Hybrid Threads for the Parrot Virtual Machine

Stefan Seifert

BACHELORARBEIT

Nr. 1010307037-A

eingereicht am
Fachhochschul-Bachelorstudiengang

Software Engineering

in Hagenberg

im July 2012

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, July 10, 2012

Stefan Seifert

Contents

Acknowledgements

I want to thank:

- Andrew "whiteknight" Whitworth for laying the foundations for my work, being of tremendous help at all stages of the project and for fixing some of my bugs.
- Nat "Chandon" Tuck for implementing the green thread basics on which my work is based on.
- Christoph "cotto" Otto for bringing me into the Parrot project.
- Brian "benabik" Gernhardt for lurking in the chat channel and at least trying to help with my many questions.
- Markus "zimmski" Zimmerman for correcting many errors.

Abstract

Parrot is a runtime system for dynamically typed programming languages. Despite several attempts in it's 10 years of history, it does not provide any support for multithreaded computation.

This thesis shows a way to implement threading support in Parrot using a hybrid approach using a combination of lightweight ("green") threads and OS threads. These lightweight threads are used as messages in a system where reading shared variables is allowed but only the one owner thread may write to it. The implementation of this hybrid approach is described in detail and benchmarks are presented demonstrating the viability of this design.

Kurzfassung

Parrot ist eine Laufzeitumgebung für dynamisch typisierte Programmiersprachen. Trotz mehrerer Anläufe in seiner zehnjährigen Geschichte, bietet Parrot keine Unterstützung für Multithreading.

Diese Arbeit zeigt einen Weg, um Threadingunterstützung in der Laufzeitumgebung zu implementieren mit Hilfe eines hybriden Systems aus leichtgewichtigen und Betriebssystem-Threads. Diese leichtgewichtigen Threads werden als Nachrichten in einem System benutzt, in dem das Lesen von gemeinsamen Daten erlaubt ist, aber nur der eine besitzende Thread darauf schreiben darf. Die Implementierung wird detailiert beschrieben und mit Benchmarks die Brauchbarkeit des Designs gezeigt.

Chapter 1

Introduction and motivation

On July 19th 2000, the Perl 6 design process was announced. Perl 5 had been a flexible and widely used programming language but had started to show its age and suffered from early design decisions.

Am example of Perl:

```
1 use common::sense;
2
3 my @friends = qw(Ann Bob);
4 say "Hello $_" foreach @friends;
```

The Perl interpreter is written in C and has accumulated a lot of cruft over the years. The general consensus among the core developers was that the code had reached a state where maintenance was approaching impossibility [11]. An attempt to reimplement these internals had failed but led to the decision that the interpreter for a Perl 6 language should be developed independently of the needs of Perl 5. Since the Perl 6 syntax was very much in flux (and parts of it still are) the designers of these new internals tried to work very independently of any syntax related questions [12]. Taking up the name born in an April Fool's Day joke announcing the merging of the Perl and Python programming languages, these new internals were called Parrot [14].

Parrot evolved from being just the interpreter for the new version of Perl to being a language independent VM providing features like garbage collection, exception handling and dynamic typing. At the time when the Parrot project started, the Java and .NET VM were widely used, but both targeted statically typed languages. Parrot thus filled a quickly growing niche.

The Perl 6 design process began with asking the Perl users what they were expecting from the new version of the language. The very first feature that got asked for was well integrated multithreading support [15]. Perl 5 had two different implementations of thread support. In the first model, called *5005threads*, data was shared by default and shared access to data had to be explicitly synchronized. This was similar to the models used by languages such as C or Java. The implementation however suffered from

data corruption and crashes and thus was not recommended for production use [8]. Perl 5.6 introduced the newer model called *ithreads*, mostly as a way to emulate *fork* on Win32 platforms. Perl 5.8 exposed this Application Programming Interface (API) to the user of the programming language. In this new model, all data is copied to each thread and afterwards thread local. Data must be explicitly shared between threads.

In other words, in Perl threads are not lightweight at all. They have severe impact on memory usage, writes to shared variables are expensive and still not all features of the language are usable in threaded programs.

Being born at a time when Perl 6 still looked much more similar to Perl 5 than it does nowadays, Parrot's threading support initially was very close to Perl's *ithreads* model. Previous attempts to change this into the more conventional model of data shared by default or implementing new technologies like *Software Transactional Memory* failed. For example Parrot has never supported running multiple threads and having garbage collection at the same time.

1.1 Why is multithreading support so important?

In the year 2005 development of faster Central Processing Units (CPUs) shifted from increased speed of a single core to adding more cores. Modern processors contain up to 12 cores with even mobile phones having up to four. To utilize a modern CPU's power, code needs to be run in parallel. In UNIX (and thus Perl) tradition, this is accomplished using multiple processes being a good solution for many use cases. For many others like auto threading of hyper operators in Perl 6, the cost of process setup and communication would be prohibitively high except for very large data sets.

1.2 Why is multithreading support so difficult to implement?

Low level programming languages like C provide only the bare necessities, leaving the responsibility for preventing data corruption and synchronization entirely to the user. A high-level language like Perl 6 on the other hand provides complex and compound data types, handles garbage collection and a very dynamic object system. Even seemingly simple things like a method call can become very complex. In a statically typed programming language the definition of a class is immutable. Thus, calling a method on an object contains just the steps of determining the object's class, fetching the required method from this class and calling it. Calling the same method again can then even omit the first two steps since their results cannot change.

In a dynamic language, the object may change its class at runtime. The inheritance hierarchy of the class may be changed by adding or removing

parent classes. Methods may be added to or removed from classes (or objects) at runtime and even the way how to find a method of a class may change. So a simple method call results in the following steps:

- determining the class of the object,
- determining the method resolution method of the class,
- finding the actual method to call,
- calling the method.

These steps have to be repeated for every method call, because the results may change any time. In a threaded environment, a thread running in parallel may change the underlying data and meta data in between those sequences and even between those steps. As a consequence, this meta data has to be protected from corruption introducing the need for locks in a performance critical area.

Many interpreters for dynamic languages like Python [4] or Ruby [1] handle this problem by using a global interpreter lock to effectively serialize all operations. This is a proven and reliable way but leaves much of the hardware's potential unused.

1.3 Current status

During the years of back and forth and failed attempts of adding threading support to Parrot, the Perl 6 specification evolved to a point where the largest parts of the language were covered and its features were implemented in the compilers. The lack of concurrency primitives in Parrot however prevents any progress in the area of concurrency support.

Before the work on this thesis started, Parrot did not have any threading support at all. The previous, defunct implementation had been removed.

This thesis suggests a new approach based on a hybrid threading system. So called *green threads* are used to simplify the implementation of a nearly lock free multithreading implementation. This approach is based on a design by Andrew Whitworth and Nat Tuck [18]. The goal of this thesis is to demonstrate the advantages of this model and produce a working implementation which can be used to investigate the performance characteristics of a hybrid threading system.

Chapter 2

Concurrency in other programming platforms

This chapter is about programming platforms. A platform is seen as a combination of a programming language and a runtime. E.g. for the Python programming language there are multiple runtimes with different implementations of threading support.

2.1 Java

In Java, the user is responsible for preventing concurrency issues. The language provides synchronization primitives like mutexes, but the interpreter (the Java Virtual Machine, JVM) does not protect the consistency of the provided data structures. The class library provides the user with high-level data structures explicitly designed for multithreaded scenarios.

Java version 1.1 used *green threads* to support multithreaded execution of Java programs. Green threads are threads simulated by the virtual machine (VM) but unable to use more than one CPU core for processing. Details are described in chapter 4. Version 1.2 introduced native Operating System (OS) threading support which since has become the standard way to do multithreading in Java [10].

Following is an example of a multithreaded program in Java. The program spawns 10,000 threads, each one waiting for the *starter* variable to be set to 1 before adding it's name to an array:

```
1 import java.util.Vector;
2 import java.util.ArrayList;
3
4 class ThreadingExample {
5     private static Vector<Integer> results = new Vector<Integer>();
6     private static int starter = 0;
7
8     private class Sayer extends Thread {
```

```
 9         private int name;
10
11         public Sayer(int name) {
12             this.name = name;
13         }
14
15         public void run() {
16             while (starter == 0) {
17                 try { Thread.sleep(100); } // milliseconds
18                 catch(InterruptedException e) { }
19             }
20
21             results.add(name);
22         }
23     }
24
25     public static void main(String args[]) {
26         ThreadingExample e = new ThreadingExample();
27         e.test();
28     }
29
30     private void test() {
31         ArrayList<Thread> threads = new ArrayList<Thread>();
32
33         for (int i = 0; i < 10000; i++) {
34             Thread t = new Sayer(i);
35             t.start();
36             threads.add(t);
37         }
38
39         starter = 1;
40
41         for(Thread t: threads)
42             try { t.join(); }
43             catch(InterruptedException e) { }
44
45         System.out.println(results.size());
46     }
47 }
```

The program exploits the fact that all methods in the *Vector* class are synchronized, i.e. thread-safe. If the *results* array was a non-thread-safe *ArrayList* instead, access would have to be locked manually:

```
1 synchronize(results) {
2     results.add(name);
3 }
```

2.2 Python

Python provides threading support through the *threading* module.

The CPython implementation of the Python runtime uses a *Global Interpreter Lock* (GIL) to protect its internal consistency [9]. This is a single lock taken whenever the interpreter executes Python bytecode. Because of this lock, only one thread can execute bytecode at any time so all built-in types and the object model are implicitly type safe. The drawback is that Python code cannot benefit from having multiple CPU cores available. However I/O operations and calls to external libraries are executed without holding the GIL, so in applications with multiple I/O bounded threads, there may still be a performance benefit from using multithreading.

