Pyjama: OpenMP-like implementation for Java, with GUI extensions

Vikas
vik609@auckland-uni.ac.nz

Nasser Giacaman
n.giacaman@auckland-.ac.nz

Oliver Sinnen
o.sinnen@auckland-.ac.nz

Department of Electrical and Computer Engineering,
The University of Auckland, New Zealand

ABSTRACT

Incremental parallelism is an uncomplicated and expressive parallelisation practice and has led to wide adoption of OpenMP. However, the OpenMP specification does not present a binding for the Java language and the OpenMP threading model finds limited use for GUI (Graphical User Interface) application development. Our research strives towards supporting OpenMP-like directives to simplify parallelism for Java and address the limitations of the OpenMP model in the development of interactive applications.

We present a compiler-runtime system for OpenMP-like directives in Java, enhanced with GUI-aware mechanisms. This paper describes the compiler and the runtime. We introduce GUI-aware directives and propose methods for incremental concurrency. We present and discuss the performance of programs written using our system by comparing them with previous attempts and traditional ways of parallelisation, using the parallel Java Grande Forum (JGF) benchmarks and a fractal-generation application with a GUI.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*

General Terms

Languages, Performance

Keywords

OpenMP, Incremental Parallelism, Incremental Concurrency, Application Development

1. INTRODUCTION

Parallel programming has been largely restricted to the high performance computing in the scientific and engineering community. The parallel programming tools and practices are not commonly been used in mainstream programming and are considered part of advanced programming. This trend is decidedly bound to change; multi-core processors are fast becoming a norm, and it is evident that software must be parallelised to leverage the power of these processors [7, 18].

Even with the parallelisation tools that already exist, they are poorly suitable for the development of interactive applications for desktops, tablets and mobile devices. The poor suitability of parallelisation tools is because of the two major ways in which interactive applications are different from the scientific applications.

The first major difference between scientific and desktop applications lies in the programming languages. Scientific applications are typically written in speed efficient languages that tend to be low level, for example C and Fortran [4]. Interactive applications, however, tend to be developed using high level and object-oriented languages to promote a software engineering approach to programming. The second major difference lies in the control flow of these applications. Scientific applications tend to be batch-type and compute intensive. The computations tend to be rather regular, and repetitive computations are performed on a vast amount of data. On the other hand, desktop, tablet or smartphone applications have their execution flow determined by the user (and other external inputs). As such, these applications tend to be more interactive and hence the computations tend to be irregular depending on the user actions [10].

With the above differences, it is clear to see why the parallelisation tools typically used in scientific applications have not been embraced by GUI application developers. As a result, many libraries are now emerging to take advantage of multi-cores. Unfortunately, these tools typically involve significant code structuring and the presumption that developers understand complicated parallelisation concepts. OpenMP [14] is an API that has essentially become the de facto choice for scientific applications using C/C++. The large appeal of OpenMP is its incremental parallelism approach, where parallelism is introduced by the addition of compiler comments that leave the original sequential program intact.

Unfortunately, developers do not have the luxury of using a parallelisation API such as OpenMP for typical desktop, tablet and smartphone applications. The reason is not

only due to the lack of official implementation for object-oriented languages (C++ being the only object-oriented language supported), but primarily due to the control flow of GUI applications. The OpenMP model does not recognise the structure of GUI frameworks and therefore violates requirements of such GUI frameworks. Below is an example Java code typical of desktop applications with inherent parallelism:

```
public void actionPerformed(ActionEvent e){
    for (File file: list){
        process(file);
        done++;
        // GUI
        progressBar.setValue(done*100/todo);
    }
     // GUI
    label.setText("Job complete");
}
```

This example contains many points of interest. First, we notice an object-oriented solution (for example, the for-each loop traversing a collection of Files). Second, there are GUI computations being performed at various stages. This includes intermittent updates to the progress bar as each file is processed, in addition to the final message displayed when all files have processed. Finally, the code snippet is contained within an event handler, which will be executed in response to an action (e.g. a mouse click).

Contributions

This paper makes the following contributions:

Pyjama: It presents Pyjama, a Java compiler-runtime system for OpenMP-like directives.

GUI-awareness: It introduces GUI-aware Open-MP like directives that are essential for the parallelisation of object-oriented applications and GUI frameworks. The novelty lies in the notion of incremental concurrency, as described in this paper

Productivity with OpenMP: It demonstrates how Pyjama directives allow OpenMP to be used in a wider set of applications, in particular everyday interactive applications as found on desktop computers, tablets and smartphones. Considering that multi-core systems are now typically executing such interactive applications.

Performance Evaluation: We evaluate conventional OpenMP-like Pyjama directives using JGF benchmarks [5]. The evaluations benchmark Pyjama against other related systems and different Java concurrency practices. The GUI-aware directives are evaluated using a fractal-generation application with a GUI, where the responsiveness and the performance is evaluated.

2. RELATED WORK

Ongoing efforts in parallel computing comprise of developing language extensions and creating language constructs such as Parallel Task [9], making parallel compilers, developing new parallel framework models [20] and new languages such as Erlang [2]. These efforts focus on creating new parallel software; apart from them, the most important effort is being put in converting existing sequential programs to parallel forms by using compiler directives or library functions. This remains one of the easiest and clutter free approach and it involves a moderate learning curve. OpenMP directives, which are the most popular compiler directives, enable engineers to adapt this approach.

Java is a popular object-oriented language [19, 13] and is widely accepted for desktop and mobile application development, such as Android [1]. But there is no specification or binding to support OpenMP on Java programming language. No wonder that there have been attempts to merge the best of both the worlds by implementing OpenMP standards in Java. JOMP [3] and JaMP [12] are two such environments that support a subset of the OpenMP specification. The JOMP source-to-source compiler transforms OpenMP-like directives to multi-threaded normal Java programs by using Java native threading; JaMP is a distributed shared memory implementation that uses Jackal framework to translate the sequential code to parallel version. Parallel Java [11] is another approach which exposes unified APIs, rather than supporting OpenMP-like directives, to target shared memory model as well as clusters.

