Parallelization of the ISAAC Code

Steve Johnson, Parallel Software Products Inc.
Technical Report TR-2004-10-01

1 Introduction

ISAAC is an open source multi-block numerical simulation application code. The aim of the parallelization of the ISAAC code is to enable scalability onto large numbers of processors where that parallel version is applicable to as wider range of parallel systems as possible. This should allow as many users as possible to exploit parallel execution on their own parallel systems. The scope of this work was restricted to a basic multi-block parallelization using message passing communications (e.g. MPI) with OpenMP directives automatically added by the ParaWise/CAPO [1] computer aided parallelization (CAP) environment to allow parallelism within each block to be exploited. This scope prevented the development of an intra-block message passing parallelization due to the significant effort required for this to be implemented.

The parallelization has been performed with an emphasis on altering as small amount of the ISAAC source code as possible. The resultant message passing inter-block parallelization has only required alterations to a small proportion of the routines and modules of ISAAC, with significant changes restricted to the main program and some of the I/O routines. The OpenMP parallelization performed by the ParaWise/CAPO environment involves minor alterations (mainly OpenMP directive addition) to many routines. As a result, an inter-block message passing only version and an intra-block OpenMP only version are provided in addition to the hybrid inter-block and intra-block parallelization that includes message passing and OpenMP.

Although the ParaWise multi-block message passing automatic parallelization facility is currently not mature enough to handle a code as complex as ISAAC, the information provided by ParaWise has proven invaluable in the manual parallelization. Navigation of the code structure and understanding of the dataflow have significantly eased the parallelization process and enabled it to be completed in a far shorter time frame than would otherwise have been possible. This not only greatly aided the addition of communications in ISAAC, but also made the resultant debugging of the parallel version far less complicated. The OpenMP parallelization was a fairly painless process as the CAPO module, developed at NASA Ames Research Center, automatically detected parallelism and added the OpenMP directives. The only interaction required was the selection of certain loops that did not perform well in parallel, allowing CAPO to select alternate loops that then provided far superior speedup and scalability in the resultant OpenMP parallel version.

2 Inter-Block Message Passing Parallelization of ISAAC

2.1 Basic Parallelization Strategy used in the Parallelization of ISAAC

The first phase in the parallelization was to implement the inter-block parallelization using message passing communication to facilitate the inter-block interactions of ISAAC. Parallelism is exploited by allocating sets of blocks to each processor where all computations relating to those blocks are performed on that processor.

The processors are each allocated a unique id number (CAP_PROCNUM) where these range from processor 1 to processor CAP_NPROC (as determined at runtime in the command line that executes parallel ISAAC). To run parallel ISAAC, the command takes the form:-

```
caprun –top full20 isaac_parallel < data.in
```

where *caprun* is a CAPLib command to instigate parallel execution and *–top full20* indicates the processor topology to be used. In this example, a topology of 20 processors where every processor can communicate directly with every other processor is setup. The CAP_NPROC variable is set by this topology and is known by every processor.

These values are set in a call to the CAPLib routine CAP_INIT made at the start of the main program.

2.2 Determination of Block Distribution to Processors

The determination of the distribution of blocks should aim to achieve load balance between processors as well as minimizing the volume of inter-processor communication. Ideally, a graph partitioner such as JOSTLE [2] or METIS [3] would be used to achieve both goals. These partitioners are exploited by creating a graph with nodes representing blocks, with weights relating to the numbers of cells/grid points in each block, and edges representing inter-block interactions (cuts in ISAAC), where each edge has a weight relating to the amount of data being moved between the blocks. Both partitioners distribute the nodes of the graph in an attempt to maximize load balance and minimize the volume of communication between processors (i.e. closely coupled blocks would be allocated to the same processor). The current parallelization does not use either graph partitioner for several reasons. The graph partitioner used by PSP (and used in automatically parallelized codes using graph partitioning such as finite element codes [4]) is JOSTLE, however, at present, JOSTLE is not open source and so would not be suitable for the open source ISAAC parallel version. METIS is open source, but in addition to our

lack of familiarity with using it, requiring users of open source ISAAC to additionally install METIS could well be a deterrent in the usage of the parallel version.

