Libraries and Learning Services

University of Auckland Research Repository, ResearchSpace

Version

This is the Accepted Manuscript version. This version is defined in the NISO recommended practice RP-8-2008 http://www.niso.org/publications/rp/

Suggested Reference

Giacaman, N., & Sinnen, O. (2011). Object-Oriented Parallelisation of Java Desktop Programs. *IEEE Software*, *28*(1), 32-38. doi:10.1109/MS.2010.135

Copyright

Items in ResearchSpace are protected by copyright, with all rights reserved, unless otherwise indicated. Previously published items are made available in accordance with the copyright policy of the publisher.

© 2011 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

For more information, see General copyright, Publisher copyright, SHERPA/RoMEO.

Object-Oriented Parallelisation of Java Desktop Programs*

Nasser Giacaman and Oliver Sinnen
The University of Auckland, New Zealand
ngia003@aucklanduni.ac.nz, o.sinnen@auckland.ac.nz

Abstract

Developing parallel applications is notoriously difficult, but is even more complex for desktop applications. The added difficulties are primarily because of their interactive nature, where performance is largely perceived by its users. Desktop applications are typically developed with graphical toolkits that in turn have limitations in regards to multi-threading. This article explores the structure of desktop applications, the limitations of the threading model and walks the reader through the parallelisation of a desktop application using object-oriented and GUI-aware concepts.

Motivation

Parallelisation is the process of decomposing a large computation into smaller parts, and consequently executing those smaller parts simultaneously to reduce the overall processing time. Despite the potential performance benefits, parallel computing has long been a programming nightmare for software developers. Theoretical challenges include requiring developers to envision the original problem into smaller, somewhat independent, sub-problems and then carefully analyse the dependences amongst those pieces. Even when this has been envisioned, the practical challenges of implementation magnifies the difficulty; to name a few, this includes implementing synchronisation mechanisms to ensure logically correct ordering, scheduling of the sub-problems and of course countless hours of debugging.

So, what about the world of *desktop parallelisation*? We are no longer talking about parallelising "trivial" scientific and engineering applications that possess large amounts of obvious and repetitive inherent parallelism (it is easy to envision the smaller sub-problems in these applications). We are now in the realm of

*To appear in a special issue of IEEE Software January/February 2011, Software for the Multiprocessor Desktop: Applications, Environments, Platforms

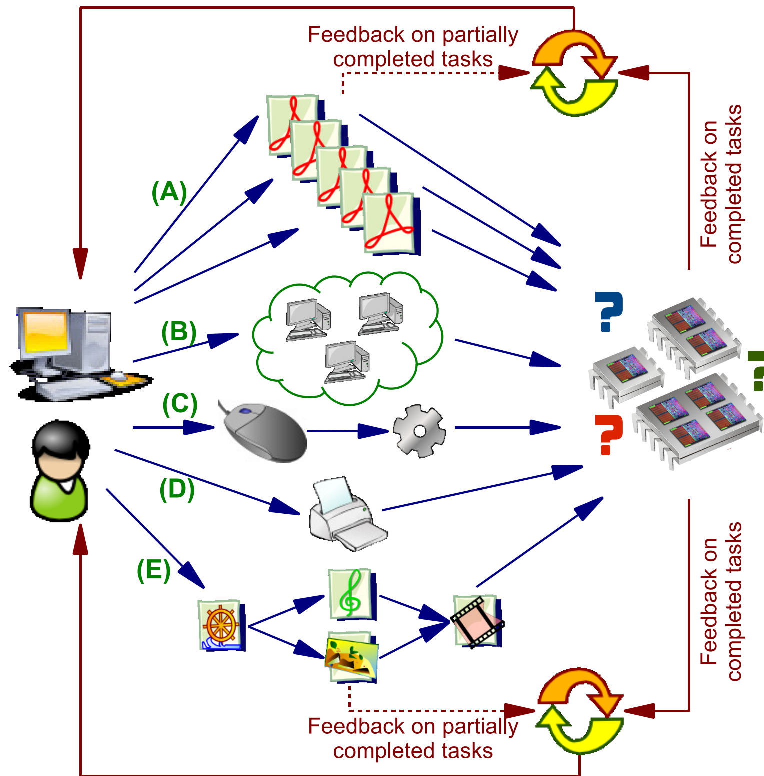


Figure 1: Desktop applications involve different types of tasks that require different implementation approaches, and it is also unknown on what system the application will run on.

irregularly structured computations with short runtimes. To add further complication, these applications will be executed on non-dedicated and unknown target systems (is the system a uni-processor? Quad-core? ... Many-core processor?). The more effort that developers invested in expressing the inherent parallelism of the problem, then the more effort that will be required to realise it: more sub-problem decomposition, more code restructuring, more synchronisation and more debugging. If any less effort is invested in realising the inherent parallelism, then the performance is punished accordingly.

