The SCC and the SICSA Multi-core Challenge

Paul Cockshott and Alexandros Koliousis

***Abstract*—Two phases of the SICSA Multi-core Challenge have gone past. The first challenge was to produce concordances of books for sequences of words up to length N; and the second to simulate the motion of N celestial bodies under gravity. We took both challenges on the SCC, using C and the Linux Shell. This paper is an account of the experiences gained. It also gives a shorter account of the performance of other systems on the same set of problems, as they provide benchmarks against which the SCC performance can be compared with.**

I. Introduction

The SICSA Multi-Core Challenge[1] is an open competition called by the Scottish Informatics and Computer Science Alliance (SICSA) to develop multi-core implementations of a set of predefined problems. Its aim is to learn about the strengths and weaknesses of current systems for parallel programming by comparing them on common grounds.

So far, two phases (viz. Phase I and II) of the Challenge have been run, having attracted entries from teams accross Europe. Each phase was announced with a problem specification, together with a candidate serial implementation for that problem. Participants then had to select a programming language, a host architecture, and a paralellisation system with the aim of achieving either the fastest implementation, or the best acceleration, relative to the performance of the serial implementation on that architecture. The results from Phase I were reported at a workshop at the Heriot-Watt University, on the 13th of December, 2010; results from Phase II were reported at a workshop at the University of Glasgow, on the 27th of May, 2011.

We have implemented both of the challenges posed by SICSA on the SCC. Our programming language of choice was C, and the parallelization system was the Linux Shell – in particular, *Lino*, a process-algebra for chips like the SCC that translates into Linux Shell commands. This paper describes the problems, the SCC implementations, and then contrast these with other reported implementations, both in terms of design and in terms of performance.

II. Phase I

The first phase of the SICSA Multi-core Challenge was to create concordances of books. The inputs to the problem were a file containing English text in ASCII encoding; and an integer N. The challenge was to find the number of occurrences of all sequences of words up to length N, together with a list of start indices. Optionally, sequences with only one occurrence could be omitted.

In addition to the problem specification, a reference implementation in Haskell was provided, together with several reference texts. In practice, most work was done with the longest of texts, the *World English Bible*, which is approximately 4.79MB in size; and Elizabeth Gaskell's *The Manchester Marriage*, a short story that is 3.58KB in size.

TABLE I
Serial benchmarks for Phase I on a twin-core 2.6GHz Intel platform

Language	OS	Print	File Size (bytes)	Time (sec)
Haskell	Windows	yes	4792092	$> 2h$
Haskell	Windows	yes	3580	0.824
C	Windows	no	3580	0.028
C	Windows	yes	3580	0.029
C	Windows	yes	4792091	3.673
C	Windows	no	4792091	0.961
C	Linux	yes	4792091	2.680
C (-O3)	Linux	yes	4792091	2.250
C	Linux	no	4792091	1.040
C (-O3)	Linux	no	4792091	0.899

A. An Improved Serial Implementation

Prior to doing any parallelisation, it is advisable to initially set up a good sequential version. Intuitively, the concordance problem is of either linear or, at worst, log-linear complexity, and for such problems – especially ones involving text files – the time taken to read the file and print out the results can easily dominate the execution time. If a problem is I/O-bound, then there is little advantage in expending effort to run it on multiple cores. However this hypothesis needed to be verified by experiment. In order to optain an efficient and non-esoteric sequential implementation, C was chosen as the implementation language. The algorithm performed the following steps:[2]

1) read the entire 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 a 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 specified;
5) if the results are to be sorted, pipe them through Linux `sort` command.

Table I shows that the performance of the C implementation was very much faster than that obtained using the Haskell reference code. Our results 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.

