THE UNIVERSITY OF AUCKLAND

NEW ZEALAND

Te Whare Wānanga o Tāmaki Makaurau

Parallellise Kiwiplan Framework

BTech Project Report

Author: Prabhjot Singh Jassal
UPI: pjas002

Contents

1 Introduction **4**

2 Multicores Vs Manycores **7**

3 About the Company **8**

4 What I Hope To Gain **8**

5 Kiwiplan Framework **9**

6 Project Goal **12**

7 Concept of Parallel Programming in C# **13**

8 Concept of Parallel Programming in C++ **13**

9 Concept of Parallel Programming in JAVA **15**
9.1 Manual Thread Creation 16
9.2 Thread Safety 17
9.3 Thread management using Executor Framework 18

10 Fork/Join Framework **19**
10.1 Fork-Join First Example 20
10.2 How to call ForkJoin program 22
10.3 Fork-Join Second Example 22

11 Parallel Array **27**
11.1 Trivial Parallel Array Examples 27
11.2 Advanced Parallel Array Examples 30

12 Kiwiplan Framework Revisited **33**
12.1 Row Visibility Filter 33
12.2 Sorting Filter 37

13 Conclusion And Future Work **40**

14 References **41**

List of Figures

1 Evolution of computer . . . 5
2 Table Example . . . 9
3 Pipeline Example . . . 9
4 MVC pattern use in Kiwiplan framework. . . . 10
5 Person table in the database . . . 11
6 Initial mapping of the Person table . . . 11
7 Mapping after sorting the Person table based on last name . . . 11
8 Result of the sorting operation . . . 11
9 Backward mapping after the sorting operation . . . 12
10 ParallelFor in C sharp . . . 13
11 Fibonacci Serial in C++ . . . 14
12 Fibonacci Parallel in C++ . . . 14
13 Task Manager Usage . . . 15
14 Issue with threads . . . 17
15 Basic Fork/Join Program structure. . . . 19
16 Demonstration of how to use Fork/Join framework . . . 20
17 Starting task 2 before task 1 . . . 21
18 Insertion Sort sequential . . . 23
19 Merge Sort sequential . . . 24
20 Merge Sort parallel . . . 26
21 Prime number sequential . . . 28
22 for-loop to check each number . . . 29
23 Prime Number Predicate . . . 29
24 How to Call isPrime Predicate . . . 30
25 Using Max reduce function . . . 30
26 Program that returns the next prime number of the given number . . . 31
27 Ops Operation for Next prime number . . . 31
28 Calling NextPrime Ops operation . . . 32
29 Parallel Array Student Example . . . 32
30 UML Diagram of TableModelFilter hierarchy . . . 33
31 Row Visibility Sequential Algorithm . . . 34
32 Row Visibility Example . . . 34
33 Row Visibility Operation Result . . . 34
34 Row Visibility Parallel Algorithm . . . 35
35 Performance comparison of Row Visibility Sequential and Parallel Algorithm . . . 36
36 Row Visibility Parallel Algorithm - Slow version . . . 36
37 Runtime of Row Visibility Slow version . . . 37
38 Point 2 Example . . . 37
39 Demonstrating why sorting algorithm should be stable . . . 38
40 Shuttle sort sequential version . . . 38
41 Shuttle sort parallel version . . . 39
42 Rest of the Shuttle Sort Parallel code . . . 40
43 Performance comparison of Shuttle sort Sequential and Parallel Algorithm . . . 40

Abstract

Developers had the privilege of making their program runs faster with no effort 10 or 20 years back. We discuss the foundation of parallel programming and try to answer why these is a requirement of doing parallel program. In 2005, HerbSutter published on the most famous paper in parallel programming field, titled : "Free lunch is over". Developers need to change with coding style to make use of multi cores of the computer. However, writing programs which makes use of the multi-cores is not an easy task to do. Fortunately, Java 7 new fork/join framework makes it easy for the developers to write parallel programs. This report discusses some of the features introduced in fork/join and ParallelArray framework. We will discuss how the Kiwiplan framework is being structured and then we discuss how could we apply these concepts in parallelizing some of the filter algorithms and present the results obtained.

1 Introduction

Computers have made significant changes in the recent world. They mainly got popular in the previous 10 to 20 years. Before that, they were mainly used by highly technical people as it required good amount of knowledge to operate (also it was expensive to buy). When computers were first built, they only used to have a *single* program [9]. That program runs sequentially from the beginning to the end. This was before the advent of operating system. During that time, user did not have the privilege to run multiple programs together. They could only start another program once they terminate the first one. This means they could only work on a single program at any given time. If we compare that time with today's world, we might realize how inconvenient it was.

Luckily, during early 1950s the concept of operating system came and the computers started building with an operating system. Operating system is nothing but the capability of storing multiple programs and data but moreover they provide common service to run various application software efficiently. With the advent of operating system the whole idea of computer manufacturing changed. The users got the privilege of using multiple programs together. They could now switch back and forth among the programs. This saves their time and they found their work much more convenient to do.

However, even when the computers started building on the operating system their clock speed was too low. The *Clock speed* is the number of instructions executed per unit time. Initially, low clock speed was not the big problem but as time goes the user requirements kept on getting high as well. To satisfy the demands of users, the computer manufacturers starts making computer with higher clock speed. They only had to add more transistors to the chip to make it faster. This means, if the given program takes n amount of time to run in early 1980s, the *same* program would take roughly half of the time to run in mid 1980s. This was because of the fact the clock speed kept on increasing. The programmer does *not* have to change their coding style and the exactly same program takes less amount of time to run. That was the time from 1990s to early 2000 when the computer manufacturers kept on increasing the clock speed by adding more and more transistor to the chip.

However, when the clock speed got to its peak, the manufacturers realized that computer would overheat if they try to increase the clock speed beyond its peek. One approach the manufacturers could take was to stop manufacturing the computers with higher clock speed at all. This approach solves that problem as computers were not getting overheated if the clock speed remains within certain limit. However, this only solves the problem temporarily. As the computers evolve, the user requirements kept on getting high as well. The manufacturers have to meet the user requirements and must be able to find new ways to increase the execution speed of the computer.

This was the time when people started coming up with the idea of multiple cores. Before that computers used to have only a single core that was responsible for performing all the tasks. People thought instead of giving all the work to the same core and wait for them to finish it, it would be better to divide the tasks among multiple chips. These multiple chips tasks can be run in parallel. Once each chip finishes

it work, the result for each one of them can be simply joined together, if necessary. Since the tasks are running in parallel, this approach lowers the execution time. From the user point of view, this was important. As long as users kept getting faster computer, they don't really care how it is being made faster. The idea of dividing the task among chips and getting the result from them was based on *Divide and Conquer* principle.

In 2005, Herb Sutter published the paper titled: "The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software" [10] . This is one of the most famous paper in the field of parallel programming. Figure 1, shows the graph presented in his paper. On the x-axis we have the time and y-axis could be any of the options given. It could be number of Transistors, clock speed etc. From the figure, it is easy to see that from 1970 to early 2000, there was an exponential growth in the clock speed. So, the developers need not had to change their programming style. Faster computers were coming on the regular basics. More and more transistors were kept on adding into the single core. At around 2003 onwards, the clock speed halted. So, manufacturers started making computers with multiple cores.

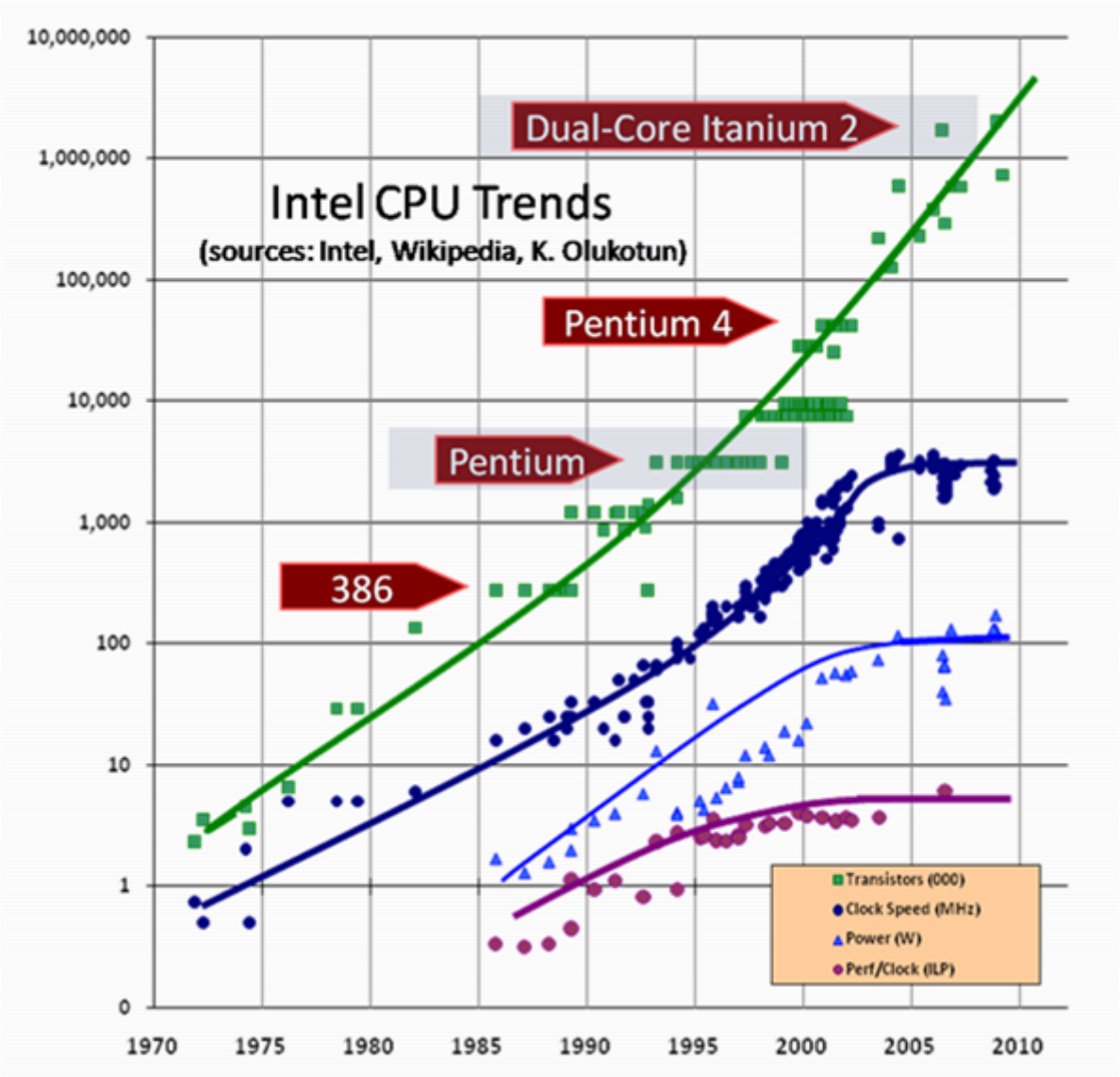

Figure 1: Evolution of computer

However, to use the multi-cores of the computer simultaneously, the developers had to change their style of coding. Because if we have a dual core system and our program is single-threaded then we are giving up with half of the resources and if the computer has 100-cores then 99% of the cores are not getting used. It means if we run our single-threaded program, we won't get any performance benefit at all regardless of whether the computer we run on has 1 core or 1000 cores. Manufacturers have predicted that there will be an exponential growth in the number of chips for another 20 years or so. So,

this makes it even important for the developer to learn the skill of writing parallel programs. We will discuss some of those techniques in this report. But for now let's come back to the Divide-And-Conquer strategy on which the manufacturing of multi-cores computer relies on.

