Java Code Transformation for Parallelization

Muhammad Usman Iftikhar
2010-06-28
Subject: Software Technology
Level: Master
Course code: 5DV01E
Abstract
This thesis describes techniques for defining independent tasks in Java programs for parallelization. Existing Java parallelization APIs like JOMP, Parallel Java, Deterministic Parallel Java, JConqurr and JaMP are discussed. We have seen that JaMP is an implementation of OpenMP for Java, and it has a set of OpenMP directives and runtime library functions. We have discussed that JaMP has source to byte code compiler, and it does not help in debugging the parallel source codes. There is no design time syntax checking support of JaMP directives, and we know about mistakes only when we compile the source code with JaMP compiler. So we have decided to contribute JaMP with adding an option in the compiler to get parallel source code. We have created an eclipse plug-in to support design time syntax checking of JaMP directives too. It also helps the programmers to get quickly parallel source code with just one click instead of using shell commands with JaMP compiler.

List of Keywords
Parallel Java, Parallel processing, Parallelization, OpenMP, JaMP, JOMP, Deterministic Parallel Java, DPJ, PJ, Cluster, Hybrid, Amdahl’s law, Parallel APIs, JConqurr
Acknowledgements
I am heartily thankful to my supervisors, Prof. Dr. Welf Löwe and Dr. Jonas Lundberg, whose encouragement; guidance enabled me to develop an understanding of the subject.

I am also thankful to all of those who supported me during the completion of the thesis.
List of figures
Figure 2-1 Amdahl's law speedup graph [31]................................................................. 3
Figure 2-2A multi-core processor architecture [26]....................................................... 4
Figure 2-3A typical cluster architecture ......................................................................... 4
Figure 2-4A typical hybrid architecture ......................................................................... 4
Figure 2-5 The core elements of OpenMP API [27]......................................................... 5
Figure 2-6 OpenMP fork-join model [28]........................................................................ 6
Figure 4-1 JaMP Builder Plugin menu options ............................................................... 20
Figure 4-2 Example of design time error detection by JaMP Builder Plugin .................. 21
Figure 5-1 The output of non-parallel version of the experiment ................................. 24
Figure 5-2 Manual parallel version of the experiment .................................................... 26
List of Abbreviations

1. **JCUDA**: Java bindings for CUDA
2. **CPU**: Central Processing Unit
3. **API**: Application Programming Interface
4. **SMP**: Shared Memory Parallel Computer Architecture
5. **MPI**: Message Passing Interface
6. **GPU**: Graphical Processing Unit
7. **OpenMP**: Open multi-processing
8. **JOMP**: Java OpenMP
9. **PJ**: Parallel Java
10. **DPJ**: Deterministic Parallel Java
11. **JaMP**: Java/OpenMP
1 Introduction

Traditionally, Computer software(s) are written as series of instructions. These instructions are executed serially one after another by the central processing unit (CPU). Usually these CPUs are capable of only executing one instruction at any moment in time [1].

On the other hand, parallelization use simultaneous multiple computing resources to solve a problem. Each problem is divided into parallel tasks which can run concurrently. Each task can have series of instructions. Then instructions from each task can run concurrently on different computing resources. These computing resources can be a, single processor with multiple cores, single computer with multiple processors, network which is made of an arbitrary number of computers connected to each other or any combination of the above [1].

Parallelization is becoming future of programming. Single core high speed processors are expensive and industry shifting the gear to put as many cores as possible in single processor. This leads to a problem that a lot of code exists written to work on single core processor. And if we have multi-core processor only one core will work, and it will not fully exploit the multi-core processor capabilities. According to Jim McGregor, research director and principal analyst at In-Stat [2];

"Parallel processing is essential because every processor architecture is now available in multi-core configurations. Programming, however, remains a challenge, which is why there is interest in research and development at the university level."

1.1 Motivation

Major processor suppliers have released processors with 2, 4 and 6 cores and are currently working on processors with 8 and 16 cores. Moreover, even standard desktop computers can contain more than one such processor. Applications can speed up if (and only if) independent application tasks are processed in parallel, e.g., using multiple cores, allocating each independent task on a separate core. However, this requires that the solution applications define independent tasks for the problems they solve.

1.2 Problem Definition

There are no good tools available for programmers, which can help programmers to quickly define independent tasks in sequential Java programs. The idea is to support the programmers for introducing parallelism in sequential Java programs and create a starting point, which will be the basis for the future projects.

1.3 Research Question

As thesis is on study of different code transformation techniques for parallelization, we will try to answer these questions:

- How can we define independent tasks in a given computing problem?
- How can we do the code transformation of independent tasks?
- How can we define specific number of threads or cores to be used by specific independent tasks?

1.4 Goals

Introducing independent tasks in sequential Java programs is the main goal of our thesis. Code annotations assure the independence of statements; code transformations introduce parallelism in the formerly sequential code. In addition we are limiting our focus on the shared memory parallel programming (see Chapter 2.2.1) in Java.
1.5 Scope
There are a lot of parallel APIs for Java which can be used to define independent tasks, but there is no useful source code editor which can be used to quickly write parallel code by using these APIs. The scope of the thesis is to select an API by exploring existing Java parallelization APIs, and create an Eclipse plugin supporting the development of parallel Java programs. The programmers annotate their code with parallelization instructions, and our tool transform the annotated (sequential) source code into a corresponding multi-threaded version. The plugin will have syntax check feature, and it will be good enough to check for design time errors in the code. The final result will be an eclipse plugin with set of directives to transform Java code for parallelization.

1.6 Plan
We plan to survey existing shared memory Java parallelization APIs. We will try to find out an API as a starting point. After that we will try to make an Eclipse plugin as wrapper on selected API.

Further, we plan to evaluate the API with applying parallelism on number of different matrix multiplication and merge sort experiments.

1.7 Report Structure
The core of report covered in chapters 2-4. Chapter 2 describes the existing research in Java parallelization APIs, e.g. JOMP, Parallel Java (PJ), Deterministic Parallel Java, and JaMP. Chapter 3 describes the OpenMP architecture and how it is used in JaMP. Chapter 4 describes our contribution, the Eclipse Plugin and how to use it.

In chapter 5, we present some preliminary performance results, from number of different experiments.

Chapter 6 concludes and discusses future tasks.
2 Background

In this chapter we are discussing existing parallelization concepts and Java parallelization APIs. First we are discussing Amdahl’s law, after that classification of parallel computer architectures, and in the end some existing Java Parallelization APIs and architecture which they support.

2.1 Amdahl’s Law

Amdahl’s law, is also known as Amdahl’s argument, and is used to find the maximum theoretical speedup of a program using multiple processors. For example, if a program takes 2 hours using a single processor, 1 hour is parallel and a particular 1 hour is critical and cannot be parallelized. So it means that program execution time cannot be decreased more than critical 1 hour regardless of how many processors are used [31]. So speedup of the program is limited to 2x as shown in figure 2-1.

![Amdahl's law speedup graph](image)

Figure 2-1 Amdahl's law speedup graph [31]

2.2 Classification of Parallel Computer Architectures

The three most common parallel computer architectures are; SMP, Clusters, and Hybrid.

2.2.1 Shared Memory Parallel Computer Architecture (SMP)

In SMP architecture two or more processors are connected with each other and shared a common memory. Each processor has equal access to memory and controlled by a single operating system. Most modern multi-core processors are based on SMP architecture. In the case of SMP, cores are treated as separate processors [3].

