USING WRAPPERS TO SIMPLIFY TASK PARALLEL PROGRAMMING

Michael Graf and David P. Bunde
Department of Computer Science
Knox College
{mgraf, dbunde}@knox.edu

ABSTRACT

Teaching parallel programming principles to inexperienced students is challenging because they have difficulty identifying and focusing on the key ideas. With this in mind, we present a simplified interface for the executor framework built into Java. The interface hides configuration choices and complicated features, leaving students with a simple work queue into which they can insert tasks. We also evaluate our interface with a visually-interesting sample problem (heat diffusion), which we provide as a sample assignment.

INTRODUCTION

Computer hardware is changing, with multicore processors now the norm. Thus, parallel programming has become a crucial skill for students, who will spend their careers programming systems with increasing core counts. Parallel programming has long been recognized as difficult and hard to teach, however. In particular, one would like to introduce parallel concepts in early courses to establish a strong foundation in parallel thinking, but this is difficult because of the students' lack of background. In this paper, we focus on introducing task parallelism to CS 2 students with a Java background.

Our goal is to introduce parallelism using high-level constructs and motivating examples. The construct in this paper is a work queue, a buffer into which tasks are submitted. The queue is associated with a thread pool, a collection of threads that work on its tasks. This construct separates parallel programming into two parts, identifying independent tasks and the mechanism for running queued tasks. For introductory students, we advocate exploiting this separation by providing an implementation of the execution mechanism. This frees the students from concerns of task scheduling and resource management so they can focus on tasks and dependencies.

A base implementation of a work queue is provided by Java's executor framework, but its use requires significant code for configuration and error handling. We introduce three wrapper classes to hide this complexity. They specify reasonable default behavior and provide a clean implementation of the work queue abstraction. This allows the student programmer to focus on the "big picture". The instructor can return to the details once students master the main concepts. The approach is similar in spirit to other efforts to facilitate parallel programming in early courses using supportive frameworks such as event-based parallel programming [2], Java's fork-join framework for divide and conquer algorithms [5], WebMapReduce [4], and a wrapper for Java's `Thread` class [1].

To demonstrate our wrapper classes and provide a sample assignment to grab students' attention, we present a graphical application that animates a heat diffusion simulation. Heat diffusion is a classic problem in computational science and not a new example, but we provide students with a serial implementation that handles the graphics so that they "just" have to incorporate parallelism. Assignments that generate graphics

have been found to be particularly useful in teaching parallelism since the speedup is easily seen and bugs can be found via anomalies in the appearance of the output (e.g. [3], [6]).

We plan to use these materials in CS 2 by presenting the concept of tasks and the wrapper classes before having the students parallelize the heat simulator. In higher-level courses, a next step could be to open the wrapper classes and explain why the hidden details are necessary in general. Despite the details we hide, our materials can demonstrate several concepts, including parallel speedup, load balance (using an uneven split), and overhead (creating too many tasks).

Code for our wrapper classes and given code for the sample application are available at `http://faculty.knox.edu/dbunde/pubs/javaHeat/`.

EXECUTOR FRAMEWORK

Now we describe the `Executor` interface and related classes from `java.util.concurrent`, collectively called the executor framework. Following that, we present our wrapper classes to encapsulate some of the details of using these classes.

Java's Executor Framework

The executor framework provides a high-level way for the programmer to write parallel code. Rather than explicitly creating and managing threads, the programmer submits tasks to the framework, which buffers them in a work queue and operates a thread pool to complete them. This frees the programmer to focus on the work to be performed rather than the number of threads to create or what each of them should do. Since each thread runs many tasks, the framework also reduces thread creation overhead.

Tasks submitted to the executor framework are objects from classes that implement either the `Callable<V>` or `Runnable` interfaces depending on whether the task returns a value or not. Each of these interfaces has a single method called to execute the task; `V call()` in `Callable` and `void run()` in `Runnable`.

The `ExecutorService` interface has methods `execute` and `submit` that take `Runnable` and `Callable` objects, respectively. Both methods add their argument to the work queue. Other important methods include `shutdown` and `awaitTermination`. The `shutdown` method begins cleaning up the thread pool for a graceful termination. Once this method is called, no new tasks can be submitted to the service. A later call to `awaitTermination` blocks until all tasks finish or a specified timeout is reached.