There exists lot of algorithms in Computer Science based on Divide and Conquer strategy even before the concept of multi-core computers. This idea was simple and based on a very logical strategy. Even in normal life we use the same logic. For example, if we have to clean our whole house, rather than allocating the cleaning work to a single person it would be better if each home member cleans their own room. They can also agree on some *protocol* for the places which are being shared. By this way, we get the desired output (cleaned house) in less amount of time. The same is exactly true from the computational point of view. Rather than spending all the computer resources in performing a single task, it is better to *distribute* the given task among multiple resources (cores). The cores may also needs to co-ordinate with each other time-to-time to get the desired output.

Coming back to our cleaning example; the time when one person was cleaning his/her room, the other person is cleaning its own. In other words, the task of cleaning the home is being done in parallel rather than being done in some sequential order. We could easily think that if we increasing the number of people to clean the house from 1 to 2, the total time taken to clean the house would be *exactly* half when only one person was used. However, this is not quite correct as some amount of time is needed for them to agree on the protocol. Once they agreed only then the work starts. However, if only one person is being given the responsibility to clean the whole house, then there would not be any need to agree on some protocol. The person could start cleaning the house in some order straightway.

The same is true for programming as well. Before we run the tasks in parallel we have to *distribute* the task among the resources. This distribution of tasks among resources takes time and the programmer must make sure that the total running time of the program; allocating work among resources, its completion and merging the results together, must take less time than the total running time if ran sequentially, otherwise there won't be any benefit for all the effort developers put in. We would not get any performance benefit at all.

The reader might also be thinking that since increasing the number of people allocated to do the job decreases the total time, so why not allocate lot of people. This approach works reasonably well but only till certain point. Once you increase the number of *workers* beyond the certain number, the performance suffers.

Let us try to understand this concept from our cleaning example. If there are only 3 rooms in the whole house and the number of people allocated to finish the task (cleaning the house) is 100, then most of the time people will be sitting ideal. But you might be thinking this is not a big problem, but let's assume instead of giving the task of cleaning the house to home members, you *hire* people and pay them agreed amount of money once the job is done. In this case, you will end up paying to all the 100 workers even though most of them were sitting ideal.

This is also true from programming point of view. If the number of workers to the job is much more than we required *optimally*, then the whole performance of our program suffers. First, we have to allocate memory to each of the worker, then we have to distribute work to each one of them and if they are sitting ideal then it is not good. This concept of choosing the value for the numbers of workers so that the task is being done optimally is very important in the field of parallel programming.

Above we briefly talked about what points the programmers need to make sure to write optimal parallel programming. But we did not mention one of the most important point. The programmer *now* needs to change their programming style to write parallel programs. Initially we described that in 1980s the programmer does not need to change their code at all and then same program would run faster if the same program is being run later say in mid 1980s. However, since the clock speed of the computer is not

increasing anymore and manufacturers had to rethink their architecture, the programmers also need to change their code if they want it to run in parallel.

One important point that needs to be mentioned before we jump on to the next section. The time when manufacturers started making multiple cores computers, they were quite expensive. Due to this, it was only used in those tasks which requires enormous amount of computation. At that time, most of the programmers did not have to learn parallel programming concepts and need not to change with coding style. However with time, computers with 2 and 4 cores have become cheap and is really common now. So, many IT companies wants to see what benefit they can get in their products if they are being run in parallel rather than sequentially. Now, programmers can not survive long enough without learning parallel programming concepts and sooner or later have to dive in this field. And according to the prediction of the computer manufacturers, we will see exponential amount of growth in number of cores in the computer in coming years.

As mentioned in the previous paragraph, the multi-core computers are very cheap now and IT companies have been looking to make use of multi-core computers to increase the productivity of their products. However, writing parallel program is very hard as compared to writing the sequential ones. The process of allocating work to different cores and getting the result back from them is not as easy as it sounds. Only programmer with good understanding of computer memory (low level details) could do that. However, understanding the requirements of the market and to make development of parallel programs easy, many popular programming languages designed some libraries which takes care of low level details. In this case, even the programmers with a reasonable amount of knowledge about *memory allocation* could able to write parallel program with less amount of time.

Before moving towards my project, we will discuss another side of parallel programming world - Multicores Vs Manycores. This has been discussed in the next section. In the later sections, we will discuss the project scope, library support offered by some important programming languages to help developers in developing parallel programs easily and quickly. Separately we will discuss the concept of parallel programming in JAVA, how the kiwiplan framework is being structured, how could we make use of Java fork/join framework in Kiwiplan framework and finally the results obtained during the project will be discussed.

2 Multicores Vs Manycores

In the introductory section, we simply mentioned that computer manufacturers started building computers with more than one core and it is possible to get computers with 16 cores as well. However, this is just one side of the picture. On the other side of the picture, we have the term called *many-cores*. In many-cores system, parallel programming is being done using graphics card (or Graphics Processing Unit).

The term *many-cores* are being used for them because graphics card could have many more chips as compared to normal computer. The graphics card with 200-300 chips are very normal. There exists many APIs dedicated to make use of graphics card. One of them is CUDA platform. Initially, it was mainly used for graphics purpose but now it has started being used for other purposes as well.

Since this report is only about multi-core programming, we will not dive more in many-cores computing. However, developers must be aware that the principals on which multi-cores programming relies on are very similar to the ones on which many-cores programming relies on. So, having a mastery of these basic concepts not only help developer in writing effective programs using multi-cores but also programs using many-cores as well. Syntax changes but the logic remains the same.

3 About the Company

This project is being collaborated with the company named Kiwiplan. The details of the company is given below.

Kiwiplan NZ Limited.

Level 3, BDO House
116 Harris Road
East Tamaki
PO Box 58-456, Botany, East Tamaki
Auckland, New Zealand
Phone: (09) 272 7622
Fax: (09) 272 7621
Web: www.kiwiplan.com

About Kiwiplan

Kiwiplan is a software development company that services the corrugating and packaging industries. Typical customers include firms that produce corrugated cardboard products such as boxes, display stands, and other packaging products.

Kiwiplan started business in 1970s as a small corrugating firm. As their throughput increased they developed computer systems to help them keep up with demand. There was considerable interest from other packaging companies in these computerised systems, and the IT department grew and eventually separated from the box plant division.

With over three decades of expertise in developing innovative software for the corrugated and packaging industry, Kiwiplan delivers the total solution for all of your software and Manufacturing Execution System needs. Kiwiplan is now one of the worlds leading software suppliers to the packaging industry. They have customers in around 28 countries. All research and development work is being done from the New Zealand offices at East Tamaki, Auckland.

Kiwiplans product range covers the entire business process for a packaging firm, from order entry to shipping. The core products relate to controlling the plant and scheduling the corrugating machinery.

4 What I Hope To Gain

1. **Real World Exposure** - The most important skill I would achieve would be the exposure to real world programming. I have mainly worked on reasonably small assignments during my BTech degree. I already worked with Kiwiplan last summer so that gave me bit of confidence to work in company environment.

2. **Analysis** - The best part from programming for me is to solve those question which requires too much thinking. During the project, I worked on company's code which took me an enormous amount of effort to understand (Kiwiplan Framework will be discussed in the next section).

3. **Writing Parallel Programs** - Before working on this project, I had no understanding of how to write programs that make use of multi-cores. According to Brain Goetz, the important member of the team which made concurrency library in JAVA; writing parallel programs are much harder as compared to writing the sequential programs, because the developer not only needs to be aware of the points in which the sequential program goes wrong but also other issues come inevitably in parallel

programming.

5 Kiwiplan Framework

Before I will discuss the project goal, it is best to familiar readers with the Kiwiplan framework. This will aid in their understanding about the project. Kiwiplan started building their framework in 2003 from the BTech project itself undertook by Timothy Paul Walker. The framework was being developed to allow users to interrogate information from the graphical user interface. The users were given the capability to visualize the records stored in the database and is flexible enough to meet user requirements. For example, the user could sort the data by clicking on top of the column header, the user may sometimes want to see only those rows which fulfills certain criteria and many more including grouping the *records* based on certain column. Above all, the user does not have to know anything about SQL to do these operations.

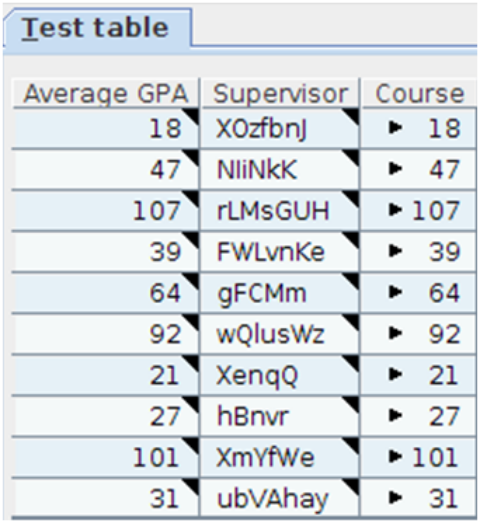

Figure 2: Table Example

An example of how the 2D table looks like on the Kiwiplan interface is shown in the figure 2. By clicking on top of the column header the items gets sorted in ascending order. Clicking again on the same column header would sort the items in the descending order. The order of the columns could be changed by simply dragging them across.

Whenever the user makes some action on the interface, the action goes through the pipeline. The pipeline is nothing but a series of filters, where each filter in responsible for performing only one specific task. Some of the common filters are sorting filter, row visibility filter, grouping filter etc. Figure 3 [11] shows the series of filters in the pipeline.

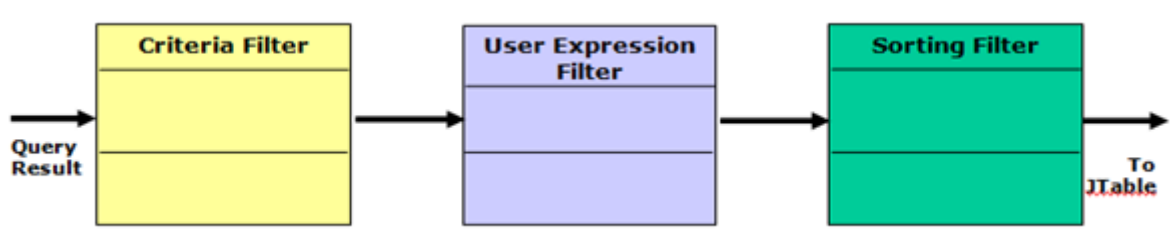

Figure 3: Pipeline Example

The framework was kept on developing over the year and many more features have been added. However, the basic idea of framework remains the same. Many design patterns are being used in the development of the framework. One of such design pattern is Model-View-Controller (MVC) pattern. This pattern separates the logic in three different parts, where *view* keeps track of how to display the information, *controller* keeps track of the mouse and keyboard input from the user (event handling), and *model* manager the data and behavior of the application domain. This pattern is particularly useful when building large size applications [12].

One of the biggest advantage of using this pattern is that, it allows us to represent data in various forms independent of how the data is being stored. The *view* component takes care of this and also allow the user to act on that data. On the other hand, most of the processing is being done on the *model* component. This component stores all the business rules (or logic) necessary to perform the required computation. This component also helps in reducing code duplication because the data returned by the model is neutral to how it will be displayed on GUI. This allows us to reuse the same model for different views.

Lastly, the *controller* component sits in the middle of other two components. No real processing is being done on this component. *Controller* interprets mouse clock or keyboard input from the user and calls necessary methods of *view* and *model* component to get the desired result. For example, if the user clicks on some button on the interface, then instead of doing the actual processing, it simply decides which part of the *model* component needs to the called and how the resulted data will be formatted. The figure 4 [11] shows the use of MVC pattern in Kiwiplan framework.

