Object-Oriented Parallelization of Java Desktop Programs

Nasser Giacaman and Oliver Sinnen, University of Auckland

// This article explores desktop applications' structure and the threading model's limitations while examining the parallelization of a desktop application using object-oriented and GUI-aware concepts. //

PARALLELIZATION IS THE PROCESS OF decomposing a large computation into smaller parts and simultaneously executing those smaller parts to reduce overall processing time. Despite its potential performance benefits, parallel computing has long been a programming nightmare for software developers because it requires developers to envision the original problem as smaller, somewhat independent subproblems and then carefully analyze the dependencies among those pieces. Even when the designer has met this theoretical challenge, practical challenges—such as implementing synchronization mechanisms, scheduling subproblems, and countless hours of debugging—remain.

So, what about desktop parallelization? We're no longer discussing parallelizing trivial scientific and engineering applications with large amounts of obvious and repetitive inherent parallelism (it's easy to envision these applications' smaller subproblems). We're now in the realm of irregularly structured computations with short runtimes. To add further complication, we execute these applications on nondedicated and unknown target systems (is the system a uniprocessor, quad-core processor, or many-core processor?). The more effort developers invest in expressing the problem's inherent parallelism, the more effort they require to realize it: more subproblem decomposition, more code restructuring, more synchronization, and more debugging. If they invest any less effort in realizing the inherent parallelism, the performance is punished accordingly.

Unfortunately, it gets even more complex for desktop applications. Users generally expect feedback on executing tasks, even intermittent updates on partially complete ones (such as a progress bar). Consequently, a desktop application's performance is primarily user perceived. We want a responsive, interactive application, even if the application executes on a single processor. Because desktop applications are user driven (unlike batch-type applications), the user interface's graphical and interactive nature contributes tremendously to the challenge. Finally, because developers can't determine the system specifications that their applications will execute on, dynamic runtime support that adjusts to hardware is vital.

So, how do we simplify desktop applications' parallelization? Object-oriented languages dominate desktop applications' development.[1] We must realize parallelization's benefits in the realm of such high-level languages, without resorting to languages like C or Fortran. Those low-level languages might be speed-efficient, but large desktop applications demand the software engineering benefits associated with object-oriented languages. In this article, we address these challenges by discussing the parallelization of an object-oriented desktop application with a GUI. We start by examining the rarely discussed challenges that are unique to GUI desktop applications—namely, user interactivity, graphical frameworks' limitations, and the large variety of target systems. Because most desktop applications are written in object-oriented languages, we must perform the parallelization in these languages. We based our parallelization approach, Parallel Task (also called ParaTask for short), on a unified task concept that integrates all common concurrency types.

Desktop Applications' Anatomy

Figure 1 illustrates a typical desktop application scenario. A user interacts with the application to perform various tasks. Some tasks might execute only once, while others might execute multiple times on different data elements (such as processing a directory full of files in Figure 1a). Some tasks are more computationally intensive, while others are I/O bound—such as an Internet search (Figure 1b), waiting for user input (Figure 1c), or printing (Figure 1d). Some tasks may execute independently, while others might have ordering constraints that depend on other tasks' completion (Figure 1e). We categorize this into different types of tasks, each with different behaviors that demand different handling. To ease the parallelization process, the first step is unifying these different task concepts into the same model.

Before attempting to parallelize desktop applications, we must understand their external and internal composition. We're most familiar with the external features, which include numerous visual input components (such as buttons and text fields) and output components (such as labels and progress bars). It's this distinguishing GUI that enables a desktop application to interact with its users.