Several types of `ExecutorService` objects can be created by calling static methods of the `Executors` class. The method `newFixedThreadPool` returns a thread pool with a specific number of threads. `newCachedThreadPool` returns a thread pool that creates new threads when needed but reuses them when possible. It is useful when many short-lived tasks are created. `newScheduledThreadPool` returns a pool for tasks that run after a specified delay or periodically.

Figure 1 shows a code fragment that prints "Hello" and "World" using the executor framework. This code creates a fixed-size thread pool with one thread per core. `ParallelPrint` is a (programmer-defined) class that stores a string and whose `run` method prints it. After the two tasks have been submitted, the pool is shut down and waited on. The while loop is needed in case `awaitTermination` times out before the service is ended (unlikely because of the timeout used, but technically possible).

```
public static void main(String[] args) {

   //create queue and 1 worker thread per core
   int num = Runtime.getRuntime().availableProcessors();
   ExecutorService e = Executors.newFixedThreadPool(num);

   //submit tasks
   e.submit(new ParallelPrint("Hello"));
   e.submit(new ParallelPrint("World"));

   //wait for all tasks to complete
   e.shutdown();
   try {
      while(!e.awaitTermination(Long.MAX_VALUE,TimeUnit.SECONDS))
         /* empty body */ ;
   } catch(InterruptedException ex) {
      /* print error mesg and exit */
   }
}
```

```
SimpleExecutor e = new SimpleExecutor();
```

```
e.terminate();
```

Figure 1. Main method for "Hello World!" using Java's executor framework. Text to right shows simplified versions of two code paragraphs using our wrapper classes. This program requires a `ParallelPrint` class implementing `Runnable` whose `run` method prints its message.

In order for tasks to return values, we need the concept of a *future*. Obviously the value cannot be returned until it is generated, but a future is an object in which the return value will be stored once it is available. In the meantime, the future provides a way to refer to the eventual return value. Futures also typically provide a way to check whether the return value is available and to wait until it is.

In the context of the executor framework, `ExecutorService`'s `submit` method takes a `Callable<V>` object encapsulating the task with return type `V` and returns a `Future<V>` object. The most important method of `Future<V>` is `get`, which blocks until the submitted task is complete and its return value is available. `Future<V>` also provides the `cancel` method to deschedule the task, and `isDone` to check whether the task has finished. (You can also pass a `Runnable` object to `submit`, in which case a `Future<?>` is returned; this allows access to `cancel` and `isDone` even for tasks without return values.)

Simplified Wrapper Classes

Although the executor framework simplifies the programmer's job in some ways, it is still complex. To simplify the code and allow beginning students to focus on the key principles of the work queue, we created a class `SimpleExecutor` as a wrapper for the executor framework. The first thing it does is to shield students from configuring their thread management policy. Instead, it makes the reasonable default choice of creating a fixed-size thread pool with one thread per core. Because of this, we can use the single line

```
SimpleExecutor e = new SimpleExecutor();
```

to replace the first "paragraph" of `main` in Figure 1 (as shown on the right side).

`SimpleExecutor` also hides the complexity in the third paragraph of Figure 1. This entire paragraph is replaced with the line

```
e.terminate();
```

which makes a new blocking call that combines the functionality of `shutdown` and `awaitTermination` (imitating paragraph 3 of Figure 1). It hides the timeout feature of `awaitTermination` as well as the try-catch block for the `InterruptedException`. Both of these features can be useful in production code, but beginning students will have

relatively short-running tasks and not interrupt their threads. Thus, our wrapper gives them all the functionality they use without requiring them to deal with other possibilities.

Our next wrapper class, `SimpleFuture`, simplifies the use of futures. The main gain here is in hiding exceptions. The `get` method of `Future<V>` can throw either an `InterruptedException` if the waiting thread is interrupted or `ExecutionException` if the task on which it is waiting throws an exception. As discussed above, we assume beginning students will not interrupt threads so the first exception should not occur. The second exception is only possible if the task has uncaught exceptions. In the case of a beginner, we can further simplify the code by assuming they will not use exceptions to communicate between a task and the waiting thread. Thus, our wrapper class catches both kinds of exceptions internally (exiting the program if they occur), removing the need for a try-catch block. Again, we have greatly simplified the beginner's experience at the cost of a small amount of unnecessary (to them) generality.

