Parallel Task for parallelising object-oriented desktop applications

Nasser Giacaman and Oliver Sinnen
Parallel and Reconfigurable Computing lab
Department of Electrical and Computer Engineering
The University of Auckland, New Zealand
n.giacaman@auckland.ac.nz, o.sinnen@auckland.ac.nz

August 25, 2016

Abstract

With the arrival of multi-cores for mainstream desktop systems, developers must invest the effort of parallelising their applications in order to benefit from these systems. However, the structure of these interactive desktop applications is noticeably different from the traditional batch-like applications of the engineering and scientific fields. We present Parallel Task (short ParaTask), a solution to assist the parallelisation of object-oriented applications, with the unique feature of including support for the parallelisation of graphical user interface (GUI) applications. In the simple, but common, cases concurrency is introduced with a single keyword. ParaTask sets itself apart from the many existing object-oriented parallelisation approaches by integrating different task types into the same model and its careful adherence to object-oriented principles. Due to the wide variety of parallelisation needs, ParaTask provides intuitive support for dependence handling, non-blocking notification and exception handling in an asynchronous environment as well as supporting a flexible task scheduling runtime (currently work-sharing, work-stealing and a combination of the two are supported). The performance is excellent compared to traditional Java parallelisation approaches, shown using a variety of different workloads.

Keywords: Parallel computing, task parallelism, object-oriented programming, desktop applications, graphical user interface, event dispatch thread.

1 Introduction

Despite the performance benefits, parallel computing has traditionally been (and still remains) notoriously difficult for software developers [1]. In addition to the usual challenges of developing a sequential program, parallel computing presents a new range of complications. First come the theoretical challenges of task decomposition, dependence analysis and task scheduling. Then there are the practical challenges such as synchronisation, debugging and portability. Since the objective behind parallel computing is to reduce the execution time of a program, parallelising code has traditionally been paired with general code optimisations for performance (especially in the scientific and engineering area). It is therefore no surprise that applications for these domains have been and are still written in low-level, but speed-efficient languages like Fortran or C [2].

The situation is even worse for parallel desktop applications since they are generally irregular with short run-times. They also run on non-dedicated systems, let alone knowing what the system specifications are in the first place (e.g. number of processors, amount of memory and so on). When it comes to desktop applications, we want to improve the performance through parallelisation but without sacrificing the benefits of high-level languages (code abstraction, encapsulation and so on).

In proposing new tools to parallelise desktop applications, it is vital to understand their structure. Consequently, this paper focuses on object-oriented languages due to their popularity [3], especially for desktop application development. For example, common languages for Windows programming include C++, C# and Java. The K Desktop Environment (KDE) for Linux is developed

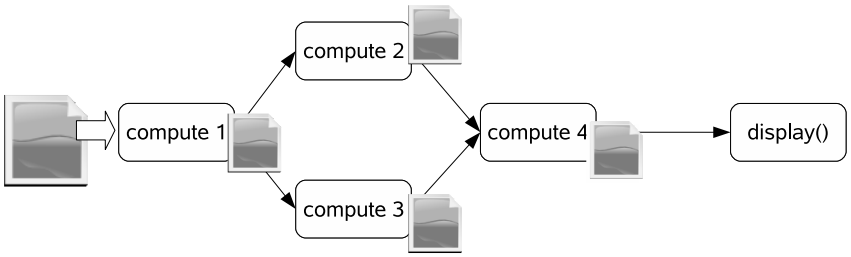


Figure 1: Four tasks with dependences amongst them. Only the second and third computations may execute in parallel, since they must wait for the first computation to complete. The `display()` method performs GUI-related computations.

in C++ using Nokia's Qt toolkit and Mac OS X is developed using Objective-C. However, focusing on the parallelisation of *object-oriented applications* is typically not enough: one must also look deeper at the *structure of desktop applications* (section 2.3).

Parallel computing has arrived at mainstream desktop systems in the form of multi-core processors because of the difficulties maintaining improvements in uni-processor clock-speed. Even though parallel computing is decades old, desktop parallelisation is fairly new. Users will not witness any performance improvements if desktop applications are not parallelised with performance in mind [4, 5].

Consider the example application of figure 1. The programmer has identified 4 independent computations to be executed in response to the user pressing a button. In a sequential program, the following code is developed:

```
public void actionPerformed() {

    File file1 = compute1(“pic.jpg”);
    File file2 = compute2(“pic1.jpg”);
    File file3 = compute3(“pic1.jpg”);
    File file4 = compute4(“pic2.jpg”, “pic3.jpg”);
    display(“pic4.jpg”);

}
```

As we will see in section 2, the `actionPerformed()` is an event handler whose computation should complete with minimal time. Even though parallelisation may help speed the 4 computations, this still might be insufficient for a responsive event handler. In fact, the `actionPerformed()` (and any other event handler) should appear instantaneous because it is executed by an event handling thread that must be free to respond to other events. Yet only this very same thread may call `display()` when the 4 computations are completed (required by most GUI toolkits). We immediately see that different threads have different roles in event-based applications.

Contributions

In light of multi-cores being mainstream on typical desktop systems, our vision is not only to introduce task parallelism but also to create a simple and intuitive approach to event-based applications in a parallel environment (section 2). We investigate a unified task concept, called Parallel Task (section 3); it allows a wide range of computation problems to be catered for by integrating different task types into the same unifying model. Section 4 discusses the various types of synchronisation and communication with Parallel Task, including support for intuitive dependence handling and non-blocking notification. The strict adherence to object-oriented concepts is analysed in section 5, including concurrent exception handling. We present the implementation and discuss scheduling options in section 6 and evaluate and compare performance in section 7 to a range of typical parallelisation approaches.

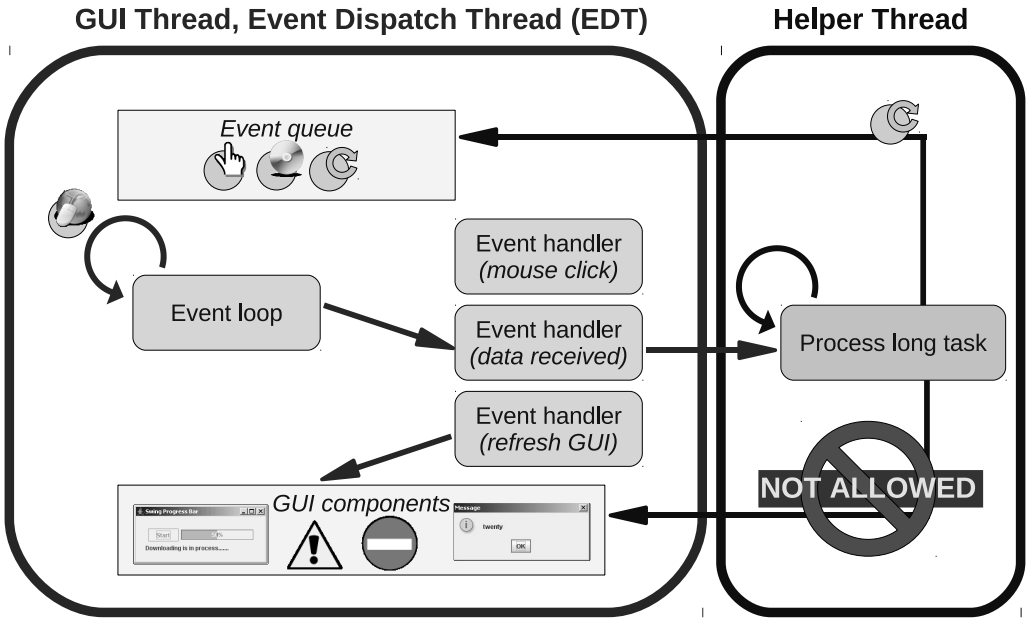


Figure 2: Structure of a multi-threaded Java GUI application. The purpose of multi-threading so far is solely to improve the application's *responsiveness* (rather than improve *performance* through parallelism). This is achieved by dispatching all long executing tasks to helper threads, allowing the EDT to return to the event loop.

2 Background

In parallelising desktop applications, we must first understand the structure of desktop applications and the threading model that programmers must adhere to.

2.1 Graphical user interfaces

Desktop applications allow users to interact through a graphical user interface (GUI). The application displays a range of visual components, some acting as a form of input (e.g. buttons, text fields) while others display application status (e.g. labels, progress bars). Such an application would be based on the event-driven paradigm, where the program's execution flow is determined by *events* (e.g. mouse clicks, messages from other threads). As figure 2 shows, the application has an *event loop* waiting for events to arrive. The events are then dispatched to the appropriate *event handler* to take the appropriate action.

Many toolkits are available for programmers to ease the development of event-based applications. These toolkits provide many graphical components as well as the event loop and event handling. Generally, programmers only need to specify the logic of event handlers (e.g. the response to a certain button being clicked). Programmers must ensure that these event handlers are short so that control returns to the event loop: otherwise, events will backlog and the application appears unresponsive (for example, GUI applications appear to "freeze" when the GUI thread does not repaint). Figure 2 also shows how GUI components are typically manipulated by a single thread, hence the name *GUI thread*.

2.2 Where are the multi-threaded toolkits?

Most GUI toolkits available are single-threaded. Only one dedicated thread is allowed to access the GUI components: the *GUI thread*. In Qt, the *main thread* (the initial thread that starts the program) is also the GUI thread. Java is similar in that it allows only one thread to access GUI components, however this thread is not the main thread: it is a special thread created by Java, known as the *event dispatch thread* (EDT) [6]. In most Java GUI applications, the main thread

will exit once the GUI and EDT are started. Why is it that such GUI toolkits are single-threaded? Even more interestingly, why are they single-threaded even though they are part of a library that contains threading support?

In attempting to develop a thread-safe GUI toolkit, one faces the standard parallelisation challenges. To avoid non-deterministic behavior, mutual exclusion (usually implemented with locks) is required when multiple threads access shared data (and at least one thread is modifying the data). Even though such synchronisation protects the data by serialising the access, it unfortunately introduces other problems. The programmer must now decide the granularity of mutual exclusion: coarse-grained locking reduces the concurrency while fine-grained locking complicates the programming and increases the possibility of deadlock.

Ultimately, a decision has to be made whether multi-threading will be supported in the toolkit. If the answer is yes, then all aspects of the toolkit need to be thread-safe. This incurs a performance penalty for applications not multi-threaded. Combined with the performance versus programmability trade-off toolkit implementers face, GUI toolkits are single-threaded [7, 8, 9, 10]. In fact, the single-threaded model discussed above is not limited only to GUI toolkits. As an example, some native libraries require that all access be made from the same thread [9].

2.3 Multi-threaded GUI applications

Since the EDT is responsible for responding to events, the event handlers must complete quickly to maintain a responsive user interface. In situations where event handlers require more time to complete, such code must be executed on another *helper thread* (figure 2) to allow the EDT control of the event loop. Programmers are responsible in creating these additional helper threads and ensuring they only perform background work. Only the EDT is responsible for GUI manipulations, such as painting components on the screen since the GUI toolkit is not coded thread-safe (section 2.2). Programmers may schedule such code to execute on the EDT using `SwingUtilities.invokeLater()` or `SwingUtilities.invokeAndWait()`.

In fact, even the "multi-threaded" application of figure 2 might be considered inadequate to benefit from multi-cores; the helper thread is overwhelmed with all the computational workload, while the EDT essentially remains idle waiting for more events. This is because the purpose of multi-threading is to achieve responsiveness by freeing the EDT. Even if the application will knowingly only execute on a uni-processor, multi-threading would still be implemented. However, desktop applications must now be multi-threaded with a different goal in mind: exploiting the inherent parallelism of these multi-core processors.

2.4 The threading model versus the tasking model

The threading model

In order to improve the parallelism, the programmer may decide to create a new thread for each computation. Although this has the potential to exploit the parallelism, many difficulties are encountered. From a performance point of view, this incurs excess overhead that causes the application response time to suffer:

- Creating new threads solely for the purpose of executing short-lived computations presents extra work for the JVM. This includes creating the thread, starting it and then cleaning up after it terminates [8].
- *Over-subscription* occurs, where the number of active threads exceeds the available number of cores. The performance could easily degrade due to resource contention, scheduling overheads and memory bandwidth limits [11].

As well as high overheads, the threading model also presents difficulty for the programmer:

- The code of the independent computations must be migrated into the `run()` method of a `Thread` class. This is performed for each of the independent computations. This is further complicated if the original code made reference to any shared variables.

- Since the implicit ordering of actions is lost, coupling is essentially introduced. For example, note the dependences between the 4 computations; the programmer must now enforce these dependences using condition variables amongst the different computations. This also reduces the re-use of these computations for other applications that do not have these dependences.

The tasking model

In overcoming the performance issues of the threading model, a tasking model is usually implemented. This involves having a fixed pool of threads that are ready to execute tasks [8, 12]. The thread pool technique serves well for applications with many short-lived independent computations. Rather than creating threads, programmers encapsulate independent code within some form of a lightweight task object. These task objects are then scheduled to be executed by one of the threads in the pool.

Although the tasking model addresses the performance issues of the threading model, it generally does not address the programming difficulty. Programmers are typically still required to restructure the code into task objects (such as `Runnable`) and handle any dependences amongst these tasks. In particular, this still presents the programmer with the problem of newly introduced coupling amongst the tasks.

2.5 Typical GUI parallelisation approaches

Since this paper focuses on the parallelisation of GUI applications (in this particular case for Java), we present the current approaches that a programmer may take to correctly use concurrency in order to parallelise a GUI application. Unfortunately, we are limited to only three options. In section 7, we discuss the performance of these approaches in comparison to ParaTask. For illustration, we parallelise the same image application example presented in section 1.

Java Threads

Threads have been an integral part of Java since its initial release. Consequently, parallelising a GUI application manually using Java Threads has traditionally been the norm. The first step in this approach is to offload all the computation away from the EDT and into helper thread(s). Since we are interested in improving performance due to parallelism (and not only just to improve application responsiveness), we offload the computation into multiple helper threads. A possible solution may resemble that below:

```
public void actionPerformed() {

    final Thread t1 = new Thread() {
        public void run() {
            File f1 = compute1(“pic.jpg”);
        }
    };
    final Thread t2 = new Thread() {
        public void run() {
            t1.join();
            File f2 = compute2(“pic1.jpg”);
        }
    };
    final Thread t3 = new Thread() {
        public void run() {
            t1.join();
            File f3 = compute3(“pic1.jpg”);
        }
```

```
        };
        Thread t4 = new Thread() {
            public void run() {
                t2.join();
                t3.join();
                File f4 = compute4(‘‘pic2.jpg’’, ‘‘pic3.jpg’’);
                SwingUtilities.invokeLater(new Runnable() {
                  public void run() {
                    display(‘‘pic4.jpg’’);
                  }
                });
            }
        };
        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }
```

The above code has many undesirable aspects:

- Notice the explicit use of `invokeLater()`. This is necessary since the `display()` method may only be computed on the EDT (as discussed in section 2.3).
- The original program code has become largely tangled with parallelisation code. This tangling of concerns significantly reduces the code legibility [13].
- The concurrency is coordinated by the caller, rather than the callee, therefore breaking encapsulation [14] (discussed further in section 5.1).
- In addition to reduced legibility, the tangling of concerns reduces code reuse since the different tasks are now coupled with each other (for example, the second thread waiting for the first thread).

There are variations as to how Java threads might be used. In the code snippet above, a `Thread` is created for *every* task. For a large number of fine-grained tasks, the excessive number of threads will lead to a decreased performance (as was discussed in section 2.4). Therefore, the programmer may decide to create a fixed number of threads and manually assign the tasks to the different threads. Although such a static decomposition reduces the runtime overhead, this unfortunately reduces the load-balancing; it is additional effort on the programmer's behalf to find the appropriate balance.

SwingWorker

SwingWorker is a significant improvement compared to manually using Java Threads. Rather than placing code inside a `Thread`, the code is placed within a lightweight `SwingWorker` instance. There are 2 methods to override: `doInBackground()` for time consuming computations and the optional `done()` for GUI related computations. Compared to the approach of manually using Java Threads, the advantage here is that the programmer no longer needs to be concerned with manually creating threads and managing a thread pool. However, many of the same disadvantages still exist (for example: tangling parallelisation code, breaking of encapsulation, coupling amongst tasks and so on). Below is the code required when using SwingWorker:

```
    public void actionPerformed() {

        final SwingWorker sw1 = new SwingWorker() {
```

```
        protected File doInBackground() {
            File f1 = compute1(‘‘pic.jpg’’);
            return f1;
        }
    };
    final SwingWorker sw2 = new SwingWorker() {
        protected File doInBackground() {
            sw1.get();
            File f2 = compute2(‘‘pic1.jpg’’);
            return f2;
        }
    };
    final SwingWorker sw3 = new SwingWorker() {
        protected File doInBackground() {
            sw1.get();
            File f3 = compute3(‘‘pic1.jpg’’);
            return f3;
        }
    };
    SwingWorker sw4 = new SwingWorker() {
        protected File doInBackground() {
            sw2.get();
            sw3.get();
            File f4 = compute4(‘‘pic2.jpg’’,‘‘pic3.jpg’’);
            return f4;
        }
        protected void done() {
            display(‘‘pic4.jpg’’);
        }
    };
    sw1.execute();
    sw2.execute();
    sw3.execute();
    sw4.execute();
}
```

The `get()` method invoked on a SwingWorker instance is used to manage the dependences amongst the tasks (by blocking until the respective SwingWorker instance completes). Unfortunately, such an approach of handling task dependences is not scalable and could quickly lead to deadlock (especially in a recursive tree-like dependency situation). This is because a SwingWorker instance gets mapped to a Thread, and Java creates a fixed number of these Threads. If all the SwingWorker instances are blocking for another instance to complete, then none of the SwingWorker instances progress as they are essentially all "processing" a blocked task.

