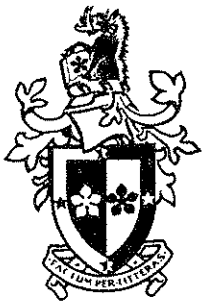

LABORATORY FOR CONCURRENT COMPUTING SYSTEMS

COMPUTER SYSTEMS ENGINEERING
School of Electrical Engineering
Swinburne Institute of Technology
John Street, Hawthorn 3122, Victoria, Australia.

Distinct Element Modelling on a Shared Memory Multiprocessor

Technical Report 31-023

S.K. Tang ++
G.K. Egan ++
M.A. Coulthard **

++ Computer Systems Engineering
School of Electrical Engineering
Swinburne Institute of Technology
John Street
Hawthorn 3122
Australia.

** CSIRO
Division of Geomechanics
Mount Waverley 3149

30/04/91

Abstract

Computational stress analysis is now widely used in geomechanics for back analysis of observed rock mass behaviour around surface and underground excavations, and as a tool for excavation design in mining and civil engineering. Program SDEM is a two-dimensional distinct element code which models the mechanical behavior of systems of simply deformable blocks, for example highly jointed rock. The data structures of SDEM are comprised largely of linked list structures, making effective implementation on vector and array processors difficult. This paper details the analysis and refinement steps used in the initial parallel implementation of SDEM on an Encore multiprocessor system, using Encore Parallel Fortran (EPF) compiler.

Distinct Element Modelling on a Shared Memory Multiprocessor

*S.K. Tang
*G.K. Egan
**M.A. Coulthard

Abstract

Computational stress analysis is now widely used in geomechanics for back analysis of observed rock mass behaviour around surface and underground excavations, and as a tool for excavation design in mining and civil engineering. Program SDEM is a two-dimensional distinct element code which models the mechanical behavior of systems of simply deformable blocks, for example highly jointed rock. The data structures of SDEM are comprised largely of linked list structures, making effective implementation on vector and array processors difficult. This paper details the analysis and refinement steps used in the initial parallel implementation of SDEM on an Encore multiprocessor system, using Encore Parallel Fortran (EPF) compiler.

1. Introduction

Computational stress analysis is now widely used in geomechanics for back analysis of observed rock mass behaviour around surface and underground excavations and as a tool for excavation design in mining and civil engineering. The distinct element (DE) method, which represents a rock mass as a discontinuum of separate blocks, has been shown to be more realistic than finite element (FE) or boundary element (BE) methods in which rock mass is modelled as a continuum, for systems such as subsiding strata over underground coal mine excavations [CD88] [CD91]. However, whereas even 3D FE and BE analyses can now be performed readily on engineering workstations or the more powerful personal computers, the DE method may require an order of magnitude more of computer processing time for analyses of comparable complexity. This has so far prevented the DE method from being applied widely in excavation design in industry.

Most DE codes are based upon an explicit time integration of Newton's second law of motion for each DE, usually involving many thousands of timesteps or solution cycles in a full analysis. The explicit numerical method implies that, within each cycle, calculations for each DE are independent and so could be carried out in parallel. The potential therefore exists for creating much faster DE codes, which would run on moderately priced multi-processor computers, and therefore more accessible machines, through the use of parallel processing.

This report describes several parallelisation techniques on SDEM [CMBLA78] and presents the run time results on an Encore Multimax multiprocessor, using Encore Parallel Fortran (EPF) compiler. It is prepared as a result of the work conducted to reach the final summarised results of the research (parallelisation of SDEM) for the presentation in [ETC90].

---

*S.K. Tang is a research scientist in the Laboratory for Concurrent Computing Systems at the Swinburne Institute of Technology, John Street, Hawthorn 3122, ACSNet: skt@stan.xx.swin.oz.au.

*G.K. Egan is Professor of Computer Systems Engineering and Director of the Laboratory for Concurrent Computing Systems at the Swinburne Institute of Technology, John Street, Hawthorn 3122, ACSNet: gke@stan.xx.swin.oz.au.

**Dr. M.A. Coulthard is Principal Research Scientist at the CSIRO Division of Geomechanics, P.O. Box 54, Mt. Waverley 3149, ACSNet: mac@dogmelb.dog.oz.au.