The final wrapper class, `SimpleTask`, is for creating tasks, thus replacing `Runnable`. A type of task is created by extending this class and overriding its `run` method. The task then provides a method (`finish`) which blocks until it is complete. Blocking until task completion with Java's built-in `Runnable` objects requires using a `Future<?>` (future with unknown value type) object. Our `SimpleTask` class hides this syntax.

A lesser benefit of all our wrapper classes is that they hide the heavy use of inheritance present in `java.util.concurrent`. None of the wrapper classes inherit from any other class so they can be presented to students as a self-contained API.

The wrappers provide a clean interface that allows students to focus on the key concepts of task-based parallelism that transcend any particular implementation. In an introductory setting, this is likely the instructor's only goal in presenting this material, in which case there may be no need to discuss the hidden details and the wrappers might be all that students see. For more advanced courses or if the instructor wants to go deeper into the topic, the wrapper classes can be a starting point, helping students master the key concepts before "looking under the covers" at the Java implementation later in the course.

HEAT SIMULATOR EXAMPLE

Now we describe our application to be parallelized. It animates a simulation of heat diffusion. It works correctly as given to students, but runs fairly slowly. The reward for parallelizing its key part is a noticeably faster animation.

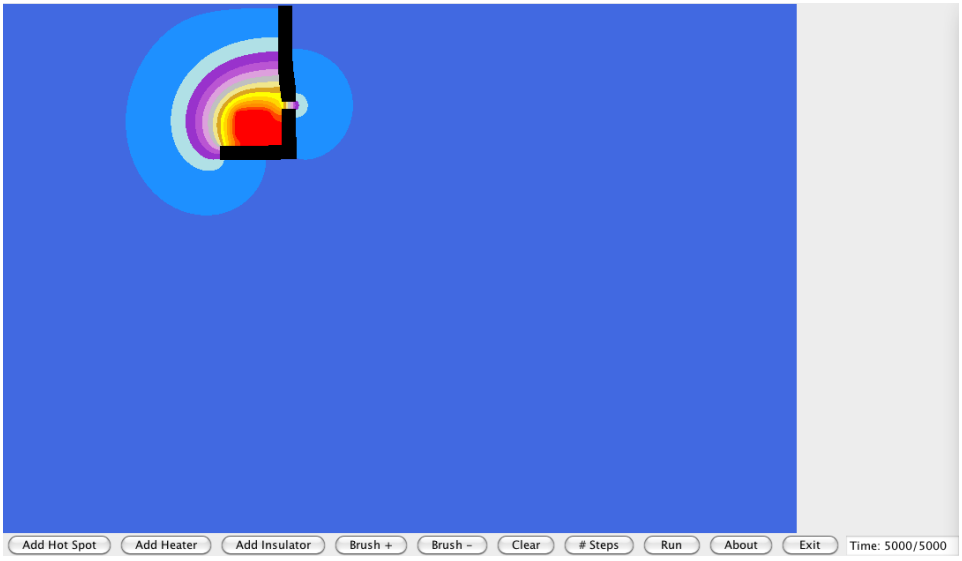
Heat diffusion is the movement of heat through some medium such as air. The simulation divides a region of interest into a 2D grid of cells. The simulation's state is the temperature of each cell at a particular time. If the value is low, the cell is cold and appears bluish. Higher values indicate warmer cells, which appear as a range of colors up to red for the warmest cells. The temperatures are updated in time steps based on the temperatures of neighboring cells and the user sees this effect as the colors change.

Most cells are initially cold, but three features add variety. The first are hot spots, locations that start warm. The second are heat sources, which start and remain warm. The last are insulators, which have no heat of their own, but reflect heat from their neighbors.

For each time step, a cell's new temperature is 0.25 times its old temperature and 0.75 times the average of its neighbors' temperature. (A cell's neighbors are the 8 cells sharing an edge or corner with it.) Looked at another way, a cell keeps 25% of its heat and

passes 75% to its neighbors. Cells off the board are assumed to have temperature 0, causing heat loss at the edges.

