An Efficient Multicore Programming Toolkit for Java

G.A.C.P. Ganegoda, D.M.A. Samaranayake, L.S. Bandara and K.A.D.N.K. Wimalawarne
Department of Computer Science and Engineering, University of Moratuwa, Sri Lanka

Abstract— **With the popularity of the multi-core architectures there is a requirement for the toolsets which supports parallel programming methodologies. In addition to that there is a need for parallelization of existing applications according to the underlying multi-core architectures. In this paper we discuss about a toolkit JConqurr which provides a solution for the above problems.**
JConcurr is a multi-core programming toolkit for Java which is capable of providing support for main parallel programming patterns, which includes task, data, divide and conquer and pipeline parallelism. Toolkit uses an annotation and directive mechanism to convert the sequential application into a parallel one. In addition to that we have proposed a novel mechanism to achieve the parallelism using the graphical processing unit.

Keywords— **multi-core, parallel programming, GPU, Java, Eclipse**

I. Introduction

In the advent of multi core processors, importance of parallel computing is much significant in modern computing era and the performance gain of the applications will mainly depend on the maximum utilization across the cores existing in the system. It is necessary that tools should exist to make the full use of the capabilities offered by the parallel computing. Even though current parallelizing compilers supports only parallelization at loop level [1] it's a hard task to detect parallel patterns and do the conversion at the compiler level.

Already there exists languages and libraries like OpenMP, Cilk++,JCilk, TBB for C++ [2,3,4,5]language to support the multi-core programming, but there is a lack in the tools for Java language. We came a across libraries like JIBU, JOMP and a extended language called XJava [6,7,8]. for Java language. Within those there are only few libraries which are capable of providing the data, task, divide and conquer, pipeline parallelism patterns [9] in a one tool.

In this paper we discuss about a multi-core programming toolkit for java language which uses an annotation and a novel directive mechanism to provide meta information of the source code. PAL [22] is one of the tools which use annotations mechanism to achieve parallelism targeting cluster networks of workstations and grids. PAL uses a mechanism with the help of Java Jini Parallel Framework. Even though we use annotations, compared to it we use a different procedure. With the use of meta information we are able to provide a source level parallelism instead of the compiler level parallelism. Our toolkit comes as a plug-in for eclipse IDE[10]. Toolkit is capable of converting the sequential java project into a new project which optimised based on the parallel patterns, using the meta information passed by the user. In addition to this we have proposed a mechanism to achieve parallelism using the graphical processing unit using the JCUDA [11] binding. With that our toolkit can be used for both CPU and GPU parallelism.

In next sections of the paper is organized as follows. Section II discusses the design of the JConqurr toolkit. Section III discusses the annotation and directive approaches used in task, data, divide and conquer, pipeline and GPU based parallelism. Section IV discusses the performance of the applications which are paralleled using the JConqurr toolkit.

II. Design

In this section we present the design of the JConcurr toolkit. Figure 1 shows the high level architectural view of the toolkit. Eclipse is an IDE based on the plug-in base architecture [12]. Also the IDE includes plug-ins to support in developing java development tools (JDT). We have considered the most of the JDT components in designing of the toolkit. JDT Java Model component used to navigate java elements in the project [13]. Eclipse AST API [14] used to manipulate sequential code and to generate the code which enhanced with parallel patterns. Eclipse Build API [15] used to build the new project in the workspace.

At the first step developer has to use our annotation and directive libraries to provide the meta information for the relevant parallel pattern. Since Java annotations [16] does not supports annotating statements inside the method we have came across to use a static method library as the directive mechanism. At the core of the toolkit we traverse through the directory hierarchy of the project and create a project similar to the existing one. Then toolkit goes through the each compilation unit and analyse the annotations types used by the developer. We use a filtering mechanism with the help of Eclipse AST to filter the annotated methods. After that based on the parallel pattern developer has specified is directed to the relevant parallel handler as shown in the Figure1. Each handler manipulates the code according to the type of the parallel pattern and generates the new methods which support underline multi-core architecture. Finally project builder uses those modified methods to generate the new compilation unit and write them back to the relevant packages of the newly created project. At the end just a click of a button, developer can create a project similar to his current project which has enhanced with the parallel patterns.

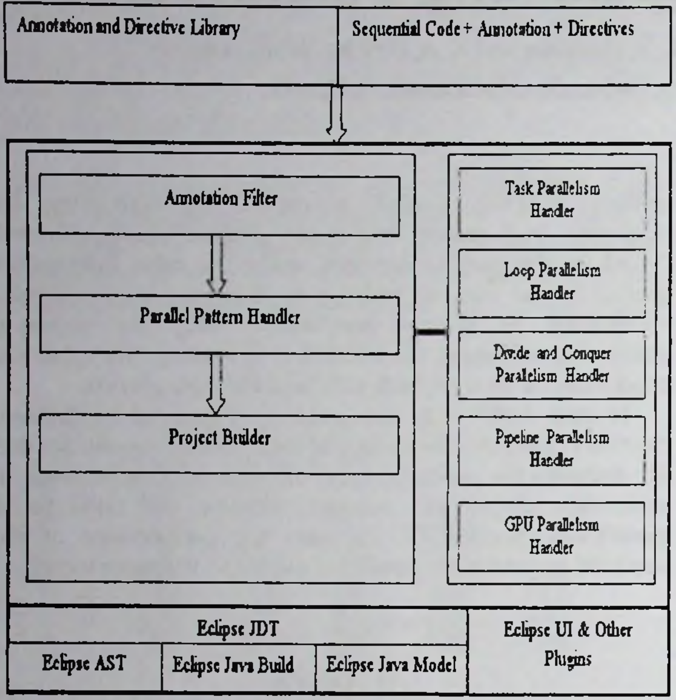

Figure 1: Architectural design of the JConqurr toolkit.