2. About SDEM

SDEM is a distinct element code which was originally developed by Cundall [CMBLA78], and has been modified subsequently by Lemos [LHC85] and Coulthard [Coultd87]. In DE programs such as SDEM, the individual blocks in the rock mass are represented by a set of distinct elements or blocks. In addition to its rigid body translational and rotational degrees of freedom, each block is allowed 3 modes of deformation internally. Analysis of the mechanics of the system of blocks is based on force-displacement relations describing block interactions, and Newton's second law of motion for the response of each block to the unbalanced forces and moments acting on it.

The problem space is divided into boxes as shown in Figure 1. The corners of all the blocks are mapped into the corresponding boxes. For example, box 6 contains $C_{(3,1)}$ and $C_{(2,1)}$, and box 3 contains $C_{(3,2)}$. The use of boxes is to enable easier search of corners in the detection of contacts. The box entries need to be constantly updated as the blocks move. Reboxing is triggered if a corner crosses an integer boundary, i.e. if the integer part of either x or y coordinates of a corner changes.

During the calculation process, blocks may move and touch different blocks, and hence new contacts may be formed. Therefore, contact data needs to be constantly updated.

All the arrays (data array for blocks, the box array, and the contact array) in SDEM are stored in a single memory partition. They are one-dimensional arrays. The data structure [LC86] is a number of linked lists in which pointers are used to link the various data items. The pointer is a memory location whose content is an address of another memory location. In linked list structure, a value which will not occur is used to indicate the end of the linked list (endmark). In SDEM, a '0' is used as the endmark.

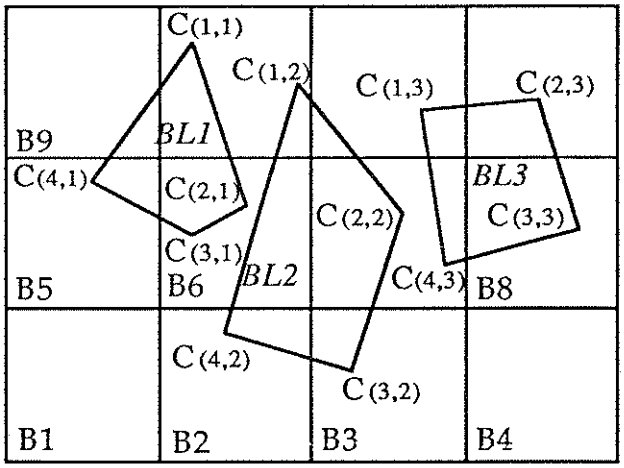

Figure 1: Subdivision of problem space into boxes

2.1 Original SDEM Code

The original computational structure of the SDEM code is shown in Figure 2. The major computation takes place in subroutine CYCLE which in turn calls subroutines UPDAT, MOTION, STRESS and FORD.

Subroutine MOTION determines the motion of a block by using Newton's law of motion. It updates the rigid body velocities of the block using known force sums acting on

centroids. It also updates the internal strain rates from known applied and internal stresses. Then, the coordinates of block corners are updated from the strain rates and rigid body velocities.

Subroutine MOTION calls REBOX if the integer part of either x or y coordinates of a corner changes. Then, subroutine REBOX re-maps the corner into a box.

The internal stresses upon each block are updated in subroutine STRESS using an elastic constitutive law.

Contacts are checked by subroutine UPDAT when cumulative displacements have exceeded a given limit. For each contact, subroutine FORD computes the normal and shear forces developed using constitutive laws. If these forces exceed the specified shear or tensile strength of the rock joints, the contact can slip or open. The forces contributed by the contact are then added to the force sums for centroids of both blocks involved in the contact. Likewise, the applied stress calculated is added to the applied stress sum of both blocks.

The calculation cycle in subroutine CYCLE is performed once per timestep. The iteration within subroutine CYCLE continues until the number of timesteps specified (ncyc) is reached. This may bring the DE system to an equilibrium state or to a state of steady collapse, or it may simply be a convenient point at which the user can assess the status of the computation.

