International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:3, No:7, 2009

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

Abstract—With the popularity of the multi-core and many-core architectures there is a great requirement for software frameworks which can support parallel programming methodologies. In this paper we introduce an Eclipse toolkit, JConqurr which is easy to use and provides robust support for flexible parallel progrmaming. JConqurr is a multi-core and many-core programming toolkit for Java which is capable of providing support for common parallel programming patterns which include task, data, divide and conquer and pipeline parallelism. The toolkit uses an annotation and a directive mechanism to convert the sequential code into parallel code. In addition to that we have proposed a novel mechanism to achieve the parallelism using graphical processing units (GPU). Experiments with common parallelizable algorithms have shown that our toolkit can be easily and efficiently used to convert sequential code to parallel code and significant performance gains can be achieved.

Keywords—Multi-core, parallel programming patterns, GPU, Java, Eclipse plugin, toolkit,

I. Introduction

WITH the advent of multi-core processors, the importance of parallel computing is significant in the modern computing era since the performance gain of software will mainly depend on the maximum utilization across the cores existing in a system. It is necessary that tools exists to make the full use of the capabilities offered by the parallel computing. Though current parallelizing compilers support parallelization at loop level [1] it is a hard task to detect parallel patterns and do conversions at the compiler level.

Already there exists languages and libraries like OpenMP [2], Cilk++ [3], Intel TBB for C++ [4] language to support the multi-core programming. Also there are few recent developments of tools for Java language such as JCilk [5], JOMP [6] and XJava [7]. Among them only a few libraries are capable of providing the data, task, divide and conquer and pipeline parallelism pattern [8] in a single toolkit or library.

In this paper we discuss about a multi-core programming toolkit for Java language which uses annotations and a novel directive mechanism to provide meta information of the source code. PAL [9] is one of the tools which use annotations mechanism to achieve parallelism targeting cluster networks of workstations and grids. Even though we use annotations, compared to it we use a different procedure. With the use of meta information we were able to provide a source level parallelism instead of the compiler level parallelism. Our toolkit comes as a plugin for the Eclipse IDE [10]. The toolkit is capable of converting a sequential Java project into a new project which is optimised based on the parallel patterns, using the meta information passed by the user. In order for Java to work with many-core processors we have propose a mechanism to achieve parallelism using the graphical processing unit (GPU) using the JCUDA [11] binding. With that our toolkit can be used for both CPU and GPU parallelism.

The rest 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 for task, data, divide and conquer, pipeline and GPU based parallelism. Section IV discusses the performance of the applications which are parallelized using the JConqurr toolkit and the final section talks about conclusions we have arrived in our research.

II. Design

In this section we present the design of the JConqurr toolkit. The Fig. 1 shows a high level architectural view of the toolkit. Eclipse is an IDE based on the plugin based architecture [12]. Also the IDE includes plugins to support in developing Java development tools (JDT). We have considered the most of the JDT components in designing of our toolkit. JDT Model [13] components are used to navigate Java source code elements in the project. Eclipse Abstract Syntax Tree (AST) API [14] was used to manipulate sequential code and to generate the code which enhanced with parallel patterns. Eclipse Build API [15] was used to build the new project in the workspace.

As the first step the developer has to use our annotation and directive libraries to provide the meta information of the relevant parallel pattern to be applied to a selected code segment. Since Java annotations [16] does not support annotating statements inside a method, we decided 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 the toolkit goes through the each compilation unit and analyse annotations types used by the developer. We use a filtering mechanism with the help of Eclipse AST to filter the annotated methods. Based on the parallel pattern developer has specified these filtered methods are directed to the relevant parallel handler. Each handler manipulates

G.A.C.P. Ganegoda is with the Department of Computer Science and Engineering, University of Moratuwa, Katubedda, Sri Lanka (e-mail: gacpgane@yahoo.com).

D.M.A. Samaranayake is with the Department of Computer Science and Engineering, University of Moratuwa, Katubedda, Sri Lanka (e-mail: dmadhawie@gmail.com).

L.S. Bandara is with the Department of Computer Science and Engineering, University of Moratuwa, Katubedda, Sri Lanka (e-mail: lahirusu@gmail.com).