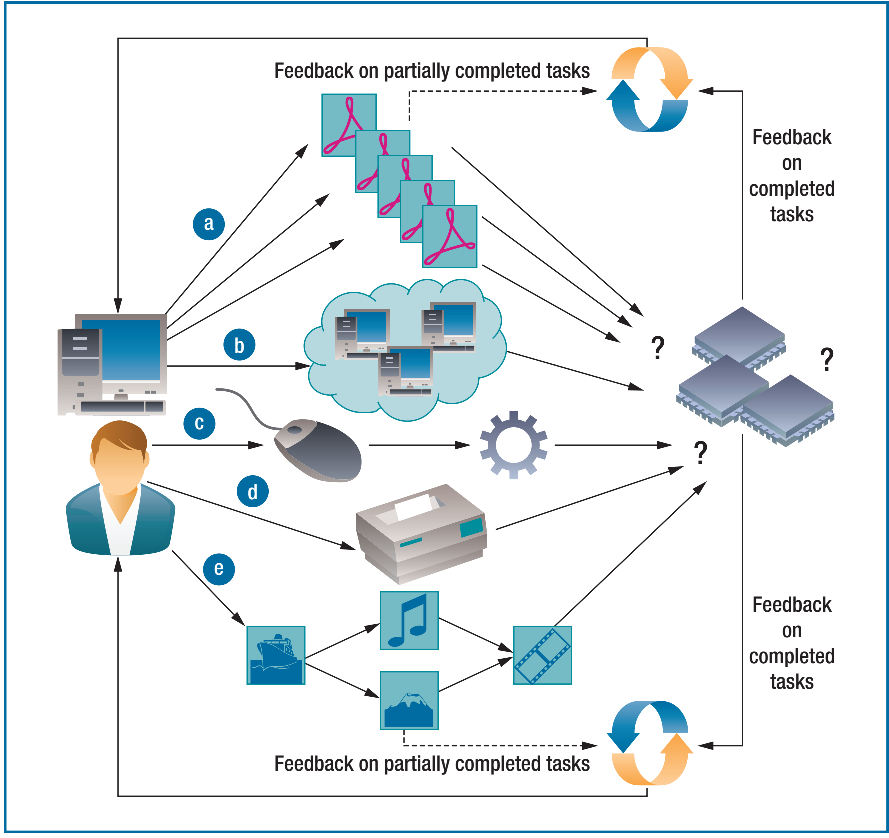

Feedback on partially completed tasks
Feedback on completed tasks
a
b
c
d
e
?
?
?
Feedback on completed tasks
Feedback on partially completed tasks

FIGURE 1. Desktop applications involve different types of tasks that require different implementation approaches. Such tasks include (a) processing a directory of files, (b) performing an Internet search, (c) waiting for user input, and (d) printing, while some tasks (e) can't execute until others are complete. Another complication is that the system that the application will run on is unknown.

What about a desktop application's internal structure? The most vital organ is the event loop (see Figure 2a), which reacts to events (such as a mouse click) by dispatching them to the appropriate event handler (Figure 2b). The GUI thread, called 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. No other thread may perform these actions—a restriction common to most desktop and GUI frameworks.[2,3] Consequently, if the event loop isn't processing events in a timely fashion, the application will become unresponsive, "freeze," and frustrate users.

To avoid such inanimate behavior, multithreading 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 offloaded to another thread (Figure 2c) so the GUI thread can return to the event loop. Both threads then share the single processor, but neither is fully stopped. Although the helping thread executes the offloaded computation (Figure 2d), it may not directly access any GUI component (Figure 2e) because the GUI components aren't thread-safe (remember, the GUI thread is solely responsible for anything GUI-related). Consequently, events must be posted to the GUI thread (Figure 2f) that will in turn be handled by the GUI thread (Figure 2g).

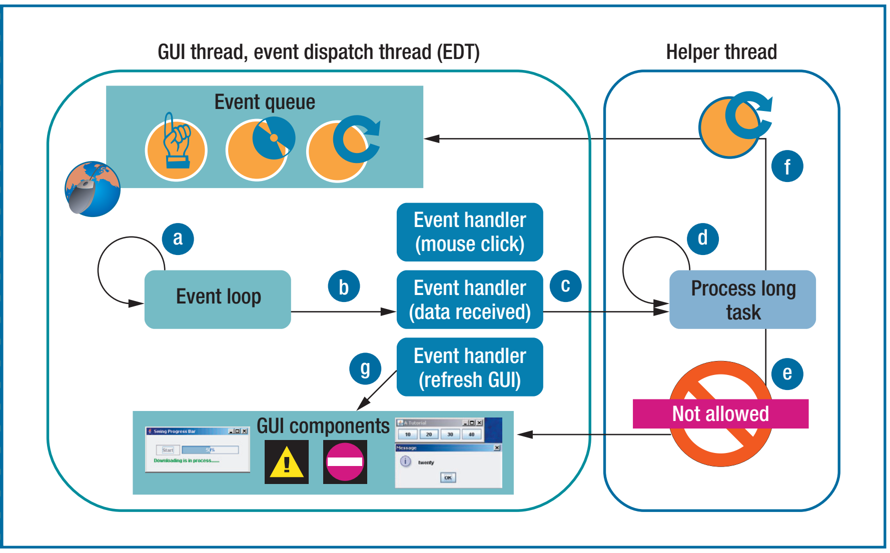

GUI thread, event dispatch thread (EDT)
Helper thread
Event queue
a
Event loop
b
Event handler (mouse click)
Event handler (data received)
c
Event handler (refresh GUI)
g
GUI components
d
Process long task
f
e
Not allowed

FIGURE 2. Structure of a multithreaded GUI desktop application. The parts include (a) the event loop, (b) the event handler, (c) another thread, (d) an offloaded computation, (e) GUI components, (f) GUI events, and (g) the GUI thread.