To run Python code in parallel, multiple processes have to be used. The *multiprocessing* module provides support for spawning processes exposed through an API similar to the *threading* module [5]. Since processes may not directly access other processes' memory, the *multiprocessing* module provides several means of communication between processes: *Queues*, *Pipes* and shared memory support.

Following is an example of a multithreaded program in Python. The program spawns 10,000 threads, each one waiting for the *starter* variable to be set to 1 before appending it's name to an array:

```
 1 from threading import Thread
 2 from time import sleep
 3
 4 starter = 0
 5 results = []
 6 threads = []
 7
 8 def sayer(name, results):
 9     while starter == 0:
10         sleep(0.1)
11     results.append(name)
12
13 for i in range(10000):
14     t = Thread(target = sayer, args = (i, results))
15     t.start();
16     threads.append(t)
17
18 starter = 1
19
20 for t in threads:
21     t.join()
22
23 print len(results)
```

Even though many threads append to the same array, they do not have to take any locks. The GIL implicitly protects the array by allowing only one Python instruction to be run at any time. This is a contrast to the Java version in section 2.1 where the user has to prevent concurrency issues either by using a thread-safe data structure or through manual locking.

Chapter 3

Parrot

Parrot consists of the VM (also called interpreter), and various tools to facilitate the implementation of programming languages on top of the Parrot VM (the *Parrot Compiler Toolkit*). This thesis concentrates on the VM itself. The interpreter is written in C. Example code and test cases are written in *Parrot Intermediate Representation* (PIR) a high-level assembly language abstracting register allocations and function calling conventions.

Contrary to other widely used VMs like the JVM or the *Common Language Runtime* (CLR) which are stack based, Parrot mirrors contemporary hardware CPUs more closely by being register based. A stack based VM usually pops the operands for an operation from the top of a stack and pushes the result back. Thus the operands are chosen implicitly by ordering of the operations allowing the opcodes to be without operands. In a register based VM on the other hand, each operation has to specify the operands explicitly. Compilers for stack machines are simpler because they do not have to care about register allocation and code is independent of prior or subsequent code [6]. The rationale behind giving up the simplicity of a stack based implementation is the hope of simplifying just-in-time (JIT) compilation and improved performance of nested function and method calls.

The current design of Parrot uses four sets of registers where each one has an unlimited number of registers. The four sets are correspondingly to Parrot's types:

- integer,
- floating point,
- string and
- PolyMorphic Container (PMC).

The first three register types are self explanatory. String in Parrot is a low-level type with the interpreter handling all memory allocation issues and Unicode encoding. String values are immutable.

3.1 *PolyMorphic Containers* (PMCs)

PMCs are containers for all high level types such as objects, arrays, hash tables or code. Thus, they are similar to Python's *PythonObject* types. PMCs are defined by C structs and are garbage collected. Their definition looks like:

```
1 struct PMC {
2     Parrot_UInt    flags;
3     VTABLE         *vtable;            /* Pointer to vtable. */
4     DPOINTER       *data;              /* Pointer to attribute structure. */
5     PMC            *_metadata;         /* Pointer to metadata PMC. */
6 };
```

In short, they contain a pointer to a type specific data structure and some meta data. The *vtable* is a virtual method table, where the type's behavior is defined. It contains a list of function pointers forming Parrot's unified type interface. This interface is a union of numeric, string, array, hash and object like behaviors. For example the *get_ integer* function returns an integer value for the data type. For a simple *int* it is the value. For an array it may be the number of elements. The *find_ method* function makes user defined methods of objects possible.

In this way, a language implementer can define the basic types of the language and available operations on them. For each *vtable* entry, there is a corresponding opcode in Parrot's bytecode. Thus an *inc $P0* instruction calls the *increment* vtable function of the PMC pointed to by the *$P0* register.

PMCs are implemented in *pmc* files with method bodies written in C. These files are preprocessed to plain C before compilation.

3.2 *ParrotInterpreter*

The Parrot interpreter is represented by a C struct called *parrot_ interp_ t*. This structure contains pointers to the garbage collector's runtime data, the loaded types, vtables, the runloop, the current continuation and other global data. A pointer to this structure is passed to almost every function as the first parameter.

There is also the *ParrotInterpreter* PMC which exposes parts of the *parrot_ interp_ t* struct to the user.

3.3 Continuation passing

Control flow in Parrot is modeled using continuation passing. A continuation is a data structure containing the state of a program at a given point in its execution. It contains all information necessary to continue a program in a certain state, e.g. a call stack, the instruction pointer and contents of local variables. Calling a continuation means restoring the state encapsulated in the continuation.

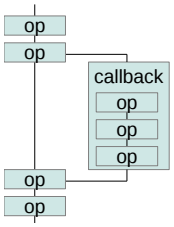

Figure 3.1: Example of a callback executing in a nested runloop.

When a function is called, instead of pushing a return address on the stack, the function is given a return continuation as part of its parameters. Returning from a function means calling the return continuation with possible return values stored in the registers where the calling code is expecting them. Continuation passing makes things like tail call optimization[1] simple and as described in chapter 4 is a important part of implementing green threads.

3.4 Runloops

A runloop is the inner most loop executing bytecode in Parrot. It consists of two steps as shown in the following pseudo code:

```
1 while (pc != NULL) { // program counter points to an opcode
2     op = fetch_opcode_at(pc);
3     pc = execute_opcode(op);
4 }
```

It is also a data structure containing information needed to support exception handling. When some operation in Parrot bytecode is calling a C function and this C function in turn is again executing Parrot bytecode, a nested runloop is started. Examples for such situations include calling a library function with a callback as parameter and exceptions thrown inside Parrot's C code calling a previously defined exception handler in user code. This is shown in figure 3.1.

3.5 Exception handling

Before entering a runloop, the *setjmp* C function is used to save the current stack position and register contents in a data structure stored in the runloop meta data. This is effectively creating something resembling a continuation

[1] When in the last statement of a function another function is called and its value returned unchanged, the outer function can just pass on its return continuation thus saving the need to create a new continuation and one step on returning from the nested function.

at the C level. When an exception is created within the interpreter, the runloop stack is searched for the runloop containing a suitable exception handler. *longjmp* is then used to unwind the call stack up to the point where *setjmp* was called and the call environment is restored. While having great similarities with continuations, this mechanism is more limited. It only allows to jump back to a point "higher" in the call stack.

3.6 Garbage collector (GC)

Parrot supports different Garbage Collector (GC) implementations which can be selected at interpreter startup. Currently, there are four implementations of different algorithms:

- Inf: a GC for debugging purposes never collecting any garbage.
- MS: a basic mark and sweep implementation.
- MS2: a non-recursive mark and sweep implementation.
- GMS: a generational, non-compacting, mark and sweep GC.

GMS is the default.

A mark and sweep GC operates in two phases: In the mark phase, starting from a known root set of objects, the GC follows pointers in the object graph, marking each encountered object as alive. In the sweep phase, all objects not being alive are destroyed and all the live flags are reset.

In a process with many objects, having to traverse the whole graph may take a considerable amount of time. To mitigate this, a generational GC extends this algorithm by assuming that the longer an object is alive, the lower the chances are that it will become unused. So, the objects are partitioned into different generations. The youngest generation will always be traversed while the older generations will be handled much less frequently or even never more at all.

3.7 Historical development

Much of Parrot's previous threading related code has been removed to clean up the code and improve performance. Since the existing threading support was known to be unreliable and seriously flawed, this was no trade off. The final parts were removed by the merging of the *kill_threads* branch on September, 21st 2011.

In 2010, Nat Tuck began working on a *green_threads* branch during his Google Summer of Code internship. The feature got prototyped using pure PIR and then implemented in Parrot's core. He got it to work in simple cases and started to work on OS thread support but the internship ended before the code was ready to be merged into the master branch. The code lay dormant until the work on this thesis started in 2011.

Chapter 4

Green threads

Green threads are one way to model concurrent control flows in a program. To get a better understanding of what they are and how they work, the possible options for supporting concurrency are discussed before explaining green threads in detail.

4.1 Coroutines

Coroutines are functions retaining their state between calls. Instead of returning, they yield control back to the calling function, possibly with an intermediate result returned. A very simple example in Python looks like:

```
1 def counter():
2     for i in range(0, 100):
3         yield i
```

This function returns one integer for every call, counting from 0 to 100. An example usage looks like:

```
1 def count\_up():
2     x = counter()
3     print x
4     x = counter()
5     print x
```

This example prints the numbers 0 and 1. For the two functions to run in parallel, they have to cooperate. One by calling the other repeatedly and the callee by yielding control back to the caller.

4.2 Operating system threads

OS threads are like multiple processes running in parallel but share their memory. The OS is responsible for giving each of these threads CPU time to run. OS threads are the only option where more than one thread may be executed by the CPU(s) at the same time if the hardware permits.

4.3 Green threads

Green threads or *lightweight threads* are threads managed by the VM instead of by the OS. For better distinction green threads are henceforth referred to as *tasks*. These tasks only run pseudo parallel. The VM has a scheduler and support for preempting running tasks. Preemption means to suspend a task regardless of what it is currently doing and probably resuming it later on. This is an implementation of the many-to-one threading model very similar to what an OS running on a system with a single CPU core does. Advantages of green threads are:

- They allow pseudo concurrent processing without endangering the interpreter's internal consistency.
- Green threads are light weight having low memory overhead and close to zero creation time.
- They do not depend on OS threading support.
- The interpreter controls the point of preemption of a green thread.
- The interpreter controls the scheduling policy and may allow the user to influence it or even take over completely.
- Garbage collection can be implemented like in a singlethreaded process.
- Critical sections can be protected by disabling the scheduler until the code is clear of the section.