B. The Parallel Implementation

The concordance problem is hard to parallelise efficiently. For example, one can not just split a book into two halves,

[1]www.macs.hw.ac.uk/sicsawiki/index.php/MultiCoreChallenge

[2]Available at www.dcs.gla.ac.uk/w̃pc/reports/SICSA/concordance.c

TABLE II
PARALLEL BENCHMARKS FOR PHASE I ON A TWIN-CORE 2.6GHZ INTEL PLATFORM

Program	OS	Time (sec)
concordance2.c	Windows	5.630
concordance2.c	Linux	2.257
conc.sh	Linux	2.120

prepare a concordance for each half, and then merge the results together; a repeated word might be missed if it was mentioned once in the first half and once in the second half. Thus, a more complicated approach was needed. The problem was parallelised 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. Then, each thread could create 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 (viz. `concordance2.c`). The programme was tested on the same dual-processor machine as the original serial version of the algorithm. Table II shows the results for creating a concordance of the Bible (WEB.txt).

There was no gain using multi-threading 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, on the other hand, there was a small gain in performance due to multi-threading – about 17% faster in elapsed time using 2 cores. Since a large part of the program execution is spent printing the results, this proved a challenge to improve using multiple cores.

The first parallel version adopted the strategy of allowing each thread to write its part of the results to a different file, which were later merged and sorted. A second parallel verion followed the same basic strategy as the previous one, but used the Linux shell, instead of Pthreads, to fork parallel processes. This latter parallel version (viz. `conc.sh`) communicates via files using the following set of commands:

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

In this `l1concordance` is the concordance programme and parameters `4 P 1 0` are: N the maximum number of words in a phrase, `P` indicates that printing is enabled, `1` is the mask to be applied to the hashcode of phrases and `0` the value that must result from this masked hash if the phrase is to be handled by this task. As shown in Table II, this version had the best performance of the lot.

TABLE III
SCC PERFORMANCE ON THE CONCORDANCE PROBLEM

Implementation	Time (sec)
1 core; full concordance	26.17
1 core; $^1/_2$ concordance	13.48
1 core; $^1/_8$ concordance	5.59
2 cores; $^1/_2$ concordance each	49
8 cores; $^1/_8$ concordance each	36
32 cores; $^1/_{32}$ concordance each	34
1 core on host processor; full concordance	1.03
2 cores on host processor and the Shell; full concordance	0.685

C. SCC Experiments

The SCC is configured with a host processor, a conventional modern Intel x86 chip. Attached to it 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 (accessed) on (from) a shared NFS system.

Table II-C shows the results from the SCC experiments. Looking at the time it took one SCC core to complete the full concordance 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. The top 3 lines of the table show the effects of trying to do smaller portions of the workload on an individual core.

We dispatched 32 tasks on the SCC cores using the `pssh` command as shown in the following commands:

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

The first line simply 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 finished, outputs are concattenated and sorted to yield the final concordance file. The script `sccConcordance32` invokes the actual concordance task:

```
cd /shared
./l1concordance WEB.txt 4 P 31 $(hostname)
```

The hostname command (returning `rck00`, or `rck01`, and so on) is used to derive a process ID, which is then used to select which words will be handled by each task. The 4^{th} 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 becomes clear from the results that on an I/O-bound task like this, the SCC has poor performance.

D. Other implementations

Phase I concluded with a workshop at the Heriot-Watt University, were 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 problem [**?**]; Stewart reported on the use of Hadoop

TABLE IV
BEST TIMES REPORTED FOR PHASE I

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

Map/Reduce [?]; Al Jabri reported on the use of parallel Haskell [?] and OpenMP [?]; Loidl reported on a parallel C# implementation [?]; Kerridge reported on the use of the new language Groovy in conjunction with JCSP [?]; and Sampson on the use of Python [?]. Apart from the results reported for the SCC, the other systems were run on multi-core Xeons, clocked at about 2.4GHz. The results are summarized in Table IV.