ExecutorService

The ExecutorService interface allows programmers to submit tasks in the form of `Runnable` instances. The ExecutorService represents an abstraction to a pool of threads, and the tasks submitted will be executed by one of the contained threads within the ExecutorService. Java provides various convenience methods to create such pools, two of which include:

- `Executors.newFixedThreadPool(int N)`: creates a pool with exactly N threads. If more tasks exist than threads, then the remaining tasks are queued until a thread is free. This relates to the tasking model discussed in section 2.4.
- `Executors.newCachedThreadPool()`: if no thread is free when a task is submitted, a new thread is created to execute it. This more closely relates to the threading model discussed in section 2.4. Idle threads return to the pool and terminate after 60 seconds of inactivity.

Consider the code below that uses ExecutorService to achieve the required parallelism:

```
ExecutorService pool = Executors.newFixedThreadPool(numberCores);
...
public void actionPerformed() {
    final Future f1 = pool.submit(new Runnable() {
        public void run() {
            File f1 = compute1(‘‘pic.jpg’’);
        }
    });
    final Future f2 = pool.submit(new Runnable() {
        public void run() {
            f1.get();
            File f2 = compute1(‘‘pic1.jpg’’);
        }
    });
    final Future f3 = pool.submit(new Runnable() {
        public void run() {
            f1.get();
            File f3 = compute1(‘‘pic1.jpg’’);
        }
    });
    final Future f4 = pool.submit(new Runnable() {
        public void run() {
            f2.get();
            f3.get();
            File f4 = compute1(‘‘pic2.jpg’’,‘‘pic3.jpg’’);
            SwingUtilities.invokeLater(new Runnable(){
              public void run() {
                display(‘‘pic4.jpg’’);
              }
            });
        }
    });
}
```

As this application requires the management of dependences, using ExecutorService in this approach will result in deadlock in the same way as the SwingWorker scenario (in the case that all threads within the ExecutorService are executing a blocked task). The easy “solution” would be to instantiate a cached thread pool as opposed to a fixed thread pool; this way, a new thread is created whenever a task is submitted and there is not an idle thread available. Unfortunately, this will only lead to the threading problem discussed in section 2.4. To overcome this, the programmer will be required to implement separate thread pools for blocking tasks and non-blocking tasks, or write their own implementation of the ExecutorService. Furthermore, the code legibility has decreased due to the amount of additional code introduced.

3 Parallel Task

This section presents an object-oriented task parallel tool called ParaTask[1] implemented in Java. ParaTask is available for download[2], including an optional Eclipse plug-in. We believe presenting the features piece-by-piece using smaller digestible code snippets will best help the reader understand the different features. But first, to inspire the reader while the above typical parallelisation approaches are fresh in mind, the ParaTask solution is presented:

```
public class ImageApplication {

    ...
    TASK public File compute1(String file) { ...  }
    ...
    public void actionPerformed() {
        TaskID<File> id1 = compute1(“pic.jpg”);
        TaskID<File> id2 = compute2(“pic1.jpg”) dependsOn(id1);
        TaskID<File> id3 = compute3(“pic1.jpg”) dependsOn(id1);
        TaskID<File> id4 = compute4(“pic2.jpg”,“pic3.jpg”)
                                                dependsOn(id2,id3)
                                                notify(display(TaskID));
    }
    public void display(TaskID<File> id) {
        File result = id.getResult();
        ...
    }
    ...
}
```

3.1 Unifying task concept

The traditional role of a thread was to act like a virtual processor in order to introduce (conceptual) concurrency. Today, threads are also routinely used to exploit parallelism, but the virtual processor origin inhibits and limits the program designer significantly. The concept of tasks [8], on the other hand, is a much more powerful approach to exploit parallelism in a program. Conceptually, it targets the program or code section, rather than focusing on the executing parallel system. Recognising this advantage, task frameworks have been introduced in some high-level programming environments (e.g. [16, 17]). Unfortunately, they are mostly introduced as add-ons alongside the existing thread concept.

In our proposed task concept, we have thoroughly analysed existing programs, their use of threads and potential parallelism. The unifying task concept we propose integrates the role of (parallel) tasks together with the concurrency of threads (e.g. for latency hiding). This is not only done for task parallelism, but also for data parallelism. Despite the universal nature, our parallel task concept ParaTask is surprisingly simple. Let us have a look at the different task types we identified.

Different types of tasks

To cover the above described concepts we only need three task types:

1. *One-off tasks*
   These tasks are CPU-bound computations. When invoked, a single instance of the task is enqueued to be executed from start to finish by any of the processors.

[1] Preliminary proof-of-concept development of ParaTask was presented here [15]
[2] To download the latest ParaTask and other related resources, visit `www.parallelit.org`

2. *Multi-tasks*
   These are multiple tasks for data parallelism, hence they map to different processors.

3. *I/O tasks*
   These tasks are I/O-bound computations, for example background tasks waiting for events (e.g. mouse or key press). They should not be defined as one-off tasks since they would cause a backlog of ready-to-execute tasks [8]. Many tasks are perfect candidates for I/O tasks, for example web-based tasks. These tasks correspond to classical threads.

We believe that these three task types cover the large majority of the cases where tasks and threads have been used so far. In the next section we will see how we unify them into one single concept.

3.2 Syntax and semantics

The three task types presented in section 3.1 are unified by ParaTask. They are implemented with a single keyword, only using small modifiers to make them multi or I/O. For illustration, the same image application example from figure 1 is parallelised. To define a one-off task, the programmer annotates the method declaration with the `TASK` keyword:

```
public class ImageApplication {

    TASK public File compute1(String file){
        ...
    }
    ...

}
```

The `TASK` keyword acts as a modifier to a method declaration. This one-off task may now be invoked like a typical method:

```
TaskID<File> id = compute1(“pic.jpg”);
```

The only difference to the standard method invocation is that a `TaskID<`*`returnType`*`>` object is always returned, even if the original method signature has a `void` return type. All tasks have a unique global ID, accessed using `ParaTask.globalID()`. How to employ multi or I/O tasks is discussed in section 3.2.2.

3.2.1 Semantics of a task (as opposed to a method)

Throughout this paper, any method annotated with the `TASK` keyword is referred to as a *task* (to distinguish it from a standard *method*). The significance of the `TASK` keyword is:

- *Asynchronous execution*
  A task is essentially separated into two components: task *enqueuing* (or *invocation*) and task *execution*. After a task is enqueued, its execution is always asynchronous with the caller (i.e. the task is executed by concurrent mechanisms of ParaTask's runtime system). Consequently, parallelism has been introduced by a single keyword. Programmers do not need to restructure the code, wrap sections of code in `Runnable` objects, or create and manage threads.

- *TASK keyword as a form of documentation*
  Since the `TASK` keyword is associated with the method declaration, it denotes that the method code is task-safe[3]. This places responsibility upon the task *implementer* to ensure safe asynchronous execution. As a result, *user*s of tasks are relieved from this burden of determining whether a task is parallel-safe.

[3]The term *task-safe* (as opposed to *thread-safe*) is used to denote parallel-safe code since ParaTask is a tasking-model (not a threading-model). This means that programmers think in terms of tasks rather than threads (explained further in section 4.2.2).

By combining these two points, the usefulness of tasks is especially evident for event handlers: invoking a task from an event handler (such as Java's EDT) will enqueue it for concurrent execution by another processor. This frees the event handler to respond to other events, supporting a responsive GUI. This also means that tasks must not contain GUI code since GUI components may only be accessed from the EDT (section 4.2.5 presents ParaTask features that support GUI-related work).

3.2.2 Different task types

Multi-tasks
Multi-tasks support the concept of data parallelism or Single Program Multiple Data (SPMD), where the same task is executed multiple times. There are differences between invoking a multi-task once versus invoking a one-off task multiple times:

- A multi-task provides better documentation since the programmer is aware of the intention to execute it multiple times.
- The subtasks of a multi-task map to different processors in a round robin fashion (static scheduling) as opposed to all going to the same queue (dynamic scheduling).
- A multi-task has group awareness; this can be used for efficient partitioning or scheduling of the workload and allow operations such as reductions to be performed.
- Multi-tasks make traditional data parallelism more obvious, and has less time overhead than a tasking approach (section 7.9).

Whereas a one-off task is annotated with `TASK`, a multi-task is annotated with `TASK(*)` meaning it is created multiple times. How many times is determined by the ParaTask runtime system and usually corresponds to the number of available processors/cores. Alternatively, annotating the multi-task with any integer n (instead of using *) will create n tasks (n may be an integer constant or integer variable name). In addition to global IDs, the following concepts are useful for multi-tasks:

- *multi-task size* represents the number of subtasks within a multi-task, and
- *relative ID* is the ID of the subtask with respect to the other subtasks in the same multi-task (starting at 0 and ending at `multiTaskSize-1`).

The following is an example of a multi-task:

```
TASK(*) public String multiTask() {
    int myPos = CurrentTask.relativeID();
    int num = CurrentTask.multiTaskSize();
    if ( myPos == 0 )
        print(“There are ”+num+” subtasks.”);
    String name = “subtask”+myPos;
    print(“Hello from ”+name);
    return name;
}
```

A multi-task is enqueued the same way as a one-off task, except that a `TaskIDGroup<`*`returnType`*`>` is returned:

```
TaskIDGroup<String> multiID = multiTask();
```

And the individual `TaskID`s of the subtasks may be traversed:

```
Iterator<TaskID<String>> it = multiID.groupMembers();
while (it.hasNext()) {
```

```
        TaskID<String> task = it.next();
        String result = task.getResult();
        ...
    }
```

I/O tasks
I/O-bound computations that block waiting for external events should not be enqueued on the global task pool. Tasks with such characteristics should be identified by the task's developer in order to preserve encapsulation (the user of the task should not need to know any implementation details). ParaTask enforces this responsibility by requiring such tasks be annotated using the `IO_TASK` keyword rather than the standard `TASK` keyword. Consider the following task that retrieves information from the web:

```
IO_TASK public File webSearch(String query) {
    // blocking and long waiting times
}
```

This I/O task is used in the same manner as an ordinary task:

```
TaskID<File> id = webSearch(“foo”);
```

I/O tasks start their concurrent execution immediately as soon as all `dependsOn` dependences (section 4.1), if any, have been satisfied. They are not put into the normal queue of the task pool. Of course, they have to (time-)share the physical processors with the task pool execution, but their concurrent execution improves the responsiveness of important I/O bound activities and the time-sharing frees the processors for computation bound tasks. Many tasks are perfect candidates for I/O tasks, for example web-based tasks: these tasks should start as soon as possible and should not block the queue of the task pool.

3.2.3 Private and shared variables

When a programmer implements a task, variables will be used to define the task. Since a task is executed concurrently with its caller (and also concurrently with other tasks), it is important that the variables are used in a task-safe manner. Consider the following example code:

```
public class ImageApplication {
    private int numberOfErrors;
    ...
    TASK public String concatenate(List argsList) {
        String tempStr = “”;
        ...
    }
}
```

Programmers should already be familiar with the following concepts since they are standard programming scope policies:

- *Private variables*
  Local variables defined within a task are immediately task-safe since their scope is limited to the task (for example `tempStr`). These task-private variables make it easy to reason about the task-safety of the task.

- *Shared variables*
  Variables defined outside a task (for example `numberOfErrors`) have a larger scope that allows them to be accessed by many tasks and methods (therefore no need to restructure code into separate classes); but, as usual in parallel programming, developers must regulate the access to such variables to ensure their task-safety. A subtle point that programmers should be aware of is that task arguments passed as references (for example `argsList`) are also shared since other code segments may have reference to the object instance.

3.2.4 Asynchronous versus synchronous

If programmers wish to have a choice at runtime whether to invoke a method asynchronously or synchronously, this is easily achieved with ParaTask by letting the asynchronous version (with a TASK in front) call the sequential version:

```
public String compute(int input) {

    ...

}
TASK public String computeTask(int input) {

    return compute(input);

}
```

3.2.5 Nested parallelism

ParaTask allows nested parallelism, defined as a task enqueuing other tasks; this is especially important for recursive divide and conquer applications, e.g. merge-sort. Consider the following nested parallelism example:

```
TASK public void taskA() {

    TaskID id1 = taskB();
    TaskID id2 = taskC();

}
```

In this example, the parent task (`taskA`) enqueues the children tasks (`taskB` and `taskC`). Since ParaTask is an asynchronous model, the parent task is considered "complete" possibly before the inner tasks even start. This has the potential for more concurrency and performance. Some threading models, on the other hand, are *fully strict* [18] and would include an implicit barrier at the end of `taskA`. Although this simplifies reasoning about the behaviour of the program, it unfortunately blocks the current method. For example, the `actionPerformed` method of any interactive Java application must never block waiting for tasks it spawned. If programmers want the current method to block until all inner tasks complete, they may explicitly code this just like they would have to do with a threading library (section 4.2 discusses various solutions to easily achieve this).

4 Synchronisation

The `TASK` keyword ensures that the *concurrency* is coordinated by the callee, by declaring computations as independent and task-safe. The caller, however, will coordinate multiple independent tasks to suit the respective specifications of the application. For example imagine a library consisting of various image filters, each defined as a task. Users of this library may wish to coordinate multiple filters, defined as tasks, to create a new image filter specific to the application. For this reason, ParaTask supports the following clauses below, coordinated by the caller.

4.1 Dependences

In sequential programs, ordering constraints are naturally obeyed since code segments are executed one at a time in the specified order. Figure 1 requires that `compute2` and `compute3` not commence until `compute1` is complete, and similarly `compute4` must wait for both `compute2` and `compute3`. In the sequential code snippet of section 1, these constraints were naturally observed since invocation of these methods is synchronous with the caller.

However, if these methods were asynchronous (i.e. annotated with the `TASK` keyword), then their execution would overlap and possibly violate the ordering constraints. Specifying such ordering constraints in ParaTask is made simple by stating the dependences at the time the task is invoked:

```
public void actionPerformed(){

    TaskID<File> id1 = compute1(“pic.jpg”);
    TaskID<File> id2 = compute2(“pic1.jpg”) dependsOn(id1);
    TaskID<File> id3 = compute3(“pic1.jpg”) dependsOn(id1);
    TaskID<File> id4 = compute4(“pic2.jpg”,“pic3.jpg”) dependsOn(id2,id3);
    ...

}
```

Tasks 2 to 4 cannot proceed until the tasks specified in the `dependsOn` wrap have completed. The programmer does not need to manually code synchronisation mechanisms to manage such dependences; in this example, no synchronisation mechanisms such as barriers or wait conditions are required at all. This ensures no coupling exists between the tasks.

Since dependences must be specified at the time a task is invoked, this ensures that cyclic dependences are not possible (a task invocation cannot depend on a future task yet to be invoked). If the dependsOn keyword is supplied a null TaskID, this will result in a runtime exception.

4.2 Task completion

A task is essentially a method that is executed asynchronously with the caller. Accessing the return value of a task (or simply synchronising with the task's completion) may be achieved using a blocking (section 4.2.1) or non-blocking approach (section 4.2.5). The definition of a *blocking* function refers to the situation where the computation (or code) following the function will not be executed until the function has completed. In other words, the caller is unable to progress until the function returns.

4.2.1 Blocking on a TaskID

Continuing on the image application example, the programmer may wish to access the result of the task invocation:

```
File finalImage = id4.getResult();
```

If the task has not yet completed, then the function `getResult()` will block until the task has completed, i.e. it will only return after the task completes. No code textually following this function will be executed until then. This is the common approach taken by implementations of the *future* concept [19]. This blocking method provides a convenient synchronisation mechanism for the programmer, and the blocking is generally acceptable so long as the waiting is for a short length of time. As discussed below, the programmer should be aware where the blocking is happening.

Blocking from within an event handler

Blocking has the disadvantage that the caller cannot progress until the task has completed. This impact is especially harmful when it is an event handler blocking (such as the EDT). Since this could cause a backlog of waiting events, programmers are encouraged never to block from an event handler. This rule of thumb not only applies to blocking on a `TaskID`, but any blocking in general made in an event handler [6]. Section 4.2.5 presents an alternative mechanism to support non-blocking notification of task completion.

Blocking from within another task (nested parallelism)

ParaTask supports nested parallelism: namely, tasks may invoke other tasks. The reason this is a special scenario is because tasks might block on the `TaskID` of another task. In any tasking system, there is a limited number of tasks that can be active at the same time[4] and typically corresponds to the number of available processors/cores. When the maximum number of active tasks is reached all other tasks that are ready to execute are enqueued until one of the currently active tasks completes.

This poses a potential problem (namely deadlock) if all active tasks block indefinitely. For example, consider the following recursive divide and conquer task. It executes a sequential algorithm for small inputs up to a certain cutoff, while larger inputs are divided and the task is called recursively (effectively creating more tasks). The current task blocks until results from the subtasks are complete, at which point the final answer is returned:

```
TASK List mergeSort(List nums) {

    if (nums.size() < cutoff)
        return sequentialSort(nums);
    int middle = nums.size() / 2;
    TaskID left = mergeSort(nums.subList(0,middle));
    TaskID right = mergeSort(nums.subList(middle,nums.size());
    return merge(left.getResult(), right.getResult());

}
```

If all active tasks block waiting for the sub-results, deadlock will occur. ParaTask takes the following approach to avoid deadlock: whenever a task blocks on a `TaskID`, it executes another task from the ready queue (the next task it receives is dependent on the scheduling scheme, section 6.3, and may not necessarily be the particular task it is waiting on). Here this happens inside the `getResult()` methods. Figure 3 shows an example execution of the above program. It is assumed that there are two worker threads (corresponding to two available processors) working on the task pool. When the first task (executed by worker thread 1) creates tasks B and C, the second worker thread starts executing task B, which creates tasks D and E. When task A blocks waiting for tasks B and then C to complete, the first worker thread gets a new task to execute: it finds task C is the next ready task. Tasks F and G are now created and enqueued. In the meantime, when task B blocks first on D and then on E, the second worker thread starts executing the next ready task: this happens to be task E, and similarly task D. When the leaf tasks are complete, the worker threads return to their original task: deadlock has just been avoided. In order to suspend a task, the work (of the newly acquired task) is continued in the getResult() method, hence the context stays in the stack (i.e. all supported in the runtime).