To avoid the practical problems associated with these graph partitioners, neither has been used and a simple, crude algorithm that only considers load balance has been implemented. As the coupling between blocks in an inter-block code is probably considerably less than that between finite elements in a finite element mesh, the cost of not considering inter-block communications in the block distribution may not be as crucial as it has proven for FE codes. The inclusion of a graph partitioner to calculate a more affective partition can easily be added to the existing parallelization if consideration of inter-block communications proves crucial.

2.3 Implementing the Block Distribution in ISAAC

To achieve the goal of minimizing the changes to the ISAAC code, each processor stores only the blocks it has been allocated, with those blocks renumbered to run from 1. An array that indicates which block is owned by which processor, based on the original global block numbers, is output from the simple block decomposition algorithm, i.e.

CAP_P(1) = 3	Original block 1 is owned by processor 3
CAP_P(2) = 1	Original block 2 is owned by processor 1
CAP_P(3) = 3	Original block 3 is owned by processor 3
CAP_P(4) = 2	Original block 4 is owned by processor 2
CAP_P(5) = 3	Original block 5 is owned by processor 3

Every processor has an identical copy of the CAP_P array. To map from local block numbers back to original global block numbers, each processor has another array that is different on every processor, i.e. on processor 3

CAP_LOC2GLO(1) = 1 Locally numbered block 1 is original globally numbered block 1
CAP_LOC2GLO(2) = 3 Locally numbered block 2 is original globally numbered block 3
CAP_LOC2GLO(3) = 5 Locally numbered block 3 is original globally numbered block 5

The ISAAC variable NBLKS is adjusted to contain the number of blocks allocated to this processor (i.e. NBLKS = 3 for processor 3 in the above example). The original number of blocks is also stored in new variable NGLOBAL_BLKS.

All information stored by ISAAC prior to the calculation of the block distribution is then renumbered on each processor to match the local block numbers. This includes all geometric information and inter-block CUT information, where only cuts relating to blocks owned by a processor are listed on that processor.

These changes allow the vast majority of ISAAC to operate with no code alterations as each processor has a smaller ISAAC mesh containing a few blocks that it processes as if it were the entire original input mesh. This block distribution and renumbering is all done

in routine CAP_MULTIBLOCK_SETUP in cap_utilities.F and is called from the main program of ISAAC.

2.4 Inter-Processor Communications

All communications in ISAAC use the CAPLib communication library [5] that is freely available from PSP. In its simplest form, CAPLib is just a thin layer that sits on top of MPI (where the source code that calls MPI can be included in the parallel ISAAC distribution so that users just need to link with MPI). Only a few of the functions in CAPLib have been required for ISAAC as listed below :-

CAP_SEND(Data,Length,Type,Direction)
CAP_RECEIVE(Data,Length,Type,Direction)
CAP_COMMUTATIVE(Value,Type,Function)
CAP_MCOMMUTATIVE(Data,Length,Type,Function)
CAP_COMMUPARENT(Value,Type,Function,FirstFound)
CAP_COMMUCHILD(Value,Type)

Where *DATA* is a scalar or the a position in an array, *VALUE* is a scalar; *LENGTH* is the number of items in an array (1 for a scalar); *TYPE* is the type of data (1=integer, 2=real, 3=double,7=character); *DIRECTION* is either *CAP_LEFT* (= -1) or *CAP_RIGHT* (= -2) where LEFT is towards processor 1 when a logical pipeline of processors is assumed, or if *DIRECTION* is greater than zero then it is the processor number to be communicated with; *FUNCTION* is a binary combination function such as *CAP_RADD* or *CAP_DMAX* for evaluating summations and maxima etc. *CAP_SEND* and *CAP_RECEIVE* are the basic synchronous communications; *CAP_COMMUTATIVE* evaluates scalar summations, maxima etc. combining contributions from all processors (local contributions passed in as *VALUE* and the resultant global value returned to all processors in the same variable); *CAP_MCOMMUTATIVE* evaluates summations, maxima etc. for all of the elements in the *DATA* array passed in and out; *CAP_COMMUPARENT* also evaluates scalar maxima/minima etc. but also detects which processor provided the final extreme value with input *FIRSTFUNC* indicating if the lowest numbered processor providing the extreme (when the value is repeated) should be used; *CAP_COMMUCHILD* always follows a *CAP_COMMUPARENT* as it simply broadcasts the input scalar from the processor that provided the final extreme value (e.g. for the location of the maximum or minimum value in the mesh).