III. Annotations and Directive approaches

Our toolkit mainly link with the Eclipse AST API in code processing. Directives are used to structure the parallel pattern approaches. Using those structures we generalise the patterns so that it will make easy for the developers to achieve the relevant parallel pattern. We assume that the developer has the basic idea of the organization of tasks inside the source code. In below subsections we discuss about annotations and directive structures used in the toolkit.

A. Task Parallelism

Task parallelism happens when we can execute set of independent tasks (set of code segments) on different processors simultaneously. The performance gain of the task parallelism depends on degree of coarse granularity [19]. Most effective way to handle the execution of those tasks can be achieved by using a pool of long lived threads. Thread pool eliminate the overhead of creating and destroying threads [18]. In Java a thread pool is called an executor service. It provides the ability to submit task, the ability to wait for a set of submitted tasks to complete, and the ability to cancel uncompleted tasks. [18] So considering these facts we use the executor framework [17] to reformat the sequential code at the conversion process. In the source conversion process we submit the created tasks to executor framework. Figure 2 shows the example of annotation and directive mechanism used in to achieve task parallelism.

```
@ParallelTasks
public void applyOperations(BufferedImage input) {

    // Task1:creates the gray image and its histogram

    Directives.startTask();
    BufferedImage gray = op.getGrayImage(input);
    PlanarImage im1 = PlanarImage.wrapRenderedImage(gray);
    (new Main()).generateHistogram(im1, "gray");
    Directives.endTask();

    // Task2:creates the negative image and its histogram

    Directives.startTask();
    BufferedImage negative = op.getNegativeImage(input);
    PlanarImage im2 = PlanarImage.wrapRenderedImage(negative);
    (new Main()).generateHistogram(im2, "negative");
    Directives.endTask();

    // Task3:creates the mirror image and its histogram

    Directives.startTask();
    BufferedImage mirror = op.getMirror(input);
    PlanarImage im3 = PlanarImage.wrapRenderedImage(mirror);
    (new Main()).generateHistogram(im3, "mirror");
    Directives.endTask();
}
```

Figure 2: Annotation and directive approach in Task parallelism.

In task parallelism we provide directives to define tasks and to define the barriers. Tasks has to be surrounded with "Directives.startTask();" and "Directives.endTask();" as shown in the Figure 2. To define a barrier developer can use the "Directives.Barrier()" directive after the relevant set of tasks.

At the conversion process toolkit first filter the methods which have "@ParallelTask" annotations. Then the filtered methods are submitted to the task parallelism handler. In the handler toolkit analyses the directives used inside the methods and filter the tasks that need to be parallel. In filtering tasks inside the methods we do a dependency analysis to filter the variables which are linked inside the tasks. Mechanism is little bit similar with the approach used in Embla [23]. Then the method is recreated by injecting relevant code segments to run the tasks using the java executor framework. Barriers are implemented using the cyclic barrier [20] of java.concurrent.util API [17].

B. Data Parallelism

Data parallelism occurs when the same operation has to apply for different set of data. Degree of the granularity is dependent on the data size. Integration of task parallelism and data parallelism is still an open area [19]. In our toolkit we provide the data and task parallelism as two separate options. We mainly focus on the "for loop" parallelization.

The annotation and directive approach for "for loop" parallelization is shown in the figure 3. We used "@parallelFor" annotation to filter the methods. Then using the "Directive.forLoop();" we can select the "for loops" which requires the loop parallelism. In addition to that shared variables and barriers can be defined using the "Directive.shared(identifierName)" and Directive.barrier(); directives. Toolkit is capable of parallelizing multiple "for loops" inside a method.

```
@ParallelFor
public void matrixMul() {
    Directives.forLoop();
    for (int i = 0; i < 1000; i++) {
        for (int j = 0; j < 1000; j++) {
            for (int k = 0; k < 1000; k++) {
                arr3[i][j] += arr1[i][k] * arr2[k][j];
            }
        }
    }
}
```

Figure 3: Annotation and directive approach in Data parallelism.

In the conversion process filtered methods annotated with "@ParallelFor" are submitted to the data parallelism handler. Based on the number of processors in the underline architecture toolkit decides the number of splits that need to do for the loop. Then the each splits of "for loop" organized as tasks and submits them to the executor framework.