K.A.D.N.K. Wimalawarne is a lecturer at the Department of Computer Science and Engineering, University of Moratuwa, Katubedda, Sri Lanka (e-mail: kishanwn@gmail.com).

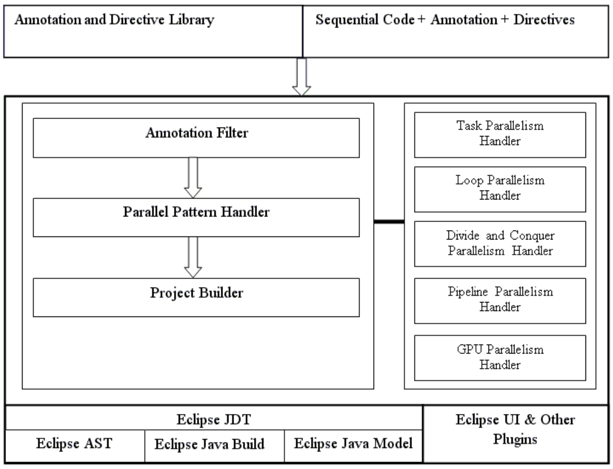


Fig. 1. Architectural design of the JConqurr toolkit

the code according to the type of the parallel pattern and generates new methods which support the underline multi-core/many-core architecture. Finally the project builder uses these modified methods to generate the new compilation unit and write them back to the relevant packages of the newly created project. At the end, with just a click of a button the developer can create a project similar to the current project which has enhanced with the parallel patterns.

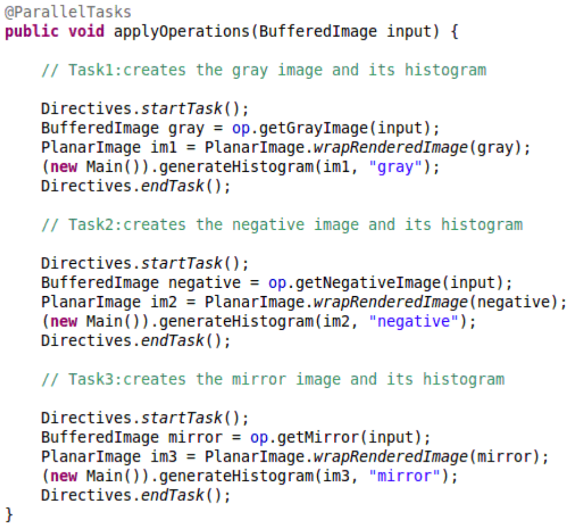
III. Annotations and Directive Approaches

Our toolkit mainly links with the Eclipse AST API for code processing. Directives are used to structure the parallel pattern approach. 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 different annotations and directive structures provided in our toolkit.

A. Task Parallelism

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

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



```
@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();
}
```

Fig. 2. Annotation and directive approach in Task parallelism

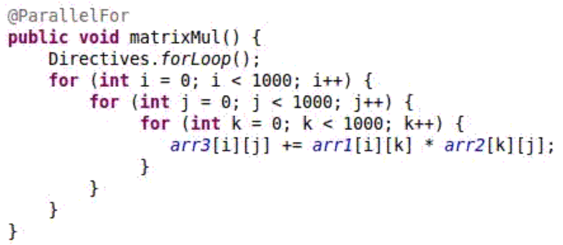
At the conversion process, the toolkit first filters the methods which have @ParallelTask annotations. Then these filtered methods are submitted to the task parallelism handler. In the handler it analyses directives used inside methods and filter tasks that need to be parallel. In filtering tasks inside a method we do dependency analysis to filter variables which are linked inside the tasks. This mechanism is somewhat similar to the approach used in Embla [18]. 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 [19] of java.concurrent.util API.

B. Data Parallelism

Data parallelism occurs when the same operation has to be applied to different sets of data. The degree of the granularity depends on the data size. Integration of task parallelism and data parallelism is still an open area of research and in our toolkit we provide data and task parallelism as two separate options.