Disadvantages include:

- Green threads do not allow more than one CPU core to be used for computations.
- Blocking calls like I/O block the interpreter including other green threads.
- The interpreter has to provide logic and timers to control green threads.

Green threads differ from coroutines in that the interpreter decides when a running task is to be preempted while a coroutine depends on explicit yield calls.

Below is an example in order to explain why concurrent processing can endanger the interpreter's internal consistency. The basic problem are concurrent writes to shared variables. Assuming an implementation of an array class consisting of the field holding the data and the number of contained elements in a separate member variable:

```
1 pmclass ResizableIntegerArray auto_attrs provides array {
2     ATTR INTVAL   size;      /* number of INTVALs stored in this array */
3     ATTR INTVAL * int_array; /* INTVALs are stored here */
```

To append a new value, the array has to read the current size, write the new value at the position *size* + 1 and then write the incremented size back into the member variable. Now, if two threads simultaneously try to do this, it may happen that both read the same size, write to the same position (with

one overwriting the value of the other) and write back the same incremented size. In this case one of the appended values is lost.

In a more complicated example, the array could have to resize its data buffer to accept the new value. It would read the current size, allocate a new buffer, copy the values to the new buffer and then destroy the old one. Again, if two threads try to do this simultaneously, one of them could still be copying data, while the other already destroys the old buffer. This leads to the copying thread accessing freed memory.

When garbage collection is brought into the mix, the possibilities for corruption grow even further. At the point where the first thread needs to allocate the new buffer, the GC could decide that it needs to clear some unused memory. It would have to traverse the object graph to mark alive objects. Between the mark and the sweep phases, the second thread could change the pointer to the buffer from the old one to the new one. So the old buffer would have been marked alive, while during the sweep phase the new buffer would be in its place, but not yet marked alive.

4.4 Green threads in Parrot

Parrot's green threads implementation is based on Nat Tuck's *green_ threads* branch developed during his Google Summer of Code internship [13].

In Parrot, green threads are called *Tasks*. Each task is assigned a fixed amount of execution time. After this time is up a timer callback sets a flag which is checked at execution of every branch operation. Since the interpreter's state is well defined at this point, its internal consistency is guaranteed. The same holds for the GC. Since task preemption is only done while executing user-level code, the GC can do its work undisturbed and without the need for measures like locking. Since user-level code is allowed to disable the scheduler, it can be guaranteed to run undisturbed through critical sections.

The scheduler is implemented as a PMC type. This allows the user to subclass this PMC thus allowing fine-grained control over the scheduling policy. Features, a user could add this way would be for example giving different priorities to tasks or implementing the possibility to suspend and resume a task.

Chapter 5

Design of hybrid threads

This chapter describes how green threads are used to solve the following problems occurring when trying to implement threading support:

- How to ensure internal interpreter consistency when doing writes to shared variables?
- How to implement critical sections?
- How to handle GC?

In keeping the analogy of the interpreter being a software CPU, multithreading is implemented by having a separate interpreter with its own register set for each thread. Thus the words *thread* and *interpreter* are used interchangeably as there is a 1:1 relationship between them. When a user starts a new task, the scheduler first looks for an idle thread. If one can be found, the task is scheduled on the thread's interpreter. If none can be found, a new thread with a new interpreter is started. Parrot tries to optimize the number of utilized threads by creating at most one for each CPU core in the system. If more tasks are started than the maximum number of threads, the tasks are distributed evenly among the running interpreters. This is effectively an implementation of the N:M threading model.

5.1 Shared data

As described in the introduction, cross-thread writes to shared variables may endanger the internal consistency of the interpreter. Traditionally, the solution to this problem is the use of locks of varying granularity. Fine-grained locking allows code to run in parallel but taking and releasing locks costs performance. It not only increases the instruction count and memory accesses but it also forces the CPU cores to coordinate and thus communicate. Even a seemingly simple operation like an atomic increment can take two orders of magnitude longer than a normal increment [7]. While the gain through being able to utilize multiple CPU cores may offset this cost, it is still impacting

the common case of having only a single thread running.

Too coarse locking on the other hand would reduce scalability and the performance gains through parallel execution by having threads wait for extended periods for locks to become available. In the extreme case of having a global interpreter lock it would effectively serialize all computations costing much of the benefits of using threads in the first place.

The other problem with locking is the possibility of introducing deadlocks. For example, two functions *F1* and *F2* both use two resources A and B protected by locks. If *F1* first locks A and then tries to lock B while *F2* has already locked B and is now trying to lock A, the program would come to a halt. Both functions would be left waiting for the other to unlock the resource which will never happen. With fine-grained locking, the possibilities for such bugs grow quickly. At the same time, it is easy to miss a case where a lock would be appropriate leading to difficult to diagnose corruption bugs.

The solution for these problems implemented in this thesis is to sidestep them altogether by disallowing write access to shared variables. The programmer (or in most cases the compiler) is obliged to declare a list of all shared variables before a newly created task is started. The interpreter would then create proxy objects for these variables which the task can use to access the data. These proxies contain references to the original objects. They use these references to forward all reading *vtable* functions to the originals. Write access on the other hand would lead to a runtime error.

In other words, all data is owned by the thread creating it and only the owner may write to it. Other threads have only read access.

For threads to be able to communicate with their creators and other threads, they still need to write to shared variables. This is where green threads come into play. Since green threads are light weight, it is feasible for a thread to create a task just for updating a variable. This task is scheduled on the interpreter owning this variable. To reduce latency, the task is flagged to run immediately. The data-owning interpreter will preempt the currently running task and process the new write task. Put another way, the data-owning interpreter is told what to write to its variables, so other threads don't have to.

5.2 Critical sections

Critical sections can be implemented by disabling preemption until the code has left the section. With other threads not being allowed to write to the data and the current task running uninterrupted it it guaranteed to complete without the data being changed from outside.

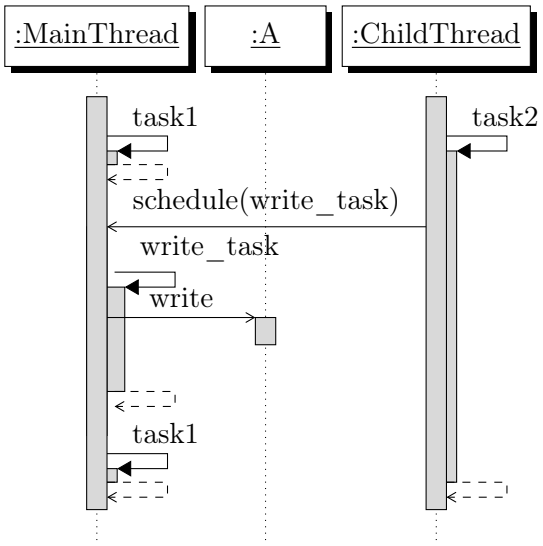

Figure 5.1: *task2* running on *ChildThread* is sending the *write_task* to the *MainThread* to write to the shared variable *A*.

5.3 Garbage collection (GC)

A GC has to traverse and process the entire graph of objects in memory. If during this traversal another thread is changing this structure by creating new objects, moving them around in the graph or removing them, the GC's data would be compromised. For example, it could happen that the GC does not mark referenced objects alive because they were added to objects which have already been processed.

In previous attempts to implement threading support in Parrot, the solution to this problem was to disable the GC while multiple threads were active. While multithreaded GC algorithms exist, their implementation would have been too complex and brittle for this thesis [2]. Other VMs solve this problem by suspending all threads while the GC is running. Suspending all threads is not as simple as it sounds because these threads can be in any state at the time. For example, they could be blocking on some long running I/O operation like waiting for a reply on a network connection. A thread can stay in this condition indefinitely thereby never being able to confirm synchronization.

Since cross-threaded writes are already forbidden and all read access to other thread's data goes through the narrow channel of proxy objects, forcing a complete separation of the thread's memory domains is only a small step. By having separate memory areas for each interpreter, it becomes possible to have each interpreter run its own GC. This way, the known to work singlethreaded GC implementation can be used nearly unchanged.

With separated memory domains, it can happen that an object is created on one thread, used on another thread but not any more on the owner thread. Without any additional measures, the GC would collect such objects, since it does not know about references from other threads. But since all objects which could be accessed from other threads have to be pushed onto the task object representing these threads, the objects can still be referenced from the task object. Since the task is still owned by the original thread, the GC knows that these objects are still in use.

Chapter 6

Implementation of hybrid threads

6.1 Scheduler

The scheduler is the place where most of the green thread logic is implemented. It consists of two parts: functions using the prefix *Parrot_cx* which are located in *src/scheduler.c* and the *Scheduler* PMC type containing the scheduler's data and public interface.

The process of executing and preempting tasks is pictured in figure 6.1 and described in detail in the rest of this section.

Parrot_cx_init_scheduler is called from *Parrot_interp_initialize_interpreter* whenever a new interpreter is created, e.g., on interpreter startup and whenever a new thread is created. It creates the scheduler PMC and sets up alarm signal handling.

The scheduler is hooked into the system by replacing the call to execute the *main_sub* by a call to *Parrot_cx_begin_execution*. The latter creates the main task using *main_sub* as the task's code. This task is then put onto the run queue. The scheduler timer gets enabled and control is given to *Parrot_cx_outer_runloop*.