So, when multiple processors came into play, it felt natural to use threads not only for virtual concurrency and responsiveness, but for real parallel execution, where different processors execute the threads. Unfortunately, the concept of threads is ill-fitted to parallel computing's diverse demands. Offloading a computation to another thread isn't enough—you must further divide and distribute the computation 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 synchronization is necessary among subtasks to ensure a logically correct execution of the original computation. In other words, desktop parallelization incorporates all generalized parallel computing challenges together 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, so manually parallelizing a GUI application using Java Threads has been the norm. However, this model is unsuitable for parallelizing desktop applications. First, Java Threads' purpose is to fork a new execution thread at a particular point in the program. As such, most independent subproblems don't necessarily demand a new execution thread. Rather, we only wish to express that such a computation can safely be performed asynchronously. In other words, we merely wish to denote this computation as a potential task, as opposed to enforcing a new execution thread for it.

Second, performance consequences make the Java Threads model undesirable. If the application creates too few threads, not enough parallelism is introduced to exploit the number of cores. Conversely, if the application creates too many threads, resource contention and scheduling overheads degrade performance. Performance issues aside, the threading model reduces code legibility because the code migrates to new threads. Programmers must then manage any dependencies among the subcomputations manually. Not only is this error-prone, it also introduces coupling among otherwise independent tasks.

For these reasons, modern parallelization tools have opted for a tasking model as opposed to the traditional threading model: programmers express independent code snippets as tasks, and the tool's runtime system manages 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 dependency handling among tasks, and avoid I/O bound tasks. Outside of Java, other modern parallelization tools include Cilk++, OpenMP Task, Intel Threading Building Blocks, Apple's Grand Central Dispatch, X10, and the .NET Task Parallel Library. Although these approaches are a huge step forward from a manual thread-based parallelization, many of them aren't truly object-oriented and often involve add-ons that aren't designed.

More importantly, when it comes to parallelizing desktop applications, the primary problem is that none of these tools consider the structure of GUI applications. Consequently, the programmer is still left with the responsibility of ensuring that GUI computations are performed only on the GUI thread, and that the GUI thread remains free. Furthermore, implementing dependencies between tasks becomes the programmer's responsibility. This results in a high amount of coupling among (otherwise independent) tasks and tangling of parallelization concerns with the actual business logic. The tangling and coupling reduce the amount of code reuse, a principle important to both software engineering and object-oriented programming.

Figure 3 illustrates ParaTask's performance as compared to typical Java parallelization approaches by examining the speedup to the original sequential benchmarks on a synthetic calculation (here, the Newton–Raphson method). For fine-grained and balanced workloads, Figure 3a shows that only a manual thread-based implementation, where the work is preallocated to the threads, can slightly outperform ParaTask. However, Figure 3b shows that this approach doesn't extend well for unbalanced workloads, requiring a dynamic runtime scheduling solution. ParaTask performs most consistently across different workloads. In numerous other performance evaluations, it has regularly outperformed other approaches, and is only occasionally surpassed by manual parallelization with threads or by JCilk for the special case of a highly recursive and fine-grained workload.

Parallelizing Desktop Applications with Parallel Task

To address these problems, we propose ParaTask for the parallelization of object-oriented desktop applications.[4] The ParaTask parallelization tool consists of a source-to-source compiler and supporting runtime system to manage tasks. Although ParaTask draws on standard parallelization concepts, it unifies these concepts into an object-oriented environment. Programmers introduce concurrency with a single keyword, and the different task concepts integrate into one model. ParaTask also supports an intuitive approach to dependency handling and has the unique feature of focusing on GUI applications. Here, we'll walk through parallelizing a GUI application, introducing the various ParaTask features as we would use them. The example application, ParaImage, provides various functionality such as an online image search and image editing. Figure 4 shows a screenshot of the image-editing project in ParaImage.

