NTNU
Norwegian University of
Science and Technology

WoolPlot: A Visual Wool Profiler

Peter Hemmen

Master of Science in Computer Science
Submission date: June 2011
Supervisor: Ian Bratt, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Following the program execution of a task-based program on a modern SMP is difficult. No guarantees are given as to how the scheduler will schedule the tasks, or even if the application will be executed in parallel at all. To help a programmer understand the flow of an application and get ideas for which areas are worth improving, a parallel profiler may prove very useful.

The focus of this thesis is to implement some new profiling capabilities for Wool, a C-library being developed at SICS. This is a young, open-source library which is built to support very fine-grained parallelism. The student will have to decide on what type of profiling to create, such as whether the data-gathering will be statistical or instrumentation-based.

The task will include creating a graphical user interface which will help both an implementer of Wool and a regular user understand what is happening in the execution of a Wool program. Some natural elements to display in the GUI are steals, spawns and critical path, however, how the GUI will look and what it should include will be established at a later stage in the project. The task should include reporting on how the profiler performs on several different benchmarks, as well as measuring the overhead incurred by the profiler.

Assignment given: 17. January 2011
Supervisor: Ian Bratt

Abstract

Task-based programming involves creating tasks, which can be run independently of each other, and letting the run-time system schedule the tasks on the underlying architecture. Wool is a new library for task-based programming created at SICS in Sweden. To assist a developer who is using Wool to parallelize a program, as well as the scientists who are actually developing Wool, a profiler which shows what happened in a computation can be very helpful.

In this project we modify the Wool library to print more data about its computations. When the output is given to a Java application also developed in this project, the Java application produces a graphical representation of the execution. Each worker thread is visualized separately, with spawns, steals, leaps, critical path and CPU usage information included at a position corresponding to when the events actually occurred.

The profiler, which we have named WoolPlot, is put to the test using a few real-world benchmarks, as well as some created especially for this project. The benchmarks show that WoolPlot works well when describing the distinct events such as steals and spawns. The reporting on the CPU load is too inaccurate to be sufficient for all practical uses. The overhead of the profiler is estimated to be between 3% and 6%.

Contents

List of Figures

Chapter 1

Introduction

Task-based programming is indeterministic in nature, because the actual scheduling of the work is left to the run-time system. It is often difficult for a programmer to know just what actually happened when an application was executed. Parallel programs might not be as parallel as one had expected, or they might not scale appropriately when run on many processors. By showing what actually happened in a run, software profilers will often be able to help a developer with discovering and resolving such issues.

Wool is an open-source C-library for task-based programming created at SICS especially to support very fine-grained parallelism. A parallel Wool run will always include spawns and steals. The threads involved in the run will steal spawned tasks and then eventually spawn their own tasks from the stolen tasks. These events realize the actual parallelism in the application.

Recording these events and visualizing them has been the main goal of this project. This report will provide the background information needed to understand the challenges met in this project, as well as describe the implementation and testing of the profiler, which has been named WoolPlot.

Outline

Chapter 2 provides a background look at parallelism in general, with a special focus on task-based programming. To prepare for the implementation chapter, some quite specific background info about the Wool library is also included.

In Chapter 3, a brief, general background on software profiling is given. In addition, it includes example output and screenshots from many of the profilers which have inspired us when designing WoolPlot.

The implementation is described in detail in Chapter 4. This chapter will explain how the Wool source code was modified, how the Java application was created, and how both parts tie together to create a profiler.

To see how WoolPlot performed, several programs are briefly explained and profiled in Chapter 5. These include both real-world benchmarks, as well as a few proof-of-concept applications made especially for this project.

Finally, Chapter 6 summarizes the project and outlines ideas for further improvements of WoolPlot.

Chapter 2

Parallelism and Wool

2.1 Parallel computations

Parallel programming has in later years stepped out of large-scale clusters and into the mainstream consumer market. As the traditional, single-cored processor hit the power wall, CPU manufacturers have instead started placing more, albeit less powerful cores on the chips [1]. To maximize execution speed, a programmer has to utilize as many of these cores as possible. This type of chip-level multiprocessing is what this report will deal with.

2.1.1 Parallel Programming Models

Message passing and shared-memory programming are the two main areas of parallel programming. The former is typically reserved for large clusters of powerful nodes performing large-scale calculations. Multi-parameter scientific simulations such as weather forecasting and analyzing geological models in search for oil are some characteristic uses. Shared-memory programming is typically more attractive to programmers because of the easier handling of data [2]. For the everyday computer user, it is also a much more important area of research. More and more home computers have two or more processing cores, and the very latest development has even seen multi-cores enter the mobile phone-market, with the first ever mainstream dual-core mobile phone being the LG Optimus 2X [3]. Clearly, creating easy ways for a programmer to parallelize code is a challenge with large implications.

Shared memory

Unsurprisingly, shared memory programming refers to computations where several processing cores share some level of the memory hierarchy. This does not necessarily mean, however, that there is few processors involved. The biggest Tile-Gx processor from Tilera has no less than 100 symmetric cores [4]. When many processors have access to the same data, great care has to be taken to avoid such problems as data races and deadlocks. In addition, depending on what programming model is used, the different threads might have to be managed explicitly by the programmer. On the other hand, because processors always have access to a level of the cache or the main memory, the user does not need to worry about explicit communication. Among the most widely used shared memory programming models are POSIX threads and OpenMP.

Message passing

Some calculations are so large and time-consuming that they have to be performed by many computers put together in a cluster. In order to accomplish this, a programmer has to explicitly specify what data is to be sent around, and also make sure that the data is sent back and aggregated correctly. The different processors communicate by sending messages, and programmers have to take care to avoid deadlocks. The most widespread model for message passing programming is MPI (Message Passing Interface).

2.2 Task Based Programming

This project will involve working with Wool, which is a library for task-based programming. Task-based programming aims to simplify shared-memory programming by removing some chores from the programmer. The developer has to specify what parts of the code are independent. The scheduler, or run-time system, will then take care of sharing the work between all available processors. What this means, is that a programmer will not have to tailor the code to fit any specific architecture. He or she should be confident that the work will be well balanced on whatever system the code eventually is run on.

Some advantages of task based programming are summarized in the following list:

Fine-grained Thread switching is slower than task switching. By keeping a thread running and just giving it more work to do, task-based programming minimizes switching in and out entire threads. Due to this, programs can be more fine-grained. In the TBB tutorial [5], Intel states that starting and terminating a task is about 18 times faster than doing the same operations for a thread. Those figures apply to Linux systems. Under Windows, the number overshoots 100.

Portable and scalable The programmer does not have to adapt a program to the underlying architecture. The task scheduler takes care of that automatically, and programs should scale to exploit all available hardware.

Efficient load balancing Rather than dividing the problem equally among all processors explicitly, a programmer should be able to focus on solving the other programmatic challenges. The scheduler takes responsibility for sharing the work. Furthermore, traditional threaded programming models typically use the operating system's fair scheduler to allow every thread some running time. The task-based scheduler can act independently of these restrictions, and may therefore be able to schedule the calculations in a more efficient manner.

Task abstraction Task-based programming aims to free the programmer of such troubles as mapping the computation to the hardware or consider data races and deadlocks. Thinking in tasks, and specifying them in code, should be an easier approach to parallel programming.

Fitting for asymmetric multiprocessors In an interesting paper, Hill and Marty argue that asymmetric multi-cores might be a viable way to achieve more speedup from multiprocessors [6]. This is due to the fact that there often will be some large un-parallelizable part of a program which should be run on the fastest CPU core possible to limit how much it dominates the computation. Given a scheduler which is aware of the architecture [7, 8], task-based programming may very well be a fitting way to utilize such a system.

2.2.1 Work-stealing

It appears that the most common scheduling technique for task-based programming is work-stealing. It was proposed as early as in 1981 [9], but Blumofe and Leiserson gave the "first provably good work-stealing scheduler for multi-threaded computations with dependencies" [10] in their 1999 paper. The authors also participated in creating *Cilk*, which is a task-based programming language based on C. Unsurprisingly, it employs work-stealing for scheduling the tasks. The following is a brief introduction to how their algorithm works.

In their paper, Leiserson and Blumofe use processors and threads where we would use threads and tasks. For this section, we will use threads and tasks to make it fit with the rest of the thesis. A precondition for the description is that the scheduler makes sure that each processor has one thread running on top of it, ready to work on tasks.

Each thread keeps its tasks in a double-ended queue, often called *deque*. A task can only be added to the bottom of a deque, but it can be removed from either end. Adding and removing tasks from the deques are governed by the four rules which dictates how a thread should behave when a task spawns a child, stalls, dies or re-enables a stalled task.

If a task spawns a child, the work in that task has to be completed before the spawning task can continue. The spawning task is thus put back on the bottom of the deque, and the thread begins work on the newly spawned task. When a task stalls due to waiting on a dependency, or dies because it is finished, the thread needs more work. It will first look to its own deque and remove a task from the bottom if there are any present. If the deque is empty, the thread will try to steal a task from the top of a randomly chosen thread's deque. It will continue to do so until it is successful. The last of the four rules concerns whenever a task enables a stalled task. The recently enabled task should then be added to the bottom of the deque of the thread which enabled it.

It is interesting to note that all the parallelism comes from the stealing. The threads themselves are responsible for fetching more work, and as long as there are enough parallelism available in the form of tasks, the threads will be kept busy. Another appealing quality is the inherent cache-friendliness of the deques. A thread which spawns a task will always try and execute it right away, while the cache is hot. The tasks at the top of the deques are always the oldest tasks in the deque. This means that if a task is stolen, it will most likely have the least cache-wise

impact on the thread it was stolen from, because the thread might have spawned and executed many tasks since it was first created.

Figure 2.1 shows a schematic drawing of a task runtime taken from [11]. The figure illustrates the point of having dedicated worker threads is to avoid thread switching, and instead rely on the much faster task switching.

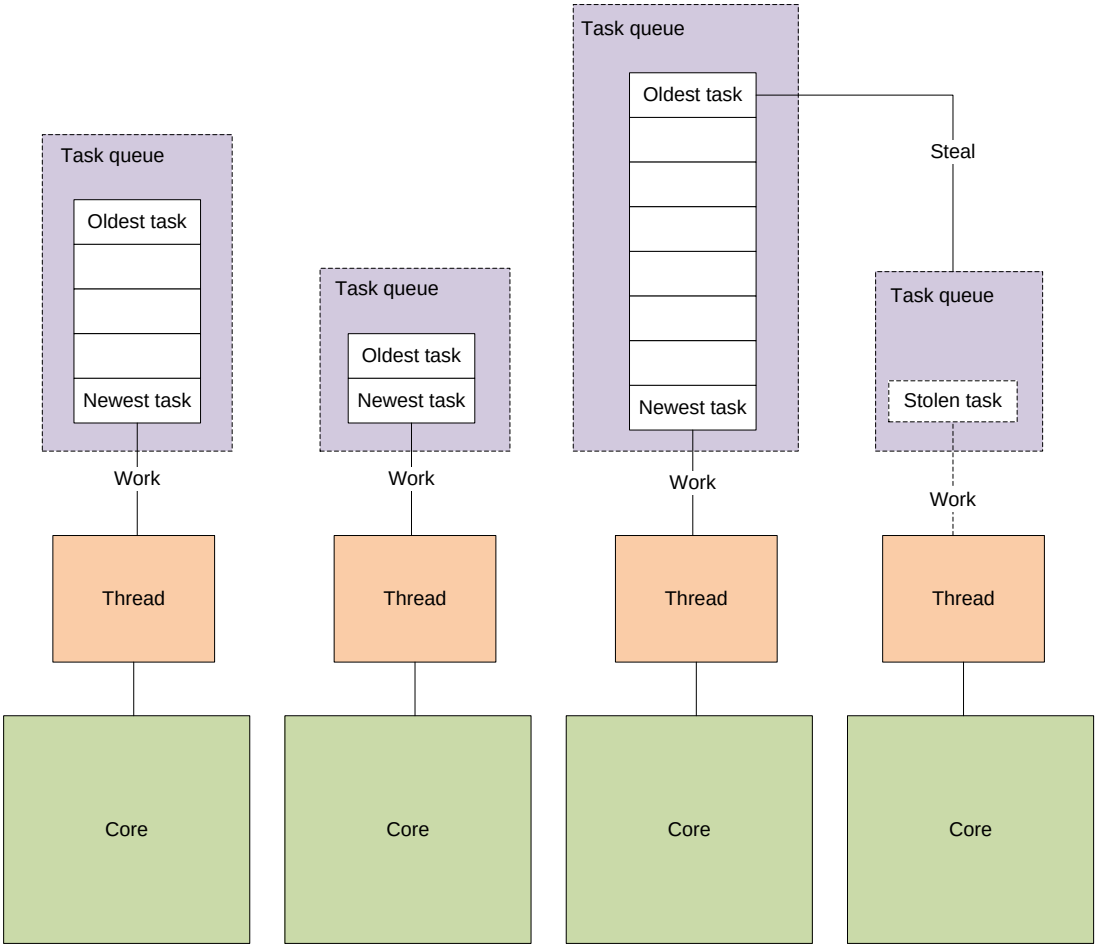

Figure 2.1: Schematic drawing of a task run-time

2.2.2 Task-based programming style

A common pattern in task-based programming is letting tasks themselves spawn tasks. For example, Intel calls this “Recursive Chain Reaction” in

their tutorial for TBB [5, p. 71]. If a programmer wants to iterate over an array using tasks, there are two main ways of doing this. Either, the "root" task can spawn all the other tasks, which each gets a piece of the array, or the root task can spawn two tasks, which each gets half the array. Those will in turn do the same with their part, and the parts will eventually be small enough to just iterate through in the leaf nodes of the task tree which has now been created.

On the surface, these two approaches look quite similar, they will perform the same job, but the first will create fewer tasks, and should even be easier to program. Because of the way a work-stealing scheduler works, however, the second approach is far superior. Say a task can be created in one timestep, and n tasks should be created, the first approach will use n timesteps to create the tasks. If we assume a perfect parallel execution, the recursive way will be able to spawn n tasks using only $\mathcal{O}(\lg n)$ timesteps[1].

An even more important issue, is that of balancing the load. When the root task spawns a task in the first timestep, no other threads will have any work to do, and one of them will steal the newly spawned task. Using the recursive scheme, the two active tasks will now each spawn a task, resulting in two new tasks which can be stolen. In the linear scheme, however, the first stolen task will not spawn another task, so there will only be one new task to steal in the next timestep. Considering that there has to be locking involved whenever a thread tries to steal work, to avoid two threads stealing the same task, the overhead of this approach cannot be ignored. Furthermore, in many work-stealing schedulers, threads in need of work will try to steal from random threads. This will undoubtedly cause many unnecessary steal attempts, seeing as there is only one thread actually spawning new tasks.

2.2.3 Other task based programming models

There are several publicly available task based programming models. Intel is using a lot of resources on parallel computing, and they offer both Intel Cilk Plus [12] and Threading Building Blocks. Some other well known include OpenMP, in which tasks were supported from version 3.0 [13], SMPSs [14] and Grand Central Dispatch (GCD) from Apple.

For many of the task based programming models designed for SMPs,

[1] Unless otherwise stated, $\lg n$ is a short form of $\log_2 n$.

the work-stealing scheduler employed in Cilk seems to be an inspiration. To name a few, both TBB, SMPSs and Wool uses very similar scheduling as Cilk.

2.2.4 Wool

Wool is a relatively young library being developed mainly by Karl-Filip Faxén at The Swedish Institute of Computer Science (SICS). With the current version being 0.1.2alpha, it is mostly a research tool, and by no means a finished product. According to Faxén, the objective of the library is "to provide a reasonably convenient programming interface (in particular not by forcing the programmer to write in continuation passing style) in ordinary C while still having a very low task creation overhead." [15]. This is accomplished by using macros extensively and utilizing `pthreads` for the actual parallelism. In addition, inline functions and a few lines of inline assembly are used [16].

Performance

According the user guide, "[Wool's] performance is competitive with that of Cilk and the Intel TBB, at least in terms of overhead." [17]. As this implies, the overall performance of the library has not been the main focus of the developers, which an early paper describing Wool also shows [15]. Nonetheless, a more recent comparison between OpenMP 3.0, Cilk++ and Wool, shows Wool to actually perform quite well [16].

Use

To create a task in Wool, one will have to define it using one of the pre-made macros. By default, Wool creates macros for tasks with and without a return value and a maximum arity of 10. A task is invoked with the keyword `spawn`, and the return value is collected using the keyword `sync`. A call to `sync` will block if the task has not yet returned, thus acting as a barrier to make sure execution does not go forward with undefined variables. There is also a `call` keyword, which is just a shorthand form of both spawn and sync, which makes the direct invocation of a task more efficient and cleaner. A more thorough explanation and some pseudo code examples can be seen in Section 2.2.5 on page 12.

As with most task-based programming models, creating a task is equivalent to marking the piece of code as something which **can** be run in parallel. If there are worker threads which have no work, the task can be stolen, but if there are no free worker threads, or if the system simply only has one processor core, the task will eventually be executed by the thread which spawned it. The task is then said to be in-lined. In practice, most tasks will be in-lined, and this should happen very quickly, so that there is no reason to use a normal function call rather than spawning a task. Ideally, there should also be very low overhead for tasks which are stolen as well. Wool has indeed a very low spawning and syncing overhead (orders of magnitude better than TBB, Cilk++ and OpenMP [18]). This allows a programmer to create as many tasks as possible, and be confident that the execution will be fast no matter the actual size of the system it will run on.

Like OpenMP and Intel TBB, Wool also provides a way to quickly parallelize a for-loop with independent instructions, through a separate `for` construct. The Wool parallel for-loops requires the programmer to specify a `loop_body`, which is what is done in one iteration of the loop. The parallel for-loops in Wool have not been used in this project. Because the for-loops are built using the other macros, however, programs using them can be profiled in the same manner as programs built using regular syncing and spawning directly.

Implementation

The driving force behind Wool, Karl-Filip Faxén, has written a paper in which he explains some of the finer points of Wool's implementation [15]. For this discussion, it is important to remember that there usually is exactly one worker thread per CPU core, so a worker is virtually analogous to a CPU core.

An interesting implementation issue is what a worker should do when it tries to sync a task, and discovers it has been stolen. If it steals some work from another random worker, it will have something to do, but it might create another issue. If a worker (X), which discovers that the task it tries to sync has been stolen by worker (Y), steals work from another task (Z), X might be busy when Y completes the task it stole. In this scenario, the code after the task X initially tried to sync will be ready to execute, but because X is busy executing other tasks, it can not be executed until X returns. Also, because it cannot be spawned by any worker other than

X, it can not be stolen. If there is an abundance of exposed parallelism in the computation, so that each worker is busy anyway, this might not matter. Generally, however, it is not desirable to have code which is ready to execute, without a worker able to execute it.

One of the ways to avoid this situation, originally proposed by Wagner and Calder [19], is leapfrogging. When the worker X, which tries to sync, finds that the task has been stolen, it is only allowed to steal from the worker, Y, which stole the task it was trying to sync. Because Y had no other tasks when it stole from X, the only tasks it will have to steal will have been created by the task it stole from X. These tasks will have to be completed before the original task can be synced anyway, so both tasks are now working towards the same goal. In addition, there is no way that Y will complete the original stolen task while X is still busy working on something, so the time when there is code ready to execute and no worker to execute it should in theory be significantly reduced. Figure 2.2 on the following page taken from [20] should help explain this concept. Core A tries to sync task T1, and finds that Core B has stolen it. Instead of just waiting, or stealing from any other worker, it steals from the worker which stole the task it was trying to sync. That way, completing T1 should go faster, and as little time as possible is wasted.

