Laboratory for Concurrent Computing Systems
Technical Report 31-032

Version 1.0 December 1991

Parallelisation of a Distinct Element Stress Analysis Program

S. Tang
Swinburne University of Technology
Australia

Prof G. K. Egan
Swinburne University of Technology
Australia

M. A Coulthard
CSIRO
Division of Geomechanics
Australia

Abstract:

Program SDEM is a distinct element code which models the mechanical behavior of two-dimensional systems of simply deformable blocks, for example highly jointed rock around excavations. The data structures of SDEM consist largely of linked lists, making effective implementation on vector and array processors difficult. This paper details the analysis and refinement steps used in the parallel implementation of SDEM on an Encore multiprocessor system, using the Encore Parallel Fortran (EPF) compiler.

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

Parallelisation of a Distinct Element Stress Analysis Program

Siong K. Tang[1], Gregory K. Egan[2], Michael A. Coulthard[3]

Abstract

Program SDEM is a distinct element code which models the mechanical behavior of two-dimensional systems of simply deformable blocks, for example highly jointed rock around excavations. The data structures of SDEM consist largely of linked lists, making effective implementation on vector and array processors difficult. This paper details the analysis and refinement steps used in the parallel implementation of SDEM on an Encore multiprocessor system, using the Encore Parallel Fortran (EPF) compiler.

1. Introduction

Computational stress analysis is now widely used by civil and mining engineers as a tool for back analysis and design of geomechanical systems. For example, stress analysis can help the engineer understand observed deformations or failure mechanisms induced by excavations, or can be used to predict the interaction of a structure with a soil or rock mass.

Usually, the rock or soil is treated as a continuum, with perhaps a few major structural features or interfaces modelled explicitly; the finite element (FE) or boundary element (BE) methods of stress analysis are used most frequently. If, instead, joint spacing in a rock mass is on a similar scale to the size of an excavation or structure, the rock will tend to behave as a discontinuum. The distinct element (DE) method, which models the mechanical behavior of a system of blocks which can slip, separate and rotate as they interact with each other and with applied loads, may then provide a more realistic approach to stress analysis. For instance, DE models of strata movements over underground coal mining excavations reproduce better the observed dependence of surface subsidence on the depth and width of the excavations than do continuum models [Coulthard & Dutton 1988]. Similarly, the stability of rock slopes may be governed by toppling mechanisms which cannot be represented adequately by continuum models [Cundall et al. 1975].

---

[1]Research scientist and [2]Director, Laboratory for Concurrent Computing Systems, Swinburne Institute of Technology, John Street, Hawthorn 3122, Victoria, Australia.

[3]Principal Research Scientist, CSIRO Division of Geomechanics, P.O. Box 54, Mt. Waverley 3149, Victoria, Australia.

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, particularly if large relative block movements develop. This has so far prevented the DE method from being applied extensively to design problems in geomechanics.

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 mutually 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 multiprocessors, through parallel processing.

The 2D code UDEC [Itasca 1990] is probably the most widely used DE program at present. It includes a range of nonlinear joint models for the interfaces between blocks, and can model quasi-static or dynamic loadings, elastic or plastic deformations within the blocks, and fluid flow in the joints. This many options mean that UDEC is complex and so is not very suitable for experimentation in parallel processing. We have therefore begun our studies with an earlier and simpler DE program, SDEM [Cundall et al. 1978]. As it is based on the same explicit algorithm as UDEC, and uses a similar linked-list data structure, experiences with SDEM are likely to transfer directly to UDEC. This paper describes several parallelisation techniques on SDEM and presents the run time results on an Encore Multimax multiprocessor, using the Encore Parallel FORTRAN (EPF) compiler.

2. About SDEM

SDEM is a simply deformable distinct element model which was originally developed by Cundall [Cundall et al. 1978]. In such DE programs, the individual blocks in a rock mass are represented by a set of distinct elements. 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 regions (boxes). The corners of all the blocks are mapped into boxes to enable easier search of corners in the detection of contacts. During the calculation process, blocks may move and touch different blocks, and hence old contacts may be lost and/or new contacts formed. Therefore, contact data needs to be constantly updated.

