the task from the DAG. Remember that DAG-based tasks have a two-step use case: they are first dequeued, then finalized. The other additional statement is `ThreadPool::instance()->taskComplete()`. This statement informs the `ThreadPool` a task has just finished execution, causing it to signal all worker threads so that all dependent tasks possibly released are now grabbed and executed.

Listing 5.12: `WorkerThread::run` Method

```
void WorkerThread::run()
{
  while (!m_done)
  {
    std::shared_ptr<Task> task = m_taskDAG.dequeue();
    if (task != nullptr)
    {
      task->execute();
      task->complete();
      m_taskDAG.finalize(task);
      ThreadPool::instance()->taskComplete();
    }
    else
    {
      std::unique_lock<std::mutex> lock(m_mutexEventTaskDAG);
      m_eventTaskDAG.wait(lock);
    }
  }
}
```

5.3 Application Task Dependencies

In addition to revising the framework to include the concept of task dependencies, the interactive Mandelbrot viewing application is also revised to utilize the ability to define task dependencies. This is done by defining a new `MandelFinishedTask` that is dependent upon the `MandelPartTask` tasks. The prime number task remains the same, as only one is ever in existence at any time. The `MandelImageTask` is now removed, with its purpose no longer necessary with the new task depencency framework in place. The other revision to the application code is that tasks are now added using the new `ThreadPool` interface.

The new `MandelFinishedTask` class is shown in Listing 5.13. Interestingly, this task doesn't actually do anything, so what is its purpose? The purpose is to act as a kind of signal to the application code that a new Mandelbrot image has completed and it is okay to copy those pixels into the image currently being rendered. The way this works, is to make this task dependent upon all of the `MandelPartTasks`. When those tasks have all finished, the `MandelFinishedTask` is removed from the DAG and executed.

Listing 5.13: `MandelFinishedTask`

```
class MandelFinishedTask : public Task
{
public:
  MandelFinishedTask(std::function<void ()> onComplete) :
    Task(onComplete)
  {
  }

  virtual void execute() {}
};
```

Still, the question remains, how does this signal the application, the task doesn't do anything! The answer to the question is in the revised `Mandelbrot::startNewImage` method, shown in Listing 5.14. The first part of the method creates an instance of a `MandelFinishedTask`, and this is the key, passes a lambda to the `onComplete` parameter that copies the newly computed pixels into the image currently being rendered. In other words, while this task performs no computation itself, its `onComplete` operation is copying the newly computed data into the rendered image.

Listing 5.14: `startNewImage` Method

```
void Mandelbrot::startNewImage()
{
  auto taskFinished = std::make_shared<MandelFinishedTask>(
    [this]()
    {
      copyToPixels();
      m_inUpdate = false;
    });

  ThreadPool::instance()->beginGroup();

  double deltaY = (m_mandelBottom - m_mandelTop) / m_sizeY;
  for (auto row : IRange<decltype(m_sizeY)>(0, m_sizeY - 1))
  {
    auto task = std::shared_ptr<MandelPartTask>(
      new MandelPartTask(
        m_image.get() + row * m_sizeX,
        m_sizeX, MAX_ITERATIONS,
        m_mandelTop + row * deltaY,
        m_mandelLeft, m_mandelRight,
        nullptr));

    ThreadPool::instance()->enqueueTask(task, taskFinished);
  }

  ThreadPool::instance()->endGroup();
}
```

The second part of the method, beginning with the `ThreadPool::instance()->beginGroup()` statement, prepares the tasks that perform the Mandelbrot image computation. The loop goes through each row in the image and creates a `MandelPartTask` for that row. As the task is added to the `ThreadPool`, the `taskFinished` is identified as being dependent upon it. After all of the row tasks are added, a call to `endGroup` is made, telling the `ThreadPool` the current group of dependent tasks is complete. With these task dependencies set up, the scheduling framework takes over and ensures all of the `MandelPartTask`s are complete, before the `MandelFinishedTask` is executed.

5.3.1 Further Demonstration

The demonstration application provided as part of this chapter includes an additional task and application code that provides a better run-time demonstration of the use of task dependencies. A new task named `SimpleTask` is added that waits for a specified amount of time. A new function named `startChapterDemo` is added that creates a non-trivial set of task dependencies that match those found in Figure 5.1.

The code for the new task is found in Listing 5.15. The task accepts an `uint16_t`, used to identify the task for display. When the task is executed it goes to sleep for 3000 milliseconds, then shows its name. The only purpose of this task is to aid in demonstrating the order in which tasks are executed.

Listing 5.15: `SimpleTask`

```
class SimpleTask : public Task
{
public:
  SimpleTask(uint16_t name) :
    Task(nullptr),
    m_name(name)
  {}

  virtual void execute()
  {
    std::this_thread::sleep_for(std::chrono::milliseconds(3000));
    std::cout << m_name << std::endl;
  }

private:
  uint16_t m_name;
};
```

The code for the new `startChapterDemo` is found in Listing 5.16. This method creates a set of `SimpleTask`s that match those found in Figure 5.15. The first part of the method creates the tasks and identifies them by their number. Finally, the method adds the tasks to the `ThreadPool`, defining the various dependencies among the tasks. Task 2 is referenced twice in order to define the relationship between it and tasks 4 and 5. Also, tasks 4 and 6 are referenced multiple times in order to define their dependencies upon more than one parent tasks. Remember that even though a task is referenced more than once doesn't mean it is computed multiple times, the `ConcurrentDAG` ensures a task is added only once, regardless of the number of times it is referenced.

Listing 5.16: `startChapterDemo` Method

```
void ScalabilityApp::startChapterDemo()
{
  ThreadPool::instance()->beginGroup();

  auto task1 = std::make_shared<SimpleTask>(1);
  auto task2 = std::make_shared<SimpleTask>(2);
  auto task3 = std::make_shared<SimpleTask>(3);
  auto task4 = std::make_shared<SimpleTask>(4);
  auto task5 = std::make_shared<SimpleTask>(5);
  auto task6 = std::make_shared<SimpleTask>(6);

  ThreadPool::instance()->enqueueTask(task1, task4);
  ThreadPool::instance()->enqueueTask(task2, task4);
  ThreadPool::instance()->enqueueTask(task2, task5);
  ThreadPool::instance()->enqueueTask(task3, task4);
  ThreadPool::instance()->enqueueTask(task4, task6);
  ThreadPool::instance()->enqueueTask(task5, task6);

  ThreadPool::instance()->endGroup();
}
```

When the program is run, the first three tasks, 1, 2, and 3 are executed in parallel (if enough CPU cores). Once task 2 has completed, task 5 becomes available and begins computation. When all three of tasks 1, 2, and 3 complete, task 4 is released and begins computation. Finally, when tasks 4 and 5 complete, task 6 is released and begins computation. On my computer, tasks 1, 2, and 3 all compute in parallel, when they are finished, 4 and 5 compute in parallel, and finally task 6 executes. This code shows how a non-trivial set of relationships between tasks is easily expressed in application code, and the hard work of scheduling the tasks is left to the underlying scheduling framework.

5.4 Summary

This chapter introduced the concept of allowing dependencies between tasks through the use of a DAG-based scheduling framework. This provides an extremely powerful tool for an application developer through simplified application code by placing the algorithmic complexity in the scheduling framework. To achieve this, the underlying framework required a number of changes, most noteworthy of which is the introduction of a DAG-based data structure used for determing the order in which tasks can be removed for computation. Having this new capability simplified the application code, while also making better use of the thread pool framework.

6 | Networking Techniques

This chapter discusses the networking techniques used for the distributed applications presented in the next chapters. The foundation for the network communication comes from the Boost (http://www.boost.org) libraries, introduced in Section 6.1. The Boost libraries are used in combination with the new C++11 threading and lambda features to form clean solutions for the networking code.

The choice to use Boost is driven by its cross-platform approach, the well designed interfaces, along with its developer friendly license. A testament to its design and utility is the large number of libraries that have made it into the C++11, and C++14, standards. In particular, the networking API (Boost.Asio) is well designed, easy to use, and works well across a number of different compiler and OS platforms.

The networking techniques presented in this chapter and used by the demonstration applications detailed in this book have worked well for my research and work. This chapter is not a comprehensive discussion of Boost.Asio, only those techniques directly applied throughout the rest of the book are discussed. There are any number of designs and philisophies that may be taken, owing to the needs of the application being developed and to personal preferences. Consider the networking technique used in this book as a recommendation, but not necessarily one that fits the needs for every application and developer.

6.1 Boost Introduction

Boost is a set of cross-platform C++ libraries that provide a broad range of capabilties. In fact, many of these capabilities have found their way into C++11 language and standard libraries, including smart pointers, threading, and lambda functions. Of primary interest for this book is the low-level networking API Boost.Asio, or simply Asio. Goals for the Boost.Asio library include *Portability*, *Scalability*, *Efficiency*, and *Ease of use*. Boost.Asio provides an asynchronous I/O abstraction for things like serial ports, file descriptors, and networking. The focus in this chapter, and the book, is on its use for networking.

The Boost set of libaries is friendly for use by both commercial and non-commercial organizations, provided for by its own license, the Boost Software License[1]. The license is BSD like, certified *Open Source* by the Open Source Initiative, comes *as is*, allows for the use and modification of the Boost libraries, does not require the license is distributed with binary or executable uses of the libraries, along with not requiring the source is made available external to the organization.

The Boost libaries are updated several times a year to include new capabilities, improve efficiency, and resolve outstanding bugs; at the time of this writing the current version is 1.57. A large number of the commonly used libraries are mature and stable, and care is taken to maintain backwards compatibility. Additionally, the library is verified against a large number of C++ compilers and operating system platforms. This gives developers confidence their code won't break with each new library update.

[1]http://www.boost.org/LICENSE_1_0.txt

6.2 Boost Asio

Boost.Asio is the library upon which networking I/O operations are built. It supports both synchronous and asynchronous operations in single and multi threaded applications. In addition to networking operations, other capabilities are exposed, such as timers, signal handling, and even SSL operations. The focus in this chapter, and the book, is on the networking operations.

The Boost.Asio library provides a model of its own that must be understood to take full advantage of its capabilities; it is not simply a thin wrapper around the *sockets* API. This chapter provides a solid introduction to the library and the way in which I have found to work well for my application development style, but it is strongly recommended to refer to the Boost.Asio online documentation[2] for a complete description of its design, rationale, and use.

6.2.1 Asio io_service

The foundation for Boost.Asio operations is the `io_service`. Understanding its role and operation is essential for developing correct and efficient networking applications. Every application must instantiate an `io_service` through which all operations are managed. The `io_service` maintains a queue of operations that it needs to service. The operations on this queue are only serviced by the thread (or threads) that call its `.run` method. The `.run` method blocks until all these operations have completed. Using the C++11 lambda and threading features, it is trivial to create a worker thread that is dedicated to the `io_service` queue, leaving the main application thread free to work on other tasks. The code in Listing 6.1 illustrates how this may be done.

Listing 6.1: io_service Thread

```
boost::asio::io_service ioService;
std::thread ioThread = std::thread(
  [&ioService]()
  {
    ioService.run();
  });
```

The first statement instantiates the `io_service`, causing it to come to life. The next statment invokes a thread, using a lambda to define the thread function, and calls the `.run` method of the `io_service`. This code will correctly create an `io_service` and create a thread to service any queued operations. Note that the `.run` method will fall through (unblock) when there are no more queued operations. Therefore, this code will immediately fall through, closing down the `io_service` thread, which is something we do not want.

The solution to the above problem is to create an `io_service::work` object which prevents the `.run` method from unblocking until the `io_service` is stopped. The purpose of this object is to inform the `io_service` it has work remaining to do, preventing it from falling through the `.run` method until `.stop` is called on the `io_service`. The code in Listing 6.2 demonstrates the use of the `io_service::work` object.

Listing 6.2: io_service::work Object

```
boost::asio::io_service ioService;
boost::asio::io_service::work work(ioService);

std::thread ioThread = std::thread(
  [&ioService]()
  {
    ioService.run();
```

[2]http://www.boost.org/doc/libs/1_57_0/doc/html/boost_asio.html

```
    std::cout << "Completed ioService." << std::endl;
  });

std::cout << "Sleeping for a couple of seconds..." << std::endl;
std::this_thread::sleep_for(std::chrono::milliseconds(2000));
std::cout << "Stopping the ioService..." << std::endl;

ioService.stop();
ioThread.join();
```

In this code an `io_service::work` object is created and initialized with a reference to the `ioService` instance. Inside the thread function (a lambda) the `ioService.run();` statement blocks until the `ioService.stop();` statement in the main thread is executed. A sleep operation is executed to demonstrate that the thread does not complete until the `ioService` object is told to stop. Following the `ioService.stop();` statement the `ioService` thread is joined and then the code is allowed to continue.

Using this approach, a separate worker thread is created and used to service all operations placed on the `io_service` queue, with the worker thread staying active until the `io_service` is specifically stopped. The next section, Section 6.2.2 discusses how to place operations on the queue.

6.2.2 The `io_service` Queue

The `io_service` object maintains a queue onto which requests are placed and from which they are serviced; it is a first in, first out (FIFO) queue. Requests get placed on this queue implicitly through the use of some of the Boost.Asio API calls, while others are explicitly placed on the queue through the use of the `.post` method. Most of the code presented in this book explicitly places requests using `.post`, therefore, the focus in this chapter is on its use and behavior.

The code in Listing 6.3 shows how to post a request to an `io_service` queue. The `.post` method expects a `CompletionHandler`, which is function whose signature accepts no parameters and has a `void` return. In this example, the function is provided by defining a lambda.

Listing 6.3: Posting a Request

```
ioService.post(
  []()
  {
    std::cout << "This runs on the io_service thread.";
  });
```

The main application thread, or really any application thread, can post to the `io_service` queue, such as the one in Listing 6.3. Once the `io_service` thread works its way through the items ahead of our request, it will invoke the `CompletionHandler`. In this example, `This runs on the io_service thread.` is output to the console.

Listing 6.4 shows a full example of defining a dedicated thread for an `io_service`, along with posting several requests to its queue. This example starts by reporting the thread ids for the main application thread and the `io_service` thread to help establish which threads are executing which code. Each lambda posted to the `io_service` reports the thread from which it is executed, in both cases they report the same thread id as the `io_service` reported.

Listing 6.4: Multiple Requests

```
void ioServiceQueue()
{
  boost::asio::io_service ioService;
  boost::asio::io_service::work work(ioService);
  std::cout << "main thread id: ";
  std::cout << std::this_thread::get_id() << std::endl;

  std::thread ioThread = std::thread(
    [&ioService]()
    {
      std::cout << "io_service thread id: ";
      std::cout << std::this_thread::get_id() << std::endl;
      ioService.run();
      std::cout << "io_service terminated" << std::endl;
    });

  auto notify =
    []()
    {
      std::cout << "This post executed on: ";
      std::cout << std::this_thread::get_id() << std::endl;
    };

  ioService.post(notify);
  ioService.post(notify);

  ioService.stop();
  ioThread.join();
}
```

The Boost.Asio `io_service` queue is fairly easy to understand when seen in action like this. Unfortunately, this is about as simple as it gets, most non-trivial applications that utilize network communication require a more sophisticated usage. The next two sections, Section 6.3 and Section 6.4 provide additional stepping stones to building the necessary understanding.

6.3 Boost Asio Thread Pools

To develop a truly scalable application, we'll need more than a single background thread servicing requests on an `io_service` queue. By definition, a *scalable* application should grow to utilize the available system resources. In the case of the system I'm using to develop the code samples for this book, it has 6 hyper-threaded CPU cores, giving the appearance of 12 CPU cores. A scalable application will take advantage of all the cores on my system, and all of the cores on any other system on which it is run. Thankfully, Boost.Asio provides easy support for *Thread Pools* on the `io_service`.

In order to have more than one thread service requests from the `io_service` queue, simply create more threads that call the `io_service .run` method. Listing 6.5 shows an example of creating a thread pool on a single `io_service`. Through the use of this technique, multiple requests can be serviced in parallel on a multi-core system.

Listing 6.5: `io_service` Thread Pool

```
void threadedIoService()
{
  boost::asio::io_service ioService;
  boost::asio::io_service::work work(ioService);

  std::cout << "main thread id: ";
  std::cout << std::this_thread::get_id() << std::endl;

  std::vector<std::thread> threads;
  for (auto thread : IRange<uint8_t>(1, std::thread::hardware_concurrency()))
  {
    threads.push_back(std::thread(
      [&ioService]()
      {
        ioService.run();
      }));
  }

  auto notify =
    []()
    {
      std::cout << "This post executed on: ";
      std::cout << std::this_thread::get_id() << std::endl;
    };

  ioService.post(notify);
  ioService.post(notify);

  ioService.stop();
  for (auto& thread : threads)
  {
    thread.join();
  }
}
```

6.4 Boost Asio Strands

Often times it is necessary to queue the requests for a single object to ensure they happen sequentially, while still allowing requests on other objects to be performed in parallel. Consider that the send operations on a socket can not be interleaved, one must complete before the next is started. We'd like to write code that makes non-blocking write requests, but also ensure that the requests are performed in order and that each request is completed before the next one is allowed to begin. In the normal free-for-all world of thread pools and the `io_service` object, that isn't guaranteed. Another synchronization capability is needed, one that allows requests on one object to be queued and completed in order while still participating in the context of a thread pool. That capability is provided by `io_service::strand`, Boost strands.

A strand can be thought of as an independent synchronization queue within an `io_service`. Each `io_service::strand` has its own request queue to which service requests are posted. Even though a strand has its own request queue, it is still associated with an `io_service`, passed in as a parameter to the strand constructor. Posts are made directly to the strand, rather than to the `io_service`. The `io_service::strand` ensures that all post requests to a strand will *not* execute concurrently, instead requests are serviced in the order they are posted and each request is completed before the next is started. Note that requests on multiple strands may execute concurrently, it is only requests on each individual strand are executed sequentially.

Strands are illustrated through a `Countdown` class that is updated within a thread pool, using an `io_service::strand` to guarantee the updates are correctly synchronized. All the `Countdown` class does is post a bunch of requests to its own strand, with the requests updating an internal counter and reporting the value until the count reaches 0. The full `Countdown` class is shown in Listing 6.6.

Listing 6.6: `Countdown` Class

```
class Countdown
```

```
{
public:
  Countdown(boost::asio::io_service& ioService,
      std::string name,
      uint16_t startCount) :
    m_strand(ioService),
    m_name(name),
    m_count(startCount)
  { }

  void start()
  {
    for (auto handler = 0; handler < m_count; handler++)
    {
      m_strand.post([&]() { next(); });
    }
  }

private:
  boost::asio::strand m_strand;
  std::string m_name;
  uint16_t m_count;

  void next()
  {
      std::cout << "Thread id: ";
      std::cout << std::this_thread::get_id();
      std::cout << " Name: " << m_name;
      std::cout << " - Count: " << m_count << std::endl;
      m_count--;
  }
};
```

This class defines a private `strand`, which is associated with the `io_service` in the class constructor. It also contains a `name`, used to identify the instance in console output, and a `count`, used to track how much longer to continue the countdown. The public method `start` is called by an application thread to kick off the countdown and reporting for the instance. When invoked, the `start` method posts `m_count` handlers (lambdas) to the object's `strand`, rather than to an `io_service`. As each request is invoked by a thread in the thread pool on the `io_service`, the handler reports which thread it is executing on, along with reporting the instance name, and the current value of the counter. The final statement in the hanlder is to subtract one from the value of `m_count`.

The application code that demonstrates the use of the `Countdown` class is found in Listing 6.7.

Listing 6.7: `Countdown` Application

```
int main()
{
  boost::asio::io_service ioService;
  boost::asio::io_service::work work(ioService);

  std::vector<std::thread> threads;
  for (auto thread : IRange<uint8_t>(1, std::thread::hardware_concurrency()))
  {
    threads.push_back(std::thread(
      [&ioService]()
      {
        ioService.run();
      }));
  }

  Countdown obj1(ioService, "One", 10);
  Countdown obj2(ioService, "Two", 12);

  obj1.start();
  obj2.start();

  for (auto& thread : threads)
  {
    thread.join();
  }

  return 0;
}
```

The first part of the code is familiar by now, the definition of an `io_service` along with a thread pool to service requests. This is followed by two instances of the `Countdown` class being created and started. Sample output from a run of this program is shown in Figure 6.8. I've modified the output a little bit to clean up an issue from multiptle threads writing to the console (which is not synchronized) concurrently, but only for readability, the meaning is not changed.