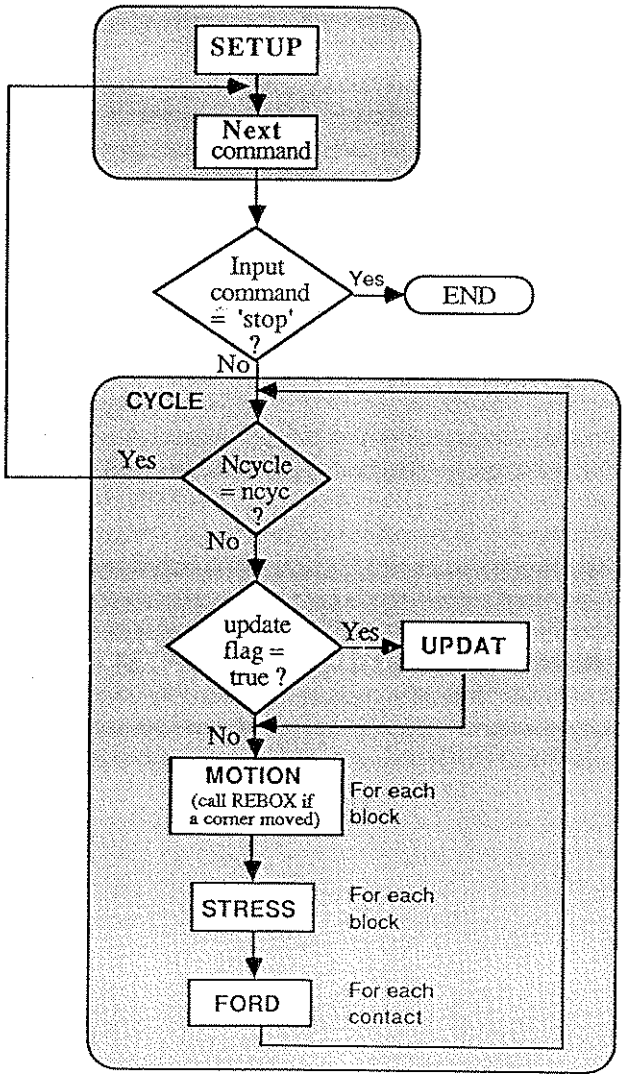

Figure 2: Original Computational Structure of the SDEM model

3. Compiler Used: EPF

The Encore Parallel Fortran compiler (EPF) is the Fortran77 compiler enhanced with parallel programming constructs [ECC88]. Standard Fortran programs can be directed to EPF to produce parallelisation optimizations. EPF determines that some loops can be executed in parallel and converts them into parallel Fortran statements, and produces annotated (.E) output files. Listing files (.lst) can also be generated by specifying *-q list* option. Parallelised loops are indicated in the listing files. Non-parallelised loops and data dependencies which limit loop parallelisation are also shown in the listing files.

The parallel execution is initiated by the **parallel** statement and terminated by the **end parallel** statement. The number of processes that are created in parallel block is determined by the environment variable **EPR_PROCS**. The processes created are then distributed to the available processors; there are four processors on the Encore Multimax used in our experiment.

EPF was used to automatically annotate and compile the original SDEM code in Fortran. Unfortunately, the resulting performance did not improve with the increase in the number of processors used.

3.1 Manual Annotation in EPF

Examination of the .lst listing files produced by EPF apparently indicated that most of the significant program parts were not parallel. It was immediately obvious that EPF did not perform *interprocedural* analysis. It was then decided that the level of concurrency could be improved by manually augmenting the annotated codes in the .E files, with additional parallel directives such as **parallel, doall, barrier** and **critical section.**

The initial parallelisation achieved by EPF was enhanced by focusing upon the non-parallel sections with major attention being devoted to *do* loops as almost all parallel compilers exploit loop concurrency. When there are nested loops, the normal approach is to tackle the outer loop if possible since less overhead is incurred. If parallelisation of a large loop is limited by a small section in the loop, then better performance could be achieved if the small loop could be extracted and executed sequentially outside the original loop. The objective is to achieve the best possible performance but minimise code modification.

The fundamental of all parallelisation efforts lies in identifying and eliminating data dependencies. In this code, data dependencies are usually not able to be removed easily. Minor code restructuring is necessary in order to resolve data dependencies and enable loop parallelisation. However, there are times when a section in a parallel block has to be executed sequentially. In such cases synchronisation mechanisms such as **wait lock, send lock, barrier** are needed so that partial parallelisation in the loop can be achieved. It should be noted, however, that it may not be advantageous to parallelise small loops as the time needed in creating and terminating a parallel block may result in an increased runtime.

4. Parallel Implementation

Parallelisation attempts were concentrated on time critical subroutines which were identified by the UNIX utility, **gprof**. The major loop in the subroutine CYCLE could not be parallelised because information in one timestep (cycle) was passed on to the next timestep. Therefore, the attempt at loop parallelisation was shifted one loop level inward. This inner level consists of loops that call subroutines MOTION, STRESS and FORD. While the profile showed that subroutines MOTION and FORD were runtime significant, the loops which call these subroutines were not parallelised at EPF's automatic pass.