C. *Divide and Conquer Parallelism*

Divide and Conquer Algorithms can be identified as a major area in which parallel processing can be applied. In these algorithms a problem is recursively divided into sub-problems and executed. When the sub-problems are solved the results of them are combined together to obtain the complete solution to the original problem. [23] The sub-problems are defined in such a way that they can be executed in parallel. The basic requirement for these sub-problems to be executed in parallel is that, a particular sub-problem needs to be independent of the result of another sub-problem. In JConqurr, we introduce a mechanism to enable parallel execution of Divide and Conquer Algorithms. [9]

We use the java.util.concurrent package [24] in enabling parallelism in JConqurr. This package provides efficient implementations of utility classes commonly encountered in parallel programming. [25]

```
@DivideAndConquer
public void mergeSort() {
    int[] workSpace = new int[noOfElements];
    sort(workSpace, 0, noOfElements - 1);
}

private void sort(int[] workSpace, int lowerBound, int upperBound) {
    if (lowerBound == upperBound)
        return;
    else {
        int mid = (lowerBound + upperBound) / 2;
        sort(workSpace, lowerBound, mid);
        sort(workSpace, mid + 1, upperBound);
        merge(workSpace, lowerBound, mid + 1, upperBound);
    }
}

private void merge(int[] workSpace, int lowPtr, int highPtr, int upperBound) {
```

Figure.4 Annotation and directive approach in Divide and Conquer parallelism.

Basically we can identify existing divide and conquer algorithms, by analysing the source code. If a particular method is invoked multiple times from within itself on independent subset arguments, those can be identified as divided sub-problems. In sequential execution, these sub-problems are solved one after the other, but in converted source code we enable parallel execution of these sub-problems via assigning them to separate tasks and executing those tasks simultaneously. The required functionalities are provided by java.util.concurrent [23] package.

```
public class sortTask extends FJTask {
    int[] workSpace;
    int lowerBound;
    int upperBound;

    public sortTask(int[] workSpace, int lowerBound, int upperBo
        this.workSpace = workSpace;
        this.lowerBound = lowerBound;
        this.upperBound = upperBound;
    }

    public void run() {
        sortTask task1 = null, task2 = null;
        if (lowerBound == upperBound)
            return;
        else {
            int mid = (lowerBound + upperBound) / 2;
            task1 = new sortTask(workSpace, lowerBound, mid);
            task2 = new sortTask(workSpace, mid + 1, upperBound)
            merge(workSpace, lowerBound, mid + 1, upperBound);
        }
        if ((task1 != null) && (task2 != null)) {
            coInvoke(task1, task2);
        } else if (task1 != null)
            invoke(task1);
        else if (task2 != null)
            invoke(task2);
    }
}
```

Figure 5: Usage of java.util.concurrent package for parallelizing merge sort.

In conversion process a new inner class, extending class FJTask (provided by java.util.concurrent) is added to the corresponding class where divide and conquer algorithm is found. The method having recursive calls to it-self is then placed as the run method with recursive calls replaced with new tasks generated correspondingly. In creating this class, method local variables of the recursive method and the types of variables which are passed as method arguments has to be highly considered and defined. The source code segment in Figure 5 shows the overview of the new class generated for merge-sort algorithm.

Then the source code is analysed to identify the first invocation of the recursive method and that invocation is replaced with a generated new task correspondingly.

D. *Pipeline Parallelism*

Pipeline parallelism can be applied when a set processes has to be applied on a set of data. To increase the efficiency, the data set can be segmented and while one data segment has completed the first process and enter to the second process, another segment of data can enter to the first stage for the execution of first process. In Pipeline processing, although the time taken for processing a particular data set is not minimized, an overall performance gain can be seen due to the pipelined process. [9, 26] The Figure 6 illustrates the pipeline process strategy.

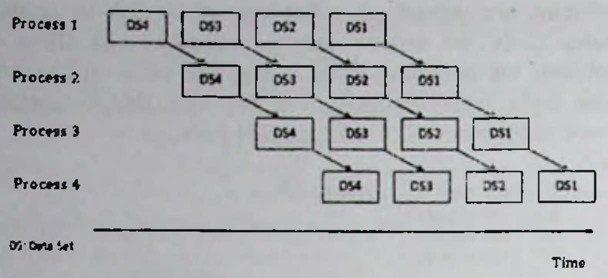

Figure 6: Pipeline Process Strategy

JConqurr provides efficient processing by delegating each pipeline stage to a separate thread of execution. [2] Data flow among the processes is handled with the use of Java bounded blocking queues.

Currently two flavors of pipeline processes are facilitated in JConqurr. They are regular pipeline processes and Split-join processes. [28]

1) Regular Pipeline Process: This approach can be used when the load on each stage is almost equal so that the set of processes on a particular dataset can be performed sequentially. This is the most generic method of pipeline parallelism. The converted pipeline process follows a producer consumer strategy where each stage works as a consumer for the previous stage and a producer for the next stage. Stage (i) consumes the data produced or processed by stage (i-1) and produces the data for the stage (i+1) by performing the expected functions at that stage.