Figure 6.8: `Countdown` Output

```
Thread id: 7788 Name: One - Count: 10
Thread id: 6652 Name: Two - Count: 12
Thread id: 11252 Name: One - Count: 9
Thread id: 7788 Name: One - Count: 8
Thread id: 11252 Name: Two - Count: 11
Thread id: 6652 Name: One - Count: 7
Thread id: 7788 Name: Two - Count: 10
Thread id: 11252 Name: One - Count: 6
Thread id: 6652 Name: Two - Count: 9
Thread id: 7788 Name: One - Count: 5
Thread id: 11252 Name: Two - Count: 8
Thread id: 6652 Name: One - Count: 4
Thread id: 7788 Name: Two - Count: 7
Thread id: 11252 Name: One - Count: 3
Thread id: 6652 Name: Two - Count: 6
Thread id: 7788 Name: One - Count: 2
Thread id: 11252 Name: Two - Count: 5
Thread id: 6652 Name: One - Count: 1
Thread id: 7788 Name: Two - Count: 4
Thread id: 11252 Name: Two - Count: 3
Thread id: 7788 Name: Two - Count: 2
Thread id: 11252 Name: Two - Count: 1
```

A careful read through the output shows that each `Countdown` instance correctly (in-order) performs the countdown as intended. Note also that different thread ids are reported being used on the same `Countdown` instances. In fact, three different threads were used by the system to service the different requests.

The `io_service` and `strands` are a part of the Boost.Asio library, and are intended for use in synchronizing I/O operations, but as this code demonstrates, they can be used as a general synchronization resource. To clarify, this code uses the `io_service` as a thread pool, then uses `strands` as a synchronization object. No where in this code is a mutex defined and used for synchronization, that is taken care of by the Boost.Asio implementation.

6.5 Boost Asio Socket Connections

Sockets in Boost.Asio are familiar to anyone already experienced in regular sockets programming, only a little easier to use because of the simplied syntax and elimination of platform initialization dependencies. The sockets API in Boost.Asio provides for both synchronous and asynchronous programming models. The code presented in this chapter, and throughout the book, relies primarily on asynchronous techniques, with a few synchronous pieces where scalability is not an issue.

There are two fundamental components needed to understand socket programming in Boost.Asio. The first is how to make and receive a connection, the second is how to send and receive data. For the purposes of this section, a *server* is the application that receives a connection, and a *client* is the application that initiates a connection. The words *server* and *client* are only ways to refer to different application components. They are used in one way in this chapter, and differently in later parts of this book. Section 6.5.2 discusses making connections, and Section 6.6 discusses sending and receiving of data.

Take note that all the techniques described in this chapter, and throughout the book, utilize TCP communication. Many of the programming concepts and techniques are similar between TCP and UDP. While knowing the TCP programming model is helpful to understanding similar looking UDP API, UDP application development is a different model, one not covered in this book.

6.5.1 Boost `endpoint` and IPv4 & IPv6

Before getting into the details of making connections and sending data, it is important to discuss the Boost `endpoint` type. An `endpoint` describes an endpoint that can be associated with a socket. An `endpoint` constructor can accept an IP address and port for a known endpoint, alternatively, it can accept an `InternetProtocol` and port when waiting on an incoming connection. In the first case, the IP address can be either an IPv4 or IPv6 address. Similarly, in the second case, the `InternetProtocol` can specify either IPv4 or IPv6. Regardless of the address type to which an `endpoint` is bound, either IPv4 or IPv6, the Boost.Asio sockets API is used the same. This makes it possible for an application to easily move to IPv6 when necessary, or alternatively support both protocols at the same time.

6.5.2 Making a Connection

There are two participants in a connection, one that waits for an incoming connection (our server), and another that initiates a connection (our client). The server knows nothing about a client before the connection is made, it must wait for an incoming connection from any address; but on a specific port. The client, on the other hand, has knowledge of the server address and the port on which it is listening. These differences result in different programming techniques between the two applications.

Accepting a Connection

The server components include an `acceptor`, a `socket`, and an `endpoint`. The `acceptor`, as its name suggests, is used to accept incoming socket connections. A `socket` is the communication channel through which data is sent and received. Finally, the `endpoint`, as described in the previous section, provides an address (IP and port) to which a `socket` is bound. Listing 6.9 shows sample code used to wait for an incoming connection.

Listing 6.9: Acccepting a Connection

```
//
// Code segment that initiates the first handleNewConnection
{
...
  std::shared_ptr<ip::tcp::acceptor> acceptor =
    std::make_shared<ip::tcp::acceptor>(
      ioService,
      ip::tcp::endpoint(ip::tcp::v4(), 12345));

  handleNewConnection(ioService, acceptor);
...
}

void handleNewConnection(
    boost::asio::io_service& ioService,
    std::shared_ptr<ip::tcp::acceptor> acceptor)
{
  std::shared_ptr<ip::tcp::socket> socket =
    std::make_shared<ip::tcp::socket>(ioService);
  acceptor->async_accept(*socket,
    [&ioService, acceptor, socket]
    (const boost::system::error_code& error)
    {
      if (!error)
      {
        std::cout << "Received connection from: ";
        std::cout << socket->remote_endpoint();
        std::cout << " on thread id: ";
        std::cout << std::this_thread::get_id() << std::endl;
      }
      handleNewConnection(ioService, acceptor);
    });
}
```

An `acceptor` is initialized with an `io_service` along with an `endpoint` which indicates the protocol to use along with the port on which the server will listen for incoming connections. It is also necessary to create a default constructed `socket`; this is the socket into which the connection will be accepted.

The `acceptor`, like much of Boost.Asio, provides both synchronous and asynchronous interfaces for accepting connections. The synchronous method is `.acccept`, while the asynchronous method is `.async_accept`, as shown in Listing 6.9. The asynchronous method needs an `AcceptHandler` to call upon receipt of a connection, which is provided by the lambda. Only one acceptor should ever be created to listen on a port. Because of this, the sample code in 6.9 shows part of the code segment that initiates the first `handleNewConnection` by creating the `acceptor` and passing it in as a parameter for continued reuse.

The reason for capturing the `acceptor` and `socket` by value is to ensure they have a lifetime beyond the invocation of the `handleNewConnection` function. `.async_accept` is a non-blocking call, which means that `handleNewConnection` is also a non-blocking call. However, the lambda continues to exist, along with everything it has captured after `handleNewConnection` has exited, being invoked when a new connection is made. The `shared_ptr`s are captured by copy, using the smart pointer reference counting ensures they live for the life of the lambda, rather than going out of scope and releasing the managed objects when the `handleNewConnection` function exits.

Initiating a Connection

The client components include a `query`, a `resolver`, an `iterator`, and a `socket`. The `query` is used to find the server; in simple terms, an IP address and port. A `resolver` takes a `query` and determines the list of `endpoints` that result. The `iterator` is used to iterate over the list of `endpoints` returned by the `resolver`; a `query` may return more than one endpoint, making an `iterator` necessary. Finally, the `socket` is the same as with the server, a channel over which communication takes place.

Listing 6.10: Initiating a Connection

```
ip::tcp::resolver::query query("127.0.0.1", "12345");
ip::tcp::resolver resolver(ioService);
ip::tcp::resolver::iterator iterator = resolver.resolve(query);

ip::tcp::socket socket(ioService);
boost::asio::async_connect(socket, iterator,
  [](const boost::system::error_code& error,
     ip::tcp::resolver::iterator iterator)
  {
    if (!error)
    {
      ip::tcp::endpoint endpoint(*iterator);
      std::cout << "Made a connection to: ";
      std::cout << endpoint;
      std::cout << " on thread id: ";
      std::cout << std::this_thread::get_id() << std::endl;
    }
  });
```

Listing 6.10 shows the code to initiate a connection with a waiting application. The `query` is initialized with the IP address and port of an application known to be waiting for incoming connections; a `query` can be more sophisticated than this, but this simple approach suffices for the purposes of this book. The `resolver` is initialized with the `io_service` and then passed a `query` when the `.resolve` method is invoked. The resolver does provide an asynchronous `.async_resolve` method, but is not necessary in this context. Next, the free `async_connect` function is used to make the connection to the server.

Like the server code, the client uses an asynchronous approach to making the connection. While the client isn't concerned with scalability in this regard, taking this approach allows the application thread to continue execution while waiting on the connection to complete. Then, once the connection is made, the connection lambda is invoked on an `io_service` thread.

6.6 Boost Asio Sending & Receiving Data

Sending and receiving of data is as simple or complex as your application requires. On the simple end of the spectrum, all writes and reads can be done synchronously. On the complex end, all writes and reads need to be done asynchronously. The ability to invoke synchronous or asynchronous writes and reads is simple in itself, the complexity arises in their correct use within an application. This part of the chapter begins with an introduction to Boost buffers in Section 6.6.1, then continues by showing simple write and read operations over sockets in Section 6.6.2, and finishes by showing the use of Boost strands to synchronize multi-threaded writes over a single socket in Section 6.6.3.

6.6.1 Boost Buffers

It is necessary to discuss the various Boost.Asio write and read techniques that utilize `boost::asio::buffer`s, which come in both constant and mutable flavors. Mutable buffers can be used for both sending and receiving of data, whereas constant buffers can only be used for sending. This chapter presents only an introduction to Boost buffers, while Chapter 7 develops a general approach to taking application data structures and efficiently converting them for use in network communicaton via Boost buffers.

Boost buffers provide a consistent representation of data to the various Boost.Asio functions and methods. A `boost::asio::buffer` is a lightweight class that points to existing data, rather than owning it. Stated simply, the buffer contains a void pointer (`void*`) to a sequence of bytes in memory, along with a count of the number of bytes in the sequence. *Boost buffers do not own the underlying* *data!* If the data goes out of scope, the pointer to the data the buffer holds becomes invalid. The original data must stay in scope for the lifetime of the buffer.

Because buffers are the only way to perform network communication, the trick is in how to create a buffer that refers to the application memory. Thankfully, for most Plain Old Data (POD) types it is easy, there are many overloads of the `boost::asio::buffer` function that take these types and return either a mutable or constant buffer.

Let's begin by creating a buffer that refers to the data contained within a `std::array`. A `std::array` contains a contiguous sequence of bytes, interpreted according to however the array has been typed. The `.data` member of the array returns a typed pointer to the underlying raw data, which is useful when creating a Boost buffer that refers to the array. Listing 6.11 shows how to create a Boost buffer for a `std::array`.

Listing 6.11: Boost Buffer on a `std::array`

```
std::array<uint16_t, 10> myArray;
auto myBuffer = boost::asio::buffer(
  myArray.data(),
  myArray.size() * sizeof(uint16_t));
```

Using the `.data` method, along with the `.size` method and the `sizeof` operator provides the parameters necessary to construct a mutable buffer.

The example in Listing 6.11 shows the general case for creating a buffer, however, it is possible to use a more simplified syntax to create a buffer over a `std::array`. Listing 6.12 shows the simplified syntax, the `buffer` function simply accepts the array as the parameter.

Listing 6.12: Simplified Boost Buffer on a `std::array`

```
std::array<uint16_t, 10> myArray;
auto myBuffer = boost::asio::buffer(myArray);
```

It isn't quite true to say that `boost::asio::buffer` creates and returns a single buffer, it actually returns either a `mutable_buffers_1` or `const_buffers_1`. These classes actually represent a sequence of buffers, and in all the examples shown in this section (and throughout the book), it is always a sequence of 1. The Boost.Asio methods also accept buffer sequences, making it valid to use the `boost::asio::buffer` function to return a buffer sequence, even though the sequence contains a single buffer.

The code segment shown in Listing 6.13 shows how to take a primitive type and create a buffer over it. It is necessary to take the address of the `uint16_t` because the buffer needs to point to the bytes of the data, and then again, the `sizeof` operator is used to determine the number of bytes in the raw data sequence.

Listing 6.13: Boost Buffer on an `uint16_t`

```
uint16_t myInt = 4;
auto myIntBuffer = boost::asio::buffer(&myInt, sizeof(myInt));
```

With the background knowledge of creating buffers, it is possible to move on to using them for sending and receiving data via network communication.

6.6.2 Simple Write & Read

Once a connection is established and both the sender and receiver share a socket for communication, the various send and receive techniques can be applied. This section utilizes a relatively simple synchronous send and asynchronous receive between two applications. The sender only sends data, and the receiver only receives data.

Every application must define a protocol for sending and receiving of data; a receiver must know what to expect from the sender. The code samples shown in this section send and receive a variable length string message. The protocol for this requires the sender first send a 16-bit (two bytes) integer value that indicates how many characters to expect in the string message. This is followed by sending that many characters.

Listing 6.14 contains code that sends ten messages to a receiver over a connected socket. The message is generated at the start of the loop, followed by using the free `boost::asio::write` function to send the size of the message, which is then followed by sending of the bytes that compose the message.

Listing 6.14: Simple Synchronous Send

```
void sendMessages(std::shared_ptr<ip::tcp::socket> socket)
{
  for (auto line : IRange<uint8_t>(1, 10))
  {
    std::ostringstream os;
    os << "This is line #" << line << std::endl;

    uint16_t sendSize = os.str().size();
    boost::asio::write(*socket,
      boost::asio::buffer(&sendSize, sizeof(uint16_t)));
    boost::asio::write(*socket, boost::asio::buffer(os.str()));
  }
}
```

The reason for using `boost::asio::write` instead of something like `tcp::socket::send` or `tcp::socket::write_some` is that the former won't return until all the data has been written (or an error occurs), whereas the later methods are not guaranteed to complete the write operations. In all cases, there are overloads that allow a write, or completion handler (a lambda) to be specified which indicates what did or didn't happen during the write. As a simple introduction, `boost::asio::write` suffices.

The reciever code for this example uses an asynchronous approach to waiting for the message to arrive. The reason for doing this is to allow the application to continue doing something on the main thread, and let one of the `io_service` threads handle the incoming data, and also only tie up one of those threads when there actually is something to receive. Listing 6.15 shows the code used to wait for, and then receive a message. At first glance it may not look so *simple*, but after working through it step-by-step it will become clear.

Listing 6.15: Simple Asynchronous Receive

```
void handleNextMessage(
  std::shared_ptr<ip::tcp::socket> socket,
  uint16_t socketID)
{
  using boost::asio::buffer;
  using boost::system::error_code;

  if (!socket->is_open()) return;

  std::shared_ptr<uint16_t> bufferSize = std::make_shared<uint16_t>();
  boost::asio::async_read(
    *socket,
    buffer(bufferSize.get(), sizeof(uint16_t)),
    [socket, bufferSize]
    (const error_code& error, std::size_t bytes)
    {
      if (!error)
      {
        std::string receiveString;
        receiveString.resize(*bufferSize);
        boost::asio::read(
          *socket, buffer(
            const_cast<char*>(receiveString.data()),
            *bufferSize));
        std::cout << "Received from " << socketID  << " : " << receiveString;

        handleNextMessage(socket);
      }
    });
}
```

At the start of the function the socket is tested to see if it is open, if it isn't, no need to try to receive. Because `handleNextMessage` is a non-blocking function itself, it is necessary to dynamically allocate memory for where to place the first two bytes of data expected, a `uint16_t` indicating the size of the message. If this memory were not allocated and a local variable was used and captured in the lambda, the local variable would go out of scope and when the read occurs, it will access an invalid memory location.

The final part of the function is a call to the free `boost::asio::async_read` function. This function expects a socket over which to listen, a buffer to write the result, and a handler to invoke when the read is complete. `boost::asio::async_read` is a non-blocking function, it calls into the `io_service` associated with the socket and leaves the read handler pending receipt of the data. When data is received, the read handler is invoked by a thread associated with the `io_service`.

Because the received message is going to be displayed to the console, a `std::string` is declared and set to the correct size and used as the destination buffer for the read. There is a Boost buffer overload that takes a `std::string`, but it returns a constant buffer, which can't be the destination for a read. Because we know what we are doing, it is necessary to do a little white lie about the type of buffer we are handing off to `boost::asio::buffer`. The `const_cast` is used to remove `const` from the type returned by the `.data` member. Doing this allows a mutable buffer to be created and used as the destination buffer for the read operation. By doing this, the read operation writes directly into the string, eliminating a write into one location in memory and then copy into the string later on.

A note of caution in relying upon `.data()` to return a pointer to the underlying `std::string` representation: Not everyone agrees this is correct, safe, and portable to do. My experience with C++11 on using both Visual Studio 12 and gcc 4.7 (on x86 platforms) have worked in the interoperable way I desire. As with all things, your mileage may vary. The only way to know is to test on your expected platforms. An alternative approach is to encode the characters from the `std::string` into a `std::vector`, transmite/recieve that vector, then recompose those characters into a `std::string`.

Inside the read handler the synchronous `boost::asio::read` function is used to receive the rest of the message. In this case, it is reasonable to use a synchronous read because the code is already running in the context of an `io_service` thread; it isn't going to block the main application. Alternatively, the handler could be written to use another asynchronous read, which may provide for better scalability. However, the initial asynchronous read, followed by a synchronous read is sufficient for a large number of applications.

6.6.3 Multi-Threaded Writes

Using multiple threads, or thread pools, in support of scalable network communications adds an additional layer of complexity. Multiple sockets can be serviced concurrently, and reads and writes on a single socket can be serviced concurrently, but over a single socket neither multiple reads nor multiple writes can be interleaved. Therefore, the key synchronization that must be handled is to ensure that reads and writes over a single socket are correctly ordered, this is where Boost strands become invaluable.

Ensuring a single thread reads from a socket is relatively straightforward by using a single `.async_receive` to wait for incoming data over each socket, then posting an `io_service` request to read the remaining data. However, we want the ability to post writes to a socket from any thread without having to require those threads coordinate among themselves to avoid interleaving their communication. This is where Boost strands come into play. Strands perform the synchronization necessary to ensure that each write request is completed in the order received and that each request is completed before the next one is started. The technique is to associate a strand with each socket, and post all write requests for a socket to its associated strand. The remainder of this section works through a code example that demonstrates this.

The receiver, or server, for this example is exactly the same as described in Section 6.6.2. The implementation for that receiver works correctly for communication with any client, even if the client is multi-threaded. In fact, the receiver works correctly for multiple sender processes, communicating over multiple sockets. The only modification that might be desired is to add a thread pool on the `io_service` to allow multiple sockets to be serviced concurrently.

The first part of the sender, or client, code is a class that represents a *sender*. A sender is an object that runs on its own thread, sending messages over a shared socket, using a strand associated with the socket to post its send requests. Listing 6.16 shows a section of the `Sender` class that contains the public interface, along with the important private attributes.

Listing 6.16: `Sender` Class Overview

```
class Sender
{
public:
  Sender() :
    m_socket(nullptr),
    m_strand(nullptr)
  {
  }

  void startSending(
    std::shared_ptr<ip::tcp::socket> socket,
    std::shared_ptr<boost::asio::strand> strand)
  {
    m_socket = socket;
    m_strand = strand;
    m_thread = std::make_shared<std::thread>(
      [this]()
      {
        sendMessages();
      });
  }

private:
  std::shared_ptr<std::thread> m_thread;
  std::shared_ptr<ip::tcp::socket> m_socket;
  std::shared_ptr<boost::asio::strand> m_strand;
}
```

The private members of the class, `m_thread`, `m_socket`, and `m_strand` are the important data components for each sender. The `m_thread` is unique to each sender, as each sender executes on its own thread. On the other hand, `m_socket` and `m_strand` are shared among all sender objects; this example has only a single socket for communication. The socket is associated with a strand, with the intention the strand is used by all threads to post send requests to the socket. The `startSending` method creates a thread that moves forward with sending messages. Therefore, each sender is sending messages on a different thread.

Listing 6.17 shows the client code that is executed after a connection is made with a receiver. Once a connection is made, it is okay to create a `boost::asio::strand` that is associated with the newly connected socket. Each sender instance is then told to start sending messages, and passed which socket to use, along with the strand assoicated with the socket.