4.2.2 Blocking on non-ParaTask data structures

In the case that a task blocks on a `TaskID`, then ParaTask determines this and automatically substitutes another task to execute as discussed above. As with any parallel programming environment, programmers should be careful with blocking on data structures that ParaTask has no knowledge about (e.g. concurrent data structures and locks from other libraries). Any form of blocking on a non-ParaTask object should be avoided while inside a one-off task or multi-task. At best, a blocked task will prohibit other ready tasks from executing (reducing performance). At worst, multiple tasks blocked will result in deadlock. To avoid these problems, programmers should use I/O tasks instead.

Thread-safe is not necessarily task-safe

The example of figure 3 illustrated an important point: it is not guaranteed which worker thread of a task pool executes a given task and it could potentially be switching between multiple tasks

[4] I/O tasks are an intentional exception, but not a general solution to the here discussed issue.

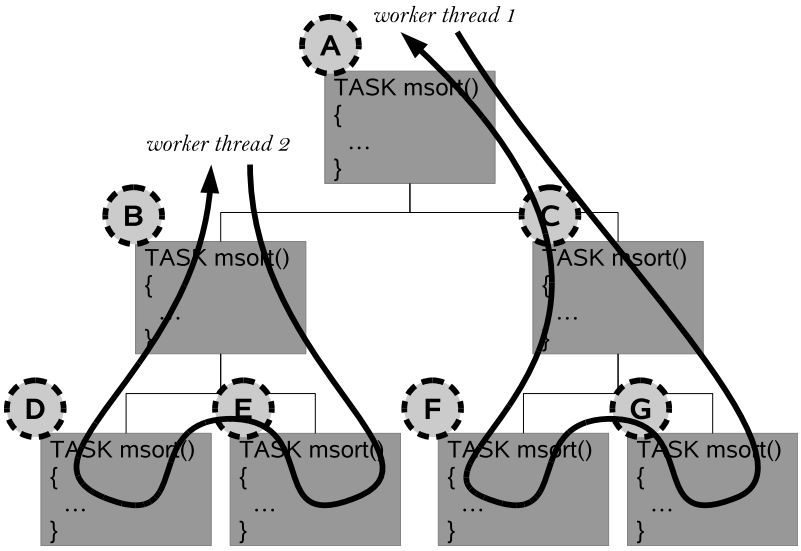


Figure 3: Visualisation of how ParaTask avoids deadlock, by executing other ready tasks when a worker thread blocks on a `TaskID` for a task that has not yet completed.

(when it blocks on an incompleted `TaskID`). For this reason, programmers should be careful when using certain mechanisms:

- *Thread-local storage*
  Using thread-local variables are discouraged since tasks might not execute atomically on a worker thread. In fact, thread-local storage generally does not make sense in a tasking model such as ParaTask. This is because the programmer is not guaranteed which worker thread will execute a particular task (unlike a threading model where the programmer explicitly controls the threads executing methods).

- *Locks and condition variables from threading libraries*
  Since semi-executed tasks might be substituted with another task, locking inside of tasks could lead to nested locking [12]. This poses a potential problem as shown in the following simple example:

```
TASK public void taskA() {

    mylock.lock(); // start critical region
    ...
    TaskID id = taskB();
    id.waitTillFinished();
    ...
    mylock.unlock(); // end critical region

}
TASK public void taskB() {

    mylock.lock(); // start critical region
    ...
    mylock.unlock(); // end critical region

}
```

Consider the worker thread executing `taskA` that has been granted exclusive access to `mylock`. Consequently, this worker thread has just entered the critical region and assumes it has exclusive access until it unlocks at the end of `taskA`. Within this critical region, it blocks on another

task. Since `taskB` is still waiting to be executed, the current worker might end up executing it (as discussed earlier). Consequently, the same worker thread has entered the critical region of `taskB` (since the worker thread already holds the lock to `mylock` [12]). In some situations, this might not be the desired behaviour since the critical region of `taskA` has not been exited yet.

4.2.3 Canceling a task

ParaTask uses a fully cooperative model [20] and allows tasks to be safely canceled:

```
TaskID id = myTask();
...
boolean canceled = id.requestCancel();
```

The `requestCancel()` method may be called on a `TaskID` in an attempt to cancel it. If `true` is returned, ParaTask guarantees the task has not started executing yet and will therefore not be scheduled to execute in the future. If `false` is returned, this means the task has already started executing, or the task has already completed. If the task is already executing, then `requestCancel()` has no effect unless the task periodically queries its own status to check if a cancel has been requested:

```
TASK public void myTask() {
    ...
    if (CurrentTask.cancelRequested())
        return;
    ...
    if (CurrentTask.cancelRequested())
        return;
    ...
}
```

4.2.4 Synchronisation with multi-tasks

Multi-task barrier
Synchronisation inside SPMD is a common and useful idiom in parallel computing and cannot be neglected. In the context of ParaTask, this particularly applies to multi-tasks when synchronisation is required between the subtasks. For example, consider the following situation:

```
// multi-task
TASK(*) public void A() {
    // perform computation
    ...
    // synchronise with sibling subtasks
    CurrentTask.barrier();
    ...
    // continue
}
```

Here, ParaTask supports a synchronisation barrier for multi-tasks, called `CurrentTask.barrier()`. This function is a barrier in the sense that the subtask does not progress past it until the other subtasks also reach it. Every ParaTask multi-task has its own barrier, initialised to the size of the multi-task. When the barrier has been reached, it is automatically reset to allow for further synchronisations within the same multi-task.

Reductions
A reduction is a standard parallelisation problem [21] where threads calculate a partial result that need to be *reduced* into a final result. Consider the following multi-task:

```
TASK(*) public int sum(ParIterator<Integer> itr) {
    int s = 0;
    while (itr.hasNext()) {
        s += itr.next();
    }
    return s;
}
```

The `ParIterator` parameter refers to the Parallel Iterator concept [22], which is essentially a thread-safe iterator allowing multiple threads to traverse it concurrently. By combining the Parallel Iterator with ParaTask multi-tasks, the programmer easily calculates and reduces the sum using multiple tasks:

```
1: List<Integer> list = ...;
2: ParIterator<Integer> pItr = ParIterator.create(list);
3: TaskIDGroup<Integer> sumID = sum(pItr);
4: int finalSum = sumID.reduce(Reduction.IntegerSUM);
```

Line 2 creates a `ParIterator` instance for the list of numbers, which is then passed as an argument to the multi-task (line 3). Consequently, the multi-task has been enqueued and each of the subtasks will share the `ParIterator` instance. Line 4 performs a reduction on the results from the multi-task. The real power of ParaTask comes out as more complex reductions are handled elegantly. The programmer only needs to focus on the business logic of their application since many parallelisation concerns (e.g. the concurrency and reduction) are hidden.

ParaTask provides a range of common reductions (e.g. sum, minimum, maximum). ParaTask also allows the programmer to define customised reductions; this object-oriented solution allows any kind of reduction while using any data type. The reduction must be *associative* (the order of evaluating the reduction makes no difference) and *commutative* (the order of the task-local values makes no difference) since the interface does not specify order. Customised reductions are easily composed by providing an object that implements the `Reduction` interface, defining the reduction of two elements into one:

```
public Reduction<Shape> biggestShape = new Reduction<Shape>() {
    public Shape reduce(Shape a, Shape b){
        if (a.getArea() > b.getArea())
            return a;
        else
            return b;
    }
};
```

4.2.5 Non-blocking: The `notify` clause

As discussed in section 4.2.1, event handlers should never block waiting for a task's completion. It is important for control to remain in the event loop so that events are handled promptly [23]. While blocked, the handler cannot process any pending events (for example, stalled repaint events will produce a frozen GUI).

Consequently, ParaTask provides non-blocking task synchronisation: when the programmer invokes a task, a comma-separated list of *slots* may be specified using a `notify` clause. Slots (and

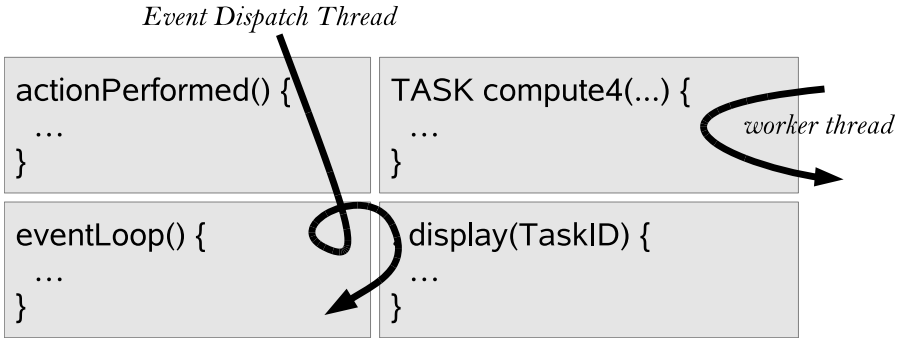


Figure 4: Using the ParaTask's `notify` clause allows the enqueuing thread to later synchronise with completion of the task without blocking. This is especially important for event handlers.

signals) are a modern object-oriented concept for communication [24, 25]. Essentially they are ordinary methods that get called ("signaled") automatically when the task has completed, meaning that no legacy thread or task is blocked while waiting for the task to complete. The notify clause requires the signature of the slot (i.e. including the argument type); if the task returns a value, the developer may `notify` a slot accepting a `TaskID` parameter (allows the result to be accessed from within that method). Adding to the code of section 4.1, the example initially introduced in figure 1 is completed:

```
public void actionPerformed() {

    TaskID<File> id1 = compute1("pic.jpg");
    TaskID<File> id2 = compute2("pic1.jpg") dependsOn(id1);
    TaskID<File> id3 = compute3("pic1.jpg") dependsOn(id1);
    TaskID<File> id4 = compute4("pic2.jpg","pic3.jpg") dependsOn(id2,id3)
    notify(display(TaskID[5]));
    // event-handling thread does not block

}
```

The `display()` slot is defined as an ordinary method:

```
public void display(TaskID<File> id) {

    File result = id.getResult();
    ...  // access GUI

}
```

This example is illustrated in figure 4. Since `actionPerformed()` is an event handler, it is executed by the EDT. Consequently, the EDT is the *enqueuing thread* of the `compute4()` task: the task pool executes this task while the EDT is immediately allowed to return to the `eventLoop()` to process other events. The EDT is informed by the task pool (via the event loop) when the task completes, therefore allowing `display()` to be executed.

`notify` on nested tasks

Since the purpose of the `notify` clause is to callback slots back to the enqueuing thread, the same applies in nested tasks. In this case, notifies of nested tasks are executed one after the other by ParaTask's runtime system. In the situation where the programmer wishes to notify the EDT (rather than ParaTask's runtime system) from within a task, this may be achieved in the usual manner using SwingUtilities.invokeLater().

[5]The `TaskID` is needed if the method being notified accepts `TaskID` as parameter, otherwise it is omitted. This ensures at compile time that the correct method signature is used.

4.3 Interim progress and notifications

To improve the interactivity of an application, it is desirable to display a task's progress as it proceeds. For example, consider a task that retrieves photos from the web: it is undesirable to update the GUI only when all photos have been retrieved. Ideally, the task should update its progress (e.g. as a percentage) and display any retrieved photos (i.e. interim results) to the user. Recall from section 2.3 that only the EDT is allowed to access GUI components (e.g. the progress bar and other visual results). Achieving this is easy with ParaTask. First, methods interested in the task's progress (and interim results) are registered using the `notifyInterim` clause:

```
TaskID<List<Photo>> id = getPhotos(list) notifyInterim(updateSearchDisplay(TaskID,Photo));
```

The `notifyInterim` clause behaves much like the `notify` clause: the methods are processed in a non-blocking fashion by the enqueuing thread. The only difference is that methods inside the `notifyInterim` clause are invoked whenever the task publishes interim results:

```
 1:  IO_TASK public List<Photo> getPhotos(List<String> names) {
 2:      List<Photo> results = new ArrayList<Photo>();
 3:      for (int i=0; i<names.size(); i++){
 4:          Photo p = Flickr.getPhoto(names.get(i)); // web I/O
 5:          CurrentTask.setProgress((i+1)/names.size()*100);
 6:          CurrentTask.publishInterim(p);
 7:          results.add(p);
 8:      }
 9:      return results;
10:  }
```

In this example, `getPhotos()` is defined as an I/O task (line 1) since it retrieves results from the web. Each time a photo is retrieved (line 4), the task updates its current progress (line 5). This allows other interested components (e.g. `updateSearchDisplay()`) to query the progress of the task. Line 6 shows an interim result (the newly retrieved photo) being published. Since the publication of interim results is non-blocking, this allows the task to progress with the computation without waiting for the interim result being delivered. All methods registered with the `notifyInterim` clause are executed by the enqueuing thread. In this example, the `updateSearchDisplay()` is executed by the EDT (the enqueuing thread) to update the GUI every time a new photo is retrieved. For example, the progress bar is updated and a thumbnail of the new photo is displayed:

```
public void updateSearchDisplay(TaskID id, Photo p) {

    progressBar.setProgress(id.getProgress());
    thumbnailsPanel.add(p);

}
```

Using the `notifyInterim` clause allows the tasks to remain decoupled from the methods interested in interim results. For example, the `getPhotos()` task above has no knowledge about `updateSearchDisplay()` or any other method interested in the interim results. This allows other methods to be connected/disconnected without affecting the task. The reason the above example was not implemented using a multi-task was to keep the worker threads free. Similarly, multiple I/O tasks were not used since we wished to limit access to the Flickr API to a single connection. Unfortunately, static type checking is not working in the current ParaTask implementation in regards to the `notifyInterim`, but should be correctable in a full featured compiler.

5 Adherence to object-orientation

By introducing concurrency to an object-oriented language, one must discuss the impact this has on important object-oriented concepts; the three most important ones are encapsulation, inheritance and polymorphism [26]. Discussions on the inheritance anomaly and exception handling are also presented.

5.1 Encapsulation

Encapsulation, an important aspect of object-oriented programming, protects the attributes and methods of an object from improper use [26]. A major aspect of encapsulation is data hiding, since any irrelevant internal details are hidden. ParaTask promotes encapsulation in the following ways:

- *Concurrency is coordinated by the callee:*
  In ParaTask, concurrency is coordinated by the *callee* rather than the *caller* (the `TASK` keyword is a modifier associated with the declaration rather than invocation). Whether a task is safe to execute concurrently is the responsibility of the implementation. This preserves encapsulation for two reasons [14]. First, the implementation details of a task remain hidden from the caller. Second, modification of a task's implementation will not require modification of any caller code since the task signature remains the same.

- *ParaTask enforces policies of access specifiers:*
  The access specifiers (for example, `public`, `protected` and `private`) of the programming language are enforced by ParaTask. Not only does this include access to tasks, but also access to methods specified inside `notify` clauses and `asyncCatch` clauses (section 5.5).

5.2 Inheritance

Reusing code is very important in object-oriented programming. One of the ways this is achieved is by having classes inherit code from other classes, known as inheritance. From Java's point of view, a ParaTask task is essentially an ordinary method; consequently, tasks are seen as standard class members and are therefore inherited by subclasses. Tasks may even be overridden by subclasses, just as methods are overridden; the only requirement is that the `TASK` keyword is also carried down (or carried up), otherwise the program will not compile. Therefore, the `TASK` keyword is viewed as a modifier and is very much a part of the method's signature.

5.3 Polymorphism

Following on closely to inheritance, polymorphism is also a very powerful object-oriented concept. Multiple subclasses inherit the interface of a common super-class, allowing for each subclass to respond differently to the same method. As discussed in section 5.2, the `TASK` keyword is a modifier and subclasses must therefore be consistent. Even though the logic of tasks may be overridden, their synchronicity may not: methods will always execute synchronously and tasks will always execute asynchronously. Therefore, there is no confusion when programmers invoke tasks.

5.4 Inheritance anomaly

The integration of concurrency with object-oriented languages is said to introduce a new set of problems termed *the inheritance anomaly* [27, 28]. Although this is an interesting and important issue, it is not confined specifically to concurrent object-oriented programming. In fact, this section shows that the inheritance anomaly still affects sequential object-oriented programs: it is merely more visible and apparent for concurrent object-oriented programs. Therefore, a counter-example against the original definition of inheritance anomaly is presented, and then an improvement of the definition is proposed.