2) Split-Join Pipeline Processes: Being another flavor of pipeline processes, this approach is used when the load on a particular stage is considerably high so that the flow of processes may block due to the excessive load on that stage. To overcome this situation, the execution of the process corresponding to that stage itself will be assigned to several threads, so that the excessive load will be divided among them in a round robin fashion. [3]

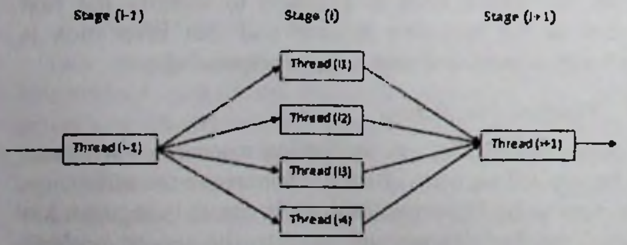

Figure 7: Illustration of Split-Join Pipeline Process Strategy

In JConqurr, this is developed as an extended process of regular pipeline process where, the basic functionality differs only for intermediate input/output processes. The user is facilitated to specify the number of split paths depending on the load of the respective pipeline stage.

Depending on the directives specified, the tool needs to recognize the split stage (stage (i)). Then after the data set is processed by the stage (i-1), the output has to be written to multiple output queues in a round robin fashion, so that processing at stage (i) can be assigned to multiple threads. The output of stage (i) will be written separately to individual output queues. Stage (i+1) is considerate on retrieving data from multiple input queues in the same order they were assigned to multiple threads. Input/ Output processes has to be carefully thought of to assure the input order of the dataset is preserved.

Annotation and Directives in parallelizing pipeline processes in JConqurr is designed as follows.

- @ParallelPipeline: This annotation is used above a method in which pipeline process exists. The purpose of this annotation is to ease the AST to filter the required methods for the conversion process.
- Directive.pipelineStart(): In pipeline processes a particular set of functions are applied to a set of data. This results in a loop in the program, a while loop most probably. To ease the extraction of the required code segment for the conversion process, this directive is used above the loop which contains the pipeline process.
- Directive.pipelineStage({input}, {output}): This directive is used to separately identify the pipeline stages. Within the loop, the code segment in between ith directive and the (i+1)th directive is considered as the process at ith stage. The last pipeline stage includes the processes from last directive to the end of the loop.

  This directive generally takes two arguments, namely, the input and output for a particular stage. Depending on the application and the process in a given stage, multiple inputs and/or outputs are also possible. In such a scenario they need to be defined as a set. In a nutshell, the input refers to what a process has to dequeue from the input queue and the output refers to what has to be enqueued to the output queue.
- Directive.pipelineSplitStage({input}, {output}, number of split paths): This directive is used to specify the stage containing a split mechanism, which indirectly says that the load at this stage is high. The inputs and the outputs will be specified in the usual way and additionally, the number of split paths will be specified. The number of threads running at the particular stage will be equal to the number of split paths specified.
- Directive.pipelineEnd(): This directive is used to mark the end of the pipeline process. If there is a particular set of functions to be done at the end of the process, such as closing the input files, they can be specified as a block following the above directive.

A sequential programme, marked according to the above specification is ready to be converted into the parallelly executable code via JConqurr.

The figure illustrates a three staged pipeline process along with the corresponding annotations and directives, where pipeline parallelism can be applied.

```
@ParallelPipeline
method(){
        ...
        Directive.pipelineStart();
        while(more input data){
                Directive.pipelineStage(input, x);
                x = functionOne(input);
                Directive.pipelineStage(x, y);
                y = functionTwo(x);
                Directive.pipelineStage(y, z);
                z = functionThree(y);
        }
        Directive.pipelineEnd();
        ...
}
```

Figure 8: Annotation and Directive Usage in Pipeline Processes

In the conversion process the three pipeline stages will be handled by three threads correspondingly. The dataflow among the threads is handled with the use of queues where a thread will dequeue the data enqueued by the previous thread or the input file. The dequeued data will be subjected to the process at the current stage and will be written to another output queue to be dequeued by the thread corresponding to the next pipeline stage. Queues are shared among the threads to enable a thread to dequeue the data enqueued by another thread.

Internal Process

First the AST filters the methods marked with the annotation @parallel pipeline. Then it searches for the loops which are marked with the directive Directive.pipelineStart(). After filtering the corresponding loop with the pipeline stages identified, the converted source code will be generated which is ready for the parallel execution.

Handling threads

The number of threads created is equal to the number of pipeline stages specified in the process. When specifying a particular stage the user will specify the inputs and outputs to that particular stage. The specified inputs and outputs are the properties of the thread corresponding to that stage; hence the constructor of the thread will depend on the passed arguments of the directive.

The run method will contain the loop in which the code segment for a particular stage goes in. The terminal condition of the loop in a particular thread depends on the place of the process to which the thread corresponds. For the thread corresponding to the first stage of the pipeline process, the terminal condition would be the same as in the sequential source code. For the other threads, the loop would continue until the input queue for that stage becomes empty.

Data Flow

The data flow among the threads is handled with the use of queues. For the convenience and reliability we use Java bounded blocking queues. This ensures that the sequence of the input data set is maintained via avoiding null returns and unsuccessful insertions. Further using bounded queues helps to manage possible memory overflows as well.

E. GPU Parallelism

GPU parallelism is pretty much suitable when we have to compute a large number of small calculations. Multiplication of a matrix of a high order would be a good example.

Today, the programmable Graphic Processor Unit or GPU has evolved into a highly parallel, multithreaded, many-core processor with higher computational power and also higher memory bandwidth. GPU is specialized for computing intensive and highly parallel computations, exactly what is needed in graphics rendering and this is also the very property that we are using in our toolkit.