2.5 Handling Inter-Block Interactions

Most of the inter-processor communications required during the execution of ISAAC relate to the inter-block interactions as listed in the CUTS array. This array has been renumbered on every processor to relate to local block numbers and ordered to only include interactions relevant to the blocks owned on the processor, where those involving

two blocks both owned by this processor are listed first. The variable NCUTS is renumbered to be the number of cuts relating two blocks on this processor to allow calls the routine CUT to be left unchanged. The inter-processor block interactions are handles by a new routine CAP_INTERBLOCK_COMMS that is called before calls to CUT.

CAP_INTERBLOCK_COMMS is based on routine CUT and follows that code to identify and place in a buffer data to be sent to another processor and also unpack data received from another processor into the correct locations. The communications are performed using CAP_SEND and CAP_RECEIVE calls where the communication schedule attempts to buffer, communicate and unbuffer in parallel as much as possible. The basic algorithm is that pairs of processors buffer the data required by the other processor in the pair, then they exchange the buffers and unpack the data they received. This allows both the processors in the pair to operate in parallel while other processors are also paired with other processors. Each processor will be paired with every other processor during the algorithm.

Another routine CAP_INTERBLOCK_GRDCOMMS performs the same algorithm as CAP_INTERBLOCK_COMMS, but is based on GRDCUT and is performed prior to a call to GRDCUT.

2.6 Reading and Writing Mesh Data to Disk

The basic strategy for reading and writing data from/to disk is to force processor 1 to perform all I/O so that files read/produced from parallel ISAAC will be identical to those used/produced by the original serial ISAAC. Most READ's in ISAAC relate to data required by every processor, so the read is performed by processor 1 and CAP_SEND and CAP_RECEIVE calls are used to broadcast the read data to every processor.

When data is read for every block from a single file, processor 1 performs all file reads, if the relevant block is owned by processor 1, the original read into the block variables is made, otherwise, a read into a buffer is made and this buffer is the sent to the appropriate processor where it is unpacked into the appropriate block variables. An exception to this is when different input files are to be read, one file per block. In this case, each processor can read the files for the blocks they own, not requiring any communications.

Similarly, when writing data for every block to a single file, data is sent from the owning processor to processor 1 where each block is processed in the original block numbering order to ensure the resultant file is identical to that from serial ISAAC so it can be read by the appropriate post-processing software etc.

Ideally, separate files for each block could always be read/written allowing parallel I/O without the need for communication, however the issues that may make this unacceptable relate to how the data read by ISAAC is produced and how data written by ISAAC is used. Both speed and ease of programming are simplified when separate files are used.

2.7 Compiling, Linking, Execution and Performance of the Inter-Block Message Passing Parallelization of ISAAC

Most of the ISAAC source code has not been altered for this parallel version, the altered files are main.F, rhs.F, io.F……A new file, cap_utilities.F has also been added to implement mesh decomposition across the processors, mesh renumbering and inter-block communications. Additionally, the CAPLib MPI "pack2go" source code must also be included along with the standard MPI libraries on the target machine when compiling and linking the parallel executable.

The parallel version can be executed using the standard mpirun command, passing in the processor topology using the –top flag i.e.

```
mpirun –np 4 isaac_interblock –top full4 < kw.1.inp
```

This will run 4 separate versions of isaac_interblock on the same machine where they will interact via MPI communications. To run on several machines, the –hosts flag can be used i.e.

```
mpirun –np 16 –hosts hydra2,hydra3,hydra4 isaac_interblock -top
      full16 < kw.1.inp
```

where the run is commenced on machine hydra1 and each of hydra1, hydra2, hydra3 and hydra4 will execute 4 MPI processes.

As the only multi-block test case available is the four block case, results for up to 4 processors only are presented. The machine used is a Compaq Alpha system.

Number of Processors	Speedup
2	1.90
3	2.42
4	2.40

The speedup on 2 processors is encouraging and is improved when an extra processor is added. The lack of improvement on 4 processors over 3 processors is due to the imbalance in the sizes of the blocks.