The argument made here is that the inheritance anomaly occurs when the (subclassed) *object* is responsible for maintaining correct usage of the object (rather than relying on the *user*). A similar example from [28] is re-used to explain this. Consider the following sequential code defining a buffer:

```
public class Buffer {

    ...
    public Object get() {
        if (empty)
            throw new NoSuchElementException();
        ...
    }
    public void put(Object v) {
        if (full)
            throw new IllegalStateException();
        ...
    }
    public int size() { ...  }

}
```

From the code above, it is evident that the *user* of the buffer object is responsible for its correct usage. For example, the user should not insert elements into a full buffer or attempt to withdraw an element from an empty buffer. The programmer now wishes to extend this buffer by introducing a new method `gget()` that works like `get()`, except that it may not be immediately executed after a `get()` (this is the same example of [28]):

```
public class HistoryBuffer extends Buffer {

    ...
    public Object gget() {
        return super.get();
    }
    // put(Object), get() and size() all inherited

}
```

In this case, the inheritance anomaly does not arise because the `put(Object)`, `get()` and `size()` methods are inherited. However, it is the responsibility of the *user* to ensure that `gget()` is used correctly (and not the responsibility of the *object* `HistoryBuffer`). The programmer now redefines the `HistoryBuffer` in such a way that it contains some error handling to ensure it is used correctly (rather than naively relying on the user):

```
public class HistoryBuffer extends Buffer {

    boolean afterGet = false;
    ...
    public Object gget() {
        if (afterGet)
            throw new IllegalStateException(“Cannot call after
            get()!”);
        afterGet = false;
        return super.get();
    }
    public Object get() {
        Object o = super.get();
        afterGet = true;
        return o;
    }
    public void put(Object v) {
        super.put(v);
        afterGet = false;
    }
    public int size() {
        int s = super.size();
        afterGet = false;
        return s;
    }

}
```

Note that the programmer has not introduced concurrency, yet the inheritance anomaly exists: `get()`, `put(Object)` and `size()` must be redefined! This counter-example illustrates that the inheritance anomaly is not specific to the introduction of *concurrency* within the object, but rather to the introduction of *responsibility of correct usage* within the object. Since synchronisation is a subset of this responsibility, this explains why the inheritance anomaly exists in concurrent objects. Therefore, we propose that the following definition is better suited for inheritance anomaly:

> The hindrance of inheritance when the *responsibility of an object's correct usage* is incorporated within the object rather than solely relying on the object's user.

For this reason, the inheritance anomaly is something that also affects sequential object-oriented programs: it is only *more likely* to occur for concurrent programs since task-safety in these programs will demand responsibility.

5.5 Exception handling

An exception is an event that diverts a program from its normal execution flow [29]. Many programming languages support exceptions to separate error-code from the user-code. This potentially produces more readable and efficient code since error handling is not integrated within the normal execution flow. Exceptions are especially important in object-oriented languages, hence exception handling with ParaTask is proposed in the following.

5.5.1 The Catch or Specify Requirement

Java exceptions are categorised into two main groups [30]:

- **Checked exceptions**
  These are exceptions that an application is expected to anticipate and recover from.
- **Unchecked exceptions**
  These are exceptions that an application is not expected to anticipate and recover from. Unchecked exceptions may further be broken down into:
  - *Errors:* These exceptions are external to the program, such as a system or hardware failure.
  - *Runtime exceptions:* These exceptions are internal to the program, such as a typical programming bug or logic mistake.

Since an application is expected to anticipate (and hence recover from) checked exceptions, how does Java enforce this? Code that might throw such an exception must conform to the Catch or Specify Requirement [30], otherwise the program will not compile. This requires the programmer to take one of two options:

- Surround the code with a try/catch block, or
- Use a `throws` clause for the current method to specify it throws such an exception.

5.5.2 Exceptions in an asynchronous model

Consider the following task that throws two checked exceptions:

```
TASK public int myTask() throws MyExceptionA, MyExceptionB { ...  }
```

Since this task throws checked exceptions, Java requires the programmer to follow the Catch or Specify Requirement. Unfortunately, the standard approaches to either "specify" or "catch" do not automatically extend to an asynchronous model. Consider this example of a programmer attempting to *catch* the checked exceptions in order to honor the Catch or Specify Requirement:

```
// incorrect exception handling for tasks
1:   try {
2:      TaskID id = myTask();
3:      ...
4:   } catch (MyExceptionA e) {
5:      myHandlerA();
6:   } catch (MyExceptionB e) {
7:      myHandlerB();
8:   }
9:   // asynchronous model:  execution continues...
```

Such a try/catch block around the method invocation only works in a synchronous model. But in an asynchronous model, such as ParaTask, the try/catch block is futile: the caller continues to progress past line 9, possibly before the execution of `myTask()` is started. Of course, one solution would be to block the caller (until `myTask()` completes). Although such blocking would honor the Catch or Specify Requirement, it would destroy the asynchronous and concurrent execution advantage of tasks.

5.5.3 Parallel semantics for exception handling

As shown above, using a standard try/catch block (or specifying exceptions in method signatures) will only produce the desired result in a *synchronous* model; it is therefore required to have parallel semantics for exception handling. In particular, this includes developing semantics for the Catch or Specify Requirement in an *asynchronous* model: ParaTask achieves this by using the non-blocking `asyncCatch` clause:

```
TaskID id = myTask() asyncCatch(MyExceptionA myHandlerA(TaskID),
                                MyExceptionB myHandlerB(TaskID));
```

Using this `asyncCatch` clause is the asynchronous equivalent to the sequential try/catch block. A ParaTask exception handler is a standard method. Like the `notify` clause, methods in the `asyncCatch` clause are executed by the enqueuing thread. The programmer may access the exception through the `TaskID`:

```
public void myHandlerA(TaskID id) {

    print(“Task ” + id.getID() + “ threw an exception:”);
    id.getException().printStackTrace();

}
```

The `asyncCatch` clause must be used when invoking tasks with checked exceptions, otherwise the ParaTask program will not compile. This ensures that the Catch or Specify Requirement is honored in the asynchronous model. The `asyncCatch` clause may also be used on any task invocation (to also catch unchecked exceptions, not just for tasks with checked exceptions).

5.5.4 Propagating up the “task call” stack

In understanding the semantics of exceptions in a parallel environment, it is important to distinguish between the synchronous and asynchronous models:

Synchronous exception handling

In the synchronous model, a single thread calls methods. As a new method is called, the method is added to the top of the thread's *call stack*. Since the methods are synchronous, the top of stack refers to the currently executing method. When a method is completed, it is removed off the call stack.

If the thread encounters an exception, it starts looking for an appropriate handler. If one is not found in the current method, then the previous method on the call stack is analysed and so on until an appropriate handler is found. If no handler was found after analysing the entire call stack, then the thread's default handler is executed. In most cases, this means the exception's stack trace is printed and the thread terminates.

Asynchronous exception handling

In a model with asynchronous method invocations, such as ParaTask, things are slightly different than the synchronous model. There are now two threads of interest: the thread *enqueuing* the task, and the thread *executing* the task. The problem is that each thread has its own call stack, modified independently of another thread's call stack. In particular, if an exception is encountered while executing a task, then which call stack should be analysed? The thread executing the task cannot analyse its own call stack since the previous method on the call stack did not enqueue the task. It also cannot analyse the call stack of the original enqueuing thread since this would have been modified since the task was enqueued.

So, what is the significance of this asynchronous model when it comes to exception handling? First, unhandled exceptions that escape a task are termed as *asynchronous exceptions* (to differentiate them from standard exceptions that escape a sequential method). Consequently, the notion of a *task-call stack* (as opposed to a *call stack*) is introduced: this task-call stack is essentially the

stack that is created due to nested parallelism (when a task enqueues another task). When an *asynchronous exception* is encountered, it is first checked if this is inside a nested task and if yes the task-call stack is traversed upwards to find an appropriate *asynchronous exception handler* (i.e. an `asyncCatch` clause):

```
 1: public void method() {
 2:   try {
 3:      TaskID id = taskA() asyncCatch(RuntimeException myHandler(TaskID));
 4:   } catch(Exception e){...}//ignored
 5:   ...
 6: }
 7: TASK public void taskA() {
 8:   TaskID id = taskB() asyncCatch(IOException fileHandler(TaskID));
 9:   ...
10: }
11: TASK public void taskB() throws IOException {
12:   ...
13:   // exception thrown
14:   ...
15:  }
```

If an `IOException` occurs at line 13, then `fileHandler(TaskID)` is called since an asynchronous exception handler was registered on line 8. However, if a `NullPointerException` occurs at line 13, then the asynchronous exception handler of line 8 cannot support this exception (since `NullPointerException` is not a subclass of `IOException`). Therefore, ParaTask determines that `taskB` was invoked from within another task (`taskA`), so ParaTask propagates the asynchronous exception up the task-call stack. It determines that when `taskA` was enqueued (line 3), an asynchronous exception handler was registered: this handler is therefore called since a `NullPointerException` is a subclass of `RuntimeException`. Access to a parent exception handler is through a reference in the child task, which thereby is not garbage collected in the case the parent task completes before the child task (section 3.2.5).

If ParaTask does not find an appropriate asynchronous exception handler (for example, assume line 13 threw a `ClassNotFoundException`), then the stack trace is printed and ParaTask continues to execute another task. This behaviour is equivalent to the EDT when it encounters an unhandled exception: the idea is that the entire application should remain responsive and not be terminated due to a single exception. Notice how the try/catch block of lines 2 and 4 are ignored by ParaTask, because such *synchronous exception handlers* are insufficient to handle *asynchronous exceptions* (even though `ClassNotFoundException` is a subclass of `Exception`).

Asynchronous "finally" clause?

If the `asyncCatch` clause is asynchronously equivalent to the `catch` component of the sequential try statement, then why is there no clause asynchronously equivalent to the `finally` component of the typical try/catch statement? The first conceptual point to understand is that the code-block that gets executed "finally" must not contain an implicit barrier (since the caller should not block). Therefore, to support a non-blocking "finally" clause is essentially the combination of the `asyncCatch` and `notify` clauses used *together*. In the following example, the specified slot will execute "finally" whether an exception occurs or not without blocking the caller:

```
TaskID id = task() notify(finallySlot(TaskID)) asyncCatch(Exception finallySlot(TaskID));
```

6 Implementation

When the enqueuing thread calls the task, this is essentially translated into an enqueue of the task to the taskpool. After the enqueue is completed, the caller continues execution. In the meantime, a worker thread will eventually execute the newly created task.

6.1 ParaTask source-to-source compiler

ParaTask applications are essentially standard Java applications with the additional use of a few keywords. Programmers develop ParaTask source files (`.ptjava` extension) that are then parsed by the ParaTask compiler into standard Java source files. The parser is generated using JavaCC (Java Compiler Compiler) [31], a popular parser generator for use with Java applications, originally developed by Sun Microsystems. To support ParaTask keywords, the stable and official Java 1.5 grammar released with JavaCC is extended. The ParaTask compiler performs one-to-one preprocessing, namely an equivalent `x.java` Java source file is produced for every `x.ptjava` ParaTask source file. Finally, all Java source files are compiled using any standard Java compiler. For convenience, an Eclipse plugin is available for programmers.

6.1.1 Parsing task declarations

Consider a ParaTask source file with the following declaration for a one-off task:

```
TASK public int myTask(String str) {
    /* user-code */
}
```

The ParaTask compiler translates the above into standard Java code:

```
 1: private Method _pt_myTask_String;
 2: public TaskID<Integer> myTask(String str) {
 3:   return myTask(str, null);
 4: }
 5: public TaskID<Integer> myTask(String str, TaskInfo tInfo) {
 6:   if (_pt_myTask_String not initialised) {
 7:      ...
 8:      Class[] params = new Class[] { String.class };
 9:      Class thisClass = ...  // Java reflection
11:      _pt_myTask_String = thisClass.getDeclaredClass(‘‘_pt_myTask’’, params);
12:   }
13:   Object[] args = new Object[] {str};
14:   if (tInfo == null)
15:      tInfo = new TaskInfo();
16:   tInfo.setMethod(_pt_myTask_String);
17:   tInfo.setTaskArgs(args);
18:   tInfo.setEnqueuingThread(Thread.currentThread());
19:   return Taskpool.enqueueOneOff(tInfo);
20: }
21: public int _pt_myTask(String str) {
22:   /* user-code */
23: }
```

The first thing to notice is that the original user-code has been moved into another method whose name has been changed (lines 21 to 23); the reflected `Method` variable of line 1 refers to this method that contains the user-code. Therefore, the name of the original task now refers to new methods that perform the enqueuing of the task. The first method, declared on line 2, is used in the case that no ParaTask clauses are used when the task is called.

In order to enqueue the task, reflection is used to retrieve the `Method` representing the user-code (lines 7 to 11). To reduce runtime overhead, ParaTask ensures this is performed at most once for every task (line 6). The `TaskInfo` object is used to save details necessary to invoking the task. This includes the task arguments (line 13), the actual user-code to execute, as well as recording the enqueuing thread. This is finally sent to the taskpool (line 19), which returns a `TaskID` after enqueuing the task.

Parsing multi-tasks and I/O tasks is similar. There is only a slight difference: rather than calling `enqueueOneOff` on line 19, `enqueueMulti` or `enqueueIO` is called respectively.

6.1.2 Parsing task invocations

If a task is invoked without specifying any ParaTask clauses, then the ParaTask compiler does not modify the invocation. This is because a task invocation is a valid method invocation from the Java compiler's point of view; the invocation of the task refers to the enqueuing of the task rather than the original user-code (line 2 of section 6.1.1). We now consider a task invocation making use of at least one ParaTask clause:

```
TaskID myID = myTask(“Hello”) (PT clause)+;
```

The ParaTask compiler creates a new `TaskInfo` object:

```
TaskInfo _pt_myID = new TaskInfo();
```

The `TaskInfo` is then populated with information regarding the respective ParaTask clause. Once all this information is stored, the final task invocation becomes:

```
TaskID myID = myTask(“Hello”, _pt_myID);
```

This call refers to the method on line 5 of section 6.1.1. We now discuss how the `TaskInfo` is populated for the respective clauses.

Parsing a `dependsOn` clause

Consider the following use of the `dependsOn` clause:

```
TaskID myID = myTask(“Hello”) dependsOn(id1);
```

Parsing a `dependsOn` clause is very simple. The ParaTask compiler only needs to produce the following:

```
_pt_myID.addDependency(id1);
```

Parsing a `notify` clause

The `notify` clause has a few components that need careful consideration:

```
TaskID myID = myTask(“Hello”) notify(update(TaskID), myObj.complete());
```

The first slot in the `notify` clause above, `update`, has a `TaskID` parameter and should be invoked on `this` instance (the default if no instance is specified). The second slot in the `notify` clause, `complete`, has no parameters but will be invoked on the instance `myObj`. In either case, ParaTask must enforce that both these slots may be accessed from the current scope. Below is the resulting code:

```
// notify(update(TaskID))
1:   Method _pt_myID_notify1 = PT.getDeclaredMethod(getClass(), ‘‘update’’,
new Class[] {TaskID.class});
2:   if (false)
3:      update(PT.dummyTaskID);
4:   _pt_myID.addNotify(_pt_myID_notify1,this);
// notify(myObj.complete())
5:   Method _pt_myID_notify2 = PT.getDeclaredMethod(myObj.getClass(), ‘‘complete’’,
new Class[] {});
6:   if (false)
7:      myObj.complete();
8:   _pt_myID.addNotify(_pt_myID_notify2, myObj);
```

The `PT.getDeclaredMethod` on line 1 is a helper function that internally uses Java reflection to retrieve the method. Note that in some cases, the method might actually be inherited from a super-class. The above example assumes that the enclosing method is an instance method (since it is making use of `getClass()`). If the ParaTask compiler determines the enclosing method is static, then the class is attained using alternative means.

Line 1 can only determine at runtime if the programmer correctly used the `notify` clause. Therefore, the purpose of lines 2 and 3 is solely to ensure correct usage of the `notify` clause. Line 3 is never executed, but is used for a compile-time check. If the programmer misspelled the method name, attempted to invoke a method with incorrect parameter type, or if the method is out of scope (e.g. declared private in a super-class), then the Java compiler will complain. The ParaTask compiler simplifies its duties by using the Java compiler to determine such problems. When using the Eclipse plug-in, the error messages (determined in the transformed code) are conveniently correlated back to the initial source files. Even when programmers are not using the Eclipse plug-in, the error messages are still understandable when viewing the transformed code.

Finally, the slot to `notify` is registered inside the `TaskInfo` instance (line 4). If the programmer does not specify an instance to invoke the method on, the default is `this` instance (or `null` if the enclosing method is a static method). The second slot is determined in a similar way (lines 5 to 8), but with the difference of finding the method within the class of `myObj` (rather than using the current class).

Parsing an `asyncCatch` clause

The `asyncCatch` also has a few aspects to it. In particular, ParaTask ensures that the programmer adheres to the Catch or Specify Requirement. Assume that the signature of a task declaration includes a `throws` clause, for example:

```
TASK public int myTask(String str) throws IOException {
    /* user-code */
}
```

This task declaration is parsed exactly as discussed in section 6.1.1: the `throws` clause in the method signature is also included in the output source code. Since `IOException` is a checked exception, then the programmer must now use the `asyncCatch` clause (the `asyncCatch` clause may also be used for general exception handling when invoking any task). Below is a corresponding task invocation written by the developer:

```
TaskID myID = myTask(‘‘Hello’’) asyncCatch(IOException handler(TaskID));
```

By specifying an exception handler to catch the `IOException`, the following code is produced by the ParaTask compiler:

```
1: Method _pt_myID_exc1 = PT.getDeclaredMethod(getClass(),“handler”, new
Class[] {TaskID.class});
2: if (false)
3:    handler(PT.dummyTaskID);
4: _pt_myID.addExcHandler(IOException.class, _pt_myID_exc1, this, false);
5: TaskID id = null;
6: try {
7:    id = myTask(“Hello”, _pt_myID);
8: } catch (IOException _pt_e) { /* unreachable try/catch block */ }
```

Notice how lines 1 to 3 are essentially identical to that produced when parsing a `notify` clause: these lines are used to ensure that the method specified in the exception handler exist (as discussed above). Line 4 registers the exception type, `IOException`, with the exception handler so that the ParaTask runtime knows which handler to invoke when an `IOException` occurs. The actual task invocation is now reconstructed as in lines 5 to 8. First, the `TaskID` declaration (line 5) is separated from the assignment (line 7) so that the `TaskID` instance stays in the same scope the programmer expects. The assignment is finally surrounded with a try/catch block, catching the exceptions specified in the `asyncCatch` clause.

