

# Distributed applications using Java threads

Anthony Jaroslav Truhlar

Manchester Visualization Centre  
University of Manchester,  
Manchester, M13 3PL.  
United Kingdom.

URL: <http://www.man.ac.uk/MVC>

January 22, 2000

## **Abstract**

There is much to be gained from an in-depth study of making existing multi-threaded programs more portable. A good example of where this is useful is in porting multi-threaded applications that have been written to use the Solaris thread API. At MVC much of the available hardware is Silicon Graphics based rather than Solaris based, and given that the Solaris threading API is one of the most prevalent, there is a strong motivation to gain information on porting existing Solaris based code to more portable threading models such as the POSIX thread API under C, and also to Java threads. This is to allow the code to be used on not only the Silicon Graphics machines, but also on other systems. The emphasis of the project has been on the production of case studies to document the steps necessary to port numerical applications to other threads models, and also the advantages and disadvantages of doing so; including performance analysis of the ported code. Much of the work has been dedicated to one study in particular, which involves the solving of Fredholm Integral Equations of the second kind, and follows on from an earlier case study at Sun Microsystems where the code on which the study is based was developed.

# 1 Acknowledgements

First of all I would like to thank the Manchester Visualization Centre for allowing me with this work placement which has been thoroughly enjoyable and has developed my skills base considerably. In particular, I would like to thank both my supervisor Dr John Brooke, and also Dr Yien Kwok for their support and guidance throughout this project, and also the other team members who were also on the summer work placement scheme.

I would also like to thank Sun Microsystems for their enlightened acceptable use policy for the SPILT and Fredholm integral equation packages. Without this policy and the very well developed code to start with, some of these issues may not have been found. The code has certainly provided a thorough and interesting case to work with. All of the code on which the ports of the Fredholm integral equation solver are based, was developed at the Solaris Migration support centre, and downloaded from their web site at:

<http://www.sun.com/workshop/threads/index.html>

Finally, I must extend my thanks to all the authors of the many technical journals and books that I have read on multi-threading, that have made the transition to multi-threaded programming much less painful.

Anthony Truhlar, September 1999

# Contents

<b>1</b>	<b>Acknowledgements</b>	<b>1</b>
<b>2</b>	<b>Introduction</b>	<b>1</b>
2.1	Assumptions . . . . .	1
<b>3</b>	<b>C-based threading models</b>	<b>2</b>
3.1	The Solaris thread API . . . . .	2
3.2	The POSIX thread API . . . . .	2
3.2.1	SPILT - The Solaris to POSIX interface Layer for threads . . . . .	2
3.2.2	Correcting the code . . . . .	3
3.2.3	New pthread features to be considered . . . . .	4
3.2.4	Compiling and using SPILT under IRIX . . . . .	4
3.2.5	Compiling and using SPILT under Solaris . . . . .	7
3.2.6	Compiling SPILT under Linux . . . . .	7
3.2.7	Comments . . . . .	7
3.3	Available POSIX thread libraries . . . . .	8
3.3.1	Cygnus pthreads for Win32 . . . . .	8
3.3.2	pthread support at MVC — where you will/won't find it . . . . .	8
3.4	Porting code to Win32 . . . . .	9
3.4.1	The sthread interface layer . . . . .	9
3.4.2	Compiling sthreads under Win32 . . . . .	9
3.4.3	Utilising the compiled library . . . . .	10
3.4.4	Unresolved issues with Win32 . . . . .	10
<b>4</b>	<b>Porting multi-threaded applications (minor case studies)</b>	<b>12</b>
4.1	Multi-threaded “Hello World!” . . . . .	12
4.1.1	Porting from Solaris to POSIX threads . . . . .	12
4.2	Producer/consumer — software race . . . . .	13
4.2.1	Porting from sthreads to POSIX threads . . . . .	14
4.2.2	Porting from sthreads to Java . . . . .	15
<b>5</b>	<b>The main study — A Fredholm Integral Equation solver</b>	<b>17</b>
5.1	The problem . . . . .	17
5.2	The portability of the single-threaded C code . . . . .	17
5.2.1	Porting this code to Java . . . . .	18

5.3	The portability of the multi-threaded C code . . . . .	36
5.3.1	Porting the code from Solaris threads to <code>pthread</code> s 'by hand' . . . . .	36
5.3.2	Compiling the Solaris code with SPILT under IRIX . . . . .	39
5.3.3	Compiling the Solaris code using <code>sthreads</code> (Windows NT) . . . . .	39
5.3.4	<code>sysconf()</code> macros — A stumbling block . . . . .	39
5.4	Porting the multi-threaded C code to Java . . . . .	40
5.4.1	Remaining issues with language differences . . . . .	41
5.4.2	Porting the threaded sections of code - potential problems . . . . .	45
5.4.3	The process - what it entails . . . . .	50
5.4.4	Multi-processor machines . . . . .	60
5.4.5	Reflection . . . . .	68
5.5	Testing the correctness of the ports . . . . .	68
5.6	Performance analysis . . . . .	80
5.6.1	Using a 40 CPU Silicon Graphics O2000 system . . . . .	81
<b>6</b>	<b>Analysis of the HPC course materials</b>	<b>88</b>
6.1	Overview of the work done . . . . .	88
6.2	Points to note . . . . .	88
6.3	The course exercises . . . . .	88
6.4	Updating of source files . . . . .	96
6.4.1	An interesting case — The matrix multiplication example . . . . .	97
6.4.2	Comments . . . . .	102
<b>7</b>	<b>Conclusion</b>	<b>103</b>
7.1	Things still to do... . . . . .	103

## List of Figures

- 1 The original producer/consumer program running under Windows NT . . . 14
- 2 The original producer/consumer program running under Windows 95 . . . 15
- 3 Initial comparison of the single threaded Fortran and C solvers and the multi-threaded C solver
- 4 Comparison of hand-ported and SPILT compiled multi-threaded solvers along with the original
- 5 Comparison of hand-ported and SPILT compiled multi-threaded solvers along with the original
- 6 Comparison of hand-ported and SPILT compiled multi-threaded solvers along with the original
- 7 Comparison of hand-ported and SPILT compiled multi-threaded solvers along with the original
- 8 Comparison of the original single-threaded Fortran-77 and C solvers against the single threaded
- 9 Comparison of the multi-threaded Java solver in green and native mode and also the multi-threaded
- 10 Comparison of the multi-threaded Java solver in green and native mode and also the multi-threaded
- 11 Comparison of the multi-threaded Java solver in green and native mode and also the multi-threaded
- 12 Comparison of the multi-threaded Java solver in green and native mode and also the multi-threaded

## **List of Tables**

## 2 Introduction

The original direction of the project was to develop programs using Java threads to run on distributed systems. The work was to be based on that done by a previous summer student at MVC — Jens Latza[1]. The previous work involved using the `sthreads`[2] interface layer to produce and port applications that would work on both shared memory parallel computers (like Manchester’s CS6400), and ordinary PCs running Windows NT. Jens then went on to produce a distributed Java application based on one of the examples in C he had been working with. My intended rôle was to test and document the work so that it could be used in workshops and demonstrations, etc., and then to extend the distributed application to allow it deal with different and potentially more complex problems and also to ensure that it runs on a wide range of platforms.

As one will have probably gathered by now, the emphasis shifted away from the distributed application. An initial investigation of the work done by Jens was performed; particularly the single and multi-threaded programs produced to run under Solaris on the CS6400 and under Windows NT on PCs. During this initial investigation, we came to the conclusion that there was much to be gained from a more in-depth investigation of getting multi-threaded C code to run on different platforms using different threading APIs, and also by porting the code to Java. There is particular interest in getting multi-threaded C code working on Silicon Graphics machines here at Manchester (using `pthreads`). This is particularly relevant to code that was written using the Solaris threading API — one of the most prevalent threading APIs (also used in teaching here at Manchester). There is a lot of material available that uses Solaris threads, and some documentation and examples of getting this code to run on other platforms would be useful. As a result, this project has been largely devoted to this side of things, with a only a brief look at the distributed Java application (which will not be covered here due to time and space constraints). Most of the material in this report is devoted to the work carried out on porting the Fredholm integral equation solver case study from Sun Microsystems, with brief coverage of the some of the smaller cases that have been covered.

### 2.1 Assumptions

It is assumed that the reader of this document is a competent programmer and is fully aware of the concept of multi-threading and the issues surrounding it. For those unfamiliar with multi-threading, materials are available from HPC web pages that give detailed and concise coverage of the topic. The URL is:

<http://www.hpctec.mcc.ac.uk/hpctec/courses/MT/MTcourse.html>

## 3 C-based threading models

This section is intended to be just a quick overview of the C threading models.

### 3.1 The Solaris thread API

The Solaris threading API is one of the most prevalent today. It is Sun Microsystems' native threading library implemented under their SYS V UNIX based operating system Solaris. The majority of the examples of multi-threaded programming that we have here at MVC are written using the Solaris threading library, included courses such as “An Introduction to Multi-threaded Programming”. It is beyond the scope of this report to go into the details of the Solaris threading API, but the book by Berg[4] is a very good starting point, along with Sun's web site[?].

### 3.2 The POSIX thread API

As one will be aware, there are a plethora of different threading libraries available today, with many manufacturers implementing their own threading models. This of course does not make multi-threaded code easily portable, and the `pthread` standard was defined to try and improve this situation. The standard is defined as POSIX 1003.1c and is commonly referred to as `threads`. `threads` have proliferated through recent UNIX releases to such an extent, that one can find a `pthread` implementation on most UNIX platforms. In the cases where `threads` aren't provided with the operating system (as in the case of HP-UX 9.0 or IRIX 6.2 for example), then one can usually find a third party implementation that can be installed on the machine such as the `proven` library[?] available from MIT (this covers many different UNIX systems), or Cygnus `threads` (for Win32).

#### 3.2.1 SPILT - The Solaris to POSIX interface Layer for threads

SPILT was a discovery I made after I had converted the multi-threaded Fredholm Integral Equation solver (section 5) “by hand” from Solaris to `threads`. SPILT is a very useful interface library that allows one to compile code written to use the Solaris thread library using the `threads` library instead. So if for example, you had a program like the aforementioned integral equation solver, and you wanted to compile it to run under `threads` then you could do so. The great advantage of this of course is that you can run code written for the Solaris threads library on machines that only possess a `threads` library. It works by redefining the Solaris thread functions in terms of `threads`, so a function like `thr_create()` would be mapped to the appropriate invocation of `pthread_create()` in the `threads` library. This is not always technically possible since some functions such as `thr_suspend()` have no `pthread` equivalents (in these cases an error will occur — the standard error code `ENOSYS` or `ENOTSUP`).

SPILT was written by Richard Marejka at Sun Microsystems and is available from their web site[?], a slightly updated version of this will also be available from my web

site[?]. It should be noted that the code has been released in the public domain and is not supported by Sun Microsystems. The library consists of the following files:

Filename	Description
README	Documentation
thread.h	Replacement main header file
synch.h	Replacement synchronisation header file
thread.c	Implementation

The two header files are provided to be used in place of the standard Solaris header files for multi-threading, whilst the c file is of course the implementation of the interface layer. My own experiences with the library have so far been good, although it was not simply a case of compiling the code straight away - there were one or two problems with the source provided that had to be fixed first. It is for this reason that I have decided to provide make available the library I used following the modifications (albeit minor ones), to save one having to make corrections. The full details of the process that I used to correct and compile the library, and subsequently Solaris multi-threaded code are outlined in this section.

### 3.2.2 Correcting the code

There are three minor errors in the implementation of the interface library (`thread.c`), they are as follows:

**Line: Problem:**

- 507 `pthread_cond_attr_t` is used instead of the correct `pthread_condattr_t`.
- 804 An illegal implicit cast from `int` to `void *` is defined, this needs to be explicitly done.
- 804 Incorrect spelling in function call — `pthread_getspecifc()` which should of course be `pthread_getspecific()`.

Having corrected these errors, the lines will read:

```
507 pthread_condattr_t attr;
804 *valuep = (void *) pthread_getspecific( key );
```

Rather than:

```
507 pthread_cond_attr_t attr;
804 *valuep = pthread_getspecifc( key );
```

As a result the code should be compilable.

### 3.2.3 New pthread features to be considered

Although not part of the POSIX 1003.1c standard that defines `pthread`, certain implementations of `pthread` now contain functions for getting and setting the concurrency level (`pthread_getconcurrency()` and `pthread_setconcurrency()` respectively). This is particularly true on operating systems that can be run on multi-processor systems such as recent releases of IRIX and Solaris (IRIX 6.5 and Solaris 2.7 both have this capability). This is of course an important update that needs to be mirrored within SPILT, which currently flags the Solaris functions `thr_getconcurrency()` and `thr_setconcurrency()` as having no `pthread` equivalents. Whilst this was the case in 1994, it is no longer the case and the library needs updating as a result.

### 3.2.4 Compiling and using SPILT under IRIX

There is the usual choice when it comes to using this library, i.e., it can either be statically or dynamically linked. Another issue that arises is whether or not one should use a shared object (known as dynamic link libraries in some circles). I took a look at all of these cases under IRIX, and found them all to work and to be equally good — the proof of this can be seen later. Anyway, I shall now show details of all of these methods. Please note that all of the details presented here are based on work carried out under IRIX 6.5 on our 40 CPU Origin 2000 system - `kilburn`. Details may well vary for other systems and configurations.

#### Statically linked without the use of shared objects

The one step needed before we do anything else is to get the object file that can then in some way be linked. To do this, we compile the code using the standard `pthread` compilation options:

```
cc -64 -D_POSIX_C_SOURCE=199506L -c thread.c -lpthread
```

The code can be compiled using the options `-n32` or `-o32` for either of the 32-bit ABIs, however, I tend to use the 64-bit ABI, and as a result, all of the work has been tested using this ABI (though should work under the 32-bit ABIs).

This is the easiest method of all — to use the newly created object file (`thread.o`) directly when compiling your code. All you have to do is to first of all compile your program as normal (e.g. this is what is done for the Solaris version of the Fredholm Integral Equation solver):

```
cc -64 -D_POSIX_C_SOURCE=199506L -c int-mt.c -lpthread -I path
```

Then, simply link the object file generated from compiling your Solaris multi-threaded code, and link it along with the object file for SPILT, to produce the new executable (this is most easily achieved through `cc`):

```
cc -64 int-mt.o thread.o -o int-mt -lpthread -lm
```

This will give you the executable you desire. Note that as far as linking is concerned, I would recommend that for an executable you let `cc` do the work, since invoking `ld` with the correct options can be a bit involved.

Please note that the path specified (indicated in sans-serif) when generating an object file for SPILT, should be the path to the include files that come with the distribution, since they define the Solaris thread library calls in terms of POSIX threads, and that is what we are interested in.

### Statically linked using Shared objects

Another alternative that may be considered to be better and more in keeping with existing thread library implementations is to generate a shared object. A shared object is an object file that in the case of static linking can be placed in-line with the rest of the code (similar to the case above except that the file is shared from a central location); or in the case of dynamic linking, is linked in at the latest possible moment as execution is about to begin — this is known as run-time or dynamic linking. The common theme of course where shared objects are concerned is that the object files are *shared* from a central location. The locations of the shared objects are known to the system through an environment variable, which is usually `$LD_LIBRARY_PATH`, although this can vary if you have differing ABIs to worry about — which is the situation we find ourselves in on the Origin 2000 systems. When the user comes to use the shared object (then actual system-level implementation of a library), they simply need to specify the library to the compiler and it does the rest.

### Statically linked shared objects

First we'll begin with the simplest type of shared object which is statically linked. These are stored centrally and then linked into the main executable file in much the same way as a `.o` file would be. To produce the shared object file, we can again use `cc` as an interface to `ld`, details are as follows:

```
cc -64 -D_POSIX_C_SOURCE=199506L -c thread.c -I path -G -all -o libspilt.so
```

Now that the shared object is compiled, we are almost ready to use it. Before we can though, it is necessary to ensure the compiler can find the library, and this is done by adding it to the relevant library path, which is usually `$LD_LIBRARY_PATH`. As one will be aware there are a number of different ABIs available under IRIX 6.5, there are the 32-bit ABIs - the old ABI (for binary compatibility with IRIX 5 systems) and the new ABI (a high performance ABI introduced under IRIX 6.2)<sup>1</sup>. In addition to the 32-bit ABIs there is a 64-bit ABI which I have been using (obtained using the `-64` flag to `cc`. One has to be aware of the different ABIs because the run time linker uses a different library path for

---

<sup>1</sup>To explicitly select one of the 32-bit ABIs the following flags are available: `-o32` selects the old 32-bit ABI, whilst `-n32` selects the new 32-bit ABI.

each ABI — the new 32-bit ABI for example has the library path `$_LD_LIBRARYN32_PATH`, whilst the 64-bit ABI uses the library path `$_LD_LIBRARY64_PATH`. This can be configured to use the new library as follows:

```
export LD_LIBRARY64_PATH=$_LD_LIBRARY64_PATH:<spilt path>
```

Now that the library path has been set up so that the shared object can be found, we can now use it. The details below show how to compile the mutli-threaded Fredholm integral equation solver (Solaris threads version):

```
cc -64 -D_POSIX_C_SOURCE=199506L -o int-mt int-mt.c -lspilt -lpthread -lm
```

Having done this, the code should compile and run as normal.

### Dynamically linked shared objects

The last type of shared object that we will look at are the dynamically linked shared objects, which are the standard way of implementing libraries on many systems (e.g. the Solaris threading library is implemented in the shared object `libthread.so`, and on `kilburn`, `pthreads` are implemented in the shared object `libpthread.so`). The process to compile a shared object is similar to that for statically linked shared objects, as one can see:

```
cc -64 -D_POSIX_C_SOURCE=199506L -c thread.c -I path -shared -all -o libspilt.so
```

As with statically linked shared objects, the path to the shared object must be added to the appropriate library path. This is done in exactly the same way, and hence the process will not be replcated here. Similarly, the process to compile an executable using a dynamically linked shared object is exactly the same as that for a statically linked shared object, therefore again, the details will not be replcated.

### Static or Dynamic?

Whether or not one is uses statically or dynamically linked shared objects is up to the individual. As stated, most libraries are implemented as dynamically linked shared objects since there is generally no appreciable difference in execution time (that can depend upon the application of course), and the binary footprints are smaller when run-time linking is employed. Both of these conditions are true for the Fredholm integral; equation solver, which has a binary footprint of the statically linked executable is 46Kb compared to 35Kb for the dynamically linked executable and 35Kb also for the original `pthreads` port (not using `SPILT` of course). As far as execution time is concerned, there is no appreciable difference between the executable when it is statically or dynamically linked or the original `pthreads` port. As one can see from the performance review section.

### 3.2.5 Compiling and using SPILT under Solaris

This has been put on hold since it isn't essential (Sun have already done a similar thing in house but the other way round to map pthreads onto Solaris threads). If it eventually does work it would be purely an academic exercise to use it (i.e. to compare performance between Solaris, POSIX and SPILT applications). There is of course no great need for such a library under Solaris since both thread models are available.

### 3.2.6 Compiling SPILT under Linux

It is possible to compile SPILT under the *newer* distributions of Linux. Most distributions released within the past three to six months should contain a new enough LinuxThreads (a native `pthreads` ) package to be able to cope with SPILT. The SPILT package is known to compile successfully on SuSE 6.2 but not on SuSE 5.3. Full details will not be presented here since it has only recently been learnt that SPILT will compile under Linux (23/09/99) and hence needs further testing, which sadly there isn't time for. The compilation procedure would however be similar to that of IRIX, apart from the fact that in Linux there is only one ABI to worry about.

### 3.2.7 Comments

From what I have seen, SPILT is a very good interface between the two threading APIs. The problem with the code as it stands on Sun's web site is that there is only documentation relating the functionality of the code rather than its use in practice, such as compilation of the interface layer for example. One of the first things I had to do was to get the library and subsequently Solaris multi-threaded code to compile with it. Hopefully this is now documented in a form that competent C programmers should be able to understand and use[?].

I have found that SPILT is a very good alternative to porting the code by hand since as has been shown, the binary footprint sizes are almost identical to those of the hand ported code (when dynamically linked), and in addition to this there is no appreciable performance overhead in using the library (as shown in the performance review). SPILT provides a fast and easy way to run Solaris multi-threaded code where direct `pthread` API equivalents are available for the Solaris thread functions used within the code. Even if one were intending to port Solaris multi-threaded code to `pthreads` by hand then using SPILT can give a quick indication of any potential problems, since it only does the same job that one would do by hand when performing a direct translation. All in all a handy package that has seemed to work well with Solaris multi-threaded code on the IRIX platform.

A potential further improvement for SPILT is to map the flag `THR_NEW_LWP` to the new UNIX 98 concurrency features for `pthreads`<sup>2</sup>. So that on those systems that support such features, code using this thread creation flag could be compiled and run on other platforms, and should not only run but perform well (since setting the concurrency level results in threads being distributed across CPUs). A very good argument for this is the

---

<sup>2</sup>This follows the recent breakthrough in getting the matrix multiplication program to perform well on IRIX which initially it was not doing, either using SPILT or with a 'by hand' port of the code.

matrix multiplication example in the HPC course “An Introduction to Multi-threaded programming” [?], which when compiled with SPILT does not perform well in terms of execution time because the flag `THR_NEW_LWP` is silently ignored<sup>3</sup> and a standard thread is created (when of course the concurrency level should have been raised as well for ideal performance).

Apart from those initial difficulties however, I have managed to compile a number of examples with the library (examples here) (results here). From this I would conclude that (conclusion here). So in comparison to converting the code by hand you are (better or worse off here).

### 3.3 Available POSIX thread libraries

Given that `pthread`s are now standardised, there are a wide variety of implementations around. Many current versions of UNIX distributions contain an implementation of `pthread`s, and for those that don't, there are implementations that sit on top of existing operating systems. Examples of operating UNIX variants that support `pthread`s include Solaris 2.6 & 2.7, IRIX 6.5, Linux, and many others. A good distribution for those systems that don't have `pthread`s support is the proven `pthread`s library available from MIT (as used by Jens Latza)[?]. `pthread`s are by definition very much tied to UNIX, and as much one isn't likely to find implementations on other platforms. Apart from the rare case stated below.

#### 3.3.1 Cygnus pthreads for Win32

One of the more obscure `pthread`s releases has been that by the cygnus project, who are working on an implementation for the Windows 32-bit API. The implementation intends to provide a high quality implementation of `pthread`s, and currently implements a great deal of the `pthread`s standard. There are of course a number of differences between UNIX and Windows (lack of compliance beyond POSIX 1 for a start), that make this task difficult. As a result, the implementation is only ever likely to be a subset of the standard, but hopefully enough to be useful. For a detailed comparison of functionality, such as the level of support for the current `pthread`s standard, and how this compares to other threading APIs, refer to section ?? of this report.

#### 3.3.2 pthread support at MVC — where you will/won't find it

As far as the systems at MVC that I have used are concerned, `pthread`s support is available on Kilburn (IRIX 6.5), and Irwell (SunOs 5.6). None of the machines in the MAN T&EC room (SG INDY workstations running IRIX 6.2) support `pthread`s. The fact that `pthread`s aren't supported on the SG workstations, cost quite a bit of time at first, since *some* of the `pthread` header files are present, but not others. It is not just a problem on the INDY work stations, since there are O2 workstations that also don't have `pthread` support. The one workstation known to have support is scooby (130.88.1.134),

<sup>3</sup>Whether or not this is silently ignored depends upon the flags specified to SPILT.

which is the system that I used for a couple of days early on at my time in MVC<sup>4</sup>. It was on this system that I was able to successfully compile some multi-threaded code written using `pthread`. The problem on the INDY workstations is that the compiler can't find certain header files (mainly `pthread.h`, because the threading library is not properly installed, and of course won't successfully compile the code, however, the problem is deeper than just missing header files, the installation is broken (it may have been partly removed). To be doubly sure that this was the case, I decided to take the compiled binary and attempt to run it on my workstation, if it was just a case of some missing header files, then the code would have run, however, it did not. The problem that prevented the code running was that the run-time linker could not find the shared object that the `pthread` library is implemented in (`libpthread.so`). The fact that the shared object could not be found, is an indication of problems with the installation of the library.

### 3.4 Porting code to Win32

I have found that porting multi-threaded code to Win32 is not quite as difficult as I'd imagined. There is already a library in existence that can be used to allow Solaris multi-threaded code to be compiled and run under Windows NT/95. In addition to this there is now a project under way to provide POSIX thread support for Win32. I have managed to get the code running using both libraries, although the latter library is very much in alpha at the moment and requires significant modifications to get the code to work.

#### 3.4.1 The `sthread` interface layer

The `sthread` library was written here at MVC by Stephan Enderlein back in February 1997, and is conceptually similar to SPILT, in so far as it takes the standard Solaris thread API calls and re-implements them in terms of another threading API (in this case the Microsoft Foundation Classes threading model rather than the POSIX threading model as in the case of SPILT). The library is implemented in C++ since MFC threads are classes, but can be used from C or C++ programs. The library is very good and implements around 3/4 of the Solaris functionality, the remainder is not technically possible. The original library was written and tested using Microsoft's Visual C++ 2.0, and I have used it under Visual C++ 6.0 (though I wouldn't image that there would be any problem in using versions in-between or in fact any future versions of the compiler). Once I had compiled the library, I had no problems using it. For a comparison of the functionality that `sthreads` provides (such as the functions from the Solaris API that have been re-implemented (and therefore can be used)), refer to section ?? of this report.

#### 3.4.2 Compiling `sthreads` under Win32

Compilation of the library isn't immediately obvious from the documentation or indeed the code, but is actually quite straightforward. Although there is no `Makefile` provided with the main package on Stephan's web site[?], the demonstration packages that are available do contain `Makefiles`. These `Makefiles` are somewhat verbose when it comes

---

<sup>4</sup>I was temporarily re-located so that some my workstation could be used by a team.

to compiling the code, and tended to produce binaries that were around 165kb. Having seen how the cygnus `pthread` library 3.3.1 applications are compiled; I experimented a little with the compilation of `sthread` code, and managed to reduce the number of included libraries by using a similar approach to the way cygnus `pthread` applications are compiled, and managed to achieve a workable and somewhat less verbose compilation command-line. So here are the details for the compilation of the library:

```
cl /W3 /MT /nologo /Yd /Zi /I. -D_WIN32_THREAD=0x400 \
  /D' '_WIN32_THREAD' ' -DSTDCALL=_stdcall -c sthread.cpp
```

### 3.4.3 Utilising the compiled library

Using the compiled library is a straightforward affair, one simply compiles the code for their program as follows (the example below is for the Win32 `sthread` port of the Fredholm Integral equation solver):

```
cl /W3 /MT /nologo /Yd /Zi /Isthread -D_WIN32_THREAD=0x400 \
  /D' '_WIN32_THREAD' ' -DSTDCALL=_stdcall -c NT_int-mt.c
```

The new object file is then linked to that of the `sthread` library as follows:

```
cl /FeNT_int-mt.exe /Zi NT_int-mt.obj sthread/sthread.obj
```

One should note that the above compilation sequence assumes that the `sthread` object file `sthread.obj` is available in the directory `sthread` which is contained within the current directory. If it is located elsewhere, this must of course be reflection in the compilation commands issued.

As one can see, compiling code that uses `sthreads` is relatively painless, and does not require a large set of commands (many given in the IDE generated `Makefile` are not entirely necessary and tend to cause the binary footprint size to be on the large size (256Kb as opposed to 148Kb using our sequence of commands).

### 3.4.4 Unresolved issues with Win32

One will no doubt have noticed that when using SPILT, I employed the use of shared objects (the UNIX term for what is effectively a Dynamic Link Library). I had intended to use DLLs under Win32 for each of the `sthread` and `pthread` implementations, but did not have time. As a result, and as one will be aware I stopped short of this using object files. This has allowed me to test the functionality of the libraries in the shortest time possible, and it was my intention to go back and provide details of their use.

Another issue to be resolved is that of compiling code using these libraries with different compilers. So far I have used Microsoft's Visual C++ compiler (version 6.0) exclusively. There are of course many other compilers available for Windows, and unlike their UNIX counterparts, tend to have differing options with each one! As one might expect, I had also hoped to at least provide compilation details for the main compilers - i.e. Visual C++, Borland C++, Borland C++ Builder and the `gcc` compiler. As with the previous point, there simply hasn't been time to do this so far, but it is something that I would like to rectify.