Furthermore, Java also provides support for concurrency and parallelism, mostly through SwingWorker and Java concurrent library (java.util.concurrent.*). Alternatively, the Java concurrent library exposes the Executor interface with a wide array of thread pool types, concurrent collections and data structures to achieve concurrency. Java 1.7 introduces the *fork-join* framework. The *fork-join* framework uses easy programming constructs and can help leverage multiple processors.

When it comes to supporting GUI awareness in OpenMP (Java or otherwise), we are not aware of any related work which introduces OpenMP-like directives for GUI applications.

3. BACKGROUND

An application with a graphical user interface needs to respond to multiple inputs from the user, at the same time performing multiple background processing, while all the time keeping the user interface responsive. In other words, performance means two things: utilising the underlying hardware potential through parallelism, and addressing a user-perceived performance through concurrency.

3.1 Structure of desktop applications

Other than the GUI aspect of applications mentioned earlier, figure 1 illustrates another distinguishing aspect of applications; *inversion of control* [8] is when the flow of control is dictated by a framework rather than the application code. In the case of desktop applications, the framework facilitates the communication with users. In comparison, batch-type programs have the control flow determined by the programmer; the application code defines the execution flow, with frequent calls to other frameworks (e.g. input/output libraries). The primary concept is that *events* are generated from within the framework code. Developers implement the necessary routines, known as *event handlers*, in the application code in response to the respective events. The control returns back to the framework upon completion of the respective event handler, namely to the *event loop*. In most GUI frameworks, this is performed by a dedicated thread named the GUI thread. In Java, this is more specifically
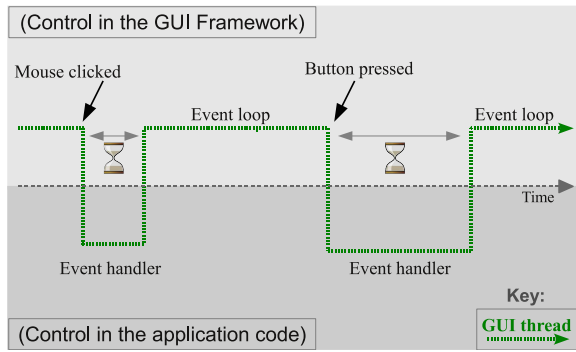
Figure 1: Control flow in a typical (sequential) desktop application. The application is processed by a single thread, known as the GUI thread (in Java this is also known as the Event Dispatch Thread). The application is unresponsive whenever control lies within the application code, rather than the GUI framework.

known as the Event Dispatch Thread (EDT).

3.2 Maintaining responsiveness

To maintain an interactive and responsive application, programmers must ensure the GUI thread minimises its execution in the application code. This is ensured by keeping event handlers as short as possible by off-loading a time-consuming computation to a helper thread. This allows the GUI thread to return to the event loop and respond to other potential events.

As the off-loaded computation is being executed, information typically needs to be presented to the user (e.g. progress updates and various messages). In most graphical toolkits, only the GUI thread may access the GUI components that communicate with the user. This means that helper threads must request GUI updates by posting an event to the GUI thread. The GUI thread consequently picks up the event and invokes the respective code to update the GUI. The reasoning behind this is because GUI toolkits are not coded thread-safe and must therefore only be accessed by the dedicated GUI thread.

This means that multi-threading is introduced solely for the purpose of achieving concurrency and just to maintain a responsive application. The performance improvement is solely for the *user perception*, and is likely to even increase the overall computation time due to the introduced overhead. No speedup is witnessed, even on a multi-core processor. In order to truly benefit from multi-core processors, the programmer needs to have parallelism in mind and take multi-threading further.

3.3 OpenMP and GUI threads

For developers wishing to take advantage of multi-core machines without multi-threading knowledge or experience, OpenMP serves as an easy, yet effective, solution. This is especially true for applications with obvious parallelism (e.g. loops without loop-carried dependencies). OpenMP involves an incremental parallelisation approach that keeps the serial version intact. It is a cross-platform and portable solution, where developers do not need to know of the underlying system. It is also a scalable software solution compared to
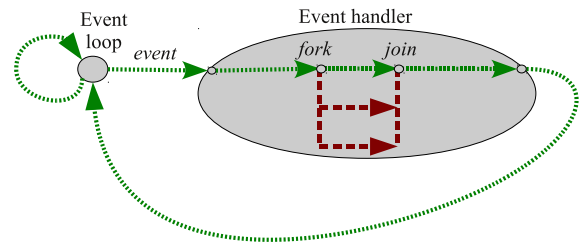

Figure 2: OpenMP's fork-join model. In GUI applications, OpenMP's fork-join model results in unresponsiveness because GUI thread assumes the role of Master thread in the parallel region rather than residing in the event loop.

manually using a threading library.

OpenMP uses a *fork-join parallelism* model, as illustrated in figure 2, which is a standard way of parallelising in shared-memory programs [15]. Initially only one thread, called the *master thread*, executes sequential portions of a program until it encounters a parallel construct, at which point it switches to parallel mode with the creation of multiple threads (known as the *fork*). This team of threads (the master thread and the created threads) execute the parallel region. At the end of the parallel region, execution control switches back to sequential mode by returning the control flow to the master thread and the other threads are destroyed or suspended (known as the *join*).

If the fork-join model is used in the application development, it essentially presents two problems shown in figure 2. First, the GUI thread assumes role of master thread and is immediately part of the thread team. As a consequence, the application becomes unresponsive throughout this time; it is not until the entire event handler completes that the GUI thread will return to the event loop. The second problem is that some GUI-related processing within the (now parallel) region will be executed by non-GUI threads. In addressing either of these issues, the programmer will be required to significantly restructure the code; such code restructuring goes against OpenMP's incremental parallelisation approach that maintains the original sequential code.