The try/catch block of lines 6 and 8 is actually an unreachable try/catch: its purpose is solely to quiet the Java compiler (since a checked exception is thrown by `myTask`) to ensure that the programmer follows the Catch and Specify Requirement. Now assume that an `IOException` was in fact thrown by `myTask` at runtime. Regardless of where the enqueuing thread is when this exception is thrown (even if it passes line 8, after the try/catch block), ParaTask will catch the exception and invoke the asynchronous exception handler specified in the asyncCatch clause. If the programmer does not correctly use the `asyncCatch` clause to handle an `IOException`, then the resulting Java code will not compile.

6.2 ParaTask runtime system

The Java implementation currently consists of three possible scheduling policies for the runtime, discussed in the next subsection. The programmer may decide which is more suitable for their particular application. The motivations behind allowing this choice is to show the flexibility of how the underlying runtime system is independent of the ParaTask language syntax. In fact, the current implementation allows for more scheduling policies [32] to be added as plugins. Figure 5 overviews the ParaTask runtime system, regardless of the particular scheduling scheme to be used.

Task enqueuing
The first phase, naturally, starts when a task is enqueued by the enqueuing thread. During this phase, a `TaskID` is created in order to record the details of the task invocation. If the task `dependsOn` other tasks, it is stored with the **waiting tasks** (therefore, the task is not scheduled until its dependences have been met). Otherwise, the task is ready to execute: one-off tasks and multi-tasks are enqueued according to the scheduling scheme plugin (and any sleeping **worker threads** are woken up), while I/O tasks execute on a new **I/O thread**. As soon as the task is enqueued, the enqueuing thread continues to process other work: in this example, the enqueuing thread returns to the event loop to process other events.

Worker thread is ready
All tasks, except I/O tasks, are executed by worker threads. The worker thread continues to execute all tasks in its **private ready queue**. Once the private ready queue is empty, a task is taken from the scheduling scheme plugin. If the task is reserved for another worker thread (i.e. in the case of multi-tasks), it is queued to that worker thread's private ready queue. Otherwise the task is executed.

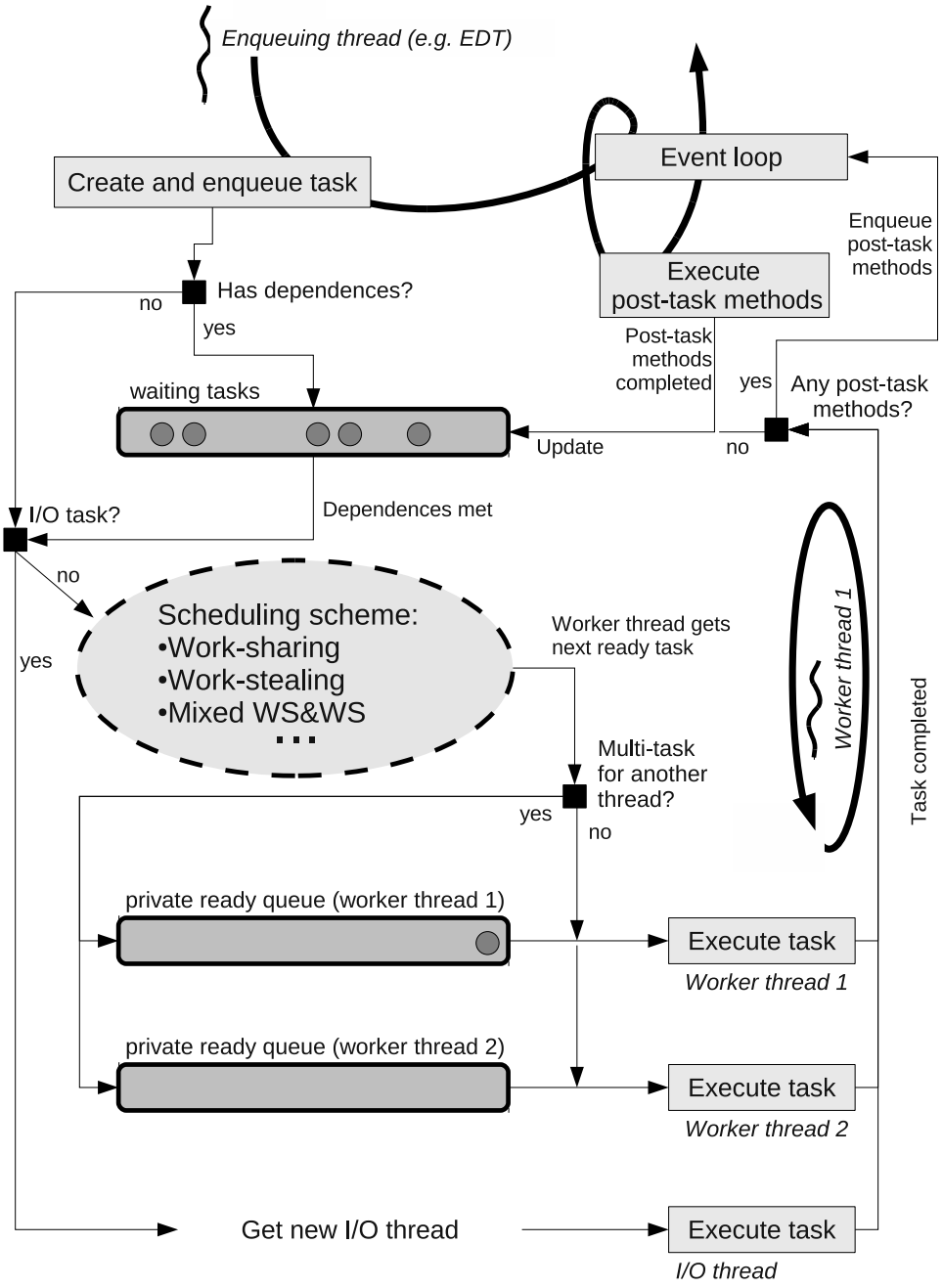


Figure 5: ParaTask's runtime system allows for different scheduling schemes to be plugged in. The life of a task starts when the enqueuing thread creates and deposits it for another thread to execute. The task is finally considered complete after any post-task methods (e.g. methods in a `notify` or `asyncCatch` clause) are executed.

Task execution
Java reflection is used to execute the user-code of the tasks. In most cases, the worker thread will execute the task in its entirety before executing another task. The exception to this is if a worker thread blocks on the `TaskID` of another task that has not yet completed (e.g. task B) while it is currently executing task A. In this case, the worker thread retrieves another task (e.g. task C) from the scheduling scheme plugin and executes it. When task C is completed, the worker thread checks the status of task B. If it has completed, then the worker thread continues where it left off with task A; otherwise, another task is executed again until task B is completed. This behavior is also repeated recursively if necessary (e.g. if task C in turn blocks on another unfinished task).

Task completion
When a task is finally executed by the worker thread (or I/O thread), it is not necessarily considered complete just yet. First, the worker thread checks to see if the task has any post-task methods that need to be executed (e.g. methods in a `notify` or `asyncCatch` clause). If no such methods exist, then the task is considered complete: the worker thread signals this by updating the task dependences.

If a task has post-task methods, then these need to be executed by the enqueuing thread (not the worker thread). Therefore, the worker thread signals the enqueuing thread (by emitting an event) that it should execute the respective post-task methods. When the enqueuing thread executes the post-task methods, a signal is sent to update the task dependences.

6.3 Scheduling schemes

The underlying scheduling scheme is crucial for the performance of a parallel application. Three scheduling schemes are currently supported, but more may be added as plugins. Developers may select the most suitable one for their application (one schedule is allowed per application). ParaTask's default schedule is *auto* (section 6.3.3) as it combines the benefits of *work-sharing* (best for fairness) and *work-stealing* (best for nested parallelism).

6.3.1 Work-sharing schedule

The first scheduling scheme is a simple work-sharing policy: tasks are executed using a fair policy (i.e. in the order they were originally enqueued). This is beneficial for *pipeline* pattern applications where the user expects older tasks to complete before newer tasks. Such a scheduling policy is important for perceived performance from the user's point of view (as demonstrated in section 7.11) since the user expects tasks to complete in the rough order they were launched (such as thumbnail previews). For example, consider an application that is expected to behave like the pipeline pattern:

```
for (int i = 0; i < data.size; i++) {

    TaskID id1 = stageA(data[i]);
    TaskID id2 = stageB(id1) dependsOn(id1);
    TaskID id3 = stageC(id2) dependsOn(id2);

}
```

The purpose of showing this pipeline pattern is to illustrate the scheduling of *dependent tasks* in ParaTask. This pipeline has three stages, and the output of each stage is used as input to the next stage. Notice that `stageA` is always ready to execute (it has no dependences), it is therefore possible to execute all the `stageA` tasks first (since all were ready). However, with the work-sharing policy, the tasks are prioritised according to their *original* enqueuing timestamp (rather than the time they were ready to execute). Of course, the actual order is not guaranteed (except when the `dependsOn` clause is used between tasks); however, the next favoured task to execute is the one with the earliest original enqueue. Implementing a work-sharing policy is straight forward. When tasks are enqueued to the scheduling scheme plugin, they are placed onto a **shared ready queue** using a first in, first (FIFO) out policy.

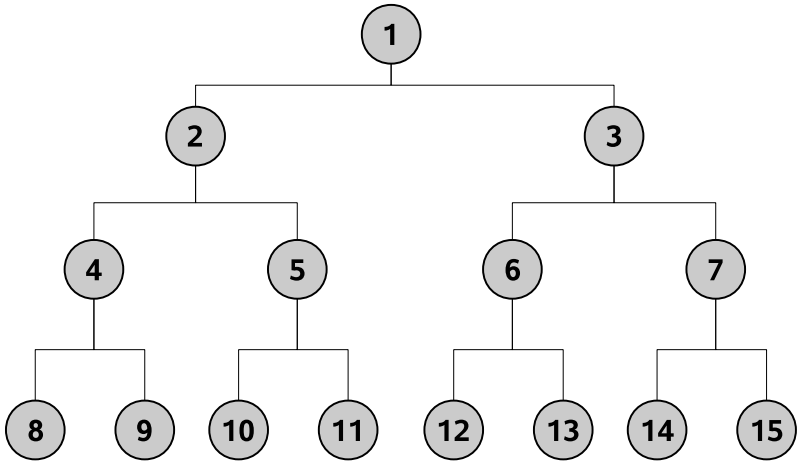


Figure 6: This recursive application is an example of nested task parallelism. In this case, a parent task is only considered complete when its children are completed. The numbers denote the time stamp of when the task was originally enqueued. To increase efficiency, tasks should not be executed in this same order (and even becomes impractical in larger applications).

6.3.2 Work-stealing schedule

The performance of some applications might suffer significantly with a work-sharing schedule (as will be shown in section 7). This is especially the case for recursive applications, for example those based on the *divide and conquer* pattern. Consider figure 6: the numbers denote the order the tasks were originally enqueued. If the tasks were executed in this same order (i.e. using the fair work-sharing policy of section 6.3.1), then this will result in extremely poor performance:

- Since each task is executed in the original order it was enqueued, this implies that the whole task tree needs to be in memory since a task is not considered complete until its children are complete. In other words, the number of pending, incomplete tasks keeps increasing until the last enqueued tasks are reached. Only then does the stack of pending tasks start to decrease again. Therefore the memory footprint is extremely high, especially for larger applications.
- Executing such an application in a breadth manner results in poor cache reuse [33, 34], since "colder" tasks are executed in preference to the latest tasks spawned.

Clearly, an intuitive schedule would be to execute the children tasks first (depth-first manner in order to complete the current task sooner); however, the work-sharing schedule of section 6.3.1 will instead jump branches and execute the tree in a breadth-first manner. For this reason, ParaTask also supports a work-stealing schedule [35]. This ensures a minimal stack of pending tasks, since it only consists of the depth of the task tree. The work-stealing variant used by ParaTask is based on the well-established randomised variant [36]. With work-stealing, recursive task execution never has large parts of the execution tree in memory as it works depth first.

Rather than having a single shared ready queue, the work-stealing runtime consists of a **local deque** (a double ended queue) for each worker thread. Note that this local deque is distinct from the **private ready queue**: the private queue is used to store only multi-tasks reserved for the owner thread, while the local deque stores all the task types and these may be stolen by other threads. When a thread operates on tasks on its own deque, a last in first out (LIFO) policy is used (therefore operating on the *latest* local node). When a thread's private deque is empty, it becomes a *thief* and selects a *victim* thread at random. The thief attempts to steal the *oldest* task on the victim's deque, therefore using a first in first out (FIFO) policy when stealing.

The benefit of work-stealing has been extensively developed and evaluated [37]. The reason that tasks are executed using a LIFO policy on the local deque is to discourage unnecessary parallelism by taking the initiative to execute a process towards the *depth* of the tree [35]; such behavior models that of a sequential depth-first traversal (therefore reducing parallelism overhead when

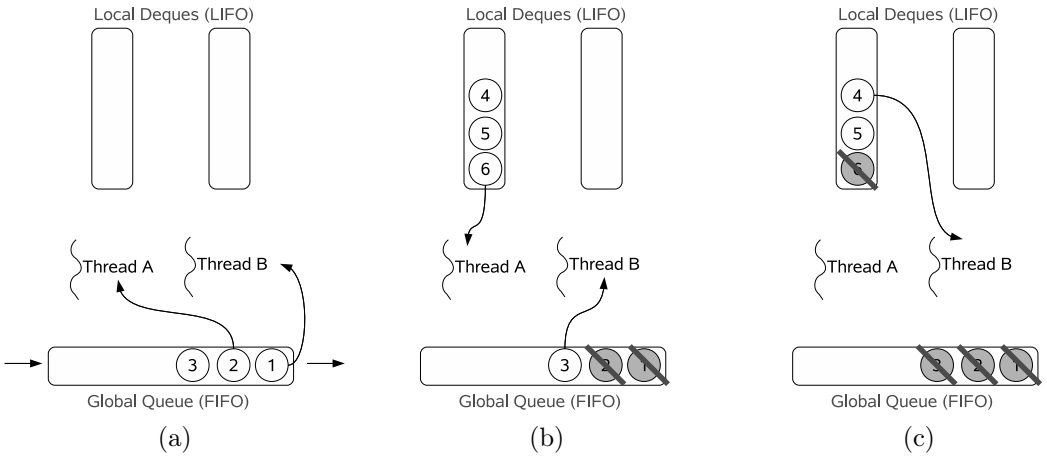


Figure 7: Auto schedule implementation. In (a), both threads are work-sharing. In (b), thread A starts work-stealing due to nested parallelism, while thread B continues work-sharing. In (c), thread B starts work-stealing as no more tasks remain on the global work-sharing queue.

sufficient work exists). Threads encourage parallelism when they steal with a FIFO policy, as this expands the *breadth* of the tree (the thief takes ownership of a new sub-branch). This naturally encourages good data locality and cache reuse [34] since thieves favor the victim's oldest task (i.e. the *coldest* task in the victim's cache), while the *hottest* tasks are left for the local thread. The difference in work-stealing is that Cilk implements a *work-first* policy, while ParaTask implements a *help-first* policy [38]; section 7.6 explains the difference between these policies, as the effect depends on the workload characteristics.

6.3.3 Auto schedule

What happens in an application containing a component that requires fairness, while another component involves highly nested parallelism? Which of the scheduling policies should be chosen? A work-sharing policy has the benefit of fairness (but fails for the nested parallelism component), while work-stealing has the benefit of aptitude for nested parallelism (but fails in fairness). ParaTask supports a combined scheduling scheme that defaults to work-sharing (i.e. is fair), but whenever nested parallelism (including recursive parallelism) is detected, these tasks are handled in a work-stealing manner. This overall policy ensures that ParaTask maintains fairness whenever possible, but temporarily (yet necessarily) resorts to a work-stealing schedule for nested parallelism components. As will be shown in section 7, this scheduling policy allows us to take the best of work-stealing and work-sharing.

Figure 7 illustrates the mixed work-sharing and work-stealing schedule, known as the auto schedule. Figure 7(a) shows 3 tasks enqueued using a FIFO policy behaving much like the work-sharing policy of section 6.3.1. In this example, task 2 happens to create 3 more tasks (i.e. nested parallelism). Since work-sharing is unsuitable for nested parallelism, these tasks are processed using the LIFO work-stealing policy of section 6.3.2 (figure 7(b)). Consequently, thread A must temporarily compromise fairness in favour of rescuing itself from the recursive parallelism. In the meantime, thread B strives for fairness by processing tasks from the global queue. When a thread finds no tasks in its local deque or the global queue, it steals a task from another thread (as discussed in section 6.3.2). For example, figure 7(c) shows thread B stealing the oldest task from thread A.

6.3.4 Memory consistency

In Java, a thread potentially works on non-volatile data for a long time without other threads seeing those changes until a synchronisation occurs. This important memory consistency is achieved in ParaTask with locks at important implicit synchronisation points, for example at end of tasks (before notification), when in asyncCatch handler, when returning a result and so on. This essentially

means that a task has flushed cached values at important synchronisation points.

7 Performance

The performance of ParaTask compared to traditional parallelism approaches is evaluated, as well as the overhead relative to sequential code and Java threads. In light of mainstream multi-cores, benchmarks typical of desktop applications are included as well as considering user-perceived performance. The benchmarks ran on a shared memory system which may be considered a typical future desktop platform running Linux. It has four Quad-Core Intel Xeon processors (total of 16 cores) running at 2.4GHz with 64GB of RAM. All benchmarks were coded in Java, and the sequential code of each benchmark forms as the baseline for all speedup calculations. The default JVM memory allocation was sufficient for most benchmarks except for those involving computation on files, where 16GB was allocated. Throughout all the benchmarks, Java's default garbage collector was used.