Another technique to reduce the waiting time of workers in this situation is to just switch the entire worker thread. By having more threads than cores, there will always be another thread which can be switched in when a worker must wait on a sync because the task was stolen. Modern thread schedulers can typically accomplish this with quite low overhead. However, just having more threads than cores and letting the thread scheduler handle the switching itself will most likely lead to threads being switched back and forth too often, causing bad cache use. *Parking*, which Faxén introduces as a novel technique, is a compromise between letting the thread scheduler handle it and the standard approach to task-based programming, where each core has exactly one worker thread. When *parking* is used, there is more threads than cores, but in regular execution, only one thread is used per core, and the others are blocked, or *parked*. When a worker thread reaches a stolen sync, it will unblock a parked worker thread and go to sleep while it waits for the sync to complete. When the sync completes, there will be one more active thread than there are cores, but when a thread has nothing to do, it will check if there are too many active workers, and either block itself or try to steal work as usual. We have tried to explain

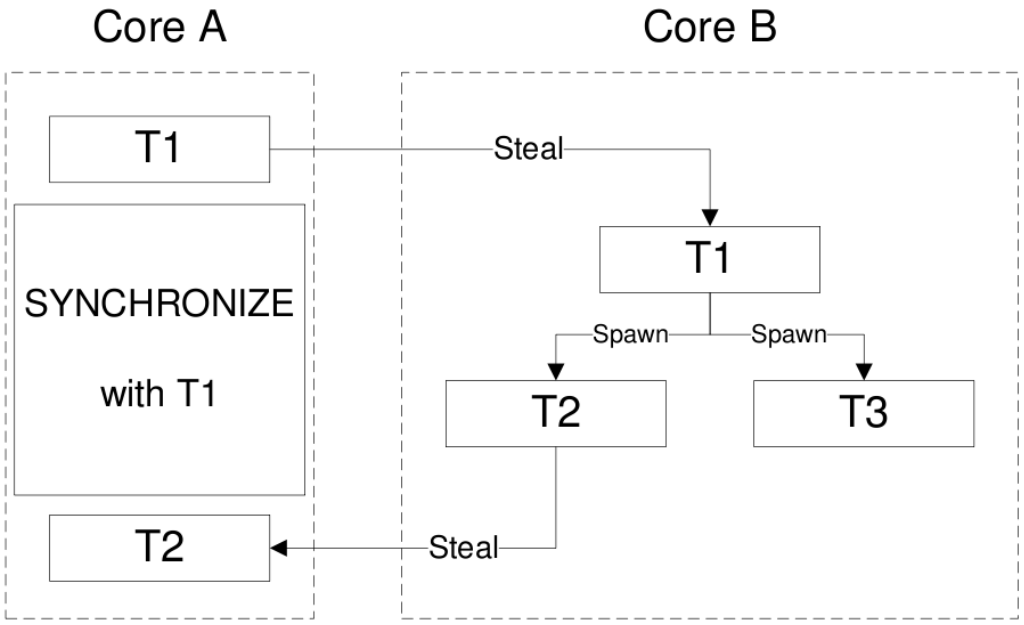

Figure 2.2: The concept of leapfrogging. Core A finds a task stolen, and steals from Core B, which stole the task.

the concept in Figure 2.3 on the next page.

2.2.5 Direct and continuation passing style

Whenever a task is spawned, it has to be synced at some point. The straightforward way of handling this is forcing the task which spawned a task to also sync it. This might however cause some problems when the task it is trying to sync has been stolen in the meantime. To show why this can happen quite easily, the pseudo code in Figure 2.4 on page 14 shows how a task normally spawns two new tasks in Wool. The `spawn` puts the task in the worker's task pool, allowing other tasks to steal it while the worker works on the task invoked by the `call` expression. Remember that `call` is equivalent to invoking `spawn` directly followed by a `sync`.

To avoid ending up in the situation where a task is stolen before it can be synced, there is a task-based programming technique called continuation passing. This is not implemented in Wool, but is described here because of its use in other task-based programming models. In fact, the developer has made a point of the fact that programmers are not forced to use continuation passing style when programming in Wool [15,

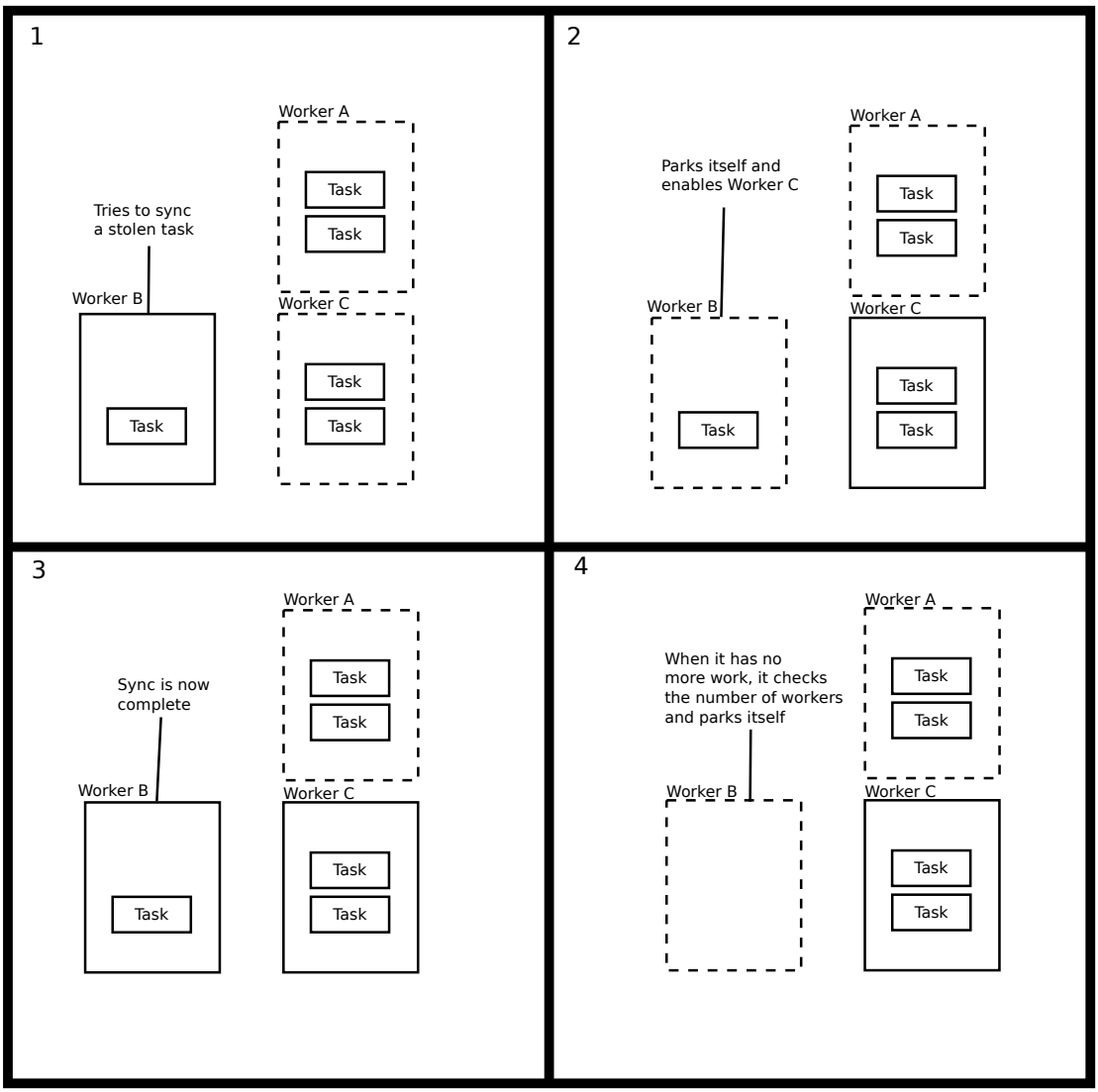

Figure 2.3: The idea behind parking. The figure shows just one of the CPU cores involved in a computation. Worker B tries to sync task which has been stolen by a worker on another CPU core. It parks itself and enables Worker C. When the task which Worker B originally tried to sync is completed, Worker B is unblocked and completes the syncing. At that point, there are two workers active on one core, but as soon as one of them, here Worker B, is idle, it blocks itself instead of trying to steal more work.

```
1 spawn SomeTask(parameter)
2 call SomeOtherTask(parameter)
3 sync SomeTask
```

Figure 2.4: The basic spawn-call-sync pattern in Wool

p. 1].

The point of continuation passing is to spawn a new task which becomes the "parent" of the other spawned tasks, and thus handles the syncing when they return. This way, the task which spawned the new tasks will not still be active while the other tasks are executing. It will instead have synced with the task that spawned it, and the worker thread will be free to work on other things. Figure 2.5 depicts in pseudo code how this might look in a language similar to Wool. None of the calls to **spawn** will block, so the task will just spawn off the three tasks and return. The continuation task is spawned just like the others, so it can also be stolen by another worker to help balance the load. Taken from [11], Figure 2.6 on the facing page illustrates the idea graphically. Where it is supported, continuation passing might provide a speedup, but it is a bit harder to program, and that is exactly why Wool does not force the programmer to write in this style [15].

```
1 spawn SomeTask(...)
2 spawn SomeOtherTask(...)
3 spawn ContinuationTask(SomeTask, SomeOtherTask)
4 return
```

Figure 2.5: Pseudo code describing continuation passing

2.3 Wool specifics

This section will describe in detail how to program, build and run applications using Wool, as well as describe the logging features already present in Wool.

Figure 2.6: The parent spawns off the children, and also a task which is the new parent of the children.

2.3.1 Programming

Because Wool needs to read arguments and set up worker threads before starting the program, it has defined the `main` function. After performing the needed setup, it will invoke the task called `main`. Instead of creating the normal `main` function, a programmer has to create a `main` task, which should look like this:

```
TASK_2( int, main, int, argc, char **, argv )
{
...
}
```

A common usage example for task based programming languages is the well-known Fibonacci example. Listing 2.1 shows a simple Fibonacci example written in Wool.

Listing 2.1: A simple Fibonacci example in Wool

```
1  TASK_1( int, pfib, int, n )
2  {
3    if (n < 2) {
4      return n;
5    } else {
6      int a, b;
7      SPAWN( pfib, n-1 );
8      a = CALL( pfib, n-2 );
9      b = SYNC( pfib );
10     return a + b;
11   }
12 }
13
14 TASK_2( int, main, int, argc, char **, argv )
15 {
16   int n = 35;
17   int result = CALL( pfib, n );
18   printf("pfib(%d) = %d\n", n, result);
19   return 0;
20 }
```

2.3.2 Building

When downloading Wool from the project's website [18], the archive includes many C-files, but they are mostly example programs. Other than the Makefile, the only important program files are `wool.c` and `wool.sh`. The point of `wool.sh` is to generate `wool.h`, which contains the task definition macros for the tasks and loop body macros. An input parameter <**n**> specifies the maximum arity of the tasks and loop body macros. If the file does not already exist, the Makefile will by default run the shell script with 10 as input parameter, creating tasks with arity 1 to <**n**> and loop body macros for arity 0 to <**n-2**>. `wool.c` defines most of the runtime, and also contains the `main()`. The point is then to compile the `wool.c` to a .o-file and link it to the program. In addition, Wool uses POSIX threads (pthreads) to manage the threads, so the pthread library will also need to be linked to the main program. The README file included in the Wool archive suggests the following typical command line:

```
gcc -pthread -O3 -o foo foo.c wool.o
```

2.3.3 Running

Wool defaults to using only one worker thread, so if the point is to do a parallel run, one has to specify the number of workers to be started with a **-p** <**n**> flag, where **n** is the number of workers to be started. This default value is stated in the Wool users guide [17].

Other useful input parameters to Wool include an **s** for specifying the number of stealable tasks in each worker's task pool, and a **t** to set the initial size of each worker's task pool. The first parameters passed to the program will be used by Wool if they are recognized, and the rest will be passed to the `main` task of the program.

2.3.4 Built-in logging

There is already some logging implemented in Wool. When Wool is compiled with the environment variable `COUNT_EVENTS` set, Wool will count all the Wool-specific events such as steals, leaps, spawns, steal attempts, spins and such, and write a summary to stderr when the execution is done.

In order to get even more detailed information, one can compile Wool after setting the variable `LOG_EVENTS`. In this case, many important events, and a timestamp for when they happened, are written out to stderr after the execution.

Naturally, LOG_EVENTS has a bigger impact on the running time than COUNT_EVENTS. LOG_EVENTS saves much more data, and each event will also trigger a call to a function to create a timestamp. COUNT_EVENTS, on the other hand, will mostly increment integers, and there is no timing involved.

While this information could have been used as a basis for the Java application created in this project, it was considered better to instead write new code to collect and write out the needed information. That way, only the information needed would be printed, and the data format could be easily chosen. In addition, the logging code already present was a nice help when trying to understand the Wool source code. For instance, the number of spawns are clearly listed, so it was easy to check whether the new code actually wrote out all the spawns.

COUNT_EVENTS focuses on 7 different events. The type of event and the timestamp are saved in the pre-allocated array of the worker the event happened to, and each event is written out in the following format: `EVENT [worker] [type of event] [timestamp]`. For sufficiently long runs, the file with the event logs will quickly become very large. It is not unusual to see files with over five million lines, which consumes over 150MB of disk space.

A '1' is a signal that a steal was completed successfully. The next step is obviously executing that task. A '2' is output when a worker has completed executing a stolen task, and all the tasks spawned from that task. This typically means that it has no more work and has to try and steal a new task. '3' is signalled when a worker has tried to sync a stolen task and starts leapfrogging to keep busy while the task is being completed. When the worker is done leapfrogging, typically because the task it was waiting for is completed, it signals a '4'. Every spawn is marked by a '5', while a '6' means that a task is synced. A number over 100 is output each time a worker tries to steal a task. The victim of the steal can be seen by reading the two last digits. These events are summarized in Table 2.1 on the next page.

Figure 2.7 on page 20 shows the most important part of the output COUNT_EVENTS produces. As we can see, the run seems pretty well parallelized, where all 12 worker threads have spawned over 100 tasks

Event	Description
1	Successful steal (or leapfrog)
2	Execution of stolen task (and all subtasks) completed
3	Leapfrog starts
4	Leapfrogging is done
5	Spawn
6	Sync
1xy	Attempt to steal from worker xy

Table 2.1: The different events created by LOG_EVENTS and their descriptions

each. As usual, worker 0 does not steal any tasks, but it actually performs the most leapfrogs of all the workers.

A very small part of the type of results generated by LOG_EVENTS are show in Figure 2.8 on the next page. In this excerpt, worker 0 (the worker that starts the whole computation) syncs two tasks, then signals that it starts leapfrogging, and attempts to steal from worker 1. The steal attempt was successful, which is signaled by outputting a 1. The worker then starts to spawn new tasks from the task it just stole. The timestamps are in nanoseconds, collected using `sys/time.h`, for which the resolution is dependent on the system. On the system used in this example, the clock does not have a high enough resolution to record nanoseconds, which is why every timestamp ends with three zeros.

A program trace like the one created by LOG_EVENTS is dependent on very accurate timing. Since the different threads will each save timestamps, the clocks are synchronized at the start of the run, and how much each clock was corrected is written out at the very start of the LOG_EVENTS output. Our experience suggests that the corrections are usually very small (a few microseconds) on the systems used in this project, but even a small offset might cause big misunderstandings about the program flow, so correct timing is crucial.

```
SIZES  Worker 256  Task 128 Lock 40
 Worker Spawns Inlined Read Wait St tries Steals  L tries Leaps  Spins
STAT  0    207     175    6   26        0      0   528489    23      0
STAT  1    180     147    8   25     2470     17   528279    23  41082
STAT  2    124      94   12   18     2475     18   598159     8  41159
STAT  3    132      95    8   29     2565     17   149858    21  40680
STAT  4    100      79   10   11     2516     20   411732     9  41268
STAT  5    163     133    5   25     2486     11   377852    19  40805
STAT  6    369     341    7   21     2418     17   526777    20  41093
STAT  7    167     140   10   17     2486     13   228528     8  40601
STAT  8    193     163    5   25     2510     22   590612    15  41466
STAT  9    120      84    9   27     2467     22   514256    13  41561
STAT 10    134      97    5   32     2462     15   706427    13  40866
STAT 11    159     129    3   27     2499     12   287296    15  40895
    ALL   2048    1677   88  283    27354    184  5448265   187 451476
```

Figure 2.7: A part of the output from a sorting run on Kongull after setting the COUNT_EVENTS compiler flag. Many interesting statistics are reported, but there are no timing associated with them.

```
EVENT  0   6      5913967000
EVENT  0   6      5916513000
EVENT  0   3      5916514000
EVENT  0 101      5916516000
EVENT  0   1      5916516000
EVENT  0   5      5916518000
EVENT  0   5      5916519000
EVENT  0   5      5916519000
```

Figure 2.8: An excerpt of the output produced by setting the LOG_EVENTS compiler flag. Each event has which worker it relates to, the type of event, and a timestamp.

Chapter 3

Profiling

For anything other than trivial computer programs, following the program flow and understanding where most of the time is spent might quickly become very difficult. Profilers can help programmers with both these problems, and in some cases they can also clarify why a particular section code consumes the time it does.

Creating these profilers can be very challenging. The goal is to collect extremely accurate and detailed information, while at the same time effecting the execution of the program being profiled as little as possible. As with many other things, computer programs will almost invariably act differently when it is being observed. Designing computer profilers will then inevitably involve a decision of where to compromise between accuracy and intrusiveness.

3.1 Data gathering

There are two main ways of collecting the data needed to create a useful output. Measurement-based profiling involves instrumenting the code so that it will report on its own execution. Statistical profiling, or sample-based profiling, will generally not modify the program, but will instead collect information by observing the system at specified times during execution. Both methods will introduce some overhead, but in general, the first method is both more accurate and costly in terms of overhead, whereas the second will be less intrusive and also less accurate.

3.1.1 Measurement-based profiling

To perform measurements on a piece of code, one will need to alter the execution in some way. Among the ways to do this are to use some sort of automatic tool to alter the source code, changing the code yourself, using the compiler to do it or running the entire program from inside another program.

Regular statistics provided by profilers are how many times each procedure is called, and how much time is spent in each procedure. For the first statistic, a straightforward way of calculating this number is just creating a counter variable for each procedure in the program and inserting code which increments the appropriate counter at the start of each procedure. Similarly, one can keep track of the time used in a procedure by collecting a timestamp at the very start and end of a procedure and then subtracting the two [21]. This will give an accurate time, but on most modern systems, the time will not be correct because of the multi-tasking operating systems which preempts processes on a regular basis.

For any programs of some size, the relatively uncomplicated methods mentioned above might soon be very time-consuming. One will always want to keep the overhead to a minimum, and one way to reduce it is to focus on basic blocks. Allen defines a basic block this way [22]: "A basic block is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed)." There can never be conditional jumps inside a basic block, so one can be certain that when a serial program has executed the first instruction, it will have to execute the last instruction in the basic block before it can do anything else. For counting how many times each basic block is executed, Ball and Larus have found the optimal way of selecting where the counter increments should be placed [23].

Hardware performance counters are also used by many profilers. These are dedicated registers on most modern processors, which keep track of many hardware events, such as instructions executed, cache misses and instruction stalls. By utilizing these, more precise data can be collected in a cheap way [24].

Measurement-based profiling can provide very accurate data, albeit at a higher cost than statistical profiling in general, because instrumentation will always incur an overhead. In addition, it is good for focusing on a specific part of code, when one already has an idea of where to look.

Different types of instrumentation will also measure different aspects of a program, so the profiler can be suited according to what one is looking for.

3.1.2 Statistical profiling

The basic idea behind statistical profiling is to collect samples about some part of the system at regular intervals, and create statistical approximations of what is happening based on those. The program counter, or instruction pointer, will for example show what instructions are being performed at any given time. A straightforward way of creating a statistical profiler is to sample the instruction pointer often, and then approximate the time for each instruction based on that. If the pause between sampling intervals are smaller, the approximation will be better, but the time it takes to collect the samples will also be more noticeable. Hardware performance counters can also be used in statistical profiling, where they can provide accurate information with a very low collection cost.

Statistical profilers will typically be able to differentiate between the program being profiled and all the background processes of the operating system. Thus, one will often get quite accurate results using statistical profiling, even though they are not based on precise measurements. By nature, statistical profilers will not need any instrumentation of the code. Thus, the programs are allowed to run at nearly full speed, and the profiler will be able to analyze a normal run of the code as opposed to an altered version which may not perform as it normally would have. Because there usually is no need to change the code before profiling, many statistical profilers are easy to use.

A regular use for a statistical profiler is to profile an entire application and get a quick overview of where the application is using time, and thus where the first optimization efforts should be focused. They will often not be useful when one wants to use profiling to answer a very specific question about only a small part of the code.

3.2 Types of output

For a single threaded performance profiler, the most important thing is generally to identify what function, method, basic block or even line

of code uses the most time. These points are often referred to as hot-spots [25, 26]. To make a program run faster, there is most to be gained by first optimizing the part that runs the slowest. There are several ways of displaying that information to the programmer.

In addition to the performance profilers, there is another set of tools designed to help a developer understand what is going on in a program. In an influential paper written in 1994, Srivastava and Eustace opens the introduction in the following way:

> "Program analysis tools are extremely important for understanding program behavior. Computer architects need such tools to evaluate how well programs will perform on new architectures. Software writers need tools to analyze their programs and identify critical pieces of code. Compiler writers often use such tools to find out how well their instruction scheduling or branch prediction algorithm is performing or to provide input for profile-driven optimizations" [27].

For the current state of computer science, it is appropriate to also extend the list of important use cases to also include seeing how a parallel computation is performed on a given system.

Seyster divides software execution visualizations into four main categories [28]: Those showing the steps a program takes as it runs, those focusing on data structures, those with features especially suitable for object-oriented programs, and those showing other related properties, such as CPU load or memory access time. For this project, only the first and last of the categories will be touched upon.

3.2.1 Flat profile

Perhaps the simplest type of profiling output is the *flat profile*. This is often generated by both measurement-based and statistical profilers, and it is normally just a list of how many times each procedure was called, and how long they took to complete. Clearly, it can still be very useful, and for a first look at a program before starting to tune, it is often all that is needed.