One problem with the analysis of these results is that, whilst a word sequence of length $N = 4$ is probably long enough to pick out unique phrases in the Bible, some participants used much longer word lengths, which must have made their output more verbose; some also used different input files, which again makes the results hard to interpret; and other participants gave only relative timings of their parallel and sequential implementations rather than absolute times. The summary of the results in Table IV shows only those implementations that are using the same text file (the Bible, i.e. WEB.txt). It was not always clear whether the reported results included the time to print the final concordance.

Nonetheless, the final conclusion with respect to the SCC is clear. Its performance falls roughly in the middle of the range, with its speed being of the same order as the Hadoop and Haskell implementations. The most sucessfull, highly-parallel version was certainly the Python one, but by a small margin the C version on the MARC host beat its time using only 2 processes. Since the SCC experiments were using exactly the same C code as the version run on the host processor, it should have been fast. The fact that even with 32 cores it took approximately 50 times longer is disappointing.

III. Phase II

The second phase of the challenge was an N-body gravitational problem – a problem of predicting the motions of a large group of N 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.[3] There are many exemplar benchmark programmes that deal with the N-body problem. SICSA took a C programme from the Computer Languages Benchmarks Game[4] 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 bodies in three dimensions were provided as a text file. There were thus seven floating point numbers describing each body.

[3]Recall that the latter was of order N and tended to be I/O bound.

[4]Cf. http://shootout.alioth.debian.org/

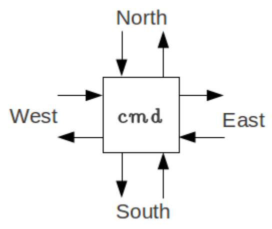

Fig. 1. A Lino tile.

If we consider the general complexity of this problem under parallelism, one component of the execution time should shrink as the number of processors increases. During each round of the simulation, the program has to accumulate the gravitational forces imposed on each body by 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 $\alpha \in \mathbb{R}$. 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 memory. For multi-processors, however, depending on their design, this communications phase can be an appreciable overhead. If the communications is done naively, the data-transfer cost is $\beta\frac{p^2N}{p}$, for $\beta \in \mathbb{R}$, 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

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

For a shared memory multi-processor, 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 used on the SCC have a similar cost structure.

A. Lino

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 [?], [?], [?]. This section gives a detailed account of one of those, the Lino system.

Lino is a scripting language originally targeted at the SCC, but it 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 as [*cmd0; cmd1; ...*] where *cmd0*, *cmd1*, 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

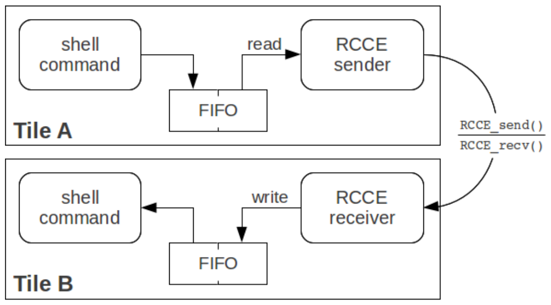

Fig. 2. On the SCC, channels pass via named pipes and RCCE relay processes.

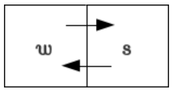

Fig. 3. A 2-core Lino layout for the N-body problem, with one starter (tile s) and one worker (tile w).

used to form a horizontal pipeline of processes running on different processors; and the '_' operator can be used to form a vertical pipeline. Shell commands communicate with those on adjacent processes by using appropriately named channels, namely North, West, South, and East (Figure 1). Thus, the sequence

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

will cause the `ls` command to run on one tile, sending its output to the east, where it is read by the `sort` command on a right-adjacent tile, whose output goes to a sorted `file`. Geometric operations of 90° rotation and reflection are also supported on tiles or rectangular tile blocks. Tiles can be replicated horizontally or vertically. For more details on the Lino algebra, see []. [Need a reference here.]

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, however, FIFO files do not work between cores so a five-stage communications process operates as shown in Figure 2.

When data are passed down a FIFO, a RCCE relay process on the same core reads and sends them 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 SCC multi-core environment without the programs having to 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 programmes on both machines.