7.1 Overview

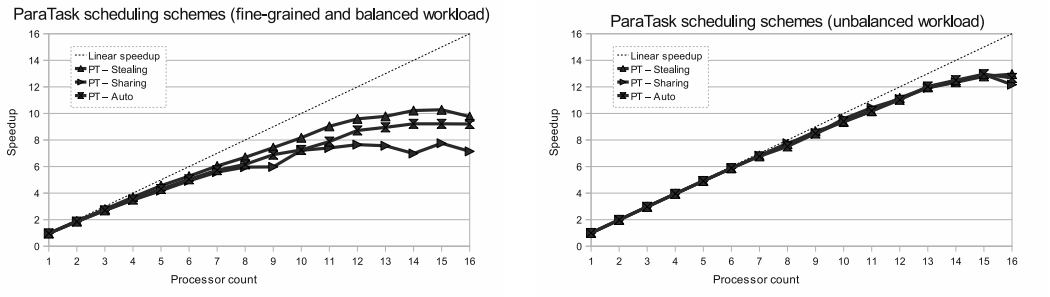
ParaTask's performance is compared across a number of typical Java parallelisation approaches a programmer may take, including:

- ***JavaThreads-max***: a new Java thread is created for *every* task. This is a typical approach programmers would take to manually parallelise an application.
- ***JavaThreads-min***: as a variation to JavaThreads-max, this involves creating the minimum number of threads in order to match the processor count. A static distribution of the tasks is created, where each thread is assigned roughly an equal number of tasks (but this might not necessarily equate to an equal workload at runtime if the tasks are unbalanced).
- ***SwingWorker***: each task is wrapped in a `SwingWorker` object. Being a tasking model, SwingWorker is an improvement to the runtime overhead of the JT-max threading model. As will be shown soon, SwingWorker tasks only map onto a maximum of 10 threads. Introduced in Java 1.6.
- ***ExecutorService-FixedPool***: each task is wrapped in a `Runnable` object and submitted to an ExecutorService, where the number of threads in the pool matches the processor count. Introduced in Java 1.5.
- ***ExecutorService-CachedPool***: each task is wrapped in a `Runnable` object and submitted to an ExecutorService. If there is no idle thread to execute an incoming task, then a new thread is created to execute it. Introduced in Java 1.5.
- ***ForkJoin***: each task is wrapped in a `ForkJoinTask` object (or a `RecursiveTask` object in the case of a recursive benchmark) and submitted to a ForkJoinPool. Introduced in Java 1.7.
- ***JCilk***: tasks are defined as methods and annotated with the `cilk` keyword. Although this model cannot be used for GUI applications, it is used in the following benchmarks to compare the work-first policy to ParaTask's help-first policy.

7.2 Benchmark applications

In sections 7.3 to 7.6, four benchmarks are used to compare the general ParaTask runtime with the other Java parallelisation approaches:

- ***Balanced and fine-grained compute-intensive***: involves computing a synthetic load (here the Newton-Raphson method) for each task. In this balanced workload, each task takes approximately 0.3ms. This benchmark tests the overhead of the various approaches.
- ***Unbalanced compute-intensive***: involves computing the same synthetic load, except tasks require varying amounts of computation. This benchmark tests the loading balancing of the various approaches.



(a) Fine-grained and balanced workload (b) Unbalanced workload

Figure 8: Comparing the three different ParaTask scheduling schemes using (a) a fine-grained and balance workload and (b) an unbalanced workload.

- ***Image resizing***: involves resizing a collection of 256 identical images, each 1MB (1600×1200 pixels), all stored on the same disk. The resizing of each image was designated as a single task, and no image was reused for another task in order to reduce cache effects. This benchmark is more realistic in terms of desktop application, as it requires high amounts of disk access.
- ***Word permutation***: involves a disk-intensive application, something typical of a spell checker. This benchmark consisted of 3505 different text files, stored on the same disk within a total of 172 sub-folders. The size of the files ranged from 22 bytes to over 100 KB, each file was designated as a task (unbalanced workload) without reusing any file for another task. Each task consists of reading the words inside a file and performing various string comparisons with the other words within the same file.

The benchmarks in sections 7.7 to 7.10 are targeted to compare ParaTask's various clauses; the particular benchmark will be discussed in each respective section. Finally, the benchmark of section 7.11 presents user-perceived performance in recognition that speed is not the only measure of interest in interactive desktop applications.

7.3 ParaTask versus ParaTask

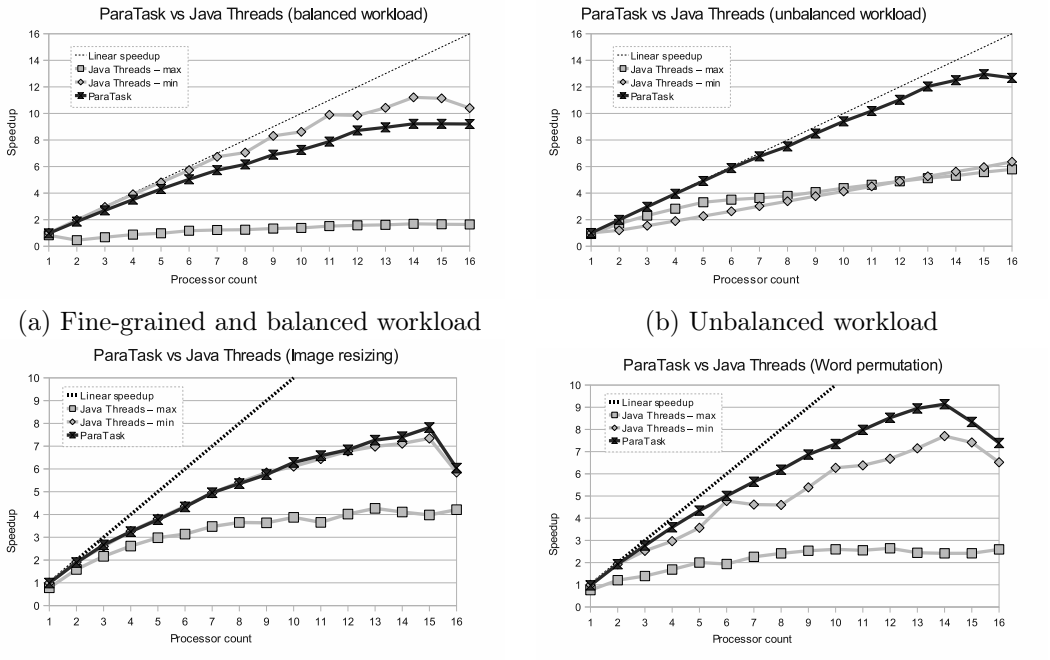
In this section, we compare the performance of the different ParaTask runtimes using the two compute-intensive benchmarks mentioned in section 7.2. Note that the three ParaTask approaches all use exactly the same coding approach; the only difference is the scheduling scheme selected at runtime.

Figure 8(a) shows the fine-grained and balanced workload with a clear distinction in performance amongst the different runtimes. Even though there is no nested parallelism in this benchmark, the reason that work-stealing performs better than work-sharing is because of the high contention. Work-sharing consists of a single queue for the tasks, therefore all threads contend on the same queue (they even contend at the same end of the queue). Work-stealing however, tasks are randomly distributed to multiple queues (in the case that tasks were enqueue by a non-worker thread). This not only means less contention because of more queues, but also because stealing threads take from the opposite end that the victim thread operates on. ParaTask's auto scheduling incurs slightly higher overhead compared to work-stealing because it checks for nested parallelism.

In the case of the unbalanced workload, figure 8(b), the specifics of the respective scheduling policies are less influential since the bulk of the ParaTask runtime is common (section 6.2). Consequently, the auto scheme is ParaTask's default scheduling scheme and is used for the benchmarks in the subsequent sections.

7.4 ParaTask versus Java Threads

Figures 9(a) to 9(d) compare the default ParaTask scheduling scheme to manually using Java threads (the JavaThreads-max and JavaThreads-min discussed in section 7.1). For balanced and fine-grained computationally intensive benchmarks (figure 9(a)), Java threads achieves the



(a) Fine-grained and balanced workload

(b) Unbalanced workload

(c) Image resizing

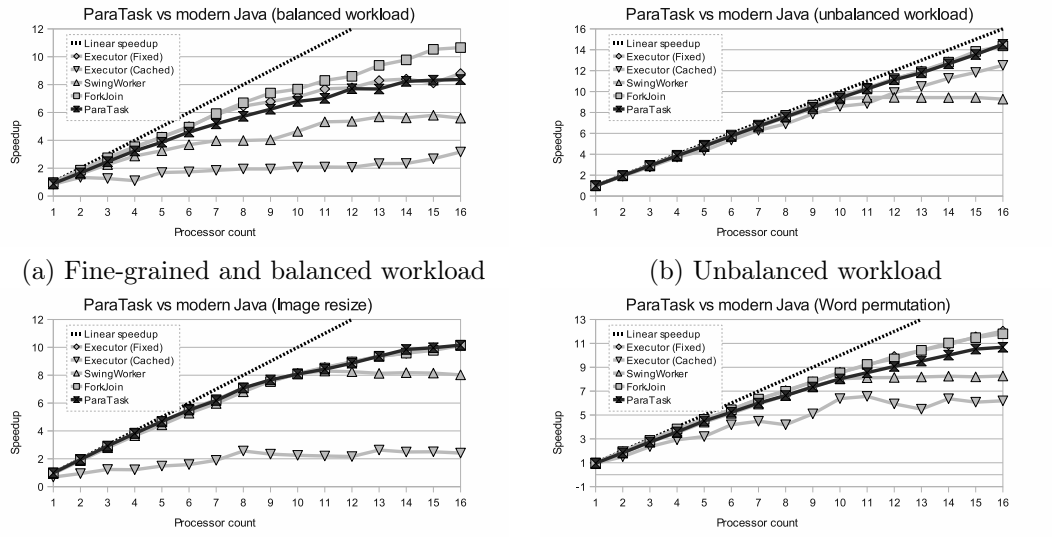
(d) Word permutations

Figure 9: Comparing ParaTask performance (using the default scheduling scheme) against low-level Java Threads with various workloads: (a) fine-grained and balance workload, (b) unbalanced workload, (c) image resizing and (d) word permutations.

best performance when statically allocating computations to the minimum number of threads (JavaThreads-min); if a thread is assigned for each computation (JavaThreads-max), this achieves the worst performance. Due to the fine granularity of this benchmark, it reveals the overhead of ParaTask's runtime scheduling. The overhead of the JavaThreads-min approach might be considered the optimal efficiency, due to the balanced workload; in this light, and considering ParaTask's flexibility across a wide variety of applications (shown in the following sections), we believe this added overhead is justified (especially since overhead optimisation has not been done for ParaTask).

Figure 9(b) shows the speedup for an unbalanced workload. Due to this imbalance, JavaThreads-min is now one of the worst performers. Since the granularity of the tasks varies from fine to coarse, both ParaTask and JavaThreads-max improved their performance (compared to the previous benchmark) due to the addition of some coarse-grained tasks. A static decomposition of the tasks however, now clearly results in an unbalance of the workload. It is not until after 11 processors that the benefits of low overhead for JavaThreads-min supersedes its poor work balancing. These results show that the overhead of ParaTask is very small amongst the two computationally intensive benchmarks, while the workload is balanced well to produce consistent speedup between the different workloads.

The image resizing benchmark of figure 9(c) again shows a balanced workload, only this time involving coarse-grained tasks that are disk-intensive. JavaThreads-min and ParaTask both perform equally well, with ParaTask only narrowly performing better with increased processor count. This might be due to a slight offset in the task's timings, due to the high level of disk accessing; in this case, ParaTask's flexibility in the dynamic scheduling supersedes the low overhead advantage of JavaThreads-min. The word permutation benchmark of figure 9(d) is again disk intensive, only this time the workload is not balanced; hence the growing gap between ParaTask and JavaThreads-min. Due to the addition of some finer-grained tasks, JavaThreads-max does not perform as well. In the case of unbalanced workloads, a better Java threads implementation would be to use a dequeue amongst a pool of threads; however, there are already technologies with a similar approach in the following sections.



(a) Fine-grained and balanced workload

(b) Unbalanced workload

(c) Image resizing

(d) Word permutations

Figure 10: Comparing ParaTask performance (using the default scheduling scheme) against modern Java parallelisation libraries with various workloads: (a) fine-grained and balance workload, (b) unbalanced workload, (c) image resizing and (d) word permutations.

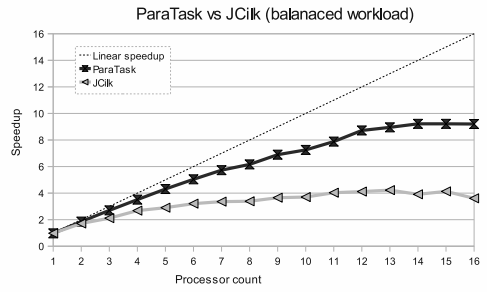
7.5 ParaTask versus modern Java concepts

Using the same four benchmarks, figures 10(a) to 10(d) compare the latest parallelism support added to Java: ExecutorService was introduced in Java 1.5 (here we compare the FixedThreadPool and CachedThreadPool implementations), SwingWorker was introduced in Java 1.6, while ForkJoin was introduced with Java 1.7. The general approach of these libraries is fairly similar to each other: the programmer implements a particular interface, instantiates an instance for each task and submits the task for asynchronous execution. In fact, the underlying implementation of SwingWorker and ForkJoin all use the ExecutorService interface.

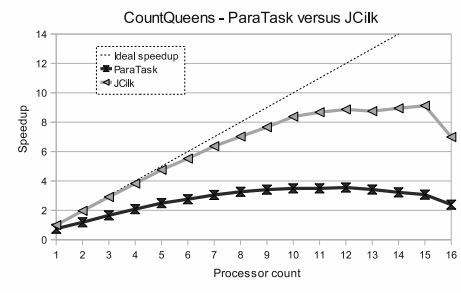
Across the different benchmarks, ForkJoin clearly dominates. This is because ForkJoin does not spawn asynchronously for every task; it sometimes executes tasks serially, therefore avoiding the parallelism overhead. The benefit of this approach is most evident with fine-grained tasks, as in figure 10(a). The ExecutorService with a FixedThreadPool essentially mirrors ParaTask: a very encouraging result. An interesting note is SwingWorker's speedup being limited to 10, this is because Java's SwingWorker implementation has a maximum of 10 worker threads. The ExecutorServce with a CachedThreadPool has high overhead because of the number of threads it creates. However, in the unbalanced workload of figure 10(b), a large number of the threads are re-used since small tasks complete promptly and there is a sufficient number of worker threads being re-used. This is clearly not the case in the balanced workloads of 10(a) and 10(c), since none of the tasks have completed during the enqueuing phase (therefore barely any threads are re-used).

7.6 ParaTask versus JCilk

Figure 11(a) only shows the speedup of ParaTask versus JCilk using the fine-grained and balanced workload, but these results are representative of the remaining benchmarks. The reason JCilk does not perform so well is because of the nature of the workloads: they involve flat parallelism (i.e. all the tasks are spawned at the same time and at the same level, rather than recursively being spawned from within other sub-tasks). JCilk's degraded performance is attributed to the high cost of stealing in the work-first policy: a steal involves a new thread taking over the context (e.g. variables and their values) of the victim thread. For flat parallelism with all tasks being immediately spawned, this essentially means each worker thread steals the spawning context, spawns a task and executes it. In the meantime, the next thread steals at the continuation point where the spawning



(a) Flat parallelism: fine-grained and balanced workload



(b) Recursive workload

Figure 11: Comparing ParaTask performance (using the default scheduling scheme) against JCilk with (a) a flat parallelism workload (b) and a recursive workload. JCilk's work-first stealing policy means it is better suited for recursive workloads rather than flat parallelism. ParaTask, on the other hand, uses a help-first stealing policy.

was interrupted.

While ParaTask performed well in all the previous benchmarks, a critical analysis must also reveal its weaker points; fine-grained recursive benchmarks are where ParaTask is weaker. Interestingly, JCilk, which did not perform too well before, can shine here; the next benchmark, CountQueens, illustrates this. Given an $n \times n$ board, CountQueens finds all the possible solutions to the Queens problem [39]. Figure 11(b) shows the speedup for both JCilk and ParaTask (using the auto schedule), using a CountQueens board size of 15. JCilk outperforms ParaTask for this benchmark, even though both implement the same work-stealing [36] schedule. The difference is JCilk implements a *work-first* policy, while ParaTask implements a *help-first* policy [38]. In the work-first policy, threads leave their current task to sequentially execute a newly created task. Other threads may then steal from where the thread left. This performs well for extremely fine-grained nested parallelism (such as CountQueens) since the thread completes the task before another thread has a chance to steal (where stealing is expensive).

The reason that a work-first policy is not implemented for ParaTask is twofold. First, as shown in all the benchmarks above (except for fine-grained nested parallelism such as CountQueens), a work-first steal is extremely expensive as it requires transferring the context (state of variables) from the enqueuing thread to the thief thread. This expense greatly affects performance when many steals occur (i.e. tasks are not fine-grained enough, therefore other threads have the chance to steal).

Second, a responsive concurrent application requires that tasks are guaranteed to always execute asynchronously [40]. A work-first policy will produce synchronous tasks. In terms of desktop application semantics, a work-first policy violates the structure of multi-threaded GUI applications: this means the enqueuing thread (i.e. the GUI thread) executes the tasks and other worker threads would continue where the GUI thread left off (but only the EDT is allowed to access GUI components). Consequently, only a help-first policy is viable for GUI applications.