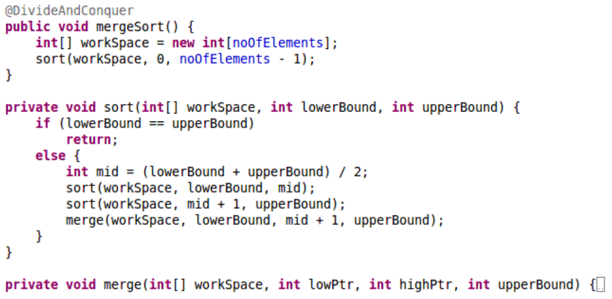
The annotation and directive approach for a *for loop* parallelization is shown in Fig. 3. We used @parallelFor annotation to filter the methods. Then using the Directive.forLoop() we can specify the *for loop* which requires the loop parallelism. In addition to that shared variables and barriers can be define using the Directive.shared(string) and Directive.barrier() directives. The toolkit is capable of parallelizing multiple *for loops* inside a method as well.

During the conversion process, filtered methods which are annotated with @ParallelFor are submitted to the data parallelism handler. Based on the number of processors in the underline architecture the toolkit decides the number of splits that are needed for the *for loop*. Then each split is organized as a task and submitted to the Java Executor framework.



```
@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];
            }
        }
    }
}
```

Fig. 3. Annotation and directive approach in Data parallelism



```
@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) {
```

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

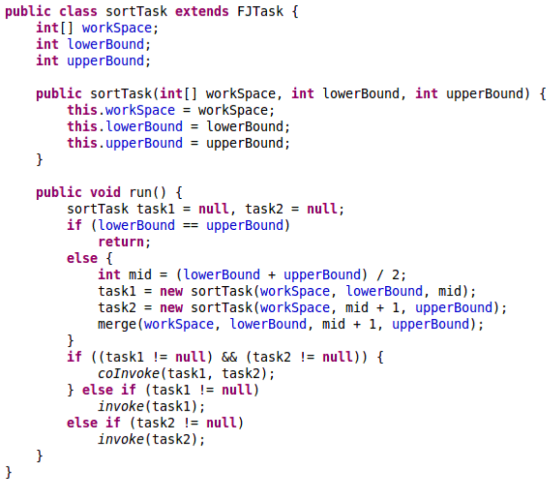
C. Divide and Conquer Parallelism

Divide and conquer algorithms can be identified as another major area in which parallel processing can easily be applied. In these algorithms, a problem is recursively divided into subproblems and executed. When subproblems are solved results of them are combined together to obtain the complete solution to the original problem. Subproblems are defined in such a way that they can be executed in parallel. The basic requirement for these subproblems to be executed in parallel is that, a particular subproblem needs to be independent of the result of another subproblem. In JConqurr, we introduce a mechanism to enable parallel execution of divide and conquer algorithms [20].

We use a Java concurrency package developed by Doug Lea [21] to implement divide and concur parallelism in JConqurr. Divide and concur algorithms have been efficiently solved using the fork-join pattern [8] which we integrated to our toolkit.

JConqurr identifes existing divide and conquer algorithms by analysing a given source code. If a particular method is invoked multiple times from within itself on independent subset arguments they are identified as divided subproblems. In sequential execution these subproblems are solved one after the other, but in converted source code we enable parallel execution of these subproblems via assigning them to separate tasks and executing those tasks simultaneously.

In conversion process a new inner class extending class FJTask [22] is added to the corresponding class where divide and conquer algorithm is found. The method having recursive calls to itself 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. Then the source code is analyzed to identify the first invocation of the recursive method and that invocation is replaced with a generated new task correspondingly. The source code segment in Fig. 5 shows the overview of the new class generated for merge sort algorithm.