4. PYJAMA COMPILER-RUNTIME SYSTEM

Pyjama supports an incremental parallelisation approach that is suitable for GUI applications. It follows the OpenMP philosophy and its shared memory fork-join model. More importantly, it addresses the two essential problems identified in section 3.3: maintaining a responsive user interface during execution of parallel regions and allowing correct execution of GUI-related code within those regions. As in OpenMP, this is achieved incrementally while maintaining the original sequential code without restructuring it.

4.1 Directive syntax

Pyjama provides compiler directives which target the OpenMP 2.5 standards. There is no specification for OpenMP directives in Java, so Pyjama specifies directives formats which are close to the OpenMP specification. There are two salient points to be noted:

- Java does not support pragma, so there is no tradi-

tional way of using conditional compilation.

- Any non-compliant compiler will ignore the directives and treat the programs as normal sequential programs.

In keeping with above points, each Pyjama directive begins with "//#omp". A program line beginning with //#omp is treated as compiler directives by the Pyjama compiler and ignored as inline comments by the other compilers. Generic syntax is as follows:

//#omp directiveName[clause[[,]clause]..]

Alternatively, an object oriented syntax to support OpenMP-like directives can be designed using the Java annotations. However, a non-conforming compiler will not be able to identify such annotations and will generate compiler time errors. This behaviour is contrary to the OpenMP philosophy to enable the OpenMP code run on non-confirming compilers too, where the directives get ignored and serial behaviour of the code is preserved.

4.2 Conventional OpenMP directives

Conventional OpenMP directives include the *parallel* directive, *worksharing* directives, *combined* directives, *synchronisation* directives (*single*, *critical*, *barrier* and *atomic*) and *master* directive. We present some selected code examples to illustrate the syntax and usage of these directives, first of them is the customary *Hello World* example:

```
int nThID;
//#omp parallel private(nThID)
{
    nThID = Pyjama.omp_get_thread_num();
    System.out.println
        ("Hello World, I'm thread "+nThID);
}
```

As in the program above, a programmer can use OpenMP-like directives with the same expressiveness of OpenMP programming, in Pyjama. This introduces parallelism without the need to add major re-structuring. We illustrate the usage of a worksharing directive below:

```
//#omp parallel
{
    //#omp sections
    {
        //#omp section
        {
            System.out.println("1st part");
            task1();
        }
        //#omp section
        {
            System.out.println("2nd part");
            task2();
        }
    }
}
```

Furthermore, Pyjama supports the OpenMP synchronisation directives like *barrier*, *critical*, *atomic* and *ordered*. Here too, the these directives have identical semantics as that of OpenMP on C/C++.

4.2.1 Execution semantics of conventional directives

The execution semantics follow the same fork-join parallelism as that of OpenMP. Pyjama directives are syntactically close to OpenMP specifications. The code within parallel region is converted to explicitly multi-threaded code by Pyjama compiler, where the directive-encountering thread is treated as the *master* thread.

4.3 GUI directives

In addition to bringing directive based incremental parallelism to Java, our research aims at identifying the limitations of OpenMP for application development and to address them. The biggest limitation of using an OpenMP approach in a GUI application is that OpenMP has a fork-join model that violates GUI application structure. Consider the EDT inside an event handler, and it encounters an OpenMP construct. The master thread (MT) would be the EDT in OpenMP and MT would be part of the processing being done in OpenMP region (see Figure 2). But in a GUI application this is a problem, because the EDT will remain busy processing and in effect, it will block the GUI.

The thinkable solution would be a GUI-aware OpenMP solution. It would employ a thread-model (say *GUI-aware model*), which is indicated in the code by an extended OpenMP-like directive (more on that in section 4.3.1). This model will determine if the EDT is the master thread, and if so, it will create a *GUI-aware region*. The GUI-aware region will handle the execution on behalf of EDT and allow the EDT to return to the event loop. When execution of this region is completed, the model will notify the EDT; the EDT will return to the end of the region and continue execution. This way, the background execution will be processed by the GUI-aware region, and the EDT will remain free for event handling. But, this model seems to be breaking the master-thread model because the master-thread (EDT in this case), which encounters the directive, is let free and is not part of the processing. So, who is the master now?

To keep the master-thread model intact, a substitute thread (ST) is created by the model and the ST can act as the master for GUI-aware region. This model preserves the master-thread model by creating a ST and maintains responsiveness in GUI applications. Pyjama implements this model by introducing the GUI-aware model and introduces two new GUI-aware directives, *freeguithread* and *gui*.

Responsiveness can be maintained in the application using the *freeguithread* directive (more in section 4.3.1).The syntactic format of *freeguithread* directive is illustrated below:

//#omp freeguithread
structured-block

Application code can use this directive from the GUI thread (EDT in Java). If it is not used on a GUI thread, the directive is ignored and compiler generates a warning.

For responsiveness, there may arise a need to update the GUI when the application code is still busy in the background processing. This may be related to conveying the partial results of the background processing or it may be just a GUI update to convey the completion of background processing. It may be one of common tasks such a progress-bar update or message box update, or it may be conveying results of execution such as rendering a part of processed im-

age. This way, the application need not wait for the whole processing to complete. In application development, this is achieved by implementing a way to provide periodic updates to the GUI. Generally, it involves careful synchronisation methods or shared global flags. Additionally, a programmer will have to add major restructuring to spawn the computational work to other thread(s) and then again to execute GUI code, commonly encapsulated within runnables and posted to the GUI thread. These methods have their own limitations and complexities and make it difficult to involve any OpenMP-like programming. It also opposes the the OpenMP philosophy of maintaining the program's original sequential structure when the OpenMP directives are ignored.