Figure 1 illustrates the typical scenario of desktop applications. The idea is that a user interacts with the application in order to perform various tasks. Some tasks might be executed once only, while another task might be executed multiple times on different data elements (such as processing a directory full of files (A)). Some tasks tend to be computationally intensive, while others are input/output bound (such as an internet search (B), waiting for user input (C) or printing (D)). Some tasks may be executed independent of other tasks, while other tasks might have ordering constraints that depend on the completion

of other tasks (E). We categorise this into different types of tasks, each with different behaviours that demand different handling. To ease the parallelisation process, the first step is unifying these different task concepts into the same model.

Unfortunately, it gets even more complex for desktop applications. To complicate matters, users generally expect some sort of feedback on the executing tasks; this even includes intermittent updates of partially complete tasks (such as updating a progress bar). Consequently, the performance of a desktop application is primarily user-perceived. We want a responsive and interactive application, even if the application is to be executed on a single processor. Since desktop applications are user-driven (unlike batch-type applications), the graphical and interactive nature of these user interfaces contributes tremendously to the challenges. Finally, since developers cannot determine the system specifications that their applications will execute on, dynamic runtime support that adjusts to hardware is vital.

So, how do we continue about easing the process of parallelising desktop applications? Object-oriented languages dominate [1] the development of desktop applications. The benefits of parallelisation must be realised in the realm of such high-level languages, without resorting to languages like C or Fortran. Even though these low-level languages may be speed-efficient, large desktop applications demand the software engineering benefits of object-oriented languages.

This paper will address these problems and challenges by discussing the parallelisation of an object-oriented desktop application with a *graphical user interface* (*GUI*). We start by looking at the rarely discussed challenges that are unique to GUI desktop applications, namely the user interactivity, the limitations of the graphical frameworks and the large variety of target systems. As most desktop applications are written in object-oriented languages, the parallelisation must be performed in these languages. The presented parallelisation approach, called Parallel Task, is based on a unified task concept that integrates all common concurrency types.

The anatomy of desktop applications

Before attempting to parallelise desktop applications, we must understand their external and internal composition. The external features are the most familiar to us, which includes numerous visual input components (e.g. buttons, text fields) and output components (e.g. labels, progress bars). It is this distinguishing GUI that enables the desktop application to interact with its users.

What about the internal structure of desktop applications? The most vital organ is the *event loop* (part (1) of figure 2), which is responsible for reacting to *events* (e.g. a mouse click) by dispatching them to the appropriate *event handler* (part (2) of figure 2). The *GUI thread*, also known as the *event dispatch thread* (*EDT*) in Java, is solely responsible for anything GUI-related: from the external display of visual components, to the internal management of events.