Even from this very restricted set of results, the effectiveness of the parallelization can be seen, and with a larger number of blocks, the potential for efficient scalability onto far larger numbers of processors is clear.

3 OpenMP Parallelization of ISAAC Using the CAPO Module of ParaWise

3.1 Exploiting Finer Grain Parallelism in ISAAC

With message passing being used to exploit parallelism between blocks in ISAAC, a considerable amount of finer grain parallelism still exists that can be exploited to both increase speedup and improve load balance. This parallelism typically exists in loops operating over a mesh dimension within a block (e.g. I, J or K loops) where a simple way to exploit this parallelism on a shared memory parallel system is to use OpenMP directives.

3.2 Using CAPO to Produce and Optimize OpenMP Version of ISAAC

OpenMP parallelization is performed by the CAPO module of ParaWise[6], taking the code structure and dependence graph information provided by ParaWise to automatically detect parallelism and determine any necessary variable privatization. The parallelization process was performed through a number of passes, where directives are generated, the code automatically produced is then executed in serial and in parallel, the profile information generated from those executions is then imported into ParaWise and the performance of the loops in the code is then inspected and changes made. These changes can relate to providing information about the values of variables used in ISAAC (such as constraint on the values of some variables) and also decisions about the dataflow in the algorithms implemented in ISAAC (where ParaWise has had to conservatively assume a dependence from an assignment to a subsequent usage and that dependence has inhibited parallelization). Additionally, some loops may have been selected for parallel execution where other alternative loops in the loop nest may provide a more efficient and scalable source of parallelism, so the user can tell CAPO to force serial execution of the ineffective loops to allow parallel execution of other loops. Additionally, some loops may benefit from an OpenMP DYNAMIC loop distribution schedule (i.e. reduce the load imbalance suffered by the loop suffered when using the default STATIC distribution), so the user can tell CAPO to enforce this for the subsequent parallel version generated.

The production of an efficient and scalable version of ISAAC using CAPO to generate OpenMP only require a few passes of the above process, requiring very little user time (a couple of hours), where without using CAPO the time required would at best be a few weeks, if not months.

3.3 Compiling, Linking, Execution and Performance of the OpenMP Parallelization of ISAAC

Most routines in the ISAAC source have been altered (i.e. OpenMP directives added) in the parallelization. To link, the appropriate OpenMP flag must be used in the compile and link commands, e.g. -omp on our Compaq Alpha system, -openmp on our Sun

Microsystems machine etc. Some of the flags and environment variables (as in the above) are system dependent.

To execute the parallel code, several machine parameters need to be set. The simplest way of doing this is to set environment variables to indicate the number of threads (i.e. processors) to be used and to set stack sizes to allow the required memory to be available at runtime. To set the number of threads to use, set the environment variable OMP_NUM_THREADS i.e.

```
setenv OMP_NUM_THREADS 4
```

to use 4 threads. To set the stack sizes, typically, the stack size in the current shell needs to be set as this is used for the master thread i.e.

```
limit stacksize unlimited
```

Additionally, the stack size available to the slave threads also needs to be specified. The value to use is problem dependent and can be determined by increasing/decreasing the set stack size until the parallel code executes correctly. For the 4 block test case, the slave thread stack size was set by :-

```
setenv MP_STACK_SIZE 300000000
```

on the Compaq Alpha system or

```
setenv stacksize 300000000
```

on the Sun Microsystems machine. To execute the parallel version, just run the OpenMP executable in the normal way i.e.

```
isaac_omp < kw.1.inp
```

Currently, we only have access to a four processor shared memory system, so the performance measurements are restricted to that at present. The results for the four block test case using OpenMP for intra-block parallelism are shown below. The Compaq Alpha system was again used for these measurements.

Number of Processors	Speedup
2	1.54
3	1.94
4	2.19

The parallel version shows reasonable speedup on 2 processors with some scalability onto 4 processors. Obviously, the hope is that scalability will also be exhibited on larger numbers of processors, but improvements to the OpenMP parallelization based on performance measurements on those larger systems may be needed to realize this (as

some OpenMP overheads will increase and the computation time will decrease making overheads that appear unimportant on 4 processors crucial when scaling to tens of processors).