Listing 6.17: Start Sending Messages

```
auto strand = std::make_shared<boost::asio::strand>(ioService);
for (auto sender : senders)
{
  sender->startSending(socket, strand);
}
```

When `startSending` is called, it creates a thread that invokes the `sendMessages` method. This method, shown in Listing 6.18, initiates the message send requests. Rather than directly sending the messages in this method, this method posts 5 requests to the strand associated with the socket. Each of these requests is another lambda function which ends up calling the `sendMessage` method to handle the actual sending of the messages. The reason for using a lambda is to provide the `.post` method a function with the correct signature.

Listing 6.18: Sending Messages

```
void sendMessages()
{
  for (auto line : IRange<uint16_t>(1, 5))
  {
    m_strand->post([this, line]() { sendMessage(line); });
  }
}

void sendMessage(uint16_t line)
{
  std::ostringstream os;
  os << "This is line #" << line << std::endl;

  uint16_t sendSize = os.str().size();
  boost::asio::write(*m_socket,
    boost::asio::buffer(&sendSize, sizeof(uint16_t)));
  std::this_thread::sleep_for(std::chrono::milliseconds(100));
  boost::asio::write(*m_socket,boost::asio::buffer(os.str()));

  std::cout << "Sent: " << os.str();
}
```

Because `sendMessages` makes posts to the strand associated with the socket, and that the code in `sendMessage` uses synchronous calls, we are guaranteed that all socket communication across all threads happens in the order they are posted and without any concern for interleaving.

Take note of the `sleep_for` call in the `sendMessage` method, why is it there? The reason I have placed it there is to help emphasize that no matter how long the blocking `sendMessage` takes, including how many (synchronous) socket calls it makes, that all calls work correctly. It is also important to know that the client code is written with a thread pool on the `io_service`. Meaning that the client code is not only multi-threaded on the `io_service`, but also multi-threaded on the senders, while still correctly sending messages over a single socket.

Sample output from the receiver in a run of this application using a single reciever and two sender processes is shown in Figure 6.19. The first line is a report of the thread id of the `io_service`, following by the connection report of the first sender process. The first sender process begins sending messages, and then the second sender process makes a connection. The remainder of the output is from both sender processes sending their messages. Remember that each sender is multi-threaded, with two threads, that is the reason there are four of each message in the output, two from each multi-threaded sender process.

Figure 6.19: Multi-Threaded Send/Receive Output

```
io_service thread id: 8836
Received connection from: 127.0.0.1:61874 on thread id: 8836
Received from 1 : This is line #1
Received from 1 : This is line #2
Received from 1 : This is line #3
Received from 1 : This is line #4
Received from 1 : This is line #5
Received connection from: 127.0.0.1:61877 on thread id: 8836
Received from 1 : This is line #1
Received from 2 : This is line #1
Received from 1 : This is line #2
Received from 2 : This is line #2
Received from 1 : This is line #3
Received from 2 : This is line #3
Received from 1 : This is line #4
Received from 2 : This is line #4
Received from 1 : This is line #5
Received from 2 : This is line #5
Received from 2 : This is line #1
Received from 2 : This is line #2
Received from 2 : This is line #3
Received from 2 : This is line #4
Received from 2 : This is line #5
```

For proof that posting to a `strand` versus the common `io_service` make any difference, replace the strand post code from Listing 6.18 with the code shown in Listing 6.20. This changes the posts to be done on the general `io_service` instead of the `strand` associated with the socket. The `io_service` does not offer the same ordering and completion guarantee provided by a `strand`. Therefore, when the client is run, the writes to the socket get interleaved and the receiver gets confused and the protocol breaks down and fails to complete.

Listing 6.20: Incorrect Service Request

```
m_socket->get_io_service().post([this, line]() { sendMessage(line); });
```

6.7 Summary

Even though this chapter is considered an introduction to network programming with Boost, in reality, a high level of sophistication is presented. This chapter has introduced Boost.Asio and its model, the use of thread pools on an `io_service`, the concept of strands, making connections, and communicating over sockets, including the use of multiple senders, each on their own thread, and over the same socket. These techniques cover almost everything that is used to build the programs presented in the remainder of the book

One outcome I hope that comes through in the code presented in this chapter is how concise and readable the programs are, in addition to being cross-platform over a wide number of operating systems and compilers. This is due to the well designed Boost.Asio library, but also made possible through the C++11 language and standard library improvements.

7 | Message Coding

All distributed systems utilize an encoding or serialization scheme as part of passing messages between different processes throughout the system. The encoding scheme used in the remainder of this book is presented in this chapter. This topic is presented as a separate chapter in order to keep the focus on the distributed framework in Chapter 8.

Several considerations are important when deciding how to perform serialization. The first is performance. By performance we mean both the computational time to serialize/deserialize and the resultant size of the serialized object. A second is how difficult the scheme is to use, incorporate, and maintain. This book is focused on computational performance, therefore, that dominates the decision making. The distributed and fault-tolerant applications make use of the Boost libraries for networking, therefore, it is natural to look to the Boost serialization library. The Boost scheme is designed for general serialization, whether to a file or over a network stream. Because of this, the computational performance and size of the resultant serialization object is larger than is possible with an approach specifically focused on network transmission, not to mention the complexity involved to use. The final decision for the code presented in this book is to create a fairly simple, and highly performant solution for the distributed and fault-tolerant frameworks.

7.1 Message Encoding/Decoding

The approach used in this book is a basic byte-by-byte encoding/decoding of data types. Values are decomposed into their individual bytes and added to a `std::vector`. When sending and receiving of data, the vector is easily wrapped by a Boost.Asio buffer. To facilitate the message coding, a base `Message` class from which all system messages are derived, provides the core message encoding and decoding support. This class provides a set of overloaded encode and decode methods which handle the primitive types used by the applications presented in this book. The section of the `Message` class declaration relevant to the encoding/decoding methods is shown in Listing 7.1.

Listing 7.1: Partial `Message` Declaration

```
class Message : public std::enable_shared_from_this<Message>
{
protected:
  virtual void encodeMessage() = 0;
  virtual void decodeMessage() = 0;

  void encode(uint8_t value);
  void encode(uint16_t value);
  void encode(uint32_t value);
  void encode(double value);

  void decode(uint8_t& value);
  void decode(uint16_t& value);
  void decode(uint32_t& value);
  void decode(double& value);

private:
  std::vector<uint8_t> m_data;
  uint16_t m_decodePosition;
};
```

The various `encode` and `decode` methods provide the byte encoding and decoding capabilities used by derived classes. The `m_data` member is used to store the encoded bytes, both for sending and receiving. An `std::vector<uint8_t>` is directly supported by Boost.Asio as a buffer from which it can send and into which it can receive. The `m_decodePosition` is used in support of message decoding, to remember the next byte to decode. Finally, the pure virtual `encodeMessage` and `decodeMessage` methods are defined for derived classes, with the expectation that their implementations perform the data encoding and decoding. A demonstration of these methods is shown in Section 7.2.

The reason for deriving from `public std::enable_shared_from_this<Message>` is described in Section 7.3.

7.1.1 Encoding Values

Listing 7.2 shows the implementations of the `encode` methods for `uint8_t` and `uint16_t` types. For the `uint8_t` type, the value is simply added to the `m_data` member. However, for `uint16_t` type a little more work in involved. The first step is to convert the value from the host platform representation to the network representation for network transmission. This is performed through the `htons` function call. With that done, a byte-by-byte encoding of the value is done, adding each of the two bytes to the `m_data` member. While not shown in the example code, this same thing is done for the four bytes of the `uint32_t` type.

Listing 7.2: Encoding Integers

```
void Message::encode(uint8_t value)
{
  m_data.push_back(value);
}

void Message::encode(uint16_t value)
{
  uint16_t network = htons(value);
  uint8_t* bytes = reinterpret_cast<uint8_t*>(&network);
  m_data.push_back(bytes[0]);
  m_data.push_back(bytes[1]);
}
```

Floating point data types can suffer from endian issues [1], although somewhat rare. In spite of this there is no equivalent host to netowrk float function or network to host float. It is the responsibility of the developer to know if floating point types are going to be transmitted between platforms that represent them differently. For the purposes of this book I don't take any special considerations for different floating point representation. The code shown in Listing 7.3 shows the `encode` method for the `double` data type. A `double` is eight bytes in size, therefore, all eight bytes are added to the `m_data` member.

Listing 7.3: Encoding Doubles

```
void Message::encode(double value)
```

[1] http://en.wikipedia.org/wiki/Endianness#Floating-point_and_endianness

```
{
  uint8_t* bytes = reinterpret_cast<uint8_t*>(&value);
  m_data.push_back(bytes[0]);
  m_data.push_back(bytes[1]);
  m_data.push_back(bytes[2]);
  m_data.push_back(bytes[3]);
  m_data.push_back(bytes[4]);
  m_data.push_back(bytes[5]);
  m_data.push_back(bytes[6]);
  m_data.push_back(bytes[7]);
}
```

7.1.2 Decoding Values

The data for a message is received directly into the `m_data` vector. After receipt of the data, the bytes in this vector need to be recomposed (decoded) into the values of the receiving class. Listing 7.4 shows the implementations of the `decode` methods for `uint8_t` and `uint16_t` types. These methods perform the reverse operation of the encode methods. They take the bytes stored in the `m_data` vector and assign their values to the `value` parameters.

Listing 7.4: Decoding Integers

```
void Message::decode(uint8_t& value)
{
  value = m_data[m_decodePosition++];
}

void Message::decode(uint16_t& value)
{
  uint8_t* bytes = reinterpret_cast<uint8_t*>(&value);
  bytes[0] = m_data[m_decodePosition++];
  bytes[1] = m_data[m_decodePosition++];

  value = ntohs(value);
}
```

7.2 Example Message

This section walks through a simplified version of the `MandelMessage` class from the distributed demonstration application detailed in Chapter 8. This class shows how the base methods provided by the `Message` class are used to encode members before sending and to decode data into members following receipt of the data.

Listing 7.5 shows the declaration of the `MandelMessage` class. The purpose of this class is to send the information regarding a section of the Mandelbrot image to compute to a distributed worker thread. This information contains which rows to compute, the range of the complex numerical plane those rows represent and how many iterations to attempt.

Listing 7.5: Simplified `MandelMessage` Declaration

```
class MandelMessage : public Message
{
public:
  MandelMessage(/* -- parameters go here -- */) :
    Message(Messages::Type::MandelMessage),
    m_startRow(startRow),
    m_endRow(endRow),
    m_startX(startX),
    m_startY(startY),
    m_sizeX(sizeX),
    m_deltaX(deltaX),
    m_deltaY(deltaY),
    m_maxIterations(maxIterations)
  {
  }

  MandelMessage() {}

protected:
  virtual void encodeMessage();
  virtual void decodeMessage();

private:
  uint16_t m_startRow;
  uint16_t m_endRow;
  uint16_t m_sizeX;
  double m_startX;
  double m_startY;
  double m_deltaX;
  double m_deltaY;
  uint16_t m_maxIterations;
};
```

The `MandelMessage` class has two constructors. The first is used when the class is being created for sending the message. The main thing done by this constructor is to assign the constructor parameters to the class members. Notice this constructor passes the type (`Messages::Type::MandelMessage`) of the class into the base `Message` constructor. Where this value comes from is described in the next chapter with the revised Mandelbrot application code. This type gets sent as part of the message to allow the receiver to identify which message was received and therefore, which class to create to complete the decoding. The second constructor's purpose is to allow the class to be created for when the message is being received. Because nothing is known about its members in advance of receipt, a default constructor suffices.

Before a message is sent, the base `Message` class makes a call to the virtual `encodeMessage` method. The purpose of this method is to take the derived class members and encode them in advance of sending. Listing 7.6 shows the implementation for the simplified `MandelMessage` class. This method simply goes through each of the class members and calls one of the overloaded `encode` methods.

Listing 7.6: `MandelMessage` Encoding

```
void MandelMessage::encodeMessage()
{
  this->encode(m_startRow);
  this->encode(m_endRow);
  this->encode(m_sizeX);
  this->encode(m_startX);
  this->encode(m_startY);
  this->encode(m_deltaX);
  this->encode(m_deltaY);
  this->encode(m_maxIterations);
}
```

Once a message has been received, the bytes received over the network must be recomposed into the members of the derived class. The bytes need to be decoded in the exact order in which they were encoded. This is done by simply calling the overloaded `decode` methods in the same order as they were encoded. Listing 7.7 shows the `decodeMessage` for the simplified `MandelMessage` class.

Listing 7.7: `MandelMessage` Decoding

```
void MandelMessage::decodeMessage()
{
  this->decode(m_startRow);
  this->decode(m_endRow);
  this->decode(m_sizeX);
  this->decode(m_startX);
  this->decode(m_startY);
  this->decode(m_deltaX);
  this->decode(m_deltaY);
  this->decode(m_maxIterations);
}
```

This is an example of a fairly simple message with only primitive data types. A message may contain any data, it only needs to be decoded into bytes for sending and receiving. The base `Message` class can be extended to support other standard types, such as `std::string`, which then become available for use by other message classes.

7.3 Network Communication

In addition to providing core message data encoding and decoding capabilities, it also provides the methods used to send and receive the messages. Furthermore, as will be seen shortly, it is in the send and receive methods where the message encoding and decoding is requested. Apart from a single byte recevied in each of the client and server applications, all network communication takes place by the `Message` class. The class provides two different send and one read method. The difference between the two send methods is that one is blocking, while the other is non-blocking. The read method is blocking. The code for these methods is detailed in the remainder of this section.

7.3.1 Sending Messages

In order to develop a fully scalable application, pending I/O must not prevent the CPU from performing useful work. At the same time, a scalable and distributed application will be computing tasks on many CPU cores spread over many different computers. This work requires messages being passed back and forth, some sending work, some sending results, all being generated at undetermined times by many different CPU cores and systems. However, there is only (usually) a single network card on each computer. This piece of hardware can only send data over one stream at a time. The OS `socket` library and the Boost.Asio framework hides some of this complexity by allowing multiple sockets to be open and written to by a multi-threaded application. However, remember from the networking discussion in Chapter 6, it isn't valid to interleave writes to a single socket. The way to synchronize multiple threads writing to the same socket is to use Boost Strands. That is the purpose of the send method shown in Listing 7.8.

Listing 7.8: Non-Blocking Send

```
void Message::send(
  std::shared_ptr<ip::tcp::socket> socket,
  boost::asio::strand& strand)
{
  auto captureThis = shared_from_this();
  strand.post(
    [captureThis, socket]()
    {
      captureThis->send(socket);
    });
}
```

The code calling this method provides both the socket over which the message is to be sent, along with the strand on which to schedule the execution. The core of this function is simply to post a lambda to the supplied strand. Immediately upon posting the lambda, the method returns, before the send has taken place; this is why it is considered non-blocking. When the lambda is eventually executed, it makes a call to the blocking send method. It is valid for the lambda to make a blocking call because it is executing in the context of a thread running on the application's `io_service`.

The first statement of this method deserves a little explanation. Referring back to Listing 7.1 we see the class is derived from `public std::enable_shared_from_this<Message>`. Deriving from this template class allows the use of the `shared_from_this()` function call, which returns a `std::shared_ptr` to the current `this` class instance.

The reason the shared pointer to the (derived) `Message` instance is necessary is to ensure the lifetime of the `Message` matches that of the lambda posted to the strand. If the lamba had been written with `this->send(socket);`, when the lambda is finally executed, the `this` pointer is no longer valid, resulting in an application failure. Instead, a shared pointer is created and captured by value in the lambda. When the lambda is executed, we are guaranteed to have a valid pointer to the original `this`, in addition to ensuring the lifetime of the `Message` instance lives for at least as long as the lambda. Note: In order for the pointer returned from the call to `shared_from_this()` to keep the class instance alive by passing the `shared_ptr` by copy into the lambda, the instance must have been dynamically allocated from the heap, and not created on the stack. Yes, this **is** tricky stuff!

The method that does the real work of encoding and sending the message data is shown in Listing 7.9. As a quick bit of error handling, the first thing the method does is to check if the socket is even open. Following this, the data for the message is encoded, using all of the technqiues discussed previously. With the message encoded, the free `boost::asio::write` function is used to send the data. The reason for choosing this function is that it blocks until all data is sent, whereas `socket::send` may return without having sent all data.

Listing 7.9: Non-Blocking Send

```
void Message::send(std::shared_ptr<ip::tcp::socket> socket)
{
  if (socket->is_open())
  {
    this->encodeMessage();
    boost::asio::write(*socket, boost::asio::buffer(m_type));

    std::array<uint16_t, 1> size;
    size[0] = htons(static_cast<uint16_t>(m_data.size()));
    boost::asio::write(*socket, boost::asio::buffer(size));

    if (m_data.size() > 0)
    {
      boost::asio::write(*socket, boost::asio::buffer(m_data));
    }
  }
}
```

The first write call sends a single byte that indicates which type of message is being sent. For sending messages, this type is specified by the derived message class, being passed as a parameter into the `Message` class constructor. The types are defined by an enumeration class found in the file `MessageTypes.hpp`; this class is shown in Listing 7.10. The next two bytes sent are the size of the message. Finally, the encoded data is sent in the third write call.

Notice the `htons` call is only made on the size of the data for the message. This is not needed for the message type, beause it is only a single byte, there is no difference in representation on different CPUs for single bytes. Because the size of the message is two bytes, it is necessary to convert to the network representation before sending. Finally, the data stored in `m_data` has already gone through any necessary `hton` calls.

Listing 7.10: Message Type Enumeration

```
namespace Messages
{
  enum class Type : uint8_t
  {
    Undefined,
    TaskRequest,
    MandelMessage,
    MandelResult,
    NextPrimeMessage,
    NextPrimeResult,
    TerminateCommand
  };
}
```

7.3.2 Receiving Messages

Receiving a message is a little different from sending messages. When sending a message, the code starts with a specific derived `Message` class instance. When receiving data, it isn't possible to know which class instance to make in advance, because any message may arrive at any particular time. Therefore, a two-step process is necessary. The first step is to receive a single byte that identifies the message type (`Messages::Type` from Listing 7.10), then based upon this value, the correct derived `Message` class is instantiated and used to read the remainder of the bytes and decode it into the instance members.

Reading of the first byte and creation of the correct derived `Message` class instance is discussed in detail in the Chapter 8, where the first distributed application is detailed. For this chapter it is enough to know that the first byte is read, examined, and the correct class instance is created.

Once the correct derived `Message` class is instantiated, the `Message::read` method is called to read the remainder of the data and decode it into the instance members. The code for this method is found in Listing 7.11.

Listing 7.11: Message Read & Decode

```
void Message::read(std::shared_ptr<ip::tcp::socket> socket)
{
  if (socket->is_open())
  {
    std::array<uint16_t, 1> size;
    boost::asio::read(*socket, boost::asio::buffer(size));
    size[0] = ntohs(size[0]);
    m_data.resize(size[0]);
    if (size[0] > 0)
    {
      boost::asio::read(*socket, boost::asio::buffer(m_data));
      this->decodeMessage();
    }
  }
}
```

The same as the **send** method, the first part of the function validates the socket is open, only if the socket is open does the code continue. Next, the size of the message is read, then based upon the size of the rest of the data, a buffer is resized and the remainder of the data is read from the socket. With all of the data read, the `decodeMessage` method is called, where the custom code for the class instance is called, which results in the data being decoded into the instance members. Notice also, that `ntohs` is called after receiving the size and before it is used. This matches the call to `htons` when sending the message.

7.4 Summary

This chapter has shown the approach through which messages are represented, coded, sent, received, and decoded. The code is fairly straightforward and easy to understand. However, there is some hidden complexity because this code runs in the context of the application defined task `ThreadPool` and the pool of threads running on the application `io_service` instance. Chapter 8 provides the full context in which this messaging scheme is utilized, along with the different messages used to send work and results and how the client and server applications are setup to send and receive the messages.

With the background details out of the way, it is now time to turn attention to the core subjects of this book!

8 | Scalability - Distributed

Some problems require more computing resources than are reasonably available on a single system. The scale and/or timing constraints of these problems force the need to bring together a number of networked systems and distribute the computing tasks among them; hence the term *distributed computing*. This chapter tackles the biggest change to the scalability framework, distributing tasks to other computers and returning the results.