A multi-core processor contains two or more cores in a single package. Cores are units that can read and execute program instructions independently. The processor which has two cores is called dual-core or core 2 duo processor and a processor which has four
cores is called **quad-core** processor. Depending on the design cores can be coupled loosely or tightly in a multi-core processor. For example, each core can have its own L1 cache, for communication between cores message-passing or shared memory communication methods can be used. And cores may or may not be identical in a multi-core processor. If cores are identical then system is called homogeneous multi-core system. If cores are not identical then it is called heterogeneous multi-core system [26].

A typical example of a multi-core processor is shown in figure 2-2.

![Figure 2-2A multi-core processor architecture [26]]

**2.2.2 Clusters**

Two or more computers working together closely behaving like a single computer is called a cluster. Most commonly computers are connected with each other through LAN. Clusters are used to share work load and provides high availability of the services [4].

A typical example of a cluster is shown in figure 2-3.

![Figure 2-3A typical cluster architecture]

Unlike SMPs, clusters use distributed memory architecture. Each computer has its own memory and other computers may or may not have access to that memory. For parallelization clusters use MPI (Message Passing Interface) [32] API.

**2.2.3 Hybrid**

When a cluster has SMP computers at its nodes then it is called Hybrid architecture. It is most famous in a sense that most modern computers are SMP already [5]. Figure 2-4 showing a typical example of hybrid architecture.

![Figure 2-4 A typical hybrid architecture]
2.3 OpenMP

OpenMP (Open Multi-processing) is an API for shared memory parallel programming in C/C++ [22] and FORTRAN [23]. It has been jointly defined by a group of major software and hardware vendors. The OpenMP ARB (Architecture Review Board) is a nonprofit organization which owns the OpenMP brand. It is responsible for creating and supervising new OpenMP specifications. In October 1997, first OpenMP specification for FORTRAN was published. The current version is 3.0 [27] [28].

2.3.1 The Core Elements

OpenMP is an explicit (not automatic) programming model. It consists of a set of compiler directives, library routines and environment variables that control the run-time behavior of a program. The core elements of OpenMP are directives for thread creation (parallel control structures), work sharing (work load distribution between threads), data environment management, thread synchronization, run-time functions of API and environment variables as shown in figure 2-5.

In C/C++ the format of OpenMP directives are:

```
#pragma omp <directive> <[clause][, clause] ...>
```

Clauses are attributes of a directive and they are optional. Example 2-1 is a simple C/C++ program that is printing “Hello, World.” using multiple threads by OpenMP.

```c
void main(void)
{
    #pragma omp parallel
    printf("Hello, world.");
}
```

Example 2-1 A simple OpenMP program

The ‘omp parallel’ will create a team of threads. All threads of the team will print the ‘Hello, World’. Because we have not specified any work sharing directives yet. Work sharing directives are ‘sections’, ‘for’, ‘master’ and ‘single’ which can be used to divide the work to one or all threads of a team.

- The ‘sections’ directive is used to assign consecutive code block to different threads.
- The ‘for’ directive is used to divide iterations of the loop between different threads.
- The ‘master’ directive is used to assign the code block to only master thread.
• The ‘single’ directive is used to assign the code block to any one thread. We have discussed OpenMP directives in more detail at chapter 3.

2.3.2 Fork-Join Model
OpenMP uses the fork-join model for parallel execution. All programs start executing sequentially by a thread called ‘master thread’ until the first parallel directive occurs. Parallel directives create a team of threads called ‘slaves’ and divide the work between them. This process is called ‘Fork’. After finishing all the work, all threads stop at the end of parallel region and only the master thread continues after it. This is called ‘Join’. Figure 2-6 explains the fork-join model.

![Figure 2-6 OpenMP fork-join model][2]

2.3.3 Pros and cons of OpenMP
Here are some pros and cons of OpenMP:
Pros:
• Suits multi-core processors
• Easy to use
• Requires less code modification in the serial version of code
• OpenMP directives can be ignored if the OpenMP compiler is not available
• Any program can be made parallel incrementally.
Cons:
• Only supports shared memory parallelization
• OpenMP compiler is needed to compile the source programs

2.4 Java support for Parallelization
Currently, Java has very basic support for Parallelization. It supports creation of threads and child processes. The base class for threads is ‘java.lang.Thread’ which executes an object of type ‘java.lang.Runnable’. The ‘Runnable’ is an interface which has only one method ‘run()’. This method is called by ‘Thread’ and it would contain the work which should be done in separate thread [30]. Example 2-2 shows a simple Java program.

```java
public class HelloWorld {
    public static void main(String[] args) {
        // creating anonymous class definition
        Runnable runnable = new Runnable() {
            public void run() {
                System.out.println("Hello, World!");
            }
        };
    }
```
// Run the class in separate thread
Thread thread = new Thread(runnable);
thread.start();
}
}

Example 2-2 A simple Java program

These threads can be managed by a thread pool. A thread pool is a collection of runnables (work queue) and running threads. In the thread pool, threads run all the time and they check the work queue for their work. If there is any runnable waiting for execution they execute it. In Java thread pools can be created by static ‘Executors’ class. Example 2-3 shows the simple Java program using thread pools.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class HelloWorld {

    public static void main(String[] args) {
        // Create anonymous class definition
        Runnable runnable = new Runnable() {
            public void run() {
                System.out.println("Hello, World!");
            }
        };

        // Create a fixed sized thread pool. And run anonymous class twice. After that shutdown the executor
        ExecutorService executor = Executors.newFixedThreadPool(2);
        executor.execute(runnable);
        executor.execute(runnable);
        executor.shutdown();
    }
}
```

Example 2-3 A simple Java program with thread pools

The ‘Executors.newFixedThreadPool(2)’ creates a thread pool which will have two fixed threads, running all the time. If additional tasks are submitted when these two threads are active, they will wait in the queue until a thread is available. The ‘executor.execute(runnable)’ function is used for submitting the runnables to work queue. The ‘runnable’ object has been submitted two times so it will print “Hello, World!” two times. The ‘executor.shutdown()’ is used to close the executor. It will execute any pending runnables and then will close itself.

Although Java has good support for parallelization we need to do it all manual. For example if we need to divide a code between four threads then we need to write manual four threads code for it. And it will not take care that system has less computing resources (cores) or more. It will always create four threads and will execute them. This kind of parallelization is not supported by Java yet, and we need to write some kind of APIs which will be built upon Java native threads and provides automatic parallelization.
2.5 Existing Java Parallelization APIs

In this section we are discussing some Java parallelization APIs and how we can use them to add independent tasks. For this purpose let’s take a sequential ‘for’ loop code and suppose that ‘N’ is total number of threads to set for a given program and we will see how it can be made parallel using each API.

```java
public void doSomething(){
    for (int i = 0; i < n; i++)
    {
        // some code here
    }
}
```

Example 2-4 sequential for loop

2.5.1 JOMP

JOMP (Java OpenMP) is introduced for shared memory parallel programming. It consists of a pre-compiler and a runtime library. Its goal was to define and implement an OpenMP like set of directives. These are special Java inline comments. The pre-compiler recognize them and treat them as OpenMP directives but any other Java compiler will treat them as comments and ignore them [6].

The JOMP pre-compiler is a source to source Java compiler. It transforms the JOMP Java source code to parallel source code with call to runtime library functions. Parallel source code then can be compiled and run with any Java compiler [9]. The JOMP Runtime library use Java threads to implement parallelism. JOMP is pure Java and can run on any virtual machine [7]. Let us take an example about using this API on our sample for loop.

