LABORATORY FOR CONCURRENT COMPUTING SYSTEMS

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

Submitted to the International Conference on Supercomputing, Manchester, 1994.

Parallelisation of the SDEM Distinct Element Stress Analysis Code on the KSR-1

Technical Report 31-048

*G.K. Egan * G.D. Riley † J.M. Bull †*

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

†Centre for Novel Computing
University of Manchester
Manchester M139PL
England.

Version 1.0 Original Document 20/1/1994

Key Words: *parallel processing, seismic modelling, distinct element method*

Abstract:

The SDEM code models systems of interacting blocks of rock using the distinct element (DE) method, which represents these systems as discontinuums with each block acting under Newton's Laws of motion. The data structures associated with the DE method are comprised largely of linked lists make the task of obtaining performance gains through vectorisation difficult. Typical systems, however, are comprised of thousands of blocks and there is the potential of performing calculations associated with groups of blocks in parallel.

This paper details the analysis and program refinement steps used in implementing a parallel version of SDEM on the Kendal Square Research KSR-1 distributed memory multiprocessor. Performance gains from a simple translation of the original Cray FORTRAN code are poor, but satisfactory performance is obtained, with minimum changes to the code, by addressing specific sources of overhead. The refinement steps focus on reducing lock costs, ensuring data locality and improving load balance.

Parallelisation of the SDEM Distinct Element Stress Analysis Code on the KSR-1

G.K. Egan * G.D. Riley † J.M. Bull ‡

Abstract

The SDEM code models systems of interacting blocks of rock using the distinct element (DE) method, which represents these systems as discontinuums with each block acting under Newton's laws of motion. The data structures associated with the DE method make the task of obtaining performance gains through vectorisation difficult. Typical systems, however, contain thousands of blocks and there is the potential to perform calculations associated with groups of blocks in parallel.

This paper details the analysis and program refinement steps used in implementing a parallel version of SDEM on the Kendall Square Research KSR-1 distributed memory multiprocessor. Performance gains from a simple translation of the original Cray FORTRAN code are poor, but satisfactory performance is obtained, with minimum changes to the code, by addressing specific sources of overhead. The refinement steps focus on reducing lock costs, ensuring data locality and improving load balance.

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, has been shown to be more realistic than finite element (FE) or boundary element (BE) continuum methods for modelling systems such as subsiding strata over underground coal mine excavations. However, whereas even 3D FE and BE analyses can now be performed readily on engineering work stations or the more powerful personal computers, the DE method generally requires orders of magnitude more 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.

The DE method represents these systems as discontinuums with each block acting under Newton's laws of motion. The data structures associated with most implementations of the DE method consist largely of linked lists, making the task of obtaining performance gains through vectorisation difficult. As the systems are comprised of thousands of blocks there is however the potential of performing block interaction calculations in parallel.

We describe the parallelisation of SDEM, a representative DE stress analysis code for the analysis of two dimensional systems of interacting, simply deformable polygonal DEs ([3],[9]). Our starting point is a version of the code written for a Cray Research YMP multiprocessor. In this study of the parallelisation of SDEM we seek to keep the number of changes necessary to the original program small. This precludes major re-organisation of the data structures.

2 The Distinct Element Method

The DE method of stress analysis was introduced [2] to deal with problems in rock mechanics which could not be treated adequately by the conventional continuum methods. The earliest DE programs (for example program RBM in [3]) assumed that the blocks were rigid, so that all deformations within the

*G.K. Egan is Professor of Computer Systems Engineering and Director of the Laboratory for Concurrent Computing Systems at the Swinburne University of Technology, email: gke@swin.oz.au.

†G.D. Riley is the Research Co-ordinator for the Centre for Novel Computing, University of Manchester, email:griley@cs.man.ac.uk

‡J.M. Bull is a Research Associate in the Centre for Novel Computing, University of Manchester, email:markb@cs.man.ac.uk

system took place at the block interfaces. A second program described in [3], SDEM, allowed modelling of three simple modes of deformation of each block—two compressive and one shear mode. The DE programs which are most widely used at present are UDEC [6] and 3DEC [7]; the blocks in each of these may be modelled as fully deformable via internal finite difference zoning.

2.1 Theoretical basis

Most DE programs are 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 normal forces developed at a point of contact between blocks are calculated from the notional overlap of those blocks and the specified normal stiffness of the inter-block joints. Tensile normal forces are usually not permitted; there is no restraint placed upon the opening of a contact between blocks. Shear interactions are load-path dependent, so incremental shear forces are calculated from the increments in shear displacement, in terms of the shear stiffness of the joints. The maximum shear force is usually limited by a Mohr-Coulomb or similar strength criterion.

The equations for the normal forces F_n and shear forces F_s on a block are as follows—

$$\Delta F_n = K_n \Delta U_n$$

and

$$\Delta F_s = K_s \Delta U_s$$

where ΔU_n and ΔU_s are the incremental normal and shear displacements respectively, and K_n and K_s are the normal and shear stiffnesses respectively (see Figure 1).