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

    public sortTask(int[] workSpace, int lowerBound, int upperBound) {
        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);
    }
}
```

Fig. 5. Usage of FJTask for parallelizing merge sort

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 [23]. The Fig. 6 illustrates the pipeline process strategy.

In our research we are introducing a novel method of annotaions and directive machnism as for other patterns to support pipeline processing using Java. Our proposed approach is different from existing tools that support pipeline processing since they need to special syntax [7] or programing effort [5] where the original source code need to be changed. JConqurr provides an efficient processing by delegating each pipeline stage to a separate thread of execution. Data flows among the processes are 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 [24].

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

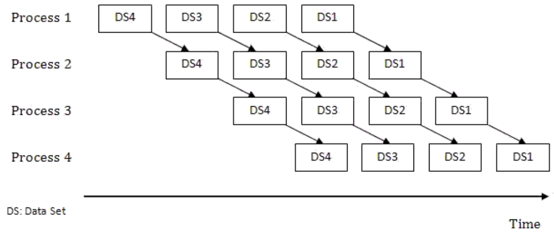


Fig. 6. Pipeline Process Strategy

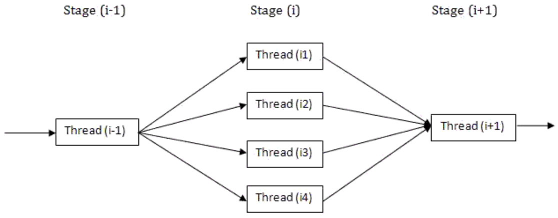


Fig. 7. Illustration of Split-Join Pipeline Process Strategy

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 [24].

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 have 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 result 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 stage directive i and the directive i+1 is considered as the pipeline process at stage i. 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.

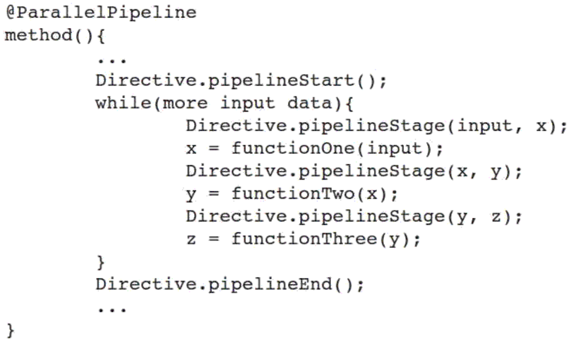
A sequential programme, marked according to the above specification is ready to be converted into the parallel executable code via JConqurr. The Fig. 8 illustrates a three stage pipeline process along with corresponding annotations and directives where pipeline parallelism can be applied.

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. 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.

E. GPU Parallelism

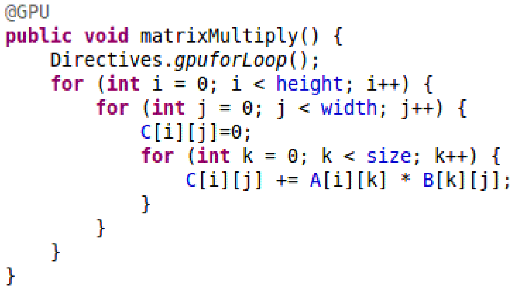
The programmable Graphic Processor Unit or GPU availabe today has evolved into a highly parallel, multithreaded, many-core processor with higher computational power and also higher memory bandwidth. GPU is specialized for computing intensive and highly parallel computations, exactly what is needed in graphics rendering and this is also the very property that we are using in our toolkit. All parallel patterns are not supported by GPU but it has a strong support for data parallelism and limited level of stream programming.

As earlier the conversion process the toolkit first filters the methods which have @GPU annotations. Inside the handler the toolkit analyses for for loops starting with Directive.gpuforloop().



```
@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();
        ...
}
```

Fig. 8. Annotation and Directive Usage in Pipeline Processes

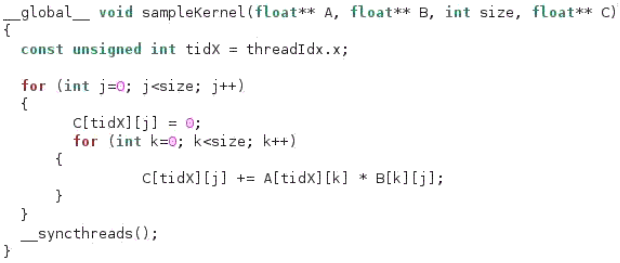


```
@GPU
public void matrixMultiply() {
    Directives.gpuforLoop();
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            C[i][j]=0;
            for (int k = 0; k < size; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Fig. 9. Annotation approach in GPU parallelism

NVIDIA has introduced CUDA [25], 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. CUDA kernels act as single instance multiple thread (SIMT) when you call such a kernel, it is executed on different parallel threads [11].