## 4 Porting multi-threaded applications (minor case studies)

Before moving onto the main case study of the Fredholm Integral equation solver, I looked at a number of smaller programs. Originally I was to produce a set of case studies, however, due to time constraints and certain implementation issues (particularly w.r.t to Java), it was decided that it would be better to have one complete study rather than many smaller ones. As a result, the work not completed was put to one side, so that I could concentrate on the Fredholm Integral equation solver (this was desirable since we are more interested in porting numerical applications). In each case, the current state of the port is listed.

### 4.1 Multi-threaded “Hello World!”

This is a very simple application (so simple it is hardly worth mentioning), that simply spawns a thread to print the word “hello”, and then waits for this thread to join (i.e. successfully complete its execution), before printing the word “world”. This was the first port of a multi-threaded program that I had done, it was simply to test the `pthread` availability on the machines within MVC (this was at the time where I was having problems getting code to compile - this program was used to find systems that had a working `threads` installation). As one can guess, it was chosen purely for its simplicity (it is the program from exercise 1 on the multi-threaded programming course[?]).

#### 4.1.1 Porting from Solaris to POSIX threads

The conversion process was simple, there are only three calls made to the Solaris thread library, and these occur on lines 84,95, and 104. They are easily converted to their `pthread` equivalents as I shall demonstrate. The first call, which occurs on line 84 is as follows:

```
thr_exit(0);
```

Which becomes:

```
pthread_exit(0);
```

The conversion above is the simplest kind, where it is just a case of changing the function name (since the arguments are the same). There are many functions, which although possessing similar functionality, have different arguments, and one must be aware of this. So in the case of thread creation, which we see on lines 95-98:

```
if (ir = thr_create( NULL, NULL, hello_thread, NULL, NULL, &htid) ) {
    fprintf( stderr, ‘‘thr_create: %s\n’’, strerror(ir) );
    exit(1);
}
```

This becomes:

```
if (ir = pthread_create( &htid, NULL, hello_thread, NULL ) ) {
    fprintf( stderr, 'pthread_create: %s\n', strerror(ir) );
    exit(1);
}
```

The key difference is that additional arguments in the Solaris thread creation function `thr_create()` are acquired differently under `pthreads` — via an attribute object, and hence the `pthread` thread creation function `pthread_create()` takes a reference to an attribute object should one be defined as it's second argument (for default attributes as in this case, an attribute object is not required). Therefore arguments 1, 2, and 6 from `thr_create()` would be defined using an attribute object under `pthreads` if they differed from the default values.

A similar situation exists with the Solaris thread joining function `thr_join()`, as can see below (lines 104-107):

```
if (ir = thr_join(htid, NULL, NULL)) {
    fprintf( stderr, "thr_join: %s\n", strerror(ir));
    exit(1);
}
```

This becomes:

```
if (ir = pthread_join(htid, NULL)) {
    fprintf( stderr, "pthread_join: %s\n", strerror(ir));
    exit(1);
}
```

As one can see this is just a minor difference. The second argument in the Solaris thread joining function `thr_join()` is a reference to an integer where the id of the exiting thread is to be placed. This is not supported under `pthreads`, which just leaves the id of the thread to be joined and the status reference which `pthreads` does of course support.

Well, that's it really. After those minor conversions, the source is effectively converted to use `pthreads` and should function as expected. As I said, this is a very trivial example that I used to test the availability of `pthreads` libraries. It quite nicely demonstrates the ease with which code can be ported from Solaris threads to `pthreads`.

## 4.2 Producer/consumer — software race

This program was developed by a Jens Latza back in 1997, it was his first threaded program, and used several functions from the `sthread` library. The intention of using `stthreads` being that the program could be used on both Solaris and Windows NT environments. The program is a simple form of software race that has two groups of threads - *producer* threads and *consumer* threads which operate on a synchronised variable `points`

(the maximum value of which is 1000). The *producer* threads increment the aforementioned variables, whilst the *consumer* threads decrement it. The program is driven by a basic menu system, which has 10 options, as one can see from the screen shot in figure 1 of Jens' program being run under Windows NT<sup>5</sup>.

```

Command Prompt - prodcons
Producers produce points, Consumer consume points
MENU
-----
1) Create a Consumer
2) Create a Producer
3) Toggle finished->destroy threads. Status: threads possible
4) Create a Menu-Thread
5) Suspend a Consumer (thr_kill not available on NT)
6) Suspend a Producer (thr_kill not available on NT)
7) Continue a Consumer (thr_kill not available on NT)
8) Continue a Producer (thr_kill not available on NT)
0) Destroy all Threads and quit
Number of Consumers:3, SizeofDIUSizeof:10
Number of Producers:4
Number of Suspended Consumers:1
Number of Suspended Producers:2
Number of Points:184
-

```

Figure 1: The original producer/consumer program running under Windows NT

In figure 2 one can see another screenshot of the program, this time using the same binary as before but being run under Windows 95 — showing quite nicely that the `sthreads` library is not just for NT<sup>6</sup>.

As one can see, the system allows you to create both types of the thread (a maximum of 10 of each) through options 1 and 2, toggle between allowing/disallowing threads to be created through option 3, creating menu threads through option 4, and suspending or resuming<sup>7</sup> the threads through options 5-9. The program's execution is terminated using option 0.

I basically took the program and converted it to POSIX threads and also to Java. Both of versions of the code function correctly, although there are issues with each of them that are worth noting.

#### 4.2.1 Porting from `sthreads` to POSIX threads

This was a relatively straightforward port, as with my initial example, in section 4.1.1. Since `sthreads` is purely an interface to map the Solaris function calls to those for MFC threads, one is effectively just converting from Solaris threads to `pthreads`. Jens has used NT specific code in places, but this was not a major problem. The program employs

<sup>5</sup>**Note:** The user interface to Jens' program appears unabridged.

<sup>6</sup>This is because the producer/consumer program uses the MFC threading model, since MFC is a Win32-wide API

<sup>7</sup>Jens used suspend and continue functions rather than kill, because `sthreads` does not implement the Solaris function `thr_kill()` in terms of MFC threads.

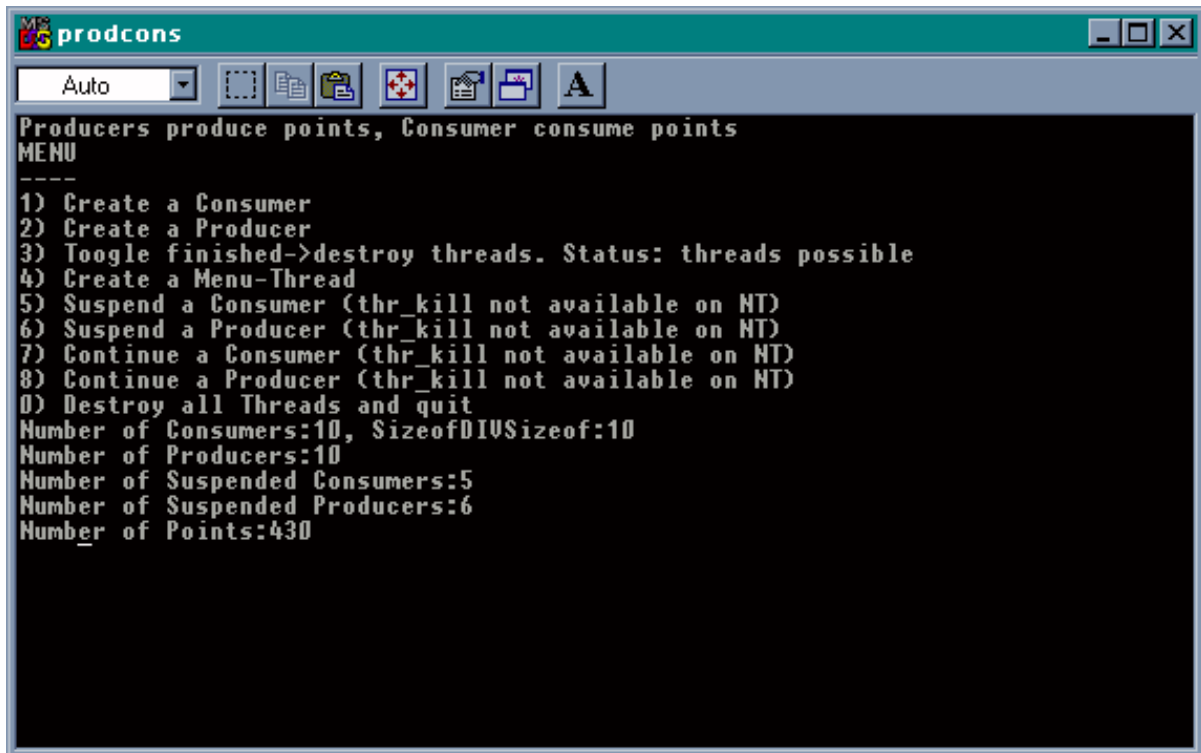


Figure 2: The original producer/consumer program running under Windows 95

a broadly convenient subset of the Solaris threading model that has equivalents in the `pthread` model, except for the suspension/resumption of threads which was a compromise in Jens' original program because the `sthread` library could not re-implement the Solaris function `thr_kill()` in terms of the MFC threading model (Stephan pointed out in his report[2] that it was only possible to send signals to processes and not to individual threads).

The conversion of the program is a relatively straightforward process and is like any other conversion of code from Solaris to `pthreads`. The first step towards converting the code is to change the specified header files to the correct ones — compilation under Win32 does not support header files such as `unistd.h` and also the synchronisation header file for Solaris threads (`synch.h`), which are now included in the main `sthread` header file.

The next stage is to go through and convert all of the Solaris thread API calls to the appropriate `pthread` API calls. As in the previous example, this is a relatively simple affair.

[Complete process details here]

#### 4.2.2 Porting from `sthreads` to Java

The conversion to Java was the first I had performed and was relatively painless, contrary to my expectations. The application is fairly primitive in so far as it requires a console interface, which brings certain portability problems due to the vastly differing array of terminal types available at the moment (for example the code relies on a system command to clear the console screen - `cls` under Win32 or `clear` under UNIX, which is largely

due to lack of support for screen clearing in C) — this is also the case in Java, and again, we are relying on being able to execute a system command (which can be rather involved in Java) to do this. Beyond this, there are of course differences with the threading model that must be overcome, as discussed in section ???. However, despite the potential difficulties, I have produced a working port that works better than the `pthread` port — i.e. in so far as all of the functionality is matched in the Java port, whereas for albeit sound technical reasons; this is not the case with the `pthread` port, as one will already be aware.

The approach employed for the producer/consumer program is the one that I would recommend in most cases; particularly where the use of threads is fairly straightforward - i.e. there aren't a large number of different functions that need to be executed by the threads, and they don't have special requirements for argument passing (i.e. a scheduler that forms part of the program that passes a variation of a set of arguments to different threads, as in the case of the Fredholm integral equation solver).

[Complete process details here]

## 5 The main study — A Fredholm Integral Equation solver

### 5.1 The problem

This work follows on from a case study conducted at Sun Microsystems in 1994, by the Solaris 2 Migration Support Centre. They had some code written in FORTRAN-77 that ran on a Honeywell 6000 series mainframe. The purpose of this code was to find approximate solutions to Fredholm Integral equations of the Second kind, by using Chebyshev polynomials to convert such integral equations into a system of linear equations for which the solution vector would be the approximate solution of  $f(x)$  - our goal (everything else is specified). This FORTRAN code was ported to C (single threaded) and then modified to use the Solaris threads library. It would not surprise anyone to find that the multi-threaded code they had created was highly efficient, and improved considerably on the single threaded C and FORTRAN code (between 5.3x and 9.2x). The main objective of their case study was to analyse the amount of effort required to port the code and then multi-thread it, and the benefit of doing so. Here at MVC we are more interested in the code than in the findings of the study, since the code is intensive enough to allow a reasonable analysis of the difficulties in porting this code to first of to `pthread`, and moreover to Java.

### 5.2 The portability of the single-threaded C code

The single threaded C code released by Sun does not appear to have any portability problems. The code uses standard features, and on all the platforms on which I have compiled and run the code, I haven't experienced any problems. Here is a list of those platforms tried:

- IRIX 6.5 (kilburn): `cc`, `gcc`
- HP-UX 9.0 (cguhpa): `cc`, `gcc`
- SunOs 5.7 (irwell): `cc`, `gcc`
- Linux 2.0.35 – SuSE 5.3 (my own PC): `cc`, `gcc`
- Windows NT (my own PC): MS Visual C++ 6.0
- Windows 95 (my own PC): Borland C++ 4.5 (`bcc`)

I expect that the code would run on other combinations of hardware without any problems. The only minor point that one must remember, is that certain compilers require the explicit specification of the mathematics library on the command line, for example on `kilburn`, the compilation line is as follows:

```
cc -o int int.c -lm
```

### 5.2.1 Porting this code to Java

Before attempting to port the multi-threaded version of the C code to Java, I ported the single threaded code. The reason behind this was really quite simple - so that I could iron out any potential problems with the porting of non thread-related functionality beforehand. This turned out to be quite a useful move since a number of interesting issues came up that I shall discuss here, along with the details of the work carried out.

My approach to porting the code, was to keep it as close to the original C as possible. I did this in a more or less top-down manner, starting with the pre-processor directives, and going through the code from there. The Java code is written using the 1.1 API (widely used at MVC), and has been tested under the Java 2 platform (there are no deprecated methods). The code will also work with minor modifications under the Java 1.0 API (the only Java 1.1 feature is the use of an inner class).

#### Where to start

The first thing to take note of is that we have to create at least one class. Java is an object-oriented language, so there is no way around this. I decided to call the class `Fe`, and declared it in the usual way. Below is an outline of the program (line numbers relate to actual code in the completed source file[?]):

```

52 public class Fe {
    // global variables (lines 54-60)
61
62 // main - kiss of life
63 // -----
64
65 public static void main(String args[]) {
66
67     new Fe();
68
69 } // end of main
70
    // class constructor (lines 71-109)
71 Fe() {
    .
    .
    .
109 } // end of constructor for class Fe
    // inner class & function declarations (lines 111-579)
580 } // end of class Fe

```

One will notice that everything apart from the instructions to include the appropriate libraries, has been included within the class `Fe`. One should note the relationship between the `main()` function in the Java port and the `main()` function in the original C code. In the Java port, the `main()` function is simply there to invoke the constructor of the

containing class, so in effect the constructor acts as a main function since the class is only ever instantiated once (when it is first run). As one can see, apart from the class structure, the containing class has an inner class (this can easily be moved to a separate file for 1.0 compliance) and the remaining functions from the existing C code (in the C code these are declared as global functions using the extern storage class specifier). This type of structure could be broadly followed for other such ports, and provides something that is as close to the original C code in structure as possible within Java.

Now that we have the building blocks for the new Java program, we can quite begin to look at specific porting issues, since it is simply a case of taking the code from the C program systematically and adding it to the skeleton outlined above.

### Pre-processor features

This section looks at the use that is made of the C pre-processor, and how this code can be converted to work with Java.

#### *Pre-processor directives*

One of the most widely used directive within the code is `#define`, which in this code is used for:

- Setting up feature test macros.
- Defining constants.
- Setting up parameterised macros.

#### *Feature test macros*

These generally inform the compiler of some specific functionality that is required, here there is only one such macro definition, which is to ensure POSIX compliance (i.e. `#define _POSIX_SOURCE`). The concept of feature test macros is unknown in Java and not needed in practice.

#### *Defining constants*

The single threaded solver program is not unlike many other C programs in its use of symbolic constants - through `#define` to define constants rather than explicitly creating a const variable. The difference being that for a given symbolic constant `foo`; all occurrences of `foo` are replaced by the defined value by the C pre-processor, so that these values exist in literal immediately prior to the compilation of the code. Java does not have a pre-processor so all instances of such symbolic constants must be replaced by a constant variable (i.e. `static final`), so using the first such symbolic constant as an example (from the original C code[?]):

```
56 #define TOLERANCE    1.0e-15          /* machine tolerance */
```

This is converted to a constant double value in the Java code:

```
55    public static final double TOLERANCE=1.0e-15;
```

This process is repeated for all symbolic constants, which occur on lines 48-53 of the original C code. The use of symbolic constants is often recommended to make code more readable, since they are defined at the start of the program and are therefore easy to change if this is necessary — this is also true for constant variables, and is of course no less true for Java; so the conversion of symbolic constants to constant variables is not a problem.

### *Parameterised macros*

Just as it is possible to define symbolic constants through `#define`, it is also possible to define parameterised macros which are single identifiers equivalent to expressions, complete statements or groups of statements. Parameterised macros are similar to functions in this respect, however, they are defined in a quite different manner and are again dealt with by the pre-processor (as in the case of symbolic constants) rather than the compiler proper. Parameterised macros are a popular construct in C programs, particularly where performance is at a premium. An example of such a macro is shown below (taken from the single threaded C code):

```
66    #define VALUE(i,n)      (cos(((i)*M_PI)/(n)))
```

Here we see a definition for the macro `VALUE` which takes the parameters `i` and `n`. The macro is treated like a function call from within the rest of the program, for example:

```
208                double x      = RE( VALUE( i, n ), lb, ub );
```

Here we see what appears to be a call to `VALUE` within a call to `RE` (this is also a parameterised macro!). These are not function calls however, and are not treated as such; instead when the program is run through the pre-processor, each instance of the macro is replaced by its definition, with the parameters appropriately substituted, so:

```
a = VALUE(b[i], 2);
```

Would become:

```
a = cos( ( b[i] * M_PI ) / 2 );
```

This is a form of function in-lining, where the body of a function is placed in the place of a call to that function (to do this with standard functions requires compiler optimisation, whereas parameterised macros will **always** behave in this manner). The key benefit of this is of course that the overhead of a function call is removed, thus decreasing the execution time of the code. This is particularly worthwhile for functions that must be frequently called (where the calling overhead is most likely to make an appreciable difference).

Having established what a parameterised macro is, and the benefits of its use, we are now faced with the question - "How do we get this working in Java?". The approach I took, was to convert the macros to functions, so for the `VALUE` macro we get the following function in Java (specified in global scope):

```

133 // VALUE(i,n)
134 double VALUE( double i, double n ) {
135
136     return( Math.cos ( ( i * M_PI ) / n ) );
137
138 }

```

This process is repeated for the other parameterised macros (taking care to get the parentheses correct!). This just leaves the issue of performance which one might realistically expect to suffer from the overhead of function calls that we have re-introduced. However, as my work has shown, modern Java bytecode compilers automatically in-line basic functions. After completing the port, I was quite concerned about performance, and decided to modify a copy of the ported code so that all the functions were in-line (in effect I in-lined the code by hand). My 'manual' in-lining was restricted to the functions resulting from the conversion of the parametrised macros. Having in-lined the code, I then performed some comparative timings on the two versions of the code, and found there to be no appreciable difference in execution time (beyond the normal fluctuations one would expect) - hence suggesting that functions are automatically inlined. A more in-depth analysis of this can be found in the performance section of this document. [Graphs/tables and java-Xrunhprof data],

### *Library inclusion*

These are again dealt with by the pre-processor in C using the `#include` directive. The single threaded version of the code uses the standard C I/O library (`stdio.h`) and the mathematics library (`math.h`). For the Java version we need to use the I/O package (`java.lang.io.*`), along with the standard Java language packages (`java.lang.*`) and the Java text package for number formats (`java.lang.text.*`). The Java packages are loaded using the `import` command, and the instructions can be found on lines 45-47 of the Java code.

### **C Language features**

In the previous section, we have seen the issues related to the use of the C pre-processor, and how this code can be ported to Java. In this section, we shall see how to get around the problem of features within the C language itself, the can cause us problems when moving code to Java.

### *Type definitions*

One of the nicer features in C that makes code more readable is the ability to define types via the `typedef` construct. This allows users to name their types more intuitively or to create new objects. In our code, the `typedef` construct is used in both of these ways. An example of the first case is the definition of a two-dimensional array of `double` values, on line 58 of the original C source:

```
71 typedef double MATRIX[N][N];
```

Our new `MATRIX` type can then be easily used by (for example – taken from the start of `main()` in the original C code):

```
107         MATRIX  a, A;
```

The above command produces two matrices named as specified of dimension  $N \times N$ . As one can clearly see, this is a convenient short hand for:

```
double  a[N][N];
double  A[N][N];
```

The problem we are presented with when moving the code to Java is that Java does not support direct aliases for types, and hence forces us to use the longer form with slightly differing syntax to that of C. The code above both defines the matrices within the scope of `main()` and allocates  $N \times N$  `double` size locations in memory for each of the matrices to use. The Java syntax differs slightly, as one can see in the Java equivalent of line 84:

```
75     double a[] [] = new double[N][N];
76     double A[] [] = new double[N][N];
```

The reason for the differing syntax is to appropriately fit arrays into Java’s paradigm of “everything is an object”. Looking at line 65, the left hand side of the statement declares a reference (an implicit pointer) to an array object, however, this array does not yet exist, so the reference is `null` until the array object is instantiated. This is of course where the right hand side of the expression comes in, since this instantiates a new array object of the appropriate size and dimension (i.e.  $N \times N$ ). This effectively leaves `a` as a reference to the new array object.

The same conversion procedure was also followed for the other `typedef` within the single threaded C code for `VECTOR` (a one-dimensional array). The same method should be employed for any occurrence of `typedef` for aliasing - i.e. the aliasing is effectively removed forcing the use of the standard types in any subsequent declarations involving the `typedef` in question. So all occurrences of `MATRIX <var>` will be replaced by `double <var>[] []`.

As mentioned earlier, the second use of `typedef` is to create new objects. In the single threaded C code, there is one such use of `typedef` to define a structure containing point values. The declaration appears in the original C code as follows:

```
74  typedef struct {
75      double x;
76      double y;
77      double z;
78  } POINT;
```

Since Java is an object-oriented language, it has its own constructs for the declaration of objects. To express such an object in Java, all we need to do is define it as a class (a C programmer could look on a class as a structure that also has methods to operate on

the data within the structure). In the final version of Java code, I used an inner class which is part of the Java API as of version 1.1 of Sun's Java SDK; although it is perfectly acceptable to define it as a standard class to allow use under a Java 1.0 virtual machine (though this would require placing the declaration in a file called `<classname>.java` rather than in the main file). Here is the definition from the final version of the Java code:

```

115     public class POINT {
116
117         double x;
118         double y;
119         double z;
120
121     }
```

The use and declaration of `POINT` object also has differing syntax to C, and I shall go over this in more depth in the section on pointers, since this is a particularly good example of Java's reference (implicit pointer) system.

### ***Storage class specifiers***

The C language has four types of storage class specifier, they are *automatic*, *static*, *external* and *register*. Storage class specifiers are used to specify the scope and type of storage that a given variable can have.

Automatic variables are local variables (i.e. they are declared within a function and local to the function in which they are declared) and can be declared with the keyword `auto` preceding the declaration, although this is usually omitted, and one need not worry about this when porting code. If the keyword `auto` is used, simply remove it.

Static variables are variables that behave similarly to automatic variables in terms of scope, but maintain their value throughout the lifetime of the program, so should a function be re-entered, the static variables will still contain their previously assigned values. There is no equivalent of this in Java, so variables that need retain their values between different invocations of a given function will need to be declared outside the given function — thankfully this is not a problem for us since non of thw single threaded code uses static variables. Java does have a `static` keyword, but it's use is entirely different, in Java `static` is used to differentiate between static and instance methods on objects. One point to note however, is that the `static` keyword can also be applied to functions; particularly function prototypes, and is used to indicate that the function is global (also implied by it's physical location within the program) and defined within the *current* file. This is not necessary in Java, since the Java ocmpiler does not forbid forward references in the same way the C compiler does, thus eradicating the need for function prototypes. To port such a function to Java, one simply needs to remove the function prototype and the static keyword from the function definition, and provided it is within scope (i.e. in the containing class or otherwise) things should work as expected.

The `extern` keyword is used to set the scope of a variable to global. So when a variable is defined as external, it becomes visible globally from that point and for the remainder of the program. The keyword is not normally needed since the variables and their storage class will normally be identified from their position within the program (in fact many C compilers forbid the use of `extern` for external variable declarations). Another use of the keyword however, is for function prototypes that can be defined within another file. As explained earlier it is necessary in C to declare a function prototype if a call is made to the function before it is physically defined. Whilst this forces the use of such prototypes for certain functions when the definitions occur within that file, it becomes imperative when the function is defined *externally* in another file, hence the use of the `extern` keyword. One should note that the use of `extern` and `static` are freely interchangeable for function prototypes relating to functions defined within the same file as the prototype. This is the situation that we find ourselves in with the single threaded C solver program. The solution to the problem is the same as that for `static` function prototypes, i.e. that the function prototypes are removed (so lines 66-79 were deleted), and the keyword `extern` is removed if used in the definition of the function (it is not in our situation). The `static` and `extern` keywords can also be applied to variables of course, and to convert such declarations to Java, one simply removes the keywords.

The `register` storage class is used for using registers within the CPU rather than main memory to store values. The use of the `register` storage class does not necessarily guarantee that the value will be stored within a register, only if there is enough register space will this happen. The use of registers is of course beneficial for large numerical operations that need to be performed quickly, since the data does not need to be transferred from memory, it is already within the registers where the operations are normally carried out — the only difference being that instead of fetching the data from main memory beforehand it is already there. This is of course very low-level functionality, and given the general abstraction from the hardware of Java virtual machines, this functionality is not found within the Java language. This is not a major grievance however, since `register` variables are very similar to `auto` (automatic) variables apart from the data being stored in registers (the scope and visibility of is the same). Therefore when porting C code to Java, one simply removes the `register` keyword, as in the following example from the C solver (from the function `MatrixLUdeterminant()`):

```
512      register int    i        = 0;
```

Which becomes the following in Java:

```
512      int    i        = 0;
```

The process is then repeated for any other occurrence of `register` (there is only one other).

### *Pointers*

This is one area, where one might expect to have the most trouble since Java is reported not to have pointers. This is quite a common misconception, since although Java does not have explicit pointers with all that they entail (dereferencing, etc.), it does have an implicit pointer system, the key difference of course is that the Java programmer has no control over pointers whereas the C programmer has a great deal of control.

In Java there are two ways of passing arguments to functions, by value and by reference. Each method applies in *specific* situations, unlike in C where something that can be passed by value can also be passed by reference. For Java, primitive types such as `int`, `double`, etc. are **always** passed by value, where as **all** objects are **always** passed by reference. This is particularly useful for the Java version of our solver program, because we can quite happily convert the `_exchange` parameterised macro to a function and the element swapping will still work, since arrays are passed by reference. So:

```
67 #define _exchange(a,b) double _T = (a); (a)=(b); (b)=_T;
```

Becomes:

```
154 // _exchange(a,b)
155 void _exchange( double a, double b ) {
156
157     double _T = a;
158     a = b;
159     b = _T;
160
161     return;
162 }
163 }
```

As with other basic functions, this will also be in-lined (when I performed the manual in-lining, this is one of the functions that I in-lined manually and there was no appreciable performance difference). This is the one area where one could easily go to a lot of work, by placing such instructions in-line when the reference-passing model makes this unnecessary.

### A caveat...

As one will remember from the example of the array declaration earlier, Java has creates a null reference to an object, which is then set to the address of that object upon instantiation. We now reach an important issue that can catch out programmers with a background in C/C++, that of creating arrays of objects. Part of the single threaded solver program in C involves the setting up of an array of `POINT` structures, this happens in `main()`, and is as follows:

```
108     POINT    pt[M];
```

In this single instruction, an array of `POINT` structures of size `M` is created and is ready for use. In our Java solver, `POINT` is now an object and is declared like this:

```
77     POINT pt[] = new POINT[M];
```

The instruction above is a standard Java array declaration, in that `pt` is a reference to an array of `POINT` objects with `M` elements. There is however, a caveat, whilst `pt` is an array of `POINT` objects, everything in Java is handled by references, so what we actually have right now, is an array of references to `POINT` objects. Since none of the `POINT` objects have yet been instantiated, these references are currently `null`. This is of course where the common beginner mistake is; if one does not understand properly how Java's referencing model works, then one can naïvely do something like (taken from the function `Points`):

```
313             pt[i].x = lower + i * dx;
```

Without first instantiating the objects, this causes a null pointer exception to be thrown, and subsequently the termination of the program's execution. This is one of my early mistakes, that was somewhat annoying. So, whenever an array of objects is declared, all of the objects with the array must be instantiated before the array is used; for the array `pt` this code will suffice:

```
83     // Create all point objects within the array
84     for( int count = 0; count < M; ++count )
85         pt[count] = new POINT();
```

This process would be (and has been in the finished Java code) repeated for all other arrays of objects.