```java
public void doSomething(){
    OMP.setNumThreads(N);
    //omp parallel for
    for (int i = 0; i < n; i++)
    {
        // some code here
    }
}
```

Example 2-5 parallel for loop using JOMP

‘//omp parallel for’ is an OpenMP directive. It is a short hand for parallel region containing a single for directive. It tells the compiler to create a parallel region containing only one for loop. Parallel region consists of team of threads and ‘for’ directive tells the compiler to divide the iterations of the loop between these threads.

The number of threads for parallel region can be set explicitly by calling runtime library function ‘setNumThreads(N)’ before any parallel directive. We can set number of threads by setting JVM arguments also. If there is no threads specified then team size depends upon the implementation and default team size [8].

As JOMP is an implementation of OpenMP it has inherited all the pros and cons of it. But there are some few more which need to be discussed.

**Pros:**
- It transfers the serial Java code to parallel version of it, which makes debugging easy.

**Cons:**
Currently, it is not an active project. There are more advance APIs are available.

- Source code is not available.
- It uses Java comments so no design time syntax checking support.

### 2.5.2 Parallel Java (PJ)

Parallel Java is an unified parallel programming API whose support SMP, cluster and hybrid parallel programming. It is 100% Java API. It contains a class library which has classes for multithreaded shared memory parallel programming and message passing cluster parallel programming, and these are fully integrated together to support hybrid parallel programming. SMP programming features are inspired by OpenMP and cluster parallel programming features are inspired by MPI (message passing interface) [9].

Let us see how we can use this API to work on our sample code.

```java
public void doSomething(){
    try {
        new ParallelTeam(N).execute(new ParallelRegion() {
            @Override
            public void run() {
                try {
                    execute(0, size, new IntegerForLoop() {
                        @Override
                        public void run(int start, int end) throws Exception {
                            for (int r = start; r < end; r++) {
                                // some code here
                            }
                        }
                    });
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Example 2-6 parallel ‘for’ loop using Parallel Java

There are two constructors available for creating an object of ‘ParallelTeam’, with integer argument like ‘ParallelTeam(N)’ we can specify the number of threads to divide and execute the loop, and without argument loop will be divided and executed with all available threads.

Here are some pros and cons of PJ:

**Pros:**
- It is not using Java Comments.
- It supports cluster and hybrid parallel programming.
- Design time checking of syntax is possible.
- Source code is available.
Cons:
- Much more coding!

2.5.3 Deterministic Parallel Java (DPJ)

DPJ is an extension to Java programming language that guarantees “deterministic-by-default” semantics for programs. “Deterministic” means that parallel program will produce the same output on all executions of the program for a given input. “By default” means that unless the programmer explicitly requests non-determinism, deterministic behavior will be guaranteed.

Every Java program is legal DPJ program but reverse is not possible, because DPJ adds several features to DPJ program to guarantee determinism [11]. Like JOMP, compiling DPJ program is a two-step process. First, we use the DPJ compiler ‘dpjc’ to translate DPJ program to plain Java program. Then we can use any Java compiler to compile and run the program [13].

DPJ have three basic components to understand: explicit fork-join parallelism, regions and effects [12].

Example 2-7 shows a simple DPJ class to represent these concepts.

```java
1 class Point {
2     region X, Y;
3     double x in X;
4     double y in Y;
5     void setX(double x) writes X { this.x = x; }
6     void setY(double y) writes Y { this.y = y; }
7     void setXY(double x, double y) writes X, Y {
8         cobegin {
9             this.setX(x);
10             this.setY(y);
11         }
12     }
13 }
```

Example 2-7 Simple DPJ program [12]

1. Explicit fork-join Parallelism
In the simple DPJ program ‘cobegin’ statement tells the compiler explicitly that the where parallel section begins (fork) and ends (join). In fig. 2-4 the code at line 9 and 10 will run parallel. All the threads created by fork will wait for each other at join point to finish the execution. Program will continue its execution after ‘cobegin’ until both lines are completely executed.

2. Regions
Regions are named heap memory locations. A region can have more than one class field but one field cannot exist in two regions. In fig. 2-4 at line 2 we are creating 2 regions X and Y and at line 3 and 4 saying that field x is belongs to region X and field y is in region Y.

3. Effects
To make the compiler analysis better programmers write effect summaries of each method. An effect summary tells the compiler that in which region method will write something and from which region it will read. It helps the compiler to make determinism behavior of the program.

Let us use DPJ on our sample code.
public void doSomething(){
    foreach (int i in 0, n, 1)
    {
        // some code here
    }
}

Example 2-8 parallel for loop using DPJ

The ‘foreach’ loop has three parameters: start, length and stride. All parameters should be integer expressions. Stride is optional and by default is one [14].

We can set the number of threads for a program by setting the command line argument ‘--dpj-num-threads N’ [13].

Here are some pros and cons of DPJ:

Pros:
- Deterministic behavior!
- Source code available.
- Generate the parallel code which can make debugging easy.

Cons:
- Much more coding!
- A new set of DPJ keywords to learn.
- No design time checking support.

2.5.4 JConqurr

JConqurr is a multi-core and many-core programming toolkit for Java language. It has directives and annotations to provide meta information of the source code. It comes as an Eclipse IDE plugin. Based on the meta information passed from the users it converts sequential Java projects to a new Java project which is optimized on the parallel patterns. It supports the patterns like task, data, divide and conquer and pipeline parallelism. For many-core programming it use the graphical processing unit GPU and JCUDA [17] if available. It supports both CPU and GPU parallelism [16].

Let us apply JConqurr on our sample code:

```java
@ParallelFor
public void doSomething(){
    Directives.forLoop();
    for (int i = 0, i < n, i++)
    {
        // some code here
    }
}
```

Example 2-9 parallel for loop using JConqurr

The ‘@ParallelFor’ is an annotation used by JConqurr for ‘for’ loop. It is used to filter the methods which have ‘for’ loop. The ‘Directives.forLoop()’ is used to specify the loop which needs parallelism. Based on the number of processors are available the toolkit decides the number of splits are needed for ‘for’ loop.

Here are some pros and cons of JConqurr:

Pros:
- Provide support for many parallelization patterns.
- Source code available.
• Delivered as an Eclipse IDE plug-in.

Cons:
• No design time syntax checking support.
• Currently, it is under construction and not properly working.

2.5.5 JaMP
JaMP (Java/OpenMP) is another implementation of OpenMP for Java. It is like JOMP but it has compiler which converts the JaMP source programs directly to Java class files. It supports full OpenMP 2.0 specification. It generates pure Java 1.5 codes that can run on every Java virtual machine. If a system has CUDA [18] enabled graphic card it can use that for parallel ‘for’ loops to gain extra speed. And if system does not have CUDA graphic card then it translates them to multi threaded code [19].

Since it is using OpenMP, its directives are similar to JOMP:

```java
public void doSomething(){
    omp.setNumThreads(N);
    //#omp parallel for
    for (int i = 0; i < n; i++)
    {
        // some code here
    }
}
```

Example 2-10 parallel for loop using JOMP

Like JOMP, it has inherited pros and cons of OpenMP, but there are some others:

Pros:
• Source code is available.

Cons:
• No design time syntax checking available.
• It converts Java programs to byte code Java class files directly, so no debugging support of parallel source code is available.

2.6 Evaluation
We have defined some factors to compare APIs. Table 2-1 is showing the comparisons.

<table>
<thead>
<tr>
<th></th>
<th>JOMP</th>
<th>PJ</th>
<th>DPJ</th>
<th>JConquerr</th>
<th>JaMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure Java</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Active Project</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Design Time Checking</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Source Code Available</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Existing Code can run without using API</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Easy to use</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Any other Issue?</td>
<td>Does not support Java 1.5.</td>
<td>Using a lot of memory. Got out of memory error in our Matrix multiplication experiment.</td>
<td>No</td>
<td>Toolkit not working properly.</td>
<td>No</td>
</tr>
</tbody>
</table>
Table 2-1 Comparison Table of parallelization APIs

From the comparison table we can see that all the APIs use pure Java code. Pure Java means, the APIs and the code generated by them can be run on any Java virtual machine. Active Project factor is to check whether the API project is closed or not and is there any support available for this API? And we can see that JOMP is not active project. JOMP, DPJ and JaMP also do not support syntax checking at design time. Only JOMP and JAMP are the APIs whose code can run without using them sequentially. All other APIs code is useless if we do not have that API. Also JOMP, JAMP and JConqurr are the APIs which are easy to use. We do not have to write a lot of code to make Java programs parallel.