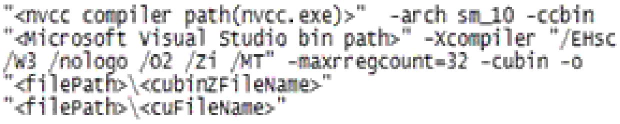
The identified method by directives is recreated as a CUDA file which is an extended version of C language, and that is saved in a separate file at the time of the conversion with the file extension *.cu*. This file is created by injecting relevant code segments to the method with according to the CUDA syntax. The CUDA kernel for the loop in Fig. 9 is shown in the Fig. 10.



```
__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();
}
```

Fig. 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>"
```

Fig. 11. A sample command used to compile the CUDA file into a *cubin* file

To execute this code segment in GPU it must be compiled using an nvcc compiler [25]. The 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 Fig. 11. The advantage of cubin files is that they can be ported to any 32-bit architecture.

After the conversion, the cubin file should be loaded using the JCuda driver bindings [11]. JCuda is a Java binding library 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.

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 can be 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 standard applications. Applications are tested on dual-core and quad-core machines.

A. *Performance of Task Parallelism*

Fig. 12 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 high performance gain with the increment of the coarse granularity. We were able to achieve considerable performance gain with quad-core processors. During the testing both JDK versions showed us similar behaviour.

B. *Performance of Data Parallelism*

We have tested the Eigenvalue calculation application with the use of data parallel annotation and directive schemes.

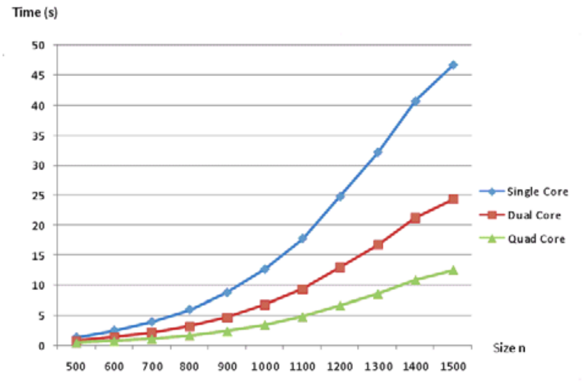


Fig. 12. Performance Monitoring in Task Parallelism in a 2.5 GHz Quad core processor machine with a 4GB RAM

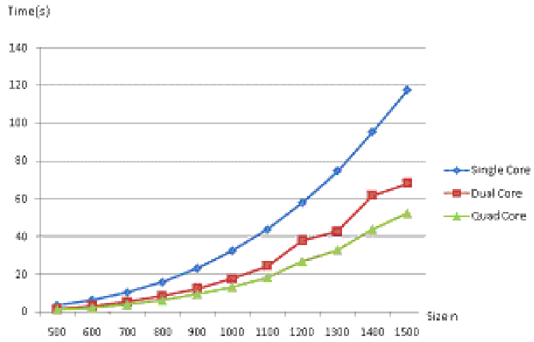


Fig. 13. Performance Monitoring in Data Parallelism in a 2.5 GHz Quad core processor machine with a 4GB RAM

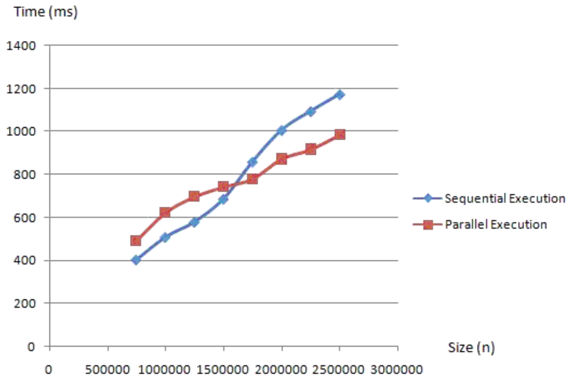


Fig. 14. Performance Monitoring in Divide and Conquer Parallelism in Merge Sort

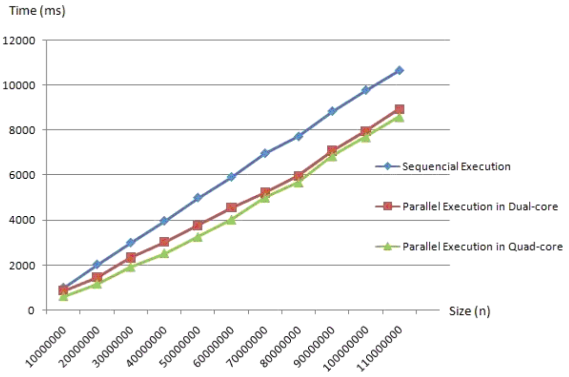


Fig. 15. Performance Monitoring in Divide and Conquer Parallelism in Quick Sort

As Fig. 13 shows the performance gain increased with the granularity of the data parallelism. Considerable performance was monitored in the quad-core processor.

C. *Divide and conquer*

The graph in Fig. 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.

Fig. 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.

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, a file processing application and a data compression application.

1) File Processing: This is an experimental application where a program 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.