In order to distribute tasks over multiple systems, two fundamental changes are required to the framework. The first and most obvious is that two different kinds processes are necessary, separate client and server processes. The second is that of a networking framework through which messages are sent and received. While the framework continues to be familiar, these are big changes and require the addition a lot of new code.

A secondary change is with respect to building the system. These changes result in three separate projects, shared code, client code, and server code. The client and server share a large amount of code in commmon, therefore most of the system code is in a shared project that is compiled to a static library and linked into both the client and server projects. The client and server each have a smaller set of code that make them unique, two projects are created that buidl the separate processes.

8.1 System Design

The system design approach presented in this chapter, and the next two, is that of a single interactive client and many connected compute servers; Figure 8.1 illustrates this design. In our case, the client is the interactive Mandelbrot application. The client generates computational tasks in response to the user changing the view. These tasks are distributed among the servers. The servers receive tasks, execute them, and return the results back to the client.

The use of the terms *client* and *server* may sound a little different than what you may expect, but they aren't. A typical description of a system is that of a server with many connected clients, for example many web browsers and a single web server. In the model presented in this chapter, there is a single client, but many connected servers. If you think about the clients and servers, regardless of their number, they perform the same roles in both models. In the case of the clients, a web server or an interactive Mandelbrot application, they both respond to user inputs and make requests to a

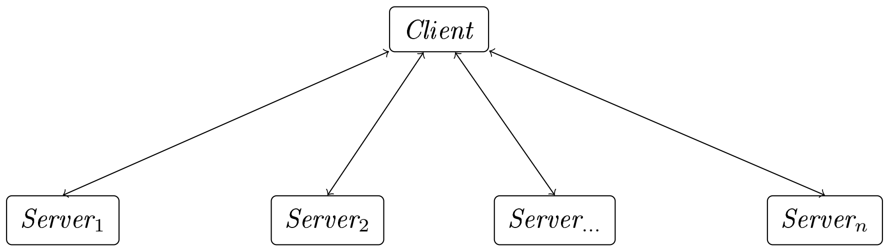

Figure 8.1: System Design

server, or servers. With respect to a server in a web system, the web server receives requests, performs computational tasks and returns those results to the client. The server in our system receives tasks, executes those tasks, and returns the results to the client. In other words, the use of the terms is the same, it is only the system topology that is different.

Figure 8.2 provides an overview of the system model. The client process accepts input from the user and generates compute tasks, it is the server(s) that perform the execution of the tasks. Rather than tasks being placed into a client thread pool as is done in the previous chapters, tasks are placed into a *Global Task Queue* from which they are distributed throughout the connected compute servers. Immediately after making a successful connection, a compute server sends a set of task requests to the client (represented by the arrow from the compute server to the *Task Request Queue*), indicating it is available to receive work. The *Task Request Queue* holds the requests from the servers which get matches with waiting tasks. When a match is made, the task is transmitted to the compute server and placed in its *Local Task Queue* (represented by the arrow from the client *Global Task Queue* to the compute server *Local Task Queue*) from which the server selects, executes the tasks, compiles the results, and returns the results back to the client.

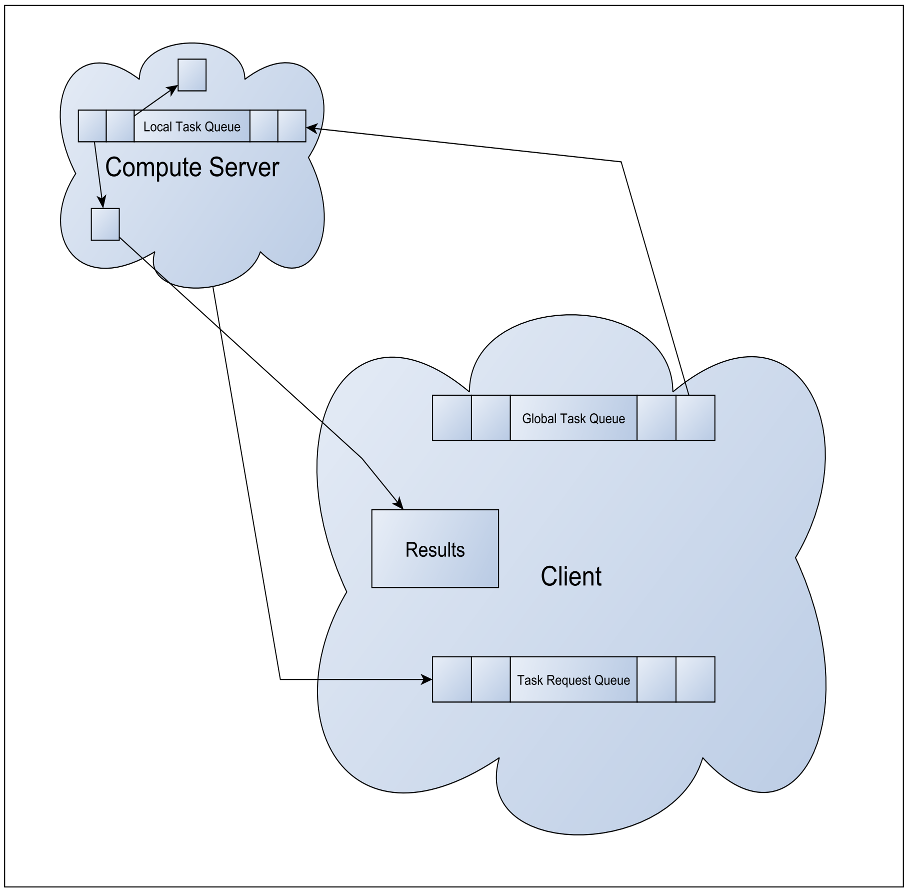

Figure 8.2: Distributed Model

The small squares with arrows pointing to them on the compute server represent tasks that have been removed from the *Local Task Queue* and are being executed. The arrow going from the small square on the compute server to the *Results* box represents a task that has finished execution and its results being sent back to the client.

The model in this figure is only that, a model, the implementation details have additional complexity. The specifics of how this model is implemented are presented in the remainder of the chapter.

Figure 8.3 shows the sequence of communication that takes place between the client and a server. The first step is for a server to initiate and establish a connection with the client. Upon successful connection, the server then sends a task request to the client, indicating it is available to perform work. When the client is ready to have a task computed, it sends a task to the server, the server computes the result, sends it back to the client, and immediately sends another task request indicating it can take on more work with the completion of the previous task. In the case the client is shutting down, it sends a termination message to all connected servers instructing them to gracefully shutdown.

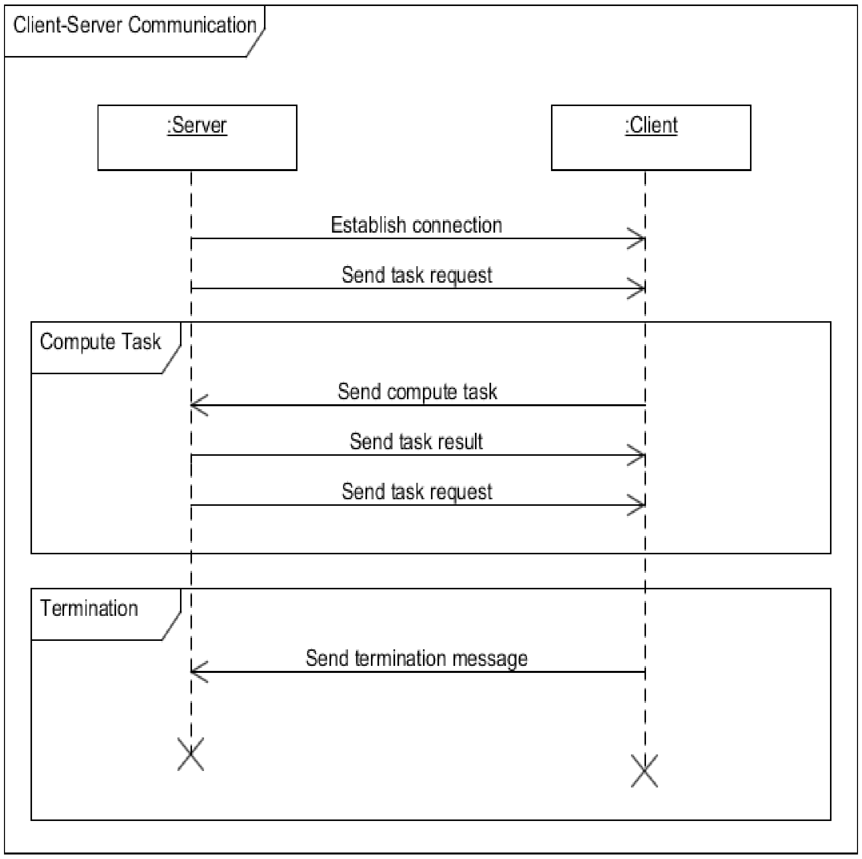

Figure 8.3: Distributed Communication

Not shown in this diagram, but implied, is once the client receives the results, it places them into the display buffer for immediate viewing; or in the case of a prime number, reports the result to the console upon receipt.

The core fundamental continues to hold true, decomposing a problem into computable tasks. The compute servers indicate their availability for work, ensuring the client only sends a task when it has available resources. Because of this, the system as a whole continues to exhibit a scalable nature by automatically load balancing the tasks over the connected compute servers. Because of the design and implementation groundwork laid in the first version of the application, all of the major components that ensure scalability are already in place, significantly easing the transition to a distributed framework, while maintaining the scalable nature.

8.2 Networking Infrastructure

Chapter 6 introduced the networking techniques used as the basic building blocks for distributed communication. The discussion was furthered in Chapter 7 with a description of how messages are coded and encoded for transmission between processes. This section continues the discussion by futher detailing the networking framework.

8.2.1 Client & Server Initialization

The code presented in the previous chapters placed the basic application framework in a class named `ScalabilityApp`, the code associated with this chapter renames the class to `DistributedApp` to better reflect its nature. In addition to a new name, quite a bit of new code has been added. One of these new additions is the initialization of the Boost.Asio library's `io_service`. This code is contained in the `DistributedApp::initialize` method, shown in Listing 8.1. Remember that the system is now composed of two processes, a client and server, this code is only part of the client process.

Listing 8.1: Client Initialization

```
bool DistributedApp::initialize()
{
  using namespace boost::asio;
  m_threadWork = std::unique_ptr<io_service::work>
    (new io_service::work(m_ioService));
  for (auto thread : IRange<uint8_t>(1, 10))
  {
    m_threadsIO.push_back(std::unique_ptr<std::thread>(
      new std::thread(
        [this]()
        {
          m_ioService.run();
        })));
  }

  TaskRequestQueue::instance()->initialize(&m_ioService, &m_servers);

  waitOnConnection();

  return true;
}
```

The first part of this function creates the `io_service`, along with assigning a pool of threads. The client will have potentially many connected compute servers, this allows many different connections to be serviced concurrently. An `io_service::work` object is also created and associated with the `io_service` to prevent it from terminating when there are no events on its request queue. The next step is to initialize the `TaskRequestQueue`, which needs to have access to the application `io_service`; this singleton is fully discussed in Section 8.3. The final step in the initialization is to begin waiting for connections from servers, which is described in the next section.

The server has an analogous initialization to the client, shown by the two code sections in Listing 8.2. It creates an `io_service` queue, `io_service::work` object, but only places a single thread on the `io_service`. The server only ever communicates with the client, therefore the need to have more than one thread to handle network communication is not necessary. Following this, an instance of a class named `ComputeServer` is created and initialized. During its initialization a familiar `ThreadPool` is initialized, from which assigned tasks are associated and executed. Finally, the server enters a state where it initiates a connection to the client, this procedure is discussed in the next section.

Listing 8.2: Server Initialization

```
int main(int argc, char* argv[])
{
  ...
  boost::asio::io_service ioService;
  boost::asio::io_service::work work(ioService);

  std::thread thread = std::thread(
    [&ioService]()
    {
        ioService.run();
    });

  ComputeServer server;
  server.initialize(ioService, ipClient, portClient);
  ...
}

void ComputeServer::initialize(
  boost::asio::io_service& ioService,
  const std::string& ipClient,
  const std::string& portClient)
{
  ThreadPool::instance()->initialize(&ioService);
  connectToClient(ioService, ipClient, portClient);
}
```

8.2.2 Making a Connection

Section 6.5.2 of Chapter 6 provides the background on how to use Boost.Asio to initiate and accept networking connections. The code presented in this chapter follows those techniques, with a few minor differences owing to the context of the Mandelbrot viewing application.

The connection model used by the Mandelbrot application is to have the client wait for an incoming connection from a compute server. For the approach used in the Mandelbrot application, the servers must know the location of the client, but the client does not need advance knowledge of the servers. The start order of the processes doesn't matter. Upon start, a compute server continues to attempt making a connection to the client, retrying until either the connection is established or the process is manually terminated. Similarly, upon start, the client begins waiting for incoming connections, and allowing them to be made at any time during the application lifetime, not only during an initialization phase. This allows the client to take advantage of ever more computing resources over time. The code presented in this chapter does not discuss how to handle servers that disconnect or fail while the client is alive; a topic complex enough to require its own discussion, and tackled in Chapter 9.

The server code for initiating a connection with the client is found in Listing 8.3. The first part of the method is almost exactly as found in Section 6.5.2 of Chapter 6. The one difference is the use of the blocking `connect` instead of the non-blocking `async_connect`. The reason for this is that the server doesn't need to do anything else while waiting for the connection to complete, therefore, the code is simplified by using a blocking call.

Listing 8.3: Initiating A Connection

```
void ComputeServer::connectToClient(
  boost::asio::io_service& ioService,
  const std::string& ipClient,
  const std::string& portClient)
```

```
{
  ip::tcp::resolver resolver(ioService);
  ip::tcp::resolver::query query(ipClient, portClient);
  ip::tcp::resolver::iterator iterator = resolver.resolve(query);

  bool done = false;
  while (!done)
  {
    auto socket = std::make_shared<tcp::socket>(ioService);
    boost::system::error_code error;
    boost::asio::connect(*socket, iterator, error);
    if (!error)
    {
      done = true;
      waitOnTask(socket);
      unsigned int count = std::max(
        1u,
        std::thread::hardware_concurrency()) * 2;
      for (auto core : IRange<unsigned int>(1, count))
      {
        ioService.post(
          [socket]()
          {
            Messages::TaskRequest command;
            command.send(socket);
          });
      }
    }
  }
}
```

Once the `connect` method returns without error, `done` is set to `true` to indicate a successful connection has been made and the connection loop can end. Next, a call to the non-blocking `waitOnTask` method is made. This method, detailed in Section 8.2.4, waits for task messages from the client. The final step is to send a number of task requests to the client. Each request indicates the availability of enough resources to compute one task. This is done by creating a bunch of `TaskRequest` messages and sending them over the connected socket.

The reason for sending more task requests than the number of available cores is to allow for I/O operations to be taking place while other tasks are executing. Consider that a system can be sending a task request, receiving a task, sending the results from executing a task, and executing several other tasks, all at the same time. On the other hand, only a single thread is assigned to the `io_service`, for a number of reasons. Foremost is that there is only one network card, and only one connection to a single client. Secondly, the expectation with the server is for it to be compute bound, working on tasks, instead of being I/O bound, sending and receving messages over the network. Time spent computing should dominate the time spent performing I/O, if at all possible. In order to more effectively keep the CPU cores fully utilized, it is necessary to have at least one task waiting for execution as soon as one completes. This means the number of task requests must exceed the number of available CPU cores by enough to prevent the system from waiting for I/O to complete to begin execution of the next task.

The other side of the connection is the client, which waits for an incoming connection request; the code shown in Listing 8.4 demonstrates this. Again, the code for the client follows the pattern described in Section 6.5.2 of Chapter 6. When a successful a connection is made, a `Server` instance is created and added to the list of known servers. This is followed by a call to the non-blocking method `waitOnMessage`, described in Section 8.2.4.

Listing 8.4: Waiting For Connection

```
void DistributedApp::waitOnConnection()
{
  auto socket = std::make_shared<ip::tcp::socket>(m_ioService);
  m_acceptor.async_accept(*socket,
    [this, socket](const boost::system::error_code& error)
    {
      if (!error)
      {
        Server server(socket);
        m_servers.add(server);
        waitOnMessage(server.id);
      }
      waitOnConnection();
  });
}
```

The `Server` struct is used by the client to maintain connection information about a connected server. Each instance is also assigned a unique identifier that is used as the key for reference in a hash table in the `ServerSet` class. The struct also holds a pointer to the socket over which communication takes place. In order to ensure valid synchronization of threads communicating with each server, as described in Chapter 6, each server is given its own `strand`; all communication requests with the server are posted to this `strand`. The code for this struct is found in Listing 8.5.

Listing 8.5: `Server` Struct

```
struct Server
{
  Server() {}
  Server(boost::asio::io_service& ioService)
  {
    static ServerID_t newId = 0;
    this->id = newId++;
    this->strand = std::make_shared<boost::asio::strand>(
        socket->get_io_service());
  }

  ServerID_t id;
  std::shared_ptr<ip::tcp::socket> socket;
  std::shared_ptr<boost::asio::strand> strand;
  std::array<uint8_t, 1> messageType;
};
```

Associated with the `Server` struct is a class named `ServerSet` which is used as a container to maintain details of all connected servers. This class primarily provides synchronized access to the list of available servers. The `add`, `get`, and `exists` are simple helper methods that control, and synchronize, access to the `std::unordered_map` of `Server` structs. The declaration for this class is shown in Listing 8.6.

Listing 8.6: `ServerSet` Class

```
class ServerSet
{
public:
  void add(Server server);
  boost::optional<Server&> get(ServerID_t id);
  bool exists(ServerID_t id);
  std::unordered_map<ServerID_t, Server> getServers();
private:
  std::unordered_map<ServerID_t, Server> m_servers;
  std::mutex m_mutex;
};
```

8.2.3 Command Pattern

This distributed application takes advantage of a design pattern known as the *Command Pattern*[1] to reduce code complexity and improve it for future maintenance, as additional messages are added to the system. The design pattern is used to associate a message type with a handler function or method. The implementation uses a hash table (`std::unordered_map`), where a message type is used as the key and the value is a `std::function`; which is invoked when the message type is received. The hash table type for the client is shown in Listing 8.7, and the server type is shown in Listing 8.8.

Listing 8.7: Client Command Pattern Type

```
std::unordered_map<
  Messages::Type,
  std::function<void (ServerID_t)>>
```

Listing 8.8: Server Command Pattern Type

```
std::unordered_map<
  Messages::Type,
  std::function<void (std::shared_ptr<ip::tcp::socket>)>>
```

For the client, the function handler accecpts a `ServerID_t` as a parameter, which is the unique identifier for the server from which the message originated. This parameter is then used to lookup the server from the `ServerSet` and retrieve the socket connection for communication. For the server, the function handler accepts the socket over which communication to the client takes place. There is only one socket between the server and client, therefore no need to manage and lookup different sockets for different routes of communication at the server.

The client and server both declare a member variable named `m_messageCommand` of their command pattern types in the `DistributedApp` and `ComputeServer` classes, respectively; I refer to these as *command maps*. During application initialization, the command maps are initialized with the different types of messages that can be received and their associated handlers. The initialization code for the client is show in Listing 8.9.

Listing 8.9: Client Command Map Initialization

```
void DistributedApp::prepareCommandMap()
{
  m_messageCommand[Messages::Type::TaskRequest] =
    [this](ServerID_t serverId)
    {
      processTaskRequest(serverId);
    };
  m_messageCommand[Messages::Type::MandelResult] =
    [this](ServerID_t serverId)
    {
      processMandelResult(serverId);
    };
```

[1] http://en.wikipedia.org/wiki/Command_pattern

```
  m_messageCommand[Messages::Type::NextPrimeResult] =
    [this](ServerID_t serverId)
    {
      processNextPrimeResult(serverId);
    };
}
```

The `prepareCommandMap` method associates a lambda function with each of the types of messages it may receive from a compute server. The lambda has the same signature as the `std::function` defined as part of the `m_messageCommand` type. Upon invocation, the lambda simply makes a call to an instance method that performs the actual work of handling the message type. The initialization of the command map for the server looks the same as the client, except that it responds to different message types and invokes different handlers.