4 Hybrid Inter-Block Message Passing and Intra-Block OpenMP Parallel ISAAC

The hybrid version was created by simply adding the OpenMP directives from the intra-block version to the Message Passing parallelization inter-block version. This allowed larger numbers of processors on our Compaq Alpha system to be exploited with the four block test case as it consists of 5 connected shared memory systems with 4 processors in each. For these executions, equal numbers of OpenMP threads have been allocated to each group of blocks (e.g. for a 4 processor message passing execution, each of those 4 groups will use four processors with OpenMP so that 16 processors are used in total).

4.1 Compilation, Linking, Execution and Performance of Initial Hybrid Version of ISAAC

Obviously, the same small set of files as in the inter-block only version have been altered for MPI and most have been altered for OpenMP. The cap_utilities.F and CAPLib MPI wrapper files are also required. The compilation and linking require both the MPI library and the OpenMP compiler flag.

The execution of the hybrid parallel version uses mpirun as in the inter-block MPI only version. One complication is that the OpenMP environment variables need to be set on all processors (i.e. the environment variables can be set on hydra1 in the usual way, but they also need to be set on hydra2 etc.). The simplest way to achieve this (as used for the following executions) was to add the setting of the OMP_NUM_THREADS and slave stack size environment variables to the .cshrc file used to setup shells started on remote machines.

Some results for the 4 block test case on the Compaq Alpha system are:

Total Number of Processors	Number of MPI Processes	Number of OpenMP Threads per MPI Process	Speedup
4	2	2	2.68
8	2	4	4.04
16	4	4	5.07

As the test case has only 4 blocks, the number of MPI processes that can be used is very limited (as they exploit inter-block parallelism). This also affects performance as the imbalance in the block size still has a very significant influence. These results do, however, show the potential for good speedup and scalability onto larger numbers of processors for larger test cases.

4.2 Improving Load Balance by OpenMP Processor Allocation

Additional load balancing can be achieved if flexibility in MPI process and thread allocation are possible when, for example, a large number of shared memory processors are available. The load balance of the hybrid inter-block and intra-block parallel ISAAC can be improved by carefully allocating processors. The message passing parallelization distributes the blocks of the mesh across the number of message passing groups. The OpenMP parallelism is then exploited within those groups so assigning different numbers of processors to different groups can promote load balance [7]. The default OpenMP allocation would provide the same number of processors to every message passing group, so a more sophisticated allocation algorithm is needed to improve this load imbalance.

5 Conclusion

The ISAAC code provides sufficient parallelism to potentially scale well onto large numbers of processors. Speedup was achieved for all the parallel versions developed in this work for the multi-block test case, however, other test cases with more blocks and more cells per block should exhibit superior parallel speedup and scale to larger numbers of processors. These other test cases may, however, uncover some further bugs in the parallelization (as not every algorithm in ISAAC is employed by every test case) as well as some other issues that limit scalability.

6 References

1. ParaWise automatic parallelisation environment, PSP Inc. http://www.parallelsp.com.
2. C. Walshaw and M. Cross, ”Parallel Optimisation Algorithms for Multilevel Mesh Partitioning”, Parallel Computing, Vol 26, Issue 12, pp 1635-1660, Nov 2000..
3. Karypis G. and Kumar V. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs" TR95-035, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, 1995.
4 S.P.Johnson, C.S.Ierotheou, M.Cross. Computer Aided Parallelisation of unstructured mesh codes. Proceedings of Parallel and Distributed Processing Techniques and Applications (PDPTA) Conference, Las Vegas, volume 1, CSREA, pp 344-353, 1997
5. P.F.Leggett, S.P.Johnson and M.Cross "CAPLib – A ‘Thin Layer' Message Passing Library to support computational mechanics codes on distributed memory parallel systems", Advances in Engineering Software 32(1): 61-83, 2001.
6. H Jin, M Frumkin, J Yan “Automatic generation of OpenMP directives and its application to computational fluid dynamics codes” International symposium on High Performance Computing, Tokyo, Japan, 2000, LNCS 1940, pp 440-456.
7. Hybrid Parallelization of CFD Applications with Dynamic Thread Balancing. Alexander Spiegel, Dieter an Mey and Christian Bischof, Lecture Notes in Computer Science, 2006, Volume 3732/2006, 433-441.