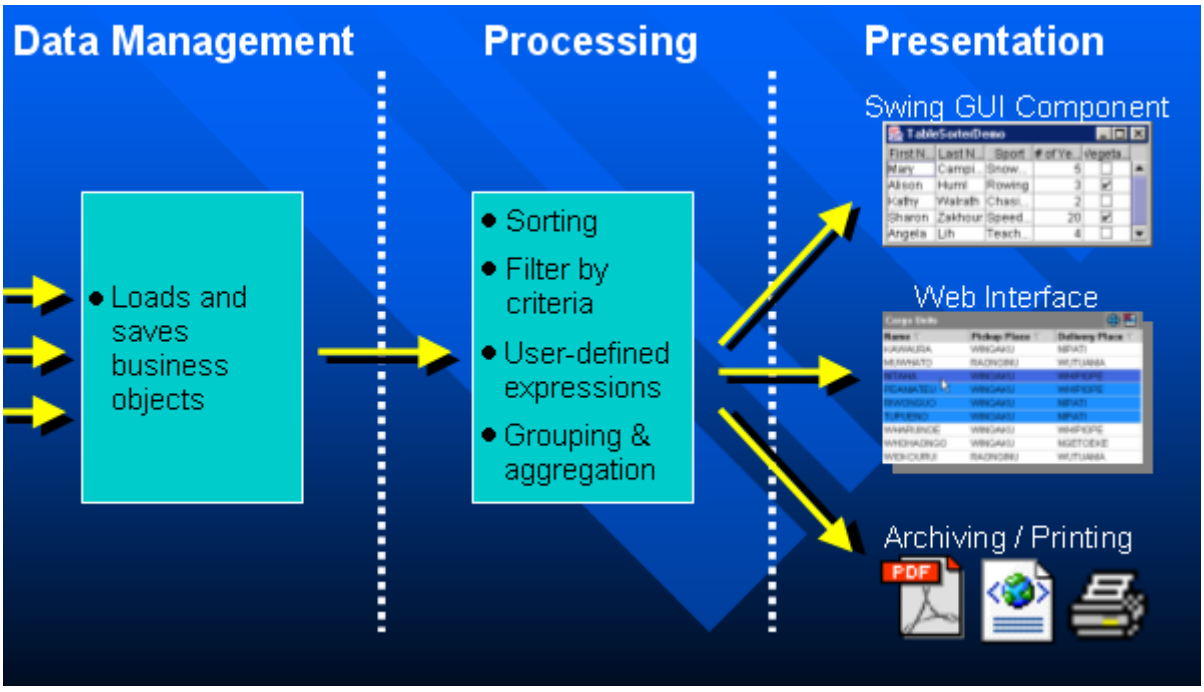

Figure 4: MVC pattern use in Kiwiplan framework.

Another pattern being used in the framework is the *Singleton* pattern. This pattern ensures that a class has only one instance and provides a global access to it. This pattern is useful when most of the time is being used in creating a different instances of the same class. Object creation takes time and slows down the performance of the whole application. Instead of creating a separate object every time, it is better to use *mutator* methods to set the values of the variable. This saves time by not creating unwanted objects and hence improves the performance of the whole application.

It is important to keep in mind that since MVC pattern is being used, each algorithm of the filter only has the pointers to the actual data but not the data itself. This can be best understood by an

example. Let's say we only have four rows in the *Person* table in the database. The table is shown in figure 5.

Last Name	First Name	Age	City
Paykel	Bob	28	Auckland
William	Kyle	23	Wellington
McCullum	Nathan	27	Auckland
Martin	Robin	22	Napier

Figure 5: Person table in the database

And our task is to sort the table based on the *last name*. Our sorting algorithm will be given a mapping to the data rather than table itself. The initial mapping is given in figure 6.

1 --->1
2 --->2
3 --->3
4 --->4

Figure 6: Initial mapping of the Person table

We will say that *key* 1 maps to *value* 1, key 2 maps to 2 and so on. Key basically represent what we are viewing on the interface and value means which row in the database table it maps to. So, 1 maps to 1 means the row one in the interface is the row one in the database table. The table itself will never change but mapping could. Once, we do the sorting based on last name our mapping changes to :-

1 --->4
2 --->3
3 --->1
4 --->2

Figure 7: Mapping after sorting the Person table based on last name

This means the row 1 we are viewing on the interface is in-fact the row 4 of the database table, row 2 of the interface is actually the row 3 of the person table and so on. This process saves a lot of memory as to do any operation all we have to pass to an algorithm is the mapping. The algorithm simply works on the mapping and the result of the user action can be fully determined from the output returned by an algorithm[1]. Before we move on to say another thing, please keep in mind that though the user will see the figure 8 on the interface after performing the sorting operation but there will *not* be any changes in the Person table.

Last Name	First Name	Age	City
Martin	Robin	22	Napier
McCullum	Nathan	27	Auckland
Paykel	Bob	28	Auckland
William	Kyle	23	Wellington

Figure 8: Result of the sorting operation

Another important point to notice is we not only keep track of the mapping from keys to values but also from values to keys. So, we have the bidirectional mapping. The backward mapping of the figure **??** is shown below :-

[1]There are few issues which needs to be addressed about the sorting algorithm. But since that is not really a framework topic, it will be discussed later on.

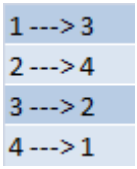

Figure 9: Backward mapping after the sorting operation

The backward mapping is being needed to make the event handling more efficient. Forward mapping keeps the mapping from view to the table and backward mapping keeps the mapping from table to the view. This means if there will be changes in the row 1 of the person table, we can see from the backward mapping that it is actually the 3^{rd} row in the interface which will get affected. This makes the processing fast for the event calls generated by the user.

As mentioned earlier, that when the user makes some operation on the interface the request being passed to the pipeline. However, rather than processing the request as a "top down" flow, the result is being processed as a bottom up flow. This mean when the user makes some operation, the request is being passed to the last filter of the pipeline, which then passes the request to the previous filter and so on. The request keep on flowing up the pipeline and when the method returns, it flows down. The reason why the framework is being designed like that because the Swing components and many other existing user interface components work similarly and this approach works really well. It is always better to reuse the existing patterns which are already been thoroughly tested rather than re-inventing the wheel from the beginning.

The pipeline idea is flexible as it is possible to add and remove filters from the pipeline very easily. The filters need not to know what filters are being added at its neighbors. This makes it easy for the developers to code and each filter can be tested independently of other filters. This section provided the overview of the framework and the reader should got some feel of how it works.

6 Project Goal

As discussed in the previous section that the request made by the user goes through the series of pipeline and pipeline consists of many filters where each filter is responsible for performing only one specific task. Each of the algorithm that solves the problem is being written in a manner which only makes use of a single core of the computer. Now with the multi-core computers affordable to buy and with the concurrency support coming with the *next* release of Java 7[2], Kiwiplan wants to see how the efficiency of the algorithms can be increased. So, the goal of the project is :-

1. To investigate the new framework introduced in Java 7 for writing parallel programs.

2. Explain how these new library features can be used in the Kiwiplan framework.

Of course this is not an easy project because before trying to make any algorithm parallel, I had to understand how everything joins together and that includes how the framework works, understand how the pipeline is being constructed. Most importantly, I have not done any parallel programming at all in any language so this project was quite challenging for me, particularly the code i have to parallelize is the real world company code, not the small assignment from the university. According to Brain Goetz, the company's best developers need to be given such tasks and they have good amount of knowledge of how the whole framework works. Also, this project was not easy because the sequential programs were already written. It requires more effort to understand the sequential code written by someone else and then trying to parallelize it rather then if the developers himself write down the sequential algorithms and then try to make it run in parallel. However, these are the challenges that needs to be faced when

[2]Java because the whole framework is being developed in it. All the algorithms of the filters are being written in Java itself.

working in real world projects.

As said, the framework is being written in JAVA itself. But initially we will discuss the support offered by *other* important programming languages, namely C# and C++. We are doing this for two reasons. Firstly, it will enable us to compare the running time of the *same* algorithm in different languages. Secondly (and more importantly), writing parallel programs is not an easy task and we want to compare how much effort the developers need to put in if the same algorithm needs to be parallelized in different languages.

7 Concept of Parallel Programming in C#

We will not go into details. We straightway show one simple example :-

```
int[] myArrary = { 5, 2, 6, 1 };
int sum = 0;

for (int i = 0; i < myArrary.Length; i++)
{
    sum += myArrary[i];
}

int sum2 = 0;
Parallel.For(0, myArrary.Length, delegate(int i)
{
    sum2 += myArrary[i];
});
```

Figure 10: ParallelFor in C sharp

In this example we are computing the sum of our array. As we can see, if the developer needs compute the sum in parallel all they have to do is use Parallel.For. These changes are very small and makes it easy for the developers. Parallel.For automatically distributes the task to threads.

Of course there is more to it rather than simply using Parallel.For all the time. C# also introduced PLINQ (parallel version of LINQ[3]) which makes it very easy for the developers.

8 Concept of Parallel Programming in C++

With visual studio 2010, C++ came up with the parallel pattern library. We can declare an instance of type *structuredTaskGroup* which does the work of distributing the task to multiple workers. We will show the Fibonacci example and do the same in Java. In this way we can compare the running of Fibonacci of C++ with Java. Fibonacci series is very common where the n^{th} term of series is the sum of the previous two terms of the series.

The sequential algorithm is shown in figure 11 :-

[3]LINQ is based on functional programming approach. It is very powerful and enable us to do some of the tasks in very few lines.

```
int serialFibonacci(int n){
    if (n < 2)  return n;
    return (serialFibonacci(n - 1) + serialFibonacci(n - 2));
}
```

Figure 11: Fibonacci Serial in C++

The parallel algorithm is shown in figure 12 :-

```
int parallelFibonacci(int n){
    if(n < 2)   return n;
    int n1 = 0, n2 = 0;

    structured_task_group tg;
    auto task1 = make_task([&](){
        n1 = parallelFibonacci(n - 1);
    });

    tg.run(task1);
    auto task2 = make_task([&](){
        n2 = parallelFibonacci(n - 2);
    });

    tg.run(task2);
    tg.wait();
    return n1+ n2;
}
```

Figure 12: Fibonacci Parallel in C++

Let's try to understand what is happening step by step. If n is less than 2 simply return n, this is our base case. Otherwise, *create* subtask of size n-1. This is happening with the help of structured task group object. As soon as we create the first subtask we start it. Then we create second subtask of size n-2 and run it as well. In this way, both subtasks are being running in parallel. Then we wait for *both* of them to get finished and done we simply sum their results.

We will see that the fork/join framework is based on the similar approach as well. This approach is known as divide-and-conquer, where recursively we solve the problem which subtasks and in the end merge the result together.

Figure 13 shows the task manager screen shot. As soon as we run the parallel Fibonacci 100% CPU is getting used.