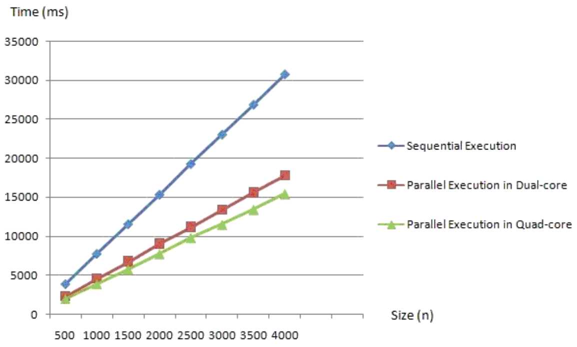


Fig. 16. Performance monitoring in pipeline parallelism - File Processing

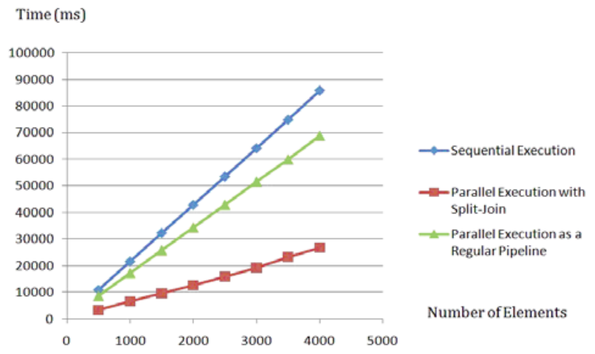


Fig. 17. Performance Comparison of Regular Pipeline and Split-join - File Processing

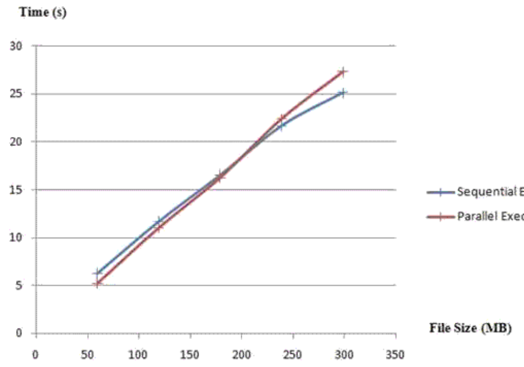


Fig. 18. Performance monitoring in pipeline parallelism - Data Compression

According to the representation of Fig.16, it can be clearly seen that the performance of the parallel executable code is higher 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 Fig. 17 depicts 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.

But most of the real world examples failed to fulfil the eligibility requirements for the pipeline process to be applied. Since the thread behaviors 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 Fig. 18 depicts the results obtained.

The data compression example demonstrates an unexpected behavior with respect to other parallel applications. When the size of the file to be compressed exceeds a certain limit, the performance of the parallel program 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 program fails to exceed the overhead of the thread execution and communication.

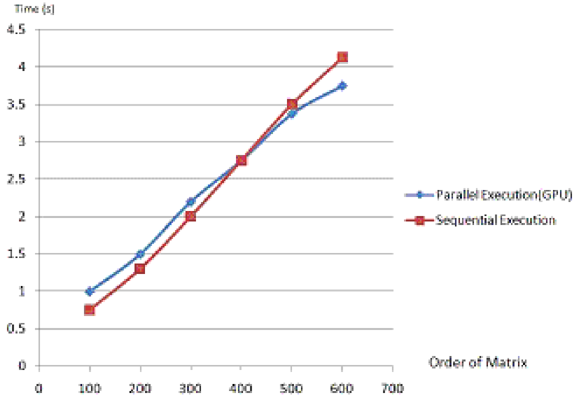


Fig. 19. Performance Monitoring in GPU based Parallelism

E. Performance of GPU based Parallelism

In this section we discuss the performance analysis for a program which was run in the GPU. The GPU used was a NVIDIA GeForce 9800GX2 model which has 256 stream processors and 1GB of memory on the card. In this program a matrix of high order was multiplied by another matrix. We found that 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, it has to go through overheads like the Java bindings, memory allocation for the variables in the device (GPU), copying the values to the device, copying back the results and, clearing the memory. We observed that when the order matrices are small, the above mentioned overhead decrease the overall performance compared to CPU execution. But when the order of the matrix is increased at a certain level it gave a performance gain as the higher number of parallel calculations overcomes those overheads. The Fig. 19 shows the performance graph of this experiment.

V. Conclusion

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. Our toolkit 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 legacy applications 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 a dependency analyzer can be used to identify the sequential code segments which can become 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 applications of pipeline and split-join processes are limited due to these factors. To get the maximum use of the conversion process better communication mechanisms will have to be researched.