Below is an example of a flat profile created by gprof, and taken from the gprof project's website [29]. Among other things, it shows the number of times each function is called, how many seconds is used inside each

function, the percentage of the entire program each function uses, and the name of the functions. Clearly, when a quick overview of the program is needed, a flat profile will suffice.

```
 Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
...
```

3.2.2 Call graph

A slightly more advanced output is the call graph. The main point here is showing how the called procedures relate to each other during a run. The profiler keeps track of all the procedure calls during execution, and can then create a graphical representation of the run after the fact. This is called a dynamic call graph, since it shows a computation run as it actually happened. A static call graph is an approximation which shows all possible ways procedures could call each other.

The actual graphical representations of a call graph vary from just an indented text list, to a color-coded image, to a drill-down GUI. Figure 3.1 on page 27 shows an example call graph generated by the Google CPU profiler, and taken from the project's website [30]. Procedures are represented by nodes in the graph, and edges between nodes indicate

that a procedure has invoked another procedure. There are also numbers on the edges representing the amount of time units spent in a function when it was called from within the other function.

3.3 Online vs. offline

By far the most usual display tool is an offline application, which will display information about the execution after the fact. There are some online profilers, however. These will allow the profiler to see the profiling information as the program executes, and even make changes to the application and watch the difference in the profiler during the run. An example of a profiler which can profile an already running process is the Intel VTune Amplifier, which is introduced in more detail in Section 3.5.3 on page 30. It cannot, however, alter the running process from within the profiler. An advantage with an online profiler is that a developer can use the application and instantly see the results in the profiler [28].

This project will result in an offline profiler. Not only is it easier to create, the point of the profiler is to watch entire runs of parallel programs. There is therefore no need to be able to make changes at runtime.

3.4 Parallel Profiling

Creating parallel applications is generally harder than writing serial ones. When many threads, divided among many cores, are working on the same computation, it might be hard for a developer to know exactly what happened where. And even when using a model like POSIX threads, where the threads have to be handled explicitly by the programmer, there is always the risk of such subtle errors as deadlocks or data races. A parallel profiler should ideally be able to help the user with all these aspects which are characteristic of parallel programming, while also having many or all of the features of a single-threaded profiler.

In a badly balanced parallel computation, a profiler might report that one of the threads is using much time in a function called `spin_lock`. A good parallel profiler would be better off instead indicating that the thread has nothing to do. A thread in a spin lock is waiting for a signal from another thread before it can move on. An inexperienced

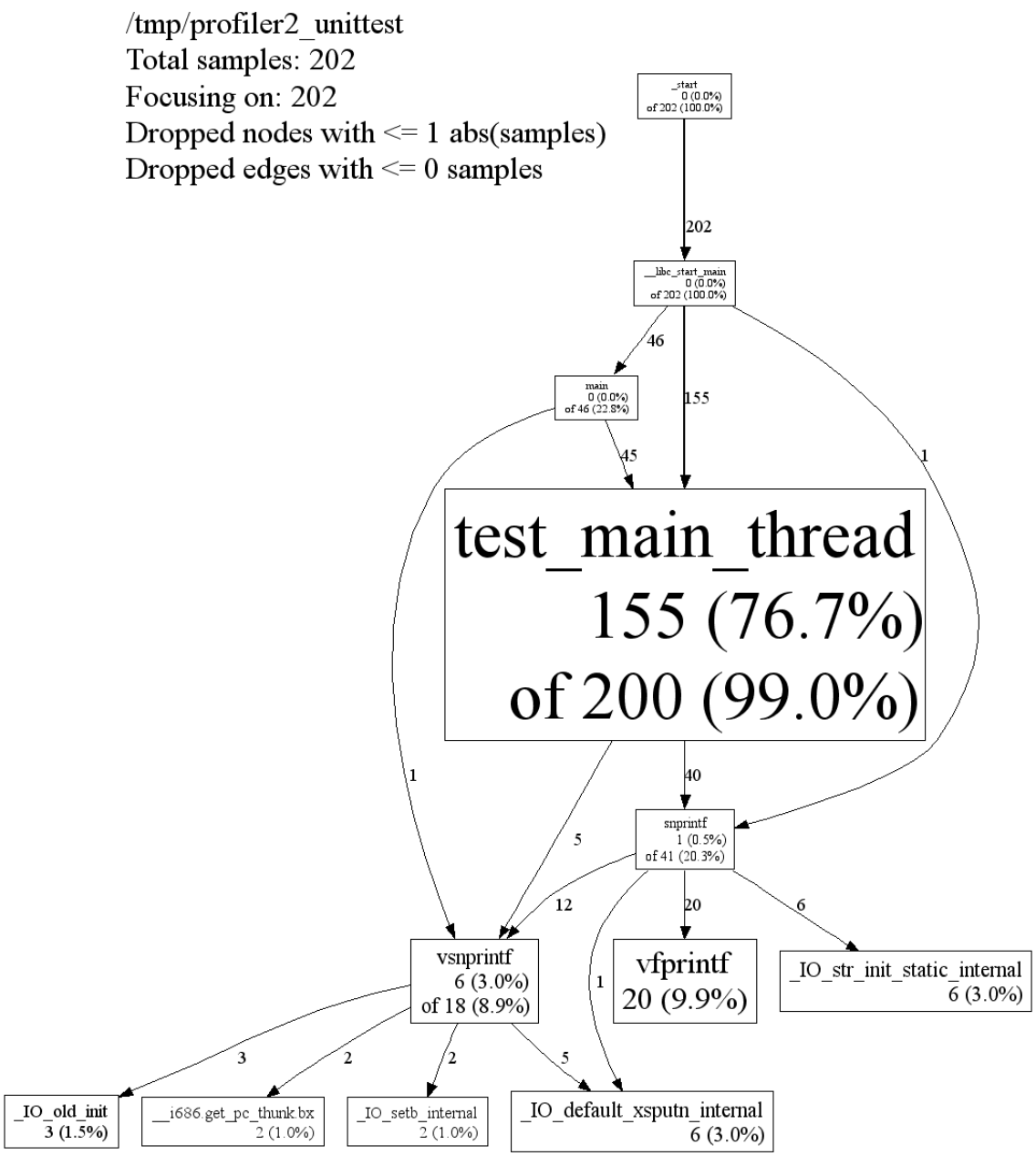

Figure 3.1: A call graph generated by the Google CPU profiler. Each node in the graph represent a procedure. Unlike the flat profile, a call graph illustrates the call relationships between procedures. Notice also that the size of the nodes change according to the amount of CPU time the functions consumed.

programmer might think that this `spin_lock` function is worth trying to optimize instead of immediately focusing on better balancing the load.

For a sequential program, there is usually no problem determining in what order events occurred, because no two events can happen at the same time. For a parallel program, however, events may happen concurrently, at least seemingly. Accurate and synchronized timing need to be in place if the exact order of events are to pieced together.

Several profilers exist to analyze runs where MPI or explicit threading is used. These tend to focus more on the cost of sending data and synchronization between the involved processors. For the purpose of this report, however, it makes more sense to focus on some of the profilers which support task-based programming. Runs using a task-based programming model will include dynamic load balancing and no guarantees as to how the distribution of work will turn out. For that type of run, it is more interesting to get some indication of whether the work was distributed evenly or some indication of whether the application will scale, rather than reports on the cost of communication or synchronization.

3.5 Related work

3.5.1 gprof

In 1982, in what has since been considered a very influential paper, Graham, Kessler and McKusick detailed the design and use of the call graph execution profiler gprof [21]. The whole idea is based on the modularity of programs, or the fact that they consist of relatively small routines which call each other. The main point of the profiler is to create a call graph, a tree showing how the routines are called, and also display a flat profile which shows each routine and how much time they, and the routines they call, use.

Gprof uses a combination of statistical and measurement-based profiling. To determine the arcs of the call graph, and count the number of times each routine is executed, gprof instruments the code with a call to a monitoring routine in the prologue of each routine call. Execution times are approximated by sampling the program counter at uniform intervals.

An example of one type of output from gprof, the flat profile, is shown in Section 3.2.1 on page 24.

3.5.2 OProfile

Originally started as a master's thesis project by John Levon, OProfile has since been incorporated into the Linux kernel. It is a statistical profiler, using periodically collected samples to create an overview of how much time each process used during the profiling run. On some architectures, OProfile will even use performance monitoring counters to provide the programmer with detailed information about such low-level events as branch predictions and cache misses [31]. In general though, because it has such a broad scope, OProfile will usually not be able to help a programmer with the most extreme, accuracy-critical, specific profiling.

OProfile is system-wide, meaning it will profile everything being executed on the system in the given time. Even though this might seem unnecessary for a programmer just wanting to profile his or her sorting algorithm, it does provide a realistic and accurate picture of what is happening on the system. The overhead is quite low, typically between 1 – 8% according to the project's web page [32]. Also, the profiling process should be easy for the programmer, since no changes are needed in order to profile a piece of code. There is a feature, however, which will create annotated source code if the binary has been compiled with the **-g** option.

The following is an excerpt of the quite simple output generated by OProfile for a "system-wide binary image summary". It is taken from the OProfile webpage [32], and shows the percentage of CPU time used on each process for the sampled timespan.

```
$ opreport --exclude-dependent
CPU: PIII, speed 863.195 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted)
with a unit mask of 0x00 (No unit mask) count 50000
   450385 75.6634 cc1plus
    60213 10.1156 lyx
    29313  4.9245 XFree86
    11633  1.9543 as
    10204  1.7142 oprofiled
     7289  1.2245 vmlinux
     7066  1.1871 bash
     6417  1.0780 oprofile
     6397  1.0747 vim
     3027  0.5085 wineserver
...
```

3.5.3 Intel VTune

Marketed as a high-end, very advanced performance profiler, the latest edition of the Intel VTune line is the VTune Amplifier XE 2011 [26]. It is available as both a standalone application in Windows and Linux, as well as a plug-in to Microsoft Visual Studio for Windows users. VTune is a mature product line, and Intel uses its expertise in hardware to provide very detailed information to users running code on Intel CPUs. In addition, a focus on parallel profiling enables features like a time-line with displays the behavior of each thread and highlighting of potential locations to optimize. A special "Locks and Waits-analysis" is intended to help a developer easily find places where threads are not being utilized because they are waiting on locks set by other threads. Other notable features include system-wide, event-based sampling, tight source code integration, and the ability to attach the profiler to already running processes under Windows.

To help programmers create correct code as opposed to making working code faster, Intel also provides Parallel Inspector which among other things can discover memory leaks and possible errors due to concurrent memory accesses. Intel Parallel Studio combines both these tools as well as the Parallel Advisor and Parallel Composer [33].

A screenshot of the concurrency timeline in the latest VTune version is shown in Figure 3.2 on the next page. The screenshot is taken from the program's online documentation [34], and it displays each thread separately, with colors marking whether they are running or waiting, and an aggregate timeline showing the overall concurrency of the execution at any given time.

3.5.4 AMD CodeAnalyst

In many ways the counterpart of Intel VTune, AMD CodeAnalyst Performance Analyzer can extract detailed information from performance counters specific to AMD's processors. Like Intel VTune, it is available as a standalone application to Windows and Linux, as well as a plug-in to Microsoft Visual Studio under Windows. Among the features are both event-based and time-based profiling used in hot-spots-analysis, as well as instruction-based sampling and good support for multi-core profiling. In addition, CodeAnalyst is able to analyze the performance of managed, (just-in-time compiled) Java and .NET-code [25].

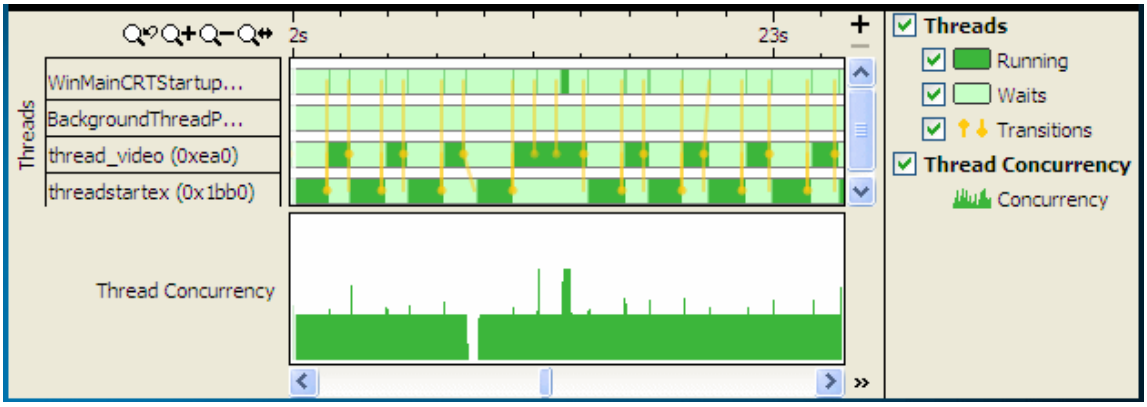

Figure 3.2: The threading timeline from the Locks and Waits in Intel VTune Amplifier XE 2011. Each thread has its separate timeline, with colors representing the status at any given point in the calculation. The overall concurrency is displayed in an aggregate timeline at the bottom.

Figure 3.3 on the following page taken from the application's website [25] shows AMD's version of a threading timeline similar to Intel's one in Figure 3.2. Green bars represent user activity in a thread, so a user is able to get a quick overview of whether all the threads are working as much as they could. It is not as polished as Intel's version, and it is also lacking an aggregate bar displaying the total concurrency.

3.5.5 ompP

OmpP is a measurement-based profiler built specifically to work on several OpenMP compilers and runtimes. Instead of reporting only on routines, ompP will also collect and display data for special OpenMP events such as critical sections. In the case of task-based programming, this means that ompP will instrument each call to `task` and `taskwait` (similar to Wool's `spawn` and `sync`), and subsequently report on which threads are executing tasks at which time. If instructed to by the user, ompP can also use hardware counters.

The following flat profile is an outtake from [35]. It shows "the time threads spend executing tasks while waiting at the implicit exit barrier of the parallel region." Thread 0 was busy for 3 seconds executing tasks, while thread 1 only spent 2 seconds executing tasks, as can be seen in the rightmost `taskT` column. Because there was not enough tasks to execute, thread 1 spent 1 second waiting at the barrier before it could continue, as

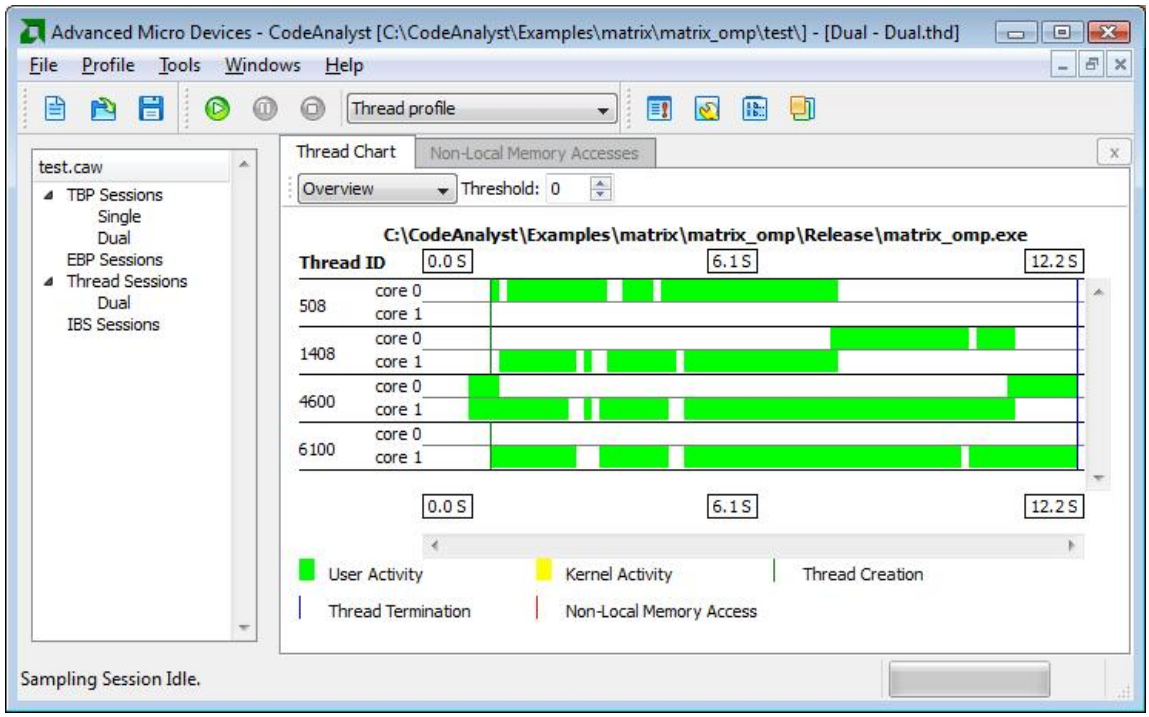

Figure 3.3: The threading timeline from AMD CodeAnalyst. User activity in a thread is represented by a green bar. Typically, one want as much concurrency as possible, so the greener, the better.

the `exitBarT` column shows.

```
R00001 main.c (15-26) PARALLEL
TID  execT  execC  bodyT  exitBarT startupT shutdwnT  taskT
0    3.00   1      0.00   0.00     0.00     0.00      3.00
1    3.00   1      0.00   1.00     0.00     0.00      2.00
SUM  6.00   2      0.00   1.00     0.00     0.00      5.00
```

3.5.6 Google CPU Profiler

As the name implies, Google CPU Profiler is both developed and used by Google [30]. The output mode is either textual or graphical. In graphical mode, the profiler paints a call graph, and displays running times based on sampling as well as how many percent of the total running time a method consumed. In Linux 2.6 and above, the profiler automatically profiles all threads. There does not seem to be any special support for parallelism as in many of the other profilers described here, however.

To use the profiler, one needs to link it into the executable, and then enable it either by function calls in the code, or by setting an environment variable. After the code is run, there are several ways of displaying the output. It is a statistical profiler, because the timing is collected using sampling, and by default it collects 100 samples every second. An example call graph output is shown in Figure 3.1 on page 27.

3.5.7 Cilkview

The Cilkview scalability analyzer [36] takes a bit different, yet very interesting, approach to profiling multithreaded applications. It is built specifically for Cilk++, and what differentiates it from many other profilers is its estimation capabilities. Instead of monitoring an actual multithreaded run, it measures the logical parallelism during an instrumented single-processor run and predicts the maximum potential speedup of the application.

To calculate the parallelism of an application, Cilkview employs the **dag model of multithreading**. This envisions the execution of a parallel program as a directed acyclic graph (dag), with each vertice being a piece of code which have to be executed sequentially. These vertices are called **strands**. Whenever a strand spawns another task, the spawning strand

naturally becomes the predecessor of the newly spawned task in the dag. An in-depth explanation of the model can be found in [37, Ch. 27].

After using the model to represent the execution, the parallelism of the application can be calculated. **Work** is the total time needed to execute all the strands in the execution. In other words, the time it would take to run the program with a single processor. If a program is perfectly parallelizable, the time needed to execute it would be $work/P$, where P is the number of processors. This is extremely rarely the case, however, because there almost always is a substantial part of the program which will have to be executed sequentially. **Span** is denoted as the time needed to execute this longest, sequential part of the program. The longest path through the dag is often called the critical path of the program. The concept of critical path is illustrated in Figure 3.4, which shows a dag of tasks, with the longest unparallelizable part marked in red. The parallelism is defined as $work/span$. This is quite logical. The more work there is, or the faster the critical path can be executed, the more parallel the program is. Using these measurements, Cilkview provides speedup estimates for different number of processors.

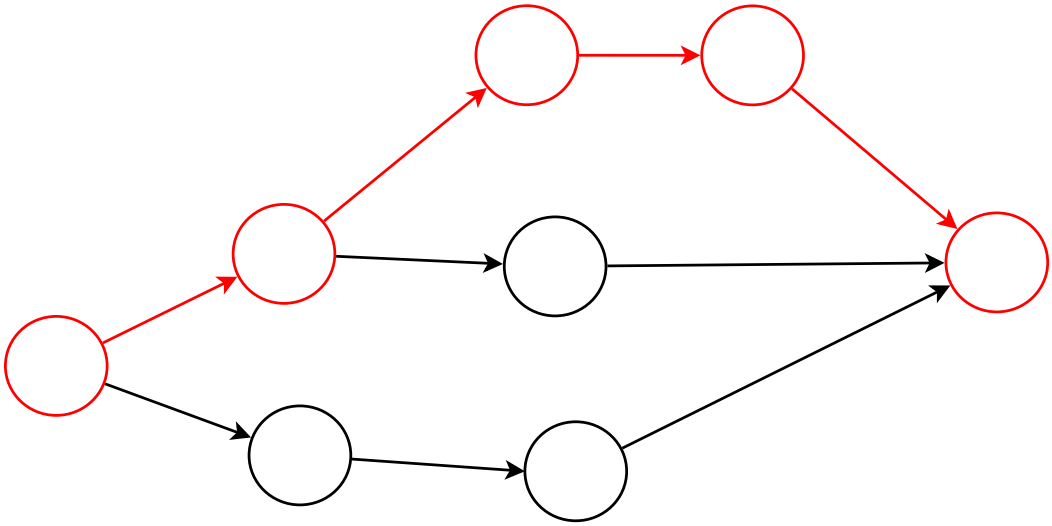

Figure 3.4: The concept of critical path. The circles are tasks, all of same size, and arrows indicate dependencies. No matter how many processors are used, the computation can not finish faster than one processors can complete the tasks in the critical path, which is marked in red.