2.7 Conclusion

From our comparisons of the APIs we have selected JaMP for our thesis. Main factors we have considered are easy to use, source code available and existing code can run (sequentially) without using this API. In the next chapter we will discuss in detail about JaMP and its OpenMP directives.
3 JaMP

JaMP is a Java implementation of OpenMP. It consists of an OpenMP compiler and a runtime library. The compiler uses the 'JavaCC' tool and it translates JaMP program to byte code class files. The runtime library has static functions that can be used to control and query the parallel execution environment [18].

3.1 JaMP Directives

JaMP directives are the same as OpenMP directives, and anyone who knows the OpenMP directives can use JaMP directives as well [24]. This section first describes the JaMP directives format and then it describes some important JaMP directives. Most of the contents are taken from [7] and [22].

3.1.1 Directives Format

JaMP directives are not an extension to the Java language instead these are special Java inline comments for OpenMP directives. The JaMP compiler recognizes them and treats them as OpenMP directives but any other Java compiler will treat them as comments and ignore them. This technique has been used by the JOMP compiler too [21].

The format of a JaMP directive is:

```
//#omp <directive> <[clause[[,] clause] ...]>
```

Each JaMP directive starts with ‘//#omp’, making it easy for the JaMP compiler to recognize that this is a JaMP directive. Clauses are optional attributes of a directive.

Directives are case sensitive. The order of clauses in a directive is not significant. Each directive should end with a new line [24].

3.1.2 Parallel Directive

The parallel directive is the directive which starts parallel execution in a program. The format of the `parallel` directive is:

```
//#omp parallel [clause[[,] clause] ...]
```

The ‘parallel’ directive creates a parallel region. The parallel region is a region which will be will be executed concurrently by multiple threads. When a thread reaches a parallel region it creates a new team of threads and becomes itself master thread of that team. At the end of each parallel region there is an implicit barrier. All the threads execute the region, and after the barrier the master thread continues [24].

Some important clauses of parallel directive are

- **if (scalar-expression):** If this clause is present then before executing parallel region scalar expression will be evaluated. If scalar expression evaluates to non-zero value then parallel region will be created. If scalar expression evaluates to zero then all the statements will run sequentially.

- **num_threads(integer_expression):** If ‘num_threads’ clause is present then number of threads for the parallel region will be equal to the value of integer expression.

**Notice:** JaMP (like OpenMP) are using non-zero values to denote ‘true’, and zero to denote ‘false’. Thus, they are not using the primitive type ‘boolean’ to represent boolean values.
3.1.3 Sections and Section Directives

‘Sections’ directive is used to define the code sections which can execute concurrently in the ‘parallel’ directive. Each defined section will execute once by a thread of the team. The format is:
```
//omp parallel
{
   //omp sections
   {
      //omp section
      <statement block>
      //omp section
      <statement block>
   }
}
```

The ‘section’ directive can only come into ‘sections’ directive and ‘sections’ directive can only be in the ‘parallel’ directive.

3.1.4 For Directive

The ‘for’ directive divide the loop iterations between each threads of the team to execute them concurrently. The syntax of loop is restricted so that OpenMP can determine iterations before loop can execute. Important syntax restrictions are; loop variable type should be only of integer type and only relational operators (‘<’, ‘<=’, ‘>’, ‘>=’) are allowed. Also like ‘sections’ directive ‘for’ directive must be defined in the ‘parallel’ directive. The format of the ‘for’ directive is:
```
//omp parallel
{
    //omp for [clause[.] clause] ...
    for (int i = 0; i < N; i++)
    {
        // some code here
    }
}
```

Like ‘parallel’ directive, ‘for’ directive also inserts implicit barrier at end of the loop. For directive have some important clauses.
- **nowait**: This clause removes the implicit barrier at the end of loop.
- **reduction**: Reduction class has the following format ‘reduction(op, var-list)’
  It performs the reduction on variables defined in the comma separated variable list with the operator ‘op’. Operators should not be overloaded and they can be +, *, -, &, ^, |, &&, or || operator.
- **ordered**: For using the ordered directive (see 3.1.5) in the loop ordered clause should be defined in the ‘for’ directive.

3.1.5 Ordered Directive

Ordered directive can be used on the statements which need sequential execution of the ‘for’ loop. It has following format
```
//omp ordered
<statement block>
```
To use this directive, ‘ordered’ clause must be present in the ‘for’ directive.

### 3.1.6 Barrier Directive

The ‘barrier’ directive is used to synchronize all the threads of a team. All the threads of a team wait for each other at barrier point until all reach there. When all reach at barrier point they start again execution after that point. The format of ’barrier’ directive is:

```c
//#omp barrier
```

### 3.1.7 Master Directive

The ‘master’ directive put restriction on its statement block that only master directive of the team can execute that block. Other threads of the team will ignore its statement block. The format is:

```c
//#omp master
<statement block>
```

### 3.1.8 Critical Directive

The statement block attached with the ‘critical’ directive can only be executed by one thread of the team at a time. The format is:

```c
//#omp critical
<statement block>
```

### 3.1.9 Combined Parallel Work-sharing Directives

We can combine parallel and work-sharing directives. Work-sharing directives are ‘sections’ and ‘for’ directive. This is a shortcut to define a ‘parallel’ directive with only one work-sharing directive. Clauses can be from both parallel and sections directives except ‘nowait’ clause.

- **Parallel Sections Directive**

  The format of combining sections directive with ‘parallel’ directive is:

  ```c
  //#omp parallel sections [clause[[.] clause] ...]
  {
      [//#omp section]
      <statement block>
      [//#omp section
      <statement block>]
      ...
  }
  ```

- **Parallel For Directive**

  This is the shortcut to create a parallel directive with only one ‘for’ directive.

  The format is:

  ```c
  //#omp parallel for [clause[[.] clause] ...]
  for-loop
  ```

### 3.2 Runtime Library

JaMP runtime library has an ‘omp.Omp’ class which has a number of static methods. These functions can be used to control the parallel execution environment. The important ones are:
3.2.1 omp_set_num_threads(int num_threads) Function
If this function has been called then all the subsequent parallel regions will have same number of threads which are set by this function. This function will not be applied on the parallel regions which have ‘num_threads’ clause.

3.2.2 omp_get_num_threads Function
This function returns the number of threads allocated to a parallel region. If this function has been called from serial portion of the program then it will return one.

3.2.3 omp_in_parallel Function
This function returns the non-zero value if it is called from a parallel region else it returns zero.

3.2.4 omp_set_dynamic(int dynamic_threads) Function
It enables or disables the dynamic adjustment of number of threads for allocating to parallel regions.

3.2.5 omp_set_nested(int nested) Function
This function enables or disables the nested parallelism. If nested parallelism is disabled then nested parallel regions will execute serialize by a single thread, and if it is enabled then a new team of threads will be created for nested parallel region.