Parrot_cx_outer_runloop is the loop taking tasks from the run queue and executing them. Despite its name, it may not be confused with runloops discussed in section 3.4. This function has to execute tasks as long as tasks are available. If no tasks are available, it lets the thread sleep to not uselessly burn CPU time. On the other hand it should simply end instead of sleep if no tasks are available, no alarms pending and no tasks are being executed on other threads.

As can be seen in listing 6.2, *Parrot_cx_outer_runloop* therefore consists of two nested loops. The inner loop fetches tasks from the task queue and executes as long as tasks are available. Even when all tasks are finished there may still be alarms pending which upon termination would trigger new tasks

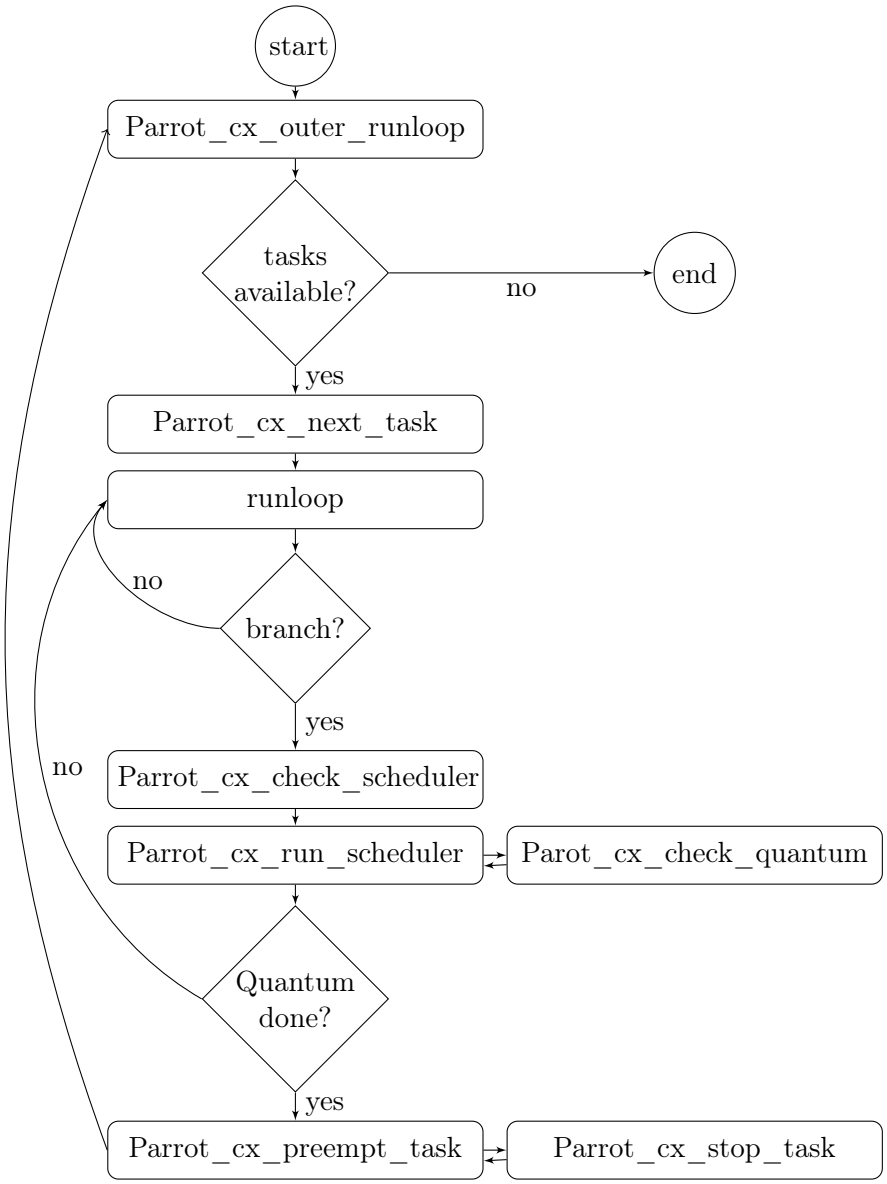

Figure 6.1: Control flow of task execution and preemption.

to be scheduled. For this reason, there is an outer loop checking for pending alarms. If it finds any it puts the thread to sleep until the next alarm expires.

Parrot_cx_next_task contains the code to take the next task from the run queue and execute it. Before calling the task, it checks if there are other tasks still in the queue. Only if other tasks are waiting, task preemption is enabled. Otherwise, the current task will run until it finishes by itself, schedules other tasks or an alarm expires. This is an optimization reducing

```
do {
    while (tasks_available()) {
        Parrot_cx_next_task()
        Parrot_cx_check_alarms()
    }

    clean_finished_foreign_tasks()

    if (
        not tasks_available()
        and (alarms_pending() or foreign_tasks_active())
    ) {
        Parrot_thread_wait_for_notification()
        Parrot_cx_check_alarms()
    }
} while (
    alarms_pending()
    or foreign_tasks_active()
    or tasks_available()
)
```

Figure 6.2: Pseudo code describing Parrot_cx_outer_runloop.

the runtime overhead of green threads in the important single tasking case to zero.

To enable task preemption, *Parrot_cx_enable_preemption* sets a flag on the scheduler PMC and uses *Parrot_cx_set_scheduler_alarm* to set an alarm at a point in time *PARROT_TASK_SWITCH_QUANTUM* milliseconds in the future.

Checking for expired alarms after every executed operation would be too costly. Therefore the *branch* operation has been picked as the point where alarms are checked. To make this as cheap as possible, Parrot increments the global *alarm_serial*. This serial is compared to the value last seen by the *Parrot_cx_check_scheduler* function. If they are different, or the *SCHEDULER_wake_requested* flag is set, *Parrot_cx_check_scheduler* wakes up the scheduler by calling *Parrot_cx_run_scheduler*.

Parrot_cx_check_quantum is used to check if the alarm signal means that the current task has used its assigned quantum and should therefore be preempted.

Preemption of a task is implemented in *Parrot_cx_preempt_task*. It uses *Parrot_cx_stop_task* to create a continuation at the current point of execution and store it as the task's code. The task itself is then appended at the end of the run queue. The function returns a *NULL* opcode which is propagated all the way up through the stack back to the *branch* operation checking for alarms. The operation then recognizes this as the signal to stop processing and to exit the runloop.

The control flow ends up back at *Parrot_cx_next_task* at the point after executing the task. *Parrot_cx_next_task* then returns to *Parrot_cx_outer_runloop*.

The *schedule* opcode is used to schedule a new task from user code. It uses *Parrot_cx_schedule_task* which starts new worker threads if needed and possible and pushes the task on the target thread's scheduler's task queue. If the target thread previously was executing only a single task, its preemption has been disabled for optimization as described above. So in this case *Parrot_cx_schedule_task* has to enable preemption to give the new task a chance to run.

Parrot_cx_schedule_immediate is used in various places like at alarm expiry for putting a task at the head of the run queue and immediately causing preemption of the currently running task. It does this by setting *SCHEDULER_wake_requested* and *SCHEDULER_resched_requested* flags the same as when the preemption alarm expires. To be precise, this mechanism leads to the current task being preempted at the next *branch* operation since this is the place where the mentioned flags will be checked.

Alarms can be registered using *Parrot_cx_schedule_alarm*. It puts the alarm in the appropriate place in the ordered alarm list and uses *Parrot_alarm_set* which sets the actual alarm but only if there is not another alarm already set for an earlier time, since there can only be one POSIX alarm pending at any time.

Parrot_cx_schedule_sleep is the actual implementation of the *sleep* op. Like in *Parrot_cx_preempt_task*, *Parrot_cx_stop_task* is used to get an updated task at the current execution position, but instead of pushing this on the run queue, it is used as a callback for a newly set alarm. Again, the *NULL* opcode is used to stop processing of the current task.

6.2 Scheduler PMC

The Scheduler PMC contains the following attributes:

- *PMC *task_queue*: list of tasks/green threads waiting to run.
- *Parrot_mutex task_queue_lock*: a lock protecting the task_queue so other threads can access it safely.
- *PMC *alarms*: list of future alarms ordered by time.
- *PMC *all_tasks*: hash of all active tasks by ID.
- *UINTVAL next_task_id*: ID to assign to the next created task.
- *Parrot_Interp interp*: a reference to the scheduler's interpreter.

push_pmc and *unshift_pmc* are used to add a task at the end, respectively the beginning of the task_queue. *shift_pmc* is used to fetch and remove a task from the tip of the *task_queue*. *get_integer* returns the number of tasks in the *task_queue*. These four methods use the *task_queue_lock* to

make accessing the *task_queue* thread-safe. Thus, the Scheduler PMC acts as a container while using *task_queue* to actually store the data.

alarms is list containing all pending alarms, sorted by their expiration time. Sorted insertion is used to keep alarms in order. Sorting makes it simple and efficient to retrieve all alarms that have expired at a certain point in time.

The *active_tasks* returns an array containing all tasks which have been run at least once and are not yet finished, e.g., they are currently being executed or are preempted.

6.3 Task PMC

The Task PMC contains the following attributes:

- *UINTVAL id*: Unique identifier for the task.
- *FLOATVAL birthtime*: the creation time stamp of the task.
- *Parrot_Interp interp*: the interpreter which created the task.
- *PMC *code*: the code to run.
- *PMC *data*: additional data for the task given as parameter to *code*.
- *INTVAL killed*: flag marking killed tasks.
- *PMC *waiters*: tasks waiting on this one.
- *PMC *shared*: list of variables shared with this task.
- *PMC *partner*: copy of this task in another thread.