He et al. are a bit unclear on exactly how Cilkview actually calculates

the critical path. It seems to be quite complex, but the concept is explained as follows: By instrumenting the run, and then carefully keeping track of which tasks can execute logically in parallel, and which tasks have to be executed after one another, they get a measure of the span of the computation. Instruction-counting is used instead of direct timing. Although it is not very accurate, it is stated to be good enough for calculating work and span.

The measurements collected on a single core does not account for the overhead incurred by the scheduler or the cost of moving tasks between processors to actually benefit from the available parallelism. The authors introduce a concept called the **burdened-dag model** which tries to account for the performance impact of migrating tasks to different worker threads. By assuming that all possible parallelism is realized, a worst-case cost of migration overhead is worked out. This is added to the already existing dag by introducing a **burden** on each continuation and sync and the **burdened parallelism** can be evaluated in a similar manner as for the unburdened dag. This new estimate is typically a little lower than the unburdened counterpart if the application is parallelized in a proper way. If there are too many spawns of small tasks, however, the burdened parallelism will be much lower than the theoretical parallelism, and the programmer will have a good idea of what to improve in the application.

Figure 3.5 on the following page shows an example graph created by Cilkview taken from [36]. It is the scalability analysis for a file compression tool called bzip2. The blue area in the graph represents the range Cilkview has estimated that the speedup will lie in based on the number of cores the application is executed on. The horizontal line represents the application parallelism as calculated by Cilkview. Naturally, the speedup will never go above the application parallelism. Furthermore, it is usually difficult exploiting all available parallelism in practice, so one would generally want to try and have a lot more parallelism than cores. This screenshot also includes real benchmarking results from a run on 8 cores represented by the line of red stars.

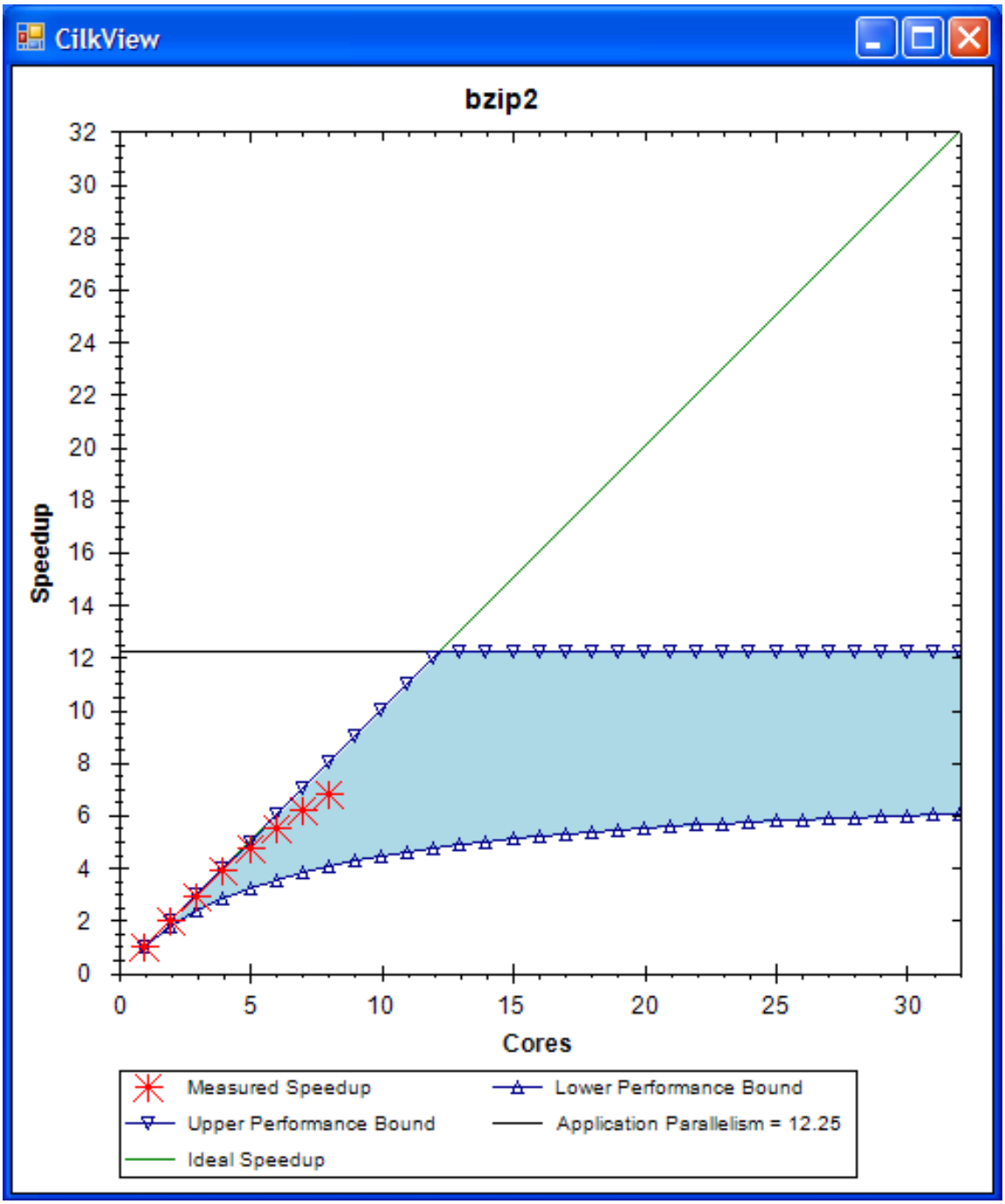

Figure 3.5: Cilkview's output for compressing a 28MB file using bzip2. The horizontal line depicts the application parallelism of 12.25, and the blue area is the scalability range Cilkview has estimated for the application. The red stars show the actual speedup achieved when using 8 cores.

Chapter 4

Implementation

In order to visualize a Wool computation, there first had to be collected some data. The idea was to change the Wool source code to make it print out some information about its own computations. That collected data could then be put into a separate application which creates a visualization of the run. Figure 4.1 on the next page shows a conceptual overview of the process.

This chapter will describe in detail how WoolPlot was implemented. First, the data collection stage will be described, which mostly involved instrumenting the Wool source code, and allowing the measuring code to be switched easily on and off by hiding it behind preprocessor symbols. Second, the profiler application itself will be explained, which is written in Java and uses the output from the instrumented Wool source code to provide a visual overview of the computation. There is also a little post-processing of the data involved to find the critical path of the computation.

4.1 Data collection

An instrumentation-based approach to collecting data was chosen quite early in the project. To get accurate data about spawns and steals, there is really no other way of doing it. In order to get information about the CPU load, however, a statistical approach where the load data is polled at regular intervals was chosen. Common among all the data collection is that it is not included in the code unless specific compiler flags are set. The reasoning behind this is that WoolPlot should have no impact on the Wool code unless it is activated.

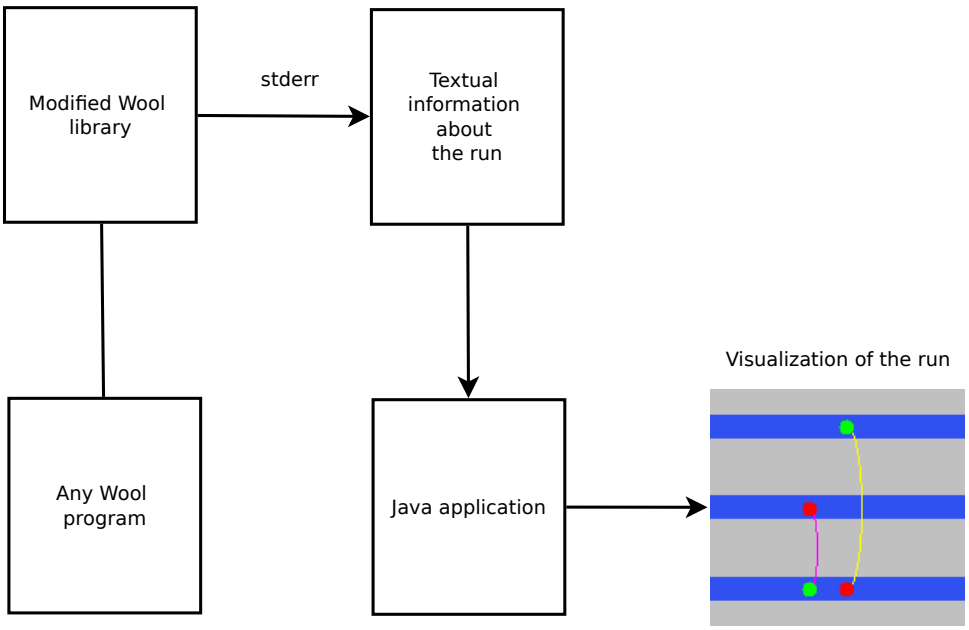

Figure 4.1: An overview of the profiling stages. The implementation chapter will describe how the Wool library was modified, and how the Java application was built.

With Wool being a task-based library, it was clear that the profiler should include information about spawns, steals and leaps. This is mostly interesting to someone interested in Wool's internals. To try and make WoolPlot more useful for an everyday user of Wool, however, CPU load and critical path was included later in the process.

4.1.1 Steals and leaps

To visualize a steal, one would need to know who the thief is, who the victim is, and when the steal occurred. The timing would clearly need to be as accurate as possible, ideally one would want to save the exact time when the steal took place. In order to accomplish collecting this timing, the steal-function in Wool had to be extended slightly.

When the STEALS compiler flag is set, the preprocessor will instrument the steal-function to save each successful steal in a special steal struct. The struct, which contains a timestamp and the victim of the steal, is then saved in a pre-allocated 2-dimensional steal array where the first dimension is decided by which worker is the thief. As for all the data collected in this project, the steals are printed to stderr after the main method, or rather the main task, returns.

The leapfrogs are collected in exactly the same way as the steals. The only real difference between the saving of the two is a small check inside the steal-function to see if the steal we are about to save is a "normal" steal or a leapfrog. The steal-function has a parameter of type Task* named `dq_top`. This is used for executing a task after it is successfully stolen. For a regular steal, this will point to the base of thief's deque. That is then used to differentiate between the two types of steals.

4.1.2 Spawns

For each spawn, we are interested in who spawned it, when it was spawned, and what the name of the task is. All this data is put in a spawn struct and saved in a pre-allocated 2-dimensional array at every spawn, in much the same way as the steal structs are saved. Getting and saving the task names from task spawn macro functions was not straightforward, but it was accomplished using the following two small macro functions: `#define STR_EXPAND(tok) #tok` and `#define STR(tok) STR_EXPAND(tok)`. By inputting the name of the

task into the STR macro function, it could be saved as a char* in the spawn struct.

4.1.3 CPU usage

A typical use for a profiler is helping a programmer make an application faster by showing what parts of the application consume most time. This is a basic feature in most profilers, and it is often displayed to the user through a flat profile or call-graph. Because this is such a regular feature, and since it has little to do with parallelism, we did not want to implement that. To make the profiler more useful for an everyday user, however, we wanted to add some information about the performance of the different cores. By displaying the utilization of each core, it is simple to see whether all the cores are working hard, and thus whether the application is sufficiently parallel. Wool is supposed to map one worker thread to each CPU core, so collecting usage data per core seems reasonable, since there should never be more than one worker thread on one core.

Options

There are several ways of calculating CPU load. Most modern CPUs have special hardware registers called performance counters. These provide detailed information about hardware events, and can be used to accurately determine the CPU usage [38]. For this early proof-of-concept profiler, the very accurate data was not considered to be worth the difficulty in implementing the use of hardware counters. Hardware counters are a very good solution, however, and it should absolutely be considered for further work.

An easier way of getting information about the CPU is by using the /proc file-system. This is included in the Linux kernel, and is a ”pseudo-file system which is used as an interface to kernel data structures“ [39]. Among other things, it provides information about all processes, for example how much CPU time is scheduled for each process. In addition, there is an aggregate file, which provides information both on the system as a whole, and on a per-core basis.

CPU specific data

The `/proc/stat` file is the aggregate file. It can vary from architecture to architecture, but the entries which is used in this project are very common throughout. The first line contains statistics for the system as a whole, and is not used by our profiler. Then follows one line for each CPU core in the system. It contains four numbers, which is the amount of time "that the system spent in user mode, user mode with low priority (nice), system mode and the idle task, respectively" [39]. The time is reported in a unit called USER_HZ, sometimes called a jiffy. On most architectures, it is 10ms, but will often vary. This use of USER_HZ shows the very real limitation of using the /proc file-system for measuring CPU load. One will never get more accurate data than the USER_HZ on the given architecture.

Because the USER_HZ varies between architectures, one might think that one will run into trouble when using WoolPlot across different architectures. This is easily avoided, however, by only sticking to using these relative numbers. By never converting USER_HZ to seconds, the entire problem disappears.

Process specific data

In addition to the aggregate data, we also need some process specific numbers. These are located in /proc/[pid], where [pid] is the process id. Each process is assigned an id, and all the specific info pertaining to only that process is then kept in the folder with the same name as the process id. Again, there is a stat-file containing a lot of useful statistics. The entries we care about are only state and num_threads. State is a single character describing the state of the process, and num_threads displays the number of threads currently associated with this process.

To calculate just how much a specific CPU core have been working on that process, we have to dig even deeper into the /proc file-system. Located in /proc/[pid]/task/[tid], where [tid] is the numerical thread ID of a thread, is a thread-specific stat-file for each thread working on the process. The format of this stat-file is equal to the /proc/[pid]/stat-file, but the data is aggregated per thread. The entries we care about in this file is utime, stime and processor. utime is the amount of time, again measured in USER_HZ, this thread has been scheduled in user mode, and stime is the same only for kernel mode. The processor entry displays

what CPU number this thread was last executed on. We use this last bit of information to map the worker threads to specific CPUs.

Load calculation

The way we calculate the CPU load over time is then as follows: Scan the aggregate and all the thread-specific files in the process-specific directory at regular intervals. Calculate how much CPU time was scheduled on the process by each thread since the last scan, and compare it to the total amount of CPU time scheduled on each core since the last scan. A percentage is then calculated for each core, and saved together with a timestamp representing when this was checked.

Accuracy

Contrary to the other data collection in WoolPlot, this is an example of statistical data collection. The resulting numbers are statistical approximations for what happened between two checks. The accuracy could have been improved by making the interval between checks smaller, but that would also make the polling more intrusive. The files in the /proc filesystem is typically only accurate down to 10ms anyway. That means that if the file is polled every 10ms, one would only either get a 0 or 1. In this project, the polling interval has been set to 200ms in order to get a useful amount of data for each sample. Clearly, this will not give a high enough accuracy to be useful for small, performance-critical applications. For sufficiently long runs, however, it will at least give a clear indication of whether the CPU cores are utilized as much as they should.

Forking

In order to alter the Wool code as little as possible, and allowing the calculation of CPU load to happen concurrently with the execution of the Wool program, the Wool process is forked. Forking splits the process into two [40], allowing Wool to execute normally, while the forked process will take care of polling for the CPU data. The process state, fetched from the /proc/[pid]/stat-file is set to Z (for Zombie) when the the process terminates. That will trigger the forked process to stop polling and write out the data it has collected. Clearly, the two processes will

not execute perfectly concurrently, but it will appear so because the monitoring process will be given enough CPU time to perform the checks.

Spinning

When experimenting with this newly implemented functionality, we tried running different programs and watching how the CPU load changed over time. Interestingly, the CPU load would never drop below 90% or so, even for benchmarks which was obviously not well parallelized. For example, a merge sort without a parallel merge would be expected to have all the CPU cores but one idle towards the end of the execution. After looking extensively for errors in the CPU polling code, the reason was found in the Wool code itself. When a worker fails to steal a task, it will back off a little by entering a `spin()` function, which contains a for loop designed to keep the worker busy for a varying amount of time. That means that the CPU cores will report a high load, even if they are not actually doing any useful work.

To try and see if we could get the desired results from the CPU load code, we tried changing the code in the `spin()` function to an appropriately sized call to `nanosleep()`. Sleeping for a microsecond was found to work adequately. With this new `spin` function, WoolPlot would now report cores as idle when they had no work to do.

We suspected that changing the `spin()` function could hurt the execution. To investigate this, we ran some experiments designed to only measure the impact of sleeping instead of doing useless work. Those can be seen in Section 5.4.2 on page 73. Anyway, the change will only come into effect when the CPU_POLL compiler flag is set. When not using the profiler, the normal `spin` function is used.

Portability

As it turned out, this method of collecting CPU load data was not actually as portable as we had hoped. The formatting of the stat-files are slightly different on the development machine and on Kongull. The difference is only that there is one more entry on the development machine, but it complicates the testing nonetheless, and one would have to expect that the code polling the CPU might have to be tweaked, or at least checked, every time WoolPlot is introduced to a new system.

4.1.4 Wool versions

The current version of Wool is 0.1.2alpha. This is the version used mainly in this project. There is also an *alternate version*, Wool 0.2. Despite the version numbering, this version is actually older. The main difference is that in Wool 0.2, the main method does not have to be a task. One can invoke a parallel execution directly from sequential code using the macros ROOT_CALL and ROOT_FOR. This makes it easier to add Wool to an already existing project, or parallelize more complicated code. A small part of the simple Fibonacci example written in Wool 0.2 is shown in Listing 4.1. On the other hand, Wool 0.1.2alpha implements event logging, which is described in Section 2.3.4 on page 17, it supports IA64 under Linux and it should also be a lot faster according to the Wool project's website [18].

There is a Wool version of the Barcelona OpenMP Tasks Suite (BOTS), which we were given access to during this project. Unfortunately, this version was written for Wool 0.2, and just changing the benchmarks was nontrivial. Thus, it was decided to try and port WoolPlot to Wool 0.2 instead. This way, we could just include the bare necessities of the compiler, and try to incorporate them as cleanly as possible in the code. As the porting progressed, however, the more subtle differences between the two versions became very clear. Version 0.2 does not implement event logging, so many of the features which was used to originally create WoolPlot were simply not available. For instance, code used to get timestamps, and methods to synchronize the clocks between the different worker threads were missing. Also absent was such a basic, convenient field such as `worker->idx`, which made it easy to save which worker was doing what. Eventually, the porting effort was abandoned, and we were so fortunate as to be given some BOTS programs written for Wool 0.1.2 by Karl-Filip Faxén.

4.1.5 C Preprocessor macros

In order to create Wool as a fast library to be used with plain C, instead of a language of its own, the author has used C macros extensively. Macros are interpreted by the preprocessor and transform programs before they are compiled [41]. There are almost endless uses for this, but below are listed the some of the ones most relevant to Wool.

Function-like macros All the task definitions are created using this

Listing 4.1: An excerpt of the Fibonacci example in Wool version 0.2

```
1
2
3  int main( int argc, char **argv )
4  {
5    int m;
6    int n = 35;
7
8    //initialize Wool and start the workers
9    wool_init( &argc, &argv );
10
11   //invoke parallel code from sequential code
12   m = ROOT_CALL( pfib, n );
13
14   //stop the Wool workers and clean up
15   wool_fini( );
16 }
```

technique. This allows defining and invoking named tasks without having to introduce new language features.

Conditional includes This makes it easy to include or ignore lines of code based on a macro variable. This variable can even come from the environment, so no changes has to be done in the code itself to turn on and off features like event logging.

Architecture-based includes Based on different environment variables being present on different architectures, this causes the same Wool archive to work on both Linux, Solaris and Apple.

Commenting out It is easy to comment out large sections of code, even if there are other comments in the middle of it. Wool is a work in progress, so there are sections in the code which is commented out in this way.

Constant definitions When pushing performance to the limit, defining constants without actually creating a variable will save the compiler some work, and enforce that no time is wasted by suboptimal optimization.

4.1.6 C macros used in this project

Using macros in C is a very easy way to switch on and off features in the code without using branches which have to be evaluated at runtime. The macros used in this project are described below.

STARTENDTIMES This collects a timestamp directly before and after the main task is called, and prints them out together with the number of worker threads.

STEALS This prints out a list with every steal in the execution, with thief, victim and a timestamp.

LEAPS This prints out a list with every leapfrog in the execution, with thief, victim and a timestamp.