The original computational structure of SDEM is shown in Figure 1. Subroutine *Motion* determines the motion of a block by using Newton's second law. It updates the rigid body velocities of the block using the summation of all the known forces acting on centroids, and 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* may call *Rebox* to re-map the corner into a box. The internal stresses within 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.

The calculation cycle in subroutine *Cycle* is performed once per timestep. The iteration within *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 point at which the user can assess the computation.

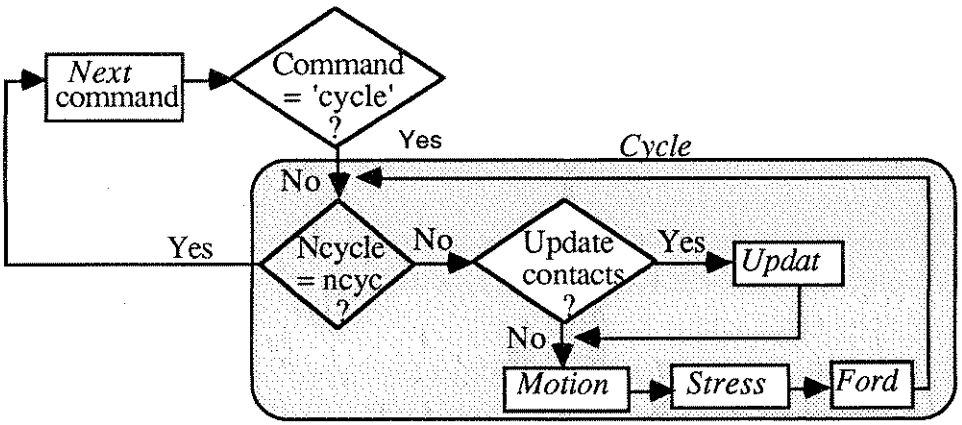

Figure 1: Original Computational Structure of the SDEM model

3. The Approach

The Encore Parallel Fortran compiler (EPF) is used in our parallel implementation of SDEM. It is a FORTRAN77 compiler enhanced with parallel programming constructs [Encore 1988]. 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.

EPF was used to automatically annotate and compile the original SDEM code in FORTRAN. Unfortunately, the resulting performance did not improve with an increase in the number of processors used. Examination of the 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** [Encore 1988].

The initial parallelisation achieved by EPF was enhanced by focusing upon the non-parallel sections with major attention being devoted to seeking concurrency in *do* loops. 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 that section were extracted and executed sequentially outside the original loop. Our objective is to achieve the best possible performance but minimise code modification.

Most of our parallelisation efforts involve identifying and eliminating data dependencies. Minor code restructuring was often 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 efforts were concentrated on time critical subroutines. 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.

The following subsections discuss the non-concurrency factors in block interaction and block motion, and explain ways to circumvent these limiting factors to parallelisation. Other parts in the code such as contacts update have also been parallelised. Contacts update is usually not done often, but it is important for cases in which sudden collapse of the system occurs, triggering frequent contact updates. Other tuning, such as inlining short subroutines and fusing loops with identical bounds and no inter-loop dependencies have also been performed. A more detailed discussion is given by [Tang et al. 1991].

4.1 Block Interaction

For all blocks
  For all contacts
    call Ford

The *do* loop which calls subroutine *Ford* describes block interactions. *Ford* calculates the forces generated at each contact, and adds these to the sum of the forces on both blocks. The calculations at each contact are mutually independent and thus can be executed in parallel. The loop parallelisation attempt is at the block level (for all blocks).

In practice, blocks will normally have direct contact with several others, as illustrated in Figure 2. Therefore, if *For all blocks* loop is executed in parallel, there may be multiple accesses to the same block variable with non-deterministic outcomes, as forces are accumulated. Hence, if the original algorithm is maintained, the sum of force in subroutine *Ford* must be updated by one process at a time.