The *invoke* method is used to run the task. It first checks the *killed* flag to see if the task has been killed while waiting in the task queue. Parrot manages a *recursion depth* counter which records, how many levels the call stack has. This counter is incremented by *Parrot_Sub_invoke* for each call to a subroutine and decremented when a subroutine returns. While *Parrot_Sub_invoke* is used to start or resume a task, the executed subroutine does not return when the task is preempted. This leads to *recursion depth* growing until a *recursion depth exceeded* exception is thrown. Therefore, as a workaround, the current *recursion depth* is saved to a local variable and restored when the task stops executing.

The next steps are to add the task to the list of active tasks and to invoke the task's *code*. *data* is an optional parameter to the subroutine given as *code*. Since this can be a compound object, a single parameter can cover all use cases.

If the task has been killed while being in the task queue or while running or it just ended, it gets removed from the list of active tasks and all tasks registered with this task's *waiters* array are added to the scheduler's task queue to be run.

push_pmc and *pop_pmc* are used to add or remove variables to the task's list of shared variables. For these variables, proxies will be created when the task is run on a different thread.

The *kill* method is used to set the task's *killed* flag.

6.4 Runloops

Runloops are created whenever a part of Parrot's C internals starts to execute bytecode. This can be at interpreter startup to start the actual program execution or when some library function needs to execute a user callback. An exception handler can also be such a callback. Each runloop has a unique id. These ids are numbered from 0 and increased by 1 whenever a new runloop is created.

When an exception is thrown, Parrot searches for a previously set up exception handler and executes it. As part of exception handling finalization, it cleans up, freeing any information no longer needed. The exception handler does not have to have been set up in the same runloop from which the exception has been thrown. Parrot uses the runloop id to identify such situations and free the nested runloops up to the level of the exception handler.

A continuation also contains the id of the runloop in which it has been created. When resuming a preempted task, this id is not the same as the one of the runloop created for executing its code. This leads to Parrot not finding the correct runloop when finalizing an exception.

To mitigate this, the *reset_runloop_id_counter* function is used by *Parrot_cx_outer_runloop* to reset the global runloop id counter back to 0 when resuming a task. This way, the task retains the same runloop id over its life time.

If Parrot were to preempt a task while it is executing a nested runloop, it would have to somehow capture not only the interpreter's current state but also the C stack between the outermost runloop and the currently executing one. It would also have to recreate these call and runloop stacks when resuming the task. Otherwise, the task would be ended as soon as the executing callback would be completed since instead of some C function called by an operation in the bytecode, the scheduler is the owner of the runloop.

Thus, the scheduler checks for the current runloop id and does not preempt a task if it is currently running a nested runloop. Furthermore even a manual yield to another task is not possible in such a situation. The only way to solve this problem is to get rid of nested runloops in the interpreter.

6.5 Alarms and timers

Timers used for *sleep*, alarms and preemption use a common timer thread implemented in *src/alarm.c*. This thread sleeps until a new alarm is set or the currently active one expires. When the latter happens, it increases the global alarm serial and notifies all threads that a timer is expired. Since it does not keep a list of pending alarms but knows only about the next one,

interpreter threads are obliged to reset their next alarms upon notification.

6.6 Threads

This section describes how OS threads are used to execute tasks in parallel.

6.6.1 Creation

Each thread is represented by an instance of the ParrotInterpreter PMC. These interpreters are kept in the *threads_array* defined in *src/thread.c*. It is a C array instead of one of Parrot's dynamic arrays implemented as PMCs because although ParrotInterpreter is implemented as a PMC itself, it is not garbage collected because of the bootstrapping issues this would create.

src/thread.c also contains thread management functions all prefixed by *Parrot_thread_*. Threads are created when *Parrot_cx_schedule_task* determines that all existing threads are busy and new threads can be started. *Parrot_cx_schedule_task* uses *Parrot_thread_create* to create a new interpreter by cloning the current one and giving it an empty *thread_data* structure.

The task is then scheduled with the new interpreter's scheduler using *Parrot_thread_schedule_task*. This function uses *Parrot_thread_create_local_task* to create a corresponding task on the new interpreter and then pushes it onto the scheduler. Since each interpreter has its own GC, all objects used on the interpreter must origin from this GC's memory pools. That is why *Parrot_thread_create_local_task* creates a new *local_task* as desribed in figure 6.3. Proxies are created for the task's *code* and *data* attributes (lines 3 and 4). The task object of the originating interpreter and the new *local_task* are linked using their *partner* attributes containing a pointer to the other object (lines 11 and 12). For all variables in the original task's *shared* array proxies are created an put into the *shared* array of the *local_task* (lines 6–9). *Parrot_thread_create_proxy* is used for this purpose.

Parrot_thread_insert_thread is used to put the new interpreter into the *threads_array*.

Parrot_thread_run is the function where the actual OS thread is created. It uses macros defined in the *include/parrot/thr_*.h* include files which act as an OS abstraction layer.

Parrot_thread_outer_runloop is used as the thread's main function. It is very similar to *Parrot_cx_outer_runloop*. The most important difference is that *Parrot_thread_outer_runloop* waits indefinitely for new tasks while *Parrot_cx_outer_runloop* ends if no tasks are queued and no alarms are pending.

```
 1 local_task = new Task()
 2
 3 local_task->code = Parrot_thread_create_proxy(task->code)
 4 local_task->data = Parrot_thread_create_proxy(task->data)
 5
 6 foreach (variable in task->shared) {
 7     proxy = Parrot_thread_create_proxy(variable)
 8     push(proxy, local_task->shared)
 9 }
10
11 local_task->partner = task
12 task->partner = local_task
```

Figure 6.3: Pseudo code describing Parrot_thread_create_local_task.

6.6.2 Proxies

Proxies are the arbiters between threads. They are the only means for a thread to access another thread's data and are implemented by the *Proxy* PMC type. This type has only two attributes, which are not garbage collected:

- *PMC *target*: the PMC this object proxies to.
- *Parrot_Interp interp*: the interpreter owning target.

As described in section 3.1, PMC source files get preprocessed before compilation by a C compiler. This preprocessing is done by the *pmc2c.pl* Perl script. This script is extended by the *Parrot::Pmc2c::PMC::Proxy* module which creates default implementations for all *vtable* functions not otherwise defined in the *proxy.pmc* file. The default implementation for all writing functions just calls *cant_do_write_method* which creates a runtime exception.

All other methods call the vtable method of the same name on the *target* passing the current interpreter as *interp* and all other parameters unchanged. This causes all access to globals by the proxied function to go through the thread's *interp* and new PMCs created during the call to be allocated from the thread's memory pool. If the method returns a PMC from the target's *interp*, another proxy object has to be created and wrapped around it so it can be safely returned to the caller.

To differ between PMCs originating from the target's *interp* and those created on the thread's *interp* during the call, the GC is told to set the new *PObj_is_new_FLAG* on newly created PMCs. The *PARROT_THR_FLAG_NEW_PMC* flag on the *interp* is used to communicate this requirement to the GC.

The alternative of calling the proxied function using the target's *interp* leads to concurrency issues with the GC. Any call may cause GC to run.

Since the GC also scans the C stack the target's GC would find the thread's PMCs there and mark them confusing the thread's GC. Disabling the GC requires a lock since the target's GC may already have started collecting.

Sub

The *Sub* PMC represents executable subroutines. A *Sub* does not only contain the code to execute but also the context in which to execute the code such as visible globals and namespaces. If a proxy to such a *Sub* were created and *invoke* called on it, the code would access this context directly since it belongs to the same *interp* as the proxied *Sub* itself. Thus, an operation like *get_global* fetches a global from an unproxied namespace and an unproxied global is be put into the target register. Since this is happening while running *invoke* on the original *Sub*, *Proxy* cannot intercept the call and create a *Proxy* for the result.

This is the reason why *Parrot_thread_create_proxy* does not create a *Proxy* for a *Sub* but uses *Parrot_thread_create_local_sub* to create a copy on the thread's *interp* with proxies for all PMC attributes like *namespace_stash* and *ctx*.

6.6.3 Writing to shared variables

As described in chapter 5, to write to shared variables, a thread creates a task and schedules it on the data owning interpreter. An example task looks like this:

```
1 .sub write_to_variable
2     .param pmc variable
3     variable = 1
4 .end
```

This is a subroutine with just one parameter. The variable passed as this parameter is the one the task should write to. In this case the constant value 1 would be written to the variable. In PIR, an assignment to a PMC gets translated to a method call. In this case, the *set_integer_native* is called changing the variable's value. Since PMCs are passed by reference, it is the original variable which gets written to.

Code to create the task looks like:

```
1     write_task = new ['Task']
2     setattribute write_task, 'code', write_to_variable
3     setattribute write_task, 'data', shared_variable
4     interp.'schedule_proxied'(write_task, shared_variable)
```

Line 1 creates a new task object. The example subroutine is used for the task's *code* attribute. *shared_variable* is used for *data*. At this point, *shared_variable* is actually the proxy object created for the shared integer PMC. The interpreter object contains a *schedule_proxied* method which is used

to schedule the *write_task* on the thread owning the original variable. This owner thread's interpreter cannot be used directly for scheduling the task, since it would have to be stored in a register to be accessible to PIR code. But then, the same problem as described in section 6.6.1 would occur with the *ParrotInterpreter* PMC tripping up the GC.

schedule_proxied uses *Parrot_thread_create_local_task* which in this case detects that the *data* given as parameter for the task's *code* is actually a proxy already and unwraps the proxied object. *Parrot_cx_schedule_immediate* is then used to make the data owning interpreter execute the task as soon as possible.