Therefore, the following subsections discuss the non-concurrency factors in the major loops and explain the solutions of these limiting factors to parallelisation. The subsections are in the order of run time contribution.

4.1 Subroutine Ford: Memory Race Contention

The *do* loop which calls subroutine FORD finds the contacts between each block and its immediate neighbours, and for each contact accumulates the applied stress and the sum of the forces of both blocks. In practice, it will normally be quite common for blocks to have direct contact with several others, as illustrated in Figure 3.

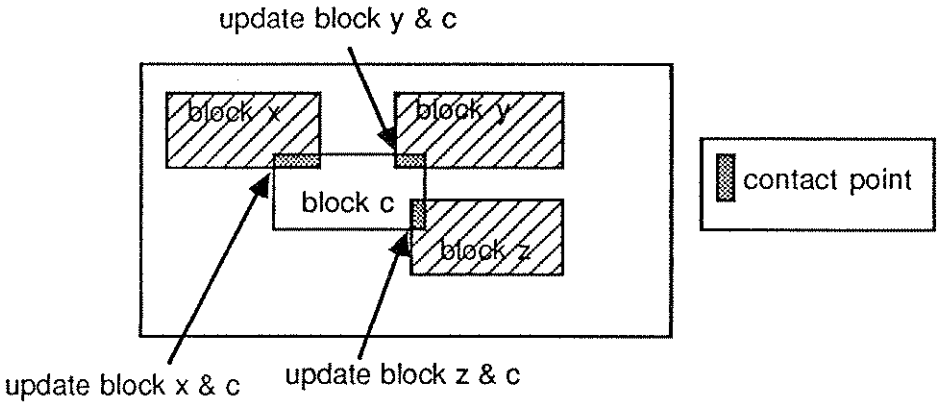

Figure 3: Contacts between adjacent blocks

Therefore, if the *do* loop calling subroutine FORD is executed in parallel, there may be multiple non-deterministic writes to the same memory locations as forces are accumulated. Hence, if the original algorithm is maintained, then update of applied stress and the sum of force in subroutine FORD has to be executed sequentially. A solution to this non-determinacy is to "lock" the data structures of the interacting blocks while accumulating the forces. EPF synchronisation mechanisms **wait lock** and **send lock** were implemented initially as follows:

```
wait lock (lok_nb(nbc)) (lok_nb(nb))
        :
        :
send lock (lok_nb(nbc))
send lock (lok_nb(nb))
```

where all elements in *lok_nb* are of type **lock,** and *nbc* is the block in contact with block *nb*. In theory, the above should lock the two lock variables, lok_nb(nbc) and lok_nb(nb), simultaneously.

However, it was discovered that the EPF compiler did not implement the required double-lock correctly, planting only the first of the two locks. It was therefore necessary to implement the locks using calls to the Encore primitives: **spinlock** and **spinunlock.** In the implementation, we ensured that the code immediately released the first lock if the second lock was not acquired. This is necessary to avoid a deadlock. The locking scheme was implemented as follows:

```
2  call spinlock(lok_nb(nb))
   if (cspinlock(lok_nb(nbc))) goto 1
   call spinunlock(lok_nb(nb))
   goto 2
1  continue
          :
          :
          :
   call spinunlock(lok_nb(nbc))
   call spinunlock(lok_nb(nb))
```

4.2 Loop calling subroutine Motion

The parallelisation in the loop which calls subroutine MOTION is limited by *udmax*, a variable in subroutine MOTION, and the problem of race condition in subroutine REBOX.

4.2.1 Subroutine Motion: Data Dependency

udmax in subroutine MOTION keeps track of the maximum velocity of all blocks by comparing the current maximum velocity with the velocity of the current block. Therefore, subsequent loop iterations cannot proceed until the value of *udmax* is computed in the current loop iteration. Since *udmax* is not used in the loop, the data dependency due to *udmax* can be removed by storing the velocities of each block in two arrays and then the maximum velocity is determined sequentially outside the loop which calls MOTION.

Alternatively, the algorithm can be modified slightly so that the determination of *udmax* will not be necessary. *udmax* is used to check if UPDAT call is necessary. However, the calls to subroutine UPDAT can be made by comparing the displacement of each block with a major displacement. This method was not implemented because it did not conserve the original algorithm.