3.3 Example of JaMP program
The following is an example JaMP program for naïve matrix multiplication.

```java
import java.util.Random;
public class MatrixMultiplicationNaive {
    public static void main(String[] args) {
        int size = 1500;
        omp.Omp.omp_set_num_threads(4);

        int[][] array = new int[size][size];
        int[][] array1 = new int[size][size];
        int[][] array2 = new int[size][size];

        Random random = new Random();

        System.out.println("Starting");
        long start = System.currentTimeMillis();

        // #omp parallel
        {
            // #omp for nowait
            for (int i = 0; (i < size); i++) {
                for (int j = 0; (j < size); j++) {
                    array[i][j] = random.nextInt();
                }
            }
        }
    }
}
```
```java
//#omp for nowait
for (int i = 0; (i < size); i++) {
    for (int j = 0; (j < size); j++) {
        array1[i][j] = random.nextInt();
    }
}
}
} // implicit barrier here

//#omp parallel for
for (int i = 0; (i < size); i++) {
    for (int j = 0; (j < size); j++) {
        for (int k = 0; (k < size); k++) {
            array2[i][j] += (array[i][k] * array1[k][j]);
        }
    }
}
}

long stop = System.currentTimeMillis();
System.out.println("Total Seconds: "+ ((double) (stop - start) / 1000));
}
}
```

Example 3-1 JaMP naïve Matrix multiplication program

This program creates two arrays and fills them with random integer values. The JaMP directives which are used are shown bold in the program. By looking at the program we can see that we do not have to change a lot of things in the program to make it a JaMP parallel program. Only few directives are added and it is now ready to run as parallel. The `omp.Omp.omp_set_num_threads(4)` is a function which can be used to set the number of threads for executing the program. Currently, it will set four threads for all the parallel regions. It implies that until explicitly specified again, all parallel regions will have a team of four threads which will divide the work.
4 Contribution

In this chapter first we are discussing problems in JaMP and then our contribution for solving these problems.

4.1 Issues in JaMP

1. JaMP API comes in a Jar file. Whenever we compile our JaMP source code, we have to use shell commands.
2. There is no design time support for JaMP directives to check that they are written correctly or not. JaMP directives will be checked only when we compile our all source code with the JaMP compiler. For big projects it will take time as we have to find the errors in the source files and then correct them. And it is a repetitive process until all the files are compiled successfully.
3. Another issue is that JaMP just generate byte code class files and the problem is that we cannot debug it. If some code after becoming parallel not working correctly, then we cannot know where the actual problem is.

4.2 Our Contribution

For solving above mentioned problems, our contribution is

1. We create an Eclipse IDE plugin named ‘JaMPBuilder’ to simplify the development of concurrent Java programs.
2. Our plugin is using JaMP compiler and can detect syntax errors of JaMP directives at design time.
3. We add an option into the JaMP compiler to generate parallel source code. We add an argument ‘-s destination-parallel-source-path’ to JaMP compiler. So when we compile with commands we can set the destination path and JaMP will create the parallel source files there.

For using the plugin, we just have to copy the plug-in jar file to the ‘plugins’ folder of Eclipse. After copying the plugin when we start the eclipse we will have “JaMP Manu” with two options; ‘Add/Remove JaMP Runtime’ and ‘Get Parallel Source’ as shown in figure 4-1.
Both options are only enabled when a Java project is selected from Project. When option ‘Add/Remove JaMP Runtime’ is clicked first time on any selected Java project, it creates a lib folder if it is not exists in selected project and copy the JaMP runtime there. After that it adds reference of copied runtime file to the ‘Build Path’ of selected project. Then it adds a JaMP project nature to the selected project.

The project nature is an association between a project and a plugin. It indicates that plugin is configured to use that project. Like by default Java projects have Java nature, which indicates that Java Development Tool (JDT) plugins are aware of it [25].

The JaMP nature adds a JaMP project builder for the selected project. The project builders are used to compile the source files [25]. The JaMP project builder is used to detect the design time errors of JaMP directives with the help of JaMP compiler. Figure 4-2 shows a design time error is detected by plug-in.
The ‘Add/Remove JaMP Runtime’ menu button works like a toggle button. If already JaMP nature is added to project on the next click it removes JaMP nature. When it is removed design time errors will not be checked.

We use the creating parallel source files argument in the plug-in in ‘Get Parallel Source’ button. When we select a project from solution explorer and then click on this button, first time it prompts for a project name to store the parallel source files there. After that it creates the new project with the name given and it copies the parallel source files there. New project will be the copy of previous project it will have all the references of Projects and Jar files in Build Path.

Let us take a simple JaMP program which is using ‘parallel for’ directive as shown in example 4-1.

```java
public class ParallelForexample {
    public static void main(String[] args) {
        //omp parallel for
        for (int i = 0; i < 100; i++) {
            <Statement-block>
        }
    }
}
```

Example 4-1 Simple JaMP program using one ‘parallel for’ directive
In the ‘for’ loop ‘Statement-block’ is just any block of code to execute by for loop. The code generated for above program is shown here.

1. import omp.*;
2. import de.fau.cs.i2.jamp.runtime.*;
3. public class ParallelForexample {
4.   public class __jampRunnable_1 extends JampRunnable {
5.     public JampRunnable getInstance() {
6.         return new __jampRunnable_1();
7.     }
8.   }
9. @Override
10. public void run() {
11.   // For Beginning:
12.   if ((JampRuntime.enterSingleRegion(1) == true)) {
13.     // For Init Beginning:
14.     // Requirements:
15.     LoopInformation.loadLoopInformation(0, 0, 100, 1, 2, (-1), 0, null);
16.     // For Init End:
17.   }
18.   LoopInformation __jamp_3 =
19.       JampRuntime.getTeam().getCurrentLoopInfo(0);
20.   LoopSlice __jamp_2 = __jamp_3.getLoopSlice();
21.   Chunk __jamp_4 = __jamp_2.getNextChunk();
22.   while ((__jamp_4 != null)) {
23.     for (int i = __jamp_4.initialization; (i < __jamp_4.condition); i += __jamp_4.increment) {
24.       // Code Beginning:
25.     }
26.     // Code End:
27.   }
28.   __jamp_4 = __jamp_2.getNextChunk();
29.   // For End:
30. }
31. }
32. public static void main(String[] args) {
33.   int __jamp_1 =
34.       de.fau.cs.i2.jamp.runtime.JampRuntime.getDefaultTeamSize();
35.   // Parallel Beginning:
36. }
37. // shared
38. // numthreads:
45.     // Requirements:
46.     // Parallel Info: numthreads(__jamp_1) shared()
47.     // Method:
48.     // Runnable Beginn
49.     ParallelForexample __jamp_7 = new ParallelForexample();
50.     __jampRunnable_1 __jamp_6 = __jamp_7.new __jampRunnable_1();
51.     // Runnable:
52.     // Runnable End
53.     JampThreadTeam __jamp_5 = JampThreadTeam.getNewTeamInstance(
54.     __jamp_1, 0, __jamp_6);
55.     __jamp_5.startAndJoin();
56.     // Parallel End:
57.     }
58. }
59. }
Example 4-2 Parallel source code of simple JaMP program using one ‘parallel for’ directive

The ‘JampThreadTeam’ is the class which is used to create the team of threads. The static method at line 53 ‘getNewTeamInstance’ has 3 parameters i.e. number of threads, team id, and runnable to execute by threads. At line 55 ‘startAndJoin’ method, starts the execution and put implicit barrier at the end. The ‘JampRunnable’ is the abstract class which extends the ‘Runnable’ interface. The code in line 14-20 will execute only once by thread that comes first and it will load the loop information to the runtime environment e.g. loop id, initial value, conditional value, increment value etc. After that from loop information, loop iterations will be divided among threads in chunks. Lines 26-30 are the lines where original loop’s statement block is copied by compiler to execute.