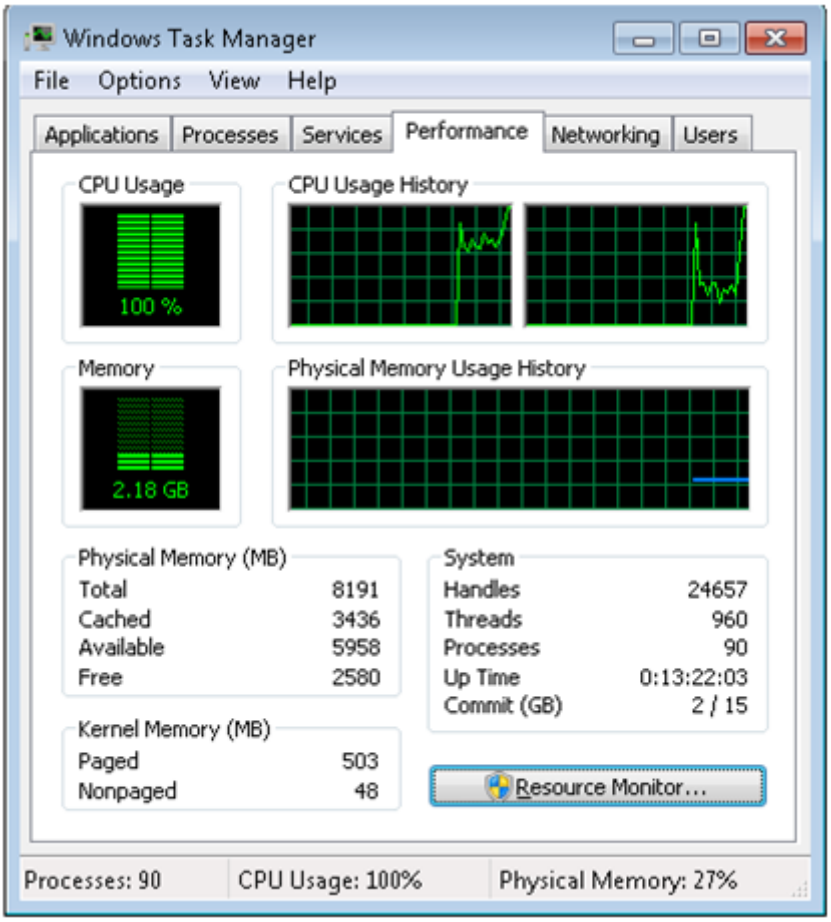

Figure 13: Task Manager Usage

Let's discuss JAVA concurrency features now.

9 Concept of Parallel Programming in JAVA

The basic building block of parallel programming in JAVA are threads. Threads are the part of JAVA since its very first release. Before we go more into the details of threads, we should be clear regarding the distinction between *Process* and *Thread* [1] .

Process - Each process runs independently of other processes. They do not share any data and completely isolated of each other. It is the responsibility of the operating system to allocate resources to the process.

Threads - On the other hand, threads are lightweight processes. Each thread has its own stack for keeping track of the variable values. However, unlike the process they do have an access to the shared data.

If we want our code to run in parallel, we can achieve this by running multiple threads simultaneously. We wait for them to finish their work and in the end, their results can be joined together, if needed. The important point for the developer to keep in mind that even though the parallel program has multiple threads, but we will start with only a single thread. This thread is called the *main thread* and has an ability to create, delete other threads [1]. In the coming material the term *asynchronous* task will be used a lot. Asynchronous tasks are those tasks that can be run independently of each other. The most common example is server-client application. Each request received from the client can be considered as an asynchronous task.

Even though the project focuses mainly on Java 7 concurrency features, it is always better to learn the basics first. This will enable the developer to optimize their code more if they are acquainted with

these low level details. Let's see how the threads can be created in Java. There exist *three* ways to create threads in Java.

1. Create an instance of in-built class Thread. This will enable the developer to *directly* control thread creation and how they need to be managed.

2. Use an *executor* framework. An executor framework came with Java 5.0 as a higher level abstraction of thread. The developer need not create or delete (including sleep) the threads manually.

3. Use the fork-join framework coming[4] as a new concurrency feature in Java 7.

We will briefly mention about the first two possibilities and elaborate more on the last one.

9.1 Manual Thread Creation

There are couple of ways to create thread manually. There is one thing common in both the possibilities - we have to provide the implementation of the run() method. The run() method will be called when we start the thread. This method should contain the code we want to run in parallel e.g. handling client request.

One way to do this is - let our application class extends the in-built class Thread. Following is one random example :-

```
public class FirstExample extends Thread {
    public void run() {
        println("Hello World");
    }
}
```

The thread can be started by making this call (new FirstExample()).start(). Please note that even though we are calling start() method, it will actually call the run() method we defined.

We can achieve the same behavior if we implement the Runnable interface. This can be done like this :-

```
public class SecondExample implements Runnable {
    public void run() {
        println("Hello World");
    }
}
```

Though both the approaches behave similarly but 2^{nd} approach is more flexible because in the first one, we have to extend the Thread class. In case if our application is already extending some other class then that approach would fail.

Before we see how the threads can be created using Executor framework, we will mention some of the important methods we can use when working with threads. One of such method is join() method. This should be used if we want one thread to wait for the completion of the other thread for an unlimited time. If we don't want to wait for an unlimited period, we could use join(x) that only waits for x milliseconds.

There exists other important methods such as sleep(x), which causes the currently executing thread to sleep for x milliseconds [2].

[4]Java 7 has not been officially released yet. It was supposed to be released in early 2010 however, it has been delayed and now expected to be released before the end of this year.

9.2 Thread Safety

Threads make it easier for us to write concurrent programs. However, the developers need to make sure that their program is *thread safe*. But what exactly does it mean for our program to be thread safe. Let's try to understand this. When our program is single-threaded we say that it is correct when it *always* produces the correct output for any input case [3]. We could easily test the single-threaded program by specifically checking the boundary cases and it is easy to reason about the sequential program because it works as how we humans think.

However, when our program becomes multi-threaded we need to be weary about how we reason about its correctness. Multi-threaded program is thread safe when it *always* produces the correct output for any input, regardless of how the threads intervene with each other [3]. Let's consider one simple example [4] :-

```
public class UnsafeSequence {
    private int value = 0;
    public int getNext() {
        return ++value;
    }
}
```

This class is responsible for generating the unique value every time. This works perfectly fine in a single-threaded program. However, not in multi-threaded environment. The following problem can happen if thread interleave each other at wrong time.

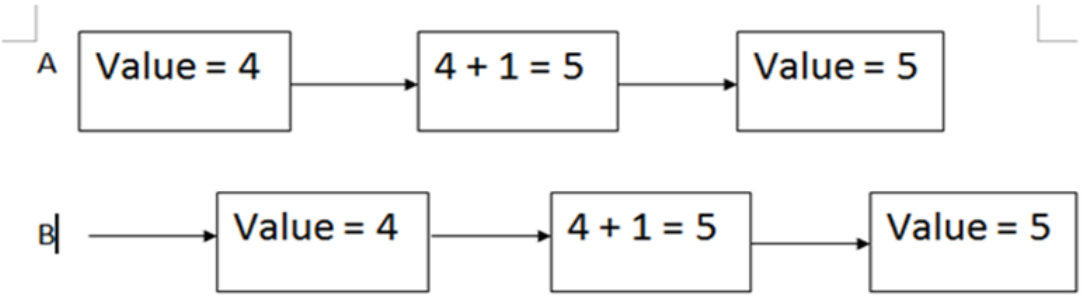

Figure 14: Issue with threads

Both the threads A and B see the value equal to 4 and both modify it 5 and return it. The problem happened because both the threads have an access to the *shared* field. The developers have to make sure that if the threads have an access to the shared data and one of the thread have a privilege to write to it, then they must not interact with each other at that step.

If the step is atomic then there should not be any problem. However, the increment step in our example is not an atomic step. There are actually 3 steps combined together in one. It reads the value from the variable, increment it and then save it back to the variable. To ensure the safety, we should make sure if one of the thread is carrying out these steps, the other thread must not intervene. This can be solved by 3 possible ways. One of the way is :-

```
public class SafeSequence {
    private int value = 0;
    public synchronized int getNext() {
        return ++value;
    }
}
```

The "synchronized" keyword makes sure only one thread can come into getNext() method at any given time. Thread acquires the *lock* at the method heading. If other thread tries to intervene, it finds the lock is acquired by some other thread. It waits for the other thread to leave the lock. Once the first thread leaves the thread, it will be acquired by the next one in the queue. The lock is on *this* instance. This solves the problem, but not a good way do it[5] if we only want a part of the method to hold the lock. The better way to solve this problem is :-

```
public class SafeSequence {
    private int value = 0;
    public int getNext() {
        synchronized (this.value) {
            return ++value;
        }
    }
}
```

Since we only want to protect the *value* field incremental part, we just need to put a lock on that step only. The third way to solve this problem is :-

```
public class SafeSequence {
    private AtomicLong value = new AtomicLong(0);
    public int getNext() {
        return value.incrementAndGet();
    }
}
```

The AtomicLong object defined for concurrent purpose and available in java.util.concurrency package. Other classes like AtomicDouble etc are also available. The methods written in these classes are atomic in nature and developer can use them with confidence.

The bottom line is - whenever we have a situation where multiple threads share the *mutable* field and one thread have the privilege of writing the value to it, then all the compound steps should be protected by the synchronized block. This will ensure that from Thread A point of view, either the whole of compound statements are executed or none of them and thus ensures the thread safety.

9.3 Thread management using Executor Framework

In the previous section, we noted how we could create Thread and manage it. This works fine for small-size application. For large size application, we want to separate the logic for thread creation and its management from rest of the application [5]. This problem can be solved by using *Executor* interface came with Java 5.0. There are actually three interfaces we could use.

1. Executor Interface - This interface provides a method called execute(runnable object). It basically replaces this line we used earlier - (new Thread(runnable object)).start() [5]. Using executing method allows us to create and run the thread at a same time. We don't have to create threads separately and should avoid it as thread creation is expensive [6]. With an Executor interface, we can create the thread pool of some size and later use execute() call to run a particular thread. In this way, we don't have to create new thread every time. This concept is known as Thread Pool. In the example, we are creating 4 threads in our pool :-

```
Executor executor = Executors.newFixedThreadPool(4);
```

And later call the threads we want to execute. Another benefit of using Executor framework is that we

[5]from performance point of view.

could allow the result of thread computation. Earlier we were using run() method which can not return any result as it is of type void and secondly run() method does not throw an exception. One benefit of using the Thread Pool is - if one of the thread suddenly stops working or dies, then other thread in our pool automatically replaces it. Threads defined in the pool are generally called *worker threads* [7].

2. ExecutorService interface - This is similar to Executor interface, but it replaces the execute() method with a submit() method. submit() method accepts Runnable objects, as well as Callable objects [5]. We will discuss more of this when we discuss Fork/Join framework since it is based on ExecutorService interface.

3. ScheduledExecutorService interface - Almost similar to ExecutorService interface, but it allows us to execute Runnable or Callable task after a specified delay [5].

10 Fork/Join Framework

The first point to note is Fork/Join framework is not a replacement for an Executor framework. They both are designed to complement each other, not to complete with each other. Execute framework is being designed if the division of the task is coarse-grained. For example, handling the client request to separate threads. On the other hand, Fork/Join framework is being designed for finely-grained tasks. For example, we not only want to handle multiple client request in parallel but also some parts of the handling request in parallel as well.

Fork/Join framework is being designed specifically for those problem that are based on divide-and-conquer strategy; basically recursive programs. The fork/join framework is based on the ExecutorService interface. The basic difference is that fork/join framework using work-stealing algorithm [8]. This will be explained later on. The basic structure of the program we solve in parallel would be [13] :-

```
Result solve(Problem problem) {
    if (problem is small)
        directly solve problem
    else {
        split problem into independent parts
        fork new subtasks to solve each part
        join all subtasks
        compose result from subresults
    }
}
```

Figure 15: Basic Fork/Join Program structure.

So, given a problem, check is the problem size small enough that can be done sequentially. If yes, then solve it sequentially, otherwise break the problem in independent parts, usually in 2 parts. Solve those recursive tasks in parallel and in the end merge the result together. Developer needs to know that it is the *merge* step that needs extra effort to get it right and most of the time the performance depends on how the merge step is being written. If we synchronized the whole *merge* step, only one thread can process it at any single time. This means the merge step is being done sequentially and due to synchronization the merge step of parallel algorithm might take more time as compared to the sequential one. This extra time is spent in acquiring the lock and releasing the lock and also keep the worker threads in the queue which have been blocked.

So, always try to minimize the synchronization block in the merge step as much as possible. Sometimes we could eliminate the synchronization but sometimes we have to put it. So, if our code needs

synchronization and we do not provide then it is broken, even if it passes the test cases[6]. The main point to note is the developers should try to make their code correct firstly and only then they should try to optimize it.

