The SCC and the SICSA Multi-core Challenge

Paul Cockshott, Alex Koliousis

The SICSA Multi-core Challenge is an open competition called by the Scottish Informatics and Computer Science Alliance to develop multi-core implementations of a set of predefined problems. So far two rounds of the challenge have been run which have attracted entries from teams spread accross Europe. Each round of the challenge has consisted in the initial publication of a problem with a candidate serial implementation of the problem. Participants had to select a programming language, a host architecture and a parallisation system with the aim of achieving either the fastest implementation, or the best acceleration relative to the serial version on that architecture.

The stated aim was "to learn about the strengths and weaknesses of current systems for parallel programming by comparing them on a common application and by discussing the results at a workshop summarising the results."

The Challenge has had two rounds so far, the first being reported at a workshop at the Heriot-Watt Edinburgh on 13th December 2010, the second round reported its results at a workshop at the University of Glasgow on the 27th May 2011. We have implemented both of the SICSA challenge problems so far on the SCC. In the case of the first challenge we were able to do this in time for the workshop, in the second case we did not have a working implementation until after the workshop. We will, in this paper, describe the problems, describe the SCC implementations and then contrast these with other published implementations both in terms of design and in terms of performance.

I. Phase I

The first phase of the challenge was a concordance problem.

A. Specification of the Concordance application.

a) Given: Text file containing English text in ASCII encoding. An integer N.

b) Find: For all sequences of words, up to length N, occurring in the input file, the number of occurrences of this sequence in the text, together with a list of start indices. Optionally, sequences with only 1 occurrence should be omitted.

In addition to the specification a reference implementation in Haskell was provided and several reference texts. In practice most work was done with the longest of these texts, the World English Bible available from Project Gutenberg which is 4.68 Megabytes in length.

B. Improved Serial Implementation

Prior to doing any parallelisation it is advisable to initially set up a good sequential version. We were doubtfull that the first challenge would provide an effective basis for parallelisation because it seemed such a simple problem. Intutively it seemed like a problem that is likely to be of either linear or at worst log linear complexity, and for such problems, especially ones involving text files, the time taken to read in the file and print out the results can easily come to dominate the total time taken. If a problem is disk bound, then there is little advantage in expending effort to run it on multiple cores. However this was only an hypothesis and needed to be verified by experiment. In order to optain an efficient an non-esoteric sequential implementation, C was chosen as the implementation language. The algorithm performed the following steps:

1) Read the whole text file into a buffer.
2) Produce a tokenised version of the buffer.
3) Build a hash table of phrases of up to N tokens and prefix tree.
4) If the concordance is to be printed out, perform a traversal of the trees printing out the word sequences in the format suggested by the Haskell version.
5) If we want it sorted, pipe through Linux sort

The source code is available here (http://www.dcs.gla.ac.uk/~wpc/reports/SICSA/concordance.c). Table I shows the performance of the C implementation was very much faster than that obtained using the Haskell reference code. It also seemed to indicate that there was little practical benefit from parallelising the application since the greatest part of its time was spent formatting and printing the output.

TABLE I
SERIAL BENCHMARKS FOR PHASE I ON TWIN CORE 2.6GHZ INTEL PLATFORM.

OS/Language	Printing on	file size	text file	seconds
HASKELL				
Windows	yes	4792092	WEB.txt	>2hours
Windows	yes	3580	TMM1.txt	0.824
C Version				
Windows	no	3580	TMM1.txt	0.028
Windows	yes	3580	TMM1.txt	0.029
Windows	yes	4792091	WEB.txt	3.673
Windows	no	4792091	WEB.txt	0.961
Linux	yes	4792091	WEB.txt	2.68
Linux/O3	yes	4792091	WEB.txt	2.25
Linux	no	4792091	WEB.txt	1.04
Linux/O3	no	4792091	WEB.txt	0.899

C. Parallel Implementation

The concordance problem is hard to parallelise efficiently. You can not just split a book into two halves, prepare a concordance for each half and then merge, since we only have to print out repeated words and phrases. A repeated word might be missed in this case if it was mentioned once in the first half and once in the second half. A more complicated

TABLE II
PARALLEL VERSION ON 2.6GHZ DUAL CORE.