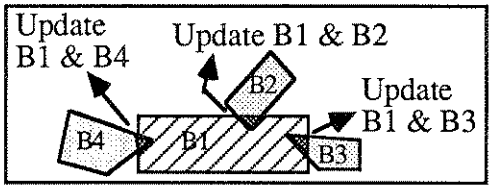

Figure 2: Contacts and forces between adjacent blocks

A solution to this non-determinacy is to "lock" the data structures of the interacting blocks while accumulating the forces. This was initially done by using EPF synchronisation mechanisms **wait lock** and **send lock**. However, it was discovered that the EPF compiler did not implement the required double-lock correctly, planting only the first of the two locks. An alternative mutual exclusion mechanism that will ensure the intended synchronisation is necessary. This was done by using calls to the Encore parallel library routines: **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.

4.2 Block Motion

For all blocks
  call Motion (Rebox if corner moved)

This loop determines motion of all blocks due to unbalanced moments and forces acting on them. During the process of calculating block displacements, if any corner of a block moves, the program also checks if the corner has moved to another box. If it has, then that corner is relocated to the box that it has moved into.

The calculations of block displacements in each timestep are mutually independent, therefore, they can be executed in parallel. However, due to the box reorganisation process and the calculation of the maximum velocity of all blocks, parallelisation in this loop becomes difficult. The following subsections explain the limiting factors and their circumvention.

4.2.1 Maximum velocity

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 eliminated by having a local *udmax* to each process. After all the loop iterations, the maximum velocity is determined from the local maximum *udmax* in a critical section.

Alternatively, the data dependency due to *udmax* can be removed by storing the velocities of each block and then determining the maximum velocity sequentially outside the loop which calls *Motion*. This alternative is not as efficient. The need for *udmax* may be eliminated if the original algorithm can be modified. *udmax* is used to check if update of contacts is necessary. However, update of contacts could be triggered by comparing the displacement of each block with a major displacement. The later method was not implemented because it did not conserve the original algorithm.

4.2.2 Corner Search

Subroutine *Rebox* re-maps a corner into a box. It determines the box which the corner should be in. If the corner is not found in the correct box, *Rebox* will seek the corner in the neighbouring boxes, and then relocate the corner. An error will result if the corner cannot be found in the nearest-neighbour boxes. This indicates a rapid

movement of a block, which usually implies that the explicit time integration has become unstable.

If the box reorganisation process is included in the parallel section, failure of corner search may occur. As illustrated in Figures 3a and 3b, the problem is the race to search for corner X so as to ensure a complete search. At timestep T3, corner Y has been appended to the linked list of corners in Box Q by *Process b*. Therefore, when *Process a* searches the next corner in Box P at T5, it cannot find the correct 'next corner', but the endmark of Box Q list. As a result, the neighbouring boxes, will fail to find corner X.

Thus, this code section must be properly synchronised. An attempt was made to include **spinlock** in the box reorganisation process, but the resulting execution performance was unacceptable especially for a system which triggers frequent box reorganisation processes. The reason was probably that the critical section is relatively long and thus the cost of implementing spinlock became apparent. Therefore, it is much better to extract the box reorganisation process from *For all blocks* loop, and execute it separately and sequentially, and enable the loop calling *Motion* to execute in parallel. Physically, it is reasonable to determine the motion of all blocks first, then rebox any corner that moves.

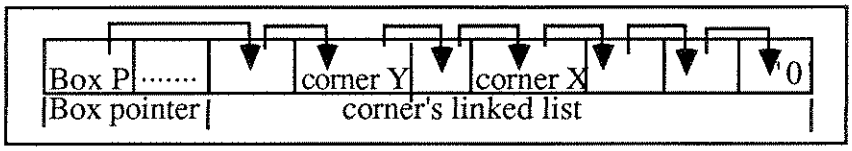

Figure 3a: Linked list of corners in Box P

Time step	Process a	Process b
	Searching for corner X in Box P, where it should be.	*Searching for corner Y. Corner Y is not found in Box Q, where it should be. 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, endmark found!	

Figure 3b: An example of corner search failure in subroutine Rebox

5. Performance Analysis

The parallelised SDEM code was run on an Encore Multimax multiprocessor consisting of six XPC processors. The runtimes and speedups of various data sets are tabulated in Table 1. *Snb,cy,up* is a data set with *nb* blocks in the system, iterated

over *cy* cycles and triggering *up* contacts update. The wallclock times 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.