SPAWNS This prints out a list of every spawn, with who spawned it, when it was spawned, and what it's called.

CPU_POLL This gives a list of how much each thread used the processor printed for regular intervals.

PROFILER This is only for setting all the macro flags at once, so it is easy to turn all the features on or off.

4.2 Java UI

4.2.1 Visualization

When considering how to portray the computation, it was quite early decided to make the timespan of the computation one of the axes in the overview. Also, because the GUI had to show spawns and steals, a natural approach would be to display each worker thread separately. In this respect, Intel's Parallel Amplifier XE, and its "Threading time line" has been an inspiration. A screenshot is included in Section 3.5.3 on page 30[1]. With all the worker threads represented by a bar, which also represents the timespan of the computation, how to visualize spawns and steals was quite straightforward. They would of course have to be placed

[1] A small introduction of it can also be seen from 1:35 of this Youtube video: `http://youtu.be/n4z5p8f5L-A`.

appropriately on the timeline based on when they occurred, and the steals would be painted on both the involved worker bars. As more features, such as critical path, were included in the visualization, the initial model turned out to hold up well, and the new features found a natural place in the interface. The following subsections will describe each GUI element in more detail, before the interface as a whole will be briefly described at the end of the section.

Steals and leaps

Steals and leaps were the first implemented features. They are visualized by painting a small red circle on the victim's bar, a small green circle on the thief's bar, and an arc in between the two. For a steal, the arc is yellow, while it is magenta for the leaps. Figure 4.2 shows a closeup of how the steals and leaps are portrayed.

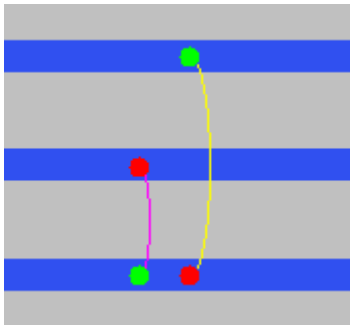

Figure 4.2: Detail view of the visualization of steals and leaps. Steals have a yellow arc, while leaps are painted in magenta. At the ends of the arcs, a red circle represents a victim, while the thief is given a green circle.

Spawns

A small white circle represents a spawn. Also, each spawn can be clicked to reveal the name of the task, and how many tasks that worker has spawned before that one. A computation will often have more than one type of task, and to make it easier to differentiate them at a glance, a small dot is added to the center of each spawn, where the color is based on the name of the task. An example of this is shown in Figure 4.3 on the following page.

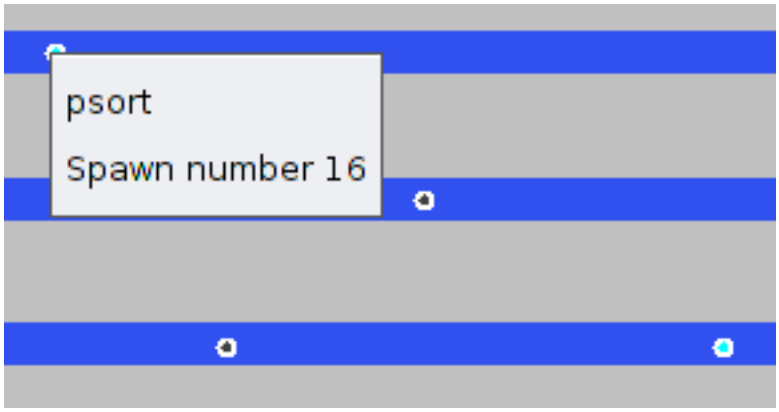

Figure 4.3: Closeup of a few spawns, with an active popup on the psort task

CPU usage

A thin, colored bar underneath each worker bar shows the CPU load. A busy CPU is represented by the color green, and the color changes gradually based on the CPU load. An almost idle CPU is represented by a red bar. In much the same way as the spawns, one can gain an overview of the CPU usage quickly by looking at the color codes, and clicking on it provides all the detail available. This functionality is demonstrated in Figure 4.4.

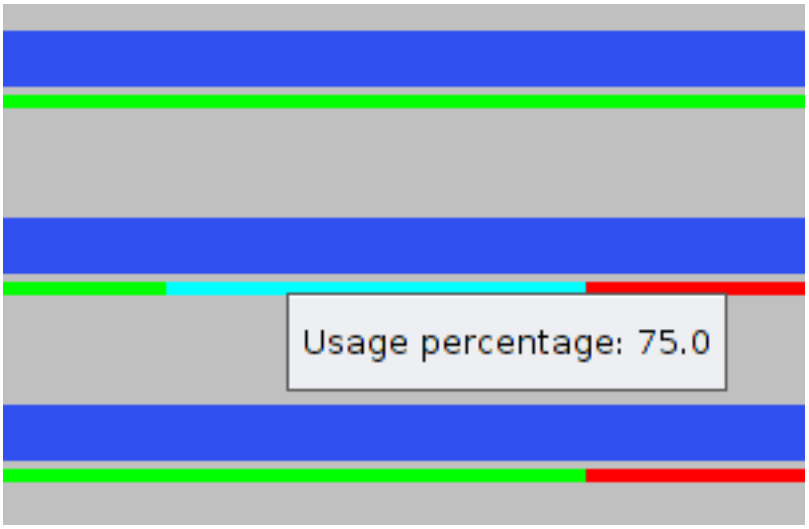

Figure 4.4: A detailed screenshot of CPU usage painting

Critical path

The critical path was added to the project quite late. There were not many possible places in the visualization to incorporate it. The most obvious placement was well suited, however, and a thin, red line will now be placed on top of the worker bar which is part of the critical path. A zoomed view of how the critical path is drawn can be seen in Figure 4.5.

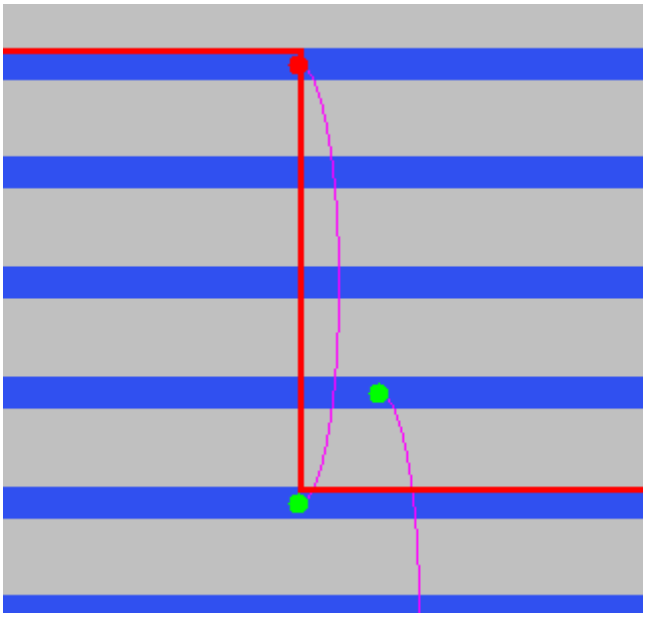

Figure 4.5: A zoomed view of how the critical path is painted

Full GUI

Figure 4.6 on the next page shows what the GUI looks like when viewing a full computation. The image is rotated 90 degrees to fit better on the page. The interface has a tendency to get quite crowded when all the features are switched on. The zoom slider was an important addition to the GUI as it helped a user acquire a detailed view as well as looking at the computation at a glance. Another nice usability feature is the buttons for switching on and off painting of all the features. By switching off some of the features, is it much easier to focus on the interesting information.

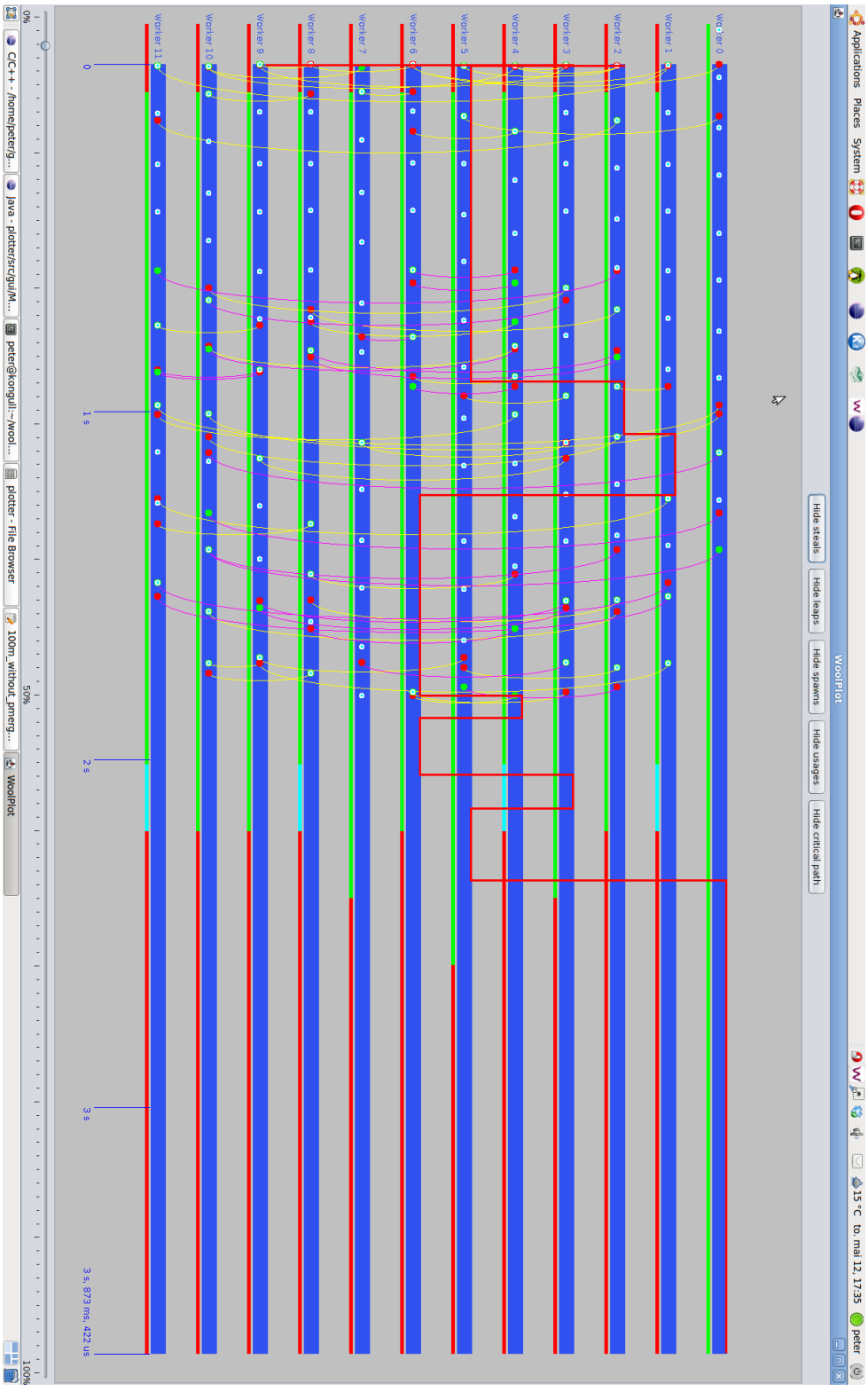

Figure 4.6: An overview of how the GUI looks when viewing an entire computation. Notice the buttons at the top for choosing what elements to include, and the zoom slider at the bottom.

4.2.2 Implementation

Among the many ready-made plotting libraries and programming languages which could have been suitable for displaying the needed information, Java was chosen mostly because of the author's experience with the language. This made it possible to create a working prototype quickly. In addition, Java was considered powerful and versatile enough to avoid limiting what could be done later in the project.

Figure 4.7 on the following page shows a simplified class diagram of the Java program. `StatsReader` reads the text file created by running an application with the instrumented Wool version, and creates the necessary Java objects. There is a `Worker` object for every worker thread. Each `Worker` object also contains list with the objects representing all the steals, leaps, spawns and CPU usage percentages pertaining to that worker thread. When all the data is read and made into objects, the `StatsReader` will pass the objects to a `PlotPanel`, which is the main GUI element.

The `PlotPanel` is a subclass of `JPanel`, and the important method `paintComponent(Graphics g)` is overridden in order to specify exactly how the element is painted. By doing this, the JPanel essentially turns into a canvas, which one can fill with whatever. Inside the method, which in many ways is the most important method in the entire Java application, the list of workers are iterated through, and all the steals, leaps, spawns and usages are painted according to the buttons and the slider elsewhere in the GUI.

Before deciding on this approach, several existing graph libraries were considered. There is a vast array of already written code which would have filled the needs of this project to varying degrees. By choosing to use one of them, however, many of the tasks will be much easier, but the possibilities will often be limited by the design of the chosen library. In the end then, the choice of doing all the painting "by hand" with the `Graphics2D` class was taken in order to keep all choices as open as possible. Exactly what the user interface would look like was still unclear at that point, and by accepting to work a little harder on painting some of the simple elements, as few opportunities as possible were lost.

The `Graphics2D` class provides such convenient methods as `drawArc()`, `fillOval()` and `fillShape()` which are used to visualize the user interface. When using the `fillShape()` method which draws and fills a shape at a certain location, it is also appropriate to use the `contains()` method

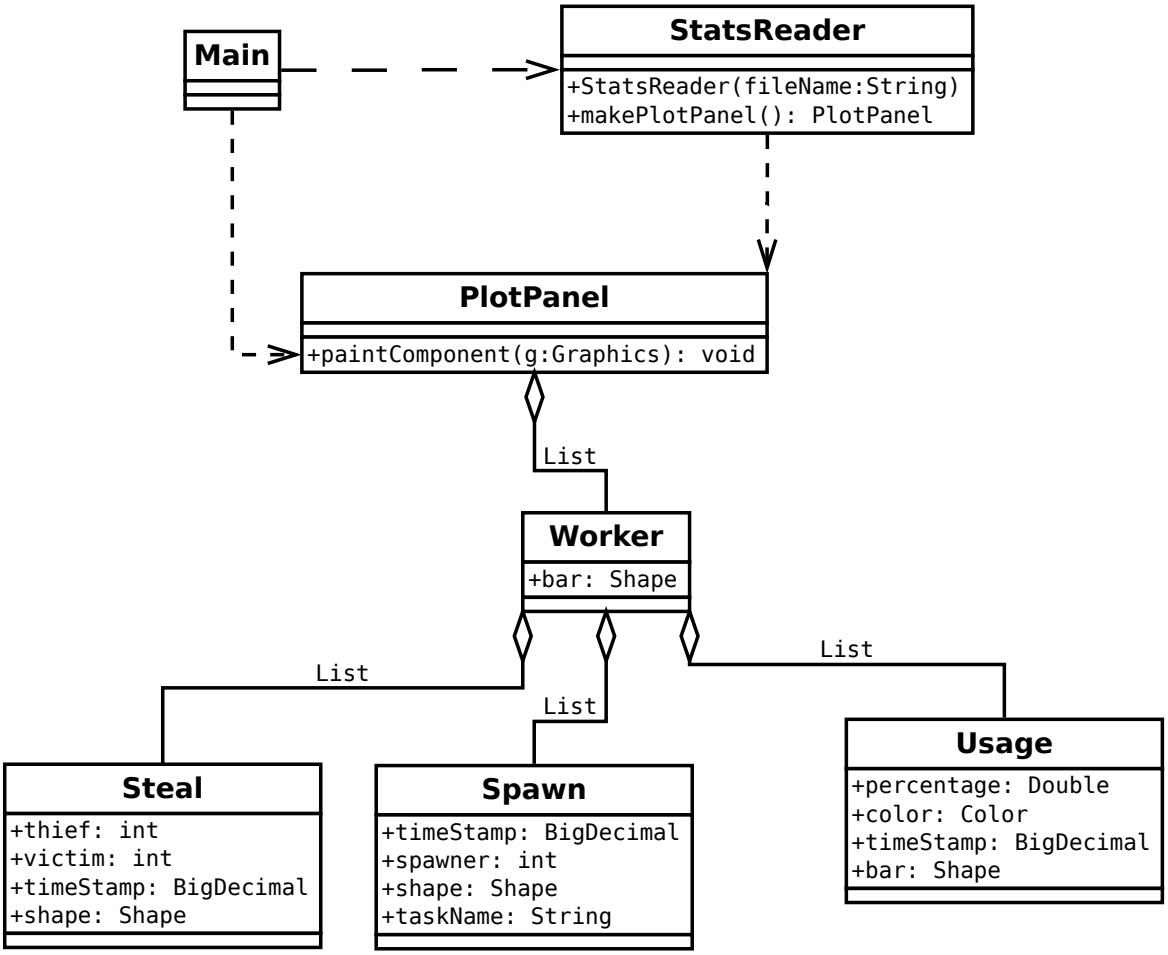

Figure 4.7: A simplified class diagram of the Java application. The `Main` class starts the execution with help from `StatsReader`. The painting of the GUI is done in `PlotPanel`.

in the `Shape` class to provide information when the user clicks on certain shapes.

To briefly show how complex the code for painting even simple elements can be, Figure 4.2 on the next page lists the `paintSteal` method. First, the x-position (in pixels) is calculated using the timestamp of the steal, the total time of the computation and the current width of the `PlotPanel`. Because the timestamps are too large to be represented using `int`, the `BigDecimal` class is used. Unfortunately, this makes the code much bigger. The y-positions for the thief and the victim are calculated in a similar manner as the x, before the distance between the two involved workers are calculated. To avoid all the steal-arcs being painted on top of each other, the distance is then used to decide the radius of the arc, so that a steal between workers further away from each other will result in a wider arc. Lastly, green and red circles are painted at both ends of the arc, to show who was the thief and who was the victim.

The `Main` class is responsible for starting the execution, and also setting up the GUI. It uses the `StatsReader` to create the `PlotCanvas`, and also creates the slider and buttons of the GUI.

Figure 4.8 on page 55 shows how the different components are put together to create the GUI. The `Main` class keeps track of all the GUI elements. That means creating all the buttons and sliders and adding all the GUI elements at the correct place. The `PlotPanel` is too complicated to handle from the `Main` class, so it has its own class.

Figure 4.9 on page 56 is a screenshot of an early version of the GUI. At that point, there was a bug in the collection of leaps, so that there are too many leaps reported. In addition, not all spawns were reported, so there are too few of them in the GUI. Notice also the very rudimentary timeline at the top and the lack of CPU usage and critical path.