To protect a critical section, preemption can be disabled so the critical section runs uninterrupted:

```
1 .sub swap_variables
2     .param pmc a, b
3     .local temp
4     disable_preemption
5     temp = a
6     a = b
7     b = temp
8     enable_preemption
9 .end
```

6.6.4 *wait* operation

Using *tasks* to write to shared variables makes such actions inherently asynchronous. This is not always what is needed by the implemented algorithm. For example, when the shared variable is a lock, processing should continue as soon as it's acquired. The *wait* operation is used to wait for a task's completion. The waiting task is added to the waited for task's *waiters* list and preempted immediately. When a task finishes, all the tasks in the *waiters* list are scheduled again for execution. Since for each task a local copy is created on the target thread, the running task not only checks its own *waiters* list but also its partner's.

If a task on the main thread was waiting for a task on another thread to finish and no other tasks are in the scheduler's queue on the main thread, the main thread exits if no alarms are pending. To prevent this unintended exit, all tasks are added to the scheduler's *foreign_tasks* list when they are scheduled on other threads. To end the program with other threads still running, an explicit *exit* operation has to be used.

6.6.5 Garbage collection

Since each interpreter has its own GC and consequently its own memory areas and can only access its own data directly, the GC can act as if there were no other threads. After replacing references to global data by proxies,

only a few special cases had to be built into the GC. The global variable *PMCNULL* is used in many places in the code, but should be handled only by the main thread's GC.

Only the main thread is allowed to load additional bytecode or create new classes since it owns the data structures used to manage these things. Child threads use these via proxies. In previous implementation attempts code segments, classes and vtables have been copied for each new interpreter making thread startup a costly operation while still causing subtle problems.

Code segments and vtables are only marked by the main thread's GC. Code segments are managed in Parrot using *PackFile* structures. Besides the code itself, these structures contain tables of constants used in the code. These constants are expanded to PMCs by the bytecode loader when the code is loaded from disk or put there directly by the PIR compiler. Since *PackFiles* are no PMCs, the Proxy PMC cannot be used to shield them from the GC. Accessing the contained constants would therefore lead to unproxied PMCs being accessed by different threads. Since PackFiles may only be loaded by the main thread, the bytecode loader and the compiler were changed to flag all *PackFile* constants as *shared*. They still are used as is on all threads, but the GC knows to only handle *shared* PMCs on the main thread.

Scheduling tasks on other interpreters may cause garbage collection due to the need of creating proxy objects. To protect the GC from concurrency issues, garbage collection is disabled during creation of the new *Task* object and the proxies on the target thread's *interp*. But since the target thread may itself be allocating memory or collecting garbage, the GC has to be protected by a lock as well. The *interp->thread_ data->interp_ lock* lock is used for this. It is checked by all allocation and freeing functions which are also the only places where a garbage collection run may be triggered. Note that despite the GC being protected from concurrent access, this alone would still not be enough to share a GC between different threads since these threads could be changing references to objects while the GC does its marking and sweeping. When only creating new proxies (which contain no references visible to the GC), accessing a foreign GC is safe on the other hand.

6.6.6 Conclusion

This threading implementation is mostly contained in a few files. Only a few changes had to be done in other places like the *ParrotInterpreter* and the GC. This is a major difference to the previous implementation where all PMCs contained threading related code. As the next chapter shows, this design helped to reach a working state and meet the performance goals quickly.

Chapter 7

Tests and benchmarks

In addition to Parrot's extensive test suite, tests were conducted using three test programs.

7.1 *tasks.pir*

tasks.pir is a simple test program, executing two tasks called *a* and *b* which run in tight loops printing their name on each 100000th iteration. The program ends after running for 10 seconds. This is a test of thread creation and task scheduling but does not access any shared variables. The code is as follows:

```
 1 .sub main :main
 2     .local pmc task, a, b, a_task, b_task
 3     task = get_global 'task'
 4     a = new ['String']
 5     a = "a"
 6     b = new ['String']
 7     b = "b"
 8     a_task = new ['Task']
 9     setattribute a_task, 'code', task
10     setattribute a_task, 'data', a
11     b_task = new ['Task']
12     setattribute b_task, 'code', task
13     setattribute b_task, 'data', b
14     schedule a_task
15     schedule b_task
16     sleep 10
17     exit 0
18 .end
19
20 .sub task
21     .param pmc name
22     .local int i
23 start:
24     print name
25     i = 0
```

```
26 loop:
27     inc i
28     if i >= 100000 goto start
29     goto loop
30 .end
```

The test shows that task creation and execution works and that tasks do run in parallel.

7.2 *moretasks.pir*

moretasks.pir is an extended test where the main task creates 50000 child tasks. These tasks poll a shared variable with pauses of 100 ms. When the variable is set to 1, they schedule tasks on the main thread to write their number to a results array. When the results array contains the expected number of results, the process is repeated.

This test stresses reading and writing to shared variables, allocation and garbage collection. It proved to be valuable for finding all sorts of concurrency issues. The complete source code is available in appendix A.

The program runs stable and does not leak memory demonstrating that the GC works.

7.3 *matrix_part.winxed*

This test implements matrix multiplication [17] using four threads. For simplicity the second matrix only has one column. The program is written in the *Winxed* programming language. Winxed is a low-level language with Javascript like syntax and the possibility to include sections of PIR code verbatim making it possible to try experimental opcodes while writing more readable and concise code than with PIR alone. The complete source code is available in appendix B.

The program consists of the parts initialization, computation and verification. Computation is parallelized using four tasks each calculating one fourth of the result vector. Runtime is compared to a simple singlethreaded implementation. Run times were measured using the *time* command and are recorded in table 7.1.

As can be seen, the multithreaded implementation gives an average speedup of 2.31 for the computation and 1.61 in total.

7.4 *chameneos.pir*

Chameneos is a game useful for testing peer-to-peer cooperation of threads presented in [3]. It revolves around fictional creatures, each being of one of three colors. Two of these creatures may meet and exchange their colors, each

Table 7.1: Runtime comparison for matrix multiplication

	singlethreaded	*computation*	*multithreaded*	*computation*
1. run	28.522 s	19.530 s	17.543 s	8.478 s
2. run	28.427 s	19.463 s	17.320 s	8.283 s
3. run	28.200 s	19.235 s	17.489 s	8.473 s
average	28.383 s	19.409 s	17.451 s	8.411 s

Table 7.2: Runtime comparison for Mandelbrot set calculation

	singlethreaded	*1 thread*	*2 threads*	*4 threads*	*8 threads*
1. run	89.931 s	89.978 s	45.813 s	24.028 s	17.445 s
2. run	89.707 s	89.871 s	45.906 s	24.048 s	17.695 s
3. run	90.318 s	89.839 s	45.951 s	24.049 s	17.573 s
average	89.985 s	89.896 s	45.890 s	24.042 s	17.571 s
speedup	1.000	1.001	1.959	3.739	5.116

one independently calculating and adopting a complementary color depending on those two. These creatures have to be implemented using preemptive kernel or lightweight threads. Since the color calculation is trivial this test's performance depends on the available thread synchronization primitives and writes to shared variables. It therefore represents a worst case scenario for the implemented threading model. The complete source code is available in appendix C.

Preliminary benchmarks have shown Parrot's performance to be within an order of magnitude of that of an optimized implementation in Perl 5. Since Parrot does not yet offer the user any synchronization primitives, locks had to be implemented using a shared variable which is written to only by the main thread. Replacing this primitive method with a native semaphore implementation would probably reduce runtime to a small fraction.

7.5 *mandel_inter.winxed*

Calculating an image of the Mandelbrot set [16] is a common benchmark for multithreading implementations since calculations of points are independent of each other and are thus easily parallelizable. A simple implementation of the escape time algorithm written in Winxed has been used to determine scalability properties of the threading implementation. The image is split into lines which are calculated alternatedly by a configured number of tasks. Run times were measured using the *time* command on an Intel Core i7 3770K processor with 16 GiB RAM running openSUSE 12.1 and are recorded in table 7.2

As can be seen, the implementation scales nearly linearly up to four threads reflecting the CPUs four physical cores. Using eight threads, the speedup is only 1.368 compared to four threads but this seems to be more a limitation of the hardware than the implementation.

Chapter 8

Conclusion, further work and experiences

This thesis shows a way to implement threading support in a dynamic language runtime. Changes to the interpreter itself could be kept at a reasonable level and very local to the threading implementation itself. Benchmarks have shown that no slowdowns were measurable in the still very important singlethreaded case, while the implementation scales well using multiple threads.

Practice will show how this threading model will fare with real world workloads. It very much depends on the ratio of read to write accesses to shared variables.

8.1 Further work

There are several ways in which the current implementation can be improved.

8.1.1 Remove locking from the GC

The initial goal of an almost lock free implementation has not been met. The GC has to take a lock in most of its functions. Though these locks are used only when more than one interpreter thread is running, they still represent an overhead which could be avoided. A way to accomplish this is to pre-allocate proxy objects instead of allocating them on demand. The allocation could be done as part of a thread's outer runloop. This eliminates the need for the locks in the GC. The number of proxy objects to pre-allocate would be an important knob for optimization, since too few objects lead to starvation quickly when many tasks are scheduled.

8.1.2 Allow threads to create threads

In the current implementation, there are several places with an implicit assumption that new threads may be only started from the main thread. Similarly only the main thread may schedule tasks on other threads. Sub threads may schedule tasks only on the main thread, not on other sub threads. These restrictions are not inherent in the implemented threading model but were introduced due to time constraints.