For more information about parallel source code of ‘parallel sections’ directive see Appendix A.
5 Evaluation

We evaluate the JaMP API on matrix multiplication and merge sort experiments. Both have been done on a workstation which has 8 Intel Xeon 2.33 GHz processors.

5.1 Matrix Multiplication Experiment

The matrix multiplication experiment is taken from the course 4DV103 Software Technology Project. The experiment has three parts. First one is about general matrix performance. It includes matrix creation, traverse/access in the matrix, and applying plus, minus, null operations on matrices. It also includes getting matrix transpose, copying matrix, checking equality between matrices, checking that whether matrix is diagonal or not and whether it is symmetric or not. Second part use naïve matrix multiplication on dense matrices. And third part is applying a recursive matrix multiplication algorithm on dense matrices. The output of the non-parallel version of this experiment is shown in figure 5-1.

Problem Size: 100
Implementation: baseline.SquareMatrix
Heap Memory: 1860 Mbytes

<table>
<thead>
<tr>
<th>General Matrix Performance</th>
<th>Matrix Sizes: 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Creation ...</td>
<td>502 MB, 3.021s</td>
</tr>
<tr>
<td>Traverse/Access ...</td>
<td>459 MB, 4.046s</td>
</tr>
<tr>
<td>Misc. Operators ...</td>
<td>396 MB, 11.183s</td>
</tr>
<tr>
<td>Plus, Minus, NULL, ...</td>
<td>855 MB, 11.979s</td>
</tr>
</tbody>
</table>

Time: 30.229

Multiply Basic
Random Dense Matrices: SZ = 500
Building Matrices: NULL 1 2 3 4 5 ONE OK!
Runs: 0 1 2 3 4 5
Mem: 114 MB
Time: 64.076

Multiply Recursive
Random Dense Matrices: SZ = 500
Building Matrices: NULL 1 2 3 4 5 ONE OK!
Runs: 0 1 2 3 4 5
Mem: 121 MB
Time: 27.555

Total Time: 121.85999999999999

Figure 5-1 The output of non-parallel version of the experiment

To compare with JaMP parallel version of experiment, first we decided to manually parallelize multiplication. Note that the parallel version is not the optimal one but we make it parallel to compare with JaMP version later. The manual parallel version is using two native threads. All the ‘for’ loops of matrix functions has been divided into two threads. If a loop has inner ‘for’ loops only outer loop has been divided. Example 5-2 shows that how a ‘for’ loop is divided between 2 threads.

Suppose we has a ‘for’ loop as shown in example 5-1.

```
for (int i = 0; i < N; i++){
    // some code here
}
```

Example 5-1 A simple ‘for’ loop

We can divide its iterations between two threads as:
// Create a new thread
Thread thread = new Thread(new Runnable(){
    void run(){
        // divide the loop to half
        for (int i = 0; i < N/2; i++){
            // some code here
        }
    }
});

// Start the newly created thread
thread.start();

// continue with the main thread
for (int i = N/2; i < N; i++){
    // some code here
}

try{
    // wait for thread to finish its execution
    thread.join();
}
catch(Exception e){
}

Example 5-2 Two threaded version of ‘for’ loop

The iterations of ‘for’ loop are divided into two half sized ‘for’ loops. One ‘for’ loop will execute by a new thread and other will execute by the main thread. When main thread will execute its loop, it will wait for newly created thread to finish its execution. After that main thread will continue its execution.

For recursive multiplication we parallelized the merging part of recursive algorithm. The merging part has a ‘for’ loop which has been divided as described above. The output of manually parallel is shown in figure 5-2.
After comparing both non-parallel and manual parallel versions of experiment we can see that general matrix performance is almost equal to its parallel version. The general part consists of calling to a lot of small methods. And these methods are small enough so that they do not give much speed up in their parallel versions.

But manual parallelization is giving advantage in naïve matrix multiplication. We divided naïve matrix multiplication into two threads and it takes almost half the time compared to non-parallel version. It shows that naïve matrix multiplication method is good candidate for parallelization.

For recursive matrix multiplication, parallel version does not take advantage of parallelization and it is slower than non-parallel version of experiment. The reason is, thread creation takes time and in each recursive call it creates a new thread. And due to a lot of recursive calls, overall time of recursive multiplication increased.

Creating manual versions of experiment for four or eight threads could be a time consuming and error prone task. We have to divide loop iterations exactly between different threads. This can be done easily with JaMP. So table 5-1 is using JaMP and shows the results of experiments with two, four and eight threads versions of experiment.

For JaMP parallel version of experiment all the outer ‘for’ loops which were earlier parallelized for manual parallel version has been changed to use ‘parallel for’ directive of JaMP. Only ‘omp.Omp.omp_set_num_threads(N)’ function has been used to set the number of threads for an experiment. No other functions like ‘omp.Omp.omp_set_nested()’ has been used. They are using their default values, like for ‘omp.Omp.omp_set_nested()’ function the default value is false.

Table 5-1 summarizes the JaMP experiment results.

<table>
<thead>
<tr>
<th>No. of threads</th>
<th>General</th>
<th>Multiply Naïve</th>
<th>Multiply Recursive</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>28</td>
<td>32</td>
<td>79</td>
<td>140</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
<td>17</td>
<td>96</td>
<td>139</td>
</tr>
<tr>
<td>8</td>
<td>28</td>
<td>12</td>
<td>113</td>
<td>153</td>
</tr>
</tbody>
</table>

Table 5-1 The JaMP matrix multiplication experiment. All times are given in seconds.

From the results of table 5-1 we can see that results are almost same like manual parallel version of experiment. The general matrix performance is like manual parallel version, almost equal to its sequential version despite of the number of threads has been used.

The naïve multiplication is same as manual parallel version of experiment too. But we can see that there is not much difference between times of 4 and 8 threads. It is just proof of Amdahl’s law. Because there will be code which is not parallel and it will take its full time regardless of number of processors devoted for the program.

The recursive multiplication times are also same as manual parallel version of experiment. They are taking more time than sequential version of experiment. And it can be the case that recursive matrix multiplication is not good candidate for parallelization. As we already discussed that thread creation takes time so the time in recursive multiplication increased exponentially with number of threads. But we can see that when number of threads is equal to two, JaMP version is taking more time than manual parallel version of experiment. It can be a result of complex algorithm or bug in JaMP.
5.2 Merge Sort Experiments

Merge sort is a divide and conquer sorting technique in which it divides unsorted list into two almost equal sized sub lists and sort them recursively. After sorting sub lists it merges both lists together to make a sorted list [33].

The example 5-3 shows a parallel merge sort algorithm. It can sort each sub list recursively into a new thread and then it merges the lists together by the thread that has started the method. An example of parallel merge sort algorithm is:

\[
\text{Parallel-Merge-Sort}(A, p, r): // \text{sort } A[p...r]
\]
\[
\text{if } p < r:
\]
\[
q = \text{floor}((p+r)/2)
\]
\[
\text{spawn Parallel-Merge-Sort}(A, p, q)
\]
\[
\text{spawn Parallel-Merge-Sort}(A, q+1, r)
\]
\[
\text{sync}
\]
\[
\text{Merge}(A, p, q, r) // \text{merge } A[p...q] \text{ with } A[q+1...r]
\]

Example 5-3 A parallel merge sort algorithm [34]

The word ‘spawn’ means to call the ‘Parallel-Merge-Sort’ method into a new thread and ‘sync’ means to wait for both threads to finish their execution.

We have done three types of merge sort experiment. First two experiments are using manual thread creation techniques and the third one is using JaMP. First experiment is using specified number of threads. We call it N-threaded experiment. It is using 1, 2 and 8 threaded version of merge sort.