4.2.3 Timing

WoolPlot collects timestamps directly before and after the main task is called. The timestamps will thus not include Wool's initial setup, nor the printing of all the profiler data after the execution is complete. It does include, however, the entire main task, with all its setup. In a normal sorting benchmark, for example, the application will have to create the array which is to be sorted. Depending on the speed of the random function, this will often take much time. When profiling a parallel application, this is not the interesting part of the application. A quick

Listing 4.2: The paintSteal method in the PlotPanel class

```
1  private void paintSteal(Graphics2D gd, Steal steal) {
2
3      int x = (int) (startWidth + barLength *
           steal.timeStamp.subtract(startTime).divide(
           endTime.subtract(startTime), 10,
           RoundingMode.HALF_UP).doubleValue());
4
5      int yOfThief = (int) (startHeight + heightIncrement *
           steal.thief + barThickness / 2);
6
7      int yOfVictim = (int) (startHeight + heightIncrement *
           steal.victim + barThickness / 2);
8
9      int distanceWeight = 5;
10
11     //Calculate the distance between the workers
12     int absoluteIndexDistance = Math.abs(steal.thief -
           steal.victim);
13
14     // The distance decides the radius of the arc
15     gd.drawArc(x - absoluteIndexDistance * distanceWeight,
           Math.min(yOfThief, yOfVictim), absoluteIndexDistance
           * distanceWeight * 2, Math.abs(yOfThief - yOfVictim),
           270, 180);
16
17     final int radius = 5;
18
19     //Paint a green circle in the bar of the thief
20     gd.setColor(Color.GREEN);
21     gd.fillOval(x - radius, yOfThief - radius, radius * 2,
           radius * 2);
22
23     //Paint a red circle in the bar of the victim
24     gd.setColor(Color.RED);
25     gd.fillOval(x - radius, yOfVictim - radius, radius * 2,
           radius * 2);
26  }
```

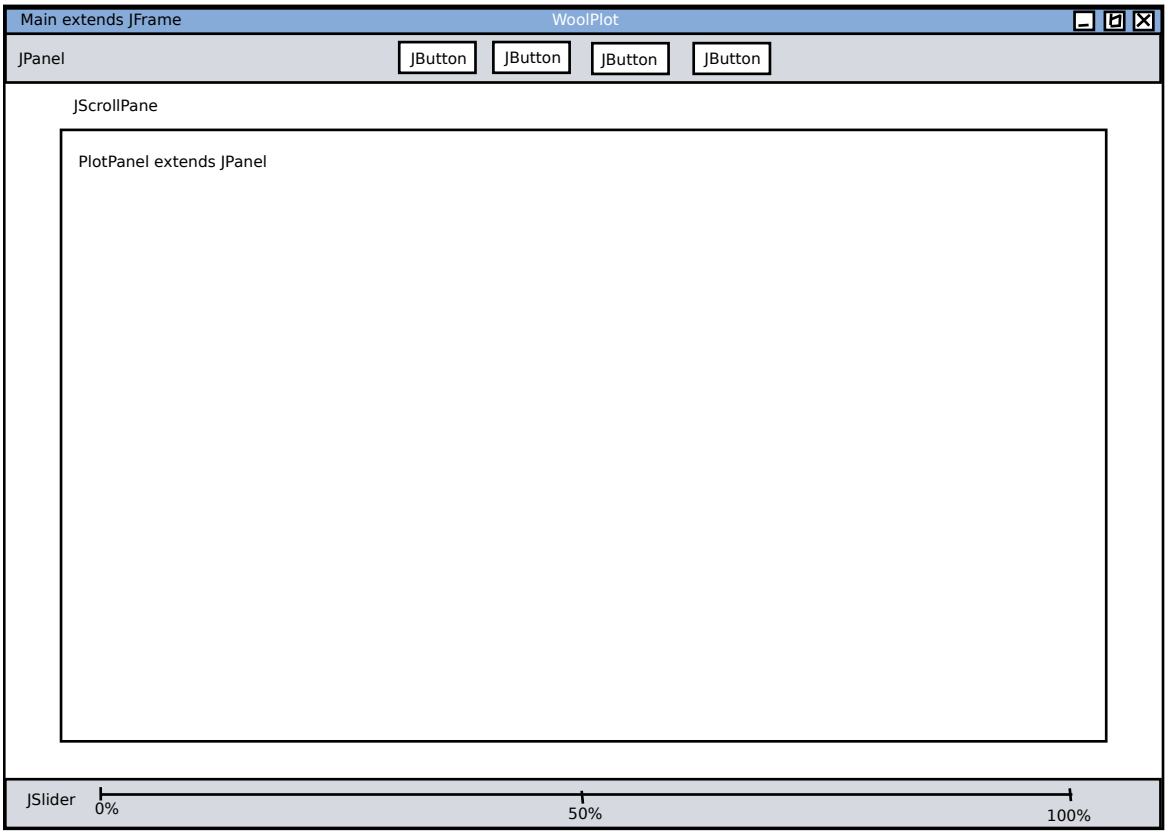

Figure 4.8: Schematic overview of the different GUI components

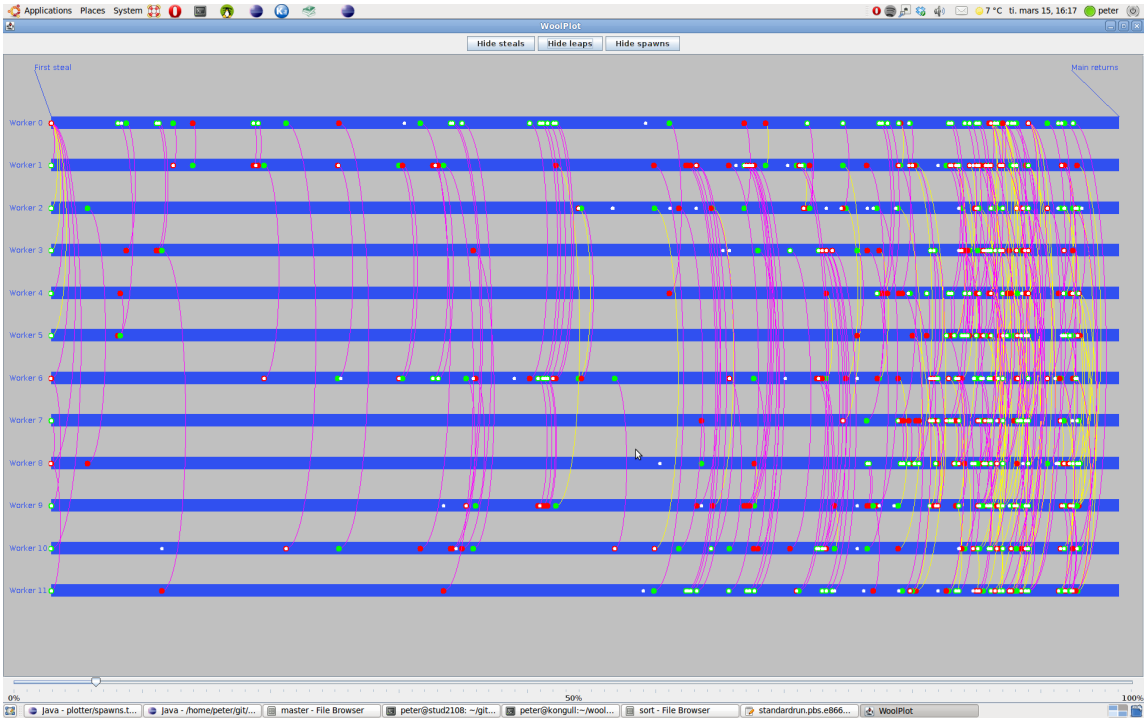

Figure 4.9: Early version of the GUI

fix, which eventually turned out to work quite well, was just finding the first spawn of the execution, and using the timestamp for that as the first timestamp in the run. That way, the visualization is guaranteed to start at the beginning of the parallel part of the run, or at least at the earliest possibility for parallelism. As long as the setup is sequential, this works well. Clearly, if the setup is also done in using tasks, for instance if an array is filled with random numbers by a parallel for loop, this trick will not have any effect. In addition, if there are any sequential work being done after the parallel execution is complete, this will also be included in the profiler. Ideally, there should have been implemented an easy way of selecting just what part of the application one would want to profile.

For calculating the critical path, the timing provided by setting the LOG_EVENTS compiler flag is used. These timestamps represent the time since the application started. In order to paint the critical path correctly, this timing had to be synchronized with the timestamps collected by WoolPlot, which are absolute timestamps. A natural choice was then to use the first spawn of the computation. When the first spawn are found in both the event logs and the spawns reported by WoolPlot, converting

one of the timestamps to the other is trivial.

4.2.4 Critical path

Calculating the critical path of a computation has always been one of the goals of this project. Exactly how it was to be accomplished, however, was long unclear. One way was to implement a version of the Cilkview algorithm. Both the Cilkview algorithm and the concept of critical path is more explained in Section 3.5.7 on page 33. In Cilkview, the theoretical critical path is calculated by executing the entire computation on a single CPU core. There were two main reasons that this path was not chosen for this project. First, precisely how to implement the algorithm were not crystal clear after reading about it in the paper describing Cilkview [36]. Second, choosing this way would mean that one would have to run an application twice to profile it. Once for collecting the data about how the execution actually happened, and once for calculating the theoretical critical path of the computation.

After exchanging some e-mails with the creator of Wool, Karl-Filip Faxén, he shared his ideas on how to calculate the critical path from the logs produced by setting the LOG_EVENTS compiler flag (described in Section 2.3.4 on page 17). His method would not calculate the theoretical critical path of a computation, but rather the critical path of the computation the way it actually was executed. Faxén's proposed way was then chosen not only because it was much clearer how it was to be accomplished, but also because it would fit better with WoolPlot, seeing as it would not require a second, sequential run of the code.

The critical path consists of the events needed to make the last event occur. One can look at it as what the computation is waiting for. In a normal execution, the main thread will at some point have to leapfrog in order to get work. As soon as the main thread has no more work, it no longer contributes directly to finishing the computation. At that point, the critical path has moved to the worker thread which stole from the main task and caused it to start leapfrogging. If that thread runs out of work, the critical path will again have moved to another thread. And so it continues, until the computation is complete.

The procedure for calculating the critical path based on the event logs is based on going backwards through the logs and following rules to decide which events to include. The events are described in Table 2.1 on page 19. All the rules will not be listed here, however, a few examples are

as follows: The last ’2’ from the event logs signals the end of the parallel computation. A ’2’ means that the worker is done with the task it stole and all tasks spawned from that task. The event before the ’2’ is either a ’1’, which is a signal that a thread completed a successful steal, or a ’4’, which means that a thread starts to leapfrog, whichever of the two occurred latest. If it was a ’1’, the event before that should be a ’1xy’ which means that the thread tried to steal from worker ’xy’. Before that, there is two main possibilities for the next event: Either, it is the spawn of the stolen task, that would be a ’5’ in worker ’xy’, or it is the reason that the worker started stealing. If it is leapfrogging, it is a ’3’, since that is the signal for “starting to leapfrog”, or it could be a ’2’. The event which happened latest of these three possibilities is added to the critical path. It’s important to note that if it was the ’5’ in worker ’xy’ which happened latest, the critical path will have moved to worker ’xy’.

Chapter 5

Results

This chapter will show and discuss the results of using WoolPlot on several different benchmarks. The profiler is tested on some real-world applications, as well as some which have been specifically programmed to showcase what WoolPlot can do. All the screenshots in this section are rotated 90 degrees to fit the page better and make it easier to see the details.

5.1 Hardware

The applications in this project have all been run on Kongull, a supercomputer at NTNU. Even though it consists of 98 interconnected nodes, we have only used one node at a time. Each node comprises two six-core AMD Opteron 2.4 GHz 2431 processors (codename Istanbul). The applications have been run with 12 worker threads, which results in one worker thread per CPU core. According to the specifications on AMD's website [42], each of the processors have six 128KB L1 caches, six 512KB L2 caches, and a shared L3 cache which holds 6MB.

5.2 Benchmarks

The benchmarks run in this section comes from the Wool version of Barcelona OpenMP Task Suite, abbreviated BOTS. BOTS was originally aimed at evaluating different OpenMP tasking implementations [43]. Because it aims to test a large set of features, it has since been ported and used to evaluate other task-based programming models. The exact

number of benchmarks may still change, but at least it contains such benchmarks as Fibonacci, FFT, Sorting and NQueens. Originally, we were given a full version of BOTS, written for Wool 0.2. After trying to port WoolPlot unsuccessfully to Wool 0.2, we were given some BOTS programs with main tasks from Karl-Filip Faxén. It is a few of those which are run in this section.

5.2.1 Sorting

BOTS includes an implementation of a sorting algorithm which is dubbed "cilksort" in the source code comments. In essence, it is a parallel merge sort, where the problem is split and sorted recursively with a quicksort which resorts to insertion sort for very small sizes, before it is merged in parallel, also using recursion. Akl and Santoro are credited as the inventors of the algorithm [44]. The algorithm is very similar to the one used in this author's specialization project which was implemented in C++ using TBB [11]. In order to run the benchmark, the user has to specify several parameters. *Merge cutoff* and *qs cutoff* decide the sizes of the merge tasks and the sorting tasks, respectively. *Insertion cutoff* specifies at what size the quicksort should resort to an insertion sort. The last two parameters are the self-explanatory *size* and *reps*.

Figure 5.1 on the next page shows the profiling visualization when sorting 100 million integers with the qs and merge cutoff values set to 75,000 and the insertion sort cutoff set to 20. The visualization is quite messy, and there is not much to draw from it, other than that the run seems quite well balanced, with all the worker threads busy. The first part of the execution, where all the worker threads spawn tasks continually is actually just initialization of the sorting, and should ideally not have been included in the profiler output. The cilksort fills the array to be sorted and also copies it in parallel. This means that it is included in the profiling. This highlights a possible area of improvement for WoolPlot. It should include a way of selecting just what part of an application to profile. The original BOTS sorting application also includes correctness verification of the sorting. That verification has been commented out in this run because there is no need in profiling it.

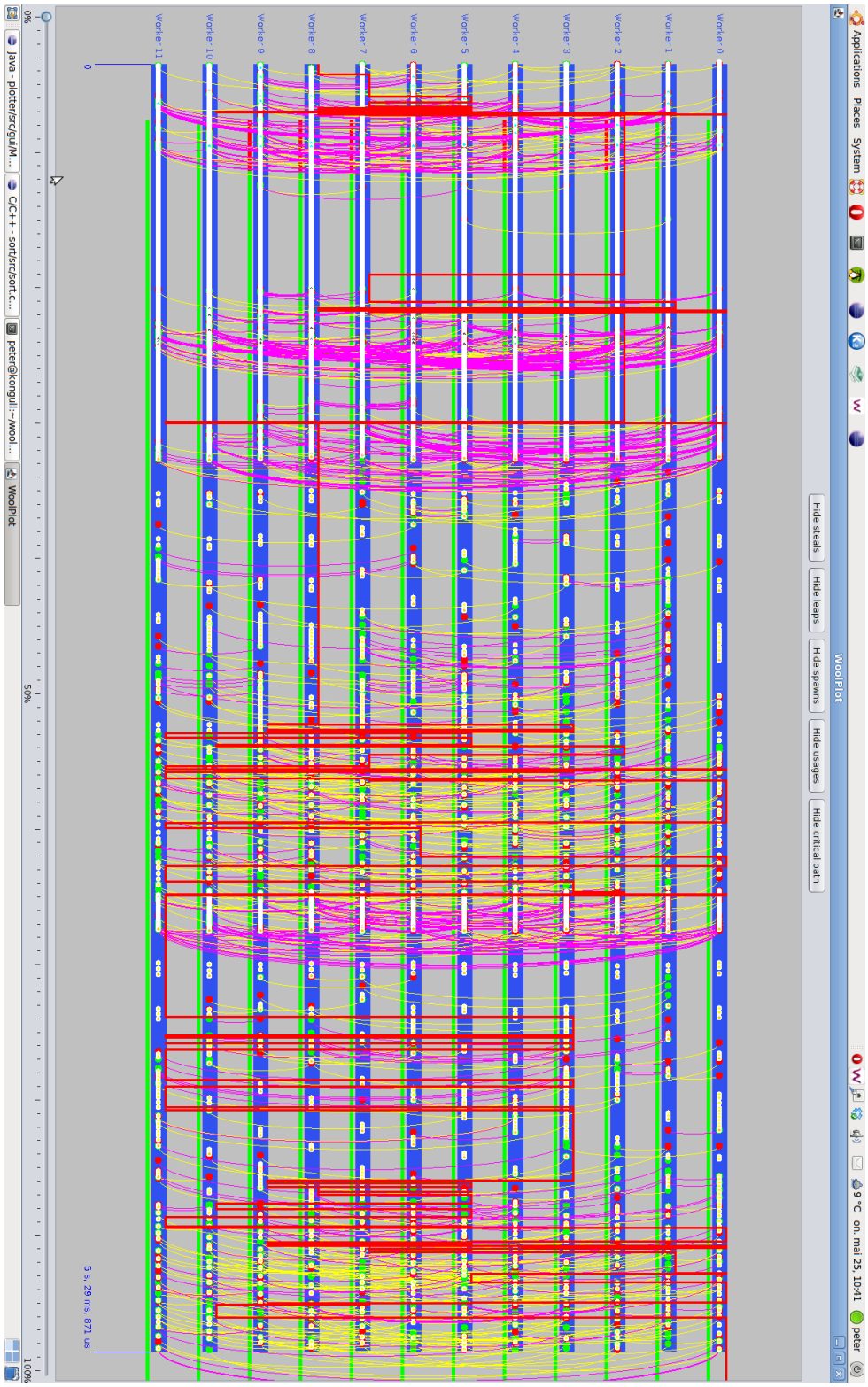

Figure 5.1: Profile result when sorting 100 million integers using the BOTS sort. The execution is well balanced, with all the worker threads busy. There is much communication between the threads, but that is to be expected.

5.2.2 Nqueens

The nqueens benchmark finds all solutions to the nqueens problem of placing n chess queens on a chess board of size $n \times n$ without any of the queens being able to attack each other [45]. For $n = 11$, there are 2,680 solutions, while there are 365,596 for $n = 13$. The benchmark solves the problem recursively, trying all possible positions by brute force. Because trying a position involves spawning a new task, the benchmark spawns an extremely high number of tasks.

As can be seen in Figure 5.2 on the facing page, which shows the result of running nqueens with problem size 11, worker 0 spawns tasks continually from the start to the end of the computation. Worker 0 spawned 308,759 tasks, and there were spawned 1,806,706 tasks in total. To make it easier to see the steals and leaps, Figure 5.3 on page 64 shows the computation without the spawns painted. The critical path never moves, since worker 0 never has to steal to get work. It never waits on another worker thread, and that is why it performs no leapfrogs. It is interesting to note that there are more steals than leaps in the execution. This is slightly unusual, and suggests that the task tree created by the algorithm is wide. A wide task tree means that each task spawns many tasks, but that the continuous strands of tasks are not very long. There is an abundance of small tasks, so that even if a worker manages to steal a task from another worker, it has often completed the task before the victim tries to sync it. The victim will therefore not have to go leapfrogging to get more work.

The run is so short that the CPU usage does not make much sense, it is therefore removed from the screenshots.

5.3 Artificial benchmarks

For real-world, well parallelized benchmarks, WoolPlot outputs will often look similar. Every worker will have work to steal, and they will all be busy most of the time. In order to see some interesting and different outputs, we created some specialized applications designed to highlight different aspects of Wool and WoolPlot.

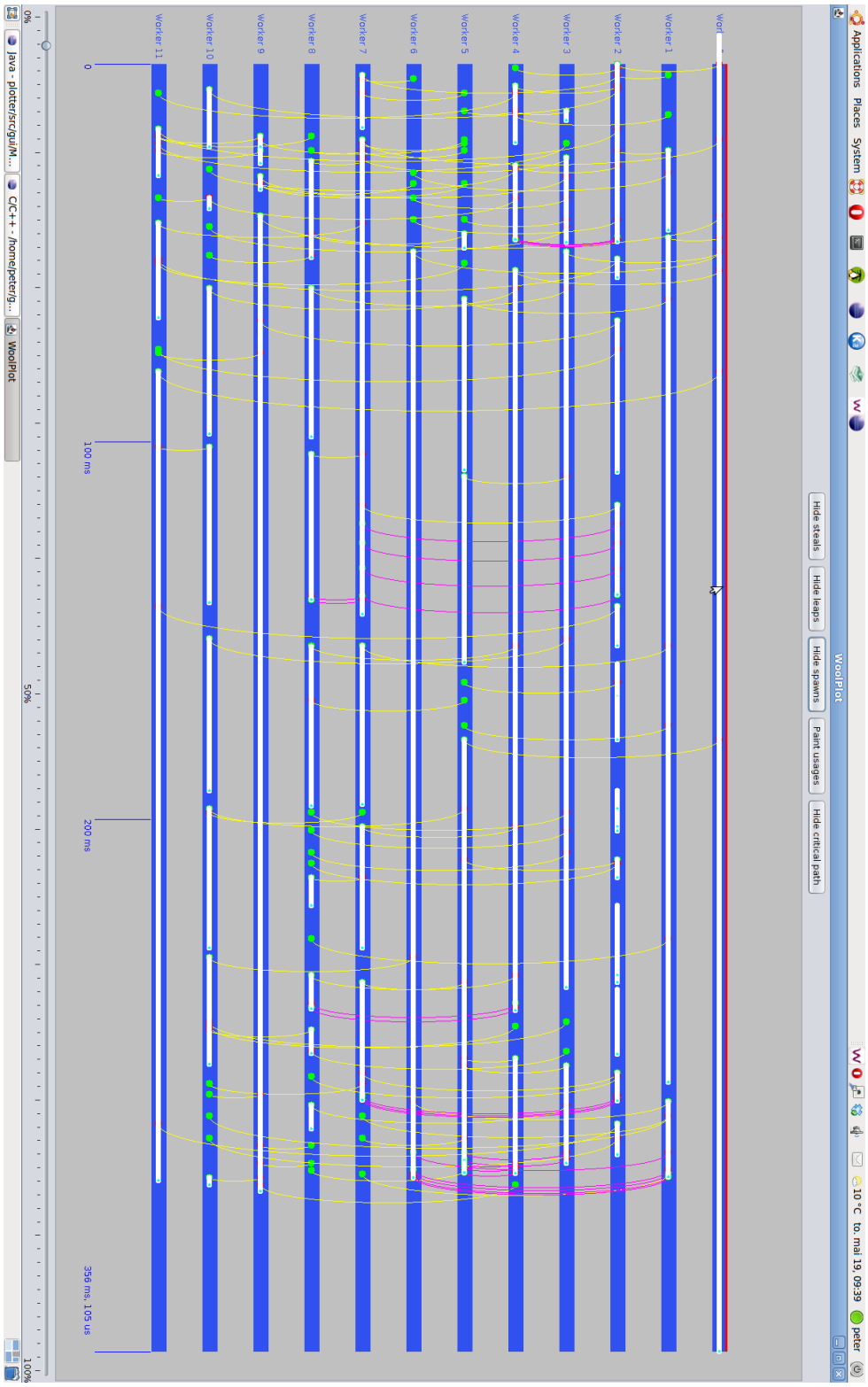

Figure 5.2: Nqueens(11) with spawns painted. This is a spawn-heavy computation, an there is not much communication between the threads. They all spawn many tasks, however, so they are busy nonetheless.