10.1 Fork-Join First Example

We have to extend Fork-Join task class in order to use Fork/Join framework. Either we could directly extend the Fork-Join task or it could be done indirectly by extending the inbuilt classes that extends Fork-Join task underneath. Some of the subclasses we could extend are - RecursiveAction, RecursiveTask and AsyncAction etc. [14]

RecursiveAction should be used when our program does not need to return any result i.e. result-less computation. The program that should display only those rows that satisfy certain criteria[7] is a good example of this scenario. On the other hand, RecursiveTask should be used when our program needs to return some result i.e. result-bearing calculation. AsyncAction generally gets useful when we are working with graphs. So, common graph programs such as graph traversals should extend AsyncAction class [8].

Of course, the developer could directly extends ForkJoinTask in case the problem does not fit into any of the above category. But most of the time RecursiveAction and RecursiveTask should be enough to work with. The program shown in figure 12 demonstrate how to use Fork-Join framework. This program computes the Fibonacci number in parallel. In Fibonacci series, the first two elements are 1 and the value at position n is the sum of value at position n - 1 and n - 2. So, these two recursive calls can be done in parallel. Note that the iterative version of Fibonacci is faster than this, because even though the two recursive calls would be done in parallel but too much time would be spent on computing the same value again and again. In the iterative version, we could save the pre-computed values in the table. This avoid the computation of a same number again and hence saves lot of time. This technique is called memoization and forms the basic of Dynamic Programming. We will not go into the details of DP more, but here is the program which uses fork/join framework to compute the Fibonacci number in parallel [14].

```
class Fibonacci extends RecursiveTask<Integer> {
  final int n;
  Fibonacci(int n) { this.n = n; }
  Integer compute() {
    if (n <= 1)
       return n;
    Fibonacci f1 = new Fibonacci(n - 1);
    f1.fork();
    Fibonacci f2 = new Fibonacci(n - 2);
    return f2.compute() + f1.join();
  }
}
```

Figure 16: Demonstration of how to use Fork/Join framework

Since we want this method to return some value, we made a new class called Fibonacci which extends RecursiveTask. To be more precise, we can explicitly put the return type generically with RecursiveTask,

[6]As failing the test cases depends on how threads interleave with each other and developer has no control over that.
[7]RowVisibility Filter
[8]Most of the information is being obtained from Java 7 concurrency API

which came with Java 5. However, it is not compulsory to specify the type of the variable we want to return. RecursiveTask itself would work as good.

Whether we implement RecursiveTask, RecursiveAction, AsyncAction or even ForkJoinTask we have to implement a method compute(). This method is an abstract method in all the subclasses of ForkJoinTask and itself. So, any class extending them have to provide the implementation of this abstract method. In the compute() method we want to put the main logic of our computation. Keep in mind that if we extend any ForkJoinTask and implement compute() method with our logic, it won't do the computation in parallel. It depends on how the compute() method is being implemented.

In the example above, our base case is *n is less than or equal to 1*, usually this is not a good sequential threshold. We will discuss more about it later on as it has big say on the performance of our program. Anyway, if the n is less than or equal to 1, we simply return n. Otherwise, we *create* our first task. Important to keep in mind that we creating the task, but not starting it at this step.

To start the task we use the predefined method (and one of the most important method in Fork/Join Framework) called fork(). To start the first task, we used f1.fork() method. Then we created the second task and similar to the first one, we only created the second task at this step. The last line of the program is very important. In the last line we are using f2.compute(), which starts the second task. It automatically allocates the work to different core. That means, the first task and second task are being done in parallel. And once we finish with the second task, that is when f2.compute() returns some value, we *join* the result of the first task. If the first task is still running, then join() method blocks until the first task is completed. Once it will be completed the join() method returns the value.

It is important to keep in mind that the order in which the last instruction is written is very important. If we change this instruction to "f1.join() + f2.compute()", then we don't achieve any parallelism at all. Because what we are telling computer to do is - get the result from the first task and once that is returned, only *then* start doing the computation of second task. In other words, task 1 and task 2 are being done sequentially. We are only starting the second task once the first one is finished. We would not be able to achieve any parallelism at all. The bottom line is we want to start the computation of the second task before we wait to get the result for the first task. This enable us to run first and second task in parallel and hence improves the performance (may not be in this example because of re-computation of the same value).

```
class Fibonacci extends RecursiveTask<Integer> {
    final int n;
    Fibonacci(int n) { this.n = n; }

    @Override
    public Integer compute() {
        if (n <= 1)
            return n;
        Fibonacci f1 = new Fibonacci(n - 1);
        Fibonacci f2 = new Fibonacci(n - 2);
        f2.fork();
        return f1.compute() + f2.join();
    }
}
```

Figure 17: Starting task 2 before task 1

One more point to discuss before moving onto the next section. The program shown in figure 17 is also correct. In this program, we are starting task 2 *before* task 1. There is no reason why this is

not valid. This is perfectly valid because our goal is to run the two tasks in parallel. It matters little whether we start the first task the before second task or after. f2.fork() starts the computation of second task. Then we call f1.compute(), which starts the computation of first task and once that is finished we join the result of our task 2 computation.

10.2 How to call ForkJoin program

We discussed about how to write the parallel program but we did not discuss one important point - how to call the class from our program. Fortunately, this is a very trivial. Moreover the style of calling the parallel program remains the same even though the code is doing entirely different thing. To call our Fibonacci code, we first have to create the ForkJoinPool instance. This is one of the predefined class comes with Java 7. It is quite similar to thread pool but optimized for fork/join framework. It's constructor method does not have to take any parameter. But the sensible option is to use Runtime.availableProcessors() as the constructor parameter. So, we will do something like this :-

ForkJoinPool forkJoinPool = new ForkJoinPool(Runtime.availableProcessors());

Once the instance of ForkJoinPool has been created, all we have to do it call the invoke() method with the instance of the class that contains our parallel program code. In our example, the class name is Fibonacci. So, to invoke our program we could use :-

forkJoinPool.invoke(new Fibonacci(30));

if we want to compute the Fibonacci number at index 30. So, the task of calling our program is divided into two steps. In the first step, we create the instance of ForkJoinPool and then we use invoke() method with our class constructor as a parameter.

10.3 Fork-Join Second Example

The example we discussed in the previous section gave us some feel of how should we structure our program and what points we need to keep in mind to do that. Let's discuss one of the most fundamental problem that can be done in parallel - Sorting. There exists many sorting algorithm that can be run in parallel. One of the less obvious sorting algorithm we can run in parallel is *insertion sort*. Insertion sort has a running time of $O(n^2)$. There is one approach called odd-even technique which can be used to make it run in parallel.

However, in this section we will discuss one of the famous sorting algorithm - MergeSort. One of the reason we want to discuss this ahead of InsertionSort or QuickSort because it is an efficient algorithm (it has a running time of $O(\text{nlogn})$), it is stable[9] and it is almost similar to the ShuttleSort algorithm used in the Kiwiplan framework.

The merge sort algorithm work as follows - The method takes three parameters, first and second keeps track of range of the array they want to sort and the third parameter is the array itself. This algorithm is based on divide and conquer approach which makes it easy to fit to fork/join framework. If the range we are working is 1 then we simply return because the number itself is always sorted. Otherwise, get the midpoint of the start and end index and makes two recursive calls on them. This is the divide part of the algorithm.

When we get to the merge step, we already have the two half sorted. Before merging we create a separate copy of our array. We simply merge the two sorted array by comparing the elements one by one. Also, rather than discussing MergeSort alone, we will discuss its hybrid with the insertion sort. Rather than making the base case of our recursion of 1 element we choose 15 and then do insertion sort

[9]Stable algorithm are those which does not change the ordering of the equal keys.

to sort those 15 elements in the base case. This hybrid of merge sort with insertion sort works faster than the merge sort itself. The algorithm is shown in next two figures.

```
private static void insertionSort(int start, int end, int[] array) {
    for (int i = start+1; i <= end; i++) {
        int j = i;
        while (j > start && array[j-1] > array[j]) {
            int temp = array[j-1];
            array[j-1] = array[j];
            array[j] = temp;
            j--;
        }
    }
}
```

Figure 18: Insertion Sort sequential

This part shows the insertion sort algorithm. The way it works is as follows - given a list of unsorted number we start from the second position. Let this position be *j*. As long as the element *j* is greater than the start position and if element at position *j-1* is greater than the element at positionj, then we swap those two values around. It means at every point of our algorithm, all the elements of the array index *j-1* to the start position will always be sorted. This is the usual way how we sort the deck of cards by hand.

Next figure shows the merge sort algorithm which we will run in parallel shortly.

```java
private static void mergeSort(int start, int end, int[] array) {
    if (end - start <= 15) {
        insertionSort(start, end, array);
        return;
    }
    int mid = (end + start) / 2;
    mergeSort(start, mid, array);
    mergeSort(mid+1, end, array);

    //Create a copy of the array
    int[] tempArray = new int[array.length];
    for (int i = start; i <= end; i++) {
        tempArray[i] = array[i];
    }

    //Merge Step
    int leftPointer = start, rightPointer = mid+1, position = leftPointer;
    while (leftPointer <= mid && rightPointer <= end) {
        if (tempArray[leftPointer] >= tempArray[rightPointer]) {
            array[position++] = tempArray[rightPointer];
            rightPointer++;
        } else {
            array[position++] = tempArray[leftPointer];
            leftPointer++;
        }
    }

    while (leftPointer <= mid) {
        array[position++] = tempArray[leftPointer];
        leftPointer++;
    }

    while (rightPointer <= end) {
        array[position++] = tempArray[rightPointer];
        rightPointer++;
    }
}
```

Figure 19: Merge Sort sequential

It is very easy to follow the code and see how it works. Without wasting time, let's discuss how can we make the merge sort run in parallel. The two halves of the array can be sorted in parallel. There is no reason why it has to be sorted sequentially. Also, while we merge the two halves other parts of the array can be sorted in parallel as well.

With merge sort we don't have to provide any synchronization at the merge step, if we run that in parallel. The reason is threads are working on the separate part of the array and most importantly, the tempArray variable is the not shared among threads because it is a local variable. Every thread has it's own copy of the local variable. Since the memory is *not* shared, we don't have to provide any kind of locking or synchronization. So, always remember that if threads are not sharing any variable we don't have to provide any sort of synchronization because no matter how they interleave with each other, due to separate copy of the local variables nothing could go wrong.

To make our merge sort run in parallel and use Fork/Join framework we follow the following procedure.

Step 1. Create a new class that extends RecursiveAction (because mergeSort() method does not

return anything)

Step 2. Create a constructor of the class we created in step 1. There will be only one constructor method we need to write and that constructor method parameters will be the parameters of our mergeSort() method in the previous page. That is, the first parameter would be *start*, second parameter would be *end* and the third parameter would be the array itself we are interesting in sorting.

Step 3. Override the compute() method as we are extending RecursiveAction class. Copy the whole code from our mergeSort() method given on the previous page to the compute() method. We need to make very few changes to make it run in parallel.

Step 4. Instead of making two recursive call we would create two tasks. Basically two instances of the class we created in step 1. One of the tasks will be created with parameters of - start, mid, array and other with - mid+1, end, array.

Step 5. We can use invokeAll() method; also comes with Java 7; and pass two instances we created in step 4 to it. It will make the two tasks run in parallel.

Note that the order in which we pass on the tasks as a parameter to the invokeAll() method is not important in this example as oppose to the Fibonacci example we provided earlier. This is because the method does not return anything.

By making all the changes listed in step 1 to 5 we will be able to run the merge sort in parallel. However, the performance we would get is not optimal yet. To optimize our program we need to make some more changes. Earlier we used 15 as the base case of our recursion. However, using the same base case the performance of our program hurts. It takes time to create threads which can run the tasks in parallel. Thread also sleeps internally and unlike this example we may need to synchronize some of the steps in our code.