When started, the program creates a main window with a display showing the cells and a toolbar. The overall interface is like a painting program that lets the user draw hot spots, heaters, and insulators. There is a button that runs the simulation for a specific number of steps (initially 5,000). The following is a screenshot of this program:



Each step of the simulation requires updating every cell in the display. Its default size is 900 by 600, for a total of 540,000 cells. At this size, the simulation runs slowly, but the update operation is easily parallelized since each cell can be updated independently. (Updates are to a new copy of the array to remove dependencies between neighbors.)

Parallelizing the application

To demonstrate the suitability of this example and the usefulness of our wrapper classes, we compare them with some other approaches to parallelize the simulator. We timed the programs on a MacBook Pro with a dual-core 2.53 GHz Intel Core i5 processor running Mac OS X 10.6.8 (Snow Leopard). The serial simulator took slightly under 2 minutes for 5,000 time steps on this system. Implementations using Java's executor framework and our wrapper class gave speedups slightly below 1.8. (The wrapper version actually performed slightly better than the other, but we attribute this to chance.) Since the program synchronizes every time step and the measured times include GUI updates, we were happy with this speedup. Importantly, the `SimpleExecutor` version achieved it with simpler code, mainly due to the lack of try-catch blocks.

To highlight the advantages of the executor framework, we also compared these solutions with some thread-based versions. A solution that created a thread for each task gave a speedup around 1.3. This has equivalent code complexity to the wrapper-based version, but substantially worse performance because it creates so many threads.

To reuse threads outside the executor framework, we also implemented a version using `java.util.concurrent.CyclicBarrier` to create a barrier at which threads wait after each time step. This version achieved a speedup around 1.6, recapturing only

some of the lost performance. It also requires understanding barriers and creating a *barrier action*, run when the last thread reaches the barrier and before the threads are released. (This action switches the displayed board and repaints the GUI.) During execution, the program switches between the threads updating temperature and this barrier action. In addition, the program's main thread must join with the update threads at the end of the program. Either way, the other threads must interact in a more sophisticated way than just performing tasks delegated to them by the main thread. Although objective measurement of this extra complexity is difficult, the student developing this code found it much harder to write and understand.

DISCUSSION

We have presented the executor framework with our wrapper classes and the heat diffusion simulation as a motivating application. To use these with students, we plan to simply combine them; present the concepts of tasks and the wrapper classes, followed by having the students parallelize the heat simulator. Depending on the level of the course, a next step could be to reveal and explain the details hidden by our wrappers.

Despite the details hidden by the wrapper classes, the wrappers still allow the demonstration of parallel speedup, load balance (an uneven split increases running time), and overhead (creating too many tasks slows the program down). While the heat diffusion application avoids race conditions, these can be demonstrated with another example.

Looking ahead, we plan to validate this approach in a classroom setting (tests so far have been with summer students). We are also interested in using our wrapper classes with other examples and possibly porting the simulator into other programming languages.

REFERENCES

[1] S.A. Bogaerts. Limited time and experience: Parallelism in CS1. In *Proc. 4th NSF/TCPP workshop on parallel and distributed computing education (EduPar)*, 2014.

[2] K.B. Bruce, A. Danyluk, and T. Murtagh. Introducing concurrency in CS 1. In *Proc. 41st SIGCSE Technical Symp. Computer Science Education (SIGCSE)*, pages 224--228, 2010.

[3] D.P. Bunde, K.L. Karavanic, J. Mache, and C.T. Mitchell. Adding GPU computing to computer organization courses. In *Proc. 3rd NSF/TCPP workshop on parallel and distributed computing education (EduPar)*, 2013.

[4] P. Garrity, T. Yates, R. Brown, and E. Shoop. WebMapReduce: An accessible and adaptable tool for teaching map-reduce computing. In *Proc. 42nd SIGCSE Technical Symp. Computer Science Education (SIGCSE)*, pages 183-188, 2011.

[5] D. Grossman. Ready-for-use: 3 weeks of parallelism and concurrency in a required second-year data-structures course. In *Proc. Workshop on Curricula for Concurrency and Parallelism*, 2010.

[6] S. Massung and C. Heeren. Visualizing parallelism in CS 2. In *Proc. 3rd NSF/TCPP workshop on parallel and distributed computing education (EduPar)*, 2013.