Without further ado, here is a simple Lino script to run a potentially parallel version of the N-body problem:

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

The corresponding layout is shown in Figure 3. For the N-body problem, consider two types of tiles, worker and controller. Tiles are connected in a circle. The controller communicates with p workers by messages. A message starts with the character "D", or "U", or "S" to instruct the workers to advance, or update, or terminate the celestial motion, respectively; followed by the number of workers p; followed by the current number of hops the messages has traversed; followed by the positions, velocities, and masses of N bodies – all in all, a 65KB message. A controller tile runs the C programme `starter` which goes through the following sequence:

1) read in the initial position of the planets from a file;
2) request from the worker(s) to perform one simulation step on the data by:
   a) writing the planet data, preceded by a "D" character, on standard output, and
   b) waiting for the corresponding "D" message to arrive on standard input;
3) request the worker(s) to update the data by:
   a) writing the planet data preceded by a "U" character, on standard output; and
   b) waiting for the corresponding "U" message to arrive on standard input, and then storing the new planet positions.
4) If the required number of simulation steps have finished, send the worker(s) an "S" message on standard output and terminate, otherwise go to step 2.

The N-body worker programme itself (referred to as `nbody`) is a slightly modified version of the reference single processor implementation in C, waiting (in a loop) to read messages on standard input. The N-body programme branches on the first character of the message as follows:

- if the header starts with a "D", then increment the increment the number of hops and write the message to standard output. Then, simulate the dynamics of N/p planets for one timestep, and remember their new positions in phase space;
- if the header starts with a "U", then increment the number of hops, and copy into the message the updated positions of the planets for which the worker is responsible for, before writing the message to standard output;
- if the message starts with an "S", terminate.

This approach allows us 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];
passleft = [./nbody >West <East];
top = [nwcorner | passright | scorner];
bottom = [swcorner | passleft | corner];
main = top _ bottom;
```

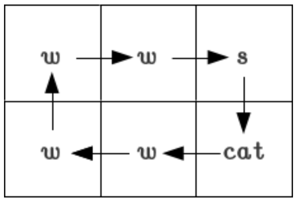

Fig. 4. A layout with 4 worker cores.

TABLE V
TIME/SIMULATION STEP OF THE N-BODY PROBLEM IN LINO ON THE SCC AND ON AN 8-CORE XEON

N-body workers (cores)	Time on Xeon (ms)	Time on SCC (ms)
16 (20)	8.1	2032
8 (10)	7.8	1025
4 (6)	9.9	702
2 (4)	17.1	648
1 (2)	30.5	967

This gives the layout shown in Figure **??**. We have tested layouts for 1, 2, 4, 8, and 16 worker cores, both 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 **??**. As with the results in Tables II-C 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 attributed to the earlier version of GCC used on the SCC, some to the slower clock used 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. That this slowdown is due to the inter core communication mechanism on the SCC rather than to the use of Linux FIFOs, is proven by the fact that the Xeon Lino implementation which also used FIFOs but which did not use RCCE was so much faster.

Fitting Equation 1 to the data in Table **??**, 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.

B. Other Implementations

Similar to the first phase of the SICSA Multi-core Challenge, Phase II concluded with a workshop at the University of Glasgow. The results are summarized in Table **??**, ordered by their overall performance. Where multiple results were reported for a given language/processor pair, we give the fastest time reported.

Thomas Horstmeyer [**?**] reported on an implementation using Eden [**?**]. As Table **??** 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 had similar performance [**?**]. 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.

TABLE VI
BEST TIMES REPORTED FOR PHASE II ON 8-CORE XEONS

Implementation	Threads	Time (ms)	Clock (Ghz)
Glasgow Pascal (SSE)	16	1.75	2.4
C++ (SSE)	12	2.05	2.27
Glasgow Pascal, AVX	4	2.12	3.1
Lino on Xeon	10	7.8	2.4
Go	16	8	2.4
C sequential	1	14	2.4
Eden	8	16.6	2.5
C#	12	18.2	2.33
Glasgow Fortran (E#) on Cell	12	23	3.2
GCC on Cell	1	45	3.2
Glasgow Pascal on Cell	4	48	3.2
Gnu Fortran on Cell	2	82	3.2
Lino on SCC	2	648	0.533

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, although the lower clock speed of the SCC is clearly a factor that has to be taken into account here. Indeed, if we normalise for clock speed, the Lino on the SCC falls into the same range of performance as GNU Fortran on the Cell.

IV. CONCLUSIONS

The SCC is described as a Single Chip Cloud. The performances we have observed for it indicate that this may be an accurate description. On the concordance application the SCC performance most closely resembled that of Hadoop on a Beowulf cluster - a more classic cloud configuration. Compared however with other multi-core chips : Nehalem, Sandybridge or the CellBE, the SCC performs poorly on both applications. We believe from our experiments, particularly those for Phase II, that the underlying cause for the uncompetitive performance of the SCC is the inefficiency of the inter-core communications system. Unlike the CellBE which performs inter-core communications using high speed DMA, or the Nehalem which uses cache coherence hardware, the SCC relies on software message passing in small shared buffers. We conclude that if tesselation processors like the SCC are to be viable, they will require high performance DMA hardware.

A separate conclusion from our experiments is that the old Unix shell model of parallelism: C programmes communicating via files and pipes, is still remarkably effective. It gave the highest performance for the Phase I problem and for Phase II, it was only beaten by compilers that made explicit or implicit use of SIMD parallelism.

Algorithm 2 The single worker N-body example compiled for the SCC.