Defining and Invoking Tasks

Owing to the complications of manually threading an application (both in terms of performance and ease of use), we would like a simple task solution. Because we're focusing on object-oriented applications, we must decide how we express tasks—using methods or objects? Choosing objects to represent tasks doesn't address threads' programming difficulties because developers must still restructure and migrate the code. For this reason, ParaTask encapsulates tasks at the method level and defines them with the TASK keyword. Here is the definition of a task that performs edge detection on an image:

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

This event handler shows the invocation of this 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 behavior, we'll generally need a handle on the task invocation should we wish to synchronize with its completion (for example, to access the return result). The TaskID serves this purpose and is essentially a future value that represents a task invocation.

The actionPerformed method is an event handler, and as such, the GUI thread performs it. In a sequential implementation, the GUI thread would execute the edge detection in its entirety (the application freezes during this time). However, in the parallel mode, the GUI thread only places the task into the queue and returns to the event loop. The tasks are then scheduled for execution by a team of threads; ParaTask

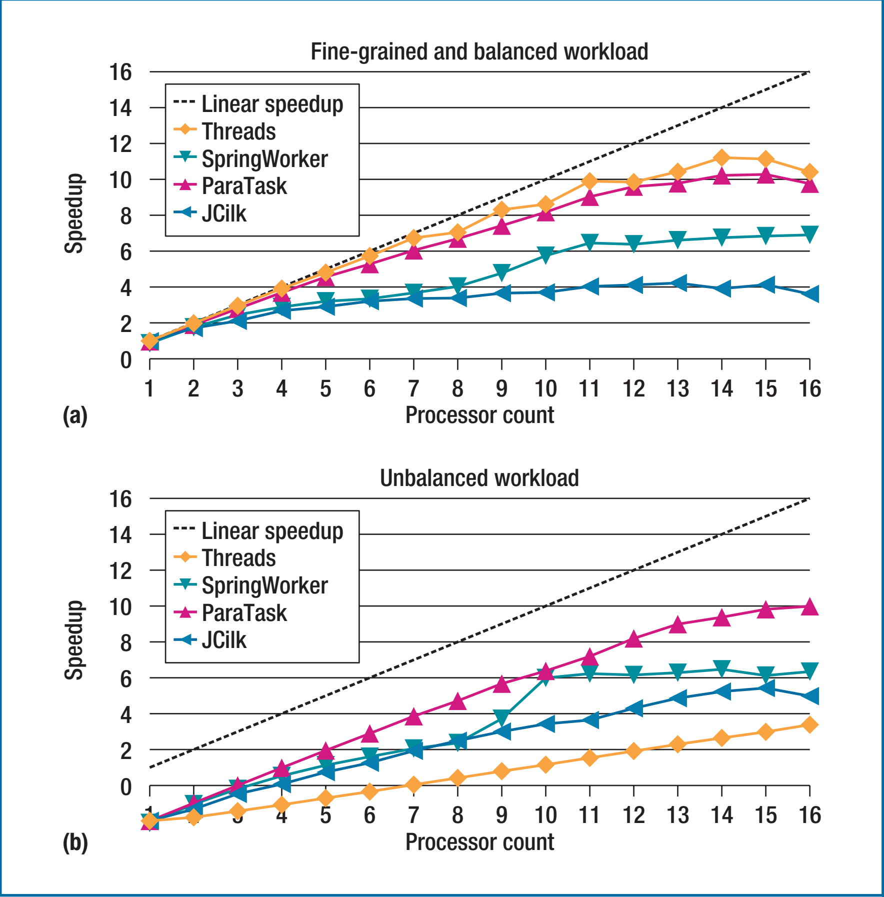


FIGURE 3. ParaTask performance as compared to typical Java parallelization approaches: (a) a fine-grained, balanced workload; (b) an unbalanced workload.

FURTHER READING ON DESKTOP PARALLELIZATION

For a discussion on issues such as responsiveness and the GUI thread, see Herb Sutter's "The Pillars of Concurrency" (*Dr. Dobbs Journal*, 2 July 2007; www.drdobbs.com/high-performance-computing/200001985).

For a discussion on task sizes and dependencies, see Herb Sutter's "Use Thread Pools Correctly: Keep Tasks Short and Nonblocking" (*Dr. Dobbs Journal*, 13 Apr. 2009; www.drdobbs.com/high-performance-computing/216500409).