The code in Listing 8.10 demonstrates how to use the command map to invoke the handler for a specific message. The `[Messages::Type::TaskRequest]` part of the statement searches the hash table for the entry corresponding to that key, which is a function. Then, the `(serverId)` part of the statement invokes the returned function.

Listing 8.10: Command Map Invocation

```
m_messageCommand[Messages::Type::TaskRequest](serverId);
```

There command map code for the server is able to be somewhat simplified over the client code because the same thing is done for all messages except for one. Because of this, the code for the server can take advantage of generic programming by having a templated `processTask` function that does the same thing for every message and task type combination. The code for this function is shown in Listing 8.11. This function simply reads the remainder of the message from the socket, then places the task onto the local `ThreadPool` for execution.

Listing 8.11: Server Templated `processTask`

```
template <typename Message, typename Task>
void processTask(std::shared_ptr<ip::tcp::socket> socket)
{
  Message message;
  message.read(socket);
  std::shared_ptr<Tasks::Task> task = std::make_shared<Task>(socket, message);

  ThreadPool::instance()->enqueueTask(task);
}
```

The initialization of the server command map makes use of the templated `processTask` function. As this method is used, the message and task types are specified as template parameters for the `processTask` function, as demonstrated in Listing 8.12. There is one exception to the messages received at the server, and that is the terminate command. A `TerminateCommand` message is not a computational task, therefore it doesn't belong on the `ThreadPool`, it is a message instructing the server process to perform a graceful shutdown. Because of this, a specific function is used to handle that message, `processTerminateCommand`, which is reflected in the initialization of the command map.

Listing 8.12: Server Command Map Initialization

```
void ComputeServer::prepareCommandMap()
{
  m_messageCommand[Messages::Type::MandelMessage] =
    [this](std::shared_ptr<ip::tcp::socket> socket)
    {
      processTask<Messages::MandelMessage, Tasks::MandelTask>(socket);
    };
  m_messageCommand[Messages::Type::NextPrimeMessage] =
    [this](std::shared_ptr<ip::tcp::socket> socket)
    {
      processTask<Messages::NextPrimeMessage, Tasks::NextPrimeTask>(socket);
    };
  m_messageCommand[Messages::Type::TerminateCommand] =
    [this](std::shared_ptr<ip::tcp::socket> socket)
    {
      processTerminateCommand(socket);
    };
}
```

8.2.4 Receiving Messages

The technique for message encoding before sending was presented in Chapter 7, along with how a message is read and decoded once the message type is known. But we still need to understand how an application waits for messages to arrive, determines the message type in order to create the correct message instance, and then passed off to be read and decoded.

Listing 8.13 shows the `waitOnMessage` that is called whenever a compute server makes a connection to the client. Almost all of the code is a lambda that is passed as a parameter to the non-blocking `async_receive` call. This becomes a handler on the `io_service` that is invoked whenever a message is received over this socket. Think carefully about this, for each connected server there is a pending receive handler associated with the socket connection; it is not a single handler for all servers, it is multiple handlers, one for each connected server/socket.

Listing 8.13: Client `waitOnMessage`

```
void DistributedApp::waitOnMessage(ServerID_t serverId)
{
  boost::optional<Server&> server = m_servers.get(serverId);
  server->socket->async_receive(boost::asio::buffer(server->messageType),
    [this, serverId, server]
    (const boost::system::error_code& error, std::size_t bytes)
    {
      if (!error && bytes > 0)
      {
        Messages::Type type =
          static_cast<Messages::Type>(server->messageType[0]);
        m_messageCommand[type](serverId);
        waitOnMessage(serverId);
      }
    });
}
```

The `Server` structure defines a single byte `std::array` that is used as a buffer into which the first byte of a message is received. This array is passed as the first parameter to the `async_recieve` call. The second parameter is the lambda to invoke when the byte is received. Section 7.3 of Chapter 7 showed that the first byte sent as part of a message is the unique message type. The client can receive three different messages from the server: a request for a task, results for a Mandelbrot task, and the results from a prime number task. These three messages correspond to the following enumeration types: `Messages:Type::TaskRequest`, `Messages::Type::MandelResult`, `Messages::Type::NextPrimeResult`. These are the three message handlers registered with the command map presented in Section 8.2.3.

As a bit of sanity checking, the first part of the lambda checks to see if an error occurred on the socket, causing the handler to be invoked. If no error occurred, the rest of the message is processed.

The next bit of code casts the received byte to a `Messages::Type` allowing it to be used as the key for invoking the message handler on the `m_messageCommand` hash table. It is during the this call that the rest of the message is read from the socket, decoded, and processed. Once the call on the command map has completed, a new call to `waitOnMessage` is made in order to place a new handler on the `io_service` to respond the next message received.

An example command map handler is shown in Listing 8.14. This method is invoked when a prime number result message is received from a compute server. The lambda registered in the `waitOnMessage` reads the first byte of the message, then based upon that message, this handler is invoked. This handler creates an instance of the `NextPrimeResultMessage` class, which then finishes reading the data from the socket and decodes the results into the class members.

Listing 8.14: Process Prime Result

```
void DistributedApp::processNextPrimeResult(
    ServerID_t serverId)
{
  Messages::NextPrimeResultMessage taskResult;
  taskResult.read(m_servers.get(serverId)->socket);

  m_lastPrime = taskResult.getNextPrime();
  m_reportPrime = true;
  std::shared_ptr<Tasks::NextPrimeTask> task =
    std::make_shared<Tasks::NextPrimeTask>(
      taskResult.getNextPrime(),
      Tasks::Task::Priority::Three);
  TaskRequestQueue::instance()->enqueueTask(task);
}
```

With the result message processed, the handler remembers the newly generated prime number to the private class member `m_lastPrime`. Next, the `m_reportPrime` boolean is set to `true`, which tells the application to display the `m_lastPrime` value to the console. Finally, a new `NextPrimeTask` is created and placed on the `TaskRequestQueue`. This function completes the prime number computation loop, keeping the computation ongoing throughout the lifetime of the application.

The compute server is substantially similar to the client. It makes a call to the non-blocking `async_receive`, passing a single byte buffer as the first parameter and a lambda to be invoked upon receipt of the byte. Once again, the byte is converted to a `Messages::Type`, the command map handler invoked, and then a new call to `waitOnTask` is made. The code for this method is found in Listing 8.15.

Listing 8.15: Server `waitOnTask`

```
void ComputeServer::waitOnTask(
    std::shared_ptr<ip::tcp::socket> socket)
{
  socket->async_receive(boost::asio::buffer(m_messageType),
    [this, socket](const boost::system::error_code& error, std::size_t bytes)
    {
      if (!error)
      {
        Messages::Type type = static_cast<Messages::Type>(m_messageType[0]);
        m_messageCommand[type](socket);
        waitOnTask(socket);
      }
    });
}
```

8.2.5 Sending Tasks

The `Task` is updated to reflect the nature of a distributed application. A task now knows how to send itself to a compute server via a socket, along with knowing how to return a computed result back to the client. These new capabilities have changed the application interface through the addition of a new `send` method, and a change in the approach to how the `complete` method is implemented. Listing 8.16 shows a portion of the revised `Task` class delcaration.

Listing 8.16: Revised `Task` Declaration

```
class Task
{
public:
  void send(
    std::shared_ptr<ip::tcp::socket> socket,
    boost::asio::strand& strand);

  void complete(boost::asio::io_service& ioService);

protected:
  virtual std::shared_ptr<Messages::Message> getMessage() = 0;
  virtual void completeCustom(boost::asio::io_service& ioService) = 0;
};
```

The new `send` method has the responsibility to send a task message to a connected server, using the socket and strand passed to it. The implementation for this method is almost trivial, as shown in Listing 8.17. The first thing the code does is to call into the derived class `getMessage` method to obtain a fully formed `Message` that is then sent to the compute server.

Listing 8.17: `Task::send` Implementation

```
void Task::send(
  std::shared_ptr<ip::tcp::socket> socket,
  boost::asio::strand& strand)
{
  auto message = getMessage();
  message->send(socket, strand);
}
```

Notice that `Task::getMessage` is a pure virtual method, forcing derived `Task` classes to provide an implementation. The only thing this method needs to do is to construct and return a shared pointer to a `Message` class that represents the work to be done at the compute server. An example implementation for a `getMessage` is shown in Listing 8.18. The code simply creates and returns a shared pointer to an instance of a `MandelMessage` using the attribute values of the `MandelTask` instance.

Listing 8.18: `MandelTask::getMessage` Implementation

```
std::shared_ptr<Messages::Message> MandelTask::getMessage()
{
  return
    std::shared_ptr<Messages::MandelMessage>(
      new Messages::MandelMessage(
        m_id,
        m_startRow, m_endRow,
        m_sizeX,
        m_startX, m_startY,
        m_deltaX, m_deltaY,
        m_maxIterations));
}
```

8.3 Client Notes

The client is the Windows part of the distributed application. It is responsible for the display of the Mandelbrot image, along with display of the prime number generation results. In the same way as the previous chapters, the client is responsible for generating the tasks that need to be computed, whether tasks for computing the Mandelbrot set or tasks for computing the next prime number. The difference now is that the client does not contain any code to perform those computations, it only generates the tasks and distributes them to the connected servers. The remainder of this section details how tasks are distributed to the connected servers, along with describing how a graceful shutdown of the client is performed.

8.3.1 Matching Tasks & Requests

Because the application is now composed of multiple processes, all executing on different computers, it requires a change in how selecting tasks for computation is handled. In the first three versions of the application, tasks were placed on a thread pool. The tasks were then grabbed by an available thread and executed. With the distributed application, the client no longer has that same thread pool, instead, the thread pool is on the server processes, where the tasks are executed. The tasks are generated by the client Mandelbrot application. As described in the first part of the chapter, servers send task requests to the client, and those requests are matched with waiting tasks. This matching takes place on a new singleton object as part of the client, the `TaskRequestQueue`.

The declaration for the `TaskRequestQueue` is shown in Listing 8.19. This class is immediately familiar because of similarities it shares with the `ThreadPool`. It is implemented as a singleton, because there should only be one in existence for any application, along with making it globally accessible. Secondly, it has the same `enqueueTask` method and supporting class members. It diverges in that it is not a thread pool, the addition of the `enqueueRequest` method, and the distribution algorithm that matches tasks with requests and distributes them to the servers.

Listing 8.19: `TaskRequestQueue` Declaration

```
class TaskRequestQueue
{
public:
  static std::shared_ptr<TaskRequestQueue> instance();

  void initialize(
    boost::asio::io_service* ioService,
    ServerSet* servers);
  void terminate();
  void enqueueRequest(ServerID_t request);
  void enqueueTask(std::shared_ptr<Tasks::Task> task);

protected:
  TaskRequestQueue();

private:
  static std::shared_ptr<TaskRequestQueue> m_instance;
  boost::asio::io_service* m_ioService;
  ServerSet* m_servers;
  std::queue<ServerID_t> m_queueRequest;
  std::mutex m_mutexRequest;
  std::condition_variable m_eventRequest;
  std::mutex m_mutexEventRequest;

  ConcurrentPriorityQueue<std::shared_ptr<Tasks::Task>, TaskCompare> m_queueTasks;
  std::condition_variable m_eventTask;
  std::mutex m_mutexEventTask;

  std::shared_ptr<std::thread> m_distributer;
  bool m_distributerDone;

  void distribute();
  void fillRequest(std::shared_ptr<Tasks::Task> task);
};
```

As indicated in the description above, the `TaskRequestQueue` has two containers that hold the task requests and tasks. The task requests are stored in the `m_queueRequest` member, which is a `std:queue`. Just like the thread pool from the chapter on priority, the tasks are stored in a new container, a `ConcurrentPriorityQueue`. This container is a simple thread-safe wrapper around the `std::priority_queue`. Each of these containers are associated with a condition variable, `m_eventRequest` and `m_eventTask` respectively, that are signaled when an item is added.

When the singleton is first used, it creates a private `m_distributer` thread. Listing 8.20 shows the `instance` method where the thread is created. The purpose of this thread is to respond to incoming task requests and tasks, matching and sending them off to be computed. The matching algorithm is discussed later in this section.

Listing 8.20: `TaskRequestQueue::instance`

```
std::shared_ptr<TaskRequestQueue> TaskRequestQueue::instance()
{
  if (m_instance) return m_instance;

  m_instance = std::shared_ptr<TaskRequestQueue>(new TaskRequestQueue());

  m_instance->m_distributer =
    std::make_shared<std::thread>(&TaskRequestQueue::distribute, m_instance);

  return m_instance;
}
```

The `TaskRequestQueue` exposes two primary public methods as its application interface, `enqueueRequest` and `enqueueTask`. The code for these methods is shown in Listing 8.21. These methods work in the same way, adding either a task request or task to the appropriate container. The `enqueueRequest` method is used to add incoming task requests from servers. The `enqueueTask` method is used to add tasks to be computed from the client. Each of these methods signal condition variables that release the thread that attempts to match a request and task. The `enqueueRequest` method uses a mutex to protect the `m_queueRequest` container because it is a non-thread safe container, a `std::queue`. No mutex protection is needed in `enqueueTask` because `m_queueTasks` is a thread safe container.

Listing 8.21: `TaskRequestQueue` Declaration

```
void TaskRequestQueue::enqueueRequest(
    ServerID_t request)
{
  std::lock_guard<std::mutex> lock(m_mutexRequest);
  m_queueRequest.push(request);
  m_eventRequest.notify_all();
}

void TaskRequestQueue::enqueueTask(std::shared_ptr<Tasks::Task> task)
{
  m_queueTasks.enqueue(task);
  m_eventTask.notify_all();
}
```

The heart of the `TaskRequestQueue` is found in the `distribute` method, shown in Listing 8.22. This is the initial method called when the `TaskRequestQueue` thread is started. The purpose of this method is to remove tasks from the `m_queueTasks` container and then call `fillRequest` to match them with a waiting task request. As long as tasks are available, they are removed from the priority queue. Just like the standalone priority application from earlier in the book, tasks of all priorities at, or below, the specified level are checked. When no tasks are available, the method enters an efficient wait state, waiting for the `m_eventTask` condition variable to be signaled.

Listing 8.22: Task Distribution

```
void TaskRequestQueue::distribute()
{
  while (!m_distributerDone)
  {
    auto task = m_queueTasks.dequeue();
    if (task)
    {
      fillRequest(task.get());
    }
    else
    {
      std::unique_lock<std::mutex> lock(m_mutexEventTask);
      m_eventTask.wait(lock);
    }
  }
}
```

Once a task is selected, the `fillRequest` method is called to match it with a task request and send it to the associated compute server; the code for this method is shown in Listing 8.23. This method doesn't return until the task is matched with a request. Even though it is a blocking call, it stays in an efficient loop until a task request is available. This is done by first checking the `m_queueRequest` container for a task request. If one is found, the unique id of the server is captured and the loop is ended. If no task request is available, the method enters an efficient wait state on the `m_eventRequest` condition variable. When a new task request comes in, the condition variable is signaled and the request container is checked again.

Listing 8.23: Filling Task Requests

```
void TaskRequestQueue::fillRequest(
  std::shared_ptr<Tasks::Task> task)
{
  ServerID_t serverId = 0;
  bool done = false;
  while (!done)
  {
    std::lock_guard<std::mutex> lockRequest(m_mutexRequest);
    if (!m_queueRequest.empty())
    {
      serverId = m_queueRequest.front();
      m_queueRequest.pop();
      done = true;
    }
    if (!done)
    {
      std::unique_lock<std::mutex> lock(m_mutexEventRequest);
      m_eventRequest.wait(lock);
    }
  }

  m_ioService->post(
    [this, serverId, task]()
    {
      task->send(
        m_servers->get(serverId)->socket,
        *m_servers->get(serverId)->strand);
    });
}
```

Once a task request and a task are matched, the loop ends and the task is sent to the compute server. This is done by posting an event to the main application `io_service` queue to call the `Task::send` method.

Tasks & Task Requests

Tasks are placed onto the `TaskRequestQueue` in the same way as was done using the `ThreadPool` from previous chapters. The big change introduced with this chapter is distributing those tasks among the various connected computers. This is done, as already described, by matching task requests with tasks.

Task requests are sent from each connected server to the client. The code shown in Listing 8.24 shows the class used to represent the message sent from a server to a client that it can accept a task. The only content of this message is its type, that is enough to indicate its purpose. This class must override the `encodeMessage` and `decodeMessage` methods because they are virtual in the base `Message`, but there is nothing do because of no additional content in the message.

Listing 8.24: `TaskRequest` Message

```
class TaskRequest : public Message
{
public:
  TaskRequest() :
    Message(Messages::Type::TaskRequest)
  {
  }

protected:
  virtual void encodeMessage() override {}
  virtual void decodeMessage() override {}
};
```

The client needs to be able to know what to do when this message is received. Section 8.2.3 described the Command Pattern infrastructure used to register and handle incoming messages, the `TaskRequest` message needs to be registered into this pattern. The code shown in Listing 8.25 shows the code used to register a handler for the `TaskRequest` message. When invoked, the handler makes a call to `processTaskRequest`; the code for this method is shown in Listing 8.26.

Listing 8.25: `TaskRequest` Message Handler

```
m_messageCommand[Messages::Type::TaskRequest] =
  [this](ServerID_t serverId)
  {
    processTaskRequest(serverId);
  };
```

The `processTaskRequest` method follows the same pattern as all message handlers. The first step is to finish reading the message, by calling the overloaded message `read` method (which in this case actually does nothing, because there is no more data to read). The next step is to perform any custom logic based upon receipt of this message. In the case of a `TaskRequest` message, the custom logic is to make a call to the `TaskRequestQueue::enqueueRequest` method, indicating the connected server is available to receive a new task.

Listing 8.26: `processTaskRequest`

```
void DistributedApp::processTaskRequest(ServerID_t serverId)
{
  Messages::TaskRequest taskRequest;
  taskRequest.read(m_servers.get(serverId)->socket);
  TaskRequestQueue::instance()->enqueueRequest(serverId);
}
```

8.3.2 Client Termination

The client application is terminated when the user presses the <ESC> key. When pressed, the Windows message loop terminates and the `DistributedApp::terminate` method is called, shown in Listing 8.27. This method performs a graceful shutdown of the client application.

Listing 8.27: Client Termination

```
void DistributedApp::terminate()
{
  m_running = false;
  TaskRequestQueue::instance()->terminate();

  std::atomic<std::size_t> sendRemaining = m_servers.getServers().size();
  for (auto server : m_servers.getServers())
  {
    Messages::TerminateCommand terminate;
    terminate.send(server.second.socket,
      [&sendRemaining](bool)
      {
        sendRemaining--;
      });
  }

  while (sendRemaining > 0)
    ;

  for (auto server : m_servers.getServers())
  {
    server.second.socket->close();
  }

  m_ioService.stop();
  for (auto& thread : m_threadsIO)
  {
    thread->join();
  }
}
```

The first step in terminating is to internally indicate to the client that it should no longer continue running, this is performed by setting the `m_running` flag to `false`. This flag is tested by the `waitOnConnection` and `waitOnMessage` methods to see if they should continue working or not. Once set to `false`, those methods will not make any further calls to themselves when the `io_service` handler is invoked. When `m_ioService.stop()` is called, this causes any pending handlers to be invoked, but with an error condition indicated. This is how these methods gracefully exit.

The next step is to perform a graceful shutdown of the `TaskRequestQueue`. The `TaskRequestQueue` is the client component responsible for matching tasks and requests, it needs to be terminated in order to stop distributing tasks to workers. The `TaskRequestQueue::terminate` method is shown in Listing 8.28. This code signals the `m_eventTask` and `m_eventRequest` condition variables so that they will come out of their waiting states and terminate. The final statement performs a join operation on the distributor to guarantee it has completed before exiting this method.

Listing 8.28: TaskRequestQueue Termination

```
void TaskRequestQueue::terminate()
{
  m_distributerDone = true;
  m_eventTask.notify_all();
  m_eventRequest.notify_all();

  m_distributer->join();
}
```

With the `TaskRequestQueue` terminated, we know there are no more tasks being sent to the connected compute servers. This makes it safe to now send a message to each of the connected compute servers to terminate. A special message is sent to each server, `Messages::TerminateCommand`. Upon receipt of this message a server will perform a graceful shutdown.