### Code already using pointers

There is of course a lot of C code that uses pointers in order to get the best performance. The single threaded solver program is certainly no exception to this, and uses pointers in many forms, such as pointers to arrays, file streams and also functions. First of all, looking at the code that uses pointers to arrays, for example, the following loop performs an assignment on the `x` and `y` members of each `POINT` structure in the array (in the function `Points`):

```
303     for ( ; i < m; ++i, ++pt ) {
304         pt->x    = lower + i * dx;
305         pt->y    = ChebyshevEval( pt->x, a, n, lower, upper );
306     }
```

Since we can't use pointer arithmetic, we will have to rely on indexing, which means that the prefix incrementing of `pt` can be taken out, and each occurrence of `pt` in the loop body can then have the index along with the subscript operator (`[]`) appended to it. The next step is to change the C member dereferencing operator (`->`) to the member access operator (`.`), giving us:

```
312     for ( ; i < m; ++i ) {
313         pt[i].x = lower + i * dx;
314         pt[i].y = ChebyshevEval( pt[i].x, a, n, lower, upper );
315     }
```

In the single threaded solver, this is the only occurrence of member dereferencing operator, although the same method could be applied to other such occurrences.

The next type of pointer used is the file stream pointer `FILE*` which is used for passing a pointer to the output functions `PointsWrite()`, `VectorWrite()` and `MatrixWrite()`. The benefit of using pointers to file streams as opposed to the standard I/O functions that output to the standard output stream only; is of course the output can be easily re-directed (to a file for example). One can have similar functionality in Java, although this must be done using a `printstream` (which can be set up to output to a file). In order to move this functionality to Java, we must first of all change the references to the type `FILE *` - a pointer to a file stream; to the Java type `Printstream` as one can see in the following example for the function `PointWrite()`:

```
int PointWrite( FILE *fp, char *header, POINT pt[M], int n )
```

When converted to Java, the function signature becomes:

```
int PointWrite( PrintStream ps, String header, POINT pt[], int n )
```

As one can clearly see, there are differences other than the change from `FILE *` to `PrintStream`. These are handled in the *miscellaneous* section below. The change of type that one can see in the function signature above, must of course be carried through for all of the affected function signatures. In addition to this, the actual pointer passed in to the function needs to be changed, since `stdout` (a predefined pointer to the system's standard output stream) is passed in each time these functions are called, as one can see from the example below:

```
132      PointWrite( stdout, "Approximate f(x)", pt, M );
```

For the Java version, the `PrintStream System.out` should replace `stdout`:

```
105      PointWrite( System.out, "Approximate f(x)", pt, M );
```

As one can see, the changes required are straightforward for the function signatures and their calls. This is of course not the only change required for dealing with output, since the output functions themselves, do of course differ between C and Java. This however, is dealt with in the section on formatted output.

The final type of pointer that is used within the code is the function pointer. Function pointers are passed to the functions `ChebyshevCoeff()` and `Chebyshev2Coeff()`, Function pointers can be very useful when you have a set of functions and want to perform different operations on the same of data, or to execute functions within separate threads of control. Close inspection of the single threaded C code reveals that the functions passed in to each of the aforementioned functions are of the same type, i.e. a pointer to `g()` is passed to `ChebyshevCoeff()` and a pointer to `K()` is passed to `Chebyshev2Coeff()`. At the time of the port, my firm impression was that Java did not have any equivalent to function pointers (I have since found evidence to the contrary - more on this in the section on multi-threaded Java port), and therefore saw no reason to keep these function pointers. As a result they were removed. So the function signatures for the host functions were changed, here is `ChebyshevCoeff()` as an example:

```

153         void
154     ChebyshevCoeff( VECTOR a, int n, double lower, double upper, double (*f)( double ) )

```

This simply becomes the following in Java:

```

166     void ChebyshevCoeff( double[] a, int n, double lower, double upper )

```

As one can see, this was simply a case of removing the fifth argument, the same is true of calling `ChebyshevCoeff()`. Of course, we also have a similarly simple modification to make when the guest functions are called, we simply replace the dereferencing of the function pointer for the actual function call, here is what happens within our host function `ChebyshevCoeff()`:

```

167         b[i]     = (*f)( p );

```

Which becomes the following in Java:

```

180         b[i]     = g( p );

```

The conversion was a relatively simple one for the single threaded solver program, and the removal of function pointers would have been justified even with similar functionality being available in Java. The exact details on the use of function pointers, and when their use is appropriate are covered in the section on the multi-threaded Java port, since function pointers were much more useful in that case than this. In fact when one seriously looks at the code, it is difficult to see why function pointers were used at all in the single threaded code, since the functions were within scope, and in both cases, each of `ChebyshevCoeff()` and `Chebyshev2Coeff()` only takes one function — perhaps if pointers to different functions had been passed to each of these functions then it would have made more sense. As it is however, there is no clear advantage in the current design.

### *Formatted output*

Getting the correct output from the Java version of the code proved to be one of the most involved areas of the porting process. The output facilities in Java differ to those in C, for example Java does not have an easy equivalent of the `printf()` from C's standard I/O library (`stdio.h`). In our code, `fprintf()` is used since we are working with pointers to file streams that are passed in to the output functions (as reported earlier). First of all we must convert all references to `fprintf()` to the appropriate `print()` and `println()` methods of the `PrintStream` object that we pass in (a key difference in execution model). So that one can see these changes, here is the function `PointWrite()` from the original C code:

```

570         int
571     PointWrite( FILE *fp, char *header, POINT pt[M], int n ) {
572         int     i     = 0;
573
574         fprintf( fp, "%s:\n\n", header );

```

```

575
576     while ( i < n ) {
577         fprintf( fp, "\t[%3d]:\t%10.6f\t%10.6f\n", i, pt->x, pt->y );
578         ( ++i, ++pt );
579     }
580
581     fputc( '\n', fp );
582
583     return( ferror( fp ) );
584 }

```

Below is the very first Java version of the function, before number formatting was introduced, which should demonstrate the basic changes needed to get output:

```

564     int PointWrite( PrintStream ps, String header, POINT pt[], int n ) {
565         int i = 0;
566
567         ps.println( header + ":\n" );
568
569         while ( i < n ) {
570             ps.println( "\t[ " + i + "]:\t" + pt[i].x + "\t" + pt[i].y );
571             ++i;
572         }
573
574         ps.println("");
575
576         return( 0 );
577
578     }

```

Before looking at the body of the function, the first thing one will notice is that the second argument has been converted from type `char *` in C to the Java type `String`. This is a conversion that is perfectly acceptable, since we do not need to manipulate the strings in any way (they were passed in effectively as strings in the C code anyway). If the strings had needed modifying then the conversion would have been somewhat more involved.

Moving on to the body of the function, one can see that there is no need for use of the newline (`\n`) character in the Java version, since `fprintf()` or `fputc()` statements containing a newline character are converted to invocations of the `println()` rather than `print()` methods of the print stream object passed to the function. Everything else is more or less as one would expect, the pointer arithmetic has been removed, and currently there is no number formatting within the output statements. The return statement at the end of the function returns zero, since error handling is performed differently in Java than in C. Rather than using a status value that is then returned, Java will throw an exception or error if there are problems (which usually results in the termination of the program if uncaught), hence it is acceptable to return 0 here.

Without number formatting, the conversion process is simple, and one simply places the variables to be output, between the appropriate text concatenating them together (so they appear in the same order as within the format string). The big problem with this

of course is that the numerical output isn't formatted, which for this type of application can lead to unhelpful output. Here is a fragment of the output from the first Java version of the solver for a matrix of size 30:

Chebyshev Coefficients for f(x):

```
[ 0]:  4.694275966146739E-16
[ 1]: -1.133648177811761
[ 2]:  1.1321714830427357E-14
[ 3]:  0.13807177658719513
[ 4]:  1.4045932117003369E-15
[ 5]: -0.0044907142465551025
[ 6]: -4.1986840427304116E-16
[ 7]:  6.770127584101668E-5
[ 8]: -7.623467240675778E-16
[ 9]: -5.891295333913212E-7
[10]:  1.1793982061321982E-16
[11]:  3.3380607302254085E-9
[12]: -7.745925552412272E-16
[13]: -1.3297096698836365E-11
[14]: -3.20020194799865E-16
[15]:  3.8426451873922516E-14
[16]: -6.7189203572299E-17
[17]: -8.08087017797333E-16
[18]:  6.012185276056793E-16
[19]:  1.3251075440035835E-16
[20]:  7.844203052735244E-16
[21]:  9.257705873796898E-16
[22]:  6.282210501370723E-16
[23]: -2.555655746368473E-16
[24]: -6.930697775034918E-16
[25]: -1.68854162899382E-15
[26]:  7.207096060514307E-16
[27]: -1.1955762340210452E-15
[28]:  4.2338683361658783E-16
[29]:  2.46779089845843E-16
```

Approximate f(x):

```
[ 0]:  0.0      1.0000000000000273
[ 1]:  0.3141592653589793    0.9510565162951653
[ 2]:  0.6283185307179586    0.8090169943749558
[ 3]:  0.9424777960769379    0.5877852522924739
[ 4]:  1.2566370614359172    0.30901699437494407
[ 5]:  1.5707963267948966    -1.1865285548941327E-14
[ 6]:  1.8849555921538759    -0.30901699437496066
[ 7]:  2.199114857512855     -0.5877852522924855
[ 8]:  2.5132741228718345    -0.8090169943749626
[ 9]:  2.827433388230814     -0.9510565162951564
[10]:  3.141592653589793     -1.0000000000000007
```

As one can see, the output is something of an eyesore, unlike the properly formatted and much more readable output from the original C version of the program — here is the same fragment of the output, this time from the single threaded C program:

Chebyshev Coefficients for  $f(x)$ :

```
[ 0]: 0.000000
[ 1]: -1.133648
[ 2]: 0.000000
[ 3]: 0.138072
[ 4]: 0.000000
[ 5]: -0.004491
[ 6]: 0.000000
[ 7]: 0.000068
[ 8]: 0.000000
[ 9]: -0.000001
[10]: 0.000000
[11]: 0.000000
[12]: 0.000000
[13]: 0.000000
[14]: 0.000000
[15]: 0.000000
[16]: 0.000000
[17]: 0.000000
[18]: 0.000000
[19]: 0.000000
[20]: 0.000000
[21]: 0.000000
[22]: 0.000000
[23]: 0.000000
[24]: 0.000000
[25]: 0.000000
[26]: 0.000000
[27]: 0.000000
[28]: 0.000000
[29]: 0.000000
```

Approximate  $f(x)$ :

```
[ 0]: 0.000000      1.000000
[ 1]: 0.314159      0.951057
[ 2]: 0.628319      0.809017
[ 3]: 0.942478      0.587785
[ 4]: 1.256637      0.309017
[ 5]: 1.570796      0.000000
[ 6]: 1.884956     -0.309017
[ 7]: 2.199115     -0.587785
[ 8]: 2.513274     -0.809017
[ 9]: 2.827433     -0.951057
[10]: 3.141593     -1.000000
```

As one can see, the output is much better. This is what we are aiming for in the single threaded Java program. Both of the outputs are representations of the same numbers, the single threaded C program formats the output in the `printf()` format string `%10.6f` whereas Java is printing the values in it's default manner which is in scientific form (equivalent to `%.17G`). Although Java does not have a `printf()` style formatting system, it does have a number formatting system of it's own, but this is designed for formatting numbers such as currency, percentages, etc. in a locale-dependant manner. In order to format numbers in the manner we desire, it is necessary to use a `DecimalFormat` object.

`DecimalFormat` is a concrete subclass of the `NumberFormat` class (an abstract base class for formatting numbers which is itself derived from the `Format` class), and is part of the `java.text.*` package. The `DecimalFormat` class gives one a reasonable degree of control in so over the format of the numbers in so far as it is possible to set the number of digits before and after the decimal point, characters used to prefix the number and so on.

### Getting started

For the single threaded C program, there are two different output formats used one is `%10.6f` for the double precision values, whilst the other is `%3d`, which is used for the integer values. The string `%10.6f` specifies that floating point values will be displayed to six decimal places in a minimum field width of 10 characters. The minimum field width is the minimum number of characters that can be displayed for that particular output value, if the number of characters in the output value is less than the minimum field width, then an appropriate number of leading blanks are prepended to that output value. The second format string (`%3d`) specifies that integer values will be display in a maximum field width of 3 characetsrs.

In Java, formatted output has to be done through the various forms of `Format` object and the `format()` method of an instance of that object. Therefore we will need to create two `DecimalFormat` objects, one for each of the format strings. Once declared, the `format()` method is then invoked on the instances we have created with the value to be formatted passed at the time of invocation (this would be done from within a `print()` or `println()` statement). Here are the declarations for the `DecimalFormat` objects we need:

```
59     public static final DecimalFormat double_fmt = new DecimalFormat(" ##0.000000;##0.000000");
60     public static final DecimalFormat integer_fmt = new DecimalFormat(" ##0;##0");
```

The format is specified by passing a string into the new object's constructor. The first line is setting up a `DecimalFormat` object for our double precision floating point values, and specifies formats for both positive and negative values (seperated by a semicolon) — these are the same bar the minus sign. The `#`'s and `0`'s represent digits, where `#` is used to represent digits that can but do not have to be displayed, whilst `0` is used to represent digits that **must** be displayed. Given this information, one can clearly see that the new `DecimalFormat double_fmt` allows for up to 10 characters, with an output precision of 6 decimal places (the 6 digits after the decimal point always being displayed). Similarly on the second line, one can clearly see that our new `DecimalFormat integer_fmt` allows up to three digits of an integer value to be displayed. Whilst we have sucessfully set up the precision, there is of course a caveat; that being that whilst we are able to specify the meximum number of digits and whether or not they should be displayed (for a given digit position), formatted values that are shorter than our desired character field width, *do not* have an appropriate number of blanks prepended as is the case with the `printf()` family of functions in C. This of course means that whilst we can specify the precision of a value when displayed, `DecimalFormat` objects alone are not enough to ensure character field widths which can be important for numerical output such as in this case. The closest you

can get to `printf()` is by using the `MessageFormat` class, which along with knowledge of number formatting explained here, may allow one a little extra control over the output - potentially at the expense of considerable effort in understanding the rather arcane API though. For details of this API I would suggest that you consult the Java platform SDK documentation — the `MessageFormat` class is part of the `java.text.*` package[?].

Having defined our new `DecimalFormat` objects, it is easy to use them, one simply invokes (with the number to be formatted) the `format()` method of the correct object. This can be clearly seen below, where the final version of the `PointWrite()` function is displayed:

```

564     int PointWrite( PrintStream ps, String header, POINT pt[], int n ) {
565         int      i      = 0;
566
567         ps.println( header + ":\n" );
568
569         while ( i < n ) {
570             ps.println( "\t[" + integer_fmt.format(i) + "]:\t" + double_fmt.format(pt[i]
571                 ++i;
572         }
573
574         ps.println("");
575
576         return( 0 );
577
578     }

```

The output is now considerably improved and close to that of the C (although not the same due to the field width problems specified earlier), as one can see from the output fragment shown below (this is the same fragment as before except from the final version):

Chebyshev Coefficients for  $f(x)$ :

```

[ 0]:    0.000000
[ 1]:   -1.133648
[ 2]:    0.000000
[ 3]:    0.138072
[ 4]:    0.000000
[ 5]:   -0.004491
[ 6]:    0.000000
[ 7]:    0.000068
[ 8]:    0.000000
[ 9]:    0.000000
[10]:    0.000000
[11]:    0.000000
[12]:    0.000000
[13]:    0.000000
[14]:    0.000000
[15]:    0.000000
[16]:    0.000000
[17]:    0.000000
[18]:    0.000000
[19]:    0.000000

```

```

[ 20]: 0.000000
[ 21]: 0.000000
[ 22]: 0.000000
[ 23]: 0.000000
[ 24]: 0.000000
[ 25]: 0.000000
[ 26]: 0.000000
[ 27]: 0.000000
[ 28]: 0.000000
[ 29]: 0.000000

```

Approximate  $f(x)$ :

```

[ 0]: 0.000000      1.000000
[ 1]: 0.314159      0.951057
[ 2]: 0.628319      0.809017
[ 3]: 0.942478      0.587785
[ 4]: 1.256637      0.309017
[ 5]: 1.570796      0.000000
[ 6]: 1.884956     -0.309017
[ 7]: 2.199115     -0.587785
[ 8]: 2.513274     -0.809017
[ 9]: 2.827433     -0.951057
[10]: 3.141593     -1.000000

```

Given the quality of output, pursuit of further improvement of the output was thought to be one of those tasks well within the law of diminishing returns, and hence shelved. After all, there were more important issues at stake such as threading, which were of more interest. If one were in the pursuit of an exact match in output terms, then the `MessageFormat` class may provide a solution, however, it is doubtful that even a combination of both `DecimalFormat` and `MessageFormat` objects would be enough to deal with the problem of minimum field widths. One easy solution would however, be to use output formatting package defined in the book *CoreJava*, which takes `printf()` format strings and appropriately formats the output.

### *Automatic type casting*

Type casting is an area where Java is much more pedantic than C. Implicit type-casting is generally not allowed, very unlike in C. In the single threaded C version of the solver program, the type casting issues seem to be related to expression evaluation. For example, the following `while` statement is acceptable in C (taken from the single threaded C solver):

```

249     while ( n-- ) {
250         c2      = c1;
251         c1      = c0;
252         c0      = mul * c1 - c2 + a[n];
253     }

```

This is not acceptable in Java however, because it relies on the evaluation of the expression being implicitly cast to a `boolean` value. In C this is done by assuming that values

greater than zero are true and all other values are false. Java will not allow typecasting to `boolean` however, therefore, the `while` loop is correctly expressed as follows:

```

260         while ( n-- != 0 )
261             c2      = c1;
262             c1      = c0;
263             c0      = mul * c1 - c2 + a[n];
264

```

Here we satisfy both the need to decrement the loop counter `n` and also test whether or not it has reached 0. All similar situations must also have a test condition added, consider:

```

172             double sum      = 0.5 * ( b[0] + ( ( i % 2 )? -b[n] : b[n] ) );

```

This again is a problem because it does not test for 1 or 0, this is rectified as follows:

```

185             double sum      = 0.5 * ( b[0] + ( ( i % 2 != 0 )? -b[n] : b[n] ) );

```

It is a simple conversion to make, and very necessary. However, one should be careful to note the use of `!= 0` rather than `== 1`, since in both examples a test for just `== 1` is inappropriate since we intend every value other than 0 to indicate success (to mimic the behaviour in C).

### *Mathematics functions*

There is one minor difference between C and Java in this respect that must be observed at all times, that being that all calls to functions from the mathematics library must be prepended by the class name `Math` since all the methods within the `Math` class are static rather than instance methods (hence the requirement for the class name for invocation). As an example, the following line from the original single threaded C program:

```

142         return( ( 1 + M_PI / 2 ) * cos( x ) );

```

Would become the following in Java:

```

143         return( ( 1 + ( M_PI / 2 ) ) * Math.cos( x ) );

```

In addition to this, certain functions such as `abs()` have different implementations in C depending upon the type of argument they take, for example our code uses the `fabs()` function for finding the absolute value of double precision floating point numbers. Java, being an object-oriented language does not have separate functions to perform the same task depending on the type, instead it uses overloaded functions. Overloaded functions allow the use of one function to perform the same task on different data types, so Java has one `abs()` function that can find the absolute value of integer, and single/double precision floating point numbers alike. Hence converting such code to Java is simple, one simply renames the function call. As an example, the following line from the original single-threaded C program:

```
390                                     t      += fabs( a[i][j] );
```

Would become:

```
395                                     t      += Math.abs( a[i][j] );
```

### 5.3 The portability of the multi-threaded C code

When the C code was multi-threaded at Sun Microsystems, they used the Sun-Solaris threading library to do this - a natural choice. This of course poses portability problems since Solaris threads are native to the Solaris operating system and hence not supported on other systems, except through interface layers, such as those discussed in section 3 of this report. I have looked at a number of approaches to using the code on other platforms, from direct conversion 'by hand' as it were, as well as the use of interface layer libraries like SPILT and `sthreads`, as well as the conversion of the code to Java. This section is devoted to covering each of these approaches in sufficient detail to allow them to be used again by a competent programmer.

#### 5.3.1 Porting the code from Solaris threads to pthreads 'by hand'

This was the first method employed, and I can't think of any waffle to put in here, so let's get down to actually discussing what was done.

##### Library inclusion:

Under Solaris two libraries are used when writing multi-threaded applications, the main one being `thread.h`, and the other being `synch.h`. There is just one equivalent library for POSIX threads, that is `pthread.h`. As a result, the first step is to modify the `#include` compiler directives, and also any explicit specification of libraries on the compilation command line using the `-l` flag. The aforementioned pair of `#include` directives occur on lines 58 and 59 of the original Sun code.

##### Thread, mutex and other declarations:

The appropriate modification of declarations is a trivial affair. The existing declarations like:

```
thread_t b_tid;           mutex_t io_lock;
Simply become:   And:  Becomes:
pthread_t b_tid;        pthread_mutex_t io_lock;
```

Most types can be converted in the same way as shown, i.e., simply replacing all instances of `thread_t` in the source code, with `pthread_t` or in the case of mutexes, prefixing the definition with `pthread_`. The first case occurs on the following lines in the original Sun code: 159-162, 596. The second case occurs on line 138 of the original Sun code.

## Creating threads

There is a fundamental difference in the way threads and their associated properties are initialised and managed within the Solaris and pthread APIs. The difference being that the `pthread` API requires an attributes object to be specify the state in which a new thread should be created, whereas the state the thread should be created in under Solaris is specified to the thread creation function (`thr_create`) through the six arguments within it's constructor:

```
int thr_create (void *stkaddr, size_t stksize, void *(*func) (void *),
              (void *arg), long flags, thread_t *tid);
```

As a result, the Solaris thread creation function allows for six different cases:

- Create a thread with all the default values.
- Create a bound thread.
- Create a detached thread.
- Create a thread with a custom stack size.
- Create a thread with a custom stack starting address.
- Create a thread with a customer stack address & stack size.

Thankfully, these cases are well documented in Sun's guide to porting code from Solaris to pthreads[3]. On pages 24-27, these conversions are documented. The first three cases are the most common and once you've know how to create an attribute object for a given thread, the rest is easy.

All of the threads used in the Fredholm integral equation solver are bound threads, so the first occurrence of a thread creation is used as an example - the method is the same for all the other thread creations in the source. The line numbers are shown to make referencing easier within in each file.

```
159  thread_t b_tid;

183  if ( s = thr_create( NULL, 0, b_vector, (void *) g,
                      THR_BOUND, &b_tid ) ) {
184    fprintf( stderr, "integral: thr_create: %s\n", strerror( s ) );
185    exit( 1 );
186  }
```

Here is the ported function using pthreads :

```
167  pthread_t      b_tid;
171  pthread_attr_t b_tattr;

195  s = pthread_attr_init (&b_tattr);
196  s = pthread_attr_setscope (&b_tattr, PTHREAD_SCOPE_SYSTEM);
```

```

197 s = pthread_create (&b_tid, &b_tattr, b_vector, (void *) g);

198
199 if ( s ) {
200     fprintf( stderr, "integral: thr_create: %s\n", strerror( s ) );
201     exit( 1 );
202 }

```

As you can see, in addition to the thread itself being declared (line 169), we also declare an attribute object for the thread (line 173). Before we can create the thread, we must initialise the attribute object, which is done by calling the function `pthread_attr_init` passing a reference to the attribute object. Then it is necessary to set the scope of the thread, this is done through it's associated attribute object. The scope is set by calling the function `pthread_attr_setscope` passing a reference to the attribute object as before, and also a flag stating the scope of the thread. In this case where we want to create a bound thread, the flag should be: `PTHREAD_SCOPE_SYSTEM`, but if we had wanted to create a detached thread, then the flag would be set to `PTHREAD_CREATE_DETACHED` (some early documentation stated this as `PTHREAD_THREAD_CREATE` which doesn't appear to be supported). Having initialised the attribute object, we can now create the thread, which simply involves passing references to the declared thread and it's (now initialised) attribute object, as well as the argument to the function and a pointer to the function itself (i.e. the function to be executed under that thread's control). These three main steps are taken on lines 197-199. It should be noted that threads with the same attributes can share an attribute object to save on resources, so for example in `MatrixOp()` where we have an array of threads, it is not necessary to have an array of thread attribute objects (although this is what was done<sup>8</sup>), just one is sufficient — in fact one attribute object could be defined globally and then used throughout the program.

## Joining threads

Porting thread join operations from Solaris threads to `pthreads`, is a simply affair. The function signatures of the Solaris (`thr_join()`) and `pthread` (`pthread_join()`) join functions do differ, as you can see. Here is the definition for the Solaris thread join function:

```
int thr_join (thread_t tid, thread_t *departdid, int *status);
```

And here is the definition for the `pthread` thread join function (this may have changed from 1003.4a/D8 to the final specification 1003.1c):

```
int pthread_join (pthread_t tid, int *status);
```

The only difference between the `pthread` and solaris implementation is that in `pthreads` the concept of being able to return the id of a terminating thread is not available. It is sufficient to specify the identifier of the thread that you wish to wait for (it's termination that is), without needing to worry about the departing thread id. So as a quick example, a thread join like the one on line 198 (Solaris):

---

<sup>8</sup>There is no special reason for this, and performance analysis not documented here seems to suggest that there is no reason not to.

```
198 thr_join( b_tid, NULL, NULL )
```

Becomes:

```
222 pthread_join( b_tid, NULL );
```

### Killing threads

The Solaris function for killing threads `thr_kill()` has an identical signature to the `pthread` thread kill function `pthread_kill()`, so all that needs to be done when porting is to simply change the name, simple! For example:

```
631 thr_kill( tid[i], SIGKILL );
```

Becomes:

```
669 pthread_kill( tid[i], SIGKILL );
```

We are now at the stage where all of the `pthread` specific conversion has been done. The remaining work to be done concerns the use of the `sysconf()` command to determine the number of CPUs that are on-line. Details of this can be found in section 5.3.4.

### So to sum up

The function ports were relatively straightforward. The POSIX thread API is more verbose than the Solaris thread API, and this can be seen by the growth of the source file from 814 lines to 854 lines (including whitespace). As we have seen, the very standard parts of the Solaris thread API have direct equivalents in the POSIX thread API, and can be easily ported.

#### 5.3.2 Compiling the Solaris code with SPILT under IRIX

[In section 2 - will not duplicate here - may move here though]

#### 5.3.3 Compiling the Solaris code using sthreads (Windows NT)

[In section 2 - will not duplicate here - may move here though]

#### 5.3.4 `sysconf()` macros — A stumbling block

Regardless of the approach taken to port the code, where there is a need to determine the number of CPUs on-line, this is likely to require alteration when the code is moved from one platform to another. In the case of moving from Solaris to IRIX, the macro changes from `_SC_NPROCESSORS_ONLN` to `_SC_NPROC_ONLN`, whilst certain systems do not even possess a `sysconf()` command like for example Linux and Windows 95/NT. On systems where the command is not supported, the obvious solution is of course to place the value in as a literal where there is no other equivalent function available (like `dwGetNumProcessors()` under Windows NT) to help you.

## IRIX

For the IRIX port using `pthread`s I have already stated that it is just a case of replacing the macro. The line concerned is line 600 of the original Solaris code:

```
600 nCPU = sysconf( _SC_NPROCESSORS_ONLN );
```

Which becomes the following in the IRIX `pthread` port:

```
634 nCPU = sysconf( _SC_NPROC_ONLN );
```

## Windows NT

Under Windows NT, the `sysconf()` command is unavailable, but the Win32 API does include it's own system information object, from which it is possible to get the number of CPUs in the system (only really relevant to Windows NT, since Windows 95 and 98 are uniprocessor operating systems). So when modifying the code for Win32, one simply removes the `sysconf()` command, and replaces it with the appropriate data field access on an instance of the `SYSTEM_INFO` object (which must be declared first - hence the second of the following lines):

```
162 static DWORD    nCPU = 0;
163 SYSTEM_INFO process_info;
619             GetSystemInfo(&process_info);
620             nCPU    = process_info.dwNumberOfProcessors;
```

The above operation is taken from the `stthreads` code, but the same operations are performed for the NT `pthread`s version of the code, except of course that they occur at slightly different line numbers. This code will function under all 32-bit flavours of Windows (95/98/NT), and stores the number of CPUs on-line in the variable `nCPU` as `sysconf()` does in the original code.