To make this correct, we should replace this number 15 with some big value - we call this value *sequential threshold*. Sequential threshold is good if it is 1% of our array size. However, even if we make it 5%, the performance of our program does not hurt much. We should avoid two extreme cases - choosing too low or choosing too high value for sequential threshold. Luckily, there is a very sweet spot in-between and we do not have to be too conservation with this number.

Also, rather than saying that if the problem size is less than some sequential threshold calls the insertion sort method we replace it with sequential mergeSort() method call. So, given an array of size ten thousand we recursively divide the work in two halves *and* run them in parallel. Once the size go below one thousand we still recursively divide our tasks but we do their computation sequentially now and once the problem size reduces to less than 15 we make use of insertion sort algorithm. So, we have two threshold - one is the insertion sort threshold and other one is sequential threshold. Once that is done we get the optimal performance from our program.

The code obtained by making all the changes is shown in the figure next.

```
private static int SEQUENTIAL_THRESHOLD = 1000;
static class SortAction extends RecursiveAction {
    int start, end;
    int[] array;
    public SortAction(int start, int end, int[] array) {
        this.start = start; this.end = end; this.array = array;
    }
    @Override
    protected void compute() {
        if (end - start <= SEQUENTIAL_THRESHOLD) {
            mergeSort(start, end, array);
            return;
        }
        int mid = (end + start) / 2;
        invokeAll(new SortAction(start, mid, array), new SortAction(mid+1, end, array));

        //Create a copy of the array
        int[] tempArray = new int[array.length];
        for (int i = start; i <= end; i++) {
            tempArray[i] = array[i];
        }

        //Merge Step
        int leftPointer = start, rightPointer = mid+1, position = leftPointer;
        while (leftPointer <= mid && rightPointer <= end) {
            if (tempArray[leftPointer] >= tempArray[rightPointer]) {
                array[position++] = tempArray[rightPointer];
                rightPointer++;
            } else {
                array[position++] = tempArray[leftPointer];
                leftPointer++;
            }
        }

        while (leftPointer <= mid) {
            array[position++] = tempArray[leftPointer];
            leftPointer++;
        }

        while (rightPointer <= end) {
            array[position++] = tempArray[rightPointer];
            rightPointer++;
        }
    }
}
```

Figure 20: Merge Sort parallel

11 Parallel Array

In the previous section, we discussed the fork/join framework. It is quite easy to parallelize the program as long as developer keeps some important points in mind. However, the amount of effort needed to convert the code the parallel is much more if have to do that in C#. Unfortunately, we can not use C# with the Kiwiplan framework as the code is too big and converting the whole to C# would take ages. It would have been far more convenient for the developer if the more abstract level had been added in Java. That would make it even easier for the developers to write parallel programs. Luckily, the support has given from Java.

We could use functional programming approach in Java itself with the help of ParallelArray data structure. It provides many predefined methods that does some of the fundamental operation in parallel such as sorting, searching, aggregation etc. Important to note is ParallelArray part would not be included in Java 7 as Dough Lea, author of the concurrency library of Java, has not decided with the syntax completely. However, Parallel Array and closures would be the part of Java 8 as they will be more refined by then. However, all the jar files of the Parallel Array classes are given in extra166y and anyone can download them from Dough Lea homepage.

Let's discuss how to create an instance of ParallelArray object. Assume that we have the array or vector (any collection object) which has all the elements we are interested in. We could create a wrapper of forkJoin pool around this. This is shown below :-

ParallelArray<Integer> myArray = ParallelArray.createFromCopy(array, forkJoinPool), where array is the collection that holds our elements and forkJoinPool is an instance of ForkJoinPool.

Once that is done we could apply many of the method comes with ParallelArray framework. For example, in the example above if we are interested in sorting the elements - all we have do is myArray.sort(). It automatically chooses the sequential threshold and do the sorting in parallel and even better the results obtained by using parallel array are more optimal as compared to the algorithm written directly with fork/join framework. This is due to the fact that the hardware has been optimized for the pre-defined code. ParallelArray[10] works on fork/join framework underneath. It automates the fork/join framework and makes it easy for the developer by providing the higher abstraction level.

11.1 Trivial Parallel Array Examples

Let's say we are interested in finding the maximum element of the array. Array could be of any primitive type as well as of object type. However, with the object type we have to pass on the comparator method that would tell computer on what bases the maximum would be selected. For example, if we are dealing with object of type Student, where each instance has three attributes - name, graduation year and GPA. If we are interested in finding the student with the maximum gpa we have to write the comparator method. We will discuss this example in the next section. For now, we stick with the simple example; finding the maximum element in the array of integer. This can be done by the following code :-

```
ParallelArray¡Integer¿ parallelArrary = ParallelArray.createFromCopy(array, forkJoinPool);
int max = parallelArray.max();
```

It is as simple as that and maximum element in the array would be computed in parallel. Also, note down we declared the parallelArray which holds down element of type Integer. However, when we did max() operation we saved the result in int. This technique is called Autoboxing. This came with Java 5.0.

Now we are interested in sorting the array. Just to be versatile we use ParallelLongArray object this

[10] Also ParallelLongArray and others included in the ParallelArray framework.

time, but similar approach can be This code does it for us :-

ParallelLongArray parallelLongArray = ParallelLongArray.createUsingHandoff(array, forkJoinPool); where array is the array of long elements.

Note we used createUsingHandoff() method instead of createFromCopy(). With createFromCopy() the copy of the given array is created first and then that copy is being used to create an instance of ParallelLongArray. However, creating a copy is not a good idea if the array is too big. In that case, we could use createUsingHandoff() method. Though we are dealing with ParallelLongArray object in this example, the same steps are equally with ParallelArray as well. To sort the element all we have to do is this :-

parallelLongArray = parallelLongArray.sort();

It automates the fork/join machinery underneath and sorts the array. However, note the instance of ParallelLongArray; parallelLongArray itself would not change if we do only parallelLongArray.sort(). The sort() method returns an instance of ParallelLongArray type and in this example we saved the result back to the original variable. We may also want the sorted values in the new instance, in case we want to keep the initial ordering of the elements intact. This is perfectly valid and can be easily done by saving the returned result from the sort() operation in some other instance.

Let's discuss one more, a bit less trivial example, before moving on to the next section. Say we have to find all the *prime numbers* within some specified range. Prime numbers are those numbers that divided by 1 and itself. For example, 11 is a prime number because only numbers that divides 11 evenly are 1 and 11 itself and 9 is not a prime number as it is divisible by 3 as well. The sequential program to do this task is shown in figure 17.

```
private static boolean isPrime(int n) {
    if (n < 2) return false;
    int sqrt = (int)Math.sqrt(n);

    for (int i = 2; i <= sqrt; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}
```

Figure 21: Prime number sequential

We could make a loop go through till *n-1*, but it is a famous fact that if there exists a perfect divisor for the number n, it has to be less than or equal to the square root of n. Given the list of numbers we have to decide which ones are prime. We can check each element in the for-loop one by one like this :-

```
for (int i = 0; i < array.length; i++) {
    if (isPrime(array[i])) {
        System.out.println(array[i]);
    }
}
```

Figure 22: for-loop to check each number

However, instead of checking each element of the array that whether it is prime or not one by one, we could do them in parallel. There is no reason why it can not be done in parallel as checking the primality test of one element is independent of other elements. This could easily be done in parallel without doing many modifications.

We need to introduce some new concepts before we show the code that does the task in parallel. Let's analyze the above example bit more thoroughly. Say we are given n amount of number, we *only* wants to select those numbers from them which are prime. That means we want to filter out some elements from our original collection. In other words, all the elements which are not prime should be excluded from our collection i.e it is a predicate. The predicate is the one which returns either true or false.

With ParallelArray framework we can define a predicate that return true if the number is prime otherwise return false. Once we define the predicate, we have to find some way to use this in our code. Luckily, ParallelArray framework also provides an easy way to do that as well. There is a method named, withFilter() to filter out elements in the collection and we can pass on the predicate we wrote earlier to this method. Once that is done we are *almost* done. Here is how the predicate looks like :-

```
static final Ops.Predicate<Integer> isPrime = new Ops.Predicate<Integer>() {

    @Override
    public boolean op(Integer n) {
        if (n < 2) return false;
        int sqrt = (int)Math.sqrt(n);
        for (int i = 2; i <= sqrt; i++) {
            if (n % i == 0) {
                return false;
            }
        }
        return true;
    }
};
```

Figure 23: Prime Number Predicate

The syntax is bit annoying in theory though not much in practice. The Ops keyword stands for operation and Ops.Predication is just one example. There exists many more option and we will discuss some in the next section. Whenever we are making any Ops method, we have to override the abstract method op(). This will be the method which contains the logic on the basis of which we want to filter out some elements. Since it is a predicate the return type would be boolean. We just have to copy the code of the method *isPrime* and paste it inside the op() method. Once that is done our predicate is ready to use.

Now we have to how could we call this predicate and as said before, we have to use the withFilter() predefined method comes with Java 7. This is shown in the figure next :-

```
ParallelArray<Integer> myArray = ParallelArray.createFromCopy(array, forkJoinPool);
myArray = myArray.withFilter(isPrime).all();
```

Figure 24: How to Call isPrime Predicate

The method all() needs some attention. The withFilter() method itself does not return an instance of ParallelArray but of type ParallelArrayWithFilter. But in our example what we want is filter out all the elements which are not prime and return *all* of them. That is why we used all() method. This is what we call a reduce function because this is the one which actually return the result. Let's say that instead of finding all the prime numbers we are interested in finding the maximum of them. We could have done something like this :-

```
ParallelArray<Integer> myArray = ParallelArray.createFromCopy(array, forkJoinPool);
int maxPrime = myArray.withFilter(isPrime).max();
```

Figure 25: Using Max reduce function

11.2 Advanced Parallel Array Examples

In the previous section we saw how we could use Parallel Array to do very fundamental tasks in parallel. The prime number example gave us one more insight. It showed that we could use two methods in one step. The first method was withFilter() and then in the same step we used all() method. This is how the functional programming is based on and we could use the same approach here.

Rather than using only two methods in one step we could use any number of them as long as we follow some rules. These rules would tell us how to build up a query. For example - in the prime if we would have used max() method before the withFilter() then our code does not compile. These rules how can we build up our query but keeping it consistent. The rules may be hard to follow in theory but comes naturally during practice. We will discuss these rules bit later. But before we need to introduce some other methods which can be useful to build up a query.

Apart from filtering the elements, ParallelArray framework also provides the facility of mapping elements. There is a method named withMapping() which we can use. With the help of this method, we could map the elements of our array to some other value. We will demonstrate this with an example. Since we already familiar with the concept of prime number, we extend our previous example. Now rather than saying that we are interested in finding whether the given number is prime or not, we are interested in task of mapping number A to number B. If number A is a prime number then we would not change it's mapping, otherwise we find the next prime number after A and maps it to that element. Hence, number B would always be a prime number.

We would reuse our isPrime() method we wrote earlier to decide whether the number is prime or not. If the number is prime we simply return that. Otherwise, we make an infinite loop of finding the next prime[11] and break out of the loop as soon as we find prime number. We do not need to check the even numbers as all the even numbers except 2 are non-prime. Here is the sequential version of the program that accomplishes this task :-

[11]Note that this is not an efficient way of doing this. One of the best optimal way to do this is to use Sieve of Eratosthenes

```
private static int nextPrime(int n) {
    if (isPrime(n)) {
        return n;
    }
    if (n % 2 == 0) {
        n++;
    }
    while (true) {
        if (isPrime(n)) {
            break;
        }
        n += 2;
    }
    return n;
}
```

Figure 26: Program that returns the next prime number of the given number

It is easy to follow the code and see how it works. By using the same approach we used earlier to create Predicate in the previous section, we will make this run in parallel as well. This is how our Ops operation looks like :-

```
static final Ops.ObjectToLong<Integer> nextPrime = new Ops.ObjectToLong<Integer>() {

    @Override
    public long op(Integer n) {
        if (isPrime(n)) {
            return n;
        }
        if (n % 2 == 0) {
            n++;
        }
        while (true) {
            if (isPrime(n)) {
                break;
            }
            n += 2;
        }
        return n;
    }
};
```

Figure 27: Ops Operation for Next prime number

Unfortunately, there is method called Ops.IntToInt() or Ops.ObjectToInt() so we have to use ObjectToLong. But this is not a big hassle as *long* is capable of storing much bigger number as compared to integer. Any time we write the Ops operation we have to override op() method. This method parameter depends on what the input type is (Integer in this example) and the return type is what the Ops operation is expected to return (long in this example and boolean in case we are dealing with Predicates).