During the send of the `Messages::TerminateCommand` messages, a short lambda is passed in to be called once the send has completed. This lambda keeps track of how many more `Messages::TerminateCommand` messages remain to be sent. Immediately below the loop to send these messages, a busy loop is entered, this loop blocks until all of the `Messages::TerminateCommand` messages have been sent. The use of a busy loop in this context is acceptable because we aren't concerned with performance. It would be possible to make use of a `condition_variable` and some other logic to create an efficient wait state, but doing so results in unnecessarily complex code for no benefit.

Once the termination messages are sent to the servers, it is now safe to close the socket connections to each of those servers. Once those connections are closed, no more work can be sent, therefore it is now safe to stop the `io_service`. As noted above, when this is stopped, all pending handlers are invoked, allowing them to gracefully stop.

The last step is to wait for the threads associated with the `io_service` to complete. A join is performed on each of those threads and once they have all completed, the application ends execution.

8.3.3 Use of `boost::optional` in `ConcurrentPriorityQueue`

Mostly as a side note to describe the use of `boost::optional`, the code in Listing 8.29 shows the `dequeue` implemention of the new `ConcurrentPriorityQueue` class. Because the `boost` library is now being used in the project, the `dequeue` method has been written to take advantage of `boost::optional` as part of the return type. This allows the return type to indicate a `true/false` whether or not a value was returned, and then the value can be obtained. Using `boost::optional` eliminates the need for any kind of *sentinal* values indicating whether a value is valid or not; the value and existence of the value are now managed separately.

Listing 8.29: `ConcurrentPriorityQueue::dequeue`

```
boost::optional<T> dequeue()
{
  std::lock_guard<std::mutex> lock(m_mutex);

  boost::optional<T> item;
  auto itr = m_queue.begin();
  if (!m_queue.empty())
  {
    item = m_queue.top();
    m_queue.pop();
  }

  return item;
}
```

Refer back to Listing 8.22 and notice the `task` variable. The use of `auto` hides the type, but it is a `boost::optional` around a `std::shared_ptr`. The `boost::optional` part of the type is first tested to see if a value exists in the return, and if it does, the `task.get()` call returns the underlying `std::shared_ptr`. This provides a clean separation of the return value from whether or not there is a return value, along with having a natural way (a boolean test) to test for the existence of a value.

8.4 Server Notes

The server's purpose is to perform the heavy computational lifting for the distributed application. With respect to the single system examples from the previous chapters, the server contains all of the task computation code, but none of the user interface. Most of the server code has already been discussed earlier in this chapter, leaving only a couple small pieces left to discuss. The first is how the server computes and then returns results back to the client. The second is how the server performs a graceful shutdown.

The server component of the system is a new process, it has no user interface, therefore, it is written as a console only application. Because it is a console application, unlike the client which is Windows only, this code compiles and runs just fine on Windows, Linux, Mac OS X, or any other platform with a compliant C++11 compiler and ability to compile Boost.Asio.

8.4.1 Computing & Returning Results

Section 8.2.5 discussed how tasks are sent from the client to the servers. Once a server receives a task, it is placed on the server's `ThreadPool` where it will be handled by one of the worker threads. The logic for the `WorkerThread::run` method is exactly the same as with the priority system described in Chapter 4 and Section 4.1.4. What is different is the implementation of the `complete` method of the task. The code for the revised `complete` method is shown in Listing 8.30.

Listing 8.30: Distributed `Task::complete` Method

```
void Task::complete(boost::asio::io_service& ioService)
{
  auto message = this->completeCustom(ioService);

  std::shared_ptr<ip::tcp::socket> socket = m_socket;
  ioService.post(
    [message, socket]()
    {
      message->send(socket);
    });

  std::shared_ptr<ip::tcp::socket> socket = m_socket;
  ioService.post(
    [socket]()
    {
      Messages::TaskRequest command;
      command.send(socket);
    });
}
```

The first step in this method is to make a call into the `Task::completeCustom` method. This is a pure virtual method on the base `Task` class, where derived classes can implement any custom behavior upon the completion of the task computation. One required part of the `completeCustom` method is that the message that contains the results to be sent back to the client is generated and returned. The next part of the method is to post a request to the `io_service` queue to send the results message back to the client. The next part of the method is to post a request to the `io_service` that handles sending a new `TaskRequest` message back to the client, indicating it is available for a new task to be sent back to the server for computation.

An example `completeCustom` method from the `MandelTask` class is shown in Listing 8.31. The only thing this example does is to generate the result message that is sent back to the client. It is necessary for this work to be implemented by the derived classes, because there is no common or generic way results are collected and placed into a message for every possible kind of result.

Listing 8.31: Derived `completeCustom` Method

```
void MandelTask::completeCustom(boost::asio::io_service&
  ioService)
{
  std::shared_ptr<Messages::MandelResultMessage> message =
    std::make_shared<Messages::MandelResultMessage>(
      this->getId(),
      m_startRow,
      m_endRow,
      std::move(m_pixels));

  return message;
}
```

8.4.2 Server Temination

The server process is terminated upon receipt of a `Messages::TerminateCommand`; as noted earlier, the client sends this message during its termination procedure. For reference, the handler for this message is shown in in Listing 8.32, along with a code segment from the `main` function of the server. The code segment from the main server thread shows that it performs a join on the `io_service` thread, waiting until the `io_service` terminates.

Listing 8.32: Server Termination

```
void processTerminateCommand(std::shared_ptr<ip::tcp::socket> socket)
```

```
{
  socket->get_io_service().stop();
}

void main()
{
  ...
  thread.join();
  ThreadPool::instance()->terminate();
  ...
}
```

When the `Messages::TerminateCommand` message is received, the handler simply tells the `io_service` to stop. When the `io_service` is stopped, it causes the `io_service` thread to complete, allowing the join on that thread to fall through. When the join falls through, the server `ThreadPool` terminates, at which point the server process exits.

9 | Scalability - Fault-Tolerant

Some systems require reliable computational results even when the computing environment itself is unreliable. For example, a company that deploys large data centers, comprised of hundreds or thousands of computers can reasonably expect one of these computers to fail on a daily, weekly, or monthly basis. It isn't reasonable for the data center to stop operation while the single faulty system is replaced. Furthermore, it isn't reasonable for whatever computation is currently taking place to have to be restarted because one of the computers failed during the execution. In order to overcome these failure scenarios, one or more strategies must be implemented that allow for graceful recovery. The general category of these strategies is known as *Fault-Tolerance*.

Fault-tolerance is often, but not exclusively, discussed in terms of an unreliable computing environment, with the causes for that unreliability coming from any number of sources. For the work presented in this book I want to narrow, somewhat, the focus of the purpose for adding fault-tolerance to our system. The kind of computing environment I have in mind isn't necessarily unreliable, but rather dynamic in nature, growing and shrinking over time. For example, someone may desire to harness the computing power of systems when not in use during regular work hours, utilizing them in the evenings when no one is working. In order to do this, one needs to have the ability to dynamically add new systems as they become available later in the evening (this capability already demonstrated in Chapter 8), but also gracefully (i.e. not lose work) handle systems that terminate unexpectedly when the regular daytime user logs on. While this is technically an unreliable network of computing resources, it is better thought of as a dynamic environment.

9.1 Introduction

What is fault-tolerance? The subject is broad, far too broad to reasonably cover comprehensively in this book. Instead this section offers a short introduction to the subject, enough to give a sense of the broader subject while focused enough to make sense out of the code presented as the primary part of this chapter.

A good definition of fault-tolerance is continued operation even when some part of the system fails, another is graceful degradation through redundancy. Generally speaking the intention with fault-tolerance is to acheive the following:

- No single point of failure
- Fault isolation
- Fault containment
- Recovery

No single point of failure indicates that no individual system failure can bring the whole to a stop. Achieving no single point of failure is the most challenging part of a fault-tolerant system, the example provided in this chapter gets close, but doesn't fully meet this goal (a future book will present a system that fully acheives this goal). Fault isolation is the ability to isolate the system component responsible for the failure; this may come about through modular system design, failure detection mechanisms, or other means. Fault containment is the concept of not allowing the failure of one system component to propagate throughout the rest of the system. Recovery is the ability for a system to detect a failure has occured and continue correct operation in the face of that failure; this may mean rerouting requests originally sent to a failed component to another component that can complete the request.

How fault-tolerance is acheived comes through a variety of means. A short list of building blocks for fault-tolerance includes:

- Redundancy (space and time)
- Atomic operations or transactions
- Acceptance tests or voting
- Data encoding
- Algorithm diversity
- Multiple correct results
- Elections

Which of these building blocks are used depends upon the requirements of the fault-tolerant system. The most common building block, and the one used in this book, is redundancy. With respect to ease of system design, it is relatively easy to add multiple hard drives versus some other means to overcome the failure of a hard drive. This tends to hold true with other kinds of redundancy. It is fairly straightforward to duplicate resources, making it a popular choice. Other building blocks such as voting, elections, algorithm diversity, and multiple correct results require much more complexity to be added to the system, therefore they tend to be less frequently utilized.

As with everything else, fault-tolerance does not come for free, there are associated costs. The most obvious of these is money. Duplicated hardware costs money, as does duplicated effort to provide multiple implementations of the same algorithms to achieve algorithm diversity. While some things may only cost money (it might be possible to hire multiple engineers to develop multiple algorithms at the same time) it is usually the case that adding fault-tolerance also costs calendar time. A system that has a single algorithm for computing results is less complex than a system that uses multiple algorithms and compares those results before continuing. Not all parts of the more complex system can be implemented in parallel. It is simply going to take more time to develop the multiple algorithm system due to its greater complexity, resulting in a longer time to develop and deploy the system. Futhermore, adding fault-tolerance increases a system's complexity, making it more difficult to design, develop, and maintain.

A few examples of everyday system components that include fault-tolerance as part of their design include: TCP, error-correcting code memory (ECC memory), and RAID. TCP communication is designed with unreliable networks in mind, while providing a *guarantee* of reliable communication. ECC memory is capable of detecting and correcting common kinds of data corruption. Some features of RAID provide for fault-tolerance through the use of redundancy and/or additional error detecting and correcting bits. These examples are not full systems in and of themselves, but represent real-world building blocks upon which a fault-tolerant system may be constructed. Looking back on the previous chapter's use of TCP for communication within the distributed system, we already see the use of fault-tolerance in the system without having to make any special effort.

It is important to note that fault-tolerance is just that and nothing more, it is not *fault-proof.* A fault-tolerant system is designed to handle some classes or specific types of failure, but is not resistant to all failures. The degree to which a system is resiliant in the face of failures owes much to the resources (i.e. money and time) available to design and build aspects of fault-tolerance into the system.

9.2 System Design

The class of fault-tolerance added to the system for this chapter is that of tolerance to unexpected failures in the compute servers. When a compute server fails, for any reason, the whole system must gracefully handle the failure. Consider that a compute server may be in the middle of working on a task for the client when it fails. The system must recognize this failure and redirect that task to another compute server in order to ensure that all computations are completed. Furthermore, it is desired to contain this failure detection and task redirection logic to the core system components, making it as transparent to the application code as possible.

The fundamental system model does not change that much from the previous chapter, as shown in Figure 9.1. The only new addition is that of an *Assigned Work* set. Even though this is an apparently small change, its impact on the implementation is more than trivial, as will be seen in the remainder of this chapter.

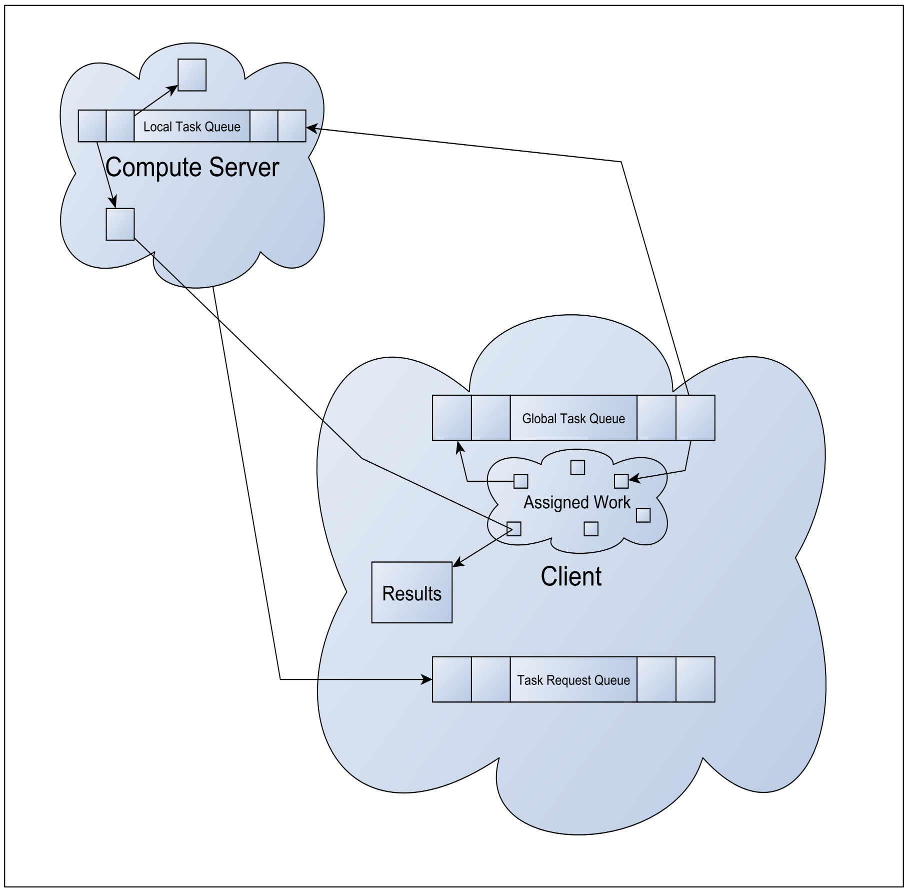

Figure 9.1: Fault-Tolerant Model

The purpose of the Assigned Work set is to have a way of identifying tasks that have been sent out for computation, but have not yet had their results returned. When a task is distributed to a compute server, that task is placed into the Assigned Work set, and given a maximum time in which a result is expected. In the normal case, when no failure has occurred, and when the task results are returned to the client, the task is removed from the Assigned Work set. In the failure case, the system periodically examines the Assigned Work set for any tasks that have not completed in their expected time and returns them to the Global Task Queue to match them with a new task request and sent back out for computation at a new compute server.

This change in the logic of the core client system to handle server failures, along with the existing ability to dynamically add new servers, is what allows the system to gracefully grow or shrink as computing resources come and go. With this model, we have now achieved the goal of a scalable, distributed, and fault-tolerant system. It is scalable in that it takes advantage of new computing resources, both CPU cores and distributed, is distributed, and can handle failures of connected servers.

I acknowledge the system has an important single point of failure at the client. If the client fails, the entire system fails. The solution to this is going to take a lot more work and is beyond the intended scope of this book. A future book is going to tackle this problem in a comprehensive manner.

9.3 Framework Changes

Based upon the new system model presented in Section 9.2, various updates and additions to the core system framework have been implemented. In some places, such as the `Task` class, the changes are almost trivial, but in other places, such as the `TaskRequestQueue` the changes are more significant. Each of the core systems touched as part of adding fault-tolerance to the system is discussed in this section.

Before moving into the discussion of what is changed, it is useful to highlight the parts of the system that have seen no changes. Essentially everything about the server remains untouched. This includes the `ThreadPool`, the `TaskRequest`, and the `WorkerThread`. The entire networking infrastructure and message communication framework is untouched. The `Task` class is slightly modified, described in Section 9.3.1, but not in a way that affects the server. Even though adding (server failure) fault-tolerance to the system is a big deal, the changes are actually localized to the client and do not impact the server.

9.3.1 Updated `Task`

Figure 9.1 shows only the code associated with the updates to the `Task` class. A new `m_duration` class member is added that represents the maximum amount of time it is expected for this task to be computed. The purpose of this duration is for failure detection. If the results for the task are not returned within this duration, a failure is detected and the task resubmitted to another server for computation. The constructor is updated to accept the duration parameter, with the expectation the derived classes set it to a value that is meaningful for the task. Finally, a new `getDuration` helper method is added to allow read-only access to the task duration.

Listing 9.1: Updated `Task` Class

```
class Task
{
public:
  Task(std::chrono::milliseconds duration, Priority priority = Priority::One);

  std::chrono::milliseconds getDuration() const
  {
    return m_duration;
  }

private:
  std::chrono::milliseconds m_duration;
};
```

As a reminder, only the changed/added parts of the `Task` class are shown, all other parts of the class remain, but are not shown in the listing in order to help highlight the changes.

9.3.2 New `AssignedTask`

In order to identify which tasks have been distributed for computation, but have not yet returned a result, a new state (and container) is needed for tasks. The `AssignedTask` class is used as a wrapper around `Task` instances and held in a new container in the `TaskRequestQueue`. This wrapper contains a pointer to the original task, along with a deadline by which the result is expected before the task computation is considered to have failed. The declaration for the class is found in Listing 9.2.

Listing 9.2: `AssignedTask` Class

```
class AssignedTask
{
public:
  explicit AssignedTask(std::shared_ptr<Tasks::Task> task);

  std::shared_ptr<Tasks::Task> getTask()
  {
    return m_task;
  }
  std::chrono::time_point<std::chrono::high_resolution_clock>
    getDeadline() const
    {
      return m_deadline;
    }

private:
  std::shared_ptr<Tasks::Task> m_task;
  std::chrono::time_point<std::chrono::high_resolution_clock> m_deadline;
};
```

When an instance of this class is created, the constructor, shown in Listing 9.3, takes the current time (`now`) and adds the task duration to it in order to come up with the deadline for when the task should have returned a result. This deadline is stored in the `m_deadline` member of the class.

Listing 9.3: `AssignedTask` Constructor

```
AssignedTask::AssignedTask(std::shared_ptr<Tasks::Task> task) :
  m_task(task)
{
  std::chrono::time_point<std::chrono::high_resolution_clock> now =
    std::chrono::high_resolution_clock::now();
  m_deadline = now + task->getDuration();
}
```

When the `AssignedTask` instances are stored in a ordered container, a `std::priority_queue` in our case, a means by which they can be sorted by their deadline is necessary. Therefore, the `AssignedTaskCompare` class shown in Listing 9.4 is necessary. This class overload the parenthesis `()` operator, taking two `AssignedTask` instances and returning a `true/false` depending upon which one is greater than the other. When the `std::priority_queue` is declared in the `TaskRequestQueue`, this class is specified as the comparison functor. Using this scheme makes it possible to easily know which tasks have gone past their deadline; how this is done is discussed in futher detail in Section 9.3.3, which details the updated `TaskRequestQueue`.

Listing 9.4: `AssignedTaskCompare` Class

```
class AssignedTaskCompare :
  public std::binary_function<
```

```
    std::shared_ptr<AssignedTask>,
    std::shared_ptr<AssignedTask>, bool>
{
public:
  bool operator()(
    const std::shared_ptr<AssignedTask> lhs,
    const std::shared_ptr<AssignedTask> rhs) const
  {
    return (lhs->getDeadline() > rhs->getDeadline());
  }
};
```

9.3.3 Updated `TaskRequestQueue`

The most significant changes to the code take place in the `TaskRequestQueue` class. Listing 9.5 shows the additions to the class; everything else presented in Chapter 8 remains part of the class, however with some of the method implementations changed. As noted before, the server code is essentially unchanged. It is the client code where the fault-tolerance logic was added, resulting in the large changes to the `TaskRequestQueue` class. Each of these additions, and changes to existing methods, is detailed next.

Listing 9.5: Updated `TaskRequestQueue`

```
class TaskRequestQueue
{
public:
  bool finalizeTask(uint32_t id, bool forceRemove);

private:
  std::priority_queue<
    std::shared_ptr<AssignedTask>,
    std::vector<std::shared_ptr<AssignedTask>>,
    AssignedTaskCompare> m_queueAssigned;
  std::unordered_map<uint32_t, std::shared_ptr<AssignedTask>> m_mapAssigned;
  std::recursive_mutex m_mutexAssigned;

  void compactQueueAssigned();
  bool isQueueAssignedEmpty();
  void popQueueAssigned();
  bool mapAssignedContains(uint32_t id);
  boost::optional<std::shared_ptr<AssignedTask>> getQueueAssignedTop();
};
```

Let's start by looking at the updated `distribute` method in Listing 9.6. While looking at the revised `distribute` method, refer back to Chapter 8 and Listing 8.22 for the orginal implemention to get a better sense of the scope of the changed logic.