```
#!/bin/sh
shift `expr $1 + 2`
[ ! -d $1 ] && exit 1
cd $1
case `hostname` in
rck00)
mkfifo fifos/East0_0
mkfifo fifos/East0_0in
./apps/HELLO/hello 2 0.533 00 02 --from fifos/East0_0 --to /dev/null &
./apps/HELLO/hello 2 0.533 02 00 --from /dev/null --to fifos/East0_0in  &
./starter.sh  > fifos/East0_0 < fifos/East0_0in  &
wait
rm fifos/East0_0
rm fifos/East0_0in
;;
rck02)
mkfifo fifos/West0_1
mkfifo fifos/West0_1in
./apps/HELLO/hello 2 0.533 02 00 --from fifos/West0_1 --to /dev/null &
./apps/HELLO/hello 2 0.533 00 02 --from /dev/null --to fifos/West0_1in  &
./nbody  > fifos/West0_1 < fifos/West0_1in  &
wait
rm fifos/West0_1
rm fifos/West0_1in
;;
esac
```

Algorithm 1 The single worker N-body example compiled for a shared memory Linux machine.

```
rm fifos/*
mkfifo fifos/East0_0
mkfifo fifos/West0_1
./starter1.sh  >fifos/East0_0 <fifos/West0_1&
./nbody  <fifos/East0_0 >fifos/West0_1&
wait
```

REFERENCES

[1] J. Singer, "Java/Fork Join Implementation," in *First SICSA Multi-core Challenge Workshop*, Heriot Watt University, December 2010. [Online]. Available: http://www.dcs.gla.ac.uk/ jsinger/pdfs/sicsa_concord_101213.pdf

[2] R. Stewart. (2010, December) Hadoop MapReduce Concordance. SICSA Multi-Core Challenge Phase I Workshop. Heriot Wat University. [Online]. Available: http://www.macs.hw.ac.uk/ rs46/multicore_challenge1/Hadoop _concordance.pdf

[3] M. Aljabri. (2010, December) A Parallel Concordance Benchmark, Haskell Implementation. SICSA Multi-core Challenge Phase I Workshop. Heriot Watt University. [Online]. Available: http://www.macs.hw.ac.uk/ dsg/events/MultiCoreChallenge/slides/aljabri _mcc10.pdf

[4] ——. (2010, December) A Parallel Concordance Benchmark, OpenMP Implementation. SICSA Multi-core Challenge Phase I Workshop. Heriot Watt University. [Online]. Available: http://www.macs.hw.ac.uk/ dsg/events/MultiCoreChallenge/slides/aljabri _mcc10.pdf

[5] H.-W. Loidl. (2010, December) Parallel Concordance in C#. SICSA Multi-core Challenge Phase I Workshop. Heriot Watt University. [Online]. Available: http://www.macs.hw.ac.uk/ dsg/events/MultiCoreChallenge/slides/hawo _mcc10.pdf

[6] J. Kerridge. (2010, December) SICSA Concordance Challenge:Using Groovy and the JCSP Library. SICSA Multi-core Challenge Phase I Workshop. Heriot Watt University. [Online]. Available: http://www.macs.hw.ac.uk/ dsg/events/MultiCoreChallenge/slides/jon _mcc10.pptx