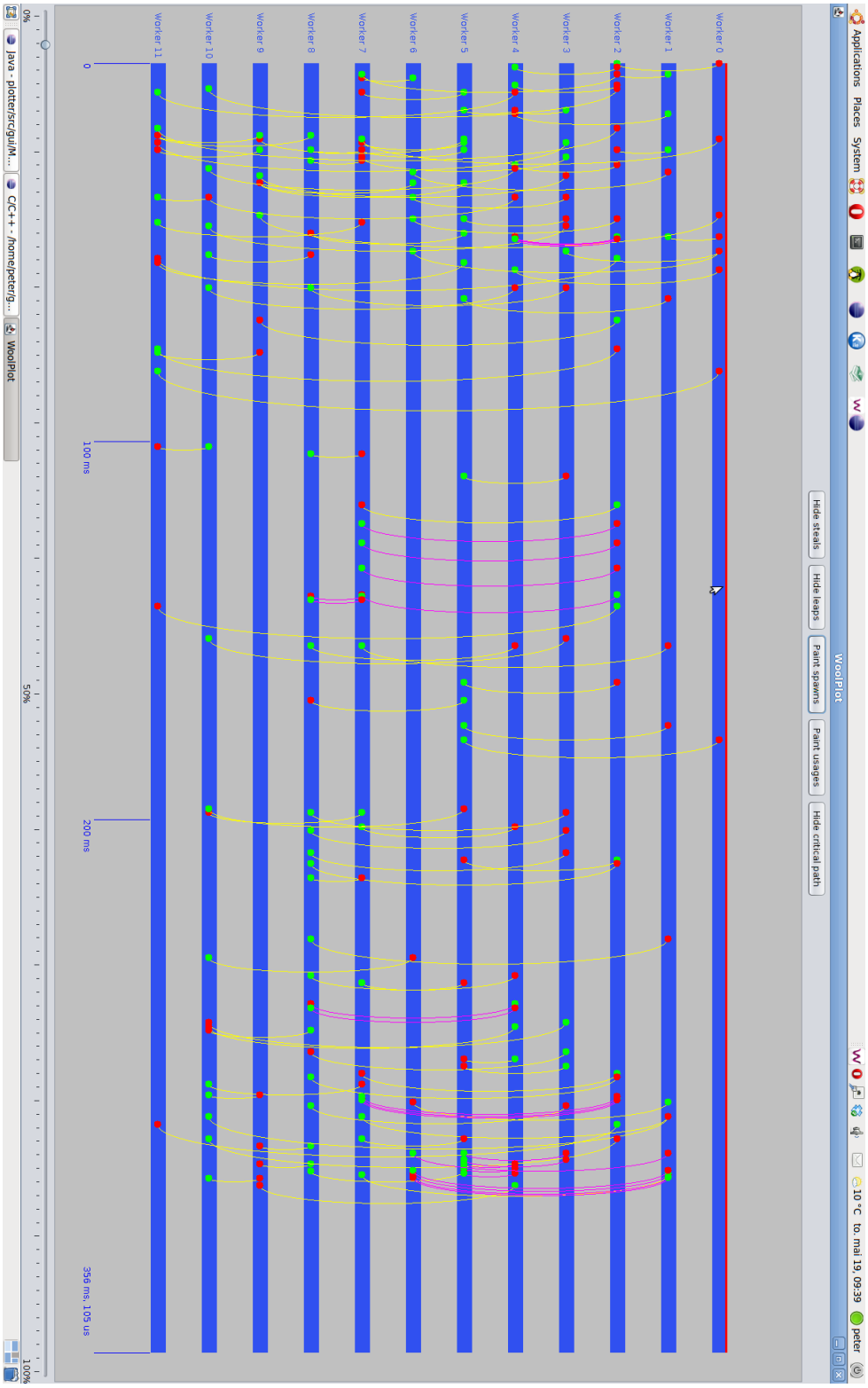

Figure 5.3: Nqueens(11) without spawns painted. There are usually as least as many leaps as steals in an execution. More steals than leaps in this execution suggests a wide, but short task tree.

5.3.1 Unoptimized merge sort

In a parallel merge sort, the numbers to be sorted are first split into many tasks recursively, and then sorted sequentially when they are small enough. At this point, there are many small, individually sorted arrays. A fast merge sort needs to also implement merging in parallel to make the computation sufficiently parallel. If the merge procedure is sequential, it will be too time consuming. To see if the profiler can pick up on this, such a sub-optimal merge sort was implemented and run.

Figure 5.4 on the following page shows the result of sorting 100 million integers with a merge sort with a sequential merge procedure. The sorting tasks are quite big, with sequential cutoff set to 750,000. That explains why there are relatively few tasks in the computation. The colored bars underneath the blue worker bars show the CPU load for each worker. Green represents that the CPU is very busy, and red means that the CPU is nearly idle. As we can see, all the CPU cores are kept busy by the initial sorting. When the time comes for merging, however, the computation is not nearly parallel enough. The very last merge requires a single worker to iterate through the entire array of numbers. The result can clearly be seen in the figure. Worker 3 is occupied for a longer time than the other workers, so we can assume that it takes part in the second to last merge step, where half the array has to be merged. At the very end, however, all the other workers have to wait on worker 0 to finish the computation alone.

5.3.2 Stealable tasks

When programming with tasks, good coding practice is to make the tasks themselves spawn additional tasks. Naturally, it would be interesting to see what would happen if one task spawned all the other tasks. The tasks consists of collecting a timestamp, and then doing some useless work while checking often to see how much time had passed since the useless work began. For these runs, the tasks returned after stalling for 100ms. Worker 0 spawned 100 of these tasks using a for-loop, before it then synced the 100 tasks using another for-loop.

A standard run of the application on Kongull is shown in Figure 5.5 on page 68. There was a total of 9 steals. Every worker except 1 and 8 stole a task, and there were obviously no leaps. The entire computation took slightly over 9 seconds, and worker 0 was the only one where the CPU

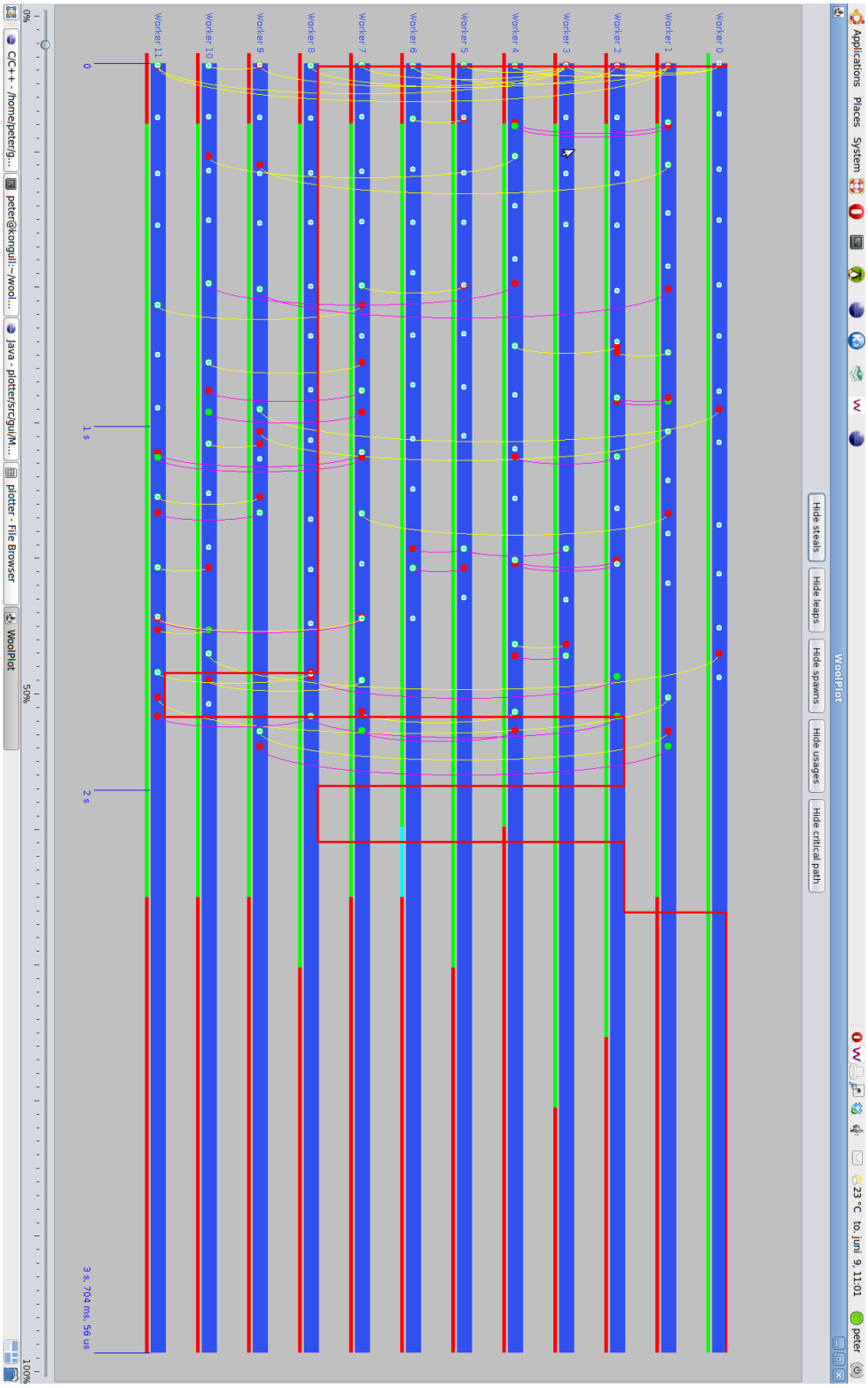

Figure 5.4: Sorting 100 million integers using a merge sort with a sequential merge. Notice how all the workers but worker 0 is idle at the end because worker 0 has to merge the entire array sequentially.

was busy. After some investigation, a setting in Wool which controls the number of stealable tasks was discovered. According to the Wool users guide, the default is $3 + \log_2 n$, where n is the number of workers. When using Kongull, which has 12 workers, this should result in about 6 or 7 stealable tasks at a time. In this case, it turned out to be 9 stealable tasks. The exact reason for this is a bit unclear, but this was not investigated further.

The Wool user guide states that limiting the number of stealable tasks result in "decreasing the overhead in recursive divide-and-conquer applications while potentially leading to loss of parallelism" [17]. For this application, the latter obviously occurred. To see the difference, the exact same compiled code was run once more, now with **-s 100** as an input parameter, to allow 100 stealable tasks per worker. The result is shown in Figure 5.6 on page 69. This time around, the run only takes about 2 seconds to complete, and the CPU cores are busy most of the time. Also, 10 tasks were executed on worker 0, while every other worker stole either 8 or 9 tasks, so the load was actually quite well balanced. Because all the tasks have to be stolen from worker 0, the critical path does not move from worker 0 until the very end of the computation. The last task is stolen by worker 4, and when the main thread tries to sync it, and sees that it is stolen, it tries to leapfrog from worker 4 until worker 4 completes the task. It is also worth noting that it takes some time before all the worker threads have work to do. They try to steal from randomly chosen threads, but there is only worker 0 which has any work. A considerable amount of time has passed since the first spawn before the last thread, worker 8, accomplished a successful steal.

5.3.3 Leapfrogging

In addition to the application where one worker thread spawns all the tasks, it was interesting to profile a very different type of benchmark, where there is a long chain of spawns emanating from one, initial spawn. This was accomplished by creating a task which spawns one single task and then does useless work, we call it waiting for the sake of simplicity, for 100ms before syncing the spawned task. This wait will allow more than enough time for the newly spawned task to be stolen by another worker thread before it is synced. The task has a counter which is decremented for each spawn, allowing control over how many tasks are spawned.

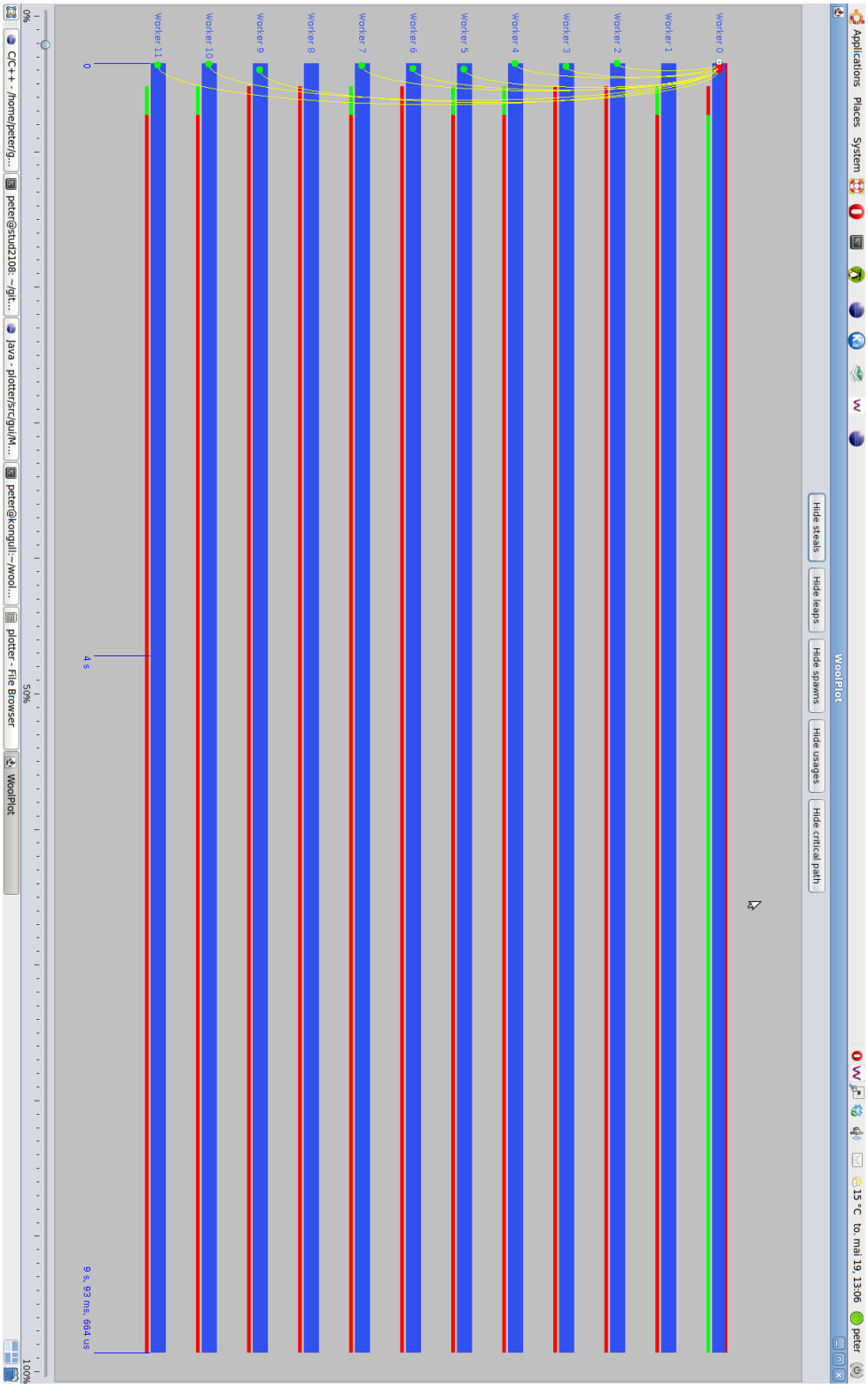

Figure 5.5: Stealable tasks benchmark with the default amount of stealable tasks. Only 9 tasks were stolen, and the run was very poorly parallelized. The critical path never moves from worker 0, since it never waits for another worker.

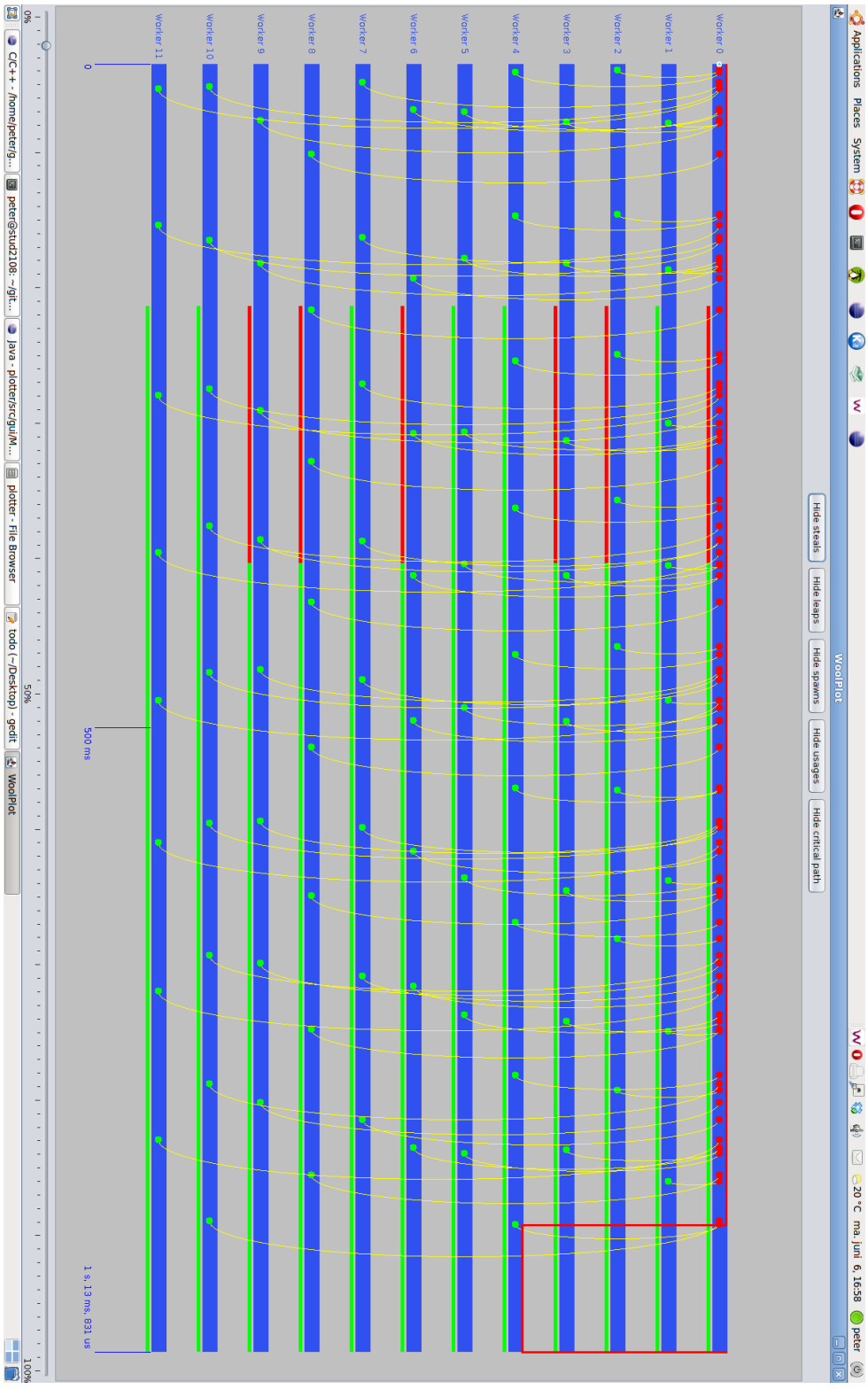

Figure 5.6: Stealable tasks benchmark with 100 stealable tasks. Now, all the tasks could be stolen, and the run is therefore much more parallel. Notice the small dip in the critical path when worker 0 is waiting on worker 4 to complete the final task.

The result when running this application on Kongull is shown in Figure 5.7 on the next page. Because there is only one task available to steal at any one time, it takes a little while before all the worker threads have work to do. They will try to steal from random other worker threads, and also back off a little when they have tried many times without getting a successful steal. When all the worker threads have stolen their first task, spawned a new one and waited for 100ms, they will try to sync the task they spawned. This will then have been stolen by another task, so they will try to leapfrog. Because the task tree actually is just a long line of spawns, which will not end before the very end of the execution, every steal after the first one for each worker will be a leapfrog. In other words, the workers will only try to steal from the worker thread which last stole from them. A result of this can be observed in the screenshot, where worker 1 has nothing to do for long spans of time because it only tries to steal from worker 5. Despite this, the execution is actually rather well parallelized. It could have been even better if the workers could have stolen from any other worker at any other time. Of course, this example is contrived, and leapfrogging is generally a good idea to avoid having code ready to execute and no worker able to execute it.

5.4 Time impact

A profiler will always have an impact on the application being profiled. An important question is how big this impact is. If the profiler changes the execution too much, one should be suspicious as to whether the results of the profiling actually is to be trusted, or if the profiling are too intrusive.