4.2.2 Subroutine Rebox: Race Condition

Subroutine REBOX re-maps a corner into a box. It determines the correct box which the corner should be in. If the corner is not found in the correct box, then REBOX will seek the corner in the neighbouring boxes, and then relocate the corner to the correct box. An error will result if the corner cannot be found in the determined neighbouring boxes, and this will cause the program to halt. Such rapid movement of a block corner usually implies that the explicit time integration has become unstable.

If subroutine REBOX is included in the parallel section, the problem of race condition will emerge. For example, consider the linked list structure of Box P in Figure 4a; the problem of race condition may occur as illustrated in Figure 4b.

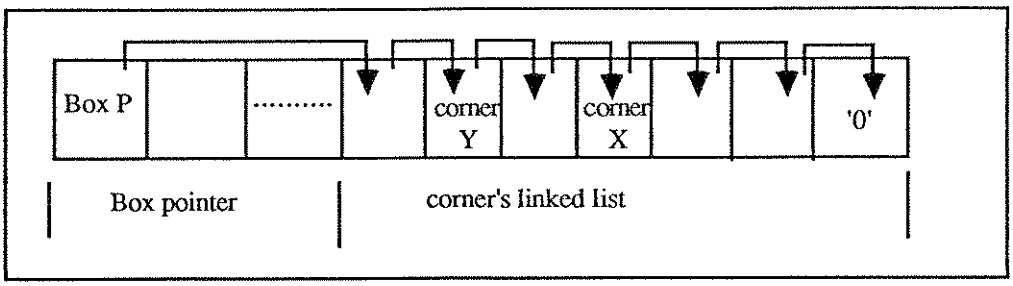

Figure 4a: Linked list structure of Box P

	Processor a	Processor b
	Searching for corner X in Box P, where it should be in.	*Searching for corner Y. Corner Y is not found in Box Q, where it should be in. Now, scanning a neighbouring box (also Box P) for corner Y.*
T1	Obtain the address of corner Y.	
T2		Found corner Y in Box P.
T3		Relocate corner Y to the correct box, which is box Q.
T4	Current corner being searched (corner Y) is not corner X.	
T5	Get next corner. But an endmark is found.	

Time step

Figure 4b: An example of race condition problem in subroutine Rebox

At T3, corner Y has been appended to the corner's linked list in Box Q by *Processor b*. Therefore, when *Processor a* searches the next corner in Box P at T5, it cannot find the correct 'next corner', but the endmark of Box Q list. Corner *x* which exists in Box P was missed in the Box P search. *Processor a* will then search the neighbouring boxes, but surely corner X cannot be found. Eventually, program error will result, and this is undesirable.

So, this code section must be properly synchronised. An attempt was made to include **spinlock** in REBOX, but the performance was unstable. For a system of smaller size, e.g. 105 blocks, performance of SDEM was acceptable. However, for a system of larger size, e.g. 3000 blocks, the performance was significantly degraded. The body within the critical section is considerably long, and this section becomes costly with a larger system size, as the results have shown. Therefore, it is not suitable to include subroutine REBOX in the parallel section.

Call to subroutine REBOX was therefore removed from subroutine MOTION, so that it could be executed separately but sequentially in subroutine CYCLE, and enable the loop calling MOTION to execute in parallel.

4.3 Subroutine Updat: Relocating sequential code section

The subroutine UPDAT is called conditionally, but not frequently. It is important for cases in which sudden collapse of the system occurs, which triggers frequent calls to the subroutine. This subroutine detects the contacts of each block. If there is a new contact detected, storage space for contact data is grasped from an empty list. For those old contacts which no longer represent the present contacts of a block, their data storage space is released to the empty list. In the original implementation, the release of contacts to the empty list was made after the detection of contacts for each successive block, as illustrated in Figure 5.