No. of Processors	S105,2000,2			S3000,100,2		
	f77 (sec)	EPF (sec)	Speedup	f77 (sec)	EPF (sec)	Speedup
1	2080.6	2292.00	1.00	889.50	998.00	1.00
2	-	1432.00	1.60	-	568.00	1.76
3	-	1034.00	2.22	-	396.00	2.52
4	-	807.00	2.84	-	330.00	3.02
5	-	696.00	3.29	-	281.00	3.55
6	-	633.00	3.62	-	256.00	3.90

Table 1: SDEM runtime and speedup on Encore Multimax Multiproceccor

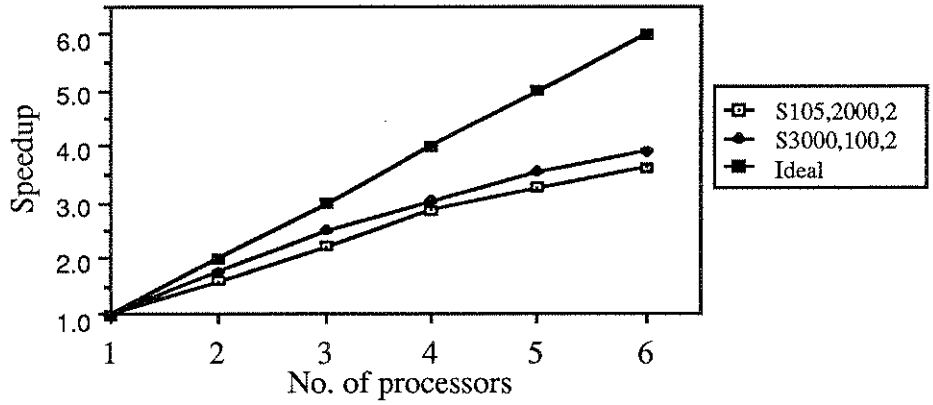

Figure 4: Speedup curve

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. Ideally, speedup increases linearly with the number of processors. Unfortunately, along with an increase in speedup there may come a decrease in efficiency, the average utilization of the *n* allocated processors. 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 [Eager et al. 1989]. As a result, ideal speedup is not achievable; the parallelisation objective is to bring the actual speedup as close as possible to the ideal .

The speedups on different system sizes are presented in Figure 4. The curves show that speedup improves when the body of the parallel code section increases in size, in this case by increasing the number of blocks in the system. The right hand ends of the curves indicate that when all six 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 from other jobs) on the multiprocessor system at the time of the SDEM runs.

6. Conclusion

In this paper, 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 EPF does not produce effective automatic parallelisation of SDEM, the performance of the program can be improved by manually optimizing the output codes after automatic annotation.

Manipulation of the linked list data structures used in SDEM 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 lists with direct access matrices. However, this would require major modifications to the code and would be counter to our aim of minimum code restructuring.

Acknowledgements

We wish to thank all members of the Laboratory for Concurrent Computing Systems, at the Swinburne Institute of Technology, in particular Pau Chang, for their contributions to the work presented here.

References

Coulthard M.A. and A.J. Dutton, "Numerical modelling of subsidence induced by underground coal mining", Proc. 29th U.S. Symp., pp 529-536, Minneapolis, 1988.

Cundall P., M. Voegele and C. Fairhurst, "Computerized design of rock slopes using interactive graphics for the input and output of geometrical data", Proc. 16th U.S. Rock Mech. Symp., pp 5-14, Minneapolis, 1975.

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

Eager D., J. Zahorjian and E. Lazowska, "Speedup versus efficiency in parallel systems", IEEE Transaction on Computers, pp 408-423, vol. 38, No. 3, March 1989.

Encore Computer Corporation, *Encore Parallel Fortran manual*, 1988.

Itasca. *UDEC - Universal distinct element code, version ICG1.6; User's manual*, Itasca Consulting Group, Inc., Minneapolis, 1990.

Tang S.K., G.K. Egan and M.A. Coulthard, "Distinct element modelling on a shared memory multiprocessor", Proc. 4th Aust. Supercomputing Conf. pp 143-155 , 1991.