[7] A. Sampson. (2010, December) "This is a parallel parrot". SICSA Multi-core Challenge Phase I Workshop. Heriot Watt University. [Online]. Available: http://offog.org/publications/mcc201012-python-slides.pdf

[8] P. Keir, W. Cockshott, and A. Richards, "Mainstream parallel array programming on cell," in *5th Workshop on Highly Parallel Processing on a Chip (HPPC 2011)*, 2011. [Online]. Available: http://eprints.gla.ac.uk/54875/

[9] W. Cockshott and G. Michaelson, "Orthogonal parallel processing in vector pascal," *Computer Languages, Systems and Structures.*, vol. 32, no. 1, pp. 2–41, April 2006. [Online]. Available: http://eprints.gla.ac.uk/3451/

[10] Y. Gdura and W. Cockshott, "A virtual simd machine approach for abstracting heterogeneous multi-core," in *ICT 2011 18th International Conference on Telecommunications*, 2011. [Online]. Available: http://eprints.gla.ac.uk/56324/

[11] M. D. Tobias Sauerwein, Thomas Horstmeyer. (2011, May) N-Body in Eden - A skeletal approach in a distributed memory setting. SICSA Multi-core Challenge Phase II Workshop. Glasgow University. [Online]. Available: http://www.mathematik.uni-marburg.de/ horstmey/sicsa/NBodyEdenSlides.pdf

[12] A. Black and U. of Washington. Dept. of Computer Science, *The Eden programming language*. Dept. of Computer Science, University of Washington, 1985.

[13] H.-W. Loidl. (2011, May) A C# implementation of the n-body problem . SICSA Multi-core Challenge Phase II Workshop. Glasgow University. [Online]. Available: http://www.macs.hw.ac.uk/ dsg/events/MultiCoreChallenge/slides/ mcc11.pdf

[14] A. Sampson. (2011, May) Colliding Blobs with Threading Building Blocks . SICSA Multi-core Challenge Phase II Workshop. Glasgow University. [Online]. Available: http://www.mathematik.uni-marburg.de/ horstmey/sicsa/NBodyEdenSlides.pdf

[15] Y. G. P. Cockshott. (2011, May) Vector Pascal implementations running on Nehalem and Cell processors . SICSA Multi-core Challenge Phase II Workshop. Glasgow University. [Online]. Available: http://www.dcs.gla.ac.uk/ jsinger/pdfs/wpc_multicore.pdf

[16] P. Keir. (2011, May) All-pairs n-body in Fortran for CellBE . SICSA Multi-core Challenge Phase II Workshop. Glasgow University. [Online]. Available: http://www.dcs.gla.ac.uk/people/personal/pkeir/mcore2.pdf

[17] I. McGinniss. (2011, May) Naive approaches to n-body parallelism using Google Go . SICSA Multi-core Challenge Phase II Workshop. Glasgow University. [Online]. Available: http://prezi.com/qrgmjzexqvgp/naive-approaches-to-n-body-parallelism-with-google-go/