To call this Ops operation we can do something like this :-

```
ParallelArray<Integer> myArray = ParallelArray.createFromCopy(array, forkJoinPool);
ParallelLongArray longArray = myArray.withMapping(nextPrime).all();
```

Figure 28: Calling NextPrime Ops operation

Let's discuss one more example. This should make it how to use ParallelArray effectively. Example is taken from [14]. Say we have an array of Student object. Each student has the graduation year record, his gpa and his name. Our task is to select the maximum GPA of the student who is graduating in year 2009.

The basic approach to do this question is filter out all the students who are not 2009 (this can be achieved by withFilter() method) and we have to map each student with his GPA (this can be achieved by withMapping() method) and then we select the maximum out of the gpa. Figure 29 shows how to do this.

```
ParallelArray<Student> students = new ParallelArray<Student>(fjPool, data);
double bestGpa = students.withFilter(isSenior)
                         .withMapping(selectGpa)
                         .max();

public class Student {
    String name;
    int graduationYear;
    double gpa;
}

static final Ops.Predicate<Student> isSenior = new Ops.Predicate<Student>() {
    public boolean op(Student s) {
        return s.graduationYear == Student.THIS_YEAR;
    }
};

static final Ops.ObjectToDouble<Student> selectGpa = new Ops.ObjectToDouble<Student>() {
    public double op(Student student) {
        return student.gpa;
    }
};
```

Figure 29: Parallel Array Student Example

We used withFilter() then withMapping() and then max(). It looks imperative that we doing the filtering first and once the filtering is done only then we are doing the mapping and so on. However, the execution does not start until we get to the last step. All we are doing is building up our query and once our query finishes only then the execution starts in parallel.

The withFilter() method uses Ops.Predicate which we have already mentioned when we discussed Prime number example. WithMapping() is using Ops.ObjectToDouble because we are mapping Object (Student in this case) to double (his gpa). In both the cases we need to write down the logic in the op() method.

In the previous section we discussed how the Fork/Join framework works and how the ParallelArray framework works. ParallelArray framework is quite powerful and makes the developer life really easy. However, ParallelArray functionality has one limitation due to which we are unable to use in Kiwiplan framework. To work with parallel array we have to have the actual data with us. However, in Kiwiplan framework we are having the mapping only as discussed in Kiwiplan Framework section. Due to this reason, I did not use ParallelArray in Kiwiplan framework. However, we can apply Fork/Join framework with Kiwiplan framework. Let's discuss how this can be done.

12 Kiwiplan Framework Revisited

We already discussed the basics of Kiwiplan Framework. In this section we will show how the algorithm in SortingFilter and RowVisibility Filter can be run in parallel. Figure 30 shows the hierarchy of the TableModelFilter [11].

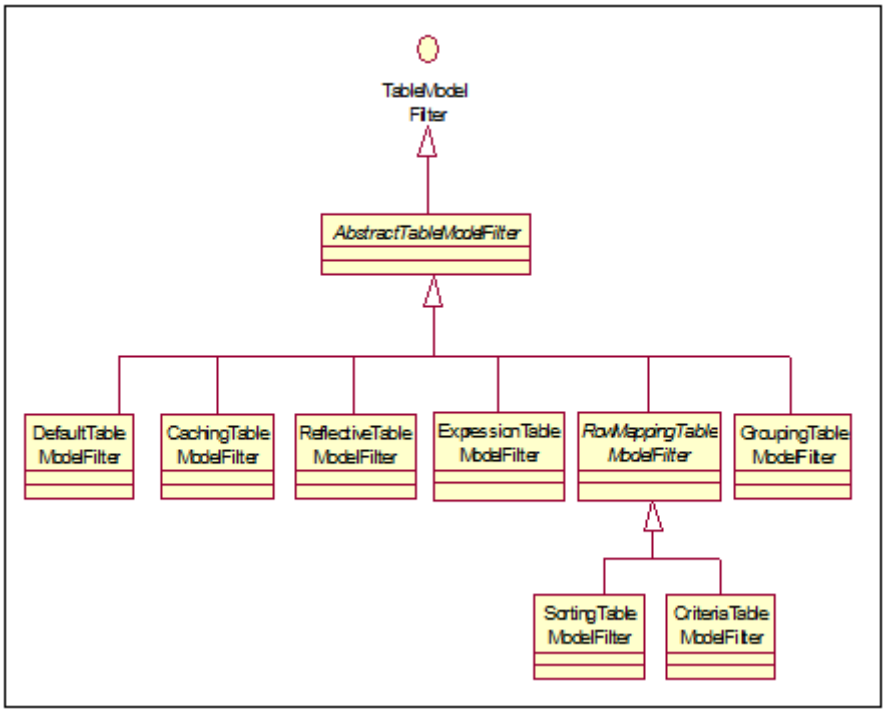

Figure 30: UML Diagram of TableModelFilter hierarchy

This is only the part of the framework. There are many more filters. TabelModel filter sits at the top of the framework AbstractTableModelFilter contains all the methods which will be common to all the methods. Also it defines some of the abstract method which must be implemented by any class that extends AbstractTableModelFilter.

We will discuss only couple of filters namely SortingFilter and RowVisibilityFilter. I also tried to look into ReflectiveFilter but could not find any part which could make use of Java 7 parallel features. Also it makes use of Swing components directly which makes it hard to parallelize. On the other hand, the grouping filter can be made parallel if sorting filter becomes parallel because if we want to group the elements we first need to sort them and then group them. It is the sorting part which could be made in parallel.

12.1 Row Visibility Filter

The row visibility filter is responsible if we want to filter out the rows based on some criteria. We have also discussed how that works previously. Of course, there is a simple O(n) algorithm to do this. We can check each row one by one and keep the information of only those rows which satisfy the filter criteria. However, to get used to the Kiwiplan framework I modified this example to divide-and-conquer algorithm. This made the algorithm runtime to O(n.log n). This obviously would not achieve any performance benefit, but I wanted to start using Kiwiplan Framework with some simple example.

The O(n.log n) algorithm recursively divide the tasks in 2 halves. In the base case, i filter out rows using the for-loop. We have an instance of RowMapper. It is the class in the Kiwiplan framework which saving the forward mapping and backward mapping we discussed previously. The program is shown in figure 31.

```
private void sequentialRecursiveTaskNoMerge(int low, int high, RowMapper mapper) {
    if (high - low <= STOP_DIVIDING) {
        for (int i = low; i <= high; i++) {
            if (shouldDisplayRow(i)) {
                mapper.setRowMapping(mapper.size(), i);
            }
        }
        return;
    }

    int middle = (high + low) / 2;
    sequentialRecursiveTaskNoMerge(low, middle, mapper);
    sequentialRecursiveTaskNoMerge(middle+1, high, mapper);
}
```

Figure 31: Row Visibility Sequential Algorithm

STOP DIVIDING is some threshold we used for the base case. In this example, I set that to 15 after making some experiments with the other values as well. We are using the method call mapper.setRowMapping() in the base case. Initially mapper.size() is 0. We makes use of the same example we used previously to explain how this algorithm works. The table is shown below :-

Name	Age	City
Bob	25	Auckland
Alice	20	Wellington
Samuel	37	Napier
John	27	Auckland
Kathy	29	Auckland

Figure 32: Row Visibility Example

And we want only those rows whose Age field is between 25 and 30 inclusively. Initially mapper.size() will be equal to 0. So, when we used mapper.setRowMapping() we set mapper.size() (0) to i (0) because it is the 1^{st} row of the table. This makes the mapper size to 1. Next time it will be called for the 3^{rd} row of the table, so we map mapper.size (1) to i (2). We follow the same logic to map other rows as well. In the end we obtain this result :-

Name	Age	City
Bob	25	Auckland
John	27	Auckland
Kathy	29	Auckland

Figure 33: Row Visibility Operation Result

This should be easy to see how it can be made to run in parallel. The method does not return anything so it gives us an idea that the new class we make should extend RecursiveAction[12]. We use some sequential threshold which works well if we set it to 1% of our array size. If we get below our sequential threshold, we simply use the for-loop.

Another point to note is this method does not have any merge step. This will not be always true and we demonstrate that when we discuss Sorting filter.

[12]If we have to use C++ parallel pattern library then we do not have to make any new class.

The using the step mentioned previously we make this algorithm run in parallel. The parallel version is shown in figure 34.

```
class FilterTaskNoMerge extends RecursiveAction {
    int low;
    int high;
    RowMapper mapper;

    public FilterTaskNoMerge(int low, int high, RowMapper mapper) {
        this.low = low;
        this.high = high;
        this.mapper = mapper;
    }

    @Override
    protected void compute() {
        if (high - low <= SEQUENTIAL_THRESHOLD) {
            for (int i = low; i <= high; i++) {
                if (shouldDisplayRow(i)) {
                    synchronized (this.mapper) {
                        mapper.setRowMapping(mapper.size(), i);
                    }
                }
            }
            return;
        }
        int middle = (high + low) / 2;
        invokeAll(new FilterTaskNoMerge(low, middle, mapper),
                  new FilterTaskNoMerge(middle + 1, high, mapper));
    }
}
```

Figure 34: Row Visibility Parallel Algorithm

We made a new class which extends RecursiveAction. All the parameters of the methods have now become the parameters of our constructor method. We have to override the compute() method. It is important to note we used synchronized(this.mapper) inside the for-loop. The reason for this is the setRowMapping() method calls mapper.size() which makes this statement non-atomic. This statement is divided into two steps. In step 1 we are getting the value of mapper.size() and in the second step we are using that value in setRowMapping() method. We want these two steps to be atomic and do not want two threads to interleave with each other when one thread is in-between these two steps. So, only one thread should have an access to this part at any given time. That is the reason, why we locked it on mapper as mapper is the mutable object in this example.

Rest of the code is very similar to our previous example. We divided the task in two halves and makes use of invokeAll() method to run these two task in parallel. And as follows from the sequential version of this algorithm there is no merge step. This is because we only have one RowMapper instance and it is actually the global variable. Since objects are being passed by reference, any changes made to the instance of RowMapper would be reflected from where this method is being called. It is easy to see that the only time where threads are not running in parallel is the synchronized block. Let's compare the runtime of the sequential and parallel version of this algorithm. This is shown in figure 35.

Problem Size	Sequential Running Tim	Parallel Running Time
100	0	29
1000	2	28
10000	4	35
100000	22	76
1000000	450	112
5000000	1312	765

Figure 35: Performance comparison of Row Visibility Sequential and Parallel Algorithm

As we can see it is better to use sequential program for the value of 1 hundred thousand and less. It takes time to create threads, allocating work to them, creating lock and releasing lock. So, if the list is not big enough it is better to use sequential version. However, once the problem size goes more than 1 million we have a clear winner. The parallel algorithm runs takes much less time as compared to the sequential one.