8.1.3 Dynamic maximum number of threads

The maximum number of threads is determined by the MAX_THREADS constant in *include/parrot/thread.h*. Instead the value should be determined at runtime probably depending on the available hardware resources, demand and system load.

8.1.4 Allow multi-level proxying

Communication between sub threads is currently restricted by the fact that proxy objects can only be transfered back to the thread which created them. To remove this restriction, *Parrot_thread_create_proxy* has to be extended to detect an attempted transfer of a proxy to a different thread and unwrap it first before creating a new proxy.

8.1.5 Allow redistribution of tasks to other threads

Currently, the scheduler picks the interpreter with the smallest number of tasks in the task queue for scheduling a new task. This can lead to a situation where threads sit idle while tasks have to wait in other thread's queues. Some tasks get scheduled on specific threads because they need write access to the thread's PMCs. To allow migration of waiting tasks to idle threads, there has to be a method to distinguish such tasks from tasks which are there simply because the thread's queue was the shortest at the time the task got scheduled. The more difficult part of implementing task migration is that proxies are allocated on the target interpreter at the time of scheduling. Since these proxies belong to the interpreter, they are created on, new proxies would have to be created on the interpreter the task is being migrated to. It is yet unclear if this overhead can be offset by better utilization of idle threads.

8.2 Experiences

I picked a very advanced topic for my Bachelor's thesis. While I had to invest many more hours than strictly necessary to get a Bachelor's degree, the work was also very interesting, instructive and hopefully valuable to the

Parrot project and the free software community. Considering that I had no prior experience with Parrot which is a complex piece of software, it is not surprising that I spent most of the time discovering how it works and how its many features interact. In addition, concurrency usually brings with it difficult to diagnose problems further complicated by Parrot's cross platform nature.

It is therefore no exaggeration to state that without the help of the Parrot community I could not have come nearly as far as I did in the time I had. They are very welcoming and were supportive and grateful from the first day, boosting my motivation. I can only recommend to other students to approach the free software communities when looking for thesis topics.

Appendix A

moretasks.pir

This program tests the GC by creating tasks and writing to shared variables.

```
 1 .sub main :main
 2     .local pmc task, sayer, starter, number, interp, tasks, results
 3     .local int i, num_results, results_rem
 4     interp = getinterp
 5     sayer = get_global 'sayer'
 6 init:
 7     starter = new ['Integer']
 8     i = 1
 9     starter = 0
10     say "1..100"
11     tasks = new ['ResizablePMCArray']
12     results = new ['ResizablePMCArray']
13 start:
14     number = new ['String']
15     number = i
16     task = new ['Task']
17     push task, results
18     push task, starter
19     setattribute task, 'code', sayer
20     setattribute task, 'data', number
21     print "ok "
22     say number
23     push tasks, task
24     schedule task
25     inc i
26     if i > 50000 goto run
27     goto start
28 run:
29     starter = 1
30 check_results:
31     pass
32     num_results = results
33     results_rem = num_results % 1000
34     if results_rem != 0 goto skip_say
35     say num_results
36 skip_say:
```

```
37      if num_results >= 50000 goto end
38      goto check_results
39  end:
40      goto init
41  .end
42
43  .sub sayer
44      .param pmc name
45      .local pmc interp, task, starter, results, result_sub, result_task
46      .local int i
47      interp = getinterp
48      task = interp.'current_task'()
49      starter    = pop task
50      results    = pop task
51      result_sub = get_global 'push_result'
52  start:
53      if starter > 0 goto run
54      sleep 0.1
55      goto start
56  run:
57      result_task = new ['Task']
58      setattribute result_task, 'code', result_sub
59      setattribute result_task, 'data', results
60      push result_task, name
61      interp.'schedule_proxied'(result_task, results)
62  .end
63
64  .sub push_result
65      .param pmc results
66      .local pmc interp, task, number
67      interp = getinterp
68      task = interp.'current_task'()
69      number = pop task
70      push results, number
71  .end
```

Appendix B

matrix_part.winxed

This program implements multithreaded matrix multiplication.

```
#!./parrot
# Copyright (C) 2012, Parrot Foundation.

function main() {
    var multi_part_code = multi_part;
    var matrix = new 'FixedPMCArray'(10000);
    var vector = new 'FixedIntegerArray'(10000);
    var results = new 'FixedIntegerArray'(10000);
    ${set_global 'results', results};

    for (int i = 0; i < 10000; i++) {
        matrix[i] = new 'FixedIntegerArray'(10000);
        for (int j = 0; j < 10000; j++)
            matrix[i, j] = 1;

        vector[i] = 1;
        results[i] = 0;
    }

    var tasks = new 'FixedPMCArray'(4);
    for (int i = 0; i < 4; i++) {
        var task = new 'Task';
        task.code = multi_part_code;
        task.data = i;
        ${ push task, matrix };
        ${ push task, vector };

        ${schedule task};
        tasks[i] = task;
    }

    for (int i = 0; i < 4; i++)
        ${wait tasks[i]};

    for (int i = 0; i < 10000; i++)
        if (results[i] != 10000) {
```

```
37              print("results[");
38              print(i);
39              print("]: ");
40              say(results[i]);
41              say("wrong result!");
42              exit(1);
43          }
44      exit(0);
45  }
46
47  function multi_part(var partition) {
48      var result = new 'FixedIntegerArray'(2500);
49      var interp;
50      ${getinterp interp};
51      var task = interp.current_task();
52      var vector;
53      ${ pop vector, task };
54      var matrix;
55      ${ pop matrix, task };
56
57      int start = partition * (10000 / 4);
58      for (int i = 0; i < (10000 / 4); i++) {
59          int r = 0;
60          var row = matrix[start + i];
61          for (int j = 0; j < 10000; j++)
62              r += (row[j] * vector[j]);
63          result[i] = r;
64      }
65
66      var res_task = new 'Task';
67      var set_result;
68      ${get_global set_result, 'set_result'};
69      res_task.code = set_result;
70      res_task.data = result;
71      ${ push res_task, partition };
72
73      interp.schedule_proxied(res_task, matrix);
74      ${wait res_task};
75  }
76
77  function set_result(var result) {
78      var interp;
79      ${getinterp interp};
80      var task = interp.current_task();
81      var partition;
82      ${ pop partition, task };
83      var results;
84      ${get_global results, 'results'};
85
86      int start = partition * (10000 / 4);
87      for (int i = 0; i < (10000 / 4); i++)
88          results[start + i] = result[i];
89  }
90
```

```
91 # Local Variables:
92 #   mode: winxed
93 #   fill-column: 100
94 # End:
95 # vim: expandtab shiftwidth=4 ft=winxed:
```