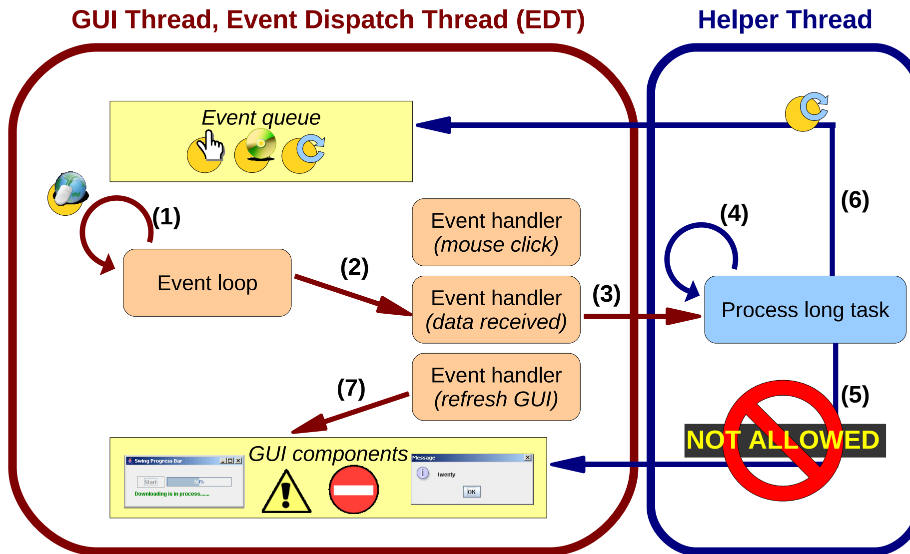


Figure 2: Structure of a Multi-threaded GUI desktop application.

No other thread is allowed to perform these actions – a restriction common to most desktop/GUI frameworks [2, 3]. Consequently, if the event loop is not processing events in a timely fashion, the application will become unresponsive, "freeze" and frustrate users.

To avoid such inanimate behaviour, multi-threading has long been necessary for GUI applications to create responsiveness. On single processor systems, multiple threads time-share the processor and thereby create concurrency. The computation is off-loaded to some other thread (part (3) of figure 2) in order to allow the GUI thread to return to the event loop. Both threads then share the single processor, but none is fully stopped. While the helping thread executes the off-loaded computation (part (4) of figure 2), it is not allowed to directly access any of the GUI components (part (5)) since the GUI components are not thread-safe (recall that the GUI thread is solely responsible for anything GUI-related). As a result, events must be posted to the GUI thread (part (6)) that will in turn be handled by the GUI thread (part (7)).

So, when multiple processors came into play, it felt quite natural to not only use threads for "virtual" concurrency and responsiveness, but for real parallel execution, where the threads are executed by different processors. Unfortunately, the concept of threads is ill-fitted to the diverse demands of parallel computing. Off-loading a computation to another thread is not enough; the computation needs to be *further divided* and distributed to *multiple* threads to keep all the available cores busy. The problem of managing threads is elevated and will be a lot worse when intricate synchronisation is necessary amongst the sub-tasks to ensure a logically correct execution of the original computation. In

other words, desktop parallelisation incorporates all the challenges of generalised parallel computing in conjunction with the challenges of developing responsive desktop applications.

The problem with existing tools

Threads have been an integral part of Java since its initial release. Consequently, parallelising a GUI application manually using Java Threads has been the norm. There are two primary reasons why this model is unsuitable for parallelising desktop applications. The first is that the intended conceptual purpose of a Thread is to fork a new thread of execution at a particular point in the program. As such, most independent sub-problems do not necessarily demand a new thread of execution. Rather, we only wish to express that such a computation may safely be performed asynchronously. In other words, we merely wish to denote this computation as a potential *task*, as opposed to enforcing a new thread of execution for it.

The second reason the threading model is undesirable is because of performance consequences. If too few threads are created, then not enough parallelism is introduced to exploit the number of cores. Conversely, if an excessive number of threads is created, then this degrades performance due to resource contention and scheduling overheads. Even with performance issues aside, the threading model reduces code legibility in that the code is migrated to new threads. Programmers must now manually manage any dependences amongst the sub-computations. Not only is this error prone, but it introduces coupling amongst the otherwise independent tasks.

For these reasons, modern parallelisation tools have opted for a tasking model as opposed to the traditional threading model: programmers express independent code snippets as tasks, and the runtime system of these tools will manage the task scheduling. But in most cases, such as Java's SwingWorker and ForkJoinTask, these modern tools are only improvements on the performance level. Programmers must still migrate code, implement dependence handling amongst the tasks and avoid I/O bound tasks. Outside of Java, other modern parallelisation tools include Cilk++, OpenMP task, Intel TBB, Apple's GCD, X10 and the .NET TPL. While these approaches are a huge step forward from a manual thread based parallelisation, many of these approaches are not truly object-oriented and often involve add-ons that are not designed into the language.