These results are obtained on dual core machine. I tried my program for the input size of 5 million and less. Going more than that I was getting out of memory error. We can run our java program with extending memory. However, since the lab only installs java 6[13] and I am only using the jar files, I could not use command prompt to run my program. However, it seams that with increasing problem size it is better to use the parallel version. Let's consider the parallel version again with slight modification (only the compute method is shown as there is no changes in the rest of the code).

```
@Override
protected void compute() {
    if (high - low <= SEQUENTIAL_THRESHOLD) {
        synchronized (this.mapper) {
            for (int i = low; i <= high; i++) {
                if (shouldDisplayRow(i)) {
                    mapper.setRowMapping(mapper.size(), i);
                }
            }
        }
        return;
    }
    int middle = (high + low) / 2;
    invokeAll(new FilterTaskNoMerge(low, middle, mapper),
              new FilterTaskNoMerge(middle + 1, high, mapper));
}
```

Figure 36: Row Visibility Parallel Algorithm - Slow version

In this the synchronized block covers the whole for-loop. Clearly this is not an efficient way as the threads are blocked for longer period of time. We set the lock before we started the for-loop and once we finish with our for-loop only then we release the lock. It means if there are 100 elements then only one thread will do these 100 steps at any given time. It's runtime is being compared with the sequential version next.

[13]Java 7 is not officially released yet.

Problem Size	Sequential Running Time	Parallel Running Time
100	0	31
1000	2	36
10000	4	40
100000	22	91
1000000	450	415
5000000	1312	1193

Figure 37: Runtime of Row Visibility Slow version

As we can see, it is definitely slower than the parallel algorithm we presented before. So, important point to remember is if we have a synchronized block in our code then we should try to minimize it as much as possible by keeping in thread safe.

12.2 Sorting Filter

Sorting filter contains the algorithm that sort the elements either in ascending or in descending order. Earlier we presented and show how the mapping changes after the executing of the sorting algorithm. The algorithm used in Kiwiplan framework is the shuttle sort algorithm. Shuttle sort algorithm is very similar to the merge sort algorithm we discussed previously, with two important changes :-

1. In Merge sort we were making the copy of the original array at every step and then copying the values back to original array once the merge step completes. In Shuttle sort make a copy of the original array at the very first time we call it and then values are shuttled between the two copies of the array. Hence, it's name is Shuttle sort.

2. In the merge step of the Merge sort algorithm we had two sorted halves and we compare each element one by one when we merge. However, let's if all the elements in the first sorted array are less than or equal to the elements in the second sorted array, then we do not need to compare each element explicitly. We can simply merge the two sorted array. This is to improve the performance. Of course, we do not have to check explicitly that whether each element of the first array is smaller than the second one or not. All we have check if array[middle] ¡= array[middle+1], then we could simply join the two arrays together without explicit checking. Figure 38 makes this point even more clear.

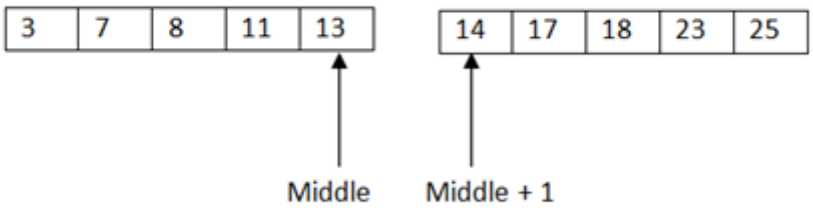

Figure 38: Point 2 Example

The example above showed the value 14 at *Middle+1* position. However, even it was 13 we would have done the same thing.

Another important point to note is that the Shuttle sort algorithm is stable in nature. Sorting algorithms are stable when they do not change the ordering of the keys that have an equal value. It is important to have that property and let's consider the example given in figure 39 to understand why it is important.

Name	Age
Bob	35
Bob	50

Figure 39: Demonstrating why sorting algorithm should be stable

Let's say we have these two rows in our table and the user had already sorted the table on "Age" column. Now, when the user sorts the table based on"Name" column the order should not be changed. Otherwise, age 50 would come before 35 which is not correct. Hence, the sorting algorithm should be stable and Shuttle sort algorithm is.

The Shuttle sort algorithm sequential version is presented in figure 40.

```
private void sequentialMergeSortWithTwoRowMapper(int low, int high, RowMapper from, RowMapper to)
    if (high - low <= INSERTION_SORT_THRESHOLD) {
        insertionSortRowMapper(low, high, to);
        return;
    }

    int middle = (low + high) / 2;
    sequentialMergeSortWithTwoRowMapper(low, middle, to, from);
    sequentialMergeSortWithTwoRowMapper(middle + 1, high, to, from);

    //Point 2 we discussed above
    if (array[from.mapRow(middle)].compareTo(array[from.mapRow(middle+1)]) <= 0) {
        for (int i = low; i <= high; i++) {
            to.setRowMapping(i, from.mapRow(i));
        }
        return;
    }

    //Merge step is similar to Merge sort
    int leftPointer = low, rightPointer = middle + 1, arrayPointer = low;

    while (leftPointer <= middle && rightPointer <= high) {
        if (array[from.mapRow(leftPointer)].compareTo(array[from.mapRow(rightPointer)]) > 0) {
            to.setRowMapping(arrayPointer++, from.mapRow(rightPointer++));
        } else {
            to.setRowMapping(arrayPointer++, from.mapRow(leftPointer++));
        }
    }

    while (leftPointer <= middle) {
        to.setRowMapping(arrayPointer++, from.mapRow(leftPointer++));
    }

    while (rightPointer <= high) {
        to.setRowMapping(arrayPointer++, from.mapRow(rightPointer++));
    }
}
```

Figure 40: Shuttle sort sequential version

Please note the two recursive calls we are making. The RowMapper instances have been swapped. We always make changes to the RowMapper instance named *to*. Similar to merge sort, we are using

the hybrid of shuttle sort with the insertion sort. That makes the algorithm bit more efficient. Now to convert this to parallel algorithm I tried various ways namely - start with one row mapper. In this case I had to make a copy at every merge step and the synchronized block was too much. Due to this i did not get any performance benefit out of it.

The shuttle sort algorithm does not return anything which might trap us in thinking that we should extend RecursiveAction. However, after trying various methods, it is best to use RecursiveTask here where the return type is RowMapper. Program is presented in figure 41.

```
class ParallelSortTask extends RecursiveTask<RowMapper> {
    private static final long serialVersionUID = 1L;
    private int low;
    private int high;

    public ParallelSortTask(int low, int high) {
        this.low = low;
        this.high = high;
    }

    @Override
    protected RowMapper compute() {
        if (high - low <= SEQUENTIAL_THRESHOLD) {
            RowMapper mapper = new RowMapper(high+1);
            sequentialMergeSortWithOneRowMapper(low, high, mapper);
            return mapper;
        }

        int middle = (low + high) / 2;
        ParallelSortTask leftTask = new ParallelSortTask(low, middle);
        leftTask.fork();
        ParallelSortTask rightTask = new ParallelSortTask(middle + 1, high);

        RowMapper right = rightTask.compute();
        RowMapper left = leftTask.join();

        if (array[left.mapRow(middle)].compareTo(array[right.mapRow(middle + 1)]) <= 0) {
            RowMapper merge = new RowMapper(high+1);
            for (int i = 0; i < left.size(); i++) {
                merge.setRowMapping(i, left.mapRow(i));
            }
            for (int j = middle+1; j < right.size(); j++) {
                merge.setRowMapping(j, right.mapRow(j));
            }
            return merge;
        }
```

Figure 41: Shuttle sort parallel version

We are creating two subtask where each of them return an instance of the RowMapper and all we have to do is to merge them together. One of the benefit of this approach is we do not have to synchronize anything because we are not making changes to the two subtask RowMapper instances. Since the are not getting changed we do not need to provide any locking to ensure thread safety. Rest of the code is shown in figure 42.

```
        RowMapper merge = new RowMapper(high+1);
        int leftPointer = low, rightPointer = middle + 1, arrayPointer = low;

        while (leftPointer <= middle && rightPointer <= high) {
            if (array[left.mapRow(leftPointer)].compareTo(array[right.mapRow(rightPointer)]) > 0) {
                merge.setRowMapping(arrayPointer++, right.mapRow(rightPointer++));
            } else {
                merge.setRowMapping(arrayPointer++, left.mapRow(leftPointer++));
            }
        }

        while (leftPointer <= middle) {
            merge.setRowMapping(arrayPointer++, left.mapRow(leftPointer++));
        }

        while (rightPointer <= high) {
            merge.setRowMapping(arrayPointer++, right.mapRow(rightPointer++));
        }

        return merge;
    }
```

Figure 42: Rest of the Shuttle Sort Parallel code

Now that we have presented the sequential and parallel version of the Shuttle sort, it is time to compare their runtime to see what performance we have achieved. This is shown in figure 43.

Problem Size	Sequential Running Tim	Parallel Running Time
100	2	31
1000	7	34
10000	25	56
100000	213	285
1000000	3500	2630
1500000	5326	3743

Figure 43: Performance comparison of Shuttle sort Sequential and Parallel Algorithm

And similar to RowVisibility algorithm, it is better to use sequential algorithm if the problem size is not much. However, if the problem size goes more than 1 million, it is better to use the parallel algorithm. Due to memory limit i could not test it for an input of more than 1.5 million, but from the last two entries of the table it is easy to see Parallel algorithm has high performance as compared to the sequential one for large input size.

13 Conclusion And Future Work

In this report, we started of presenting why parallel programming is important to learn and what the project goal is. We presented some of the important features of the concurrency library introduced in Java 7. Later we showed how could we make use of these techniques to make the algorithm written in Kiwiplan framework to run in parallel. Still there is lot of work needs to be done. I only looked at some of the filters algorithm, but still needs to investigate how other filters can be made to run in parallel as well. Java 7 is not officially released yet, so I found it hard to find any material on the Internet which provides shows some in-depth features of Java 7. Once Java 7 will be released, many more resources would be available which would make it easier to understand how to apply fork/join framework to some more advanced problems.

14 References

1. Java Tutorials, Processes and Threads, Retrieved October 20, 2010, from "http://download.oracle.com/javase/tutorial/essential/concurrency/procthread.html".

2. Java API, Threads, Retrieved October 20, 2010, from "http://download.oracle.com/javase/1.4.2/docs/api/java/lang/Thread.html"

3. Goetz B. (2006). Java Concurrency in Practice.Addison-Wesley Professional, page 17.

4. Goetz B. (2006). Java Concurrency in Practice.Addison-Wesley Professional, page 9.

5. Java Tutorials, Executor Framework, Retrieved October 21, 2010, from "http://download.oracle.com/javase/tutorial/essential/concurrency/executors.html"

6. Learning Java, Executor Framework, Retrieved October 21, 2010, from "http://www.particle.kth.se/ lindsey/JavaCourse/Book/Part1/Java/Chapter10/concurrencyTools.html"

7. Java Tutorials, Thread Pools, Retrieved October 21, 2010, from "http://download.oracle.com/javase/tutorial/essential/concurrency/pools.html"

8. Java Tutorials, ForkJoin Framework, Retrieved October 21, 2010, from "http://download.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html"

9. Operating System, Retrieved October 21, 2010, from "http://en.wikipedia.org/wiki/Operating_system"

10. Sutter, H. (2005). The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal, 30(3).

11. Timothy W. (2003). Kiwiplan Reporting Tool. BTech Project Report.

12. Model-View-Controller Pattern, Retrieved October 19, 2010, from "http://www.enode.com/x/markup/tutorial/mvc.html"

13. Java 7: More Concurrency, Retrieved October 22, 2010, from "http://www.baptiste-wicht.com/2010/04/java-7-more-concurrency/"

14. Java 7 API, Fork/Join Framework, Retrieved October 23, 2010, from "http://download.java.net/jdk7/docs/api/"

15. Goetz, B. (2008) Java Theory And Practice: Stick A Fork In It - Part 2, Parallel Array, Retrieved October 24, 2010, from "http://www.ibm.com/developerworks/java/library/j-jtp03048.html"