In the conversion process the toolkit first filters the methods which have "@GPU" annotations. Then the filtered methods are submitted to the GPU handler via "CompilationUnitFilter". Inside the handler the toolkit analyses the method which is annotated and filters any "for" loops which can be parallelised.

```
@GPU
public void matrixMultiply() {
    Directives.gpuforLoop();
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            C[i][j]=0;
            for (int k = 0; k < size; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Figure 9: Annotation approach in GPU parallelism

NVIDIA has introduced CUDA [28], a general purpose parallel computing architecture with a new parallel programming model. In this architecture we can instruct to execute our programmes in GPU using this extended C language. C for CUDA extends C by allowing the programmer to define C functions, called "kernels". And when you call such a function it is executed N times in parallel by N different CUDA threads [28].

So our identified method is recreated as a CUDA file which is an extended version of C language, and that is saved in a separate file at the time of the conversion with the file extension ".cu". This file is created by injecting relevant code segments to the method and according to the CUDA syntax. A sample CUDA file is shown in the figure 10.

But to execute this code segment in GPU it must be compiled using an "nvcc" [28]. "nvcc" is a compiler driver that simplifies the process of compiling C for CUDA code. Then the compiled code is saved in another separate file which is a binary file and is saved with the file extension called ".cubin". This compilation process, and saving it to a separate file is also done during the conversion and it is done using a command. Such command line argument is showed in the Figure 11.

```
__global__ void sampleKernel(float** A, float** B, int size, float** C)
{
  const unsigned int tidX = threadIdx.x;

  for (int j=0; j<size; j++)
  {
        C[tidX][j] = 0;
        for (int k=0; k<size; k++)
     {
              C[tidX][j] += A[tidX][k] * B[k][j];
     }
  }
  __syncthreads();
}
```

Figure 10: A sample CUDA file which is created using the annotated method.