Concordance benchmark		
Program	OS	seconds
PARALLEL VERSION		
concordance2.c	Windows	5.63
concordance2.c	Linux	2.257
SHELL SCRIPT		Avg 4 runs
conc.sh	Linux	2.12

approach was needed. We chose to parallelise by getting several threads to read the entire book, since reading turns out to be relatively fast. The words themselves are then divided into disjoint sets - one obvious split would be into 26 sets on the first letter. We then set each thread to do the concordance for a disjoint subset of the words. A large part of the time is also taken up with output – the printed concordance can be 5 times as large as the input file. If distinct cores are producing this, there is an inevitable serial phase in which the outputs of the different cores are merged into a single serial file.

As a first parallel experiment a dual core version of the C programme was produced using the pthreads library and it was tested on the same dual processor machine as the original serial version of the algorithm. Conclusions are derived from the results shown in Table II.

There was no gain using multithreading on windows. It looks as if the pthreads library under windows simply multi threads operations on a single core rather than using both cores. On Linux there was a small gain in performance due to multithreading - about 17% faster in elapsed time using 2 cores. Since a large part of the program execution is spent writing the results out, this proves a challenge to improve using multicore. The first parallel version adopted the strategy of allowing each thread to write its part of the results to a different file these were later merged and sorted.

A second parallel verions follows the same basic strategy as the previous one, but uses the shell, rather than pthreads to fork parallel processes off and communicates via files using the following command:

```
./l1concordance WEB.txt 4 P 1 0 >WEB0.con&
./l1concordance WEB.txt 4 P 1 1 >WEB1.con
wait
cat WEB1.con >>WEB0.con
```

This has the best performance of the lot as seen in the attached tables above.

D. SCC Experiment

We have also tried the programme out on the Intel SCC with relatively poor results.

The SCC is configured as a host processor, which is a conventional modern Intel x86 chip. Attached to this is the experimental 48 core SCC chip, each of whose cores runs a discrete copy of Linux. A major worry here was the problem of file i/o for the multiple cores. The source file and the output files were placed on a shared NFS system, and accessed from there. Looking at the time for one core doing the full task one can see that it is much slower than a single core on the host doing the same task. It is unclear how much of this slowdown is due to the slow access to files from the daughter copies of Linux and how much is due to the poorer performance of the individual cores on the SCC.

TABLE III
SCC PERFORMANCE ON CONCORDANCE.

Implementation	**seconds**
1 core doing full concordance	26.17
1 core doing half concordance	13.48
1 core doing 1/8 concordance	5.59
2 cores doing half each	49
8 cores doing 1/8 each	36
32 cores doing 1/32 each	34
1 host processor doing it all serially	1.03
2 cores on host processor and the shell	0.685

The top 3 lines of the table show the effects of trying to do smaller portions of the workload in an individual core.

We dispatched 32 tasks using the pssh command as shown in Algorithm 1.

The first line removes any temporary output from a previous run. We then use pssh to run the script sccConcordance32 in a shared directory, sending the output to the /shared/stdout directory/

When all tasks have run the output from all the tasks is concattenated and sorted to yield the final file.

The script sccConcordance32 invokes the actual concordance task

```
cd /shared
./l1concord WEB.txt 4 P 31 $(hostname)
```

The hostname is used to derive a process id which is then used to select which words will be handled by the task. the 4th parameter to l1concord is the mask that is applied to give the number of significant bits in the process id, 5 in this case.

It is clear that on a file i/o bound task like this, the SCC is a poor platform.

E. Results from other teams

At the workshop a number of other implementations were presented. Singer [?]reported on the use of Java Fork Join primitives to implement a parallel version of the concordance. Stewart [?] reported the use of Hadoop Map/Reduce to solve the problem. AlJabri reported on the use of parallel Haskell [?] and Open MP[?]. Loidl [?] reported a parallel C# implementation. Kerridge [?]reported on the use of the new language Groovy in conjunction with JCSP. The language Python was used by Sampson [?]in his implementation. Appart from the results reported for the SCC the other systems were run using multi-core Xeons clocked at about 2.4GHz.