Keeping into account the above perspective, Pyjama provides the *gui* directive. This directive is used to execute a region on the GUI thread and it shines when used in combination with the *freeguithread* directive. The format of *gui* directive is as below:

//#omp gui [nowait]
structured-block

4.3.1   Execution semantics of freeguithread *directive*

Pyjama's GUI-aware thread model addresses the responsiveness related bugs. The basic principle is that an application will not have the tendency to become unresponsive, or block, if the GUI thread is free to process user inputs. Pyjama's GUI-aware thread model incorporates this basic principle by using the *freeguithread* directive to begin a GUI-aware region. This region involves creating a substitute thread (ST) to replace the GUI thread in the region. The GUI thread is released to return back to the event loop, while ST performs the processing on behalf of the GUI thread. The end of the GUI-aware region serves as a continuation point and execution control is handed back to the GUI thread at this point. Once done, the GUI thread may continue execution in the same method or may return to the event loop. In effect, *freeguithread* introduces concurrency into the application by releasing the GUI thread back to event loop. Figure 3 illustrates Pyjama's GUI-aware thread model.

When the GUI-aware region is busy processing, it may start another GUI-aware region (i.e. another GUI-aware region begins while one is already executing). For example, in Java, suppose that EDT started a GUI-aware region in *actionPerformed( )* and since it remains responsive to more inputs, it performs another call to *actionPerformed( ).* In fact, this would be a common scenario in responsive applications. In such cases, the execution of GUI-aware regions are queued up and handled in the linear order. The execution semantics ensures that the same ST is used in all GUI-aware regions and no new threads are spawned. Effectively, this amounts to a responsive application with no impact on the overall performance.

4.3.2   Execution semantics of gui *directive*

Using the *gui* directive, a program can execute part of the code on the GUI thread from a background-processing region. This eliminates the need to maintain complexities of synchronisation or global memory to share information. The combination of *freeguithread* and *gui* directives enable
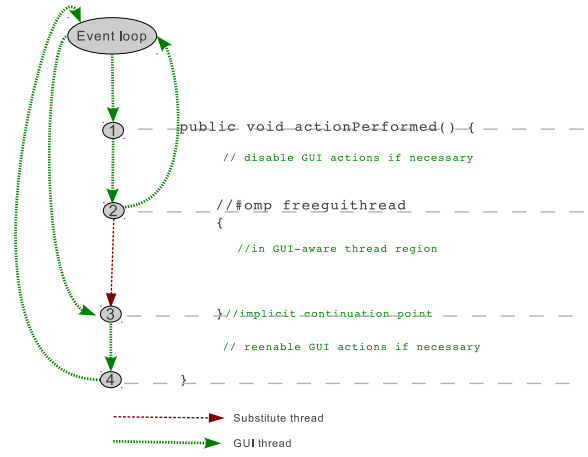

Figure 3: Pyjama's GUI-Aware thread model. Point 1. Event handler method begins (actionPerformed( )). Point 2. Directive *freeguithread* is used and GUI-aware region is created where ST takes control. Point 3. End of GUI-aware region, continuation point is inserted. Control given back to GUI thread. Point 4. End of event handler, GUI thread returns to event loop.

programmers to achieve responsive application development and the threading model still adheres to the OpenMP model.

The program below shows a typical example of event handling in application code, and how these GUI directives can be used.

```
public void actionPerformed(ActionEvent e)
{
    //#omp freeguithread
    {
        for(File file: list){
            processImage(file);
            done++;
            //#omp gui
            progressBar.setValue
                (done*100/todo);
        }
    }//implicit continuation point
}
```

Let us look into the contents of this program and understand how GUI-aware directives aid application development. The program needs to perform a potentially time consuming computation: process a list of image files. Under normal execution, the GUI thread would become busy and the application is unresponsive to any new inputs.

The programmer may use the *freeguithread* directive in such a scenario. At this juncture, a substitute-thread is created and it assumes the role of master-thread and the GUI thread can return back to handle other events. An implicit continuation point is placed after the end of *freeguithread* region. When execution of the region is completed, i.e. the substitute-thread reaches this point, control is given back to the GUI thread. To provide feedback to the user, e.g. in

form of progress bars, wait icons etc., the *gui directive* is used here. In this program, the *gui* region is executed on the event loop and progress bar is updated.

The *gui* directive by default contains a barrier (this may be compared to Java's *SwingUtilities.invokeAndWait()*) Nevertheless, there may be scenarios when execution control need not wait at the barrier after processing *gui* directive region (this may be compared to Java's *SwingUtilities.invokeLater()*). The barrier for the *gui* directive may be dropped by using *nowait* clause, as shown in code snippet below:

```
//#omp freeguithread
{
    for(File file: list){
        processImage(file);
        done++;
        //#omp gui nowait
        progressBar.setValue
            (done*100/todo);
        doSomeOtherTask();
    }
}//implicit continuation point
```

In summary, the continuation point is always enforced for the *freeguithread* directive, but the barrier for the *gui* directive is optional. It can be seen that the GUI-awareness model of Pyjama introduces concurrency and addresses two quintessential requirements of the application development: responsiveness and application GUI update from background process. The incremental use of these directives and resultant introduction of concurrency leads to the practice of *incremental concurrency* with Pyjama. These directives combine the expressiveness of OpenMP usage and the essence of object-orientation.

4.4 Combined GUI and conventional directives

The GUI-aware directives can also be combined with conventional OpenMP-like directives. For example, we can add a small change in the example code shown for the GUI-aware directives, as shown below, and it will make the GUI-aware thread region parallel:

```
//#omp parallel freeguithread
{
    ...
```

Here, the programmer uses a *parallel* directive to spawn a number of threads. As per OpenMP threading model, the substitute thread becomes master-thread (due to the *freeguithread* directive) and shares the work. The *freeguithread* directive introduces concurrency for application responsiveness, conventional OpenMP-like directives are used to introduce parallelism.