For N-threaded solution we have done a little change in the parallel merge sort as shown in example 5-4.

\[
\text{Parallel-Merge-Sort}(A, p, r, N): // \text{sort } A[p...r], N = \text{number of threads}
\]
\[
\text{if } p < r:
\]
\[
q = \text{floor}((p+r)/2)
\]
\[
\text{if } (N > 1):
\]
\[
\text{spawn Parallel-Merge-Sort}(A, p, q, N - 1) // \text{Create a new thread}
\]
\[
\text{//do not create a new thread instead use the thread who initiates the method call}
\]
\[
\text{Parallel-Merge-Sort}(A, q+1, r, N - 1)
\]
\[
\text{Sync} // \text{wait for new thread to finish its execution}
\]
\[
\text{else:}
\]
\[
\text{// go sequential}
\]
\[
\text{Parallel-Merge-Sort}(A, p, q, N)
\]
\[
\text{Parallel-Merge-Sort}(A, q+1, r, N)
\]
\[
\text{Merge}(A, p, q, r) // \text{merge } A[p...q] \text{ with } A[q+1...r]
\]

Example 5-4 A N-threaded parallel merge sort algorithm

The algorithm creates number of threads specified by ‘N’ parameter. There is slightly modification in parallel part of method compare to example 5-3. It creates only 1 thread in each recursive call for one sub list and other sub list is sorted by the thread that is main thread for the method. Table 5-2 shows the result of N-threaded experiment with 1, 2 and 8 (number of threads equal to number of available processors) threaded versions for merge sort.
<table>
<thead>
<tr>
<th>Size</th>
<th>N = 1</th>
<th>N = 2</th>
<th>N = 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>0.027</td>
<td>0.0252</td>
<td>0.0356</td>
</tr>
<tr>
<td>20,000</td>
<td>0.0628</td>
<td>0.0476</td>
<td>0.0606</td>
</tr>
<tr>
<td>30,000</td>
<td>0.1452</td>
<td>0.1108</td>
<td>0.119</td>
</tr>
<tr>
<td>40,000</td>
<td>0.2606</td>
<td>0.2004</td>
<td>0.201</td>
</tr>
<tr>
<td>50,000</td>
<td>0.4118</td>
<td>0.314</td>
<td>0.3068</td>
</tr>
<tr>
<td>60,000</td>
<td>0.596</td>
<td>0.4556</td>
<td>0.4334</td>
</tr>
<tr>
<td>70,000</td>
<td>0.815</td>
<td>0.621</td>
<td>0.5876</td>
</tr>
<tr>
<td>80,000</td>
<td>1.0714</td>
<td>0.813</td>
<td>0.7654</td>
</tr>
<tr>
<td>90,000</td>
<td>1.3614</td>
<td>1.0318</td>
<td>0.9672</td>
</tr>
<tr>
<td>100,000</td>
<td>1.6842</td>
<td>1.2796</td>
<td>1.1864</td>
</tr>
</tbody>
</table>

Table 5-2 Fixed threads merge sort experiment. All times are given in seconds.

From the first experiment we can see that 2-threaded version of merge sort is faster than 1-threaded version. But for comparison between 2 and 8-threaded versions, in the start when list size was small, 8-threaded version was slower. It was slower than 1-threaded version too for the first row of experiment. But later 8-threaded version becomes faster than all. So its mean that for small list size, creating the 8 threads takes too much time that, other versions become faster. But for bigger list size, an 8-threaded version is so fast that it also covers its thread creation time.

Our second experiment is using an algorithm which always creates new threads when recursive call has been made. Algorithm is same as example 5-3 but with a slight change that it creates only a single thread in each recursive call as shown in example 5-5.

Parallel-Merge-Sort(A, p, r): // sort A[p...r]
if p < r:
    q = floor((p+r)/2)
    spawn Parallel-Merge-Sort(A, p, q, N - 1) // Create a new thread
    //do not create a new thread instead use the thread who initiates the method call
    Parallel-Merge-Sort(A, q+1, r, N - 1)
Sync // wait for new thread to finish its execution

Merge(A, p, q, r) // merge A[p...q] with A[q+1...r]
Example 5-5 A parallel merge sort algorithm with creating a new thread on each recursive call

The experiment is using the techniques discussed in section 2.4; native threads creation and using a thread pool. This experiment will help us to understand which thread creation and management technique is better. The results are:

<table>
<thead>
<tr>
<th>Size</th>
<th>Native Threads</th>
<th>Thread Pool</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>1.9915</td>
<td>0.2918</td>
</tr>
<tr>
<td>20,000</td>
<td>6.39283333</td>
<td>0.5404</td>
</tr>
<tr>
<td>30,000</td>
<td>12.8201667</td>
<td>0.9524</td>
</tr>
<tr>
<td>40,000</td>
<td>26.4318333</td>
<td>1.2938</td>
</tr>
<tr>
<td>50,000</td>
<td>45.1093333</td>
<td>1.2612</td>
</tr>
<tr>
<td>60,000</td>
<td>56.1676667</td>
<td>2.0222</td>
</tr>
<tr>
<td>70,000</td>
<td>72.5346667</td>
<td>1.9944</td>
</tr>
</tbody>
</table>
Table 5-3 Parallel merge sort algorithm experiment using Java threading techniques. All times are given in seconds.

For experiment we have used a thread pool with fixed size of 8 threads. From the table 5-3, it is obvious that thread pool is as better technique than native threads. The reason is that, thread pool creates a fixed size collection of threads (i.e. 8) and a work queue. Whenever a work has been submitted thread pool stores it in work queue. The collections of threads looks for the work in work queue and process it. So thread pool does not create more threads and manage the existing threads. That is why thread pool is faster.

On the other hand, if we compare thread pool version of merge sort with N-threaded version we can see that thread pool is slow. Thread pool is using fixed size of threads like N-threaded version. But major difference is thread pool always create ‘Runnable’ object and add it to work queue in each recursive call which makes it slow.

From first two experiments we can see that 8-threaded and thread pool solutions are best among all others. The third experiment is using the JaMP on parallel merge sort algorithm. It is using nested parallelism technique by setting it enabled and disabled for algorithm. If nested parallelism is disabled then JaMP after first recursive call does not create the new threads and results should be similar to 2-threaded version of merge sort results in first experiment. Similarly, if nested parallelism is enabled then JaMP should create the new threads on each recursive call and results should be similar to native threads version of merge sort in second experiment. The nested parallelism has been discussed more in section 3.2.5. The results are:

<table>
<thead>
<tr>
<th>Size</th>
<th>Nested Parallelism Enabled</th>
<th>Nested Parallelism Disabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>2520.8565</td>
<td>0.0452</td>
</tr>
<tr>
<td>20,000</td>
<td>1478.3045</td>
<td>0.0756</td>
</tr>
<tr>
<td>30,000</td>
<td>2729.715</td>
<td>0.1506</td>
</tr>
<tr>
<td>40,000</td>
<td>2605.252</td>
<td>0.25</td>
</tr>
<tr>
<td>50,000</td>
<td>3046.898</td>
<td>0.3794</td>
</tr>
<tr>
<td>60,000</td>
<td>3482.713</td>
<td>0.5304</td>
</tr>
<tr>
<td>70,000</td>
<td>7625.325</td>
<td>0.7106</td>
</tr>
<tr>
<td>80,000</td>
<td>6276.577</td>
<td>0.9134</td>
</tr>
<tr>
<td>90,000</td>
<td>6472.3675</td>
<td>1.1464</td>
</tr>
<tr>
<td>100,000</td>
<td>6903.1335</td>
<td>1.4006</td>
</tr>
</tbody>
</table>

Table 5-4 The merge sort algorithm experiment using JaMP. All times are given in seconds.