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

automatically creates and manages an ideal number of threads to keep each processing core busy. This means that invoking multiple tasks is much more efficient than creating a thread for each computation.

You've probably figured out our next problem. Now that we've offloaded the task from the GUI thread, how will we 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:

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

If the task is already complete by this stage, no problem—the result is already available. Otherwise, the thread invoking this call blocks until the task completes. This behavior is clearly unacceptable for the GUI thread. Instead, we need a way for the GUI thread to be notified when the task completes—that is, a nonblocking notification solution:

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

Even though we offload the filter computation to another thread, the GUI thread must still update the GUI. By using the notify clause, the GUI thread returns to the event loop and later learns when the task completes. The methods specified inside the notify clause and the task definition remain decoupled:

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

Dependencies

What happens when a newly invoked task depends 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's a dependency between the tasks applied on the same image. Using standard threading libraries, programmers would have to manually code for such dependencies using synchronization mechanisms such as wait conditions. Besides being error-prone, this approach would couple the tasks with each other. For such cases, we suggest using the dependsOn clause:

```
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 TaskIDs that make up the TaskIDGroup, line 2). Whenever we apply a new filter (that is, a new task) to an image, that filter will only apply once the previous filters (tasks) have completed on the image (line 6). Otherwise, without this dependency, the filter will be applied on the original image (rather than be accumulated on the previous filters). Once the task is invoked, it's then added to the image's history (line 8) so that other future tasks will wait for it to complete. Deadlocks can't happen with dependsOn, because dependency cycles can't be created. Unlike the fork-join model, this model requires dependencies within a task to be explicit (that is, there is no implicit barrier). Because ParaTask is a substitute for threads, and threading libraries don't impose such an implicit barrier, ParaTask also doesn't impose this restriction.

Interactive Tasks

Let's now consider a task that isn't computationally intensive—maybe the task performs an online search as in Figure 1b. In this situation, assigning such a task to a worker thread is undesirable if there are other computationally intensive tasks that would make better use of the thread (see the "Further Reading on Desktop Parallelization" sidebar for more information). ParaTask lets us identify such tasks using the INTERACTIVE_TASK keyword:

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

The difference between interactive and standard tasks is that the former don't queue to the worker threads. Other than this, ParaTask treats interactive tasks the same as standard tasks (for example, you can still use the dependsOn clause and other features). This allows for a unified tasking model, meaning that the concepts behind the threading model integrate into the tasking model.

Interim Results, Progress, and Canceling

Sometimes, we might want to display partially complete tasks to the user—for example, showing the images retrieved so far rather than having to wait until they've all arrived. ParaTask extends the notify clause concept with the notifyInterim clause:

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

Just like with the notify clause, the GUI thread executes the methods inside the notifyInterim clause. The currentSearchID (line 3) represents the TaskID for the current search being performed. It's declared globally to keep it in scope should the search be canceled. The receivedAnotherImage method (line 5) is defined to update the panel with a new thumbnail and overall progress:

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

We now explore the code behind searchTask. We update task status and check for cancel requests:

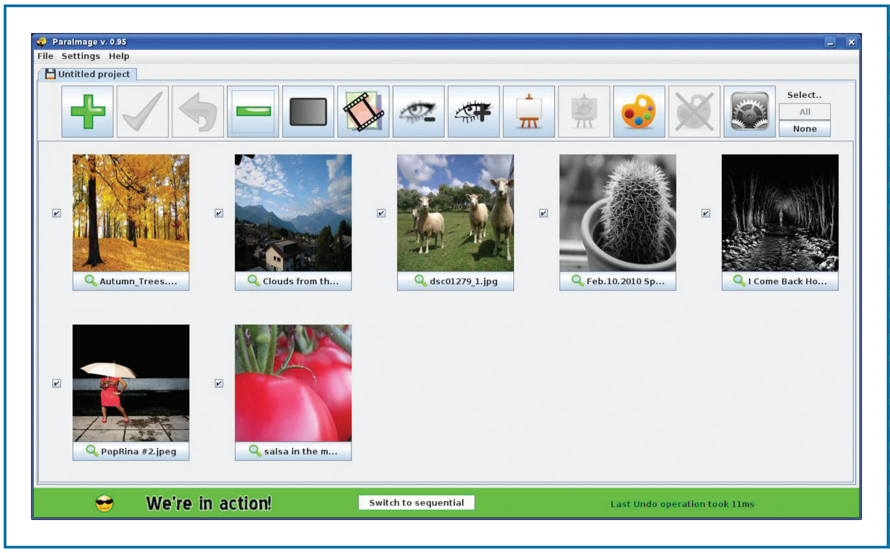


FIGURE 4. ParaImage, a GUI application developed using ParaTask, involves many of the different parallelization concepts discussed in Figures 1 and 2.