One problem with analysing the results is that whilst a word sequence of length 4 is probably long enough to pick out unique phrases in the Bible, some participants used much longer word lengths, which will have made their output somewhat more verbose, some also used different input files which again makes the results hard to interpret. Some participants only gave relative timingings of their parallel and sequential implementations rather than absolute times. In the summary of the results in Table we only show those implementation that

Algorithm 1 Shell script to run on host to run concordance on 32 scc cores.

```
rm /shared/stdout/*
pssh -t 800 -h hosts32 -o /shared/stdout /shared/sccConcordance32
cat /shared/stdout/* |sort > WEB.con
```

TABLE IV
BEST TIMES REPORTED FOR PHASE 1.

Implementation	Tasks	N	Time
Java Fork/Join	1	4	134.5sec
Hadoop Map/Reduce	57 in Beowulf cluster	10	36sec
Haskell	8	4	27sec
Groovy	12	4	61sec
Python	16	3	2sec
C on SCC at 0.533 Ghz	32	4	34sec
C on MarcHost	2	4	0.6sec

are using the same text file. It is also not always clear whether people were reporting results that included the time to print the final concordance.

However the final conclusion with respect to the SCC is pretty clear. Performances using it fell roughly in the middle of the range, with speed being of the same order as the Hadoop and Haskell implementations. The most sucessfully highly parallel version was certainly the Python one, but by a small margin the C version on the Marc host beat this using only 2 processes. Since the SCC example was using exactly the same C code as the version run on the host, it should have been fast, so the fact that even using 32 cores it took some 50 times as long is disappointing.

II. PHASE II

The second phase of the challenge was an N-body gravitational problem. If we consider the problem of predicting the motions of a large group of bodies under gravity. This is inherently a problem of order N^2 on a sequential machine since each body interacts with every other under gravity. As such it makes a better candidate for parallelisation than the concordance problem since the latter was of order N and tended to be I/O bound. There are many example benchmark programmes that deal with the N-body problem. SICSA took a C programme from the Programming Languages Shootout website as a reference implementation and modified it slightly so that it handled 1024 bodies rather than 5. The starting positions, masses and velocity vectors of all bodies was provided as a text file. There were thus 7 floating point numbers describing each body.

If we consider the general complexity of this problem under parallelism, it is clear that one component of the run time should shrink as the number of processors added increases. During each phase of the simulation we have for each body to accumulate the forces on it due to all other bodies. Since these calculations are independent, they can in principle be done using different processors in parallel. If p is the number of processors, this stage should have a cost $\alpha\frac{N^2}{P}$ for some constant α. After this calculation has been done, all of the processors would have to ensure that all other processors have access to the same updated data on planetary positions. For a uni-processor this is unproblematic, there is only a single state vector in RAM. For multiprocessors, depending on their design, this communications phase can be an appreciable overhead. If the communications is done naively, this has a cost in terms of data transfered that $\sim p^2N/p$ because processor to processor messages will grow as p^2 and each message will have to send data on N/p planets, we can thus model the overall time taken per simulation step as something like

$$t = \alpha\frac{N^2}{p} + \beta Np + \gamma \tag{II.1}$$

For a shared memory multiprocessor, the communications mechanism is effectively the memory bus in association with the cache coherency mechanism, since each processor will have updated its local cache copy of its 'own' planets' positions in phase space, and these local cache copies will have to propagate to the other machines, but this work is also proportional to Np, since each of the p caches has to read a complete copy of the positions of each of the planets. Other communications architectures, including the one we used on the SCC have a similar cost structure.

A. Approaches taken

The compiler group at the Glasgow University School of Computing Science has performed evaluations of the Phase II challenge using a number of our experimental parallelising compilers[**?**], [**?**], [**?**]. In this paper we will give a detailed account of one of these, the Lino system, as it was initially designed with the SCC as its target architecture. We will give a shorter account of other systems presented at the workshop because they are provide benchmarks against which the SCC performance can be measured.