These results show that ParaTask executes recursive applications well, but only for higher granularity. The CountQueens benchmark may be considered a worst-case for help-first scheduling policies, but a best-case for work-first policies. Consequently, ParaTask would perform better for higher grained applications.

7.7 The `notify` clause

Figure 12 shows screenshots of the application used to benchmark the notify clause. Naturally, the benchmark must consist of a GUI component, but without user intervention for reproducibility. The selected application is the calculation of the Mandelbrot fractal that gets displayed as it is computed; each column is a pixel wide and represents a single task unit. The color of each column denotes the thread that computed it, in this case four threads were involved in the fractal's calculation. The notify clause is used to request the GUI thread to update the image with the respective colors as soon as it is calculated. As such, it is now compared to other Java parallelisation

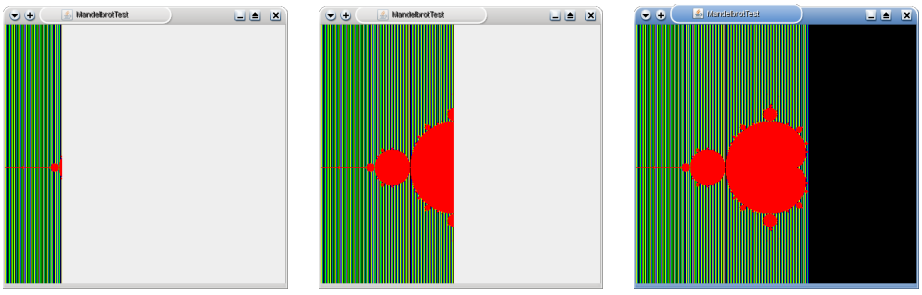
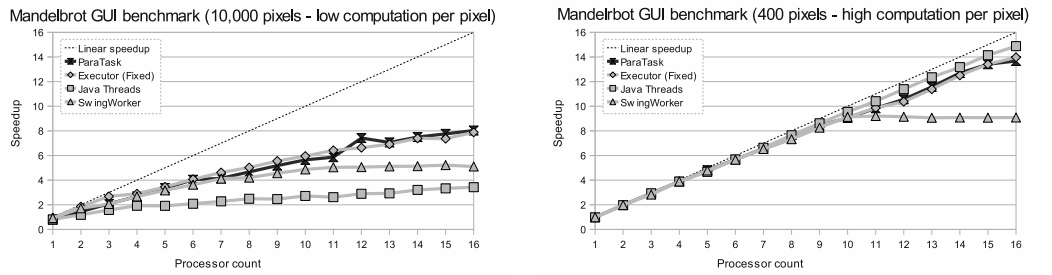


Figure 12: Screenshots of the GUI benchmark application at various stages of its execution. Each column of the Mandelbrot image represents a unit of task, where the column colour represents the thread that executed it. The tasks at the end (in black) contained so little work that a single thread was able to complete them.



(a) Fine-grained and balanced workload

(b) Unbalanced workload

Figure 13: Comparing ParaTask's notify clause to other Java GUI solutions, using (a) a large number of fine-grained tasks and (b) few coarse-grained tasks. The number of pixels represents the image width (and therefore the number of task units).

approaches that support GUI updates.

Figures 13(a) and 13(b) show the speedup of the Mandelbrot calculation using two computational intensities of the fractal. Figure 13(a) denotes a fractal 10,000 pixels wide (therefore 10,000 tasks), however the calculation of each pixel is limited to produce fine-grained tasks. ParaTask and the FixedThreadPoolExecutor (using SwingUtilities.invokeLater() to signal the GUI thread) produced the highest speedup. As expected, using Java threads (maximum) results in poor performance because of the fine granularity.

The same benchmark is repeated in figure 13(b), only this time a fractal of 400 pixels is used, with a much higher amount of computation per pixel. In this case, using Java threads (maximum) now results in the best *speedup* performance. This might be attributed to the increased overhead required for the other thread pool approaches where the overhead of enqueuing the GUI computation is essentially serialised amongst the fixed number of threads in the pool. In the case of Java threads (maximum), there are a lot more threads overlapping this overhead.

Despite the impression that figure 13(b) gives, Java threads (maximum) actually performed worst from a user perceived point of view. Due to the high number of threads created, the GUI thread did not get a chance to execute until all the 400 threads completed (the operating system has to schedule 401 threads compared to 2-17 threads on 1-16 cores). The delay is especially evident since the 400 columns progress (more or less) at the same time, and the GUI update requests occur at the end (when all 400 threads have completed); therefore, the entire fractal appears instantaneously once the entire computation completes. In addition to this lack of interactivity, the system as a whole was noticeably lagging.

7.8 The `dependsOn` clause

In this section, we evaluate the performance of the `dependsOn` clause. The importance of dependences is most evident in the case of a fixed thread pool size where tasks are typically executed in

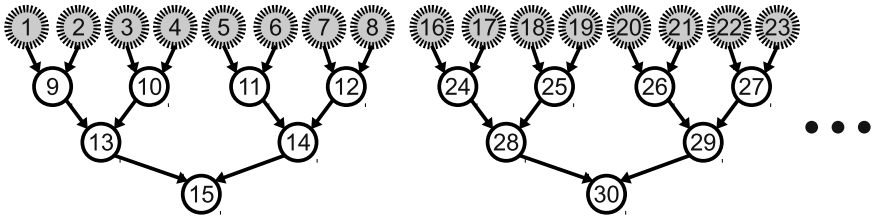
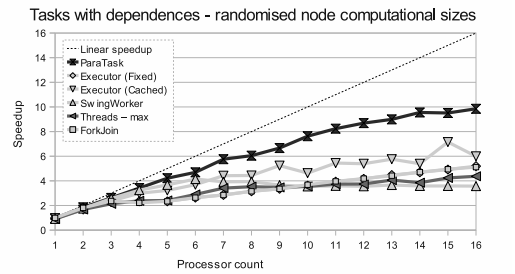
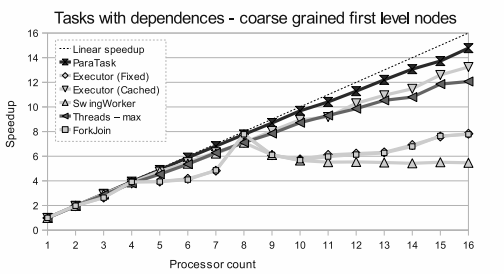


Figure 14: Dependence graph of the tasks, with dependences only within each group of 15 tasks. The top-level tasks involve considerably more computation than the "reduction" tasks. The numbers correspond to the order in which the tasks are created and enqueued.



(a) Random amount of computation on top-level tasks



(b) Coarse-grained top-level tasks

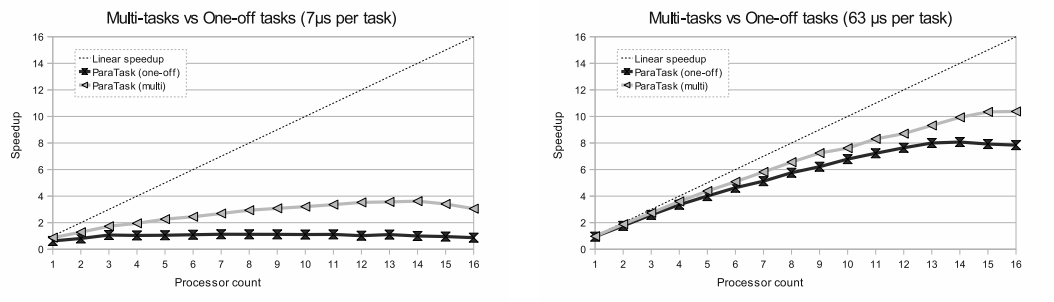
Figure 15: Comparing the performance of ParaTask's `dependsOn` clause to typical approaches where dependences are handled, as described in figure 14. The performance degradation is most notable in fixed-size thread pools since tasks are picked up in the order they were submitted, but might not necessarily be ready for execution.

the order they were created. The natural approach is to spawn the tasks, with the later dependent tasks blocking until the earlier respective tasks complete. In addition to the coupling this introduces between tasks, this also means threads will constantly be idle when they are executing tasks that are in fact waiting for other tasks to complete. An example is shown in figure 14, showing a group of 15 tasks with dependences amongst them; the group of 15 tasks is repeated 100 times. The top level tasks are computationally intensive, whereas the remaining 7 reduction tasks are quick to execute.

Due to the nature of the dependences in this example, it possesses a complication for programmers hoping to achieve good performance. To avoid the performance degradation of blocking threads, programmers will need to structure the creation of tasks and inter-task notification in a rather complicated way; for example, rather than task 10 waiting for tasks 3 and 4 (easy to implement within task 10), task 3 must now notify task 10 to start and the same for task 4 (this needs additional countdown latches and careful reasoning to ensure correctness such that task 10 is not executed twice).

Not only does ParaTask's `dependsOn` clause eliminate the complexity of implementing task dependences, it provides an efficient runtime solution that does not fall victim to the degradation discussed above. Recall figure 5, where it shows that a task with dependences is first placed aside with the **waiting tasks**. When the task's dependences have been met, only then will it get scheduled for execution. Worker threads therefore never pick up waiting tasks that are not ready to execute. Applying this to the example of figure 14, this means worker threads will only grab reduction tasks when the respective top-level tasks have completed. Therefore, worker threads will grab other ready tasks that were enqueued later, since they are ready to execute – hence keeping all threads busy with real computation.

Figure 15(a) shows the results where the granularity per top-level task is randomised between 1-10ms worth of computation (but of course consistent between execution runs), and each reduction involves approximately 1ms worth of computation. We see that the CachedPool implementation of the ExecutorService performs better than the other Java thread pool implementations; even though the CachedPool introduces high amounts of overhead (as shown in earlier benchmarks), it



(a) Extremely fine-grained ($7\mu s$) (b) Fine-grained ($63\mu s$)

Figure 16: Comparing ParaTask's one-off task (using the auto scheduling) to ParaTask's multi-tasks. These iterations are extremely fine-grained, especially in regards to interactive desktop applications. The performance advantage that multi-tasks contribute becomes less distinguishable as the task granularity increases, however the model remains useful for data parallel situations.

nonetheless has the benefit in situations involving a high number of task dependences.

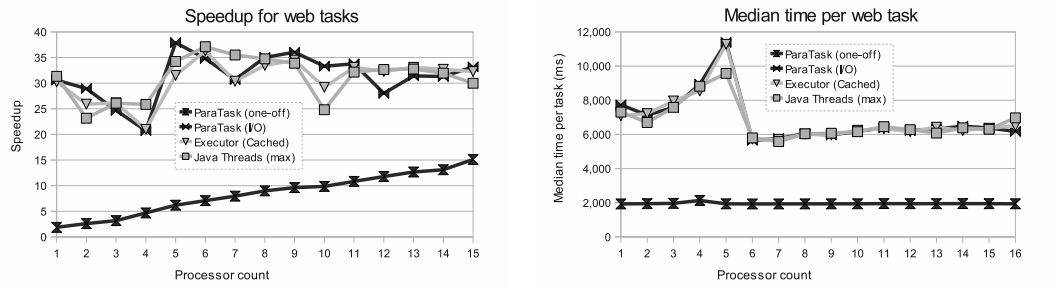
The principle behaviour is made more clear in the extreme benchmark of figure 15(b), where the top-level tasks are considerably more computationally intensive than the reduction tasks. Of first interest is the similarly good performance from CachedPool ExecutorService and Java threads (maximum); as expected however, the CachedPool is slightly more efficient as it manages to re-use some of the threads it creates. Since SwingWorker and ForkJoin are both based on the FixedPool ExecutorService, it also makes sense how they result is near-identical performance; SwingWorker however only scales to 10 processors since it creates a maximum of 10 worker threads, consequently diverging from FixedPool and ForkJoin thereafter.

The dependence property of this benchmark deteriorates the performance for fixed thread-pool implementations that pick up non-ready tasks. The "peak" at 8 processors for the three FixedPool-based implementations is attributed to the benchmark properties. As shown in figure 14, this means the computation of the 8 top-level tasks of each set is always in-sync with the threadpool of size 8 (therefore threads theoretically do not block on dependent tasks). In fact, we even see at this point these implementations have better speedup than CachedPool and Java threads (maximum) due to the low overhead.

7.9 Multi-tasks

While ParaTask achieves task parallelism through the use of one-off tasks, it is also well suited for data parallelism with the use of the multi-task concept introduced in section 3.2.2. Although one-off tasks do not have the overhead of a new thread per invocation, they still have a slight overhead for introducing asynchronisation; in the case of iterating over a data collection, this results in slight asynchronous overhead per iteration. In the case of multi-tasks, this asynchronous overhead is significantly reduced as it only exists once per worker thread (the respective iterations within a sub-task essentially execute sequentially).

Figures 16(a) demonstrates the difference between multi-tasks and one-off tasks for very fine-grained computations, in this case only $7\mu s$ per task (100ms to complete 100,000 iterations in the sequential version). Despite the extremely fine-granularity of this workload, multi-tasks still managed a convincing speedup of almost 4. It is also encouraging to see one-off tasks not performing much worse than the sequential version, considering the fine-granularity of the benchmark. By slightly increasing the granularity to $63\mu s$ per task (6.25s to complete 100,000 iterations in the sequential version), we immediately see a dramatic improvement in the speedup in figure 16(b). As soon as the granularity of tasks are increased further, the performance difference between one-off and multi-tasks becomes indistinguishable; regardless, the whole motivation for multi-tasks is their ease of use for data parallelism.



(a) Speedup for entire benchmark

(b) Median time per task

Figure 17: Comparing ParaTask's performance for I/O tasks versus one-off tasks for web-based computation (Wikipedia queries). Fluctuations in the results are more likely to due to network loads.

7.10 I/O tasks

To understand the benefits of distinguishing I/O tasks from one-off tasks, we present a web-based benchmark that involves retrieving Wikipedia pages. A list of 200 different country URLs represents the tasks to perform. Figure 17(a) shows overall speedup for the entire benchmark (i.e. all 200 tasks), while figure 17(b) shows the median time per task. To help reduce the effects of the network fluctuation, the benchmarks were repeated 20 times and medians were constructed from the 10 most reliable times (after neglecting any extreme outliers). We see an expected gradual speedup improvement for ParaTask one-off tasks, where the median time per task remains constant.

Without the notion of a fixed-sized thread-pool, we see a superlinear speedup (due to the overlapping of the threads), however not at the theoretical speedup we might initially expect. Theoretically, all 200 threads should be overlapped; however, the speedup only seems to hover around 30. After observing the median time per task, it shows the median time per task has at least tripled compared to having a fixed-sized thread-pool. In practice, programmers should not create a thread for each server in order to reduce connections [41]. In this case, the number of connections may be reduced by assigning a larger number of requests per I/O task. Although ParaTask is not optimised for web-connections, the case for I/O tasks is still necessary since it is vital that worker threads do not block [41]. A future implementation of ParaTask might consider limiting the number of I/O threads.

7.11 User perceived performance

In section 7.3, it was shown that the work-stealing tends to outperform work-sharing as the thread count increases (this was the case in most of the other benchmarks too). Therefore, for batch-type applications, the work-stealing would be most useful in reducing wall-clock time. But what about an interactive environment? Due to the interactive nature of desktops, it is generally agreed that the users perception of performance becomes a vital metric in measuring performance [42]. For example, a web-based application addressing multiple users or a desktop application where a single user launches multiple tasks. In such a situation, users expect tasks to be treated fairly: in particular, using a FIFO policy.

The aim of the following benchmark was to quantify how much the actual order deviates from the expected order. In particular, how the deviation is effected as the number of tasks increases. Therefore, the benchmark is interested in the *order* that the tasks are completed in, and not the actual computational time. The tasks used had identical runtimes and involved computing a synthetic load (the Newton-Raphson method). The following table is used as an example to explain the process in calculating the deviation:

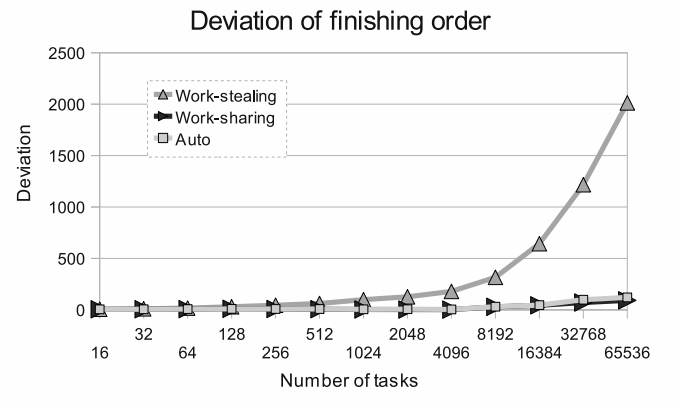


Figure 18: Deviation of actual finishing order compared to the expected finishing order. The higher the deviation, the lower the perceived performance from the user's point of view.

Task number (enqueue order)	1	2	3	4	...	16
Expected finishing order: e	1	2	3	4	...	16
Actual finishing order: a	3	8	2	1	...	11
Difference: $e - a$	-2	-6	1	3	...	5
Difference^2: $(e - a)^2$	4	36	1	9	...	25
Sum: $\sum (e - a)^2$	218					
Deviation: $\sqrt{\frac{1}{N}\sum (e - a)^2}$	3.69					

This example dataset shows how the deviation would be calculated for 16 tasks in a single run of the benchmark (number of tasks, N = 16). The 1st row lists the tasks and the order they were queued in. Naturally, a user would expect these tasks to finish in this very same order (2nd row). Unfortunately the tasks are likely to finish in a different order at runtime, for example the 3rd row. The deviation of the actual finishing order from the expected finishing order is calculated similarly to the standard deviation. The differences (4th row) are squared (5th row) and summed (6th row), and finally the average is computed (last row). This whole process is repeated for a total of 5 times, and the average is plotted as the deviation of finishing order for 16 tasks. This is then repeated for 32, 64, ... and 65536 tasks for each of the scheduling schemes.