In our research we only considered *for loop* parallelism using GPU and investigation on parallelism using other patterns remains as interesting research areas. One such area would be pipeline processing using the streaming capabilities provided with GPU. Another open research area is hybrid programming to utilize both CPU and GPU to achieve maximum parallelism.

References

[1] C. T. yang, S.S. Tseng, M. C. Hsiao, S. H. Kao, "Portable Parallelizing Compiler with Loop Partitioning", *Proc. Natl. Sci. Counc ROC(A)*, Vol. 23, No. 6, 1999. pp.751-756

[2] B. Chapman, L. 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 207216, Jul 1995.

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

[5] J. S. Danaher, "The JCilk-1 Runtime System", Masters thesis, Massachusetts Institute of Technology Department of Electrical Engineer-ing and Computer Science, Jun 2005.

[6] M. Bull, S. Telford. "Programming Models for Parallel Java Applications", Edinburgh Parallel Computing Centre, Edinburgh, EH9 3JZ, 2000.

[7] F. Otto, V. Pankratius, W. 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.

[8] T. G. Mattson, B. A. Sanders, B. L. Massingill, *Patterns for Parallel Programming*, Addison-Wesley Professional, Sept 2004.

[9] 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.

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

[11] Y. Yan, M. Grossman, V. Sarkar, "JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA", *Europar*, 2009.

[12] "Notes on the Eclipse Plug-in Architecture". Azad Bolour and Bolour Computing. [Online]. Available: http://www.eclipse.org/articles/Article-Plugin- 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] L. Nemtiu, J. S. Foster, M. 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] "JDK 5.0 Developers Guide: Annotations Sun Microsystems". [Online].Available: http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html. [Accessed: 30/04/2010].

[17] M, Herlihy, N, Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[18] K. F. Faxen, K, Popov, S, Janson, L, Albertsson, "Embla data dependence profiling for parallel programming," in *Complex, Intelligent and Software Intensive Systems*, 2008.

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

[20] R. Rugina, M. 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.

[21] D. Lea, "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].

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

[23] M. I. Gordon, W. Thies, S. Amarasinghe, "Exploiting Course-Grained Task, Data and Pipeline Parallelism in Stream Programs", in *Proc. of the 2006 ASPLOS Conference*, pp.151-162, 2006.

[24] B. Thies, M. Karczmarek, S. Amarasinghe, "StreaMIT: A language for Streaming Applications", *International Conference on Compiler Construction*, Grenoble, France. Apr, 2002.

[25] "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].