In the third experiment when nested parallelism is disabled, its results are same as of 2-threaded version of first experiment. But we can see that JaMP solution with nested parallelism enabled is much worse compare to the second experiment results using native threads and thread pool. It shows that JaMP is not properly handling the nested parallelism.

Our suggested solution for improving JaMP is to use thread pools for nested parallelism enabled and N-threaded solution (number of threads should be equal to available processors) for nested parallelism disabled problems.
5.3 Conclusion
This result shows that it is not always the case that parallel code will execute faster. An intelligent algorithm is needed which can determine that serialize code will execute faster or parallel with taking care of number of threads in count.
6 Summary and Future Work

In this chapter we are discussing our conclusion about the thesis. Furthermore we are discussing about future improvements which can be done later.

6.1 Summary

Parallelization is becoming the future of computing. Technology trend is moving from single core processor to multi-core and many core processors. However software technology is currently behind hardware parallelization support. Many major languages like Java do not give much support for implementation of parallel programs. Today it is both time consuming and error prone. Applications can speed up if we define independent tasks which can run concurrently on each separate core.

OpenMP is a standard parallelization API for C++ and FORTRAN. It has set of directives which help to define independent tasks in programs. JaMP is the Java implementation of OpenMP. Using JaMP directives we can easily define independent task in a Java program. JaMP directives are having format of inline comments. This can be ignored by ordinary Java compilers. JaMP consists of a compiler which converts the JaMP programs into byte code class files. There is no design time support for writing JaMP programs.

We have made an Eclipse IDE plugin which can detect design time errors. Also we add an option in JaMP compiler to get the parallel source which can be used for debugging. In the end we have evaluated JaMP with Matrix Multiplication and merge sort experiments. We have found that it is not always the case that parallel code will execute faster. An intelligent algorithm is needed which can determine that serialize code will execute faster or parallel with taking care of number of threads in count.

6.2 Future Work

There is a lot of works which can be done to improve our Eclipse plugin, e.g.

- There is need to improve the handling of recursive calls. JaMP should adopt to use ‘ThreadPool’ that was introduced in JDK 1.6.
- There should be a directive for for-each loop to run it parallel.
- JaMP does not support full 3.0 specification of OpenMP. It should handle it.
- Code-assistant feature should be available in the plug-in which can help programmers to write JaMP directives easily.
7 References

[1]. Introduction to Parallel Computing, Blaise Barney, 
[2]. Intel, Microsoft: The Future of Computing Is Parallel, Richard Adhikari, 
[3]. Symmetric multiprocessing, 
[5]. Parallel Java: A unified API for shared memory and cluster parallel 
programming in 100% Java., Alan Kaminsky, 21st IEEE International Parallel 
and Distributed Processing Symposium (IPDPS 2007), March 2007
[6]. JOMP home page, 
http://www2.epcc.ed.ac.uk/computing/research_activities/jomp/index_1.html
[7]. JOMP—an OpenMP-like Interface for Java, J. M. Bul, M. E. Kambite, ACM 
2000
[8]. JOMP Application Program Interface Version 0.1 (draft), Jan Obdrzalek, Mark 
Bull 
http://www2.epcc.ed.ac.uk/computing/research_activities/jomp/papers/jompAPI.pdf, reviewed May 11, 2011
[10]. Parallel Java: A Unified API for Shared Memory and Cluster Parallel 
Programming in 100% Java, Alan Kaminsky, Presented at the Rochester 
[12]. The Deterministic Parallel Java Tutorial Version 1.0, 
[13]. The Deterministic Parallel Java Installation Manual Version 1.0, 
[14]. The Deterministic Parallel Java Language Reference Manual Version 1.0, 
[16]. JConqurr - A Multi-Core Programming Toolkit for Java, G.A.C.P. Ganegoda, 
D.M.A. Samaranayake, L.S. Bandara, K.A.D.N.K. Wimalawarne, International 
Journal of Computer and Information Engineering 3:4 2009
[18]. JaMP home page, http://www2.informatik.uni- 
erlangen.de/EN/research/JavaOpenMP/index.html
[21]. A Proposal for OpenMP for Java, Klemm M., Veldema R., Bezold, 
540-68554-5
[22]. OpenMP C and C++ Application Program Interface Version 2.0, March 2002
[23]. OpenMP Fortran Application Program Interface Version 2.0, March 2002
[24]. JaMP: An Implementation of OpenMP for a Java DSM. Klemm M. Bezold 
M. Veldema R. Philippens M. In: Concurrency and Computation: Practice and 
[25]. Project Builders and Natures, John Arthorne, IBM OTI Labs, reviewed 15 May, 2011
Appendix A

A simple example of a JaMP program with 'parallel sections' directive is shown in example A-1.

```java
class ParallelSectionsExample {
    public static void main(String[] args) {
        //@omp parallel sections
        //@omp section
        <Section1-Statement-block>
        //@omp section
        <Section2-Statement-block>
    }
}
```

Example A-1 A simple JaMP program with 'parallel sections' directive

The statement blocks following section directives are any statement block to be execute by sections independently. The parallel source code of above program is shown here.

```java
1. import omp.*;
2. import de.fau.cs.i2.jamp.runtime.*;
3.
4. public class ParallelSectionsExample {
5.     public class __jampRunnable_4 extends JampRunnable {
6.         public JampRunnable getInstance() {
7.             return new __jampRunnable_4();
8.         }
9.     }
10. }
11. @Override
12. public void run() {
13.     // For Beginning:
14.     if ((JampRuntime.enterSingleRegion(5) == true)) {
15.         // For Init Beginning:
16.         // Requirements:
17.         LoopInformation.loadLoopInformation(4, 0, 2, 1, 2, 1, 0, null);
18.         // For Init End:
19.     }
20.     
21.     LoopInformation __jamp_29 = JampRuntime.getTeam().getCurrentLoopInfo(4);
22.     LoopSlice __jamp_28 = __jamp_29.getLoopSlice();
23.     Chunk __jamp_30 = __jamp_28getNextChunk();
24.     while ((__jamp_30 != null)) {
25.         for (int __jamp_26 = __jamp_30.initialization;
26.             __jamp_26 < __jamp_30.condition; __jamp_26 += __jamp_30.increment) {
27.             // Code Beginning:
28.         }
29.     }
30. }
```
// __jamp_26:4:1
switch (__jamp_26) {
    case 0:
    ;
    <Section1-Statement-block>
    break;
    case 1:
    ;
    <Section2-Statement-block>
    break;
}
// Code End:
__jamp_30 = __jamp_28.getNextChunk();
}
// For End:
}
public static void main(String[] args) {
    int __jamp_27 = de.fau.cs.i2.jamp.runtime.JampRuntime
    .getDefaultTeamSize();
    // Parallel Beginning:
    // shared
    // numthreads: __jamp_27:1:1:__jamp_27:
    // Requirements:
    // Parallel Info: numthreads(__jamp_27) shared()
    // Method:
    // Runnable Beginn
    ParallelSectionsExample __jamp_33 = new ParallelSectionsExample();
    __jampRunnable_4 __jamp_32 = __jamp_33.new __jampRunnable_4();
    // Runnable:
    // Runnable End
    JampThreadTeam __jamp_31 =
    JampThreadTeam.getNewTeamInstance(__jamp_27, 0, __jamp_32);
    __jamp_31.startAndJoin();
    // Parallel End:
    }
}
Example 0A-2 Parallel source code of simple JaMP program with ‘parallel sections’ directive

The code is same as we saw earlier for ‘parallel for’ directive. At line 40, loop information is stored in the runtime environment with 2 termination condition. And in line 30-40 section selection is dependent on the chunk value.