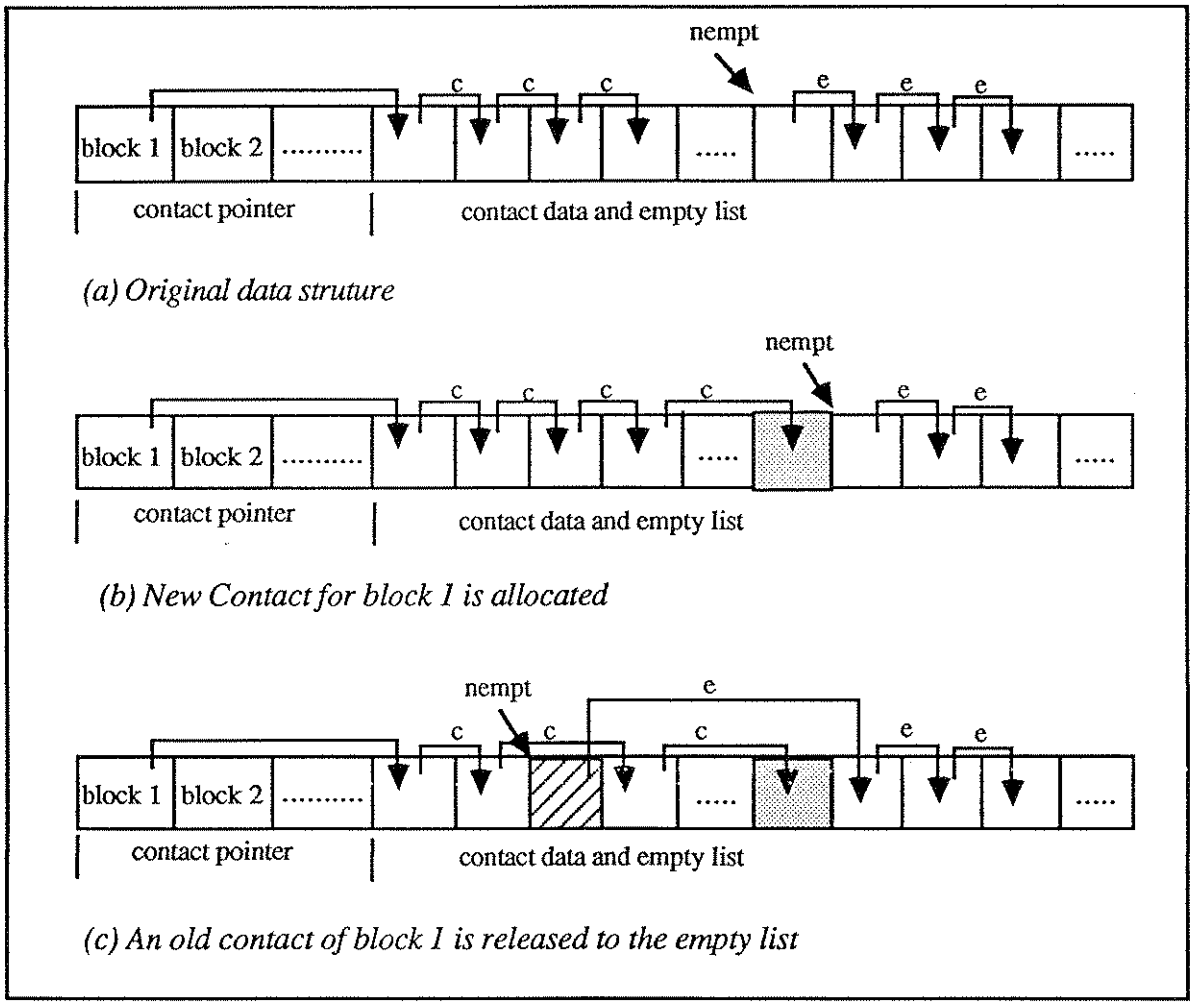

Figure 5: Data structure of block contacts. (c = contact data list, e = empty list)

There are 4 nested loops in subroutine UPDAT, as follows:

```
For each block
   Do each edge
      Do each box in j direction
         Do each box in i direction

         End boxes i scan
      End boxes j scan
   End edges scan
Scan next block
```

The initial parallelisation approach was to attempt to parallelise the outermost loop (For each block). However, as the release of the old contacts of each block was embedded in the loop which scanned the blocks, scanning of blocks was thus forced to be sequential. Less memory space was needed using this method. However, some amendment of the algorithm is necessary if the scanning of blocks is to be parallelised. The old contacts can be released after all detection of contacts has been performed. As a result, in the new implementation, all blocks are scanned in parallel followed by the sequential release of the old contacts.

In the 'Doall blocks' loop, more than one contact may try to acquire space from the empty list. Therefore, a 'lock' is required during the update of *nempt*, the pointer to the start of the empty list, while acquiring new contacts.

5. Further Fine Tuning

The primary objective of parallelising codes is to reduce computation time. Other tuning, such as inlining short subroutines and fusing loops with identical bounds and no inter-loop dependencies have also been performed because EPF does not have inlining options, nor does it fuse loops.

For example, in subroutine CYCLE , the *do* loops which call subroutine MOTION and subroutine STRESS had the same bounds and there were no data dependencies between the two loops, so the loops were fused. Also, the short subroutine STRESS was inlined into subroutine CYCLE .