Listing 9.6: Updated `distribute` Method

```
while (!m_distributerDone)
{
  Tasks::Task::Priority currentPriority = m_priority;
  bool donePriority = false;
  while (!m_distributerDone && !donePriority)
  {
    bool distributed = false;
    std::shared_ptr<Tasks::Task> task;

    compactQueueAssigned();

    if (!isQueueAssignedEmpty())
    {
      auto top = getQueueAssignedTop();
      if (top)
      {
        std::chrono::time_point<std::chrono::high_resolution_clock> now =
          std::chrono::high_resolution_clock::now();
        if (top.get()->getDeadline() <= now)
        {
          task = top.get()->getTask();
          popQueueAssigned();
          finalizeTask(task->getId(), true);

          fillRequest(task);
          distributed = true;
        }
      }
    }
    if (!distributed)
    {
      task = m_queueTasks.dequeue(currentPriority);
      if (task)
      {
        fillRequest(task.get());
        distributed = true;
      }
    }
    Tasks::updatePriority(
      distributed,
      m_priority,
      donePriority,
      currentPriority);
  }
  if (!m_distributerDone)
  {
    std::unique_lock<std::mutex> lock(m_mutexEventTask);
    m_eventTask.wait_for(
      lock,
      std::chrono::milliseconds(100));
  }
}
```

The code uses the same logic to stay in the method, waiting for the `m_distributerDone` flag to change to `true`. The fault-tolerant design retains the same priority scheduling algorithm as before, therefore, the same steps and priority loop are used. At this point, the same fundamental loop to keep distributing tasks and working downward through priority is intact. The changes to the method begin inside the priority loop.

The first change is a call to the `compactQueueAssigned` method, which removes items from the assigned queue that have returned results since the last time this loop was executed (all new methods refered to in this section are further detailed below). The next step in the logic is to look at the assigned queue to see if it contains anything (the items in this queue are sorted by their deadline). If it does, the top (i.e. the oldest) item's deadline is examined to see if it is past the current time. If it is, it is removed from the assigned queue (`popQueueAssigned`), the task is finalized (`finalizeTask`), then the task is used to fill the next available request. If the assigned queue does not contain anything, a task is dequeued from the main task queue (`m_queueTasks`) and that is used to fill the next available request.

One additional change is made at the bottom of the loop inside the `if (!m_distributerDone)` condition. Rather than doing an indefinite `.wait` on the condition variable a `.wait_for` is used instead. This is done because it is possible for there to be a single task being computed, and no other tasks waiting to be distributed, and then a failure occurs. The `.wait_for` allows the loop to periodically wake up and check the assigned queue for tasks that have gone past their deadline and attempt to fill them on another server.

The next piece of code to take a look at is the revised `fillRequest` method. The core logic is the same as from Chapter 8, but a few important additions have been made. Listing 9.7 shows the updated sections of this method; for reference, the original method from the previous chapter is located at Listing 8.23. The first new addition happens at the beginning with a call to the `removeDisconnected` method of the `m_servers` member. The purpose of this is to detect servers that are no longer connected (i.e. have failed) and remove them from consideration for filling task assignments.

Listing 9.7: Revised `fillRequest` Method

```
void TaskRequestQueue::fillRequest(std::shared_ptr<Tasks::Task> task)
{
  m_servers->removeDisconnected();

  ...

  m_ioService->post(
    [this, serverId, task]()
    {
      {
        std::lock_guard<std::recursive_mutex> lock(m_mutexAssigned);
        std::shared_ptr<AssignedTask>
          assigned = std::make_shared<AssignedTask>(task);
        m_queueAssigned.push(assigned);
        m_mapAssigned[task->getId()] = assigned;
      }

      task->send(
        m_servers->get(serverId)->socket,
        *m_servers->get(serverId)->strand);
    });
}
```

The next revision to the `fillRequest` method is in the post to the `io_service`. In the previous implementation it was enough to send the task to the selected server and forget about it. Now, with the desire to handle server failures, it is necessary to track the task. This is done through the addition of two new data structures to the `TaskRequestQueue`. The first is a `priority_queue` of `AssignedTasks`, with priority determined by the deadline of the task. The second is an `unordered_map` of `AssignedTasks`.

Why two different data structures to hold the same information? Performance! The `priority_queue` is used during the `distribute` method to find out if there are any tasks past their deadline. A `priority_queue` is the most efficient data structure to do this, only the top of the queue needs to be examined. The `unordered_map`, on the other hand, is used to keep track of which tasks have been assigned, but results not yet returned. When the `compactQueueAssigned` method is invoked during the `distribute` loop, the `m_mapAssigned` is used to know which items should be removed from the top of the `m_queueAssigned` queue. By using a hash table for the `m_mapAssigned`, once again, the best possible performance is achieved. The code in Listing 9.8 shows this process taking place.

Listing 9.8: Compacting The Assigned Queue

```
void TaskRequestQueue::compactQueueAssigned()
{
  std::lock_guard<std::recursive_mutex> lock(m_mutexAssigned);

  bool done = false;
  while (!isQueueAssignedEmpty() && !done)
  {
    auto top = getQueueAssignedTop();
    if (top)
    {
      if (!mapAssignedContains(top.get()->getTask()->getId()))
      {
        popQueueAssigned();
      }
      else
      {
        done = true;
      }
    }
  }
}
```

The `removeDisconnected` method deserves a discussion. The purpose of the method is to go through the set of currently known servers and remove any that are no longer connected. The code for the method is shown in Listing 9.9. As with the others, this method is synchronized on the instance mutex to ensure no other code is accessing the `m_servers` member during its operation.

Listing 9.9: Remove Disconnected Servers

```
void ServerSet::removeDisconnected()
{
  std::lock_guard<std::mutex> lock(m_mutex);

  std::vector<ServerID_t> removeMe;
  for (auto server : m_servers)
  {
    if (!server.second.socket->is_open())
    {
      removeMe.push_back(server.first);
    }
  }
  for (auto server : removeMe)
  {
    m_servers.erase(server);
  }
}
```

A two-step process is used to discover disconnected servers and then remove them from the hash table. The first is to iterate over the set of servers and check to see if their socket connection is still open. If it is no longer open, the `id` (the `first` member of the `std::pair`) of the server is added to the `removeMe` vector. Following this, the servers identified in the first step are removed from the `m_servers` hash table.

When the results for a task are received, the task needs to be removed from tracking so that it isn't considered for resubmission to another server when its deadline passes. To do this, application code must call into the `finalizeTask` of the `TaskRequestQueue`, the code for this method is shown in Listing 9.10.

Listing 9.10: Finalizing A Task

```
bool TaskRequestQueue::finalizeTask(uint32_t id, bool forceRemove)
{
  bool removed = false;

  std::lock_guard<std::recursive_mutex> lock(m_mutexAssigned);

  auto it = m_mapAssigned.find(id);
  if (it != m_mapAssigned.end())
  {
    std::chrono::time_point<std::chrono::high_resolution_clock>
      now = std::chrono::high_resolution_clock::now();
    if (it->second->getDeadline() >= now  || forceRemove)
    {
      m_mapAssigned.erase(it);
      removed = true;
    }
  }

  return removed;
}
```

The first step in this method is to check to see if the task exists in the tracking hash table, `m_mapAssigned`. If it doesn't, nothing is done and `false` is returned. If it is found, the deadline is checked to see if it is in the future. If the deadline is in the future (or the `forceRemove` flag is set), then the task is removed from tracking. On the other hand, if the deadline has passed, it isn't removed. The reason for leaving a task past its deadline in the hash table is to have the function return `false`, indicating to the application code the results for this task should be ignored. This can happen when the system gets too far behind in computing results and re-sends a task out for comptuation even when no server failures have occurred. The fixed estimate for how long a computation should take has its limitations. The next book in this series will solve that problem.

9.4 Application Changes

Because fault-tolerance is applied at the core system level, changes to the application code are essentially trivial. The only change to the application code is to make use of a new method on the `TaskRequestQueue`, a call to the `finalizeTask` method described earlier in Section 9.3.3. Whenever a result is received by the application code, the task associated with the result needs to be finalized.

The code in Listing 9.11 demonstrates the use of the `finalizeTask` method. As described in Chapter 8, reading of the result message is performed first. But now, before the result is processed by application specific code, the task associated with the result must be finalized. If the `finalizeTask` method returns `true` the result may be used, otherwise it must be ignored (likely because it is a duplicate from a resubmitted task).

Listing 9.11: Use of `finalizeTask`

```
void FaultTolerantApp::processMandelResult(
  ServerID_t serverId)
{
  Messages::MandelResultMessage taskResult;
  taskResult.read(m_servers.get(serverId)->socket);
  if (TaskRequestQueue::instance()->finalizeTask(taskResult.getTaskId()))
  {
    m_mandelbrot->processMandelResult(taskResult);
  }
}
```

That is it, no other changes to the application code are necessary!

10 Scalability - Fault-Tolerant - Task Dependencies

The final evolution of the code in this book is to return to the topic of Chapter 5 and incorporate the ability to have the executation of a task, or tasks, depend upon the completion of another task, or tasks, within the context of the distributed and fault-tolerant system. With this final step, we end up with a powerful system, one that exceeds the original goal of the book: a scalable, distributed, and fault-tolerant system, with the added sophistication of being able to handle arbitrary dependency complexity among tasks.

The heavy lifting for the task dependency work was done back in Chapter 5 with the introduction and integration of the `ConcurrentDAG`. As a result, this chapter is fairly light, focusing only on the changes necessary to migrate the fault-tolerant code from Chapter 9 to make use of the `ConcurrentDAG` to ensure task dependecies are supported.

10.1 Framework Changes

The core framework is changed in three fairly minor ways. The first is the complete removal of the concept of priority. While it is possible to have the concept of priority with task dependencies, that is not a topic covered by this book. The next change is to use the `ConcurrentDAG` in place of the `ConcurrentMultiqueue`. The final change is an update to the `finalizeTask` method of the `TaskRequestQueue`.

As just noted, the `m_queueTasks` is changed from a `ConcurrentMultiqueue` to the `ConcurrentDAG` first introduced in Chapter 5. Making this change does result in some other code changes because its interface is different. For example, the `ConcurrentDAG` has the concepts of adding a node (a single task) or an edge (dependent tasks), whereas the `ConcurrentMultiqueue` only has the concept of adding tasks (a node in the `ConcurrentDAG`). Additionally, the `enqueueTask` method that allows dependent tasks to be defined is added to the `TaskRquestQueue`, providing a way to add tasks with dependencies between them.

In addition to using the `ConcurrentDAG`, there is one small change in the code from Chapter 5; this is the same change as was done to the `ConcurrentPriorityQueue` as discussed in Section 8.3.3 of Chapter 8. The return type for the `dequeue` operation is now a `boost::optional`. Again, now that the `boost` library is a part of the project, it makes sense to change this return type.

The change to the `finalizeTask` method is worth a closer look; Listing 10.1 shows the updated code. The first new item is the addition of the `dagRemove` parameter. When this parameter is set to `true`, the task is removed from the DAG (`m_queueTasks`) allowing its dependent tasks to be released for computation. The reason for not removing the task from the DAG is for fault-tolerance. In the case of a server failure it is desired the task remains in the DAG to ensure its dependent tasks are not yet released. The final change is at the bottom of the function, a `.notify_all` call is made on the condition variable `m_eventTask` to allow the distributor to immediately look at any tasks that may have been released for computation.

Listing 10.1: Updated `finalizeTask`

```
bool TaskRequestQueue::finalizeTask(
  uint32_t id,
  bool dagRemove,
  bool forceRemove)
{
  std::lock_guard<std::recursive_mutex> lock(m_mutexAssigned);
  bool removed = false;

  auto it = m_mapAssigned.find(id);
  if (it != m_mapAssigned.end())
  {
    if (dagRemove)
    {
      m_queueTasks.finalize(it->second->getTask());
    }

    std::chrono::time_point<std::chrono::high_resolution_clock>
      now = std::chrono::high_resolution_clock::now();
    if (it->second->getDeadline() >= now || forceRemove)
    {
      m_mapAssigned.erase(it);
      removed = true;
    }
  }

  m_eventTask.notify_all();
  return removed;
}
```

That is all, the changes to the framework are quite modest. This is due to a reasonable (admittedly not perfect) design that abstracts different layers of the system from each other. In particular, this allows a near drop-in replacement of going from a priority queue like data structure with a DAG data structure for the ordering of tasks for computation.

10.2 Application Interface

There are two ways in which tasks are added to the updated framework. The first is to add tasks and have them computed in the order they were placed on the `TaskRequestQueue`. This is as simple as creating the task, same as the previous two chapters (minus priority), and calling `enqueueTask` on the `TaskRequestQueue`. The second way is to add tasks but define a dependency relationship between them. This is the same concept introduced in Chapter 5. Nothing new here from the application interface. The big change, and transparent to the application code, is that the tasks are distributed among as many compute servers as are connected, in addition to being fault-tolerant.

With all of the above in place, the application interface to the `TaskRequestQueue` includes the members shown in Listing 10.2:

Listing 10.2: `TaskRequestQueue` Interface

```
void beginGroup();
void endGroup();
void enqueueTask(std::shared_ptr<Tasks::Task> source);
void enqueueTask(
  std::shared_ptr<Tasks::Task> source,
  std::shared_ptr<Tasks::Task> dependent);
bool finalizeTask(
  uint32_t id,
  bool dagRemove,
  bool forceRemove);
```

To add a task with no dependencies, only the `enqueueTask` method is called. When the task is complete (i.e. results returned), the `finalizeTask` method must be called.

To define dependencies among tasks, all tasks that have dependency relationships must be added as a group. The first step is to call `beginGroup`, followed by using the overloaded `enqueueTask` method that allows two tasks to be added; where the second task cannot be computed until the first task is completed. Once all tasks are added and `endGroup` is called, which allows the scheduling to begin selecting tasks and distributing them for computation. The next two sections of this chapter, Section 10.3 and Section 10.4 show detailed examples of using the interface to add tasks with dependencies.

10.3 Mandelbrot Changes

In Chapters 8 and 9 the `Mandelbrot` class used a couple of member variables, `m_tasksSent` and `m_tasksReceived`, to track when all the results for a single Mandelbrot image were complete and available for display. This works well enough, but I thought it would be interesting to show another way to achieve this same functionality through the use of task dependencies, letting the system framework help track when an image is complete. I am not necessarily recommending this is a better approach (in fact it can be argued it is somewhat less performant because of the networking time involved) but merely making this change to show how task dependencies work.

The first change is to create a new task type, it is called `MandelFinishedTask`; the class declaration for this task is shown in Listing 10.3. This task has no actual computation associated with it, instead, it is used as a way for the application code to send a message to itself when all other Mandelbrot tasks are complete. In other words, its *execution* is dependent upon the completion of all `MandelTask` executions.

Listing 10.3: `MandelFinishedTask` Class

```
class MandelFinishedTask : public Task
{
public:
  MandelFinishedTask() :
    Task(std::chrono::milliseconds(1000))
  {
  }

  MandelFinishedTask(
    std::shared_ptr<ip::tcp::socket> socket,
    Messages::MandelFinishedMessage& message) :
    Task(socket, message.getTaskId())
  {
  }

  virtual void execute() override;

protected:
  virtual std::shared_ptr<Messages::Message> getMessage() override;
  virtual std::shared_ptr<Messages::Message>
    completeCustom(boost::asio::io_service& ioService) override;
};
```

With the new `MandelFinishedTask` available as a kind of application signal, it can be used in combination with the existing `MandelTask` task to inform the application code when the full Mandelbrot image is available. Take a look at the updated `startNewImage` method in Listing 10.4 to see how the task dependencies are defined.

Listing 10.4: Generation Mandelbrot Tasks

```
void Mandelbrot::startNewImage()
{
  TaskRequestQueue::instance()->beginGroup();

  auto taskFinished = std::make_shared<Tasks::MandelFinishedTask>();

  double deltaX = (m_mandelRight - m_mandelLeft) / m_sizeX;
  double deltaY = (m_mandelBottom - m_mandelTop) / m_sizeY;

  for (auto row : IRange<uint16_t>(0, m_sizeY - 1, MANDEL_COMPUTE_ROWS))
  {
    auto task = std::make_shared<Tasks::MandelTask>(
        row,
        std::min(row + MANDEL_COMPUTE_ROWS - 1, m_sizeY - 1),
        m_sizeX, m_mandelLeft, m_mandelTop + row * deltaY,
        deltaX, deltaY,
        MANDLE_MAX_ITERATIONS);
    TaskRequestQueue::instance()->enqueueTask(
      task,
      taskFinished);
  }

  TaskRequestQueue::instance()->endGroup();
}
```

The first step in this method is to call `beginGroup` to ensure all of the Mandelbrot tasks are added before any are considered for scheduling. The second step is to create the `MandelFinishedTask` so that it can be identified in the `enqueueTask` calls as a dependent. The third step is to define `taskFinished` as being dependent upon every other `MandelTask` computation before it can be sent out for *execution*. Finally, the `endGroup` method is called on the `TaskRequestQueue` which releases all of these tasks for computation.

In the `FaultTolerantApp` class, a new `processMandelFinishedResult` handler is defined for the `MandelFinishedTask`, shown in Listing 10.5. When the `MandelFinishedTask` result is sent back to the client this handler is inovked, and this handler is guaranteed to only be invoked after all other `MandelTask` results have been sent back because it is dependent upon all of those tasks first having been executed. At this point, it is known the full new Mandelbrot image is returned and a new one can be generated.

Listing 10.5: `MandelFinishedTask` Handler

```
void FaultTolerantApp::processMandelFinishedResult(
  ServerID_t serverId)
{
  Messages::MandelFinishedResultMessage taskResult;
  taskResult.read(m_servers.get(serverId)->socket);

  if (TaskRequestQueue::instance()->finalizeTask(
    taskResult.getTaskId(), true, true))
  {
    m_mandelbrot->processMandelFinishedResult(taskResult);
  }
}
```

The code for this handler follows the familiar pattern already established in the previous chapter. First step is to finish reading the message, the next step is to finalize the task, and the final step is to do something application specific based upon the result. In this case, the application specific code is to call back into the `Mandelbrot` class to let it know the full image is complete. At this point, the `Mandelbrot` code sets the `m_inUpdate` flag to `false`, which allows another set of Mandelbrot computation tasks to be generated when necessary.

Having this new task added removes the need for the member variables in the `Mandelbrot` class to track how many tasks were sent, received, and constantly checking them to see if the correct number have been returned. Instead, that accounting is implicit information contained in the graph representation of the Mandelbrot tasks. When the `MandelFinishedTask` result shows up, that is the signal the full image is complete, no other accounting is necessary.

10.4 Complex Demonstration

This section provides an example that hints at the kind of complexity supported by this framework, it demonstrates the creation of a non-trivial task dependency graph. The diagram in Figure 10.1 shows a graph of tasks and their dependencies. The tasks at the top of the graph (8 through 15) have no dependencies and therefore, can be computed in any order. The second level of tasks (4 through 7) have dependencies on the first level of tasks, this continues through the third level. Next, task 1 shows dependencies on tasks 2 and 3, which have cascading dependencies on all other tasks, meaning that task 1 can only be computed after all other tasks above it in the graph have completed their execution. Once task 1 has completed its execution, tasks 16 and 17 can compute in parallel, and this continues until the last row of tasks (22 through 29) are free from dependencies and can compute in parallel.