More importantly, when it comes to parallelising desktop applications, the primary problem is that none of these tools take into account the structure of GUI applications. As a result, the programmer is still left with the responsibility to ensure GUI computations are only performed on the GUI thread and that the GUI thread remains free. Furthermore, implementing dependences between tasks becomes the programmer's responsibility. This results in a high amount of coupling amongst (otherwise independent) tasks and tangling of parallelisation concerns with the actual business logic. The tangling and coupling reduce the

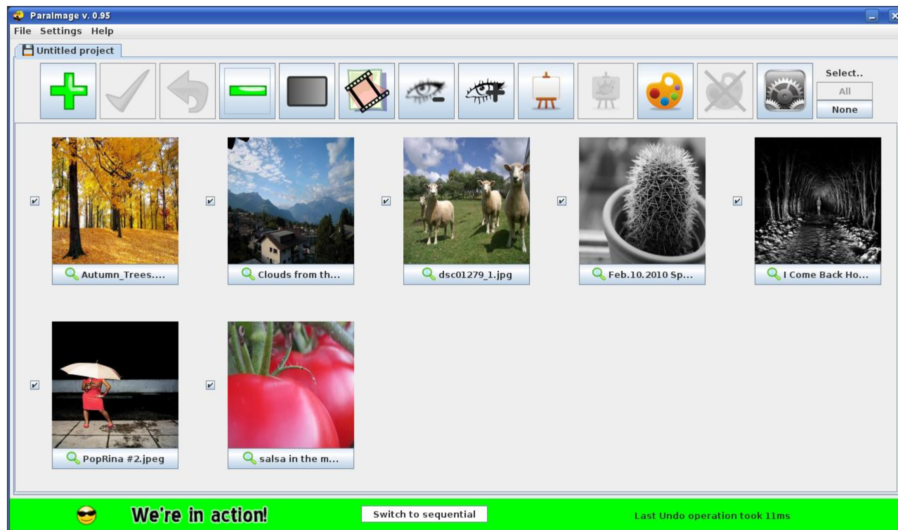


Figure 3: ParaImage, an application developed using ParaTask.

amount of code reuse: a principle important to both software engineering and object-oriented programming.

Parallelising desktop applications with Parallel Task

To address the above described problem we proposed Parallel Task (ParaTask for short) for the parallelisation of object-oriented desktop applications [4]. ParaTask is a parallelisation tool that consists of a source-to-source compiler and supporting runtime system to manage tasks. Although ParaTask is motivated by standard parallelisation concepts, ParaTask contributes by unifying these parallelisation concepts into an object-oriented environment. Programmers introduce concurrency with a single keyword, and the different task concepts are integrated into one model. ParaTask also supports an intuitive approach to dependence handling and has the unique feature of focusing on GUI applications. We will walk through the parallelisation of a GUI application and introduce the various ParaTask features as they are used. The example application, ParaImage, provides various functionality such as an online image search and image editing. Figure 3 shows a screenshot of the image editing project in ParaImage.

Defining and invoking tasks

As discovered above, we would like a simple task solution. Since we are focusing on object-oriented applications, we must decide on how we express tasks: using methods or objects? If we opt for objects to represent tasks, this will not address the programming difficulties of threads since developers are still required

to restructure and migrate the code. For this reason tasks in ParaTask are encapsulated at the method level and defined by the `TASK` keyword. Below is the definition of a task that performs an edge-detection on an image:

Defining a task

```
TASK public Image edgeDetectTask(Image i) {
  // detect the edges
}
```

The event handler below shows the invocation of this task:

Invoking a task

```
public void actionPerformed(ActionEvent e) {
  ...
  for (Image image: selectedImages) {
    TaskID<Image> result = edgeDetectTask(image);
    ...
  }
}
```

Invoking a `TASK` is essentially the same as invoking a standard sequential method, except that it executes in parallel to its caller. Because of this asynchronous behaviour, we will generally need a handle on the task invocation should we wish to synchronise with its completion (for example, to access the return result). The `TaskID` serves this purpose and is essentially a *future* that represents a task invocation.

The `actionPerformed` method is an event handler, and as such, is performed by the EDT. In a sequential implementation, the EDT would execute the edge detection in its entirety (the application "freezes" during this time). However, in the parallel mode, the EDT only *enqueues* the task and returns to the event loop. The tasks are then scheduled for execution by a team of threads; ParaTask automatically creates and manages an ideal number of threads to keep each of the processing cores busy. This means that invoking multiple tasks is a lot more efficient than creating a thread for each computation.