## Operating systems that don't have support for this functionality

In these cases such as under Linux 2.\*, the solution is simply replace the call to `sysconf()` with a literal value — I used 1 on my PC for example.

## 5.4 Porting the multi-threaded C code to Java

This has been one of the most complex areas of the project so far, and is the culmination of the work elsewhere on the solver program. As one can see, it was necessary to gain experience with threading in C, and then general Java programming; particularly porting the single threaded code from C to Java, in order to minimise the number of problems that could potentially occur at this stage. Although it can be tempting to jump straight in, the cautious approach has paid off, since the non thread-specific language differences were

already known and did not pose any problems, leaving the way clear for the threading issues to be dealt with.

The code and the procedures followed for the multi-threaded port are heavily based on the single threaded port from C to Java. All of the issues related to the differences between C and Java occur again in this case, with of course additional differences due to the threading. Areas of overlap will *not* be covered, since the methods employed in the single threaded port for dealing with those issues (e.g. parameterised macros) are equally relevant here.

#### 5.4.1 Remaining issues with language differences

Many of the issues related to the differences between C and Java have already been covered in the section on porting the single threaded solver to Java. One or two however, appear in addition to those already stated.

##### Assertions

Assertions are a diagnostic feature of C programs, which test for a given condition and abort if the condition is false putting an error message onto the standard error stream. Assertions are used when a condition must always be true to ensure correct operation of the program — if it is not, then this is usually a sign that something has gone horribly wrong. Whilst a useful debugging feature, an assertion isn't any different (apart from its compiler implementation) from having an `if` statement to test the condition and then output an error message before calling `exit()`. As a result, the Java developers saw fit to omit assertions since they are not entirely necessary.

There are three assertions within the multi-threaded solver, which are all related to determining the number of CPUs within a system. Unfortunately there are problems in doing this within Java, and these problems will be examined in more detail later. For now, it is worthwhile to abstract away the problems, and look at one of the assertion statements and how it should be converted to Java:

```
637          assert( nCPU != 0 );
```

As one can see, this is a fairly crucial assertion, and must obviously be true, since if the number of CPUs were indeed zero we would be unable to run the program in the first place! It may appear to be nonsense to have such an assertion. But the reason is that a call is made earlier to `sysconf()` using the macro `_SC_NPROC_ONLN` to determine the number of available CPUs, and this may well be undefined - something that could cause huge problems later on. In Java we do not have access to such information, whilst enclosed in the confines of the virtual machine, without the use of native code. Whilst I have looked at native code to determine this system information, the current incarnation of the Java code does not provide any form of testing for these values (subsequent versions will do so). If it were to do so, the following would suffice:

```

if( nCPU != 0 ) {
    System.err.println(“Assertion failed! Aborted.”);
    System.exit(0);
}

```

### Command-line argument parsing

The multi-threaded C solver takes up to two arguments, the first of which is the size of the matrix that you wish to use, whilst the second is the number of CPUs that you wish to use for the integration. The code simply uses the standard argument variables `argc` and `argv`, the first of which is an integer specifying the number of arguments on the command line (including the program name), whilst the second is a pointer to an array of pointers to character arrays containing the arguments. The values are obtained as follows (code fragment from the `threads` version):

```

165 main( int argc, char *argv[] ) {
...
176     if ( argc > 1 )
177         n      = atoi( argv[1] );
178
179     assert( n <= N );
180
181     if ( argc > 2 )
182         nCPU   = atoi( argv[2] );

```

As one can see, the code simply checks the length of the argument list, and uses `atoi()` to parse integers from the character array found at each element position. The code for this is similar in Java, although instead of `atoi()` one uses the `parseInt()` method of the `Integer` object, and the argument list is represented as an array of strings (and given the OO nature of Java, the length is determined using the `length()` method of `args`):

```

81 public static void main(String args[]) {
82
83     // Get size of matrix and number of CPUs to utilise from argument list
84     try {
85         if ( args.length > 0 )
86             n = Integer.parseInt(args[0]);
87         else if ( args.length > 1 )
88             n = Integer.parseInt(args[0]);
89             nCPU = Integer.parseInt(args[1]);
90     } catch (NumberFormatException e1) {
91     } catch (ArrayIndexOutOfBoundsException e2) {
92     };

```

The rest is familiar except for the exception handling, which is required in case an integer cannot be parsed (then a `NumberFormatException` is thrown), or an attempt is made to access beyond the length of the array (in which case an `ArrayIndexOutOfBoundsException` is thrown) - the latter would never happen because we only access `args` if the arguments are present. Notice that both function signatures for `main()` have been shown to aid understanding — they are important when it comes to parsing arguments!

This is a simple conversion but worth noting.

## Structures

As we saw in the breakdown of the port of the single threaded solver from C to Java, structures must be converted to classes within Java. Whilst there was only one structure within the single threaded solver, there are three within the multi-threaded solver (`POINT`, `matrix_t`, and `MatrixOpArg`). As with the single threaded solver these classes are included as inner classes, although initially were separate classes. On `kilburn` there have been no real difference in making these class inner classes, though `irwell` has problems with this and does not complete execution of the code. As a result of this, two versions of the code are being made available until the problem is fixed - an 'all-in' one Java source file, and a gzipped tape archive which contains all the classes.

Both `matrix_t` and `MatrixOpArg` have seen changes. In `matrix_t` the changes are minor, in that the type `MATRIX` is simply substituted for its real types (since Java doesn't have `typedefs`). Although this issue was covered earlier in the section on porting the single threaded code; it is worth mentioning again, since it relies on a constant `N` to define the size of the matrix, if the class is not defined as an inner class, then this variable must be referenced as `Fe_mt.N` rather than just `N`.

As far as `MatrixOpArg` is concerned, two constructors have been added: a default constructor (for when a `MatrixOpArg` is instantiated with no parameters) and a copy constructor (to produce a clone of another `MatrixOpArg` object) — these are a logical part of the transition from a structure to an object (and very helpful when it comes to creating new threads as we shall see).

Another difference is that `opd` is removed from the Java implementation - this was of type `void *` in the C version and was used solely for passing pointers to the functions `g()` and `K()`. The removal of this functionality was largely due to the belief that there was no equivalent to function pointers in Java. Even though later on the functionality was discovered and used, it would not have made sense to keep `opd`, since the same function pointers were being passed to the same functions (similar to the situation with the single threaded solver), hence it was better to simply call the guest functions directly within the host functions. If one traces through the code, it is quite clear that the pointers to the functions `g()` and `K()` are unnecessary, and can be removed. Since as in the case of the single threaded solver, we find that these pointers are being passed to the same functions (each host function takes just one of the pointers), and hence might as well be replaced by direct function calls. For example, when the thread to execute the function `b_vector()` is created, it is passed a pointer to the function `g()`. Within `b_vector()` this function pointer is then passed to the function `ChebyshevCoeff()`, which then invokes its guest function. As we discovered in the single threaded code, `ChebyshevCoeff()` only takes one type of pointer - that to the function `g()` (the function is identical in the multi-threaded version), and hence the function might as well be called directly (which is what happens in the Java version). Looking at another chain of execution (the only one where `opd` from `MatrixOpArg` is actually utilised, one finds that similar savings can be made. In this particular case, a thread is created to execute the function `K_matrix()` and is passed a pointer to the function `K()`. Within the function `K_matrix()`, the function

`MatrixOP()` is called with a pointer to the function `K_mop()`, and also the pointer to the function `K()` — which is placed into the `opd` field of the `MatrixOpArg` structure `arg` which is passed to `K_mop()`, which then later invokes that function. It should be noted that this is the only function passed in to `K_mop()`, and as a result, the whole procedure of passing the function pointer around can be circumvented by removing `opd` from `MatrixOpArg` (it is no longer needed), and all the function pointer related arguments (such as `void *arg` in `K_matrix()` and of course passing the pointer in, in the first place!

So to compare the changes to `MatrixOpArg` itself following the aforementioned changes, here is the code for the original structure in C:

```

107 typedef struct {
108     matrix_t    *a;    /* the matrix          */
109     int         row;   /* starting row number */
110     int         nrow;  /* number of rows      */
111     void        *opd;  /* operation private data */
112 } MatrixOpArg;

```

Which after the addition of constructors and other modifications detailed is as follows in Java:

```

661 public class MatrixOpArg {
662
663     matrix_t    a;    /* the matrix          */ // wasptr
664     int         row;  /* starting row number */
665     int         nrow; /* number of rows      */
666
667     // copy constructor for instantiation in ExecThread
668     public MatrixOpArg( MatrixOpArg tmp ) {
669
670         a = tmp.a;
671         row = tmp.row;
671         nrow = tmp.nrow;
672
673     }
674
675     public MatrixOpArg() {
676
677     }
678
679 } // end of class: MatrixOpArg

```

## References

When documenting the port of the single threaded solver, the issue of pointers was given a thorough treatment. Most of the issues, such as dereferencing a pointer are the same (i.e. by using `.` rather than `->`), but the issue of explicit references using `&` were not present in the single-threaded port. Such referemces are used in the multi-threaded code, for example:

```

806     MATRIX *m = &arg->a->m;

```

Which requires that field `m` of field `a` of the object `arg` be treated as a references. In Java this will happen anyway, so we do not need the `&` operator, and all that remains to be done is to change the member access operators as we would normally do, giving:

```
839         m         = arg.a.m;
```

Therefore the process for dealing with such explicit references is simply to remove the `&` operator.

#### 5.4.2 Porting the threaded sections of code - potential problems

The major problem of course when porting multi-threaded code between C and Java is the difference between the threading models. C allows a function to be executed within it's own thread of control, by allowing the user to create a thread passing a pointer to the function to be executed by the thread in to the thread creation function. So for example, if you have a function `void foo()`, and want this to be executed in a different thread of control to the main program, you create a new thread passing a pointer to that function in (example shown for `pthread`):

```
pthread_t foo_thread;
pthread_create( &foo_thread, NULL, (void (*)(void *)) foo, (void *) NULL) ;
```

Java on the other hand, treats threads as objects and any new threads must be developed either by inheriting from the `Thread` object, or implementing the `Runnable` interface and overriding the `run()` method (this is done for either approach). Also rather than using API calls to control the threads (i.e., to start, join, etc.). these operations are provided as methods of the `Thread` object which each new thread inherits (as one would expect of an object-oriented language). In Java the code that you wish to execute in a separate thread of control, must go within the `run()` method:

```
public class fooThread extends Thread {

    public fooThread() {

        public void run() {
            // implementation of foo
        }

    }

}
```

The threads are then created as follows::

```
fooThread ft = new fooThread(); // create
ft.start(); // start execution
```

As one can see there is an appreciable difference even for a simple implementation. There does however, seem to be a straightforward pattern that can be followed to allow porting of such code. i.e. to take the implementation in the function to be executed by the thread (i.e. the code within `void foo()`) and then subclass (inherit from) `Thread` (or implement a `Runnable` interface overriding the `run()` method with that code — ensuring that standard C to Java issues are taken care of. This approach ensures that the new code comes within Java's object-oriented paradigm, and is the most straightforward approach where there are only a small number of different threads. This is of course the approach that was taken when producing the Java port of Jens Latza's producer/consumer program (by inheriting from `Thread`).

Unfortunately, this approach causes us some problems with the port of the multi-threaded solver program. This was an approach initially taken and then abandoned, the reason for which is a clear demonstration of the problems of moving code from C's threading model to Java's for a certain class of numerical application. The problem is perhaps demonstrated by a look at a typical chain of execution.

Starting in `main()`, a number of the threads are created to execute functions. All of the functions that work on matrices (i.e. those with the suffix `_matrix`, invoke the function `MatrixOp()` passing a pointer to their respective worker function (suffix `_mop`, e.g., the worker function for `t_matrix()` is `t_mop()`). The job of the function `MatrixOp()` is to perform argument filling, and thread creation, thus splitting the job over different threads (the number being dependant on the CPUs available). Function pointers are very useful here, since it is not possible to determine the name of the calling function and hence invoke the correct worker function by using that information. The only other alternative in C would be to duplicate this code for each of the `_matrix` functions, but instead of using the function pointer that would normally be passed to `MatrixOp()`, one would simply use the a pointer that particular worker function when creating the thread. In C this is wasteful and unnecessary, and the use of one function to perform all of the argument filling and threading is rather more elegant.

The problem is of course when porting the code is that `MatrixOp` and Java's threading model do not work well together, particularly when it comes to argument passing. If one were to follow the earlier suggestion for converting multi-threaded C code to Java, i.e., to have a new type of thread object for each different function executed within threads, the `run()` method of the object would contain the code previously found in such a function. At first glance this may work, since one could be under the impression that instead of passing a function pointer to `MatrixOp()`, the appropriate thread for a given worker function could be passed in instead. One of the problems with this approach is that the thread would have to be created in the function that calls `MatrixOp()`, and this would not be very helpful, given that the arguments need to be passed to the thread on it's creation (hence requiring this to be done in `MatrixOp()` of course the major problem with this that as stated before, `MatrixOp()` - even in Java has no concept of the calling function. A slightly more subtle approach would be to simply declare the thread in the function that calls `MatrixOp()` and pass this `null` reference to `MatrixOp()` where it could be instantiated with the correct arguments (using `thr = new t_mop_thread(arg)` for example); again though, this will not work, since in order to get around Java's strict type checking mechanism, the thread must be downcast to the type `Thread` losing the specific type information and thus preventing correct instantiation. If one looks at the

an early version of `MatrixOp()` that was initially implemented this way:

```
int MatrixOp( matrix_t a, Thread thr ) {

    int          i;
    int          j;
    int          rpt;
    int          mod;
    Thread[] thrs = new Thread[PARALLEL];
    MatrixOpArg[] arg = new MatrixOpArg[PARALLEL];

    // CPU determination (1 assumed for now)
    if ( nCPU == 0 ) {
        nCPU = 1;
        //      nCPU = sysconf( _SC_NPROCESSORS_ONLN );

        //          assert( nCPU != -1 );
        //          assert( nCPU != 0 );
    }

    if ( nCPU > PARALLEL )
        nCPU = PARALLEL;

    rpt = a.nrow / nCPU;
    mod = a.nrow % nCPU;

    /* fill the argument structures */
    for ( i=0, j=0; i < nCPU; ++i ) {
        arg[i] = new MatrixOpArg();
        arg[i].a = a;
        arg[i].nrow = rpt;
        arg[i].row = j;

        if ( mod != 0 ) {
            arg[i].nrow++;
            mod--;
        }

        j += arg[i].nrow;
    }

    /* create the threads */
    for ( i=0; i < nCPU; ++i ) {

        thrs[i] = new Thread(arg); // ??? !
        thrs[i].start();

    }

    /* wait for them to finish */
    for ( i=0; i < nCPU; ++i )

        try {
            thrs[i].join();
        } catch (InterruptedException e) {
        };
};
```

```

    return( 0 );

}

```

One can see that although downcasting to `Thread` is fine for getting the threads passed into `MatrixOp()` we cannot then instantiate the thread with the reference we have. The statement:

```

    thrs[i] = new Thread(arg);

```

Will not work, because `Thread` is what we inherited from, and hence has none of the functionality required (the Java compiler will quite correctly complain that `Thread` doesn't have a constructor for objects of type `MatrixOpArg`). This leaves of course leaves a problem in so far as the thread cannot be instantiated before it is passed in, since arguments required are not available until within `MatrixOp()`, and the thread cannot be instantiated within `MatrixOp()` cleanly, since we do not have the necessary information (such as originating function, etc). The example shown was a simple conversion, where the code from functions to be executed within a given thread of control, was taken and placed into thread objects, so for example, the `t_` functions from the multi-threaded C:

```

void *
t_matrix( void *arg ) {
    MatrixOp( &t, (void *(*)(void *)) t_mop, NULL );

    return( NULL );
}

```

And:

```

void *
t_mop( MatrixOpArg *arg ) {
    int    i;
    int    lim;
    MATRIX *m      = &arg->a->m;

    lim      = ( i = arg->row ) + arg->nrow;

    for ( ; i < lim; ++i ) {
        int    j      = ( i % 2 )? 1 : 0;
        int    k      = ( i % 2 )? 0 : 1;

        for ( ; k < n; k+=2 )
            (*m)[i][k]      = 0.0;

        for ( ; j < n; j+=2 )
            (*m)[i][j]      = 0.5 * ( 1.0/(i+j+1.0) - 1.0/(i+j-1.0) + 1.0/(i-j+1.0) - 1.0/(i-j-1.0) );
    }

    return( NULL );
}

```

Became the following in Java (bear in mind that pointers have been removed):

```

public class t_matrix_thr extends Thread {

    private t_mop_thr thr;

    public t_matrix_thr() {

    }

    public void run() {

        MatrixOp( t, thr );

    }

} // end of class: t_matrix_thr

```

And:

```

public class t_mop_thr extends Thread {

    private double[] [] m = new double[N][N];
    private MatrixOpArg arg = new MatrixOpArg();

    public t_mop_thr( MatrixOpArg arg ) {

        this.m = arg.a.m;

        return;

    } // was t_mop

    public void run() {

        int    i;
        int    lim;

        lim    = ( i = arg.row ) + arg.nrow;

        for ( ; i < lim; ++i ) {
            int    j      = ( i % 2 != 0 )? 1 : 0;
            int    k      = ( i % 2 != 0 )? 0 : 1;

            for ( ; k < n; k+=2 )
                m[i][k] = 0.0;

            for ( ; j < n; j+=2 )
                m[i][j] = 0.5 * ( 1.0/(i+j+1.0) - 1.0/(i+j-1.0) + 1.0/(i-j+1.0) - 1.0/(i-j-1.0) );
        }

    }

} // end

```

These functions do not differ greatly, apart from the declaration of the threads and of course removal of pointers. One can see the intention and of course the weakness of this

method where `MatrixOp()` is concerned. This would have been the soundest approach, and of course closest to Java's object oriented paradigm, had there not been a problem with argument passing within `MatrixOp()`. As a result, a new approach had to be found. The main idea being that the Java port of the solver should be kept as close to the original C code as possible, and one way to do this would be to have just one type of thread that could take a reference to a method, so that this thread could be instantiated within `MatrixOp()` and then the method passed to it would be invoked when the thread started execution. Thankfully this turned out to be not quite the impossible task one might imagine, since Java now provides a new API — reflection (from Java 1.1) which can be used to get, and invoke static and instance methods, as well as many other features that we are not interested in at this time.

A final late note (January 22, 2000) is that whilst it was not possible to pass arguments to the threads within `MatrixOp()`, because we didn't know which of the thread types we wanted a new instance of, there was another possibility. That being to pass a `Constructor` object into `MatrixOp()` as well as the different thread types. It would then be possible using the Reflection API to create a new instance using a method of the `Constructor` objects such as `newInstance()` and therefore pass the arguments in at that stage. That way, all of the code would be implemented as separate thread types, that could still be treated as type `Thread` for the purposes of joining, etc. It would also have required very little modification from the original proposal. However, at the time that the choice to use method references was made, this functionality was unknown, and it is only through use of Reflection that this option became apparent. It is therefore something that should be considered for the future, since it is likely to be more reliable than method references and is much more in the spirit of Java.

The reflection API is not one of the most well documented areas of the Java language, and is notorious for its complexity. The best documentation that I have found for the API is within the excellent book *Core Java* [9], which gives the topic a good introduction. As stated in the book, the use of method references should really be a last resort, due to their complexity and also speed - they are *slow*. Having said that, it is still worthwhile to use method references due to the calling mechanism used in the program; anything else would require considerable modifications to the code (something that was to be avoided if possible, since the key area of interest was to get a multi-threaded numerical application from C to Java with the minimum of fuss).

### 5.4.3 The process - what it entails

As one will have gathered by now, a fair proportion of the code is the same as that within the single threaded solver. By comparing the implementation of functions in the single and multi-threaded C (by eye), the following functions were found to be identical:

- `g()`
- `K()`
- `RE()*`

- VALUE()\*
- \_exchange()\*
- ChebyshevEval()
- ChebysehvCoedd()
- Points()
- MatrixSolve()
- MatrixLUsolve()
- VectorWrite() - slight diff with printf() format specifier.
- MatrixWrite()
- PointWrite() - as above (%18.12f used instead of %10.6f).

\*: These are parameterised macros in the C code that have been converted to functions in the Java port - there are no differences between the single and multi-threaded versions for this function.

For the functions stated above, the code was taken from the single threaded port where the conversion had already been done (there was no point re-porting the same code), with the appropriate minor modifications being made where necessary. The remainder of the functions are all thread related, those being the key function `MatrixOp()` at then all the `_matrix` and `_mop` functions. Since all of the language specific issues such as pointers, remain the same for this port and hence the same techniques apply, The focus of the remainder of this section is on actually getting the multi-threading working using the Java reflection API.

### ExecThread

As stated the idea of having one type of thread that could take some form of function pointer was very appealing. This has after quite a bit of difficulty getting to grips with reflection, been managed. The current version of the code has just one thread type `ExecThread`, the implementation being as follows (this is completely new code):

```

868     public class ExecThread extends Thread {
869
870         private Method md2;
871         private Object[] w_args;
872
873         public ExecThread( Method md, MatrixOpArg arg ) {
874
875             if ( arg != null ) {
876                 Object tmp[] = { new MatrixOpArg(arg) };
877                 w_args = tmp;
878             }

```

```

879
880     md2 = md;
881
882     }
883
884     public void run() {
885
886         try {
887             md2.invoke( new thr
888 _fns(), w_args );
889         } catch (IllegalAccessException e) {
890             System.err.println(thr_fns.class.getName() + e);
891         } catch (InvocationTargetException e) {
892             // System.err.println(thr_fns.class.getName() + ": " + e + " -> " + e.getTargetExceptp
893         };
894         return;
895     }
896
897 } // end of class ExecThread

```

The implementation is relatively straightforward, with the body of the `run()` method consisting simply of an instruction to invoke the method passed in along with the arguments passed in, thus executing the function within a thread of control — following the C threading paradigm (hence making the conversion of code easier). One will notice that exception handling is required — the code will not compile without it. The first type of exception that is caught is `IllegalAccessException` thrown when trying to invoke the method, whilst the second is an `InvocationTargetException` which is a wrapper for an exception that has occurred within the invoked method.

It is useful to note that when passing the parameters to the function you wish to invoke, they must be wrapped up in an array of type `Object`, this effectively means that a new instance must be created within a new `Object` array — this is shown above (873). The array is then unwrapped by the virtual machine when the method is invoked. For this program, we were lucky that the original C program had a special type for passing arguments of a matrix operation (`MatrixOpArg`) in the form of a structure, which was easy to convert to an object - all that then needed to be done was to add a copy constructor to allow a new instance to be created from an existing one specifically for this purpose (the copy constructor was used, because the standard method `clone()` inherited from `Object` did not work). The key point is that *everything* must be wrapped up as an object - even primitive types (these must use their wrapper objects, so `double` would use its wrapper `Double`, `int` would use `Integer` and so on). Another complication is return types, if the return type is anything other than `void` then you must have an array of type `Object` for the reflection API to place the returned values from the invoked method (which will be wrapped up in other objects if necessary). Thankfully, this is not an issue for this program since all the functions have a `void` return type.

Also note that instance methods are invoked on an instance of an object (if the method is static `null` can be used here. In this case a containing object is used (`thr_fns()`), so a new instance of this must be created (unless there is another instance available).

*The existing thread functions*

Having created a wrapper object to allow us to execute the functions in separate threads of control, It is now a straightforward step to move those functions executed by the threads [list follows]:

- `b_vector()`
- `t_matrix()`
- `K_matrix()`
- `b_matrix()`
- `A_matrix()`
- `t_mop()`
- `K_mop()`
- `A_mop()`
- `b_mop()`

All one needs to do is to change their return type from the C type `void *` to `void`, perform any of the usual language specific conversions (examples of which have already been seen) within the body of the function, and then of course ensure that plain `return` statement is used at the end of the function (the C functions returned a NULL pointer). Here is a quick example of this, the function `t_mop()` appears as follows in the `threads` multi-threaded version of the solver:

```

684     void *
685 t_mop( MatrixOpArg *arg ) {
686     int    i;
687     int    lim;
688     MATRIX *m      = &arg->a->m;
689
690     lim    = ( i = arg->row ) + arg->nrow;
691
692     for ( ; i < lim; ++i ) {
693         int    j      = ( i % 2 )? 1 : 0;
694         int    k      = ( i % 2 )? 0 : 1;
695
696         for ( ; k < n; k+=2 )
697             (*m)[i][k]      = 0.0;
698
699         for ( ; j < n; j+=2 )
700             (*m)[i][j]      = 0.5 * ( 1.0/(i+j+1.0) - 1.0/(i+j-1.0) + 1.0/(i-j+1
701     }
702
703     return( NULL );
704 }
```

Following porting the function appears as follows in Java:

```

697     public void t_mop( MatrixOpArg arg ) {
698
699         int    i;
700         int    lim;
701         double[][] m = new double[N][N];
702
703         m      = arg.a.m;
704         lim    = ( i = arg.row ) + arg.nrow;
705
706         for ( ; i < lim; ++i ) {
707             int    j      = ( i % 2 != 0 )? 1 : 0;
708             int    k      = ( i % 2 != 0 )? 0 : 1;
709
710             for ( ; k < n; k+=2 )
711                 m[i][k] = 0.0;
712
713             for ( ; j < n; j+=2 )
714                 m[i][j] = 0.5 * ( 1.0/(i+j+1.0) - 1.0/(i+j-1.0) + 1.0/(i-j+1.0) - 1.0/(i-j-1.0) );
715         }
716
717     }

```

The conversion is a very simple one, with the usual language issues of course being dealt with (pointers). The only additional point that one should bear in mind is that the functions in the list above that are executed by threads; need to be contained within an instantiable object, since they cannot be declared as `static`. Therefore the best solution is simply to have an object just to contain these functions (rather than creating new instances of the main class — that would not be an entirely sensible move), which has been called `thr_fns`. The actual process of creating a method reference is covered in the section on creating threads, since this is the only place where these are required.

### Converting the threaded code

Now that we have our new thread object, we can set about converting the code. The conversions shall be categorised in terms of the standard threading functionality.

#### *Creating threads*

With our new class, the procedure for creating a thread is very similar to that in C. For example, in the original code, the thread to execute the function `t_matrix()` is created as follows:

```

206         s = pthread_create (&t_tid, &t_tattr, t_matrix, NULL);

```

Which in the Java version is:

```

127         ExecThread t_thr = new ExecThread(thr_fns.class.getMethod("t_matrix", new Class[] {

```

As one can see, the `pthread` thread creation function has four arguments, the first is the thread id, which is used as a handle for the thread now declared, whilst the second is an attribute object that is used to set the properties for the object (such as the contention scope of the thread for example). The third argument is the pointer to the function to be executed by the thread, and the fourth argument contains the argument for the function that is to be executed by the thread. The first two arguments of the `pthread` call are of no interest to us, since we do not need attribute objects (Java threads are much simpler, and in any case, all information related to a thread is contained within the particular thread object), and neither do we need thread id's since all Java threads are accessed via their implicit reference or 'handle' (in this case `t_thr`). The latter two arguments are the same as those required by our `ExecThread` object, the first being a method reference to the method that will be invoked within the thread, and the second the argument with which that method is to be invoked.

Notice how the method reference is obtained — a much more involved process than in C. In order to obtain a method reference, one must invoke the `getMethod()` method of the containing object (`thr_fns`, supplying the name of the method as a string, along with an array of `Class` objects where each element in the array is the class object for the types as they occur in the method signature. If the function does not take any arguments as in the case of `t_matrix()` shown above, one simply creates an empty array of `Class` objects. Otherwise, one can simply create the array on the fly using the `class()` method of an object, as is the case when the method references are created for `MatrixOp` - here is how it is done in the method `t_matrix()`:

```

687     try {
688         MatrixOp( t, thr_fns.class.getMethod("t_mop", new Class[] { MatrixOpArg.class}));
689     } catch (NoSuchMethodException e) {
690         System.err.println(thr_fns.class.getName() + e);
691     };

```

Here the method signature for the method we want a reference to has just one argument which is of type `MatrixOpArg`, so we simply create our array of `Class` objects by calling `class()` on `MatrixOpArg`, which will ensure that the appropriate class object is placed in the first position of the array. For methods with more than one parameter, the `Class` objects for each type must be ordered within the array in the same order in which they occur in the signature of the method, so if you had a method that took an integer and double value — specified in that order in the method signature, the corresponding array of `Class` objects would have to contain `Integer` and a `Double` objects in the first and second positions (elements 0 and 1). As far as this case is concerned, the functions that are run within separate threads of control only either take no arguments or one argument of type `MatrixOpArg`; both cases have already been shown.