```
"<nvcc compiler path(nvcc.exe)>"  -arch sm_10 -ccbin
"<Microsoft Visual Studio bin path>" -Xcompiler "/EHsc
/W3 /nologo /O2 /Zi /MT" -maxrregcount=32 -cubin -o
"<filePath>\<cubinZFileName>"
"<filePath>\<cuFileName>"
```

Figure 11: A sample command used to compile the CUDA file into a CUBIN file.

After the conversion, the CUBIN file should be loaded using the JCuda driver bindings [11]. JCuda is a Java bindings for the CUDA runtime and driver API. With JCuda it is possible to interact with the CUDA runtime and driver API from Java programs, as it is a common platform for several JCuda libraries. [11]

Using these libraries we can control how to execute a kernel (CUDA function) from the loaded module, initialize the driver, load the CUBIN file and obtain a pointer to the kernel function, allocate memory on the device and copy the host data to the device, set up the parameters for the function call, call the function, copy back the device memory to the host and clean up the memory, etc. So after the calculation inside GPU is done, the result is copied back to our Java project and carried forward. Using this method we are able to occupy large number of cores in the GPU and get results calculated simultaneously with increased performance.

IV. Performance

We have tested our toolkit in converting projects and applications into parallel enhanced projects. Projects are tested in both sun JDK 1.6 and OpneJDK 1.6 environments. In below sections we discuss the performance of each parallel pattern using the most standard applications. Applications are tested on dual core and quad core machines.

A. *Performance of Task Parallelism*

Below graph shows the performance analysis of converted matrix multiplication project vs. the sequential matrix multiplication project. In here developer has used the task parallel annotation and directives approach. The graph clearly shows the exponential growth performance gain with the incensement of the coarse granularity. We were able to achieve considerable performance gain in quad core processor. During the testing both JDK versions showed us similar behaviour.

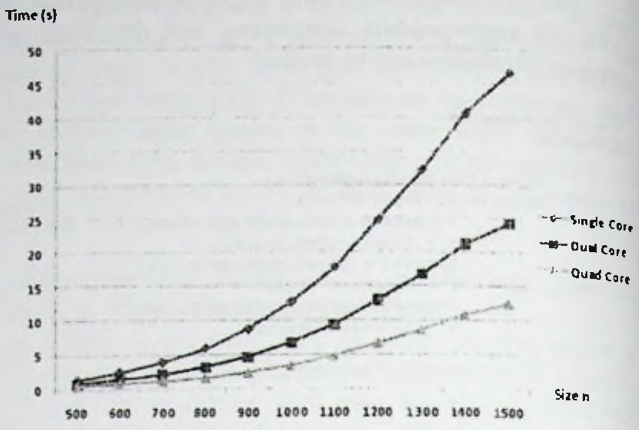

Figure 12: Performance Monitoring in Task Parallelism in a 2.5 GHz Quad core processor machine with a 4GB RAM.

B. *Performance of Data Parallelism*

In here we have tested the Eigen value calculation application with the use of data parallel annotation and directive schemes. In here also performance gain increased with the granularity of the data parallelism. Considerable performance has monitored in the quad core processor.

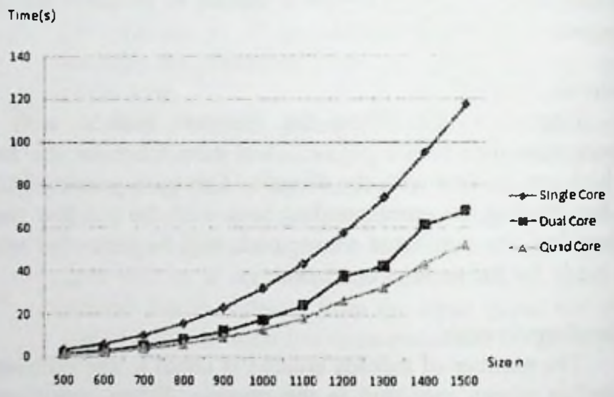

Figure 13: Performance Monitoring in Data Parallelism in a 2.5 GHz Quad core processor machine with a 4GB RAM.

C. *Divide and conquer*

The graph in figure 14 depicts the results of performance testing of sequential and parallel execution of merge sort algorithm. When the graph is analysed it is seen that the performance of the parallelism enabled source code exceeds that of the sequential code only after a certain limit. This limitation occurs because when the data to be processed is low, the overhead caused by the threads exceeds the achievable gain of performance. But when the data to be processed is high the gain of performance in parallel execution is considerably high.

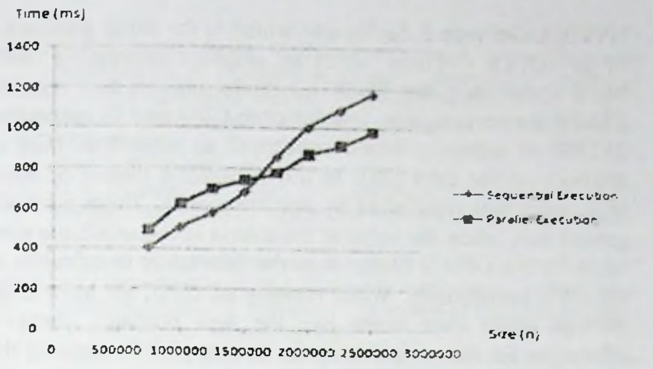

Figure 14: Performance Monitoring in Divide & Conquer Parallelism in Merge Sort.

Figure 15 depicts the performance analysis of quick sort in sequential execution and in parallel execution in dual core and quad core machines. The performance has improved with increasing size of the problem and the number of processors. At considerably small problem sizes, quick sort also demonstrates the behaviour of the sequential programme being faster than the parallel programme.

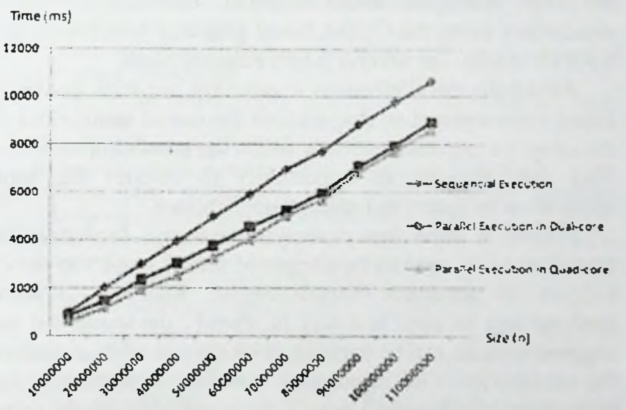

Figure 15: Performance Monitoring in Divide & Conquer Parallelism in Quick Sort

D. Performance of Pipeline Parallelism

In this section we discuss the performance analysis for the test results for pipeline parallelism. The process was tested with using two applications mainly; a file processing application and a data compression application.

1) File Processing: This is a hypothetical application where a programme reads a text input and applies a set of heavy numerical calculations for the data read. For the testing purposes, the calculations were designed in a way that each stage carries equal loads. The following graph depicts the time taken for processing different sizes of data sets.

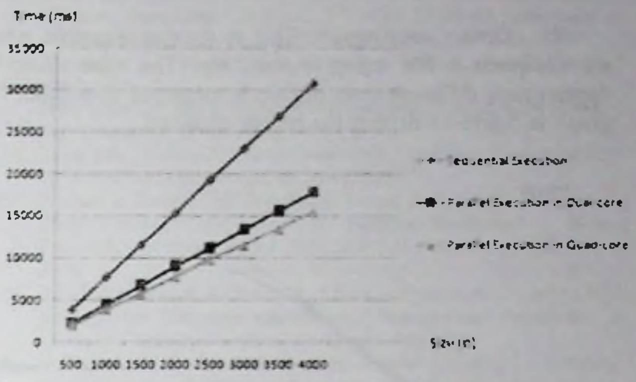

Figure 16: Performance monitoring in pipeline parallelism – File Processing

According to the representation of Figure16, it can be clearly seen that the performance of the parallelly executable code is a way beyond than that of the sequential code. Further the increasing gap in between the two plots illustrates the increasing performance gain with the increasing number of data to be processed.