In EPF, there is no automatic barrier at the **end parallel** statement, in other words, the parent does not wait at the **end parallel** statement for the children to be destroyed. Therefore, it is possible for the parent process to finish its job and go on to execute statements following the parallel region while the child processes are still performing their jobs. Problems may occur if these statements which are executed by the parent process, require the completion of the execution of child processes. Care should therefore be taken to include a **barrier** before **end parallel** directives when performing manual annotation.

In Fortran programs, there are a number of error checking statements such as

```
If (error) then stop
```

If this statement is in a parallel block, the block may never complete. This happens when any child process executes the **stop** statement before it reaches **end parallel**. Then, when the parent reaches the end of the parallel block, it will wait forever for all processes to arrive. This

condition was avoided by using a shared flag which is set to false if an error occurs. It is implemented as follows:

```
parallel
If (error) then cont=.false.
end parallel
if .not. cont then stop
```

6. Performance Analysis

The final version of SDEM was run over several data sets on Encore Multimax multiprocessor. The results, as tabulated in Table 1, indicate that the single processor runtime of the code produced by the EPF compiler is slightly higher than that produced by the f77 compiler. This is the result of the overhead created by the former in process creation and termination.

The speedup and efficiency on different system sizes are tabulated in Table 2. *Snb,cy,up* is a data set with *nb* number of blocks in the system, iterated over *cy* cycles and triggering *up* calls to subroutine UPDAT. Speedup is defined as the ratio of the execution time on a single processor to that with *n* processors sharing the workload, which is

$$S(n) = T1/Tn$$

Efficency is defined as the average utilization of the *n* allocated processors. It is related to S(n) by:

$$E(n) = S(n)/n$$

Ideally, speedup increases with the increase in the number of processors. Unfortunately, along with an increase in speedup there may come a decrease in efficiency. As more processors are devoted to share the execution of the program, the total amount of processor idle time may increase due to factors such as contention for shared resources, the time required to communicate between processors and between processes, and the inability of the compiler to produce an execution code which would keep an arbitrary number of processors usefully busy [EZL89]. As a result, ideal speedup is not achievable although the parallelisation objective is to bring the actual speedup as close to the ideal speedup as possible.

The speedups on different system sizes are presented in Figure 7. The curves show that speedup improves when the body of the parallel code section increases in size, in this case by increasing the size of the systems (number of blocks). As subroutine UPDAT has been parallelised efficiently, therefore, a run on a data set which triggers frequent calls to subroutine UPDAT gives an encouraging speedup. The right hand ends of the curves indicate that when all four processors are used in the experiment, the performance of the code is adversely affected. This is principally due to the context switching (contention for processors by other processes) running on the multiprocessor system at the time of the program run.

No. of Processors	S 105, 20000, 2		S 3000, 100, 2		S 105, 20000, 20000
	f77 (sec)	EPF (sec)	f77 (sec)	EPF (sec)	EPF (sec)
1	2412.7	2595.00	997.2	10231.70	17836.00
2	-	1495.70	-	5774.05	9288.45
3	-	1078.10	-	4037.57	6315.53
4	-	885.80	-	3202.60	4973.17

Table 1: SDEM runtime on Encore Multimax Multiproceccor

No. of Processors	S 105, 20000, 2		S 3000, 100, 2		S 105, 20000, 20000	
	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.74	0.87	1.77	0.86	1.92	0.96
3	2.41	0.80	2.53	0.84	2.82	0.94
4	2.93	0.73	3.19	0.80	3.59	0.90