B. Lino

Lino is a scripting language originally targeted at the SCC, but which also runs on other Linux machines. It allows Unix shell commands to be placed on 'tiles' which represent individual processors in an array of available processors. A tile in Lino is represented thus `[` *cmd0; cmd1;...* `]` where *cmd0* etc is some shell command.

Tiles can be named, and can be laid out in a rectilinear grid using the — and _ operators. The — operation can be used to form a horizontal pipeline of processes running on different processors. The _ operator can be used to form a vertical pipeline. Shell commands communicate with those on adjacent processes by using appropriately named channels. Thus the sequence

```
[ls >East]|[sort <West >sortedfile]
```

will cause the ls command to run on one core sending its output to the east, where it is read by the sort command on a second core whose output goes to sortedfile.

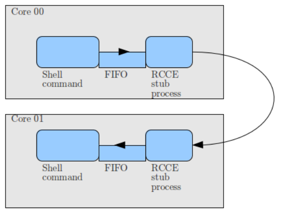

Fig. II.1. On the SCC channels pass via FIFOS and RCCE relay processes.

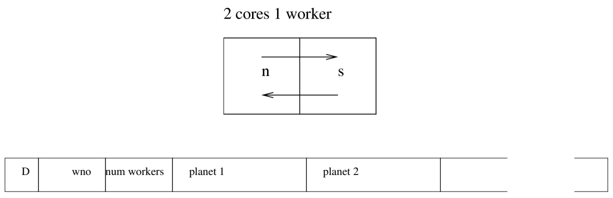

Fig. II.2. Top, a 2 core Lino layout for the N-body problem. Bottom, the format of the packages used in communication between controller and workers.

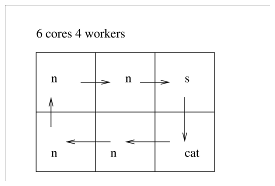

Fig. II.3. A layout with 4 worker cores.

Geometric operations of 90° rotation and reflection are supported on tiles or rectangular tile blocks. Tiles can be replicated horizontally or vertically.

The Lino compiler translates into standard bash shell scripts. In the case of the SCC each tile is allocated a processor core, on other machines each tile becomes a Linux process. In the latter case the channels are mapped onto appropriately named Linux FIFO file. On the SCC FIFO files do not work between cores so a five stage communications process operates as shown in Figure II.1. Data passed down a FIFO to a RCCE relay process on the same core and is then sent as RCCE messages to a corresponding relay process on another core before being finally piped to another shell command. This approach allows unmodified C and shell programs to be linked up in the multi-core environment without the programs having too know about the underlying communications mechanism. It also allows us to benchmark parallel applications both on the SCC and other Intel processors, using the same C programmes on both machines.

Here is a simple Lino script to run a potentially parallel version of the N-body problem:

```
controller = [./starter1 >East <East];
worker = [./nbody >West <West];
main = controller | worker;
```

The corresponding layout is shown in Figure II.2. There are two types of tiles : worker and controller. A controller tile runs the C programme starter1 which goes through the following sequence

1) Read in the initial position of the planets from a file.
2) Request the worker/s to one simulation step on the data by:
   a) write the planet data preceded by a "D" header on standard output,
   b) wait for a corresponding "D" package on standard input.
3) Request the worker/s to update the data by:
   a) write the planet data preceded by a "U" header on standard input
   b) wait for the corresponding "U" package on standard input and remember the new planet positions.
4) If the required number of simulation steps have been done, send the worker/s an "S" message on standard output and terminate, otherwise go to step 2.

The N-body programme itself is a lightly modified version of the reference single processor implementation in C, it sits in a loop reading messages on standard input. Each message read from the controller takes the form shown at the bottom of Figure II.2. The N-body programme branches on the first character of the message as follows:

- If the header is a "D" then increment the `wno` field and write the message to standard output. Then set `p= num workers` and simulate the dynamics of $1/p$ planets for one timestep, and remember their new positions in phase space.
- If the header is a "U" then increment the wno field, and copy into the message the updated positions of the planets for which worker wno is responsible for before writing the message to standard output.
- If the message is "S" echo it to standard output and terminate.