The Figure 17 depicts the results obtained for a similar file processing application with unbalanced load. The calculations are designed in such a way that. the load at a given stage is a multiple of the load at other stages. When the regular pipeline process is applied on the given application, the performance gain is limited because of the bottleneck at the stage with higher load. When the split-join process is applied this bottleneck is avoided and hence results in an excellent performance gain.

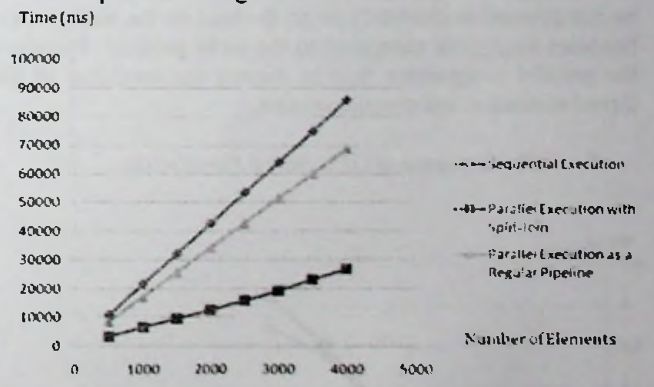

Figure 17: Performance Comparison of Regular Pipeline and Split-join – File Processing

But most of the real world examples fail to fulfil the eligibility requirements for the pipeline process to be applied. Since the thread behaviours are dependent on the decisions of the JVM, the performance gain is not guaranteed. But any application with considerably higher load at each stage is eligible for the conversion process.

2) *Data Compression:* This is another example where we compress a file using java.util.zip. The time taken for compressing different sizes of files is measured over time. The graph in figure 18 depicts the results obtained.

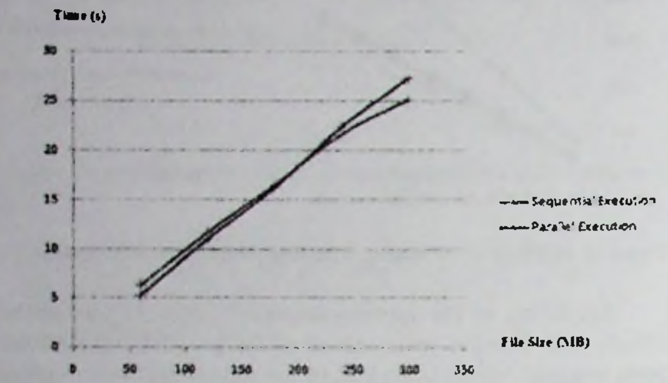

Figure18: Performance monitoring in pipeline parallelism – Data Compression

The data compression example demonstrates an unexpected behaviour with respect to other parallel applications. When the size of the file to be compressed exceeds a certain limit, the performance of the parallel programme degrades. The reason for this can be interpreted as the unbalanced load among the stages of the pipeline. The process consists of two steps namely, reading the input file and writing in to the zip output stream. When the file size to be compressed is extremely large, the load on the read process becomes negligible compared to the write process. Therefore the parallel programme fails to exceed the overhead of the thread execution and communication.

E. Performance of GPU based Parallelism

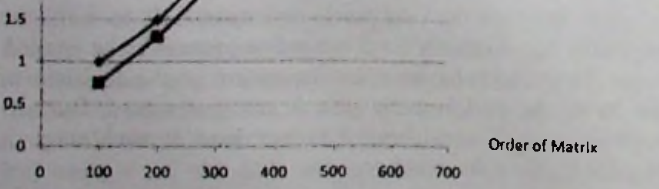

Figure 19: Performance Monitoring in GPU based Parallelism

In this section we discuss the performance analysis for a program which was run in the GPU. The GPU used was an NVIDIA GeForce 9 Series one which is the ninth generation of NVIDIA's GeForce series of graphics processing units. More specifically the 9800GX2 model utilizes two separate 256-bit memory busses, one for each GPU and its respective 512MB of memory, which equates to an overall of 1GB of memory on the card [28]. In this program a matrix of some high order was multiplied by another matrix. There what we gained was, when the order of the matrix is too small, the time taken by the GPU is higher than the time taken to calculate in the CPU sequentially. When running on GPU, we have to go through some over heads like the Java bindings, memory allocation for the variables in the device (GPU), copying the values to the device, copying back the results, clearing the memory etc. So what we observe was that for small order matrices the above mentioned overhead heaver. But when the order of the matrix is increased at a certain level it gave a performance gain. So when the calculation is bigger it has overcome that overhead. The results were analysed in the following graph.

V. Conclusion And Future Work

We have discussed about a toolkit which can heavily support multi-core programming in Java. In addition to that we have discussed about a novel technique to achieve parallelism using the CUDA based graphical processing units. Tool kit is open for several future enhancements.

Automatic parallelization is one area we look forward in future enhancement so that without the use of annotations and directive we can automatically fulfill the parallel optimization. This may provide an opportunity to convert the legacy application to optimized parallel applications.

Further a dependency analysis tool and load balancing techniques will need to be integrated into JConqurr to provide support to automatic parallelization. Mainly dependency analyzer can be used in a way to identify the sequential code segments which can be parallel. With the use of these features, the complexity of the annotation and directive mechanism can be avoided which would benefit the pipeline processes greatly.

Improvements will need to be introduced in order to reduce the overhead of communication and thread scheduling. Currently the applications of pipeline and split join processes are limited, due to these facts. To get the maximum use of the conversion process, better communication mechanisms will have to be thought of.