5. IMPLEMENTATION

In this section, we describe the Pyjama compiler and a brief overview of how directives are supported. In the last part of the section, we will discuss the Pyjama runtime.

5.1 Pyjama Compiler

The Pyjama compiler is a hybrid implementation. On one hand, it handles *program normalisation*, *program optimisation* and generates destination code in the same language as the source code (*.javamp files get converted to *.java), thus falling into rephrasing category [16]. On the other hand, it provides *program translation* of directives by converting source code into explicitly multi-threaded destination code, thus falling into translation category. In the following sections we discuss the important units in the Pyjama compiler.

5.1.1 Front End

An OpenMP compiler must recognise and validate the directives and translate them according to the semantics [6]. It must also compile the associated language. The Pyjama compiler front-end performs these tasks. Syntactical errors in directives are identified here and notified. After that, normalisation of directives is done, for example, splitting of combined directives into worksharing/GUI directives, conversion of *sections* directive and *single* directive into parallel loop and loop with one iteration respectively to simplify the translation.

5.1.2 Back End

This unit of compiler transforms the sequential regions into multi-threaded code. Pyjama uses its runtime library to add the thread pool implementation, scheduling mechanisms (as per schedule clause: *static*, *dynamic* or *guided*), work queuing and notify mechanism (in the case of gui-aware regions).

Code generation involves moving the syntactical code contained in a parallel region to a new method (called *worksharing* method) and is replaced by a Pyjama parallel region. This parallel region enques the worksharing method on a task-queue. A barrier is placed after this parallel region to implement the join model. The runtime environment retrieves methods from the task-queue and uses Java reflection to execute the worksharing method.

Similarly, for GUI-aware directives, the code present in the *freeguithread* directive is moved to a new method and is enqueued to be executed in parallel. For a *gui* directive, Java's *SwingUtilities.invokeAndWait( )* (or *SwingUtilities.invokeLater( )* in the case of a *nowait* clause) is used, with the user's code being moved to the *run( )* method. The code example below illustrates such transformation (the user code within Pyjama directives is shown in *italics red*, the generated code is shown in **bold blue**):

```
/input code
public void actionPerformed(){
    //#omp parallel freeguithread
    {
        processImage(file);
        //#omp gui
        updateProgressBar();
    }
    showImage(file); // user code
}
```

```
//transformed code
public void actionPerformed(){
    _ompEnqueue("_omp_workRegion_0",
       "_omp_cont_pt_0");
}

private void _omp_workRegion_0(){
    processImage(file);
    if(false == EventQueue.
        isDispatchThread()){
        SwingUtilities.invokeAndWait(
            new Runnable() {
            public void run() {
                updateProgressBar();
            }
        });
    }
}

private void _omp_cont_pt_0(){
    showImage(file); // user code, remains intact
}
```

Two important aspects of these transformations are the adherence to OpenMP semantics for OpenMP-like directives and object-oriented approach for GUI-aware directives. Data variables in data clauses are managed as mentioned in [6].

5.2 Pyjama Runtime

Pyjama provides runtime support for the thread-pool and queue implementations and it implements runtime routines conforming to OpenMP 2.5. The execution environment routines and timing routines are provided through static methods of a class called *Pyjama.* They follow the OpenMP naming, e.g. *Pyjama.omp_get_num_threads()*, *Pyjama.omp_set_num_threads( )*, and more.

6. PERFORMANCE EVALUATION

The Pyjama system was evaluated using a subset of the JGF benchmarks [5]. We used the *Series Benchmark* from Section II of the benchmark suite, *MonteCarlo Benchmark* and *RayTracer Benchmark* from Section III of the benchmark suite. For measuring the speedup, the baseline was the speedup using a single CPU for the sequential version of the benchmarks. The performance was compared to JOMP [3] and benchmarks with Java native threads [17] and we used the same data sets (size A, size B, size C, where C is the largest and A the smallest) as used by JOMP benchmarks. The benchmarks were run on hardware with 4 Quad-Core Intel Xeon processors, total of 16 cores, running at 2.4GHz with 64GB of RAM.

To evaluate the GUI directives, we implemented a fractal-generating application which renders Mandelbrot. The performance was tested against sequential, Java thread, Swing-Worker and ExecutorService (cached and fixed pool) versions of the same application.

6.1 JGF Section II Benchmark

Section II of the JGF benchmarks consists of simple kernels which implement the commonly occurring computations. They provide us an opportunity to evaluate the scal-
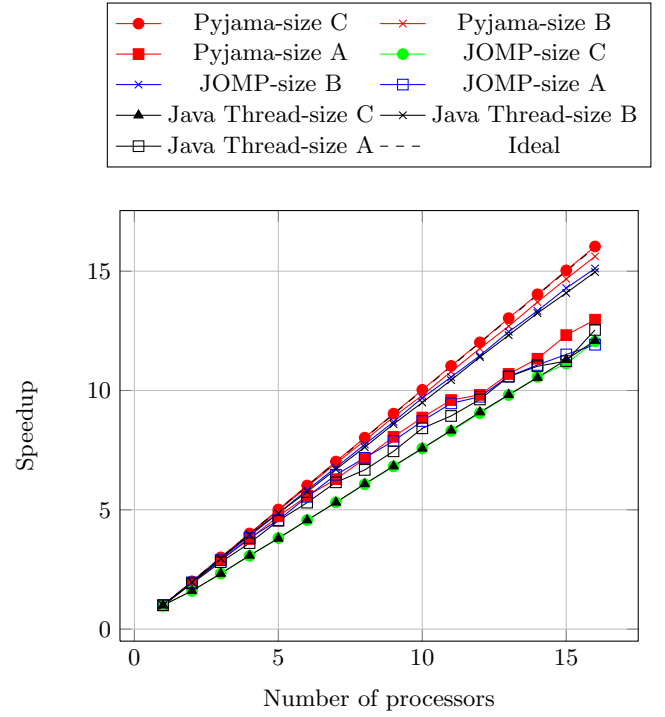

Figure 4: Series Benchmark

ability and the performance.