5.4.1 Profiler

To measure the intrusiveness of WoolPlot, all the features were turned on, including the built-in LOG_EVENTS and COUNT_EVENTS, and a sorting application was run 100 times. Then, all the logging features were switched off, before the job was scheduled once more. Timing was done by having a small piece of inline assembly code read the processor's cycle counter directly before and after the first call to the sorting task. To avoid clogging up Kongull with huge amounts of uninteresting log files, the stderr output was directed to /dev/null, in order to destroy it.

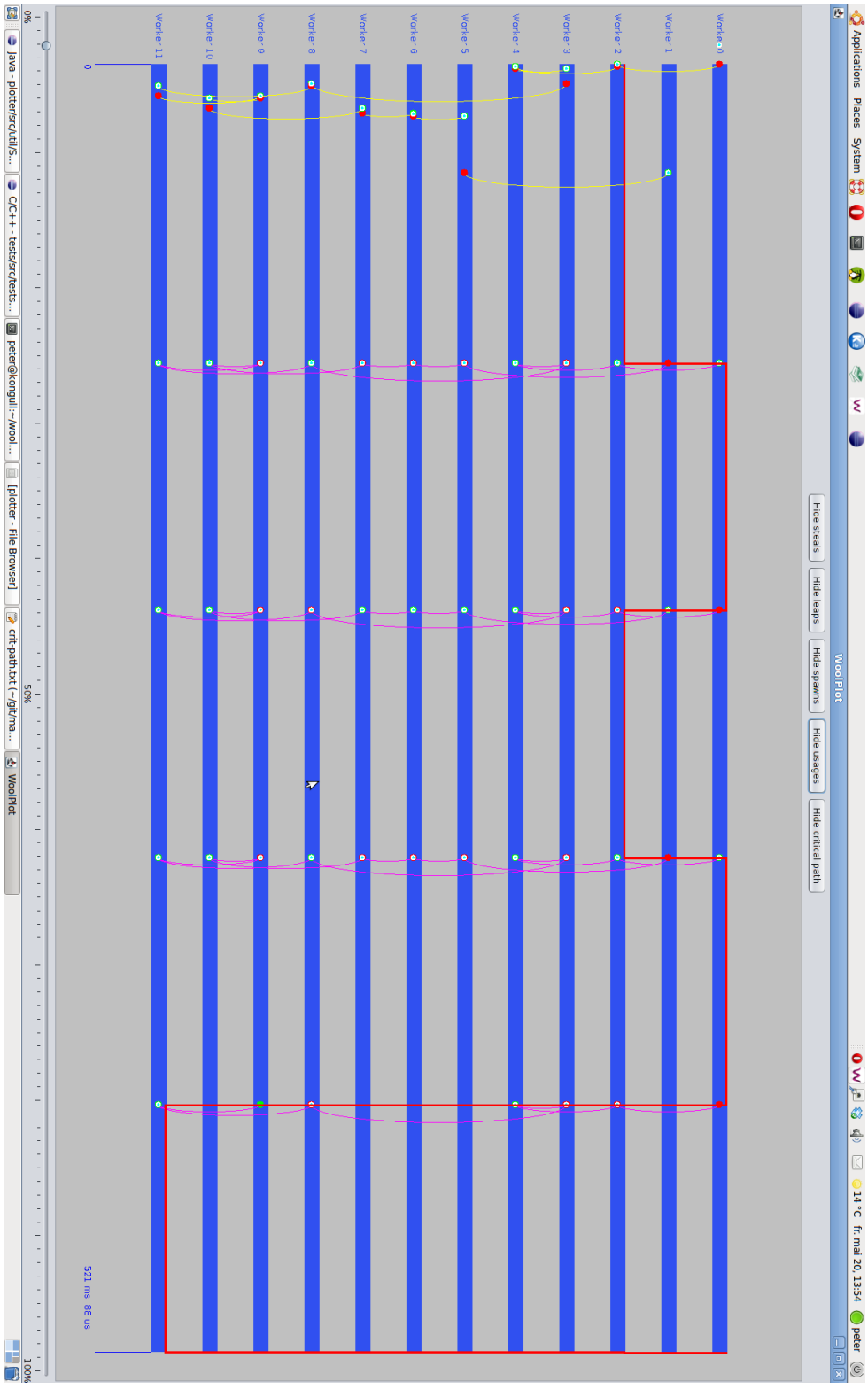

Figure 5.7: An execution of the leapfrogging application with a long strand of tasks in the task tree. Notice how worker 1 is left with nothing to do for long spans of time because it is only allowed to try and steal from worker 5.

The sorting application is a parallel merge sort created especially for this project. It is a Wool port of the merge sort implemented in Intel TBB for the author's specialization project [11]. It splits the array of numbers recursively until the size drops below the sequential sort threshold when they are sorted sequentially using `qsort`. The arrays are then merged in parallel. Because we have written the sorting algorithm ourselves, it is easy to know exactly what it does, and it is easy to time just the sorting, and nothing else.

For WoolPlot, the time it takes to collect data is directly linked to the execution. The profiling is event-based, in that every spawn or successful steal will incur overhead because information about that event will have to be saved. With that in mind, the results should be different when the amount of tasks is varied. To put this theory to the test, the problem size was kept constant, while the cutoff values were significantly reduced, before the procedure was repeated.

Each configuration was scheduled three times, totaling 1,200 runs. The problem size was 100 million, and the serial cutoff was first set to 750,000, which gave a total of 2,048 spawns. The results are summed up in Table 5.1 on the facing page. The serial cutoff was then set to 50,000, resulting in 22,528 reported spawns. Table 5.2 on the next page shows the results. The expected outcome was that WoolPlot would be more intrusive for the run with the smallest task size. The exact opposite was shown to be the case, however. After careful inspection, the reason for this was pinpointed to the built-in event logging. In addition to reporting successful steals and spawns, it also logs each steal attempt. For a run with such big tasks as sorting 750,000 integers, a worker thread which is left without a task to work on, will have time to perform a huge amount of steal attempts. To investigate this theory further, a few runs were performed where the stderr logs were kept. The sizes of the log files confirmed the suspicion. Table 5.3 on the facing page shows the average time impact for the different problem sizes, along with the average size of the outputted text file for five runs with the two different task sizes. The big tasks cause the log file to be more than four times as large as with the smaller tasks, and the performance suffers somewhat because of it. The writing of the log file was not timed, but a larger log file also means that more data was collected, which explains the increased overhead.

In summary, WoolPlot causes about a 5% increase in running time. The fact that the running time is not significantly changed, one can then be confident that the information about the run is fairly accurate. The

Description	1st job	2nd job	3rd job
Cycles w/ profiling	5.4663e+09	5.42726e+09	5.40457e+09
Cycles wo/ profiling	5.1563e+09	5.17208e+09	5.2092e+09
Percentage slower	6.01%	4.93%	3.75%

Table 5.1: Cycle counts and percentages with cutoff values set to 750,000

Description	1st job	2nd job	3rd job
Cycles w/ profiling	5.16896e+09	5.15116e+09	5.10604e+09
Cycles wo/ profiling	5.02098e+09	5.00690e+09	4.90288e+09
Percentage slower	2.95%	2.88%	4.14%

Table 5.2: Cycle counts and percentages with cutoff values set to 50,000

OProfile project's web page talks about the application's low overhead, and reports an overhead of between 1 – 8% [32]. Clearly, this test has only taken into account the overhead incurred by collecting the data. In a real situation, a developer will also have to wait for the data to be written to stderr. Because this is done after the main task returns, however, it will not have an impact on the execution, and it is directly proportional to the amount of data collected. Thus, it has not been relevant to measure.

5.4.2 Spin function

When a thread tries to steal a task several times without succeeding, it will back off a little by entering a function called `spin()`. This contains code designed to keep the thread busy for a little while by doing some useless work. Unfortunately, this will prevent WoolPlot from showing when a thread has no tasks, because a thread will be busy whether it is doing useful work or not. Because of this, we changed the contents of the `spin` function to instead containing a call to `nanosleep` which makes the thread sleep for a microsecond. This made the threads appear idle when

Cutoff value	750,000	50,000
Percentage slower	4.9%	3.32%
Average size of text file	159M	39M

Table 5.3: Aggregate overview of the time it takes to collect data using WoolPlot

they were not working on tasks, but we also wanted to see if this change hurt performance.

All profiling was turned off, and the `time` program [46] was used to time 10 runs of nqueens where $n = 15$ with the normal `spin` function. The call to `nanosleep` was then inserted, and the process was repeated. The results are summarized below:

```
Average time with normal spin
real 3m39.2233s
user 43m49.3377s

Average time with sleeping
real 3m43.9296s
user 9m39.403s
```

The `real` time is the wall time of the computation, that is to say how much time passes between the program is started and finished. The `user` time is the amount of time scheduled on the CPU cores by the program.

Sleeping instead of regular spinning hurts performance slightly. The runs are quite long, with the spinning version using 3 minutes and 39 seconds. The sleeping version uses about 4 seconds more, which amounts to an increased running time of 2.14%. We consider this a price worth paying to get information about the CPU core usage during the run.

On an interesting side note, the amount of user time spent in the function when sleeping instead of spinning is dramatically decreased. When spinning normally, the threads are working almost nonstop. When sleeping, however, the program as a whole runs a little slower, but the total amount of CPU time scheduled by the program is only about 22% of the CPU time scheduled by the spinning version of the program.

Chapter 6

Conclusions and Further Work

6.1 Conclusion

This project has resulted in WoolPlot, a working, off-line Wool profiler. The data collection is mostly measurement-based. The modified Wool source code produces an output text file which contains information about all steals, leaps and spawns in the execution, as well as CPU load. When the text file is used as input to the Java application, the run will be visualized in a graphical user interface. If the built-in logging already present in Wool was used, the Java will application will also calculate the critical path of the computation the way it happened, and paint it in the GUI.

The creation and execution of several benchmark have shown that WoolPlot performs well in practice, especially on the discrete events such as spawns and steals. The CPU load reporting, on the other hand, can be improved upon. These shortcomings will be described in more detail in Section 6.2.3 on the next page. Experiments suggest that WoolPlot incurs between 3 and 6% overhead.

6.2 Further Work

6.2.1 Output format

Keeping the output in the form it is now has the advantage that it can be easily read by a human which can see if it makes sense, and in some cases also gain insights directly from viewing the output. The disadvantage,

however, is that it is slow to both write and read, and it consumes much space. For big runs especially, the output files will sometimes exceed 300MB. If both the C and Java code was re-programmed to instead work with binary files, both some time and space would be saved.

6.2.2 Profiling Specific Sections

It is usually not interesting to profile the set-up and clean-up of an execution. WoolPlot will now automatically exclude showing what happened in the run before the first spawn. This helps a little, but there should ideally be a way to choose exactly what to profile. Either by starting and stopping the profiling with function calls directly in the code, or at least having an easier way to zoom in on the visualization in the GUI, so that it is trivial to select just what part of the computation to view.

6.2.3 Hardware Counters

The decision to use the /proc/stat filesystem for acquiring CPU load data was, in retrospect, not a wise one. The data is not accurate enough to provide any more than an idea of whether the cores are busy or not. In addition, because of how often the stat-file is updated, the polling interval has to be quite large. This means that the CPU usage information is practically useless for short, tight runs.

Implementing use of hardware performance counters would provide WoolPlot with much more accurate data, which could provide valuable insights into where the time is spent. Also, there is a library which could make the implementation quite a lot easier. PAPI aims to "specify a standard application programming interface (API) for accessing hardware performance counters available on most modern microprocessors" [47]. The project is still very much alive today, the latest version, 4.1.3, was released May 2011 [48].

6.2.4 Other ideas

To provide even better information about an execution, WoolPlot could include more specific data. An idea is to visualize the size of the task queue for each worker over time. That way, one could easily see the effect of steals on the task queue size, and also how many tasks there are

available at any time in the computation. Clearly, the task queue is empty when a worker tries to steal a new task. When a thread goes leapfrogging, on the other hand, there is no way of knowing how many tasks it has in its task queue.

Another idea is to somehow show the steal attempts, as well as the successful steals. That would show whether a thread is backing off or actively looking for work, and combined with information about the task queues, it could be very interesting. Careful consideration has to be taken when deciding how to show this, because there is usually many more attempts than successful steals, and the visualization is already a bit crowded.

References

[1] S. Fuller and L. Millett, "Computing performance: Game over or next level?," *Computer*, vol. 44, no. 1, pp. 31–38, 2011.

[2] B. Wilkinson and M. Allen, *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1998.

[3] LG Corporation, "Lg optimus 2x p990 product page." `http://www.lg.com/uk/mobile-phones/all-lg-phones/LG-android-mobile-phone-P990.jsp`. Retrieved 30. May 2011.

[4] Tilera, "Tile-Gx Processor Family." `http://tilera.com/products/processors/TILE-Gx_Family`. Retrieved 24. March 2011.

[5] Intel, "Intel threading building blocks tutorial." `http://www.threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Tutorial.pdf`. Retrieved 1. November 2010.

[6] M. Hill and M. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[7] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas, "Single-isa heterogeneous multi-core architectures for multithreaded workload performance," in *ACM SIGARCH Computer Architecture News*, vol. 32, p. 64, IEEE Computer Society, 2004.

[8] N. Lakshminarayana, S. Rao, and H. Kim, "Asymmetry aware scheduling algorithms for asymmetric multiprocessors," in *Proc. of the Fourth Annual Workshop on the Interaction between Operating Systems and Computer Architecture*, 2008.

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

[10] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, pp. 720–748, September 1999.

[11] P. Hemmen, "Task-based Programming on a 64-core Tilera CPU." TDT4590 Complex Computer systems, Specialization Project NTNU, December 2010.

[12] Intel, "Intel cilk plus website." `http://software.intel.com/en-us/articles/intel-cilk-plus/`. Retrieved 2. December 2010.

[13] "OpenMP Application Program Interface, Version 3.0 May 2008." Online: `http://www.openmp.org/mp-documents/spec30.pdf`(cited 15. November 2010).

[14] J. M. Pérez, R. M. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan*, pp. 142–151, IEEE, 2008.

[15] K.-F. Faxén, "Wool-a work stealing library," *SIGARCH Comput. Archit. News*, vol. 36, pp. 93–100, June 2009.

[16] A. Podobas, M. Brorsson, and K.-F. Faxén, "A comparison of some recent task-based parallel programming models," *3rd Workshop on Programmability Issues for Multi-Core Computers , Pisa, Italy.*, 2010.

[17] K.-F. Faxén, "Wool 0.1 users guide." `http://www.sics.se/~kff/wool/users-guide.pdf`, June 2009. Retrieved 9. November 2011.

[18] "Wool home website." `http://www.sics.se/~kff/wool/`. Retrieved 23. March 2011.

[19] D. Wagner and B. Calder, "Leapfrogging: A portable technique for implementing efficient futures," in *In SIGPLAN Notices*, pp. 208–217, 1993.

[20] A. Iordan, A. Podobas, L. Natvig, and M. Brorsson, "Investigating the Potential of Energy-savings Using a Fine-grained Task Based Programming Model on Multi-cores," *A4MMC 2011 : 2nd Workshop on Applications for Multi and Many Core Processors*, 2011.

[21] S. Graham, P. Kessler, and M. Mckusick, "Gprof: A call graph execution profiler," *ACM Sigplan Notices*, vol. 17, no. 6, pp. 120–126, 1982.

[22] F. E. Allen, "Control flow analysis," in *Proceedings of a symposium on Compiler optimization*, (New York, NY, USA), pp. 1–19, ACM, 1970.

[23] T. Ball and J. Larus, "Optimally profiling and tracing programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 4, pp. 1319–1360, 1994.

[24] G. Ammons, T. Ball, and J. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," *ACM Sigplan Notices*, vol. 32, no. 5, pp. 85–96, 1997.

[25] AMD, "Amd codeanalyst performance analyzer product page." `http://developer.amd.com/cpu/codeanalyst/Pages/default.aspx`. Retrieved 22. March 2011.

[26] Intel, "Intel vtune amplifier xe 2011 product page." `http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/`. Retrieved 22. March 2011.

[27] A. Srivastava and A. Eustace, "Atom: A system for building customized program analysis tools," in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pp. 196–205, ACM, 1994.

[28] J. Seyster, "Techniques for visualizing software execution," 2008. A Research Proficiency Exam, Stony Brook University.

[29] J. Fenlason and R. Stallman, "Gprof project website." `http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html`. Retrieved 2. June 2011.

[30] "Google CPU Profiler." `http://goog-perftools.sourceforge.net/doc/cpu_profiler.html`. Retrieved 21. March 2011.

[31] W. Cohen, "Tuning programs with OProfile," *Wide Open Magazine*, vol. 1, pp. 53–62, 2004.

[32] "Oprofile." http://oprofile.sourceforge.net/. Retrieved 21. March 2011.

[33] Intel, "Intel parallel studio product page." http://software.intel.com/en-us/articles/intel-parallel-studio-home/. Retrieved 22. March 2011.

[34] Intel, "Interpreting locks and waits analysis results." http://software.intel.com/sites/products/documentation/hpc/amplifierxe/en-us/lin/ug_docs/olh/common/interpreting_locks_waits_window.html. Retrieved 2. June 2011.

[35] K. Fürlinger and D. Skinner, "Performance profiling for openmp tasks," *Evolving OpenMP in an Age of Extreme Parallelism*, pp. 132–139, 2009.

[36] Y. He, C. Leiserson, and W. Leiserson, "The Cilkview Scalability Analyzer," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pp. 145–156, ACM, 2010.

[37] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd ed., 2009.

[38] K. W. Cameron and Y. Luo, "Performance Evaluation Using Hardware Performance Counters." http://people.cs.vt.edu/~cameron/prof/isca99/. Retrieved 30. May 2011.

[39] "Proc man-page." http://www.kernel.org/doc/man-pages/online/pages/man5/proc.5.html. Retrieved 9. May 2011.

[40] "Fork man-page." http://www.kernel.org/doc/man-pages/online/pages/man2/fork.2.html. Retrieved 21. May 2011.

[41] Free Software Foundation Inc, "The C Preprocessor." http://gcc.gnu.org/onlinedocs/cpp/index.html. Retrieved 25. March 2011.

[42] AMD, "Amd opteron processor solutions." http://products.amd.com/en-ca/OpteronCPUDetail.aspx?id=552&f1=Six-Core+AMD+Opteron%e2%84%a2&f2=&f3=Yes&f4=&f5=512&f6=Socket+F+(1207)&f7=&f8=45nm+SOI&f9=&f10=4800&f11=6&. Retrieved 9. June 2011.

[43] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona openmp tasks suite: a set of benchmarks targeting the exploitation of task parallelism in openmp," in *Parallel Processing, 2009. ICPP'09. International Conference on*, pp. 124–131, IEEE, 2009.

[44] S. Akl and N. Santoro, "Optimal parallel merging and sorting without memory conflicts," *Computers, IEEE Transactions on*, vol. 100, no. 11, pp. 1367–1369, 1987.

[45] R. Bruen *et al.*, "The n-queens problem," *Discrete Mathematics*, vol. 12, no. 4, pp. 393–395, 1975.

[46] "Time man.page." `http://www.kernel.org/doc/man-pages/online/pages/man1/time.1.html`. Retrieved 10. June 2011.

[47] S. Browne, C. Deane, G. Ho, and P. Mucci, "Papi: A portable interface to hardware performance counters," in *Proceedings of Department of Defense HPCMP Users Group Conference*, 1999.

[48] The PAPI Project Team, "Papi project website." `http://icl.cs.utk.edu/papi/index.html`. Retrieved 8. June 2011.

Appendix A

Detailed documentation

The code for WoolPlot is made available together with this thesis, and our hope is that it has been documented well enough in the course of the report. The only documentation in this appendix is the data format for the output produced by the modified Wool source code.

A.1 Data format

The first output the Java application reads is the number of worker threads, followed by two timestamps, which the C application collects at very start and end of the execution. Next in line is every steal, every leap, every spawn, the event logs produced by activating the LOG_EVENTS macro and the CPU usages over time. The formats for all the data are given below.

```
---BEGIN STARTENDTIMES---
[number of workers]
[start timestamp]
[end timestamp]
//begin steals
[[thief]:[victim]:[timestamp]]
.
.
.
[[thief]:[victim]:[timestamp]]
//begin leaps
<leaps>
```

```
[[thief]:[victim]:[timestamp]]
.
.
.
[[thief]:[victim]:[timestamp]]
//begin spawns
<spawns>
[[spawner]:[timestamp]:[task name]]
.
.
.
[[spawner]:[timestamp]:[task name]]
//output from COUNT_EVENTS

//begin output from LOG_EVENTS
//some information about clock synchronization
<event_logs>
EVENT [worker] [type] [timestamp]
.
.
.
EVENT [worker] [type] [timestamp]
//begin cpu usages
<cpu_usage>
[[worker thread]:[timestamp]:[usage percentage]]
.
.
.
[[worker thread]:[timestamp]:[usage percentage]]
```