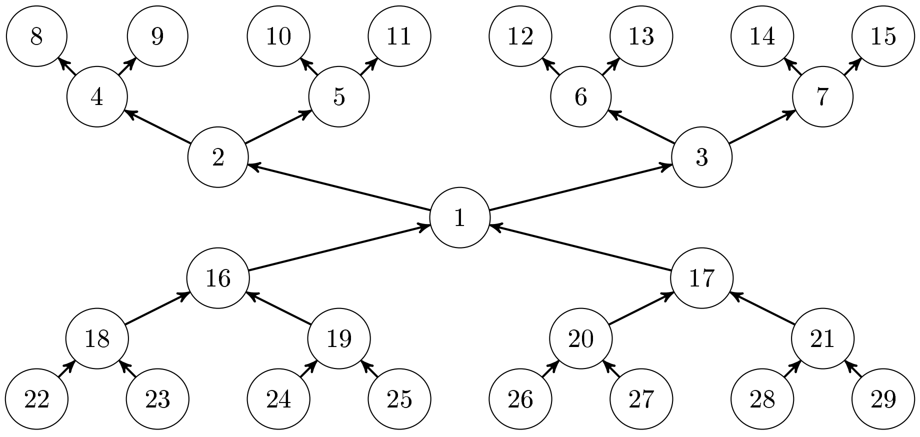

Figure 10.1: Complex DAG

The code involved in creating this task graph is relatively straightforward, following the pattern we know by now. The first step involved writing a new task class for the demonstration. The next step was to create the two messages used to send the task to a compute server and return the results back to the client. Finally, some application code was written to define the relationships among the tasks.

The task class for this demonstration is named `DAGExampleTask`, its class declaration is found in Listing 10.6. This code should be quite familiar by now. The only items of note are the `name` constructor parameter and the private data member `m_name`. When an instance of this task is created, it is given a `std::string` name that is sent to the compute server and returned as part of the result, allowing the application to report to the console the name of the task results just received.

Listing 10.6: `DAGExampleTask` Class

```
class DAGExampleTask : public Task
{
public:
  DAGExampleTask(std::string name) :
  Task(std::chrono::milliseconds(5000)),
    m_name(name)
  {
  }

  DAGExampleTask(std::shared_ptr<ip::tcp::socket>
    socket,
    Messages::DAGExampleMessage& message) :
    Task(socket, message.getTaskId()),
    m_name(message.m_name)
  {
  }

  virtual void execute() override;

protected:
  virtual std::shared_ptr<Messages::Message> getMessage() override;
  virtual std::shared_ptr<Messages::Message>
    completeCustom(boost::asio::io_service& ioService) override;

private:
  std::string m_name;
};
```

The `DAGExampleTask` doesn't perform any real computation, instead it sleeps for 4 seconds, as shown in Listing 10.7. The reason for having the execution sleep for this amount of time is to give time for one to progressively watch the results returned and displayed to the console. Note back in Listing 10.6 this `DAGExampleTask` constructor is defined to have a fault-tolerant timeout of 5 seconds (5000 milliseconds), 1 second longer than the execution. For any reason, if you change the execution sleep period, make sure to adjust the fault-tolerant timeout to be longer than the sleep.

Listing 10.7: `DAGExampleTask` Execution

```
void DAGExampleTask::execute()
{
  std::this_thread::sleep_for(std::chrono::milliseconds(4000));
}
```

Once the results for a `DAGExampleTask` are returned to the client, the `processDAGExampleResult` method is invoked, as shown in Listing 10.8. The only thing the handler does is to report the name of the task to the console.

Listing 10.8: `DAGExampleTask` Results Handler

```
void FaultTolerantApp::processDAGExampleResult(
  ServerID_t serverId)
{
  Messages::DAGExampleResultMessage taskResult;
  taskResult.read(m_servers.get(serverId)->socket);
  if (TaskRequestQueue::instance()->finalizeTask(
    taskResult.getTaskId(), true, true))
  {
    std::cout << "DAG Example Task: " << taskResult.getName() << std::endl;
  }
}
```

The big question is, with such a simple interface for adding tasks, how can such a complex dependency graph as shown in Figure 10.1 get defined. It turns out to be quite easy, refer to the code in Listing 10.9 to see how it is done.

Listing 10.9: Defining Task Dependencies

```
TaskRequestQueue::instance()->beginGroup();

auto task1 = std::make_shared<Tasks::DAGExampleTask>("1");
auto task2 = std::make_shared<Tasks::DAGExampleTask>("2");
auto task3 = std::make_shared<Tasks::DAGExampleTask>("3");
auto task4 = std::make_shared<Tasks::DAGExampleTask>("4");
auto task5 = std::make_shared<Tasks::DAGExampleTask>("5");
auto task6 = std::make_shared<Tasks::DAGExampleTask>("6");
auto task7 = std::make_shared<Tasks::DAGExampleTask>("7");
auto task8 = std::make_shared<Tasks::DAGExampleTask>("8");
auto task9 = std::make_shared<Tasks::DAGExampleTask>("9");
auto task10 = std::make_shared<Tasks::DAGExampleTask>("10");
auto task11 = std::make_shared<Tasks::DAGExampleTask>("11");
auto task12 = std::make_shared<Tasks::DAGExampleTask>("12");
auto task13 = std::make_shared<Tasks::DAGExampleTask>("13");
auto task14 = std::make_shared<Tasks::DAGExampleTask>("14");
auto task15 = std::make_shared<Tasks::DAGExampleTask>("15");

 ...

TaskRequestQueue::instance()->enqueueTask(task2, task1);
TaskRequestQueue::instance()->enqueueTask(task3, task1);
TaskRequestQueue::instance()->enqueueTask(task4, task2);
TaskRequestQueue::instance()->enqueueTask(task5, task2);
TaskRequestQueue::instance()->enqueueTask(task6, task3);
TaskRequestQueue::instance()->enqueueTask(task7, task3);
TaskRequestQueue::instance()->enqueueTask(task8, task4);
TaskRequestQueue::instance()->enqueueTask(task9, task4);
TaskRequestQueue::instance()->enqueueTask(task10, task5);
TaskRequestQueue::instance()->enqueueTask(task11, task5);
TaskRequestQueue::instance()->enqueueTask(task12, task6);
TaskRequestQueue::instance()->enqueueTask(task13, task6);
TaskRequestQueue::instance()->enqueueTask(task14, task7);
TaskRequestQueue::instance()->enqueueTask(task15, task7);

TaskRequestQueue::instance()->endGroup();
```

The code in this listing shows the definition of the tasks in the top half of the example shown in Figure 10.1 to reduce the length of the listing in the book; the full code is available as part of the source code for the book. The first part of the code is to create the different tasks that are part of the dependency graph, no dependencies are defined at this point. The next step is to add tasks and define the dependencies as they are added to the `TaskRequestQueue`.

The order in which tasks and their dependencies are added does not matter, as is seen in this code.

The first two tasks added are `task2` and `task1`. The call using `enqueueTask(task2, task1)` identifies `task1` as being dependent upon the completion of `task2`. The next call using `enqueueTask(task3, task1)` identifies `task1` as being dependent upon the completion of `task2`. At this point, we have identified `task1` and being dependent upon `task2` and `task3`. It is okay that `task1` is used in two different calls to `enqueueTask`. It doesn't mean that `task1` will be computed twice, it is still only added once, but the additional dependency relationship between the two tasks is added. Once a task is added to the framework, any subsequent `enqueueTask` calls refering to the same task do not result in duplicating the task. The framework recognizes the task by its unique identifier and guarantees it exists only once.

After tasks 1, 2, and 3 are added, the process continues from the middle of the graph up to the top, finishing with the top row of tasks, tasks 8 through 15. When the call using `enqueueTask(task4, task2)` is made, it defines that `task4` must complete before `task2`. Because of the existing dependency between `task2` and `task1`, we are guaranteed that `task1` will only execute after `task2` and `task3`.

The order in which the task dependencies are made does not matter. The code in Listing 10.9 for enqueuing the tasks can be reversed and the same graph and order of execution will occur; in fact, any ordering of the `enqueueTask` calls creates a correct graph definition.

With all of this code in place, it is time to run the application and see the results. The output shown in Figure 10.10 comes from an example run on my computer using the graph from Figure 10.1. In looking through it, you can see the tasks at one level all finish before any tasks in the next level of the graph finish. Watching it in real time, the following groups complete in order: (12, 15, 14, 8, 9, 10, 11, 13), (7, 6, 5, 4), (3, 2), (1), (17, 16), (19, 20, 18, 21), (27, 29, 22, 28, 26, 25, 24, 23).

Figure 10.10: Demonstration Output

```
DAG Example Task: 12
DAG Example Task: 15
DAG Example Task: 14
DAG Example Task: 8
DAG Example Task: 9
DAG Example Task: 10
DAG Example Task: 11
DAG Example Task: 13
DAG Example Task: 7
DAG Example Task: 6
DAG Example Task: 5
DAG Example Task: 4
DAG Example Task: 2
DAG Example Task: 3
DAG Example Task: 1
DAG Example Task: 17
DAG Example Task: 16
DAG Example Task: 19
DAG Example Task: 20
DAG Example Task: 18
DAG Example Task: 21
DAG Example Task: 27
DAG Example Task: 29
DAG Example Task: 22
DAG Example Task: 28
DAG Example Task: 26
DAG Example Task: 25
DAG Example Task: 24
DAG Example Task: 23
```

Remember that while the tasks in this graph are being executed, it is possible to manipulate the

Mandelbrot view, with those tasks being executed (and completed) in parallel on the other available distributed CPU cores. Additionally, for this demonstration I have commented out the prime number computation, but it could be added back in and all types of tasks can be executed in parallel, while still having the correct dependencies occuring, in addition to gracefully handling any server failures through the fault-tolerant component of the framework.

10.5 Summary

The work presented in this chapter represents the end of a long progression made through this book. The framework presented in this chapter is a sophisticated system, one that meets and actually exceeds the original stated purpose for the book. With this framework we now have a scalable, distributed, and fault-tolerant system. Additionally, we have two different ways of ordering tasks. From Chapter 9 the tasks may be ordered by priority, and this chapter adds the ability to define complex dependency relationships among tasks. I also like to think the code for providing these powerful capabilities is straightforward and understandable, not having unnecessary complexity, thereby making it within the grasp of as wide an audience as possible.

A | Misc. Code

This appendix contains various snippets of code referred to, but not shown, in the main text of the book. Each code snippet is accompanied by a description of what it does and any interesting parts of the implementation.

A.1 IRange

One of the missing pieces from the updated standard libary is a way to define an integral range over which a range-based for loop can iterate. Part of what we are trying to do with C++11 is to get away from counted loops wherever possible. To overcome this limitation I have written an `IRange` template class that provides this capability, shown in Listing A.1. The class is templated on type `T`, allowing the developer to choose the integral type most appropriate for the context (e.g., signed or unsigned).

Listing A.1: `IRange` Class

```
template <typename T>
class IRange
{
public:
    class iterator
    {
    public:
        T operator *() const { return m_position; }
        const iterator &operator ++()
        {
            m_position += m_increment;
            return *this;
        }

        bool operator ==(const iterator &rhs) const
        {
            return m_position == rhs.m_position;
        }
        bool operator !=(const iterator &rhs) const
        {
            return m_position < rhs.m_position;
        }

    private:
        iterator(T start) :
            m_position(start),
            m_increment(1)
        { }
        iterator(T start, T increment) :
            m_position(start),
            m_increment(increment)
        { }

        T m_position;
        T m_increment;

        friend class IRange;
        };

    IRange(T begin, T end) :
        m_begin(begin),
        m_end(end + 1)
    { }

    IRange(T begin, T end, T increment) :
        m_begin(begin, increment),
        m_end(end + 1, increment)
    { }

    iterator begin() const { return m_begin; }
    iterator end() const { return m_end; }

private:
    iterator m_begin;
    iterator m_end;
};
```

This class works by storing only the beginning and ending values of the range, and the current position of the iterator. Although this requires an object instance to be created at runtime, it is lightweight with respect to the time to construct and its memory footprint.

B | Introduction to CMake

All of the code samples provided as part of this book are developed as cross-platform C++ code (well, with the exception of some Windows client code) and utilize CMake[1] to generate project files. In recent years I have *seen the light* and now fully embrace and use CMake for cross-platform development using C++. CMake is an open-source build system that works across a variety of platforms, including Windows, Linux, MacOS, and many others. The concept of CMake is that of defining a *meta* project build description from which the CMake software can then generate native project files for build systems such as Visual Studio, Eclipse, or even good old fashioned makefiles.

The purpose of this appendix is to help familiarize those new to CMake with enough information to understand how to generate native project files from the provided source code. To get up and running with the provided code samples, Section B.1 is enough. it describes how to use CMake to generate a native project for your preferred build system. For those interested in a deeper understanding of the syntax of the CMake configuration files, please visit the CMake web site at http://www.cmake.org.

B.1 Using CMake

The first step that needs to be accomplished is to download and install the CMake system to your computer. CMake is provided as both a command line and GUI-based utility. I'm a bit of a GUI person, therefore that is the general perspective this appendix provides, but everything described herein can be done from either the GUI or command line. For most users, it is enough to download and install from the installers provided on the CMake website. Alternatively, for Linux systems such as Ubuntu, using a package installer such as `apt` is recommended. When using a package installer it might be necessary to install both `cmake` and something like `cmake-qt-gui` (in the case of Ubuntu).

B.1.1 Generating Project Files

Inside each sample project folder is a file named, `CMakeLists.txt`. This file provides the instructions for CMake on how to generate native project files for your system. The rest of the files, usually `.cpp` and `.hpp` files, in the folder are the source files. The source files and the `CMakeLists.txt` are all that is needed to generate the project files.

When using CMake it is important to keep the source files and the project files in separate folders. If you are used to something like Visual Studio where the project and source files are located in the same folder tree structure, this is a different approach. When CMake generates the build environment, it correctly generates links back to the source folder. Using the `RangedFor` example from the book, a possible approach to laying out a folder for the source and project files may look like the example if Figure B.1.

Figure B.1: Source and Build Folder Layout

```
/Chapter-Cpp11
```

[1] http://www.cmake.org

```
/Chapter-Cpp11/RangedFor
/Chapter-Cpp11/Build
/Chapter-Cpp11/Build/RangedFor
```

The `CMakeLists.txt` file and the source files are all located in the `/Chapter-Cpp11/RangedFor` folder. Following generation of the project files, they are located in the `/Chapter-Cpp11/Build/RangedFor` folder.

Use the following steps to generate the project file for your system, using the `RangedFor` example from 2:

Step 1 Start the CMake GUI application.

Step 2 Select the `Browse Source...` button.

Step 3 Navigate to the `RangedFor` folder inside of the C++11 chapter source code.

Step 4 Select the `Browse Build...` button.

Step 5 Navigate to a *build* folder location.

Step 6 Press the `Configure` button.

Step 7 The first time `Configure` is pressed for a project, a dialog is presented asking for what type of generator to use for the project. Select the appropriate generator, such as `Visual Studio 12`.

Step 8 Often times you'll need to press `Configure` again, due to some value(s) being displayed in red.

Step 9 Press the `Generate` button. Following this step, the project files will now be created and placed in the build folder.

With the project files generated, it is now possible to build the code. In the case of generating a Visual Studio project navigate to the `/Build`, and appropriate sub-folder, then open the `.sln` file and build the solution. In the case of a Unix/Linux makefile, navigate to the `/Build`, and appropriate sub-folder, then type `make` to build the project.

C Mandelbrot Set

The computation and visualization of the Mandelbrot set is used as the basis for demonstrating the techniques presented in this book. Its computation has the features that work well for these topics: computationally intensive, asymmetric complexity, and a visually pleasing result. The purpose of this appendix is to describe its nature and computation for those not already familiar.

C.1 Description

The Mandelbrot set is a fractal[1] named after its discoverer Benoit Mandelbrot[2]. The Mandelbrot set is visually represented as a two-dimensional image because the numbers in the set lie in the complex number plane. The image shown in Figure C.1 shows an example visualization of the set.

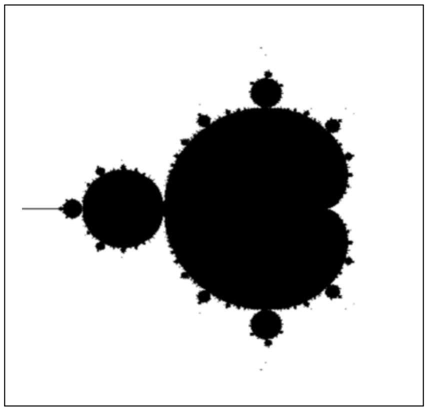

Figure C.1: Mandelbrot Overview

The black portion of the image are the points considered to be part of the set, everything else is not. Typically the points outside the set are colored based upon how many iterations it took to determine the point is not part of the Mandelbrot set. In fact, this is the technique used by the demonstration code associated with this book; Section C.3 briefly references how the coloring of the points is determined by the sample code.

C.2 Computation

The Mandelbrot set is given by Equation C.1, where $z_0 = C$, and C is a complex point in the plane. At any point while iterating this equation, if the distance of z_{n+1} with respect to the origin is greater

[1]http://en.wikipedia.org/wiki/Fractal
[2]http://en.wikipedia.org/wiki/Benoit_Mandelbrot

than 2, then C is not part of the set. Alternatively, if the distance is less than 2 and some maximum number of iterations has been met, then then C is considered to be part of the set. This iterative equation is evaluated for every point over some region of the complex plane.

$$z_{n+1} = z_n^2 + C \tag{C.1}$$

The code in Listing C.1 shows how to compute the number of iterations taken to determine whether or not a point in the complex plane ($C = (x0, y0)$) is part of the Mandelbrot set. The demonstration code associated with this book uses a modified version this listing to perform this same computation.

Listing C.1: Mandelbrot Point Computation

```
uint32_t computePoint(double x0, double y0)
{
  double x = 0;
  double y = 0;
  uint32_t iterations = 0;
  bool done = false;

  while (!done)
  {
    double tempX = x * x - y * y + x0;
    y = 2.0 * x * y + y0;
    x = tempX;

    double distance = x * x + y * y;
    // Saving sqrt by comparing vs 4
    if (distance > 4.0 || iterations >= MAX_ITERATIONS)
    {
      done = true;
    }
    iterations++;
  }

  return iterations;
}
```

The parameters `x0` and `y0` are the coordinates of a point in the complex number place. This point is C of the equation, which means it is also z_0; the initial value for the iterative computation. Remember, the mathematics are in terms of numbers on the complex plane. With this in mind, the first two lines that follow the `while (!done)` statement are the computation for z_{n+1} in the iterative sequence. The next value of `y` is already updated, the third line then is to capture the updated value of `x`. With the next value of z computed, it is time to test the distance of it from the center of the complex number plane. The square of the distance is computed and stored in `distance`. The reason for storing the square of the distance is to save the (expensive) computation of taking a square root. With the square of the distance the comparison is made against the value of 4, rather than 2; with 4 being the square of 2. At the same time, the test for the maximum allowed iterations is made. If either of their conditions is true, the iterative computation is terminated and the final `iterations` result is returned.

What makes the Mandelbrot set a great example for this book is that the number of iterations for each point varies from one point to the next. This results in the computational complexity varying over the image, especially when lines, or groups of lines, are used as the basic comptuational building block. Having this computational asymmetry over the image works well to demonstrate the computational load balancing that occurs through the application of the techniques described in this book.

C.3 Visualization

To create a visualization of the Mandelbrot set, all points within some complex region must be computed. Given that the entire set, by definition, must be within a distance of 2 from the origin, it only makes sense to define a plane that lies in the range of $(-2, -2)$ and $(2, 2)$. This complex region must be mapped onto a rectangular region of image pixels, for example a 1000 x 1000 pixel sized image. Given this mapping, each pixel is then associated with a complex number. This complex number is then fed into a function that computes the number of iterations for which the equation was evaluated. Finally, the number of iterations is used to determine the color for that pixel.

There are many different ways to decide the color for a Mandelbort visualization. The simplest is to select from two colors, for example black or white, depending upon whether or not the pixel is part of the set. Another is to use color to indicate the number of iterations used to calculate the pixel set membership. In order to to this a range of colors is defined and evenly divided based upon the the maximum possible iterations, then a color is selected based upon the iteration count. Another is to create a histogram that tracks how many times (the frequency) each iteration count was used. Then associate a color with each bucket in the histogram and color. Another is to use a continuous, or smooth, coloring technique that eliminates the *banding* associated with many of the other coloring algorithms. The examples in this book all use a smooth coloring technique.

For those interested in more detail about the Mandelbrot set and the various coloring algorithms, please visit http://en.wikipedia.org/wiki/Mandelbrot_set.