6.1.1 Series Benchmark

This set of benchmarks computes the first N Fourier coefficients of the function $f(x) = (x + 1)^x$. The computationally intensive main loop is easily parallelised using *parallel for.* Figure 4 exhibits the Pyjama performance over three data sets, and compares them to JOMP and native threads.

It is seen that a near linear speedup and scalability is achieved. Using varying sizes of data shows that the performance improves as the data size increases. This can be attributed to the fact that the parallelisation overhead gradually becomes smaller when compared to the increased performance, with the increasing data size. When comparing the three system, Pyjama environment is better performing. It should be noted that the Pyjama and the JOMP systems require mere additions of the directives, whereas usage of native concurrency involves restructuring efforts.

6.2 JGF Section III Benchmark

Section III of the JGF benchmark suite represents the real world applications. These benchmarks do not consist of many "embarrassingly parallel" code. This suite of benchmark provides the opportunity to evaluate conventional construct like the *parallel*, the *parallel for* construct and the *critical* directives along with the data clauses (*private*, *shared*) and the *reduction* clause.

6.2.1 MonteCarlo Benchmark

MonteCarlo benchmark implements a financial application simulation, where it uses the Monte Carlo methods to calculate product prices deduced from an underlying resource. Figures 5 and 6 illustrate the performance comparisons of Pyjama implementation with JOMP and Java
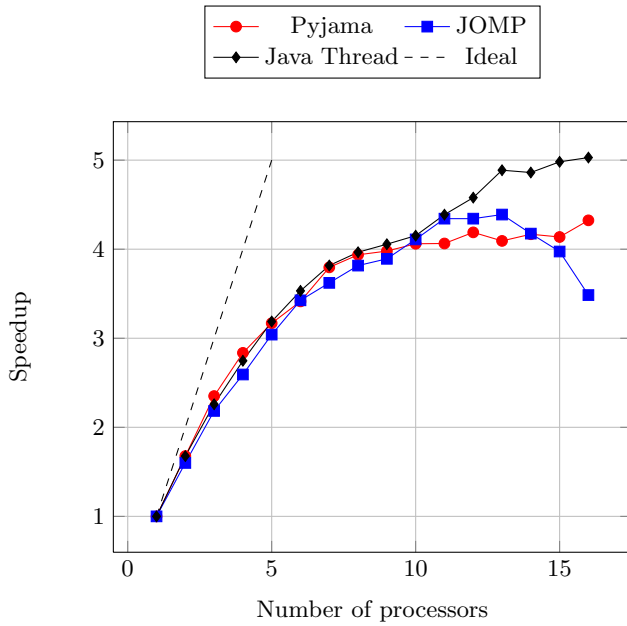
Figure 5: MonteCarlo Benchmark - Size A

Figure 6: MonteCarlo Benchmark - Size B

threads.

The results show that the speedup is linear up to 8 processors and then levels up. The evident reason behind this is the way in which Monte Carlo method is typically parallelised; the results are computed in a parallelised loop but synchronisation (using the *critical* directive) is needed to store the partial results. Nevertheless, the processing with the larger data set is able to overcome the loss due to the expensive syncronisation and provides a better performance.

6.2.2 RayTracer Benchmark

This benchmark provides the opportunity to evaluate the usage of the data clauses and the *reduction* clause, apart from the conventional directives.

RayTracer benchmark implements a ray tracer that renders a scene containing 64 spheres and is rendered at N x N pixels. The main loop is parallelised using the Pyjama directive and each thread renders a part of the scene. Figures 7 and 8 show the benchmarking results.

In this evaluation, the trends are similar to the Monte-Carlo benchmark but the speedup increases for the large data sets. There is little differentiating the three environments but increasing data sets presents an opportunity to compensate for parallelisation overheads. However, Pyjama and JOMP do not perform as good as Java thread, and this can be attributed to the presence of global shared copy of scene and environment data. Java thread benchmark has private copies of these and allow for sequential optimisations.

6.3 Fractal Application

To evaluate the GUI-aware directives, we developed a Fractal generating application. Specifically, this application evaluates the *freeguithread* and the *gui* directives for responsiveness and performance. The tests were repeated with different load granularity.

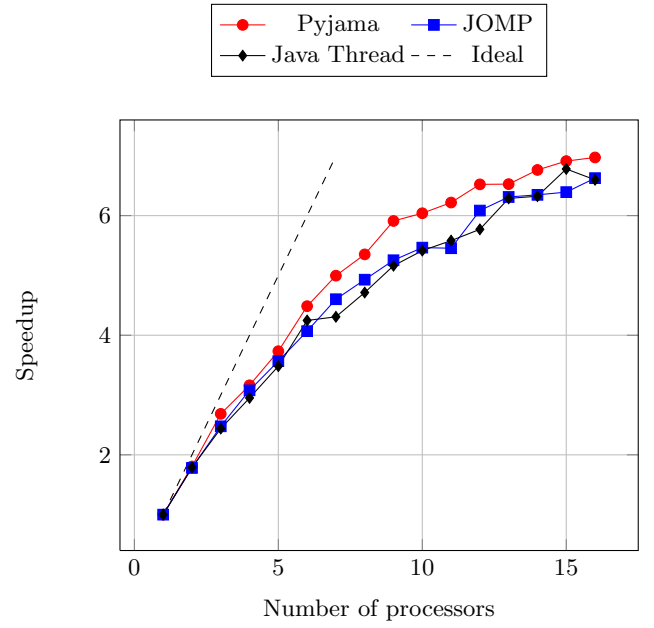This application's UI renders a Mandelbrot pattern on the
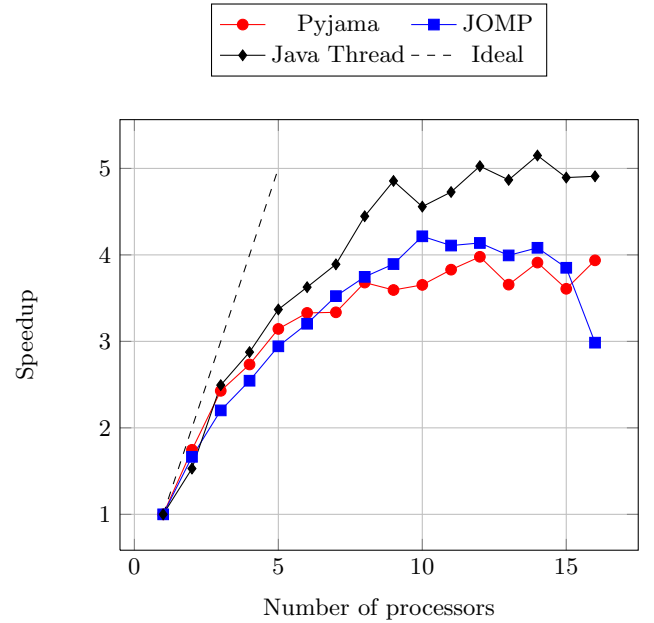