You have probably figured out our next problem. Now that the task has been off-loaded from the EDT, how will we be able to know when the task completes (to update the results to the user)? The first thing that comes to mind might be to block on the `TaskID`:

Accessing the task result (blocking)

```
Image i = result.getResult();
```

If the task has already completed by this stage, then there is no problem since the result is already available. Otherwise, the thread invoking this call blocks until the task completes. This behaviour is clearly unacceptable for the EDT. We instead need a way for the EDT to be notified when the task completes, i.e. a non-blocking notification solution:

Non-blocking task synchronisation

```
public void actionPerformed(ActionEvent e) {
  ...
  for (Image image: selectedImages) {
    TaskID<Image> result = edgeDetectTask(image)
      notify(updateGUI(TaskID));
    ...
  }
}
```

Even though we off-load the filter computation to another thread, the update of the GUI must still be performed by the EDT. By using the `notify` clause, the EDT returns to the event loop and later gets informed when the task completes. The methods specified inside the `notify` clause and the task definition remain decoupled:

Updating GUI after a task completes

```
public void updateGUI(TaskID<Image> id) {
  Image thumbnail = id.getResult();
  // display thumbnail, update progress bar...
}
```

Dependences

What happens when a newly invoked task is dependent on previous tasks? For example, assume the user wishes to perform multiple filters on a single image (the filters should have an accumulating effect). Maybe the user applies an edge detection filter and then immediately applies a blur filter twice. In this case, there is a dependence between the tasks applied on the same image. Using standard threading libraries, programmers would have to manually code for such dependences using synchronisation mechanisms such as wait conditions. In addition to being error prone, the disadvantage with this approach is that tasks would become coupled with each other. For such cases, the `dependsOn` clause is suggested:

Specifying dependences between tasks

```
1 for (Image image: selectedImages) {
2   TaskIDGroup history = historyMap.get(image);
```

```
3
4     TaskID result = blurTask(image)
5       notify(updateGUI(TaskID))
6       dependsOn(history);
7
8     history.add(result);
9   }
```

Each image has a history of filters (in the form of `TaskID`s, line 2). Whenever a new filter (i.e. a new task) is applied to an image, then that filter will only be applied once the previous filters (i.e. tasks) have completed on the image (line 6). Otherwise, without this dependence, the filter will be applied on the original image (rather than be accumulated on the previous filters). Once the task is invoked, it is then added to the image's history (line 8) so that other future tasks will wait for it to complete. Note that deadlocks cannot happen with `dependsOn`, because dependence cycles cannot be created. A difference of this model compared to the fork-join model is that dependences within a task must be explicit (i.e. there is no implicit barrier). Since ParaTask should be viewed as a substitute for threads, and threading libraries do not impose such an implicit barrier, then ParaTask also does not impose this restriction.

Interactive tasks

Let us now consider a task that is not computationally intensive. For example, maybe the task performs a online search as in (B) of figure 1? In this situation, it is undesirable to assign such a task to one of the worker threads in case there are other computationally intensive tasks that would make better use of the thread (see *Use Thread Pools Correctly: Keep Tasks Short and Nonblocking* in Further Reading). ParaTask allows such tasks to be identified using the `INTERACTIVE_TASK` keyword:

Defining an interactive task

```
INTERACTIVE_TASK public List<Image> searchTask(String query) {
  // perform internet search
}
```

The difference of interactive tasks with standard tasks is that they do not get queued to the worker threads. Other than this, interactive tasks are treated the same as the standard tasks (e.g. the `dependsOn` clause and other features may all still be used). This allows for a unified tasking model, meaning that the concepts behind the threading model is integrated into the tasking model.