Table 2 : Speedup and efficiency on Encore Multimax Multiprocessor

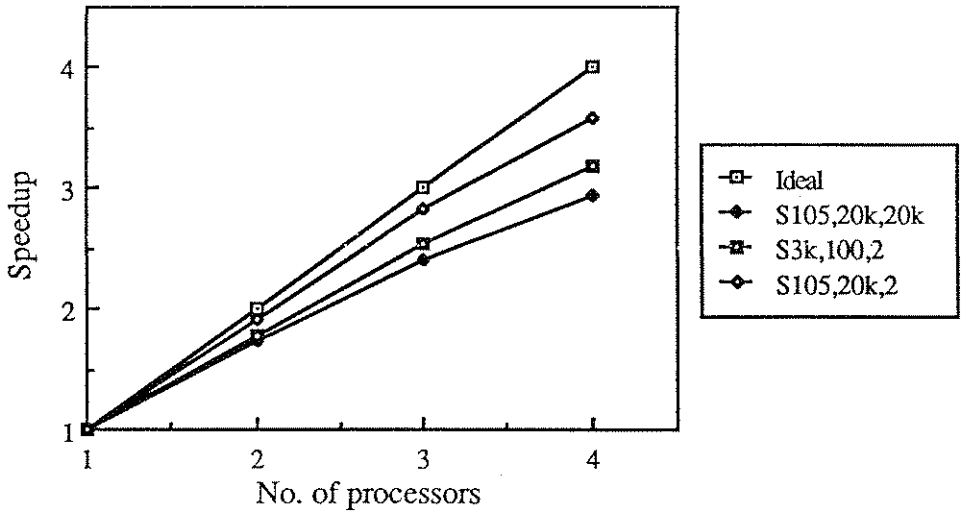

Figure 7: Speedup curve

7. Conclusions

In this report, details of the parallelisation of SDEM using EPF were discussed. The limiting factors in the parallelisation of major loops were explained and ways of circumventing them were described. Although SDEM does not produce effective automatic parallelisation using EPF, the performance of the program can be improved by manually optimizing the output codes after automatic annotation.

The linked list data structures used in the SDEM program and their manipulation presented the major difficulty in parallelising the code. Access to the structure by a number of parallel tasks necessitated a significant number of synchronisation points reducing the obtainable speedup.

The program spends a significant amount of computation time in sequential searches for data along the linked lists. To further improve performance, it may be possible to reorganise the data structures, replacing some linked list structures with direct access matrices. However, this would require major modifications to the code and this would be counter to our aim of minimum code restructuring.

The work presented in this report shows good results for the parallelisation of SDEM. The research is now being extended to a more complex distinct element model, UDEC (Universal Distinct Element Code) [ItascaU90].

Acknowledgements

We wish to thank all members of the Laboratory of Concurrent Computing System, at the Swinburne Institute of Technology, for their contributions to the work presented here. In particular, we would like to thank Pau Chang for his assistance in parallelisation of the model and presentation of this report.

References

[Coultd87] Coulthard M.A. "User manual for program SDEBE - Hybrid Distinct Element-boundary element stress analysis of jointed rock" (preliminary version), CSIRO, Division of Geomechanics, August, 1987.

[CD88] Coulthard M.A., A.J. Dutton, "Numerical modelling of subsidenceinduced by underground coal mining", Proceedings of the 29th U.S. Symposium, University of Minnesota, Minneapolis, 1988.

[CD91] Coulthard M.A., Dutton A.J., "Distinct Element modelling of mining under surface reservoirs - Parametric studies" (under preparation), CSIRO, Division of Geomechanics, 1991.

[CMBLA78] Cundall, P.A., J. Marti, P.J. Beresford, N.C. Last, M.I. Asgian, "Computer modeling of jointed rock masses" Technical Report N-78-4, U.S. Army Engineer Waterways Experiment Station, Vicksburg, Miss., 1978.

[EC90] Egan G.K., Coulthard M.A., "Parallel Processing for the Distinct Element Method of Stress Analysis", Third Australian Supercomputer Conference, Melbourne, December, 1990.

[ECC88] Encore Computer Corporation, *Encore Parallel Fortran manual*, 1988.

[ETC90] Egan G.K. , Tang S.K. and Coulthard M.A. , "Parallelisation of the SDEM Distinct Element Stress Analysis Code", Technical Report 31-021, Laboratory for Concurrent Computing Systems, Swinburne Institute of Technology, Dec 1990.

[EZL89] D. Eager, J. Zahorjian, and E. Lazowska, "Speedup Versus Efficiency in Parallel Systems", IEEE Transaction on computers, pp. 408-423, vol. 38, No. 3, March 1989.

[ItascaU90] Itasca. *UDEC - Universal distinct element code, version ICG1.6; User's manual*, pp3.1-3.48, Itasca Consulting Group, Inc., Minneapolis, 1990.

[LC86] Lorig L.J. and S.K. Choi, *Hybrid Distinct element - boundary element code (HYDEBE) version 1.0 - User's Manual*, Division of Geomechanics, CSIRO, 1986.

[LHC85] Lemos J.V., Hart R.D. and Cundall R.A., "Ageneralized distinct element program for modelling jointed rock mass", Proceeding conference on Fundamentals of Rock Joints, Bjorkliden, 1985.