Considering the GPU based parallelism we look forward to have a mapping between mathematical functions of Java and CUDA. Another area to be improved is to utilize both CPU and the GPU to achieve parallelism so that while one computation happens in the GPU, CPU can carry out the other independent tasks. But tool kit provides huge flexibility to the developer. It opens the gate to developer to think correctly and utilize the code segments so that he can achieve parallelism in different manner. For example it's possible to call the GPU based parallel method through a CPU thread. In terms of these facts JConqurr can consider as a more flexible toolkit for multi-core programming.

References

[1] Manish Gupta, Sayak Mukhopadhyay, Navin Sinha, "Automatic Parallelization of Recursive Procedures," *International Journal of Parallel Programming*, v.28 n.6, p.537-562, Dec 2000.

[2] Barbara Chapman and Lei Huang. "Enhancing OpenMP and Its Implementation for Programming Multicore Systems." Invited Paper, *Proc. PARCO*, 2007: pp. 3-18.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "CILK: An efficient multithreaded runtime system," in *Proc. 5th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pages 207–216, Jul 1995.

[4] J. S. Danaher, "The JCilk-1 Runtime System", Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, Jun 2005.

[5] James Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*, 1st ed, O'Reilly, Jul 2007.

[6] *AXON7 MULTICORE SOLUTIONS White paper* ©Axon7 – Mar 2008.

[7] Mark Bull, Scott Telford. "Programming Models for Parallel Java Applications," Edinburgh Parallel Computing Centre, Edinburgh, EH9 3JZ, 2000.

[8] Franko Otto, Victor Pankratius, Walter F.Tichy. "High-level Multicore Programming with XJava", *31st ACM/IEEE International Conference on Software Engineering (ICSE 2009), New Ideas and Emerging Results*, May 2009.

[9] Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill, *Patterns for Parallel Programming*, Addison-Wesley Professional, Sept 2004.

[10] Yonghong Yan, Max Grossman and Vivek Sarkar "JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA," *Europar* 2009.

[11] "Eclipse.org home". [Online]. Available: http://www.eclipse.org/. [Accessed: 30/04/2010].

[12] "Notes on the Eclipse Plug-in Architecture". Azad Bolour and Bolour Computing. [Online]. Available: http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html. [Accessed: 30/04/2010].

[13] "Eclipse Java development tools (JDT)". [Online]. Available: http://www.eclipse.org/jdt/. [Accessed: 30/04/2010].

[14] Lulian Nemtiu, Jeffrey S. Foster and Michael Hicks. "Understanding Source Code Evolution Using Abstract Syntax Tree Matching," In *Proc. International Workshop on Mining Software Repositories (MSR)*, pages 1-5, Saint Louis, Missouri, USA, May 2005.

[15] "JDT Core Component". [Online]. Available: http://www.eclipse.org/jdt/core/index.php. [Accessed: 30/04/2010].

[16] Tongxin Bai, XingShen, Chenliang Zhang,William N.Scherer,Chen Ding and Michael L.Scott. "A Key-based Adaptive Transactional Memory Executor." Computer Science Department, University of Rochester, Computer Science Department, The College of William and Mary, Computer Science Department, Rice University, Tech.Rep 2007.

[17] "JDK 5.0 Developer's Guide: "Annotations" Sun Microsystems". [Online]. Available: http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html. [Accessed: 30/04/2010].

[18] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[19] Carwyn Ball and Mark Bull. "Barrier Synchronization in Java," Tech.Rep High-End Computing programme (UKIEC), 2003.

[20] Ursula Fissgus, Thomas Rauber, Gudula Rünger, "A Framework for Generating Task Parallel Programs," In *Proc 7th Symposium on the Frontiers of Massively Parallel Computation – Frontiers*, pp.72-80,1999.

[21] M.Danelutto, M.Pasi, M.Vanneschi, P.Dazzi, D.Laforenza and L.Presti "PAL: Exploiting java annotations for parallelism," in European Research on Grid Systems, pp.83-96. Springer US 2007.

[22] K.-F. Faxen et al., "Embla – data dependence profiling for parallel programming," in Complex, Intelligent and Software Intensive Systems, 2008.

[23] Radu Rugina, Martin Linard. "Automatic Parallelization of Divide and Conquer Algorithms," in *Proc. 7th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp 72-83, 1999.

[24] "Overview of package util.concurrent Release 1.3.4." [Online] Available: http://g.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html. [Accessed: 20/10/2009].

[25] Doug Lea. "A Java Fork/Join Framework," in *Proc. of the ACM 2000 conference on Java Grande*, pp.36–43, 2000.

[26] Michael I. Gordon, William Thies, Saman Amarasinghe, "Exploiting Course-Grained Task, Data and Pipeline Parallelism in Stream Programs," in *Proc. of the 2006 ASPLOS Conference*, pp 151-162, 2006.

[27] Bill Thies, Michal Karczmarek, Saman Amarasinghe, "StreaMIT: A language for Streaming Applications." *International Conference on Compiler Construction*. Grenoble, France, Apr, 2002.

[28] "NVIDIA CUDA™ Programming Guide". [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf. [Accessed: 30/04/2010].