Interim results, progress and canceling

Sometimes we may want to display partially complete tasks to the user. For example, showing the images retrieved so far rather than having to wait until the end when all have arrived. ParaTask extends the `notify` clause concept with the `notifyInterim` clause:

Registering for interim results

```
1  public void actionPerformed(ActionEvent e) {
2    ...
3    currentSearchID = searchTask(query)
4      notify(searchCompleted())
5      notifyInterim(receivedAnotherImage(TaskID,Image));
6    ...
7  }
```

Just like the `notify` clause, the methods inside the `notifyInterim` clause will get executed by the EDT. The `currentSearchID` (line 3) represents the `TaskID` for the current search being performed. It is declared globally to keep it in scope should the search be canceled. The `receivedAnotherImage` method is defined to update the panel with a new thumbnail and overall progress:

Updating GUI with interim results

```
private void receivedAnotherImage(TaskID id, Image image) {
  panel.addImage(image);
  progressBar.setValue(id.getProgress());
}
```

We now explore the code behind `searchTask`:

Updating task status and checking for cancel requests

```
1  INTERACTIVE_TASK public List<Image> searchTask(String query) {
2    List<Image> results = new ArrayList<Image>();
3
4    PhotoList pList = Flickr.getPhotoIDs(query);
5
6    for (int i = 0; i < pList.size(); i++) {
7      Image thumb = Flickr.getThumbnail(pList.get(i));
8      results.add(thumb);
9
10     if (CurrentTask.cancelRequested()) {
11       CurrentTask.setProgress(100);
12       return results;
```

```
13        } else {
14          CurrentTask.setProgress(++i/pList.size()*100);
15          CurrentTask.publishInterim(thumb);
16        }
17      }
18      return results;
19    }
```

The first part of the search involves retrieving a list of IDs for images that match the search criteria (line 4). For each of the IDs, the actual image is retrieved (line 7) and saved to the result set (line 8). The task then checks to see whether a cancel request has been submitted (line 10). If so, the current result set is returned (line 12). Otherwise, the task computes its new progress (line 14) and publishes the newly retrieved image (line 15). Notice how all these features (the canceling check, progress updates and publishing of interim results) are performed without the task's knowledge of other code. Such canceling is also essential for implementing exception handlers.

Multi-tasks

Data parallelism is a very common form of parallelism where the same computation is to be performed multiple times (for example, (A) from figure 1). The problem with invoking a task multiple times is that we would not get any sense of group awareness amongst the multiple invocations. We prefer to invoke a task once, yet that task is automatically invoked multiple times. ParaTask's multi-task concept is perfect for such situations, allowing the sub-tasks to determine their position in the group (lines 2 and 4) and a barrier to synchronise with the sibling sub-tasks (line 12):

Defining a multi-task

```
1    TASK(*) public void multiTask(ParIterator<File> pi) {
2      int myPos = CurrentTask.relativeID();
3      print("Hello from sub-task "+myPos);
4      int numTasks = CurrentTask.multiTaskSize();
5      if (myPos == 0)
6        print("Multi-task has "+numTasks+" sub-tasks.");
7      ...
8      while ( pi.hasNext() ) {
9        process( pi.next() );
10     }
11     ...
12     CurrentTask.barrier();
13     ...
```

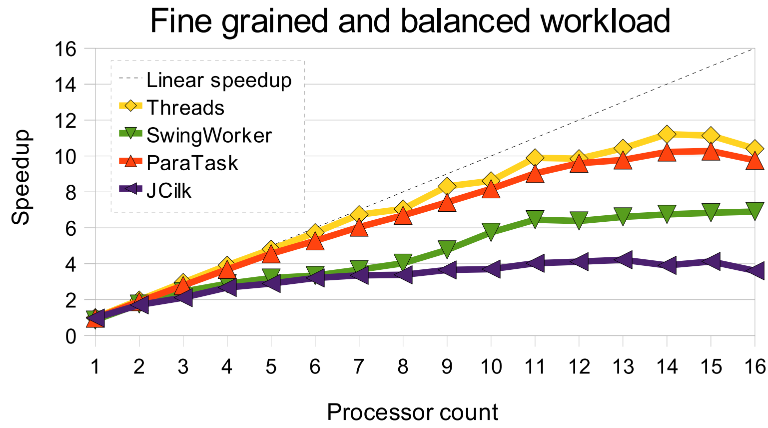
```
14  }
```

Whereas a standard task is annotated with `TASK`, a multi-task is annotated with `TASK(*)` meaning it is created once for every worker thread. Alternatively, annotating the multi-task with any integer n (instead of *) will create n tasks. The `ParIterator` (line 1) refers to the Parallel Iterator [5] concept, which is essentially a thread-safe iterator that extends the Java-style sequential iterator to allow the parallel traverse of an arbitrary collection of elements. The Parallel Iterator is particularly useful when used in combination with ParaTask's multi-task feature; the programmer does not need to concern with the creation of threads (handled by ParaTask's multi-task), nor concern with the distribution of elements (handled by the Parallel Iterator).