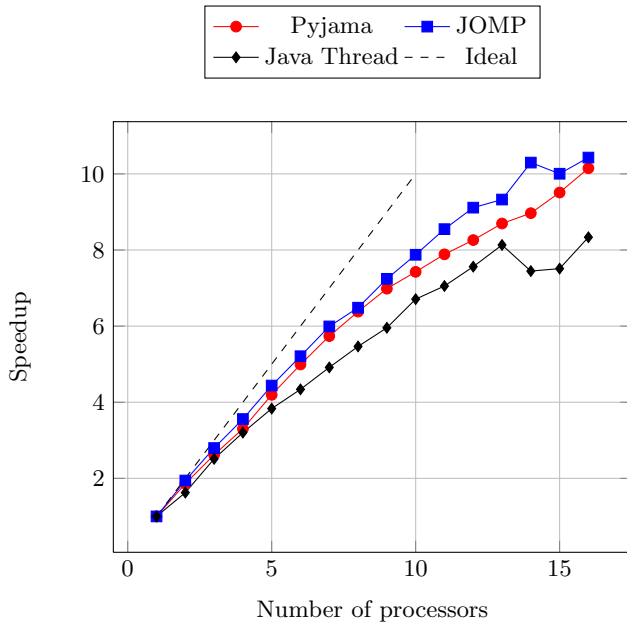
Figure 7: RayTracer Benchmark - Size A
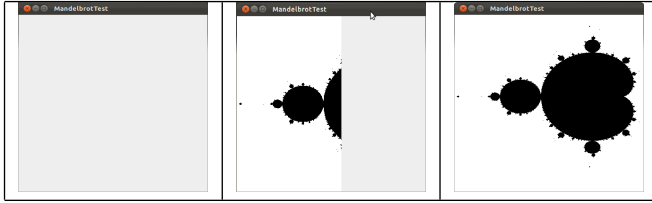
Figure 8: RayTracer Benchmark - Size B

Figure 9: Snapshots from Fractal Application. Left to Right: 1.Sequential Behaviour, application is blocked till all calculations are completed. 2. Pyjama Application Behaviour(same as SwingWorker), calculations and rendering are done simultaneously. 3. Render complete.

Figure 10: Performance in Fractal Application for coarse-grained load.
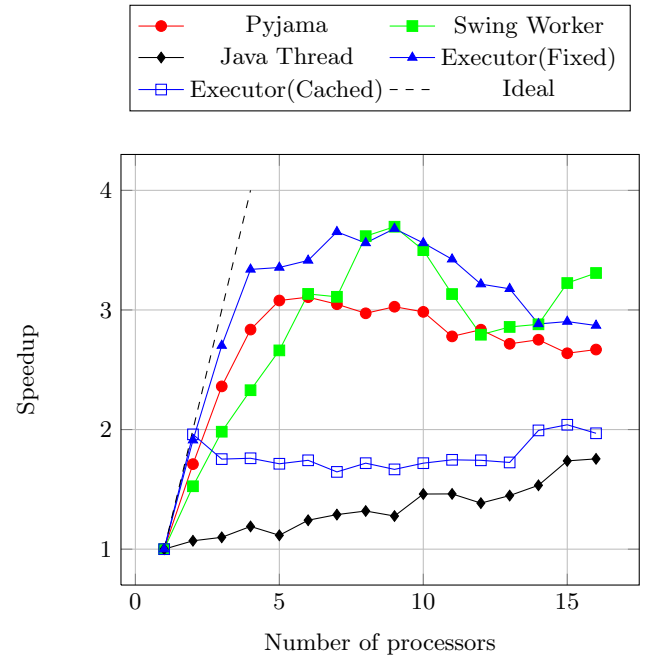
Figure 11: Performance in Fractal Application for very fine-grained load.

screen. The back-end consists of a central loop whose each iteration represents a column of the rendering, and the computations done in the iteration is the workload. In figure 9, we present a small gallery of snapshots from the application. The application involves creating a 400x400 screen panel at 10000 iterations for each pixel rendering. The application GUI involves rendering a single complete column of marked pixels in one task. We implemented sequential, Java thread, ExecutorService (fixed and cached pool) and SwingWorker versions of the same application. Thereafter, we incrementally added GUI directives to sequential version and got a Pyjama version and observed the performance, as illustrated in figure 10.

The first set of tests involved course grained load with 10000 iterations per pixel. We repeated the benchmarks with a 10000X10000 screen panel with 20 iterations per pixel. In this case, we have more tasks but finer grained per pixel. Figure 11 illustrates the performance.