The result is shown in figure 18. It shows how the *actual* finishing order deviates from this expected FIFO order as more tasks are introduced (the thread count is fixed at 16). This shows that work-sharing best models the user's expectation of the ordering, consequently resulting in better perceived performance. Of particular interest is the auto scheduling policy, which achieves similar user-perceived performance of the work-sharing. This result, along with those of section 7.3, shows that the ParaTask auto-scheduling allows us to achieve the advantages of work-sharing and work-stealing in the same policy.

8 Related work

An important aspect of developing new parallelisation tools is to focus on the user requirements, in this case the desktop application developer. Most related work have the objective of *combining parallelism with object-oriented applications*. Even though most desktop applications are in fact object-oriented, this objective is insufficient for successfully parallelising desktop applications. Rather, we claim that the focus should be on *combining parallelism with (graphical) object-oriented desktop applications*. The keyword here, *desktop*, is vital: one must understand and respect the structure of graphical desktop applications (as discussed in section 2.3) before attempting to introduce parallelism [43]. It is this particular concentration that we believe distinguishes ParaTask from previous work. ParaTask's objective is not only to use concurrency to improve performance, but also to create responsive GUI applications that do not freeze. Moreover, ParaTask remains

generalised enough to be applicable to console applications without a GUI.

There has been numerous research on combining concurrency with object-oriented programming: Briot et al [44] has classified some previous work into various categories, while Philippsen [14] surveyed over 100 concurrent object-oriented languages. In both surveys, the most relevant work includes Concurrent Object Oriented Language (COOL) [45], Compositional C++ (CC++) or some form of the *active object* pattern [46, 47]. More recent work includes Cilk [48], Cilk++ [49], ThreadWeaver [50], QtConcurrent [51], Intel Threading Building Blocks (TBB) [52], the new OpenMP tasking feature [53], Java 1.6's SwingWorker [54] and Foxtrot [55]. Other languages currently under development include X10 [56] and the Task Parallel Library (TPL) [57] in the .NET Framework.

The first primary difference is that ParaTask, as presented here, uniquely integrates different task types into one concept. Second, none of the related work provide support for automatically handling task dependences (except for ThreadWeaver). Third, ParaTask has the primary focus of providing parallelism to desktop applications without code restructuring (but still requires some code reworking of course): this implies adherence to the threading model of graphical applications (section 2.3) and conforming to object-oriented concepts. In particular, ParaTask guarantees that tasks are always executed asynchronously with the enqueuing thread (unlike, for example, OpenMP, Cilk++, TBB and TPL). This is a fundamental requirement for responsive concurrent applications [40, 58]. Further specific differences are discussed below.

An active object (supported in languages such as Scala [59] and Akka [60]) is essentially a proxy that separates method invocation from method execution. Although this pattern works well for many distributed computing applications, it is not well suited for shared memory models such as programming desktop applications. Furthermore, most active object implementations do not support intra-object concurrency: only one method executes at a time while others are delayed. Even though this makes the program correctness easier to justify, it greatly reduces parallel performance [14]. ThreadWeaver and Intel TBB have a task concept where programmers enclose independent code snippets within a separate object. But just like active objects and thread libraries, these tools require considerable code restructuring.

Some languages, such as X10 (which is designed for non-uniform cluster computing rather than the shared memory systems of current desktops) and CC++, create concurrency using a special keyword in front of the method invocation. This brings convenience to the programmer since code does not need to be restructured into a separate object. However, here lies a fundamental disadvantage typical of concurrent languages that coordinate concurrency by the *caller* (as opposed to the *callee*): *encapsulation*, an essential object-oriented programming concept, is immediately broken [14]. First, implementation details of an invoked method must be understood by the caller (in order to determine if the invoked method is thread-safe). Second, all caller code must be carefully analysed every time the callee code is modified.

OpenMP's `task` construct also breaks encapsulation, where programmers wrap code with compiler directives. The `parallel` pragma itself includes an implicit barrier at the end, therefore all tasks must complete before exiting this construct; it is therefore unacceptable to place the `parallel` pragma within an event-handler. Synchronisation is achieved by using an optional and explicit `taskwait` pragma (for nested tasks), causing the *current task* to block until *all* its *children tasks* finish. In order to achieve finer-grained synchronisation (i.e. subset of the tasks), programmers must wrap the task subset within a *dummy* task and wait on that subset. All of these show that the underlying threading model of OpenMP is not well suited for developing interactive GUI applications.

Rather than using a keyword, other concurrent languages coordinate concurrency by the use of a special enqueuing method. The advantage here is that concurrency is introduced within libraries, therefore eliminating the need for an additional intermediate compiler. Examples of these languages include QtConcurrent and TPL. Below is a QtConcurrent example:

```
QFuture<String> future = QtConcurrent::run(QtConcurrent::bind(myMethod, "Hello,
World"));
```

Unfortunately, these languages still possess the disadvantage of breaking encapsulation since concurrency is coordinated by the caller. Furthermore the resulting code is less legible, especially when the programmer must bind the arguments to the method.

COOL takes a different approach: concurrency is coordinated at the callee side by prefixing the `parallel` keyword to method declarations (just like ParaTask). Although COOL addresses the broken encapsulation issue discussed above, it still possesses some setbacks that are also common to CC++. First, return results are discarded. Second, these languages spawn user-level threads for each task, resulting in poor performance when a large number of smaller tasks are created (especially in recursive divide and conquer applications) [8].

Cilk allows programmers to annotate method declarations with a special keyword to denote parallelisable methods. The advantage is that the serial version of the program is produced when the keywords are removed from the parallel code. Unfortunately, Cilk does not focus on event-based desktop applications; one cannot parallelise a GUI application using a work-first work-stealing model. The `TASK` keyword in ParaTask serves as a form of documentation to ensure encapsulation is maintained for tasks (only task implementers need to know the task details). This is not so for Cilk, since methods that call other tasks must themselves be annotated with the same keyword[6]. The same semantics have also been applied to JCilk [61]: a Java implementation that also takes into consideration exception handling. Nonetheless, ParaTask's work-stealing implementation is motivated by Cilk's well-proven work-stealing scheduling policy [36] (except that ParaTask uses help-first policy, not a work-first policy as discussed in section 7.6).

The Habanero-Java language [62] is also a task-based language that builds on X10. It uses the `async` keyword in front of statements to execute them asynchronously with their parent, while the `finish` keyword joins spawned (sub)tasks.

The languages mentioned so far do not make any special consideration for parallel GUI applications, which is vital for desktop applications. Java 1.6 introduced the SwingWorker class to assist programmers in developing responsive GUI applications. Every invocation of a task is enclosed in a SwingWorker object where programmers implement a `doInBackground()` method to be executed on a worker thread. When the worker thread completes, the EDT will execute the `done()` method. SwingWorker is only designed to be used once, therefore the programmer must create a new instance for each task invocation. This model is only applicable to the EDT (hence the name), whereas ParaTask is also generalised to be useful for non-GUI parallel applications.

Although SwingWorker is a simpler alternative to manually using threads, programmers unfortunately end up breaking encapsulation since they must know whether the code inside `doInBackground()` is thread-safe. ParaTask, on the other side, encourages encapsulation (and therefore code reuse) since the tasks are naturally documented as thread-safe. SwingWorker requires explicit calls (e.g. to start the worker) and provides no support for specifying dependences amongst tasks.

Foxtrot provides two solutions to avoid freezing the GUI. The first approach is the asynchronous solution, which is essentially identical to the SwingWorker discussed. In the second approach, the synchronous solution, code executes "sequentially" as it appears. In order not to block the EDT when it invokes a task, the EDT is re-routed to continue processing events. When the task completes, the EDT is again re-routed to resume executing where it left off. The benefit of such a model is that callback methods are not needed. Unfortunately this model is less intuitive to use, especially when multiple tasks are invoked. Foxtrot, like SwingWorker, only applies to the EDT.

The underlying implementation of Java's ExecutorService is similar to ParaTask, in that both tools provide programmers with the option of assigning tasks to fixed thread pools (for computational tasks) or new threads (for I/O tasks). However, ParaTask's contribution is the ease of use, automatic handling of dependences and the programmers relief of taking many technical decisions (for example nested parallelism). Java 7 released the `ForkJoinTask`, suitable specifically for fork-join scenarios. ParaTask, however, provides a more generic model to solve a wider range of problems. For example, I/O computations are discouraged from this fork-join implementation. The java.util.concurrent package does not provide some of the data parallelism features supported by ParaTask, such as barriers and reductions for multi-tasks.

In 2009, Apple released GCD which supports task parallelism. The major difference, compared to ParaTask and all the above related work, is that GCD is integrated into the operating system rather than at the application-level. This has the advantage that the operating system manages the thread pool and task execution across a number of applications. This is similar to some of

[6]Cilk Art's Cilk++ removes this requirement, since all functions are compiled by default to use the Cilk calling convention.

the above approaches, where tasks are queued to various `dispatch` queues. Unfortunately such a syntax breaks encapsulation and requires developers specify which queue to send tasks.

More recent work includes the development of parallel programming features in .NET 4.5, namely TPL. Like ParaTask, TPL is also task-based; a task in TPL is equivalent to a one-off task in ParaTask. However, ParaTask further identifies multi-tasks and I/O tasks using a consistent syntax to one-off tasks (without having to resort to parallel loops and threads). The ParaTask scheduling system also allows for recursive tasks to be used in this unified task model. Dependencies in TPL are supported by the `Task.ContinueWith()` method, but there is no support for reductions and non-blocking task notification.

9 Conclusions

With the arrival of multi-cores for mainstream desktop systems, our daily desktop applications must be parallelised if we wish to benefit from these systems. These event-based desktop applications possess a different structure from typical batch-like applications that programmers must adhere to. ParaTask is an object-oriented solution for the parallelisation of a wide range of applications. The model supports a range of task types, unified into the same model, and ensures adherence to important object-oriented principles. Additional features allow for intuitive dependence handling, exception handling and non-blocking notification. These features keep the tasks decoupled from each other and allow programmers to easily achieve common parallelisation idioms. With flexible scheduling policies, the benchmarks reveal ParaTask's ability to perform well for a wide range of applications. The benchmarks show great performance compared to other Java parallelisation tools.

References

[1] M. Creeger, "Multicore CPUs for the masses," *Queue*, vol. 3, no. 7, pp. 63–64, 2005.

[2] 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*, (New York, NY, USA), pp. 97–105, ACM, 2001.

[3] TIOBE Software BV, "TIOBE programming community index." `http://www.tiobe.com/tiobe_index`, 2016.

[4] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[5] L. A. Barroso, "The price of performance," *Queue*, vol. 3, no. 7, pp. 48–53, 2005.

[6] D. Lea, *Concurrent programming in Java: design principles and patterns*. Addison-Wesley, 2 ed., 1999.

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

[8] P. Hyde, *Java Thread Programming*. Sams, 2001.

[9] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*. Addison-Wesley Professional, 2006.

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

[11] M. Buddhikot and S. Goil, "Throughput computing with chip multithreading and clusters," *Lecture Notes in Computer Science*, vol. 3769, pp. 128–136, 2005.

[12] S. Oaks and H. Wong, *Java threads*. O'Reilly Media, Inc, 3 ed., 2004.

[13] B. Harbulot and J. R. Gurd, "Using AspectJ to separate concerns in parallel scientific Java code," in *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, (New York, NY, USA), pp. 122–131, ACM Press, 2004.

[14] M. Philippsen, "A survey of concurrent object-oriented languages," *Concurrency: practice and experience*, vol. 12, no. 10, pp. 917–980, 2000.

[15] 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.

[16] Microsoft, *Parallel Extensions to the .NET Framework Community Technology Preview (CTP)*, June 2008.

[17] Oracle, "java.util.concurrent ExecutorService." `http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ExecutorService.html`, 2010.

[18] R. D. Blumofe, *Executing Multithreaded Programs Efficiently*. PhD thesis, Massachusetts Institute of Technology, 1995.

[19] J. Robert H. Halstead, "Multilisp: a language for concurrent symbolic computation," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 4, pp. 501–538, 1985.

[20] H. Sutter, "Interrupt politely," *Dr. Dobb's Journal*, April 2008.

[21] A. L. Fisher and A. M. Ghuloum, "Parallelizing complex scans and reductions," in *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, (New York, NY, USA), pp. 135–146, ACM, 1994.

[22] N. Giacaman and O. Sinnen, "Parallel Iterator for parallelising object oriented applications," in *The 7th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS '08)*, (Cambridge, UK), 2008.

[23] R. H. Carver and K.-C. Tai, *Modern multithreading*. John Wiley & Sons, 2006.

[24] Nokia Corporation, "Signals & slots." `http://qt-project.org/doc/qt-4.8/signalsandslots.html`, 2011.

[25] D. Gregor, "Boost signals." `http://www.boost.org/doc/libs/1_49_0/doc/html/signals.html`, 2004.

[26] M. Weisfeld, *The Object-Oriented Thought Process*. Addison-Wesley Professional, 3 ed., 2008.

[27] S. Matsuoka and A. Yonezawa, "Analysis of inheritance anomaly in object-oriented concurrent programming languages," *Research directions in concurrent object-oriented programming*, pp. 107–150, 1993.

[28] G. Milicia and V. Sassone, "The inheritance anomaly: ten years after," in *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, (New York, NY, USA), pp. 1267–1274, ACM, 2004.

[29] J. B. Goodenough, "Exception handling: issues and a proposed notation," *Communications of the ACM*, vol. 18, no. 12, pp. 683–696, 1975.

[30] S. Zakhour, S. Hommel, J. Royal, I. Rabinovitch, T. Risser, and M. Hoeber, *The Java Tutorial: A Short Course on the Basics*. Prentice Hall, 4 ed., 2006.

[31] Sun Microsystems, "JavaCC Home." `https://javacc.dev.java.net/`, 2009.

[32] O. Sinnen, *Task Scheduling for Parallel Systems*. Wiley, 2007.

[33] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, (New York, NY, USA), pp. 1–12, ACM, 2000.

[34] W. Lu and D. Gannon, "Parallel XML processing by work stealing," in *SOCP '07: Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*, (New York, NY, USA), pp. 31–38, ACM, 2007.

[35] W. F. Burton and R. M. Sleep, "Executing functional programs on a virtual tree of processors," in *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, (New York, NY, USA), pp. 187–194, ACM, 1981.

[36] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[37] R. D. Blumofe and D. Papadopoulos, "The performance of work stealing in multiprogrammed environments (extended abstract)," *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 1, pp. 266–267, 1998.

[38] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2009)*, 2009.

[39] P. J. Campbell, "Gauss and the eight queens problem: A study in miniature of the propagation of historical error," *Historia Mathematica*, vol. 4, no. 4, pp. 397 – 404, 1977.

[40] H. Sutter, "The pillars of concurrency," *Dr. Dobb's Journal*, vol. 32, August 2007.

[41] Microsoft, "MSDN library, improving ASP.NET performance." `http://msdn.microsoft.com/en-us/library/ff647787.aspx`, 2012.

[42] S. C. Seow, *Designing and Engineering Time: The Psychology of Time Perception in Software*. Addison-Wesley, 2008.

[43] Sun Microsystems, Inc, "Concurrency in Swing." The Java Tutorials, 2008. `http://java.sun.com/docs/books/tutorial/uiswing/concurrency/index.html`.

[44] J.-P. Briot, R. Guerraoui, and K.-P. Lohr, "Concurrency and distribution in object-oriented programming," *ACM Comput. Surv.*, vol. 30, no. 3, pp. 291–329, 1998.

[45] R. Chandra, A. Gupta, and J. L. Hennessy, "COOL: An object-based language for parallel programming," *Computer*, vol. 27, no. 8, pp. 13–26, 1994.

[46] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artificial Intelligence*, vol. 8, pp. 323–364, 1977.

[47] G. Agha, *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press, 1986.

[48] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," in *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 207–216, ACM, 1995.

[49] Cilk Arts, Inc., *Cilk++ Programmer's Guide*, 2009.

[50] M. Boehm, "KDE 4.0 API Reference - Introduction to ThreadWeaver." `http://api.kde.org/4.0-api/kdelibs-apidocs/threadweaver/html/index.html`, 2012.

[51] Nokia Corporation, "QtConcurrent." `http://doc.trolltech.com/4.4/qtconcurrent.html`, 2008.

[52] Intel Corporation, *Reference for Intel Threading Building Blocks*, 2010.

[53] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 3.0*, 2008.

[54] Sun Microsystems Inc, *Java Platform Standard Edition 6 API Specification*, December 2006.

[55] S. Bordet, "Foxtrot - easy API for JFC/Swing." `http://foxtrot.sourceforge.net/`, 2012.

[56] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, (New York, NY, USA), pp. 519–538, ACM Press, 2005.

[57] Microsoft, "Task Parallel Library (TPL)." `http://msdn.microsoft.com/en-us/library/dd460717.aspx`, 2012.

[58] H. Sutter, "Use threads correctly = isolation + asynchronous messages," *Dr. Dobb's Journal*, March 2009.

[59] École Polytechnique Fédérale de Lausanne, "The Scala programming language." `http://www.scala-lang.org/`, 2012.

[60] Typesafe Inc., "Akka." `http://akka.io`, 2012.

[61] J. S. Danaher, I.-T. A. Lee, and C. E. Leiserson, "Programming with exceptions in JCilk," *Science of Computer Programming*, vol. 63, no. 2, pp. 147–171, 2006.

[62] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: the new adventures of old X10," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pp. 51–61, 2011.