Appendix C

chameneos.pir

An implementation of the *Chameneos* game testing thread synchronization.

```
# Copyright (C) 2012, Parrot Foundation.

.sub 'main' :main
    .local pmc colors, start_colors, at_most_two, at_most_two_waiters,
    mutex, sem_priv, first_call, a_color, b_color, chameneos, chameneo,
    code, data, number, color, dummy, count
    .local int i

    dummy = new ['Continuation'] # workaround, see TODO in Proxy
    instantiate

    count = new 'Integer'
    count = 0
    set_global 'count', count

    colors = new ['ResizableStringArray']
    colors = 3
    colors[0] = 'Blue'
    colors[1] = 'Red'
    colors[2] = 'Yellow'

    start_colors = new ['ResizableIntegerArray']
    start_colors = 4
    start_colors[0] = 2
    start_colors[1] = 0
    start_colors[2] = 1
    start_colors[3] = 0

    # init cooperation
    at_most_two_waiters = new ['ResizablePMCArray']
    at_most_two = new ['Integer']
    at_most_two = 2
    mutex       = new ['Integer']
    mutex       = 1
    sem_priv    = new ['Integer']
    sem_priv    = 0
```

```
34      first_call  = new ['Integer']
35      first_call  = 1
36      a_color     = new ['Integer']
37      a_color     = -1
38      b_color     = new ['Integer']
39      b_color     = -1
40
41      code = get_global 'chameneos_code'
42      chameneos = new ['ResizablePMCArray']
43      chameneos = 4
44      i = 0
45  init_chameneos:
46      chameneo = new ['Task']
47      chameneos[i] = chameneo
48      data = new ['FixedPMCArray']
49      data = 2
50      number = new ['Integer']
51      number = i
52      data[0] = number
53      color = new ['Integer']
54      color = start_colors[i]
55      data[1] = color
56      setattribute chameneo, 'code', code
57      setattribute chameneo, 'data', data
58      push chameneo, b_color
59      push chameneo, a_color
60      push chameneo, first_call
61      push chameneo, at_most_two
62      push chameneo, at_most_two_waiters
63      push chameneo, mutex
64      push chameneo, sem_priv
65      push chameneo, colors
66      schedule chameneo
67
68      inc i
69      if i < 4 goto init_chameneos
70
71      say "going to sleep"
72      sleep 10
73      say "woke up just in time for exit"
74      say count
75      exit 0
76  .end
77
78  .sub chameneos_code
79      .param pmc data
80      .local pmc interp, task, number, color, colors, at_most_two,
        at_most_two_waiters, mutex, sem_priv, cooperation, first_call,
        a_color, b_color, other_color
81      .local int old_color, other_color_int, color_int
82      .local string color_name
83
84      interp      = getinterp
85      task        = interp.'current_task'()
```

```
86      colors      = pop task
87      sem_priv    = pop task
88      mutex       = pop task
89      at_most_two_waiters = pop task
90      at_most_two = pop task
91      first_call  = pop task
92      a_color     = pop task
93      b_color     = pop task
94
95      number      = data[0]
96      color       = data[1]
97      color_int   = color
98      color       = new ['Integer']
99      color       = color_int
100     cooperation = get_global 'cooperation'
101
102 start:
103     color_name  = colors[color]
104     #print 'This is '
105     #print number
106     #print " and I'm "
107     #say color_name
108
109     other_color = cooperation(number, color, sem_priv, mutex,
        at_most_two, at_most_two_waiters, first_call, a_color, b_color)
110     other_color_int = other_color
111
112     color_int = color
113
114     if color_int == other_color_int goto start
115
116     color_int = 3 - color_int
117     color_int = color_int - other_color_int
118
119     color = color_int
120
121     goto start
122 .end
123
124 .sub cooperation
125     .param pmc id
126     .param pmc color
127     .param pmc sem_priv
128     .param pmc mutex
129     .param pmc at_most_two
130     .param pmc at_most_two_waiters
131     .param pmc first_call
132     .param pmc a_color
133     .param pmc b_color
134     .local pmc interp, sem_wait, sem_unlock, call_core, call_task
135     .local int other_color
136
137     interp      = getinterp
138     sem_wait    = get_global 'sem_wait'
```

```
139     sem_unlock  = get_global 'sem_unlock'
140
141     call_task = new ['Task']
142     setattribute call_task, 'data', color
143     push call_task, b_color
144     push call_task, a_color
145
146     sem_wait(mutex)
147     if a_color > -1 goto second
148         call_core = get_global 'first_call_core'
149         setattribute call_task, 'code', call_core
150         interp.'schedule_proxied'(call_task, a_color)
151         wait call_task
152
153         sem_unlock(mutex)
154         sem_wait(sem_priv)
155         other_color = b_color
156         sem_unlock(mutex)
157         goto done
158     second:
159         other_color = a_color
160
161         call_core = get_global 'second_call_core'
162         setattribute call_task, 'code', call_core
163         interp.'schedule_proxied'(call_task, b_color)
164         wait call_task
165
166         sem_unlock(sem_priv)
167 done:
168     .return(other_color)
169 .end
170
171 .sub first_call_core
172     .param pmc data
173     .local pmc interp, task, a_color, b_color
174     .local int a_color_int
175     interp = getinterp
176     task = interp.'current_task'()
177
178     a_color = pop task
179     b_color = pop task
180
181     a_color_int = data
182     a_color     = a_color_int
183     b_color     = -1
184 .end
185
186 .sub second_call_core
187     .param pmc data
188     .local pmc interp, task, b_color, a_color, count
189     .local int b_color_int
190     interp = getinterp
191     task = interp.'current_task'()
192
```

```
193     a_color = pop task
194     b_color = pop task
195
196     b_color_int = data
197     b_color     = b_color_int
198     a_color     = -1
199     count = get_global 'count'
200     inc count
201     #say count
202 .end
203
204 .sub sem_unlock
205     .param pmc sem
206     .local pmc interp, sem_unlock_task, sem_unlock_core
207
208     interp = getinterp
209     sem_unlock_core = get_global 'sem_unlock_core'
210     sem_unlock_task = new ['Task']
211     setattribute sem_unlock_task, 'code', sem_unlock_core
212     setattribute sem_unlock_task, 'data', sem
213
214     interp.'schedule_proxied'(sem_unlock_task, sem)
215 .end
216
217 .sub sem_wait
218     .param pmc sem
219     .local pmc interp, waiter, sem_wait_task, sem_wait_core
220
221     interp = getinterp
222     sem_wait_core = get_global 'sem_wait_core'
223
224     sem_wait_task = new ['Task']
225     setattribute sem_wait_task, 'code', sem_wait_core
226     setattribute sem_wait_task, 'data', sem
227     interp.'schedule_proxied'(sem_wait_task, sem)
228     wait sem_wait_task
229     returncc
230 .end
231
232 .sub sem_wait_core
233     .param pmc data
234     .local pmc sem
235     sem = data
236 test:
237     disable_preemption
238     if sem > 0 goto lock
239     enable_preemption
240     pass
241     goto test
242 lock:
243     dec sem
244     enable_preemption
245 .end
246
```

```
247 .sub sem_unlock_core
248     .param pmc data
249     .local pmc sem
250     sem = data
251     inc sem
252 .end
253
254 .sub sem_ackquire
255     .param pmc sem
256     .param pmc sem_waiters
257     .local pmc interp, waiter, sem_wait_task, sem_ackquire_core
258
259     interp = getinterp
260     sem_ackquire_core = get_global 'sem_ackquire_core'
261
262     sem_wait_task = new ['Task']
263     setattribute sem_wait_task, 'code', sem_ackquire_core
264     setattribute sem_wait_task, 'data', sem
265     push sem_wait_task, sem_waiters
266     interp.'schedule_proxied'(sem_wait_task, sem)
267     wait sem_wait_task
268     returncc
269 .end
270
271 .sub sem_ackquire_core
272     .param pmc data
273     .local pmc sem, sem_waiters, interp, task, cont
274
275     interp = getinterp
276     task = interp.'current_task'()
277     sem_waiters = pop task
278
279     disable_preemption
280     sem = data
281
282     if sem > 0 goto lock
283     cont = new ['Continuation']
284     set_label cont, lock
285     setattribute task, 'code', cont
286     push sem_waiters, task
287     enable_preemption
288     terminate
289 lock:
290     dec sem
291     enable_preemption
292 .end
293
294 .sub sem_release
295     .param pmc sem
296     .param pmc sem_waiters
297     .local pmc interp, sem_release_task, sem_release_core
298
299     interp = getinterp
300     sem_release_core = get_global 'sem_release_core'
```

```
301     sem_release_task = new ['Task']
302     setattribute sem_release_task, 'code', sem_release_core
303     setattribute sem_release_task, 'data', sem
304     push sem_release_task, sem_waiters
305
306     interp.'schedule_proxied'(sem_release_task, sem)
307 .end
308
309 .sub sem_release_core
310     .param pmc data
311     .local pmc sem, sem_waiters, interp, task, waiter
312     .local int waiters_count
313
314     interp = getinterp
315     task = interp.'current_task'()
316     sem_waiters = pop task
317
318     disable_preemption
319     sem = data
320     inc sem
321     waiters_count = sem_waiters
322     if waiters_count <= 0 goto done
323     waiter = pop sem_waiters
324     schedule_local waiter
325 done:
326     enable_preemption
327 .end
328
329 # Local Variables:
330 #   mode: pir
331 #   fill-column: 100
332 # End:
333 # vim: expandtab shiftwidth=4 ft=pir:
```

References

Literature

[1] David Flanagan. *The Ruby Programming Language*. O'Reilly Media, 2008.

[2] Lorenz Huelsbergen and Phil Winterbottom. *Very Concurrent Mark & Sweep Garbage Collection without Fine Grain Synchronization*. whitepaper. Bell Labs, Lucent Technologies, 1998. URL: http://doc.cat-v.org/inferno/concurrent_gc/concurrent_gc.pdf.

[3] Claude Kaiser and Jead-Francois Pradat-Peyre. *Chameneos, a Concurrency Game for Java, Ada and Others*. whitepaper. DEDRIC - CNAM Paris. URL: http://doc.cat-v.org/inferno/concurrent_gc/concurrent_gc.pdf.

[4] Mark Lutz. *Learning Python*. 4th ed. O'Reilly Media, 2009.

[5] *multiprocessing – Process-based "threading" interface*. URL: http://docs.python.org/library/multiprocessing.html.

[6] Yunhe Shi et al. *Virtual Machine Showdown: Stack Versus Registers*. whitepaper. University of Dublin, Trinity College and TU Wien, 2005. URL: http://static.usenix.org/events/vee05/full_papers/p153-yunhe.pdf.

[7] Arwed Starke. *Locking in OS Kernels for SMP Systems*. whitepaper. TU Berlin, 2006. URL: http://irl.cs.ucla.edu/~yingdi/paperreading/smp_locking.pdf.

[8] *Thread – manipulate threads in Perl (for old code only)*. URL: http://search.cpan.org/~nwclark/perl-5.8.8/lib/Thread.pm.

[9] *threading – Higher-level threading interface*. URL: http://docs.python.org/library/threading.html.

[10] *Threading*. URL: http://java.sun.com/docs/hotspot/threads/threads.html.

Online sources

[11] Carl Masak. 2010. URL: http://use.perl.org/use.perl.org/_masak/journal/40451.html.

[12] Allison Randal, Dan Sugalski, and Leopold Tötsch. *Perl 6 and the Parrot Project*. 2003. URL: http://www.developer.com/lang/perl/article.php/10940_3076571_3/Perl-6-and-the-Parrot-Project.htm.

[13] Nat Tuck. 2010. URL: http://parrot.org/blog/836.

[14] Larry Wall and Guido Van Rossum. *Programming Parrot*. 2001. URL: http://www.perl.com/pub/2001/04/01/parrot.htm.

[15] Bryan C. Warnock. *Implementation of Threads in Perl*. 2001. URL: http://dev.perl.org/perl6/rfc/1.html.

[16] Eric W. Weisstein. *Mandelbrot Set*. URL: http://mathworld.wolfram.com/MandelbrotSet.html.

[17] Eric W. Weisstein. *Matrix Multiplication*. URL: http://mathworld.wolfram.com/MatrixMultiplication.html.

[18] Andrew Whitworth. 2010. URL: http://wknight8111.blogspot.co.at/2010/08/gsoc-threads-chandons-results.html.