One final point that is worth noting about the use of `ExecThread` is that it can be used for both types of functions, i.e. those not taking an argument and also those that take a `MatrixOpArg` argument. If the function to be executed within the new thread does not take an argument, then a `null` reference is used as the second argument rather than a `MatrixOpArg` object. As one can see from the definition of `ExecThread` above, the array of type `Object` is only constructed and passed in to `invoke()` when the second argument to `ExecThread` is not `null`. This ensures that the correct information is available to allow

`invoke()` to find and invoke each method correctly (if the array of type `Object` were to be supplied in all cases for instance, then `invoke()` would fail to find those methods that take no arguments — throwing a `NoSuchMethodException` or `IllegalArgumentException` in these cases, hence the need for caution here).

### *Joining threads*

Joining threads (i.e. waiting for specified thread of control to finish executing) is a simple undertaking in both C and Java. The following thread join function occurs within `main()` in the `pthread` port of the solver:

```
222         pthread_join( b_tid, NULL );
```

Bearing in mind that all threads functionality is contained within the `Thread` object that we inherit from, we simply invoke the `join()` method on the thread (by its implicit reference) that we wish to wait for. So the join operation shown above is represented as follows in Java:

```
152     try {
153         b_thr.join();
154     } catch (InterruptedException e) {
155         System.err.println("b_thr join: " + e);
156     };
```

The same mapping should be made for all joining operations. One should of course note the exception handling which is mandatory.

### *Mutexes*

The concept of a mutual exclusion lock, which only allows one thread at a time to modify shared data, can be replaced by the use of Java's synchronisation mechanism which behaves similarly. In the case of the multi-threaded solver, such a lock is only employed once within `b_vector()`:

```
721         IO_lock();
722         VectorWrite( stdout, "Chebyshev Coefficients for g(x)", b, n );
723         IO_unlock();
```

`IO_lock()` and `IO_unlock()` were defined as parametrised macros earlier:

```
74 #define IO_lock()         (pthread_mutex_lock(&io_lock))
75 #define IO_unlock()      (pthread_mutex_unlock(&io_lock))
```

The use of the mutual exclusion lock is of course to ensure that if `b_vector()` calls the output function `VectorWrite()`, then `main()` does not do so at the same time. This can be achieved in Java by placing the call in a synchronised block:

```

724     synchronized ( io_lock ) {
725         VectorWrite( System.out, "Chebyshev Coefficients for g(x)", b, n );
726     }

```

One thing to notice is that with the `pthread pthread_mutex_lock()` function is that it takes a parameter which is of type `pthread_mutex_t`. All synchronisation is carried out on these mutual exclusion locks represented by `pthread_mutex_t`. In Java, the value used can be any variable that is to be modified within the critical section you are protecting (i.e. a shared variable), or alternatively one can define a variable specifically for this purpose - a static object is quite common (which is of course what I did, since `io_lock`) is one of these. If one is porting mutual exclusion locks as in this case, then one can simply perform treat the `pthread_mutex_lock()` and `pthread_mutex_unlock()` as delimiter for the code that should be placed within a synchronisation block within Java — as above. Then one can simply convert the `pthread_mutex_t` declarations to those of type `static Object` as below:

```

146 pthread_mutex_t      io_lock;

```

Now:

```

55     public static final Object lock = new Object();

```

Thus giving a quick and simple conversion.

### *Killing threads*

Although the `Thread` object within Java has a `stop()` method for terminating the execution of a thread, this method is deprecated in the latest Java release (1.2), along with the `suspend()` and `resume()` methods. All of these methods are considered unsafe, and despite deprecation in the current Java release, the Java development team recommend that their use is avoided within any other release. In the case of the `stop()` method that we are currently interested, the current recommendation is that if possible its use should be avoided, i.e. if the `run()` method of the thread one is considering terminating could complete and terminate normally in a reasonable length of time, then it should be left to do so. In the case of strongly CPU bound applications such as iterative loops and particularly infinite loops that unless explicitly terminated would continue to run indefinitely, the recommendation is that the `stop()` method is overridden and a form of boolean is used (that might be named `canRun` for example), this can be set to false when `stop()` is invoked; this condition would then of course be tested on each iteration of the loop, allowing the loop to terminate when the condition is no longer true (when it has been set by `stop()`). There are many reasons for not using `stop()` such as the fact that all locks are released when `stop()` is called giving huge potential for inconsistencies, for the full details of the deprecation and methods to improve code, one should refer to the documentation at Sun's web site[?].

( ) Having given details of the problems with `stop()`, everything really boils down to whether or not the threads can be left to terminate normally or not. In the case of the multi-threaded solver, it is quite safe to let the threads terminate normally, which they will do since there are no infinite loops in any of the thread based functions. Within the C code, the only use of `pthread_kill()` is within the function `MatrixOp()` where threads are created to run different parts of the integration in parallel:

```

660         for ( i=0; i < nCPU; ++i ) {
661             int      n;
662 #define THR_FLAGS      PTHREAD_SCOPE_SYSTEM
663             n = pthread_attr_init (&tattr[i]);
664             n = pthread_attr_setscope (&tattr[i], THR_FLAGS);
665             n = pthread_create (&tid[i], &tattr[i], op, &arg[i]);
666             if ( n ) {
667                 for ( ; i; i-- )
668                     pthread_kill( tid[i], SIGKILL );
669             }
670             return( n );
671         }
672     }
673 }

```

As one can see from the context in which the call to `pthread_kill()` occurs, it is only there to kill all threads should one of the threads fail to create properly (of course if one thread is absent the whole process will fail); after that the error code from the thread creation is returned to the calling function. Therefore under normal circumstances, when each thread is created without problems; the threads are allowed to exit through the normal means of completing and returning (technically it is better to call `pthread_exit()` at the end of a thread based function, though not everyone seems to follow this convention, the developer of this program being one of them). Given that Java would throw an exception which is likely to result in the termination of the program's execution, we can quite safely avoid the use of the `stop()` method and just leave this section (lines 666-672) out altogether).

### MatrixOp() after porting

Here is how the `MatrixOp()` function appears after completely porting it to Java and using our new class:

```

585     int MatrixOp( matrix_t a, Method md ) {
586
587         int      i;
588         int      j;
589         int      rpt;
590         int      mod;
591         ExecThread[] thrs = new ExecThread[PARALLEL];
592         MatrixOpArg[] arg = new MatrixOpArg[PARALLEL];
593
594         if ( nCPU == 0 ) {
595
596             nCPU = 1; // defaults to 1 for green threads - modify if required

```

```

597             // or place native call here for sysconf
598             // nCPU is most easily specified as args[2].
599         }
600
601     if ( nCPU > PARALLEL )
602         nCPU = PARALLEL;
603
604     rpt = a.nrow / nCPU;
605     mod = a.nrow % nCPU;
606
607             /* fill the argument structures */
608     for ( i=0, j=0; i < nCPU; ++i ) {
609         arg[i] = new MatrixOpArg();
610         arg[i].a = a;
611         arg[i].nrow = rpt;
612         arg[i].row = j;
613
614         if ( mod != 0 )
615             arg[i].nrow++;
616         mod--;
617
618         j += arg[i].nrow;
619     }
620             /* create the threads */
621     for ( i=0; i < nCPU; ++i ) {
622
623         thrs[i] = new ExecThread( md, arg[i] );
624         thrs[i].start();
625
626     }
627
628             /* wait for them to finish */
629     for ( i=0; i < nCPU; ++i )
630
631         try {
632             thrs[i].join();
633         } catch (InterruptedException e) {
634             System.err.println(Fe_mt.class.getName() + e);
635         };
636
637     return( 0 );
638 }
639 }

```

For those interested, here is how `MatrixOp()` appears in the `pthread`s (C) version:

```

622     int
623 MatrixOp( matrix_t *a, void *(*op)( void * ), void *opd ) {
624     int i;
625     int j;
626     int rpt;
627     int mod;
628     pthread_t tid[PARALLEL];
629     pthread_attr_t tattr[PARALLEL];
630     MatrixOpArg arg[PARALLEL];
631

```

```

632     if ( nCPU == 0 ) {
633
634         nCPU    = sysconf( _SC_NPROC_ONLN );
635
636         assert( nCPU != -1 );
637         assert( nCPU != 0 );
638
639     }
640
641     if ( nCPU > PARALLEL )
642         nCPU    = PARALLEL;
643
644     rpt    = a->nrow / nCPU;
645     mod    = a->nrow % nCPU;
646
647                                     /* fill the argument structures */
648     for ( i=0, j=0; i < nCPU; ++i ) {
649         arg[i].a        = a;
650         arg[i].opd      = opd;
651         arg[i].nrow     = rpt;
652         arg[i].row      = j;
653
654         if ( mod )
655             ( arg[i].nrow++, mod-- );
656
657         j        += arg[i].nrow;
658     }
659                                     /* create the threads          */
660     for ( i=0; i < nCPU; ++i ) {
661         int    n;
662 #define THR_FLAGS    PTHREAD_SCOPE_SYSTEM
663         n = pthread_attr_init ( &tattr[i] );
664         n = pthread_attr_setscope ( &tattr[i], THR_FLAGS );
665         n = pthread_create ( &tid[i], &tattr[i], op, &arg[i] );
666         if ( n ) {
667             for ( ; i; i-- )
668
669                 pthread_kill( tid[i], SIGKILL );
670
671             return( n );
672         }
673     }
674
675                                     /* wait for them to finish    */
676     for ( i=0; i < nCPU; ++i )
677         pthread_join( tid[i], NULL );
678
679     return( 0 );
680 }

```

#### 5.4.4 Multi-processor machines

The one issue that so far has not been touched upon is that of multi-processor machines and how to get the Java threads to distribute properly across multiple CPUs. One will notice that within the C code, the function `sysconf()` (defined in `unistd.h`) is used (on

UNIX systems, NT differs slightly as already demonstrated) with the appropriate macro, such as `_SC_NPROCESSORS_ONLN` for Solaris and `_SC_NPROC_ONLN` for IRIX. The purpose of this is to determine the number of available CPUs so that an appropriate number of threads can be created to endeavour to get the best performance for the available CPUs. In the case of the multi-threaded solver, as one can see, the C version of the code creates one thread for each of the CPUs that are available for use. This has proved to be effective in C, and naturally we would like to be able to mimick this in Java.

Unfortunately, when we move to Java there are a number of issues that complicate matters. The first is that Java has no concept of the number of available CPUs on the system, and in order to use a command like `sysconf()`, it must be implemented natively and compiled to a shared object from which the Java virtual machine can gain access to it. The major problem with this of course is that there needs to be a module available on each system on which the code is to be run, which is inconvenient and can affect the portability of the code. In order to avoid the mandatory use of native methods, I have not implemented functionality in the main release of the code to determine the number of available CPUs (it is available in an optional support package that I shall discuss later). This is largely because it is not entirely necessary anyway. The only reason that `sysconf()` is used, is for when the number of CPUs is not specified on the command line; in that case the solver then uses the maximum number of available CPUs (determined through `sysconf()` which is not always optimal anyway due to thread contention problems on such a high number of CPUs. In addition to this, it is not appropriate to make such an assumption in Java, due to the way in which some Java Virtual Machines schedule threads; which leads me onto my next point.

### *Green threads*

The main reason behind a different approach for the default number of CPUs to be used is Java's green threading model. The green threading model is the simplest and most portable model, and is the original model used by Java Virtual Machines for scheduling threads. Green threads exist purely within the virtual machine, and are scheduled by the virtual machine - the operating system has no knowledge of these threads; as far as the operating system is concerned it is a single process running within a single thread of control. This kind of threading implementation is known as a pure user space model, and is the most portable, since it is not necessary for the operating system to have a threading system of its own. This has allowed Java virtual machines to be implemented on operating systems such as MS-DOS (using the Windows 3.1x GUI and Win32 subsystem) which do not support threads<sup>9</sup>.

Multiple threads are run a single thread at a time. When a new thread is to be run by the Java virtual machine, the information regarding the state of the current thread, such as the stack and program counter are saved, before the target thread's information replaces them. This way the stack of the target thread becomes the virtual machine's stack and the next instruction to be executed is that pointed to by the target thread's program counter. This is a logical overview of what happens, but as one can see, the virtual machine handles just one thread at a time swapping the relevant information in

---

<sup>9</sup>IBM has produced a port (ADK) of Sun's 1.0.2 Java virtual machine, it is approximately 4Mb in size and is available from IBM's alphaworks web site[?]

and out of the active context as necessary. The actual scheduling algorithm that is used to determine when a different thread should start running can differ between the various implementations of Java virtual machines that are available. The reference model states that green threads should be scheduled by priority inheritance, whereby the priority of a thread is temporarily increased if it is executing a critical section for which a thread of higher priority is waiting to complete (i.e. it is holding a synchronisation lock for the same shared data that the thread of higher priority wishes to modify). One may see differing scheduling methods such as round robin scheduling, although this is not a requirement.

Given that the green threading model is the most platform independent and easiest to implement, it can be found on many Java virtual machines, particularly UNIX ones. There are three classes of virtual machine that one will come across, those that implement just the green threading model, those that implement a mixture of green and native threading or those that implement only native threads. However, since the green threading model is the most prevalent, this is the one that we must assume the user of the solver will have. As a result, the number of available CPUs is set by default to 1, which can be increased from the command line for those with native threading, one could even do it under green threads although it wouldn't do much good! It is quite important to understand that where green threads are used within a virtual machine; because it is seen as a single process running on a single CPU, the threads *cannot* be distributed across CPUs, in fact it is quite likely that only one thread at a time will be able to execute (this of course depends on the scheduling algorithm used by the virtual machine). This problem is highlighted by the timings taken on `kilburn` using its JVM and default threading model (green threads), as one can see from the results presented in the performance section, no matter how many CPUs are specified, the execution time is the same, which serves to confirm the information presented here (i.e. that the code is executed on one CPU<sup>10</sup>, and potentially one thread at a time - though without reference to `sgi`'s implementation details one cannot be clear about this).

### *Native threads*

The other form of threading model that Java virtual machines can use is native threading where a mapping of some sort native threads (i.e., threads provided by an operating system library) and those within the Java virtual machine. The use of native threads has considerable benefits to green threads since the threads are now visible to the operating system and can therefore be scheduled by it, in addition to this, they can also be distributed across CPUs which is something we are of course interested in, given that green threads have turned out to be a disappointment for anything other than uniprocessor systems. Whilst native threading has great benefits, it also has a number of problems such as the fact that varying implementations result in vastly different performance on different systems.

### **Win32**

---

<sup>10</sup>This is further backed up by CPU usage figures from the command `top` which show CPU usage never to go over 99.5% (typically an indication that only one CPU is in use).

The way in which threads within the operating system and virtual machine are related does of course vary on different systems. For example, Win32 virtual machines (those running under Windows 95/98/NT) have a direct one to one mapping between operating system and virtual machine threads. These threads are scheduled in the same manner as processes (priority inheritance). The performance of the code on version 1.2 of Sun's Java virtual machine under Windows 95 is very good as one can see from the provided timings in the performance review. The details of the timings taken can be found in the performance review. The only thing that remains to be said about using the code under Win32 is that the Java virtual machine doesn't need to be invoked with any special parameters (only one threading model is used - native), simply the standard:

```
java Fe_mt <size> <CPUs>
```

Note that the number of CPUs would not normally be specified unless the code was run on a multi-processor machine under NT where SMP is properly supported.

### Solaris

Although a simple mapping prevails on Win32 virtual machines, the Solaris virtual machines are much more complex. The Solaris virtual machine employs LWPs (lightweight processes) from the Solaris native threading library, and typically the virtual machine will start off with one LWP. Additional LWPs are created by the Solaris threading library according to a set of rules which can be found in the book *Java Threads*[8] on page 151 along with a much more detailed description of Solaris and other threading models and how they are mapped onto threads within Java virtual machines. To summarise the process (as stated in *Java Threads*), the Java virtual machine effectively gets an allocation of LWPs equal to the number of threads that have been simultaneously blocked at any stage (i.e. through performing a system call requiring kernel intervention) plus one. This in practice means that one would typically get between 5 and 7 LWPs, where between 4 and 6 may be blocked, the remaining LWPs are available for threads within a Java program to be scheduled onto, with timesliced scheduling imposed (even if the threads are not of the same priority). This means that typically one can rely on having two LWPs onto which threads can be timesliced, this value is generally sufficient, although may require adjusting for multiprocessor machines with more than two CPUs to ensure that one gets desirable scaling - a general rule of thumb for any Solaris thread programming is that there should be one LWP for each CPU for best performance (as stated in "An Introduction to Multi-threaded Programming").

If better scaling is required, then the concurrency level (i.e., the number of available LWPs) must be increased. This is where the Java code becomes platform specific, since there is no way to set the concurrency level from Java itself - it has to be done via a native method compiled into a shared object (using the Solaris thread library function call `thr_setconcurrency()`). The timings that have been taken for the multi-threaded Java solver on `irwell` the 22 CPU Sun E6500 box, have been very respectable indeed and do not really indicate the need for any native calls, although the possibility that these timings could be further improved is exciting (something that could be looked at in the future). The timings taken were performed without modifications to the code in any way, and give performance that in the best case is only about half the speed of C which is better than most estimates for Java performance. The Java virtual machine under

Solaris 2.6 (version 1.1.3 is available on `irwell`, as with the Win32 virtual machine does not need to be invoked with any special parameters simply:

```
java Fe_mt <size> <CPUs>
```

For more details on the performance data obtained and how it was obtained, please refer to the performance review section.

## IRIX

IRIX is really the odd one out here. The main reason for this is that the Java virtual machine is produced by sgi rather than Sun Microsystems, and also that unfortunately, one has to resort to native methods to get the performance desired. So far the use of native methods has been avoided (even where recommended in the book - for Solaris threads) since performance has not been poor enough to warrant it's use, however, as one can see from the timings in the performance review, there is no escaping the fact that they are poor, and show poor scaling.

Under the latest sgi release of the IRIX Java virtual machine (version 3.1.1 which is equivalent to Sun's 1.1.6 release), one can use either green or native threads (specified by the flags `-green` and `-native` respectively). The green threads as one would expect, do not scale across CPUs, and the performance is identical per matrix order regardless of the number of CPUs specified on the command line (which follows) - this is quite clear to see from the results in the performance review. Where native threads are concerned, the virtual machine threads are mapped onto POSIX threads (`pthreads`), the ratio of the mapping is unclear and would need to be obtained from sgi. Although the threads do distribute across CPU (in improvement on green threading I admit), the performance is poor in comparison with Solaris and Win32. When `top` is run, the virtual machine tends to use a maximum of 396% when the code is executed unaltered (using `java -native Fe_mt <size> <CPUs>`) regardless of the number of CPUs specified on the command-line (it is effectively a performance ceiling). One can also see from the performance review data that the number of CPUs specified does not have a great impact on performance. Since the Java code alone is not enough to deliver the performance we desire we have no choice but to use native methods to assist us.

The key problem appears to be that the number of kernel execution vehicles onto which the scheduled, which isn't sufficiently high for the performance levels desired (this is similar to the issue with LWPs under Solaris already discussed). The only way to remedy this is to raise the concurrency level through the command `pthread_setconcurrency()` which has to be compiled into a shared object so that the Java virtual machine can access it. A module has been produced based on the one for Solaris/Win32 native threads in the book *Java Threads* (Page 156), except that this just caters for IRIX and is expressed in terms of `pthreads`. There are two parts to the module itself, first of all there is the Java interface to our native module, and then the native module itself which is written in C. Here is the code for the Java interface:

```
1  /** CPUSupport class to provide native pthread calls under IRIX to improve
2     * performance of native threads within the JVM.
3     *
4     * Java interface class.
```

```

5  *
6  * By A. J. Truhlar, Manchester Visualization Centre, 21/09/1999
7  *
8  * Note: This class is based on that presented in Chapter 6 of the book
9  * Java Threads by Scott Oaks and Henry Wong
10 * (published by O'Reilly & Associates Inc).
11 **/
12
13 import java.io.*;
14
15 public class CPUSupport {
16
17     static boolean loaded = false;
18
19     static {
20
21         try {
22             System.loadLibrary("CPUSupportIRIX"); // libCPUSupport.so
23             loaded = true;
24         } catch (Error e) {
25             System.out.println("No platform library for CPUSupport");
26         }
27     }
28 }
29
30 // native methods
31 private static native int getConcurrencyN();
32 private static native int setConcurrencyN(int i);
33 private static native int getNumProcessorsN();
34
35 // interface methods
36 public static int getConcurrency() {
37
38     if(!loaded)
39         // Assume green threads when library can't be found
40         return 1;
41
42     return getConcurrencyN();
43 }
44
45 public static void setConcurrency(int n) {
46
47     if (loaded)
48         setConcurrencyN(n);
49 }
50
51
52 public static int getNumProcessors() {
53
54     if (!loaded)
55         // Assume green threads when library can't be found
56         return 1;
57
58     return getNumProcessorsN();
59 }
60
61 }

```

```
62
63 }
```

As one can see, the class is very simple just containing three methods. We will only be using two of them, although the `getNumProcessors()` method could be used to determine the number of CPUs if we want (this calls `sysconf()`). The shared object that the Java virtual machine will dynamically link to at run-time is named `libCPUSupportIRIX.so`, and must be defined within a run-time linker path (more on this later). The `loadLibrary()` method of the `System` object is used to link to the module, and the methods within the module are given forward references so that they can be called from later on in this interface, for example:

```
private static native int setConcurrencyN(int i);
```

The keyword `native` indicates that it is defined within the loaded library. One will notice that only the IRIX library is loaded here, it is possible to load other libraries in the event of one not being found, and this is demonstrated within the book (for Solaris and Win32). It may be worthwhile developing a set of native libraries to accompany `libCPUSupportIRIX.so` for other platforms, although it is quite likely that concurrency levels would need to be different for different platforms, hence the current limitation to just the IRIX platform at the moment. Also from the tests carried out so far, the two platforms for which the package caters in the book (Solaris and Win32), provide very respectable performance levels that are close to those of the multi-threaded C code, and hence do not really warrant the use of native methods (although it would be interesting to see if these could be improved yet further).

Now, having seen the interface, the actual module definition in C is as follows:

```
1  /** CPUSupport module to provide native pthread calls under IRIX to improve
2   * performance of native threads within the JVM.
3   *
4   * IRIX Native pthread support module for JVM native thread scheduling.
5   *
6   * By A. J. Truhlar, Manchester Visualization Centre, 21/09/1999
7   *
8   * Note: This class is based on that presented in Chapter 6 of the book
9   * Java Threads by Scott Oaks and Henry Wong
10  * (published by O'Reilly & Associates Inc).
11  **/
12
13  /* compile:
14
15     prepare DSO:
16     =====
17
18     cc -n32 -D_POSIX_C_SOURCE=199506L -c CPUSupportIRIX.c -lpthread      19
20     ld -n32 -shared -all CPUSupportIRIX.o -lpthread -o libCPUSupportIRIX.so  -I/usr/java
21
22     environment:
23     =====
24
25     BASH:
```

```

26     export LD_LIBRARYN32_PATH=$LD_LIBRARYN32_PATH:sopath:.
27
28     csh:
29     setenv LD_LIBRARYN32_PATH $LD_LIBRARYN32_PATH:sopath:.
30
31     Java:
32     =====
33     javac CPUSupport.java
34
35     N.B: sopath is the path to the location of the shared object (.so).
36
37     */
38
39     #include <jni.h>
40     #include <pthread.h>
41     #include <unistd.h>
42
43     JNIEXPORT jint JNICALL Java_CPUSupport_getConcurrencyN( JNIEnv *env, jobject class ) {
44         return pthread_getconcurrency();
45     }
46
47     JNIEXPORT void JNICALL Java_CPUSupport_setConcurrencyN( JNIEnv *env, jobject class, jint n ) {
48         pthread_setconcurrency(n);
49     }
50
51     JNIEXPORT jint JNICALL Java_CPUSupport_getNumProcessorsN( JNIEnv *env, jobject class ) {
52         int num_threads;
53         num_threads = sysconf(_SC_NPROC_ONLN);
54         return num_threads;
55     }

```

The complete compilation details are also included in the introductory comment for the file. As one can see the process is involved but does work. The important thing to note from the compilation details is that support module must be compiled and linked using the new 32 bit (`-n32`) ABI, the IRIX Java virtual machine does not currently support the 64 bit ABI hence the need to use the new 32 bit ABI. It is possible to use the old 32 bit ABI (`-o32`) if necessary but the Java virtual machine defaults to the new 32 bit ABI (since the virtual machine is intended for use on IRIX 6.2 or greater systems that use the new ABI), so this is what has been used in this case. There is nothing particularly spectacular about the file, it simply invokes the appropriate native methods within the functions defined. When developing one's own JNI code, it is possible to use the utility `javah` to provide a header file with the correct function prototypes, this will make the job of implementation easier. To create a header file one does:

```
javah -jni CPUSupport
```

Now that a native interface class has been defined, it can now be put to use. This is simply a matter of importing the class and then invoking the required methods. For the multi-threaded solver, we simply add the import instruction at the start of the file:

```
54 import CPUSupport;
```

Then all that needs to be done is add the instructions to adjust the concurrency level, which are best placed when the threads are created, this could either be in main where some threads are created or within `MatrixOp()` where the processing is distributed across multiple CPs (where available) or both. The trial of native methods only adjusted the concurrency within `MatrixOp()`. The modification was simply the insertion of the two instructions needed; below is the loop in which threads are created (*j* indicating additional instructions):

```

626         for ( i=0; i < nCPU; ++i ) {
627
> 628             // raise concurrency level by 1 for each thread
> 629             int tmp=CPUSupport.getConcurrency();
> 630             CPUSupport.setConcurrency(tmp+1);
631
632             thrs[i] = new ExecThread( md, arg[i] );
633             thrs[i].start();
634
635         }
```

Having done this the code will now distribute across multiple CPUs better than before - `top` now shows 790%, and the execution time is noticeably reduced. As one can see from the figures in the performance review, the scaling is better than before and the execution time a little more respectable. Unfortunately, due to time pressures, only the method stated was used to modify the concurrency level. It is quite possible that a more in depth look will yield better results. As it is, when comparing the best cases of the `pthreads` solver and this Java port, one finds that the Java port is around 2.25 times slower than the C (a similar figure to on `irwell`).

#### 5.4.5 Reflection

[What has been achieved, issues still to be dealt with]

### 5.5 Testing the correctness of the ports

In order to test the correctness of the ports, I have written a small shell script to run each of the original and ported programs and compare the output (using `diff`). This is necessary since the solver programs can solve problems of order 1 to 100, and in the case of the mutli-threaded ports this can be done on between 1 and the maximum number of CPUs available (40 on `kilburn` and 22 on `irwell`). This of course leads to a large number of outputs that need to be checked, and given the nature of the output (groups of digits with a precision of between 6 and 12 decimal places - depending upon the version that is being run), this is not a trivial task. Therefore, as one can appreciate, the best way to check all of these is mechanically within a script, so that any potentially problematic cases are weeded out without spending many hours and potentially making several mistakes doing the job 'by eye'. The script basically tests each possible case by running the original executable and the ported executable and catenating their outputs to separate temporary files. These files are then compared using `diff` which indicates any discrepancies (when comparing the Java ports to the original C code, `diff -w` has

to be used to ignore whitespace since the spacing could not be exactly matched between output columns (as explained earlier)). Also notice that each case to be tested is echoed to the terminal so that the location of anomalies can be easily identified. The script takes the following options:

- `-s` : Compare single threaded programs only.
- `-m` : Compare multi-threaded programs only.
- `-sj`: Compare single threaded code without pedantic difference checking (i.e. ignore whitespace).
- `-mj`: Compare multi-threaded code without pedantic difference checking (i.e. ignore whitespace).