These benchmarks were concentrated on observing application responsiveness and evaluating the ease of incremental parallelisation and incremental concurrency. As shown in figure 9, the responsiveness in Pyjama version is at par
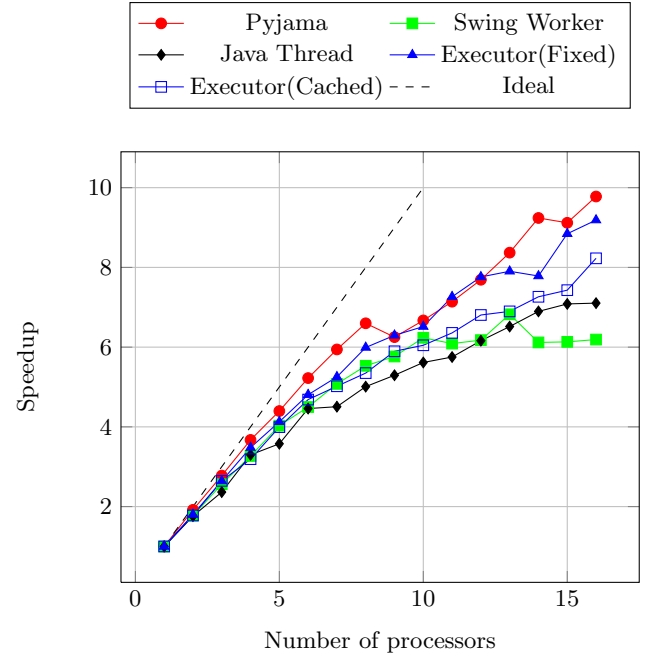
with Swing Worker implementation. The GUI was updated continuously and the application was not blocked. The application was responsive to user actions, such as the user action to close the application.

None of the systems scale very well for a very fine grained load. Java native threads produce the least speedup and this can be due to the repeated thread creations. In thread pool based systems, this overhead is minimised. For the same reason, the other systems perform better. Nevertheless, even for the other systems the overheads of the context switching between the threads, the contentions and the scheduling remain expensive. Pyjama produces better performance than Java threads, cached pool Executor service and SwingWorker.

From the productivity aspect, converting the sequential code to the Pyjama version did not need require any code restructuring or new implementation, as shown in the code below:

```
public void generateSequential(BufferedImage image){
    ...
    //#omp parallel for freeguithread
    for (int i = 0; i < width; i++)
    {
        Boolean[] columnPixels = calculateColumn(i);
        //#omp gui
        colorColumn(i, columnPixels);
    }
}
```

7. CONCLUSION

We presented an OpenMP-like directive support for Java and provided the GUI application development specific extensions. We presented methods to introduce concurrency for application responsiveness using GUI-aware directives and to introduce parallelism using conventional directives. This provides a complete solution to create parallelised GUI applications. The programmer can generate object-oriented application code by using GUI directives and can save code restructuring and re-implementation efforts.

We discussed the Pyjama compiler and runtime and illustrated our implementation. Pyjama was used in JGF benchmarks and in application development and we presented the results. The performance was found comparable to JOMP environment and to traditional parallelisation tools such as Java thread, ExecutorService (fixed and cached pool) and SwingWorker. In the future, we plan to upgrade the Pyjama runtime for more efficient handling of fine-grained parallelism and introduce more object-oriented features in the spirit of OpenMP.

8. REFERENCES

[1] Android, Google Inc. *http://developer.android.com/guide/basics/what-is-android.html*.

[2] J. Armstrong. *Programming Erlang Software for a Concurrent World*. The Pragmatic Bookshelf, August 2007.

[3] J. M. Bull and M. E. Kambites. JOMP; an OpenMP-like interface for Java. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 44–53, New York, NY, USA, 2000. ACM.

[4] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for scientific applications. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 97–105, New York, NY, USA, 2001. ACM.

[5] J. M. Bull, L. A. Smith, M. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency, Practice & Experience*, 12(6):375 – 388, 2000.

[6] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[7] M. Creeger. Multicore CPUs for the masses. *Queue*, 3(7):63–64, 2005.

[8] M. Fayad and D. C. Schmidt. Object-oriented application framework. *Communications of the ACM*, 40(10):32 – 38, 1997.

[9] N. Giacaman and O. Sinnen. Parallel task for parallelizing object-oriented desktop applications. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –8, april 2010.

[10] N. Giacaman and O. Sinnen. Object-oriented parallelisation of Java desktop programs. *IEEE Software, Software for the Multiprocessor Desktop: Applications, Environments, Platforms*, 28(1):32–38, Jan-Feb 2011.

[11] A. Kaminsky. Parallel Java-a unified API for shared memory and cluster parallel programming in 100% Java. In *Parallel and Distributed Processing Symposium, 2007.IEEE International*, pages 1 – 8, march 2007.

[12] M. Klemm, M. Bezold, R. Veldema, and M. Philippsen. JaMP: an implementation of OpenMP for a Java DSM. *Concurrency & Computation: Practice & Experience*, 19(18):2333 – 2352, 2007.

[13] Language Popularity Index. *http://lang-index.sourceforge.net/*.

[14] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, May 2005.

[15] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. Mc Graw Hills, 1st edition, 2004.

[16] E. Renault, C. Ancelin, W. Jimenez, and O. Botero. Using source-to-source transformation tools to provide distributed parallel applications from OpenMP source code. In *Parallel and Distributed Computing, 2008. ISPDC '08. International Symposium on*, pages 197 –204, july 2008.

[17] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, pages 8–8, New York, NY, USA, 2001. ACM.

[18] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[19] Tiobe Programming Index. *http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html*.

[20] Typesafe Inc. *Akka Documentation*, release 2.1-snapshot edition, 2012.