```
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 task retrieves the actual image (line 7) and saves it to the result set (line 8). The task then checks to see whether a cancel request has been submitted (line 10). If so, it returns the current result set (line 12). Otherwise, the task computes its new progress (line 14) and publishes the newly retrieved image (line 15). All these features (the canceling check, progress updates, and publishing of interim results) perform without the task's knowledge of other code. Such canceling is also essential for implementing exception handlers.

Multitasks

Data parallelism is a common form of parallelism where the same computation is to be performed multiple times (see Figure 1a). The problem with invoking a task multiple times is that we wouldn't get any sense of group awareness among the multiple invocations. We prefer to invoke a task once, yet that task is automatically invoked multiple times. ParaTask's multitask concept is perfect for such situations, allowing the subtasks to determine their position in the group (lines 2 and 4) and a barrier to synchronize with the sibling subtasks (line 12). We define a multitask using the following code:

```
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, we annotate a multitask with TASK(*), meaning that it's created once for every worker thread. Alternatively, annotating the multitask with any integer *n* (instead of *) will create *n* tasks. The ParIterator (line 1) refers to the Parallel Iterator concept,[5] which essentially extends the Java-style sequential iterator to allow the parallel traverse of an arbitrary collection of elements. Parallel Iterator is particularly useful in combination with ParaTask's multitask feature; the programmer doesn't need to create threads (done by ParaTask's multitask) or distribute elements (done by the Parallel Iterator).

ParaTask and Parallel Iterator aim to achieve a truly object-oriented approach to parallel programming by integrating different task concepts into the same model, minimizing code restructuring, and promoting code reuse. Various performance benchmarks exist for both Parallel Task and Parallel Iterator, showing that developers have introduced these concepts without sacrificing performance. Comparing their performance to different parallelization approaches using various benchmarks shows that they create low overhead and high speedups. Developers have used Parallel Iterator and ParaTask in creating several applications (such as a parallel graph library, image application, PDF application, and Web interaction), many of which are available for download at www.parallelit.org.

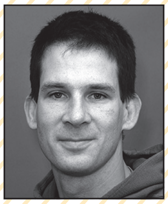
Future work for both ParaTask and Parallel Iterator includes optimizing runtime to improve speed and introducing memory awareness for scheduling—for example, to avoid false sharing by seeing how cache effects and NUMA (Non-Uniform Memory Access) systems affect performance.

References

1. TIOBE Software, "TIOBE Programming Community Index," Nov. 2010; www.tiobe.com/tpci.htm.
2. E. Ludwig, "Multithreaded User Interfaces in Java," doctoral dissertation, Dept. Mathematics and Computer Science, Univ. of Osnabrück, 2006.
3. H. Muller and K. Walrath, "Threads and Swing," *Oracle Sun Developer Network*, Apr. 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," *2010 IEEE Int'l Symp. Parallel & Distributed Processing, Workshops and PhD Forum* (IPDPSW), IEEE CS Press, 2010, pp. 1–8.
5. N. Giacaman and O. Sinnen, "Parallel Iterator for Parallelizing Object-Oriented Applications," *Int'l J. Parallel Programming*, Sept. 2010, doi:10.1007/s10766-010-0150-5.

ABOUT THE AUTHORS

NASSER GIACAMAN is a postdoctoral research fellow in the Department of Electrical and Computer Engineering at the University of Auckland and a lecturer in the university's software engineering program. His research interests include parallel computing for desktop environments and source-to-source compilers. Giacaman received his PhD in desktop parallelization from the University of Auckland. Contact him at ngia003@aucklanduni.ac.nz.



OLIVER SINNEN is a senior lecturer in the Department of Electrical and Computer Engineering at the University of Auckland. His research interests include parallel computing, scheduling, reconfigurable computing, graph theory, and algorithm design. Sinnen received his PhD in electrical and computer engineering from Instituto Superior Técnico (IST), Technical University of Lisbon. He authored the book *Task Scheduling for Parallel Systems* (Wiley, 2007). Contact him at o.sinnen@auckland.ac.nz.