The script is implemented as follows (using the Bourne Again SHell (BASH)):

```
#!/bin/bash

# Shell script to test output consistency from ports
#
# Takes three arguments:
# 1) -m or -s Multi/single threaded test.
# 2) path of existing executable
# 3) path of ported executable.
#
# To turn of pedantic difference checking (i.e. ignore whitespace) add j to
# the end of the option so -s becomes -sj for example.

TMP_FILE1=/tmp/test_diff_1_$$$.tmp
TMP_FILE2=/tmp/test_diff_2_$$$.tmp
CPUS=40
SIZE=100
P_FLAG=0;

# Parse arguments
case 'echo $1 | sed -e "s/-//"' in
s ) S_FLAG=1;M_FLAG=0;;
m ) S_FLAG=0;M_FLAG=1;;
sj ) S_FLAG=1;M_FLAG=0;W="-w";;
mj ) S_FLAG=0;M_FLAG=1;W="-w";;
esac

PR2=$2
PR3=$3

# Produce performance stats for MT Java solver
# Native threads
if 'test "$M_FLAG" = "1"'
then
  C=1
  while test "$C" -le $CPUS
  do
    N=1
```

```

while test "$N" -le $SIZE
do
    $PR2 $N $C > $TMP_FILE1
    $PR3 $N $C > $TMP_FILE2
    echo "Testing size $N on $C CPUs:"
    diff $W $TMP_FILE1 $TMP_FILE2
    N='expr $N + 1'
done
C='expr $C + 1'
done
fi

if 'test "$S_FLAG" = "1"'
then
    N=1
    while test "$N" -le $SIZE
    do
        $PR2 $N > $TMP_FILE1
        $PR3 $N > $TMP_FILE2
        echo "Testing size $N:"
        diff $W $TMP_FILE1 $TMP_FILE2
        N='expr $N + 1'
    done
fi

rm -rf $TMP_FILE1 $TMP_FILE2

exit 0

```

The script is simple to use, one simply invokes using the flags commented within the source specifying the programs whose output you wish to compare, so for example, to compare two single threaded programs `a.out` and `b.out` that should give identical results, one would use:

```
test.sh -s a.out b.out > output.txt
```

One should note that where additional arguments need to be supplied to the program or in the case of the Java code, the virtual machine needs to be started, the invoking shell commands need to be enclosed in quotation marks so that they are treated as one argument to my script. So for example, to check the correctness of the output from my single threaded Java port along with the original single threaded C solver, one would use:

```
test.sh -sj int "java Fe" > output.txt
```

As one can see the script is easy to use and lends itself to the kind of “industrial checking” that one is simply unable to do manually.

### *Tests performed*

The script has been used to check the first port done which was the port from Solaris threads to pthreads of the multi-threaded solver. At the time the port was completed I

did not have full access to a Solaris machine so it is only recently that this has been done. The comparison was performed on `irwell` using original Solaris code and the `pthread` port and found no differences in the output. Later versions of the port compiled with SPILT have also been tested (on `kilburn`), but not industrially since the compiling with SPILT is effectively the same as directly porting the code by hand and the spot checks showed no differences. The NT code has not been so rigerously tested due to time pressures, but has given me no reason to doubt it's integrity (quick comparisons with the single threaded code were correct).

Moving onto the Java, this is where things become a little more complicated. All of the release versions of the code have been tested, but with some dismaying results as one will see.

### The single threaded Java port

First of all the single threaded solver. Most cases showed one difference, which occurs in the output of the Chebyshev coefficients for  $f(x)$ ; the example below is the default case - a matrix of size 30):

```
Testing size 30:
49c49
<      [ 9]:  -0.000001
---
>      [ 9]:   0.000000
```

The output from the Java solver is the bottom line, and of course the output from the C solver is the top line. Here it is a simple case of rounding, since the values are the same. When formatting is removed from the Java program, this line is printed in scientific form to between 15 and 16 (usually 15) decimal places, if one modifies the C program to use the format specifier `%.16G` so that it also prints the output in scientific form to 15 decimal<sup>11</sup> places, the values are identical, as we can see. Here is the Java output for this line without formatting (which defaults to `%.16G`):

```
[9]:  -5.891295333913212E-7
```

And here is the output from the C solver set to output at `%.16G`:

```
[ 9]:  -5.891295333913212E-07
```

As one can clearly see this is purely a rounding error, over which we have little control. This particular case is a persistent error and occurs in every possible matrix size from 10 onwards. In addition we have a additonal but similar problem with the case for a matrix of size 6:

```
Testing size 6:
35c35
<      [ 5]:  -0.004491
---
>      [ 5]:  -0.004490
```

---

<sup>11</sup>16 is intentionally used here - it gave the output below, which is to 15 decimal places.

Again it is the same problem as we can see by comparing the crude Java output (no applied formatting just it's default as before), and C output when adjusted to use the `%.16G` format specifier. First the Java:

```
[5]: -0.0044905080308394185
```

Now the C output in `%.16G` form:

```
[ 5]: -0.004490508030839419
```

Here we get 18 decimal places worth because the value is large enough not to warrant being expressed in scientific form<sup>12</sup> (we get 19 in the Java output because exponents are normally expressed with a leading zero when less than 10). Again though, we can see that the numbers are the same, the C output having been rounded up. To convince ourselves that these numbers are indeed the same, here is the output from the C program in `%.17G` format:

```
[ 5]: -0.0044905080308394185
```

As we can see the values are in fact the same, and this is yet another rounding error resulting from the use of the decimal format objects.

Apart from the two output errors stated, the cases for matrices of size 8 and size 10 have a slight error in the output for the matrices section of the output. This is again just one value (here is the case for a matrix of size 8):

```
Testing size 8:
30c30
<      0.000000      0.005719      0.000000      -0.004156      0.000000      0.999610
---
>      0.000000      0.005718      0.000000      -0.004156      0.000000      0.999610
```

Further examination as in the previous cases shows that it is a rounding error by the decimal format object.

So those are all of the errors found when comparing the output of the single threaded Java port and the original single threaded solver in C (they are errors provided that the code from Sun is correct of course). One should note carefully that they are only errors in the output where the decimal format objects have failed to round numbers in the same way that `printf()` does, the internal representations of the numbers is correct as we have seen. If there were problems with the calculations themselves the output differences would be considerably greater! In practice most cases (apart from 6, 8 & 10), will just have the error on row nine of the Chebyshev coefficients for  $f(x)$ .

Apart from this the only other point worth mentioning is that for the case of a matrix size 1, the representations for not-a-number differ between C and Java (of course this isn't an error):

---

<sup>12</sup>This is so that it uses up the same amount of space as those numbers that are expressed in exponential form, which are given to 15 decimal places.

```

Testing size 1:
3c3
<      [ 0]:  nan0x10000000
----
>      [ 0]:  ?
7c7
<      nan0x10000000
----
>      ?
11c11
<      nan0x10000000
----
>      ?

```

One should note that the details just shown are for the IRIX virtual machine on `kilburn` and could well be different elsewhere. The one system on which I know they are different is Solaris 2.6 on `irwell` which is running `jdk 1.1.3`. Here the errors are again rounding errors, but much more prevalent and obvious. For example (matrix size 30 - default):

```

4c4
<      [ 1]:  -2.914379
----
>      [ 1]:  -2.914378
8c8
<      [ 5]:  -0.011545
----
>      [ 5]:  -0.011544
12c12
<      [ 9]:  -0.000002
----
>      [ 9]:  -0.000001
43c43
<      [ 3]:   0.138072
----
>      [ 3]:   0.138071
45c45
<      [ 5]:  -0.004491
----
>      [ 5]:  -0.004490
47c47
<      [ 7]:   0.000068
----
>      [ 7]:   0.000067
49c49
<      [ 9]:  -0.000001
----
>      [ 9]:  -0.000000
73,77c73,77
<      [ 0]:   0.000000      1.000000
<      [ 1]:   0.314159      0.951057
<      [ 2]:   0.628319      0.809017
<      [ 3]:   0.942478      0.587785
<      [ 4]:   1.256637      0.309017
----
>      [ 0]:   0.000000      0.999999

```

```

>      [ 1]:    0.314159      0.951056
>      [ 2]:    0.628318      0.809016
>      [ 3]:    0.942477      0.587785
>      [ 4]:    1.256637      0.309016
79,83c79,83
<      [ 6]:    1.884956     -0.309017
<      [ 7]:    2.199115     -0.587785
<      [ 8]:    2.513274     -0.809017
<      [ 9]:    2.827433     -0.951057
<      [10]:    3.141593     -1.000000
---
>      [ 6]:    1.884955     -0.309016
>      [ 7]:    2.199114     -0.587785
>      [ 8]:    2.513274     -0.809016
>      [ 9]:    2.827433     -0.951056
>      [10]:    3.141592     -1.000000

```

As one can see these are again clearly rounding errors and the same sort of deeper investigation that we did for those on `kilburn` reveals this (which isn't replicated here to save space). Having run the single threaded Java code on another Solaris machine that is running Solaris 2.7 with jdk 1.2, the situation is more like that on `kilburn` where there are just a small number of differences. It is quite clear to see when comparing the results that these are rounding errors, but it is an annoying inconsistency between platforms, particularly when one considers that the C code supplied by Sun gives the same results on all the platforms I have tried (even if when the numbers are output to a greater degree of precision and differ at that stage; the level of precision specified by `printf()`'s `%10.6f` format string yields the same output on all the platforms tried).

It is therefore clear that if the goal is to have identical output to that specified within a C program, the rounding provided by `DecimalFormat` leaves a lot to be desired in terms of cross platform consistency, which is a shame when one considers that the results themselves are quite correct as has been shown.

### *The multi-threaded Java port*

For the multi-threaded port things become more complicated. Here we have two cases to consider, the first is when the code is run in green threaded mode (which is the least susceptible to errors since the threads are likely to join in the correct order), and the second is when the code is run in native mode. Two separate runs have been carried out on `kilburn`, and the results differ in each case. First the good news (if you can call it that):

### Green threads

When comparing the output of the multi-threaded Java port running in green threaded mode, the differences between that and the C code are similar to those that we have seen for the single threaded solvers, i.e. rounding errors. Since on both Solaris and IRIX, all of the threads join correctly due to the way green threads are scheduled (discussed earlier), there are no inconsistencies due to threads joining incorrectly and one part of

the calculation continuing before another is complete<sup>13</sup>. All we have are more rounding errors. The same errors occur at the same matrix sizes (1, 10, 15, 50, 61, 67, and 75) for each number of CPUs from 1 to 40 (which is what we expect since the output should be the same regardless of the number of CPUs). Here is an example, (matrix size 75 on 14 CPUs)<sup>14</sup>:

```
Testing size 75 on 14 CPUs:
166c166
<      [ 7]:      2.199115          -0.587785252293
---
>      [ 7]:      2.199115          -0.587785252292
```

As one can see this is exactly the same kind of error that was present when checking the single threaded solver. Thankfully these are only rounding errors and so do not point to anything fundamentally wrong with the calculating code, simply the way it is represented when output (it's those `DecimalFormat` objects again).

### Native threads

Looking at the code when run using native threads (without the IRIX performance enhancement), the situation is altogether less pleasing (the bad news). In this case we not only have the same rounding errors as before, but also arbitrary inconsistencies where the output is completely incorrect to that of the multi-threaded C solver. There are two types for this, the first is where presumably the threads do not join at all, and all you get is the output from the first thread (not run in parallel) which is the Chebyshev coefficients of  $g(x)$ , the remainder of the output is not produced. An example of this is a matrix of size 7 distributed across 37 CPUs:

```
Testing size 7 on 37 CPUs:
10,33d9
<
< Chebyshev Coefficients for f(x):
<
<      [ 0]:      0.000000000000
<      [ 1]:     -1.133647326518
<      [ 2]:      0.000000000000
<      [ 3]:      0.138071084180
<      [ 4]:      0.000000000000
<      [ 5]:     -0.004423009662
<      [ 6]:      0.000000000000
<
< Approximate f(x):
<
<      [ 0]:      0.000000          0.999999252000
<      [ 1]:      0.314159          0.951108611451
<      [ 2]:      0.628319          0.809087489651
```

<sup>13</sup>One must bear in mind, however, that although the virtual machines used have executed the threads in order of creation when in green threaded mode, there is no absolute guarantee that this will be the case on all platforms.

<sup>14</sup>The error occurs on row 7 of the Approximate  $f(x)$  section of the output

```

< [ 3]: 0.942478 0.587707266933
< [ 4]: 1.256637 0.308891815044
< [ 5]: 1.570796 0.000000000000
< [ 6]: 1.884956 -0.308891815044
< [ 7]: 2.199115 -0.587707266933
< [ 8]: 2.513274 -0.809087489651
< [ 9]: 2.827433 -0.951108611451
< [10]: 3.141593 -0.999999252000

```

Another and more disturbing problem is where all of the output is produced but the values are completely incorrect, suggesting that the threads have not joined in the correct order, hence producing completely incorrect output. An example of this is the output for a matrix of size 18 distributed across 33 CPUs:

```

Testing size 18 on 33 CPUs:
24,41c24,41
< [ 0]: 0.000000000000
< [ 1]: -1.133648177812
< [ 2]: 0.000000000000
< [ 3]: 0.138071776587
< [ 4]: 0.000000000000
< [ 5]: -0.004490714247
< [ 6]: 0.000000000000
< [ 7]: 0.000067701276
< [ 8]: 0.000000000000
< [ 9]: -0.000000589130
< [10]: 0.000000000000
< [11]: 0.000000003338
< [12]: 0.000000000000
< [13]: -0.000000000013
< [14]: 0.000000000000
< [15]: 0.000000000000
< [16]: 0.000000000000
< [17]: 0.000000000000
---
> [ 0]: -0.104396192479
> [ 1]: -1.097249895199
> [ 2]: 0.146128402830
> [ 3]: 0.309253451896
> [ 4]: -0.001855401544
> [ 5]: -0.178339358576
> [ 6]: -0.125792599803
> [ 7]: 0.064829224345
> [ 8]: 0.157215983658
> [ 9]: 0.058377946628
> [10]: -0.088165431709
> [11]: -0.110659754860
> [12]: -0.013089154640
> [13]: 0.071249703225
> [14]: 0.064027018141
> [15]: 0.006685135297
> [16]: -0.034072624454
> [17]: -0.022391689351
45,55c45,55
< [ 0]: 0.000000 1.000000000000

```

```

<      [ 1]:    0.314159          0.951056516295
<      [ 2]:    0.628319          0.809016994375
<      [ 3]:    0.942478          0.587785252292
<      [ 4]:    1.256637          0.309016994375
<      [ 5]:    1.570796          0.000000000000
<      [ 6]:    1.884956         -0.309016994375
<      [ 7]:    2.199115         -0.587785252292
<      [ 8]:    2.513274         -0.809016994375
<      [ 9]:    2.827433         -0.951056516295
<      [10]:    3.141593         -1.000000000000
---
>      [ 0]:    0.000000          0.898245236595
>      [ 1]:    0.314159          0.861889730364
>      [ 2]:    0.628319          0.727309250161
>      [ 3]:    0.942478          0.518202390448
>      [ 4]:    1.256637          0.288499452359
>      [ 5]:    1.570796          0.007605221084
>      [ 6]:    1.884956         -0.558668652506
>      [ 7]:    2.199115         -1.714237696380
>      [ 8]:    2.513274         -0.739483363649
>      [ 9]:    2.827433         -0.914814077171
>      [10]:    3.141593         -0.898245236595

```

The really worrying issue here is that this happens arbitrarily and is difficult to reproduce. All of the individual tests on the command line returned the correct results, it is only when the code is run industrially that such weaknesses show through. So if for instance one wanted to reproduce the inconsistencies shown above; one would probably find that when tested from the command line the output would be correct, and even if done through a script, the output may be correct with the error occurring at a different configuration of matrix size and number of CPUs. This of course means that the code is not very reliable and only serves to increase doubts over method references (here they are being used for a purpose they were not really intended for).

One cannot be completely sure of the problem, but the following hypothesis seems to make sense when compared to use of the code in practice. One will notice that within the program the sole purpose of the `ExecThread` class is to invoke the method reference that is passed to it. One might then be under the impression that the invoked method is executed within the control of that thread. If this were true however, and a set of `ExecThreads` were created in a given order and completed execution of the methods invoked within them before terminating naturally, then they could be joined in order and one would always get the correct results, even when these `ExecThreads` are distributed across multiple CPUs. However, if one considers the case where it *may* be possible for the `ExecThreads` to terminate naturally some time before the invoked method has terminated naturally; then it is quite possible for incorrect and inconsistent results to be produced. This is particularly true when one considers that `join()` can be invoked on a thread that has been terminated (this will be silently ignored and the next instruction will be executed). This seems to be a plausible explanation, although it is not entirely reassuring to think of the method that one wishes to execute within the thread being executed elsewhere. The reason for this suggestion is that it is quite possible for one to see the problems the virtual machine has in determining when a thread has terminated naturally given that the return type of all the functions used is `void` so the object returned by reflection (containing the wrapped up return value) is not used (if it were then the invoked method must terminate before the `ExecThread`). The sole purpose of `ExecThread` is to invoke the methods passed in, and it may just be the case that it is only that that is done, i.e. to merely invoke the method before terminating. This is something that is not clear, and hence worth mentioning for future reference, it certainly highlights the potential difficulties with this approach.

### A final desperate attempt...

To try and test whether this is the case or not, I added a `boolean` to the `MatrixOpArg` class that could be set by the invoked method to `true` immediately before it returns:

```

public class MatrixOpArg {

    matrix_t    a; /* the matrix */
    int         row; /* starting row number */
    int         nrow; /* number of rows */
    boolean complete = false; /* complete? */

    // copy constructor for instantiation in ExecThread
    public MatrixOpArg( MatrixOpArg tmp ) {

        a = tmp.a;
        row = tmp.row;
        nrow = tmp.nrow;
        complete = tmp.complete;

    }

    public MatrixOpArg() {

    }

} // end of class: MatrixOpArg

```

This also required a modification to the `ExecThread` class so that it would wait if the boolean was `false`, thus ensuring that the thread does not terminate before the invoked method:

```

public void t_mop( MatrixOpArg arg ) {

    int     i;
    int     lim;
    double[] [] m = new double[N][N];

    m       = arg.a.m;
    lim     = ( i = arg.row ) + arg.nrow;

    for ( ; i < lim; ++i ) {
        int     j     = ( i % 2 != 0 )? 1 : 0;
        int     k     = ( i % 2 != 0 )? 0 : 1;

        for ( ; k < n; k+=2 )
            m[i][k] = 0.0;

        for ( ; j < n; j+=2 )
            m[i][j] = 0.5 * ( 1.0/(i+j+1.0) - 1.0/(i+j-1.0) + 1.0/(i-j+1.0) - 1.0/(i-j-1.0) );
    }

    arg.complete = true;

}

```

The final modification required is to set the new field of `MatrixOpArg` within each of the invoked methods, so for example `t_mop()` appears as follows:

```

public class ExecThread extends Thread {

```

```

private Method md2;
private Object[] w_args;
private MatrixOpArg arg2;

public ExecThread( Method md, MatrixOpArg arg ) {

    if ( arg != null ) {
        arg2 = new MatrixOpArg(arg);
        Object tmp[] = arg2 ;
        w_args = tmp;
    }

    md2 = md;

}

public void run() {

    try {
        md2.invoke( new thr_fns(), w_args );
    } catch (IllegalAccessException e) {
        System.err.println(thr_fns.class.getName() + e);
    } catch (InvocationTargetException e) {
        // System.err.println( thr_fns.class.getName() + ": " + e + " -> " + e.getTargetException() );
    };

    if (arg2 != null) {
        while( !arg2.complete ) {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                System.exit(1);
            };
        }
    }

    return;

}

} // end of class ExecThread

```

To be completely sure of success some action needs to be taken for those threads started within main. Here the area where the largest number of threads is created, has been dealt with. This potential solution seems to back up some of what has been said, since now there are no cases where a complete output (all sets of coefficients and the approximations) is given and is completely incorrect. However, there are cases where the code waits forever (usually resulting in the job being killed when the 10 minute limit for interactive CPU usage is reached<sup>15</sup>) or crashes with a core dump (bus error or segmentation violation). So whilst method references may not be entirely reliable, the code may not be either - since there is a known problem of `ArrayIndexOutOfBoundsException` being thrown on matrix sizes between 96 and 100 which is where such crashes occur. Unfortunately I have been unable to trace the fault since the numerical code is logically unchanged from the C (just the threading). The fault does not show in output either, but since this happens in the invoked methods it may indeed be a source of the problems. The release version of the code does not include these booleans (this is a very late addition),

<sup>15</sup>This could probably be avoided by using the form of `join()` that takes a timeout value in milliseconds, so that if after this given time, the thread had not joined, the program could terminate.

but seems to work on demand (the failures mentioned have all happened under brute force testing using the testing script shown earlier). That said, problems can of course occur, it is worth noting that getting the multi-threaded Java code to work happened very late on in the project and as such there has not been much time to sort out problems, the best that can be done with the time available is to explain all known issues for future reference with some possible suggestions for their existence (they may not be entirely correct, but should provide at least some starting point).

Anyway, to conclude this rather long section; there are intermittent problems that do show out under brute force testing of the code; although in general when these faults do not show up, the only problems are the rounding errors which are also present on the single threaded code (these are relatively minor as we have seen - nonetheless annoying though). At least by having done brute force testing the problems have been revealed, which one can be fairly certain would not have been picked up if testing had been left at just a few invocations on the command line then running `diff` on the output; it is only through at least several hundred invocations that such subtle problems surface. Unless stated, testing was performed on `kilburn`.

The use of method references within threads is not documented, and general documentation about them seems to suggest that they are best avoided (much of this discovered since their use was decided upon). An example of the comments one will find on method references is shown below which is taken from page 222 of the latest edition of the book *Java in a Nutshell*[?]:

Java does not allow methods to be passed directly as data values, but the Reflection API makes it possible for methods passed by name to be invoked indirectly. Note that this technique is not particularly efficient. For asynchronous event handling in a GUI though: indirect method invocation through the Reflection API will always be much faster than the response time required by the limits of human perception. Invoking a method by name is not an appropriate technique, however, when repetitive, synchronous calls are required. Thus, you should not use this technique for passing a comparison method to a sorting routine or passing a filename filter to a directory listing method. In cases like these, you should use the standard technique of implementing a class that contains the desired method and passing an instance of the class to the appropriate routine.

These comments seem to further suggest that whilst allowing a solution to be reached (something that could not be seen when every function was implemented within a separate type of thread), the use of method references may not have been the best way forward. Whilst the code runs reliably when used in green threaded mode, and there is a performance increase, it is not always reliable when used in native mode.

## 5.6 Performance analysis

This section contains some basic comparative details showing the difference in execution time between original and ported code, and variants of the ports such as manually inlined and compiler inlined code. Wherever possible graphs will be used over data sets, although there may be occasions where the latter is rather more appropriate than the former. All times compared are 'real' execution times (i.e. the actual time that is taken before interactive control within the shell is returned to the user following program invocation). All timings have been taken using the UNIX `time` command all within the control of a timing script (unless otherwise stated). Two scripts exist, one for IRIX<sup>16</sup> and the other for Solaris<sup>17</sup>.

For those interested, the scripts used are shown in the sections covering performance on the machines.

<sup>16</sup>The only reason that this is mentioned separately is that it was developed beforehand and was the one used for all of the timings on `kilburn`.

<sup>17</sup>A separate script had to be developed for Solaris, since the original script which captured the output from `time` directly from the standard error stream failed to work correctly. The only way to successfully capture this output on Solaris was to have a separate shell script that simply invoked the `time` command on a given program, and then within the main timing script the output from this 'wrapper' shell script could be easily captured. For some reason capturing the output directly from `time` under Solaris is problematic. As things stand the script has been tested and found to work on IRIX, Linux and Solaris

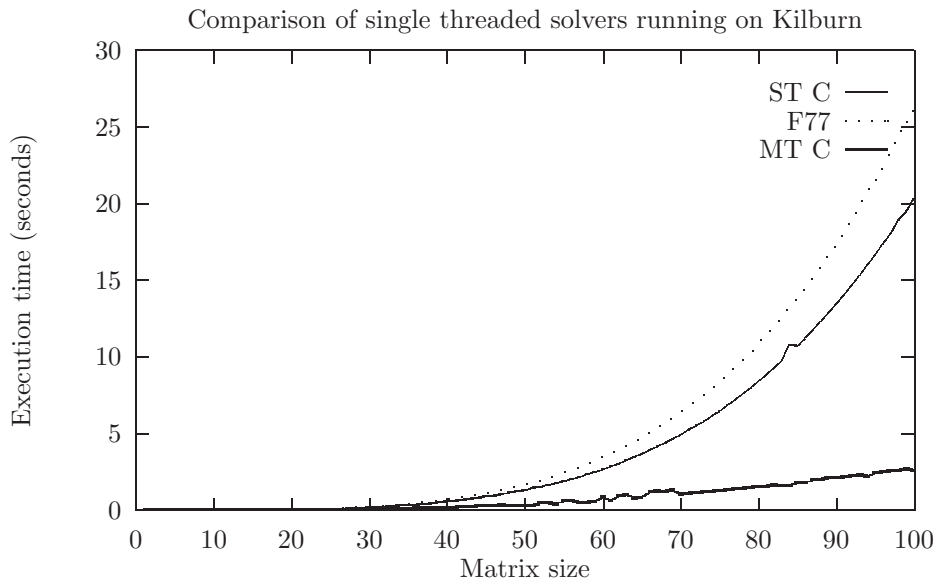


Figure 3: Initial comparison of the single threaded Fortran and C solvers and the multi-threaded C solver on kilburn

### 5.6.1 Using a 40 CPU Silicon Graphics O2000 system

Most of the work has been carried out on kilburn our 40 CPU (R10000 @ 195MHz) Origin 2000 system, and as a result the most extensive set of timings are available from this system. All of the timings were taken interactively to ensure that a maximum of 40 CPUs were available to the processes; no restrictions were placed on the number of available CPUs by using a resource management system such as the resource manager *miser*. Another important point to note is that all timings are taken with all I/O turned off from the program, so that I/O delays do not distort the true performance of the program. The results are categorised according to the particular port and area they are related to:

#### General comparison

At the outset of the project, there were three source files for this solver package, they were the original F77 code that Sun worked on, then the C port that was made of this at the Solaris migration support centre, and of course the multi-threaded version using Solaris threads that they developed. To start with here are the results

#### Initial timings

The graph in figure [?] shows the real execution time of the original code from Sun Microsystems. Note that the multi-threaded code is run at its optimal number of CPUs (9 CPUs):

As one can clearly see the multi-threaded code offers considerably better performance being around 7 times faster than the single threaded C code and over 8 times faster than the original Fortran solver (although to be fair this was re-written by the team at Sun, it was not simply a direct translation from Fortran into C). It should be noted that the multi-threaded solver used here was the *pthread*s port of the Solaris code. The improvement between the single and multi-threaded C solvers is between 4.2x starting with a matrix of order 50, and increasing to 7.9x for a matrix of order 100. These compare with an improvement of between 5.3x and 9.2x on the 8x50MHz CPU, SS1000 with 512Mb of RAM running Solaris 2.3 — which was the test system that Sun used during their case study.

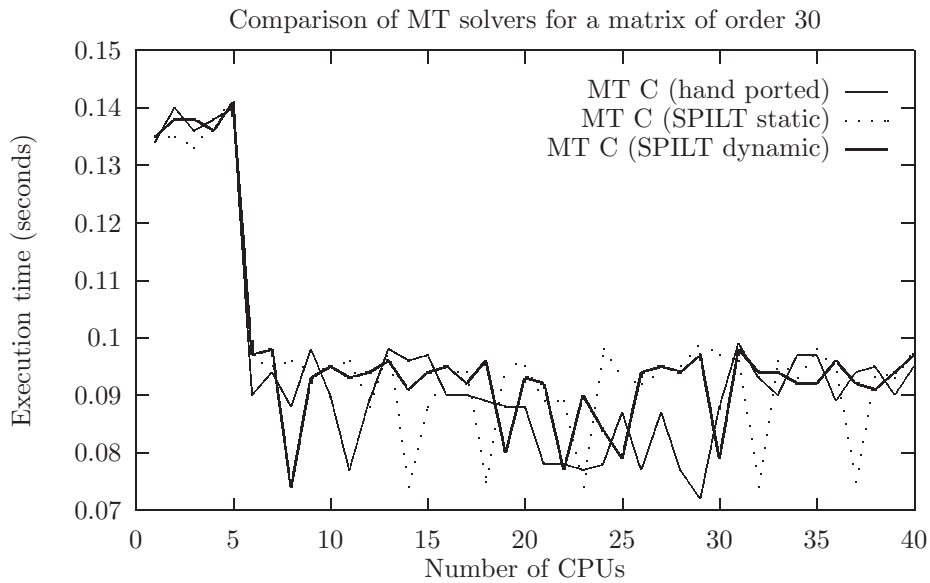


Figure 4: Comparison of hand-ported and SPILT compiled multi-threaded solvers along with the original single threaded solver – matrix order 30.