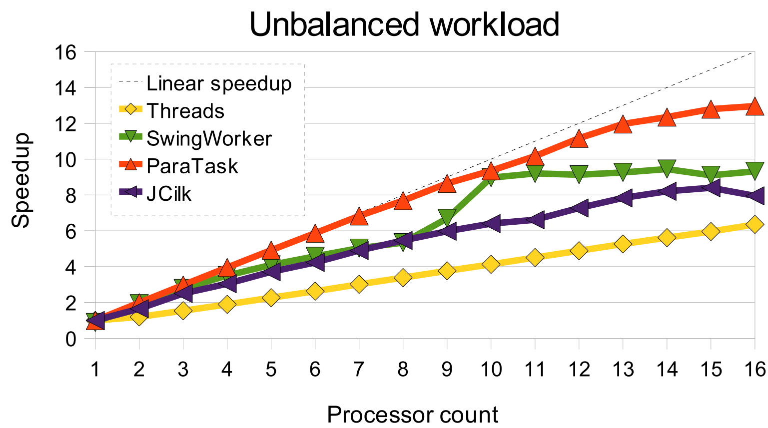
Applications and evaluation

The design goals of Parallel Task and Parallel Iterator were to achieve a truly object-oriented approach to parallel programming. This included integrating different task concepts into the same model, minimising code restructuring and promoting code reuse. Various performance benchmarks have been published for both Parallel Task [4] and Parallel Iterator [5] showing that these concepts have been introduced without sacrificing performance. The performance was compared to different parallelisation approaches using various benchmarks, showing low overhead and high speedups. The Parallel Iterator and Parallel Task concepts have been used in developing a number of applications (e.g. a parallel graph library, image application, PDF application, and web interaction), many of which are available for download.

Figure 4 illustrates the performance of ParaTask compared to typical Java parallelisation approaches a programmer may take (see section *The problem with existing tools*), by comparing the speedup to the original sequential benchmarks on a synthetic calculation (here the Raphson-Newton method). For very fine grained and balanced workloads, figure 4(a) shows that only a manual thread-based implementation, where the work is preallocated to the threads, can slightly outperform ParaTask. However, figure 4(b) shows that this approach does not extend well for unbalanced workloads, requiring a dynamic runtime scheduling solution. ParaTask performs most consistently across the different workloads. In numerous other performance evaluations it regularly outperforms the other approaches and is only occasionally surpassed by manual parallelisation with threads or by JCilk for the special case of a highly recursive and fine grained workload.



(a) Fine grained and balanced workload



(b) Unbalanced workload

Figure 4: ParaTask performance in comparison to typical Java parallelisation approaches.

Further reading

- For a discussion on issues such as responsiveness and the GUI thread, see Herb Sutter's *The Pillars of Concurrency*, Dr. Dobb's Journal:
  http://www.drdobbs.com/high-performance-computing/200001985

- For a discussion on task sizes and dependences, see Herb Sutter's *Use Thread Pools Correctly: Keep Tasks Short and Nonblocking*, Dr. Dobbs Journal:
  http://www.drdobbs.com/high-performance-computing/216500409

- To download Parallel Task and/or Parallel Iterator (under GPL license), including source files, example applications, documentations, publications, an Eclipse plugin and tutorials, visit:
  http://www.parallelit.org

References

[1] TIOBE Software BV, "TIOBE programming community index." http://www.tiobe.com/tpci.htm, 2010.

[2] E. Ludwig, *Multi-threaded User Interfaces in Java.* PhD thesis, University of Osnabrück, Germany, May 2006.

[3] H. Muller and K. Walrath, "Threads and Swing." The Swing Connection, 2008. http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html.

[4] N. Giacaman and O. Sinnen, "Parallel Task for parallelizing object-oriented desktop applications," in *IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC) held in conjunction with 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*, (Atlanta, USA), 2010.

[5] N. Giacaman and O. Sinnen, "Parallel Iterator for parallelizing object-oriented applications," *(accepted for publication) International Journal of Parallel Programming*, 2010.