Our approach allows you to vary the number of workers associated with each controler without changing the C code. For example to have 4 workers we use the Lino script:

```
nwcorner = [./nbody >East <South];
swcorner = [./nbody >North <East];
scorner = [./starter4.sh >South <West];
corner = [cat >West <North];
passright = [./nbody >East <West];
```

TABLE V
N-BODY IN LINO ON SCC AND ON AN 8 CORE XEON, TIMES ARE IN MILLISECONDS/SIMULATION TIMESTEP.

Nbody tiles	Total Tiles	Xeon time	SCC time
16		8.1	2032
8		7.8	1025
4		9.9	702
2		17.1	648
1		30.5	967

TABLE VI
COMPARISION WITH OTHER APPROACHES AT THE WORKSHOP, ALL TIMES ON 8 CORE XEONS. RESULTS ORDERED BY OVERALL PERFORMANCE. WHERE MULTIPLE RESULTS WERE REPORTED FOR A GIVEN LANGUAGE PROCESSOR PAIR WE GIVE THE FASTEST REPORTED.

Language	threads	time		clock Ghz
Glasgow Pascal SSE code	16	1.75	ms	2.4
C++ threaded building blocks, SSE	12	2.05	ms	2.27
Glasgow Pascal AVX code	4	2.12	ms	3.1
Lino on Xeon	10	7.8	ms	2.4
Go	16	8	ms	2.4
C sequential	1	14	ms	2.4
Eden	8	16.6	ms	2.5
C#	12	18.2	ms	2.33
Glasgow Fortran (E#) on CellBE	12	23	ms	3.2
GCC on the CellBE	1	45	ms	3.2
Glasgow Pascal on CellBE	4	48	ms	3.2
Gnu Fortran on CellBE	2	82	ms	3.2
Lino on SCC	2	648	ms	0.533

```
passleft = [./nbody >West <East];
main = (nwcorner | passright | scorner)_(swcorner | passleft | corner);
```

This gives the layout shown in Figure II.3. We have tested layouts for 1,2,4,8, and 16 worker cores on the SCC and on an 8 core Xeon clocked at 2.4Ghz. On both machines the same C and Lino code was used. Results are given in Table V. As with the results in Tables III and II for the Phase I challenge, the SCC performance was very slow compared to that obtained on other Intel multi-core chips. The SCC is almost two orders of magnitude slower than the Xeon. Some of this may be attributable to the earlier version of GCC used on the SCC, and some of it to the earlied Pentium design used. But one might have hoped that these disadvantages would have been offset by the opportunity too use more parallelism. On the contrary we find that the SCC implementation peaks at 2 worker processes, whilst the Xeon peaks at 8.

Fitting Equation II.1 to the data in Table V we obtain for the Xeon $\alpha = 27$ns and $\beta = 223$ns whereas for the SCC $\alpha = 677$ns and $\beta = 94\mu$s. Recall that α is the time to compute the interaction between two planets and β the time taken to communicate one planets data between two workers. The SCC is slower on both counts, but is much slower on communications. This means that the level of parallelism that can be supported before the costs of communications comes to dominate is lower on the SCC.

C. Other Implementations

Thomas Horstmeyer [**?**] reported on an implementation using Eden[**?**]. As Table VI shows, this had a relatively poor performance, being slower than the single thread C reference version, and about half the speed of Lino on the same hardware. The C#[**?**] implementation reported by Loidl was similar. Sampson[**?**] whose Phase I entry was very fast, reported on an impressive implementation using SSE vector intrinsics and Threading Building Blocks. This appears to have one of the fastest performances of all, which is a credit to the efficiency of the TBB and the gains to be had from SSE intrinsics. The Glasgow results [**?**], [**?**], [**?**] are polarised according to the processor and type of language used. Lino and Go are slower than Pascal, the Cell is slower than conventional Intel machines, and the SCC is slower than the Cell. This ranking of machines is born out accross all results reported at the workshop. The lower clock speed of the SCC is clearly a factor that has to be taken into account here. If we normalise for clock speed, the Lino on the SCC falls into the same range of performance as Gnu Fortran on the Cell.

III. CONCLUSIONS