### More on the pthreads port

As one will be aware there were two approaches taken to getting the original Solaris threadedc solver to run on `kilburn`, they were to first of all convert the code by hand (i.e. change all the Solaris API calls to the equivalent `pthread`s ones), and secondly to use the SPILT package discussed earlier on this report. In order to demonstrate the performance increase and scalability of the code, the following graphs show the execution times of both versions of the multi-threaded code (the hand ported code and that compiled with the SPILT package). The graphs are shown for the default case — matrix order 30, and also matrix orders 50, 75 and 100:

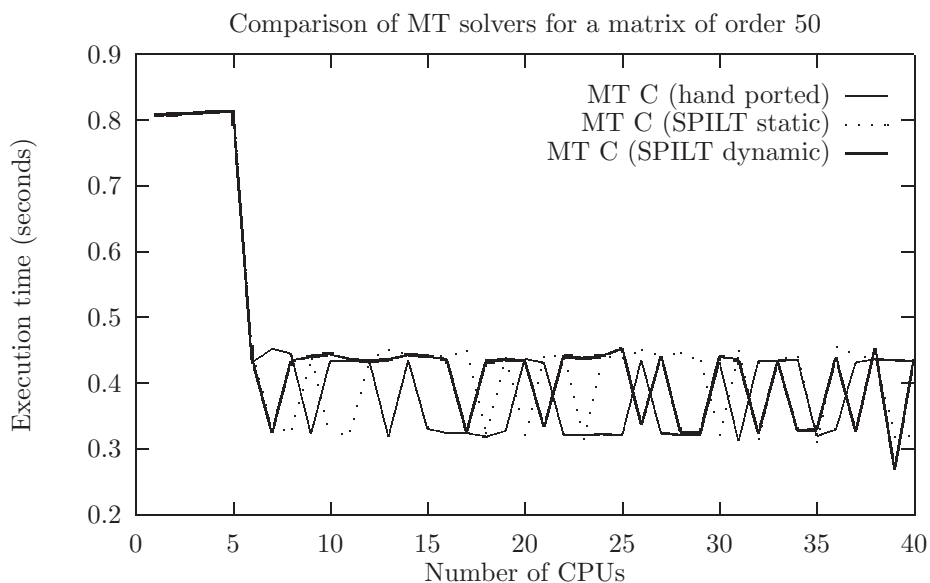


Figure 5: Comparison of hand-ported and SPILT compiled multi-threaded solvers along with the original single threaded solver – matrix order 50.

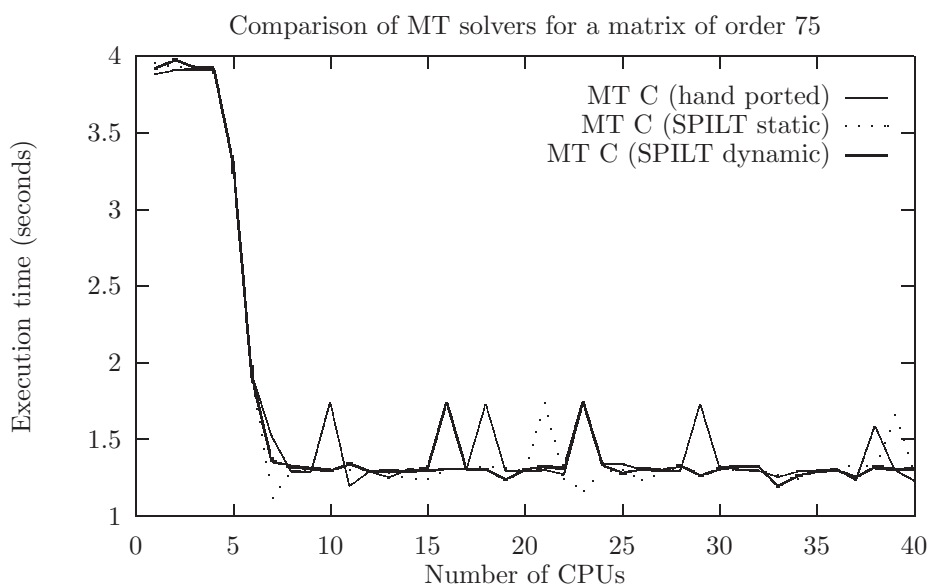


Figure 6: Comparison of hand-ported and SPILT compiled multi-threaded solvers along with the original single threaded solver – matrix order 75.

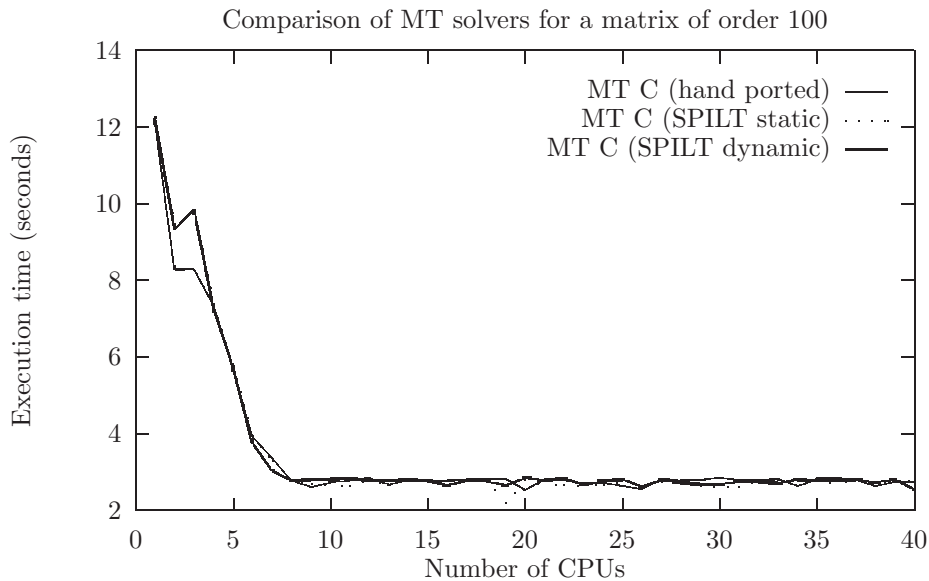


Figure 7: Comparison of hand-ported and SPILT compiled multi-threaded solvers along with the original single threaded solver – matrix order 100.

Despite the spikes within the graphs (probably due to being run interactively), one can see that there is no appreciable difference in execution time between the solvers. All exhibit the same behaviour where the execution time is reduced significantly before plateauing at around 9 CPUs after which there is no appreciable improvement. The timings do quite obviously fluctuate and this would be true if just one of the programs were timed three times over, and bearing in mind the scale of difference, it is quite clear to see that these fluctuations are not a problem in implementation, since they are random enough to suggest that they are temporary resource allocation glitches (the timings were *not* done within a resource manager). It is quite clear to see, however, that using SPILT is just as much an acceptable solution in terms of performance as hand porting the code.

### Single threaded C vs single threaded Java

The next stage of the work of course involved porting the single threaded solver to Java in order to iron out any language difference problems before the multi-threaded port was started. The port to Java worked, but worked slowly and was appreciably slower than the C solver, as one can see from the graph in figure[?].

As one can see, the performance is worse than both the C and Fortran-77 solvers, with an increase in execution time of between 7.4x for a matrix of order 30 and 2.5x for a matrix of order 100. Interestingly though much of the slowdown on matrices of smaller order is due to initialisation overhead since the code is 3.5x slower than the C for solving a matrix of order 50, the main degradation being up to that point (where the C code is at its fastest). When comparing the performance to that of the original Fortran-77 solver, one can see that although not quite as bad, the same problems are evident, with the increase in execution time ranging from 5.9x for a matrix of order 30 and 1.9x for a matrix of order 100. So whilst the code works, we have clearly paid a high price for converting it to Java; the initialisation overheads clearly demonstrate the overhead in initialising the virtual machine<sup>18</sup>.

### Compiler vs manual function in-lining

<sup>18</sup>A slightly fairer comparison would involve using the UNIX native Java compiler written in C — guavac

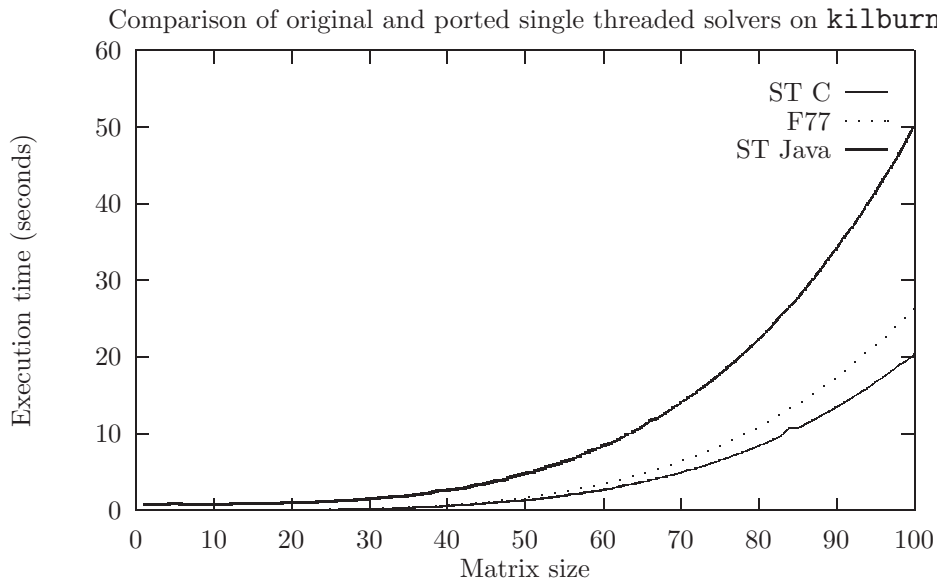


Figure 8: Comparison of the original single-threaded Fortran-77 and C solvers against the single threaded Java port

One of the major areas of concern when porting the code to Java was that functions would not be in-lined which would be particularly critical for things like parameterised macros which had to be converted to function calls. It was found however, that basic functions were in-lined, as a performance comparison between two versions of the solver where one had all minor functions manually in-lined, had shown<sup>19</sup> As one can see from the table below, the manually in-lined code is not appreciably different from the standare code, hence suggestion that the compiler performs automatic inlining of basic functions.

### *Compiler optimisation or not?*

Given the depressing performance of the single threaded Java port of the solver program, it was worth taking the desperate measure of trying compiler optimisation. Most Java bytecode compilers have the `-O` flag for optimisation, unlike optimisation on C compilers, this is the only level of optimisation available. Unfortunately the performance difference was so negligible that it was unnoticed! Therefore for this particular case, optimisation won't make the performance figures look any better, One can quite clearly see that this is the case from the table below:

### Multi-threaded C vs multi-threaded Java

The next stage of the work was of course the multi-threaded Java port of the Solaris threaded C solver (the work was actually based on the `pthread`s solver because the work was carried out on *kilburn* and this was more convenient, as one will have seen though, there is no real difference, certainly not when it comes to converting the code to Java!). There have been a number of problems in this area and at the time of the final presentation, the code performed uniformly on any number of available CPUs because it would only work in green threaded mode. Getting the code to run in native mode has since been possible due to the removal of a number of bugs. As one will be aware there are currently two versions of the code with three modes of operation between them. The first is a generic version that should work either in green or native mode but has a weakness in terms of performance on *kilburn* and the second is a modified version of the code that will perform considerably better but is specific to native threads on an IRIX virtual machine (it uses native code via the Java Native Interface).

<sup>19</sup>The code and timing data is available from the downloads area of the project web site.

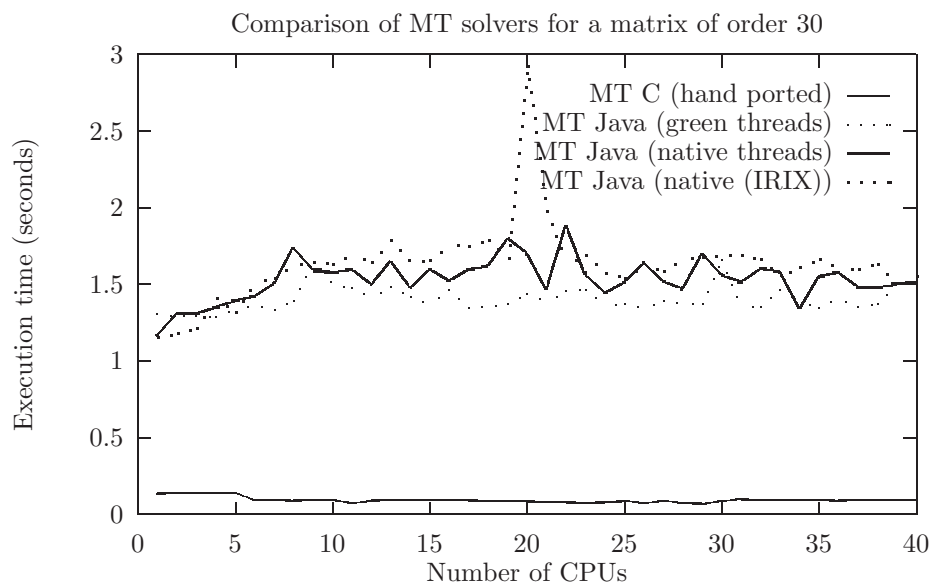


Figure 9: Comparison of the multi-threaded Java solver in green and native mode and also the multi-threaded C solver — matrix order: 30.

Figure[?] compares the multi-threaded Java solver in both green and native mode (along with the code modified for better performance under IRIX). In addition to this the values for the multi-threaded C solver are given to allow a comparison. As before the matrix orders 30, 50, 75, and 100 are used.

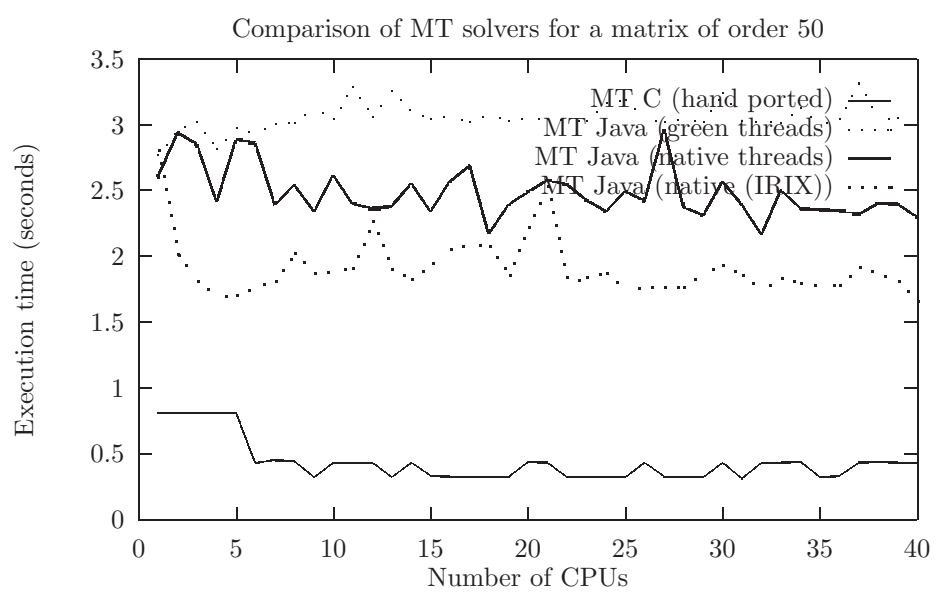


Figure 10: Comparison of the multi-threaded Java solver in green and native mode and also the multi-threaded C solver — matrix order: 50.

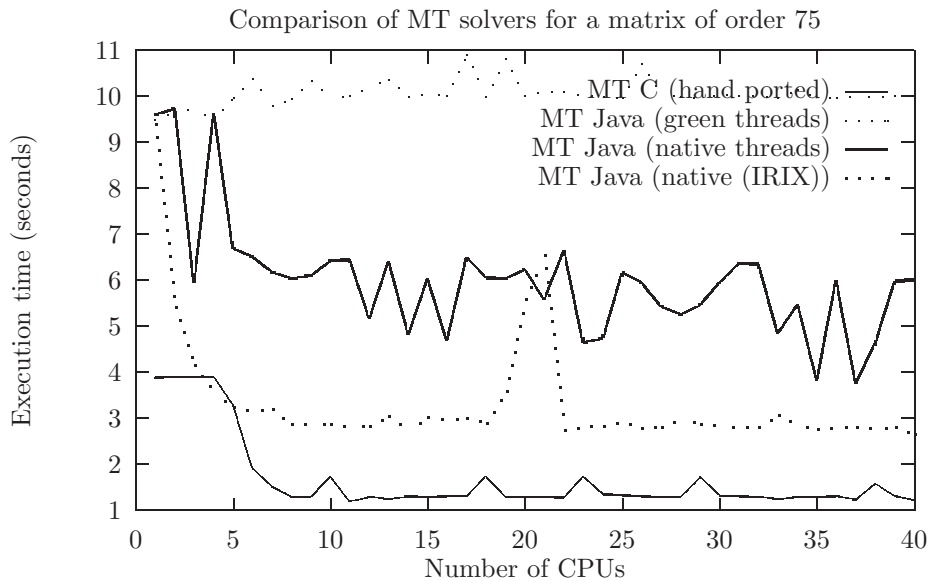


Figure 11: Comparison of the multi-threaded Java solver in green and native mode and also the multi-threaded C solver — matrix order: 75.

## 6 Analysis of the HPC course materials

This was an additional area of work which was to follow on from that of Dr Keith Taylor here at MVC, who ported the course examples for the course “An Introduction to Multithreaded Programming” from the Solaris threading library to use `pthread` .

### 6.1 Overview of the work done

My job was simply to put myself in the position of a student on the course and try all of the example code (the new ported code), to see if there are any problems or inconsistencies with the code. Also, given that I have only been programming using threads for a short time<sup>20</sup>, my naïvity was thought to be useful also; in so far as I might get confused by similar sorts of things as the students on the course. Having done all of the exercises, I then constructed a report on my findings, which is available from my web site on the downloads page, and is also included below in the various sections.

### 6.2 Points to note

I have made several modifications to the examples in order to make them clearer (in terms of commenting) or fix problems with the code. In each case where an example has been modified, the original copy (in the state it was in when I received the files from Keith Taylor) has been kept, but with the additional suffix of `.kt`.

### 6.3 The course exercises

All of the example solutions compiled and ran, and generally most of the problems were related to commenting. There were one or two more subtle problems, and these are explained later on. Each set of

<sup>20</sup>Only whilst I have been at MVC in fact; most of my experience before this of shared memory programming had been with basic IPC found in most C libraries (e.g. `fork()`, `popen()` and so on))

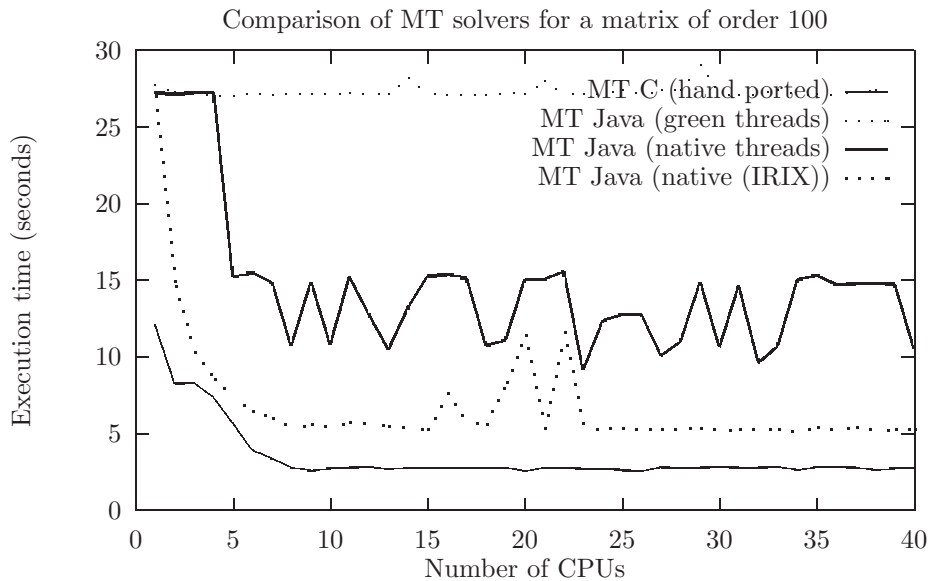


Figure 12: Comparison of the multi-threaded Java solver in green and native mode and also the multi-threaded C solver — matrix order: 100.

problems that I came across are grouped into categories, starting with problems relating to the return types of variables.

### *Return types*

One of the very first things that struck me about the example code is that the main function within each source file and in the examples within the lecture notes does not have an associated return type explicitly declared, so you see things like:

```
main(int argc, char *argv[]) {
    .../* body of function */
    return(0);
}
```

Most C compilers implicitly convert this function signature to have a return type of Integer (the correct form for a main function in C). Although this implicit conversion takes away the problem, it results in a compiler warning, and will annoy the more pedantic programmers.

Another issue with return types is that many functions of return type `void *`, which are used for executing instructions under the control of a thread seem to return 0. Of course this is intended to be a status code from the function, but relies on an implicit typecast from integer to `void *`. This should of course be explicitly typecast since it is intentional to return zero from the function. This isn't the only example of implicit typecasting within the code - the others I shall cover in the typecasting section of this document. As with the previous issue, this occurs in every source example and solution, and throughout the lecture notes/OHPs.

```
void *worker(void *arg) {
    .../* body of function */
}
```

```

    return(0);
}

```

### *Inconsistencies with calling pthread\_exit()*

Although calling the function `pthread_exit()` to terminate the calling thread is largely optional, there doesn't seem to be a consistent approach to it within the source code or examples. Often one finds that it is called in some thread functions within a given program but not all of them. Although the lecture notes state that the use of this function is optional, it may be better to provide a consistent approach to its use, since the current inconsistent approach to its use may imply some additional meaning. The source examples below contain have an inconsistent approach to calling this function (all paths relative to `MultiThreadingCourse/PTHREADS/mt_src`):

- `ex1/simple_mt1.c`:

A call to `pthread_exit()` is present in the function:

```
– void *m_thread(void *arg)
```

However, there is no call to `pthread_exit()` from the function:

```
– void *s_thread(void *arg)
```

- `ex1/simple_mt2.c`:

Exactly the same situation as above, since `simple_mt2.c` is a modification of `simple_mt1.c`.

Those programs that have a consistent approach to calling `pthread_exit()` are:

- `ex1/hello1.c`:

Only one thread-based function in here and the call is made to `pthread_exit()` at the end of the block of instructions to be executed by the thread.

- `ex1/sum_array_mt1.c`:

Again this code contains only one thread-based function and as with the above program, the call to `pthread_exit()` is made at the end of the block of instructions to be executed by the thread.

- `ex1/answer/pi_mt1.c`:

This program contains just one thread-based function (`void *pi_integrate(void *arg)`), which does not call `pthread_exit()` at the end of the block of instructions to be executed by the thread.

- `ex2/hello2.c`:

Like `ex1/hello1.c` this program has just one thread-based function (`void *h_thread(void *arg)` as before). However, unlike `hello1.c`, this program does not call `pthread_exit()`.

- `ex2/sum_array_mt2.c`:

Here there is only one thread-based function (`void *sum_array(void *arg)`). This function does not call `pthread_exit()` at the end of the block of instructions to be executed by the thread.

- `ex2/answer/pi_mt2.c`:

Since this program is based on `ex1/answer/pi_mt1.c`, it has the same thread-based function model, and as in the case of `ex1/answer/pi_mt1.c`, `pthread_exit()` is not called.

- `ex2/answer/matmul_mt.c`:

This program has just one thread-based function (`void *worker(void *arg)`), which does not call `pthread_exit()` (possibly because it is simply a `while`-loop that repeats forever).

- `ex2/network/answer/serv.c`:

As with all the other programs, it has just one thread-based function (`void *server(void *arg)`). This function does not call `pthread_exit()`.

Perhaps it might be better to adopt a single approach of either placing the calls to `pthread_exit()` at the end of every thread-based function, or not to use any calls at all. Or in certain cases where there is no point in calling `pthread_exit()`, such as where the block of code to be executed by the thread is further contained within a `while(1)` block, actually explaining this (of course whether or not this is necessary depends upon the level of experience that the students have).

Something that I feel is worth mentioning is that I have noticed that in each skeleton source file; where a `pthread_exit()` function is used in the accompanying solution source file, a comment is placed in that skeleton file to reflect where the student should place the call. When the call to `pthread_exit()` is missing in the solution file, the comment is also missing in the skeleton file. This could confuse some students if there are comments implying that a call to `pthread_exit()` should be placed in some thread based functions but not others (where such comments are absent, hence the implication). One should bear in mind that it is considered good practice to call `pthread_exit()` at all times when a thread-based function is about to terminate.

*N.B.:* The accompanying skeleton source files are contained in the directory above the answers directory so if you have a program `ex1/answers/pi_mt1.c` then its skeleton will be in `ex1/pi_mt1.c`. The most complete program is always shown, so if there is an exercise the code representing the solution of this exercise will be used rather than the skeleton source file. Path references that do not contain 'answer' therefore imply (and correctly so) that there is no accompanying skeleton source file (this approach is only relevant to this section).

**Source modifications:** All the source files should now contain comments in the case of skeleton files or actual calls to `pthread_exit()` in the case of the solution files. These can be found in the archive stated at the top of this document.

**Final Note:** It should be noted that this is not just a problem with the code that has been ported to POSIX threads, but is also present in the Solaris versions of the code that the course was originally based on. In fact every instance of a missing call to `pthread_exit()` is a result of Sun not placing a call to `thr_exit()` in their original examples.

### *Areas of the skeleton & accompanying files requiring an update*

Many of the skeleton source files still contain Solaris rather than POSIX thread API calls, and also may contain Solaris specific system macros or compile directives. The following list states the skeleton source files that I found were not up to date when trying out the exercises, and also what needs updating (all paths relative to `MultiThreadingCourse/PTHREADS/mt_src`):

- `ex1/README`:
  - Line 24: Users are directed to refer to the program `hello.c` to help them with their solution to `simple_mt1.c`. Of course `hello.c` does not exist - this should be `hello1.c`.
  - Line 43: The question refers to `thr_exit()` rather than `pthread_exit()`.
  - Line 48: The question concerning the use of the default flag (`NULL`) rather than (`THR_NEW_LWP`) is only relevant to the Solaris thread API, since the POSIX thread API has no equivalent.

- `ex1/pi_mt1.c`:
  - Line 61: `#include <thread.h>` specified rather than `#include <pthread.h>`.
  - Line 116: Call to `sysconf` states the Solaris macro `_SC_NPROCESSORS_ONLN` - needs to be changed to `_SC_NPROC_ONLN` for running on Kilburn.
- `ex2/README`:
  - Line 23: The filename given in the hints section is `hello.c` rather than `hello2.c`.
- `ex2/simple_mt2.c`:
  - Line 41: `#include <thread.h>` used instead of `#include <pthread.h>`.
- `ex2/matmul_mt.c`:
  - Line 64: `#include <thread.h>` used instead of `#include <pthread.h>`.
  - Line 77: `mutex_t` is used in the declaration instead of `pthread_mutex_t`.
  - Line 78: `cont_t` is used in the declaration instead of `pthread_cond_t`.
  - Line 83: `mutex_t` is used in the declaration instead of `pthread_mutex_t`.
  - Line 238: Call to `sysconf` states the Solaris macro `_SC_NPROCESSORS_ONLN` - needs to be changed to `_SC_NPROC_ONLN` for running on Kilburn
- `ex2/network/Makefile` & `ex2/network/answer/Makefile`:
  - These are Solaris makefiles with Solaris specific compiler flags. They will need updating for use on Kilburn. One of the differences between Solaris and IRIX is that you don't have to explicitly specify the sockets library on the command line. If the code is to be compiled under Solaris then these makefiles will work, although the makefile in the main network directory has not been changed and still has `-lthread` rather than `-lpthread` as one of the library flags. Not that's it's critical in the main directory anyway, since this code is single threaded (only the answer directory has the multi-threaded code).
- `ex2/network/serv.c`:
  - Line 95: This is a commented out `#include` directive that when uncommented will allow the server to be modified to be multi-threaded. In the single-threaded source file (which is used as a template from which to build the multi-threaded server), this directive points to `thread.h` rather than `pthread.h`.
  - Line 125: Here we find `mutex_t` within a comment, which should of course be `pthread_mutex_t`. It would probably be enough to jog a student's memory as it is, but may cause confusion....
  - Line 278: Again, another misleading comment. This comment pertains to setting up threads for each accepted connection. Within the comment is the appropriate Solaris (rather than POSIX) thread creation function to do this `thr_create(...)`. This of course needs changing with the appropriate information about attribute objects being added.

**Source modifications:** All of the modifications stated in this section have been made to the source code within the archive at the top of this document. These modifications extend to the `README` files, such as the deletion of the question regarding LWPs which is not relevant to `pthreads`.

***Type problem***

The problems with implicit typecasting are inherited from the original code from Sun Microsystems. The main problem centres around the passing of thread IDs to other functions so that operations such as `join()` can be performed. Of course in order to pass the thread IDs (of type `(pthread_t)` for `pthreads`), they must be cast to type `(void *)` and then recast back to `(pthread_t)`. This must be dereferenced first so that cast appears as `*(pthread_t *)` which effectively casts the `(void *)` to a `(pthread_t *)` and then dereferences it. This is not done in the original Solaris code (and generates warnings when duplicated using `pthreads`), as one can see by studying the examples. Here is an example from the program `simple_mt1.c`, with the first code fragment coming from the original Solaris code, and the second code fragment coming from the final ported code:

```
58  thread_t tid = (thread_t) arg;
```

Which must become:

```
61  pthread_t tid = *(pthread_t *) arg;
```

This is a simple conversion and most of the examples if not already done could be changed easily in the manner demonstrated. There was one case however, which doesn't work - the pi integration programs. These programs resist attempts to appropriately cast; which simply results in segmentation violations. This is probably due to the large amount of implicit typecasting goes on. The following code fragment is the integration function from the original Solaris code (`pi_mt1.c`):

```
76  void *pi_integrate(void *arg)
77  {
78      thread_t tid = (thread_t) arg;
79      double  x, x_start, x_end, height_sum = 0.0;
80
81      /* calculate x_start and x_end for each sub-region */
82      x_start = (tid * sub_no_of_strips + 0.5) * strip_width;
83      x_end   = (tid * sub_no_of_strips + sub_no_of_strips) * strip_width;
84
85      /* rounding on the last sub-region */
86      if (x_end > 1.0) x_end = 1.0;
87
88      /*
89       * For each strip on a sub-region,
90       * find the value of x at the midpoint,
91       * calculate the height f(x) of the rectangle strip,
92       * and sum the result
93       */
94      height_sum += 4.0 / (1.0 + x * x);
95
96
97      /*
98       * Don't put sub_sum[tid] inside the loop
99       * Otherwise the performance will go down sharply
100      * due to the massive computation for the array element indexing
101      */
102      sub_sum[tid] = height_sum;
103
104      return(0);
105 }
```

As one can see there are a number of areas where `thread_t` (the Solaris thread type) is assumed to cast correctly to an integer (lines 82, 83 and 102). Such an assumption would not be a problem for Sun Microsystems because they have the knowledge as to how the types are implemented and cast within their compilers. When porting the code to `pthread_t`, it is much less safe to assume that `pthread_t` will implicitly cast to an integer, there are known cases where this is not true (such as cygnus `pthread_t` under Win95), and this is a potential source of problems. Although this code will work, it is unsafe and although known to work on IRIX it cannot be guaranteed to work elsewhere. To fix the problem, the code needs re-implementing in places to remove the use of implicit casts (particularly for things such as array indexing (102)!). As I have said this is an issue that has been left (apart from the obvious conversion from `thread_t` to `pthread_t` - which had already been done), and can be followed up if necessary (the code will work, but will cause compiler warnings when compiled).

### *Program output inconsistencies*

There are one or two cases where the output from the solution file differs from that in a completed skeleton file, such as a different `printf` format string being used in the skeleton file to the one that is being used in the solution file. Inconsistencies in output between skeleton and solution source files are listed below (all paths listed relative to `MultiThreadingCourse/PTHREADS/mt_src`):

- `ex1/pi_mt1.c` & `ex1/answer/pi_mt1.c`:

There are two output inconsistencies with this code. Perhaps better demonstrated with an example. Here is output from the completed skeleton code provided when run with 10000 as an argument:

```
bash\$ pi_mt1.x 10000
pi = 3.14159265442313010000, number of strips = 10000, number of CPUs = 40
```

Now here is the output from the solution provided in the answers directory (run with the same argument):

```
bash\$ pi_mt1.x 10000
sub_no_of_strips = 250
pi = 3.1415926544, number of strips = 10000, number of CPUs = 40
```

The first inconsistency as you can see from the program's output is that the amount of information output differs. The solution provided outputs the number of strips that have been integrated, whereas the completed skeleton does not. The reason being that the appropriate `printf` statement is not provided and there is no comment hinting to students that they ought to put this in. To fix this either the appropriate instruction or a comment telling the student to do this, needs to be inserted between lines 138 & 140.

The second difference is that different `printf` format strings are used for the output of the pi integration, In the skeleton source file, the format string `%2.20f` is used to print the output to 20 decimal places, whereas in the solution source file, the format string `%2.10f` is used to print the output to 10 decimal places. To fix this the `printf` statements needs adjusting in one of the programs. The appropriate `printf` statements occur on line 164 in the skeleton source file, and line 176.

*N.B.:* It should be noted that this problem will also affect the program `ex2/pi_mt2` since it is based on `ex2/pi_mt1`; basically students have to copy the first program and modify it so that access is synchronised. Of course if they have not added the appropriate `printf` statement themselves, the students will have the same problem as with the unsynchronised solution program since the solution code for this exercise also outputs the number of strips.

**Source modifications:** To fix the first inconsistency, a `printf` statement has been introduced at line 140 of the skeleton source file to print out `sub_no_of_strips`. The second inconsistency has been fixed by replacing the existing `%2.10f` `printf` format specifier for printing out `pi` with the format specifier `%2.20f`. This second change has been made to the solution source file to ensure its output matches that of the skeleton in terms of format.

### *Attribute objects*

As one will now be aware, one of the major changes when moving from the Solaris to POSIX thread APIs is that threads need to have an associated attribute object if they need anything more than the default set of attributes (in these examples, attribute objects are needed in `ex2` for the creation of detached threads). It is possible for more than one thread to use the same attribute object if the properties required are the same, and of course it is within the same scope as the thread. One of the common problems with the skeleton source files was the lack of any kind of hints to the student that attribute objects need to be set up. Although students are likely to be taught the need for these it is still something that could be forgotten, and inclusion of such comments would keep the approach more consistent with the rest of the skeleton files. The following exercises use attribute objects, and require additional comments (the location of where these comments should be inserted is also specified):

- `ex2/simple_mt2.c`: Line 102 - Modify the comment informing the student to create a detached thread to include a reminder about the attribute object.
- `ex2/matmul_mt.c`: Line 182 - Again a hint about attribute objects would be helpful.

**Source modifications:** For the program `ex2/simple_mt2.c`, I have inserted a comment at line 103 reminding students to create an attribute object. In the case of the program `ex2/matmul_mt.c`, an existing comment beginning at line 180 has been modified to remind the student that if they are using detached threads they'll need an attribute object.

### *Condition variables*

These are an important part of synchronisation, and as with attribute objects it may be helpful to include comments indicating where they need to be used in exercises. The following cases demonstrate where I think action should be taken to prevent confusion (all path names are relative to `MultiThreadingCourse/PTHREADS/mt_src`):

- `ex2/simple_mt2.c`: Within the function `s_thread()` a comment is needed to indicate that a condition variable should be used - this is indicated within all of the other functions, and hence is a potentially confusing inconsistency.

Secondly, within the function `m_thread()` although a comment is used to indicate the need for a condition variable; there is no skeleton `while` loop to indicate that consistent checking of the condition variable is required. This ensures that the function waits until the string "Simple" has been output before printing the string "Multithreading". It is therefore recommended that such a skeleton construct is inserted.

**Source modifications:** Both of the above steps have been taken; i.e., a comment inserted in `s_thread()` to indicate the need for a condition variable, and a skeleton `while` construct has also been inserted.

### *Comments*

One common problem with the comments on the skeleton and solution source files is that they contain the details for compiling the code under Solaris. At the bottom of each comment is a pointer to a shell

script that will compile and run the code. Are these scripts to be provided as part of the course material? If so, the compile and run questions like exercise 1 on page 1 of the examples and exercises booklet then become redundant; if not, these comments will need modifying. The old comments are present in every source file. Also it is not just compiling details that are for Solaris, some of these header comments such as that in `ex1/pi_mt1.c` also contain system macro details which are specific to the Solaris platform; such as the `sysconf` macro `_SC_NPROCESSORS_ONLN` which is for determining the number of processors available. If the code is to be compiled on an SG machine such as Kilburn then this needs to be changed to `_SC_NPROC_ONLN`.

**Source modifications:** As far as the header comments are concerned, I have replaced the Solaris specific compiler commands with the appropriate IRIX commands for the SGs. Also the references to the shell scripts have been removed, since I would imagine that the students will be expected to compile and run their own code as they have done in the past. Other comments such as `/* KT START */` indicating sections of code that have been modified have been removed (they remain in the `.kt` files). The intention being that the files should require virtually no modification before being made available to the students.

### *Possible mistakes*

The only error discovered in the code occurs in `MultiThreadingCourse/PTHREADS/mt_src/ex2/answer/simple_mt1.c`. It is not a major error since it won't cause the program to fail during compilation or execution.

Line 120. A detached thread is being created here. The appropriate attribute object is set up at line 113 and the program then enters a loop to create the group of detached threads. The mistake lies in the creation of the threads, which is done as follows (line 114-117):

```
.
.
113 pthread_attr_setdetachstate(&sattr, PTHREAD_CREATE_DETACHED);
114 if (ir = pthread_create( &stid, NULL, s_thread, NULL)) {
115     fprintf(stderr, "pthread_create: %s\n", strerror(ir));
116     exit(1);
117 }
.
.
```

The problem with this of course is that the attribute object is created and then isn't actually associated to the threads that are being created. As a result the threads are being created with default attributes instead. In order to associate an attribute object with a thread, the attribute object must be passed to `pthread_create()` as its second argument. So the code above, should read:

```
.
.
113 pthread_attr_setdetachstate(&sattr, PTHREAD_CREATE_DETACHED);
114 if (ir = pthread_create( &stid, &sattr, s_thread, NULL)) {
115     fprintf(stderr, "pthread_create: %s\n", strerror(ir));
116     exit(1);
117 }
.
.
```

## 6.4 Updating of source files

All of the source files so far have been updated to fix the mistakes and anomalies mentioned within this report. The tape archive of all of the corrected source examples still needs a final review, since I believe

that there may still be some lingering development comments from either Keith or indeed myself that need removing. Most of the files are fine as far as I am aware, and should be correct (even those with these additional comments!).

#### 6.4.1 An interesting case — The matrix multiplication example

All of the programs tested behaved as expected (i.e., if a given program was parallelised, a speedup was evident), bar one program - `ex2/matmul_mt.c` the multi-threaded matrix multiplication example, which is part of the second set of exercises. In this exercise, the students are asked to modify the supplied single threaded version to make it multi-threaded. Once they have done this, the students are then asked to time the execution of the single and multi-threaded versions of the program and note the speed improvement. Unfortunately the multi-threaded code (including the provided solution) runs slower than the single threaded code on `kilburn`, whether run interactively, or interactively via 1 CPU (`runon`) or via NQE. Here are some timings to illustrate this point (using the supplied solution):

*Single threaded:*

Real	User	System
3.924	3.892	0.015
3.950	3.897	0.015
3.951	3.923	0.019

*Multi-threaded (2 CPUs):*

Real	User	System
4.172	4.116	0.024
5.292	5.231	0.021
4.150	4.105	0.025

*Multi-threaded (4 CPUs):*

Real	User	System
4.638	4.576	0.023
3.651	4.227	0.030
4.476	5.335	0.402

*Multi-threaded (8 CPUs):*

Real	User	System
5.363	5.27	0.023
7.750	9.293	1.905
8.234	8.656	2.061

It is perhaps worth noting that the timings for the program improve to the extent where the multi-threaded code runs around 0.2 seconds slower than the single-threaded code - the optimal figure seems to be around 2 or 3 CPUs. Beyond that thread contention gets in the way (40 is significantly slower at 12.647 seconds real CPU time). The performance anomaly with `pthreads` is not limited to `kilburn`'s implementation under IRIX 6.5, since the `pthreads` code runs slower than that ST code under other systems, here is a set of timings from a Linux box (a 400MHz Intel Pentium II running SuSE Linux 5.3 with a 2.0.35 kernel):

*Single-threaded:*

Real	User	System
1.757	1.750	0.000
1.756	1.750	0.000
1.759	1.760	0.000

*Multi-threaded:*

Real	User	System
1.837	0.000	0.000
1.835	0.000	0.000
1.838	0.000	0.000

The most interesting set of timings one can do is on a Solaris box such as `irwell` - a 22 CPU Sun E6500 system running Solaris 2.6. Here one can compare the performance of the original solution written using Solaris threads, the port of this to `pthread`s and also the single threaded code. One finds that the relationship between the single threaded solution and multi-threaded solution using Solaris threads is as one would expect, i.e., there is an appreciable speed up; whilst the relationship between the single threaded solution and the `pthread` based port of the multi-threaded solution is as we have been seeing (i.e., the MT code being slower than the ST code). Here are some timings (three runs for each shown):

*Single-threaded:*

Real	User	System
5.247	5.200	0.020
5.142	5.110	0.020
5.172	5.150	0.010

*Multi-threaded (2 CPUs):*

Solaris threads:

Real	User	System
2.883	5.670	0.040
2.824	5.570	0.030
2.827	5.600	0.020

POSIX threads:

Real	User	System
5.780	5.750	0.020
5.415	5.380	0.030
5.431	5.390	0.030

*Multi-threaded (4 CPUs):*

Solaris threads:

Real	User	System
1.530	5.970	0.020
1.479	5.810	0.010
1.443	5.640	0.030

POSIX threads:

Real	User	System
5.359	5.360	0.010
5.493	5.450	0.020
5.381	5.350	0.020

*Multi-threaded (8 CPUs):*

Solaris threads:

Real	User	System
0.843	6.070	0.060
0.805	5.840	0.100
0.897	6.510	0.150

POSIX threads:

Real	User	System
5.392	5.360	0.020
5.405	5.360	0.040
5.381	5.340	0.030

The current evidence seems to suggest that there is a problem with `pthread`, especially when one considers that the Solaris code compiled to use the `pthread` library via the `SPLT` library also underperforms in a similar fashion (identical timings). Had the code badly performed under Solaris then the problem could be written off as a design flaw in the algorithm, but since this is not the case, the evidence seems to be pointing to a problem with `pthread`. The core of the problem seems to be the thread creation function used by the Solaris code:

```
thr_create(NULL, NULL, worker, NULL, THR_NEW_LWP|THR_DETACHED, NULL);
```

This is an example of a user customised thread creation within the Solaris threading API. On it's own `THR_NEW_LWP` signifies that a new detached thread should be created and the concurrency level raised by one, i.e. effectively execute this new thread on a different CPU (a new LWP is created for the thread). The other attribute `THR_DETACHED` simply states that the thread is detached (i.e. not joinable). What effectively happens here is that a new detached thread is created and the concurrency level is raised by one so that the new thread runs on a new CPU if available (repeated several times, this causes threads to be distributed across CPUs on a multiprocessor machine).

As one may already be aware, the Solaris thread creation flag `THR_NEW_LWP` has no equivalent in `pthread`. This area of ensuring a concurrency level is the real achilles heel of the `pthread` standard. The original standard specified in June 1995 did not take this into account, and as a result, the aforementioned flag and related functions `thr_setconcurrency()` and `thr_getconcurrency()` have no equivalent mapping in the original standard. Whilst this has recently been rectified following the release of the single UNIX specification (UNIX 98), which introduces the functions `pthread_getconcurrency()` and `pthread_setconcurrency()` which are equivalent to their Solaris counterparts `thr_getconcurrency()` and `thr_setconcurrency()`, it is of course not something that can be fully relied upon due to its absence from the original standard. The UNIX 98 standard is not supported on `kilburn` (IRIX 6.5) or `irwell`; but `sgi` have clearly decided to provide the concurrency support defined within the standard (although how well the standard is complied with in this area remains unclear). To compile code for this standard, the feature test macro `-D_XOPEN_SOURCE=500` should replace the existing `-D_POSIX_C_SOURCE=199506L` (subsumed by UNIX 98).

Having established the functionality of the Solaris thread creation function used, the problem with the code in this case appears to be underperformance due to a sub-optimal concurrency level (the default), largely due the `pthread` port not raising the concurrency level (perfectly legitimate given that it is not part of POSIX 1003.1c — the original `pthread` standard) when threads are created like the Solaris version does using the flag `THR_NEW_LWP` (in the `pthread` port, the threads are simply created as detached threads).

This seems to be confirmed when one introduces the Solaris function `thr_setconcurrency()` into the example code (this function can be used to set the concurrency level for `pthread`

as well as Solaris threads on Solaris 2.6 systems instead of `pthread_setconcurrency()`. As a quick test, the concurrency level was set<sup>21</sup> to the proposed number of CPUs on which the processing was to be distributed across plus one - roughly what the original Solaris thread creation statement does. So we get:

```

178  /*
179  * Check if its worker threads have been created.
180  * If not, create one for each CPU.
181  */
182  if (work.workers == 0) {
183      work.workers = nCPU;
184      thr_setconcurrency(nCPU+1);
185      for (i = 0; i < work.workers; i++)
186          pthread_create(&tid, &tattr, worker, NULL);
187  }
```

This had a marked impact upon the timings, which were now respectable and showed evidence of scaling. Here are the timings on 2, 4 and 8 CPUs as before (taken on `irwell` of course):

*Multi-threaded (2 CPUs):*

POSIX threads (with concurrency improvement):

Real	User	System
2.783	5.470	0.010
2.772	5.570	0.030
2.821	5.470	0.030

*Multi-threaded (4 CPUs):*

POSIX threads (with concurrency improvement):

Real	User	System
1.874	5.480	0.060
1.419	5.500	0.030
1.457	5.520	0.040

*Multi-threaded (8 CPUs):*

POSIX threads (with concurrency improvement):

Real	User	System
0.801	5.620	0.110
0.770	5.660	0.100
0.793	5.620	0.070

As one can clearly see, setting the concurrency level has had a considerable impact on the timings. In this case, the timings are undistinguishable from those of the original Solaris threads code. It is quite clear that increasing the concurrency level is the additional missing step, ... or is it? Whilst everything has behaved as expected for Solaris (we have increased the number of available active threads and hence the performance), this had only limited success under IRIX 6.5 on `kilburn`. If one replaces `thr_setconcurrency()` for the function `pthread_setconcurrency()` that `kilburn` and other UNIX 98 compliant systems support, the code becomes:

<sup>21</sup>This should really have been increased rather than set in order to mimick the normal creation of the threads specified above, i.e. `thr_setconcurrency(thr_getconcurrency()+nCPU+1`

```

178  /*
179  * Check if its worker threads have been created.
180  * If not, create one for each CPU.
181  */
182  if (work.workers == 0) {
183      work.workers = nCPU;
184      pthread_setconcurrency(nCPU+1);
185      for (i = 0; i < work.workers; i++)
186          pthread_create(&tid, &tattr, worker, NULL);
187  }

```

One would expect this to work similarly well since the Solaris and `pthread` versions of these functions are supposed to be functionally identical. However, this change in the concurrency level only works to a point, as one can see from the following timings taken for 2, 4 and 8 CPUs on `kilburn`:

*Multi-threaded (2 CPUs):*

```

POSIX threads (with concurrency improvement):
  Real    User    System
  2.370   4.573   0.048
  2.320   4.530   0.035
  2.353   4.550   0.044

```

*Multi-threaded (4 CPUs):*

```

POSIX threads (with concurrency improvement):
  Real    User    System
  11.691  12.370  6.969
  11.167  12.000  6.143
  10.821  11.781  6.468

```

*Multi-threaded (8 CPUs):*

```

POSIX threads (with concurrency improvement):
  Real    User    System
  15.400  15.031  8.310
  16.976  17.858  8.277
  16.276  16.160  7.654

```

As one can see, whilst there is an improvement is evident for the 2 CPU case, the 4 and 8 CPU cases deliver significantly worse timings. This could be an indication that the concurrency level needs some further tweaking, or that this has to be used in conjunction with the IRIX non-portable interface `pthread_setrunon_np()` which allocates the calling thread to the CPU it requests (see the manual page for scheduling details). Use of this additional API call along with an appropriate increase in the concurrency level would hopefully allow the program to perform correctly (ensuring threads had different CPUs would be more in the spirit of Solaris thread creation where threads created with the flag `THR_NEW_LWP` are generally allocated to a new CPU on multi-processor systems).

The tests performed were very crude and purely done in order to ascertain the nature of the problem (which has now been done), and whilst the initial test provided idea performance

under Solaris, some investigation still needs to be done to improve performance under IRIX - `pthread_setrunon_np()` may be the way forward bearing in mind the tests already done. The matrix multiplication source files have not been modified to set or increase the concurrency level due to the absence of this functionality from the original standard, and also due to the performance issues on `kilburn`. However, there is now hopefully enough information to allow this problem to be rectified.

#### 6.4.2 Comments

Apart from the problem with the matrix multiplication exercise, the rest of the code behaved as expected, and demonstrates the ease with which code is generally ported from Solaris to POSIX threads. The one or two inconsistencies that were to be found within the code, were generally inherited from the original Solaris examples from Sun Microsystems. Hopefully the dangers of directly porting code have been highlighted by the matrix multiplication and pi integration examples.

As well as one or two problems with the examples and porting issues, there were some inconsistencies that have also been cleared up. Considerable steps have been taken to ensure that the skeleton source is as clear as possible for students without actually solving the problem. The intention is that all of the `.c` files should now be suitable for distribution to students, apart from one or two final modifications (perhaps taking into account the comments regarding the two problematic sets of examples). The complete directory structure including the corrected files, original HPC Solaris examples, and files prior to further modification as part of this project, is available as a gzipped tape archive from the project web site:

<http://www.man.ac.uk/MVC/students/summer99/truhlar/ce.tar.gz>

The file is made available from the on-line documentation of the work carried out on the course examples, which is available from the downloads page.

## 7 Conclusion

Hopefully this report has shown that porting threaded numerical applications to Java is possible. However, it should be quite clear that direct ports are not necessarily the answer, particularly when it comes to applications like the multi-threaded solver, where as we have seen it may have been better to re-implement the solution or at least sections of it rather than aiming for a direct translation which can be problematic. Where the C is concerned, it has been shown just how easy it is to get code working on different platforms, arguably easier than porting the code to Java, since the C threading paradigm remains the same across a large number of threading libraries, hence making the conversion process as simple as using an interface layer like SPILT or converting the code by hand by swapping API calls. Whilst a Java port has been possible, and the flexibility is there to run the code on platforms that don't even fully provide user threading systems, this has come at a price both in terms of speed and also reliability (though reliability is only an issue with the current incarnation of the code when it is run natively). However, it must be said that the multi-threaded performance of the code when run in native mode was markedly better than expected, although still not quite up to the performance levels of C.

Given that one of the main reasons for porting the code to platform independant threading systems such as `pthread`s and also to Java was to allow the code to be used on much of the Silicon Graphics hardware at MVC (this is particularly true for `pthread`s), it is worth highlighting certain concerns over the `pthread`s implementation. It appears that in certain cases such as the matrix multiplication example covered as part of the HPC course review, and also Java programs when using native threads (virtual machine threads are mapped onto `pthread`s), there are problems distributing threads across CPUs. In these cases, the performance is clearly sub-optimal, and requires the raising of the concurrency level to affect an improvement. Experiences with the MT Java code and also the matrix multiplication example can set one wondering as to whether performance elsewhere could be improved<sup>22</sup>. Of course performance was not a major part of this project, just a yardstick for determining the effectiveness of ports, but experience within this project shows that it may be something worth looking into in more depth.

### 7.1 Things still to do...

This is a project that has so many different possible turning points that it is impossible to cover every angle in just twelve or thirteen weeks, and as such there are several areas that could do with greater coverage or looking into at a later date.

The author's main area of concern is currently the multi-threaded Java port, which may well benefit from a new approach, such as using the original idea that every piece of functionality is encapsulated in a different thread type, or using interfaces. Both of these would provide a good alternative to method references and are worth considering. Once the reliability has been sorted out, various work on optimisation could be done, and more work could be done with both C and Java to ascertain the impact of concurrency levels on execution time, since as shown there are instances where it has been an issue.

---

<sup>22</sup>A current concern is that although the performance of the `pthread`s port of the solver is adequate and shows scaling, perhaps it could be better.

Also worth looking at would be alternatives to the `DecimalFormat` objects that we have seen so much of. It was quite clear from the testing that these were at times less than reliable for displaying a number that has is represented correctly internally within the Java virtual machine (i.e. there were rounding errors). Perhaps a class like the `Format` class shown in the book *CoreJava* would do a better job (it is meant to re-implement much of `printf()`'s functionality).

Finally, it would perhaps be useful to obtain a wider set of timing data from platforms such as Win32, Linux, HPUX, and so on. This was something that was planned, but time constraints prevented. In particular it would be useful to see how compiling Java fully into native code using a compiler like `guavac` affects the performance, and how this compares with existing languages like C and Fortran. All of the Java code within this project has been executed within a standard Java virtual machine, which means that the compiled virtual machine code instructions must then be interpreted or at best compiled to native code on the fly, which is still slower than a direct one-time compilation to native code<sup>23</sup>.

---

<sup>23</sup>Of course compiling to native code would mean that the code had to be compiled before use on different platforms, rather than just having one virtual machine code, but if performance were enhanced, this would be a small price to pay.

## References

- [1] Jens Latza.  
*Threads and Java.*  
Summer Project.  
Manchester Visualization Centre, University of Manchester, 1997.
- [2] Stephan Enderlein.  
*Port from Solaris 2.5 threads to Windows NT threads.*  
Manchester Visualization Centre, University of Manchester, 1997.
- [3] Solaris Migration Support Centre  
*pthreads and Solaris threads: A comparison of two user level threads APIs.*  
(Early Access Edition (May 1994): *pthreads Based on POSIX 1003.4a/D8*)  
©1993, Sun Microsystems Inc, Mountain View, CA, USA.
- [4] Daniel J. Berg  
*Java Threads - A Whitepaper*  
©1995, Sun Microsystems Inc, Mountain View, CA, USA.
- [5] Richard Marejka.  
*Threads Case Study #2.*  
Solaris 2, Migration Support Centre,  
Sun Microsystems Inc, Mountain View, CA, USA, 1994.
- [6] David Flanagan.  
*Java In A Nutshell (2nd edition).*  
ISBN: 1-56592-262-X.  
O'Reilly & Associates Inc., Sebastopol, CA, USA, 1997.
- [7] Bradford Nichols, Dick Buttlar & Jacqueline Proulx Farrell.  
*Pthreads programming.*  
ISBN: 1-56592-115-1.  
O'Reilly & Associates Inc., Sebastopol, CA, USA, 1998.
- [8] Scott Oaks & Henry Wong,  
*Java Threads.*  
ISBN: 1-56592-418-5.  
O'Reilly & Associates Inc., Sebastopol, CA, USA, 1999.
- [9] Cay S. Horstmann & Gary Cornell.  
*Core Java 1.1 Volume 1 - Fundamentals.*  
ISBN: 0-13-766957-7.  
Sun Microsystems Press, ??, USA, 1997.
- [10] Peter van der Linden.  
*Just Java (2nd edition).*  
ISBN: 0-13-766957-7.  
Sun Microsystems Press, ??, USA, 1997.

- [11] Bruce Eckel.  
*Thinking in Java*.  
ISBN: 0-13-766957-7.  
Prentice-Hall, Inc, ??, USA, 1998.
- [12] Nancy Winnick Cluts  
*Geek Speak Decoded #7: Tasks, Processes, and Threads*  
URL: <http://msdn.microsoft.com/workshop/essentials/geekspeak/geekthread.asp>  
MSDN, Microsoft Corporation, 1999.