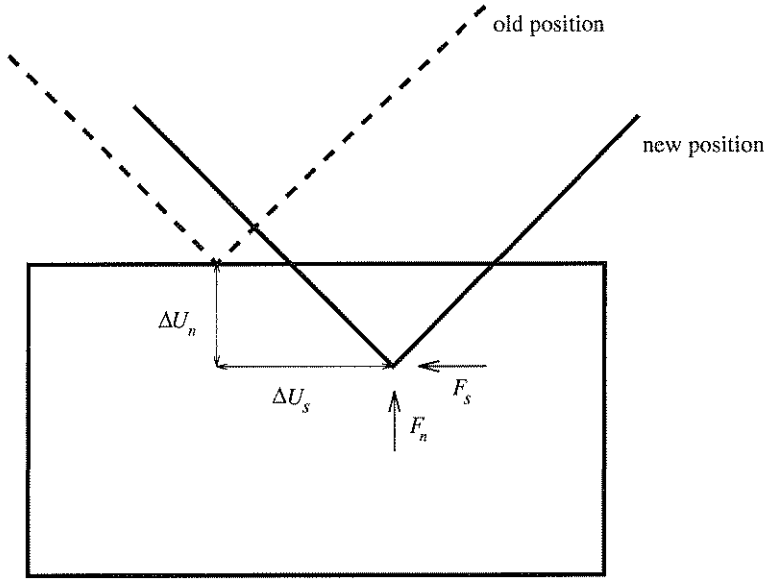

Figure 1: Block interactions

The motion of each block under the action of gravity, external loadings and the forces arising from contact with other blocks is determined from Newton's second law. A damping mechanism is also included in the model to account for dissipation of vibrational energy in the system. The equations of motion are integrated with respect to time using a central difference scheme to yield velocities and then integrated again to yield displacements—

$$\begin{aligned} \dot{u}_i(t + \Delta t/2) &= \dot{u}_i(t - \Delta t/2) + (F_i(t)/m + g_i)\Delta t \\ u_i(t + \Delta t) &= u_i(t) + \dot{u}_i(t + \Delta t/2)\Delta t \end{aligned}$$

where $i = 1, 2$ correspond to the x and y directions respectively. The u_i are the components of displacement of the block centroid, the F_i are the components of non-gravitational forces acting on the block, the g_i are the components of gravitational acceleration and m is the mass of the particular block. The

velocity-dependent damping terms have been omitted here for simplicity, but the same form of equations hold even when damping is included.

Block velocities and displacements are expressed explicitly in terms of values at the previous time step and so each may be calculated independently. The calculated displacements are used to update the geometry of the system and to determine new block interaction forces. These, in turn, are used in the next time step.

This explicit time integration scheme is only conditionally stable. Physically, the time step must be small enough so that information cannot pass beyond neighbouring blocks in one step, thus justifying the assumption of the independence of the integrated equations of motion.

3 SDEM

SDEM's main computational cycle is contained within the CYCLE subroutine (Figure 3). The new positions of blocks are computed using the current forces acting on them (MOTION); from these positions, the stresses (STRESS) and new forces induced by blocks on their neighbours (FORD) are determined. These steps are repeated until the system stabilises. Contact lists, or lists of the other blocks a block is touching, are maintained to exploit locality; these data structures are the principal source of difficulties for vectorising compilers. If the velocity of any block is greater than some threshold then the contact lists of all blocks are updated (UPDAT); this deals with sudden events or collapses in the system occurring in a particular time step. REBOX is used to maintain bounding boxes for possible post processing using the Boundary Element Method. These bounding boxes are not utilised in this study.

```
while not stable do {
  if update required then
    call UPDAT
  do each block
    call MOTION{   compute motion
                   if current block velocity > max velocity then
                       max velocity = current block velocity
                   do each block corner
                       if corner outside box then
                          call REBOX
    }
    do each block
      call STRESS
    do each block
      do each contact
        call FORD
}
```

Figure 2: SDEM's main computational cycle

A system comprising 6496 blocks arranged in a 'brick wall' with fixed boundary blocks was used for this study. After a number of initial 'settlement' cycles, eight blocks at the base are removed to form a tunnel and a further, usually very large, number of cycles is performed. The expected result is for a section of the roof of the tunnel to collapse possibly with some buckling prior to the collapse (see Figure 3). For a system of this size there would normally be in the order of 10,000 settlement cycles followed by more than 200,000 further cycles. With this many iterations any data structure initialisation cost may be ignored. To keep the run times within reasonable bounds we have used 500+500 cycles for our experiments.

4 The Kendall Square Research KSR-1

The KSR-1 is a Virtual Shared Memory multiprocessor. The machine consists of processor-memory pairs (*cells*) arranged in a hierarchy of *search groups*. The virtual memory is implemented on the physically distributed memories by a combination of operating system software and hardware support through

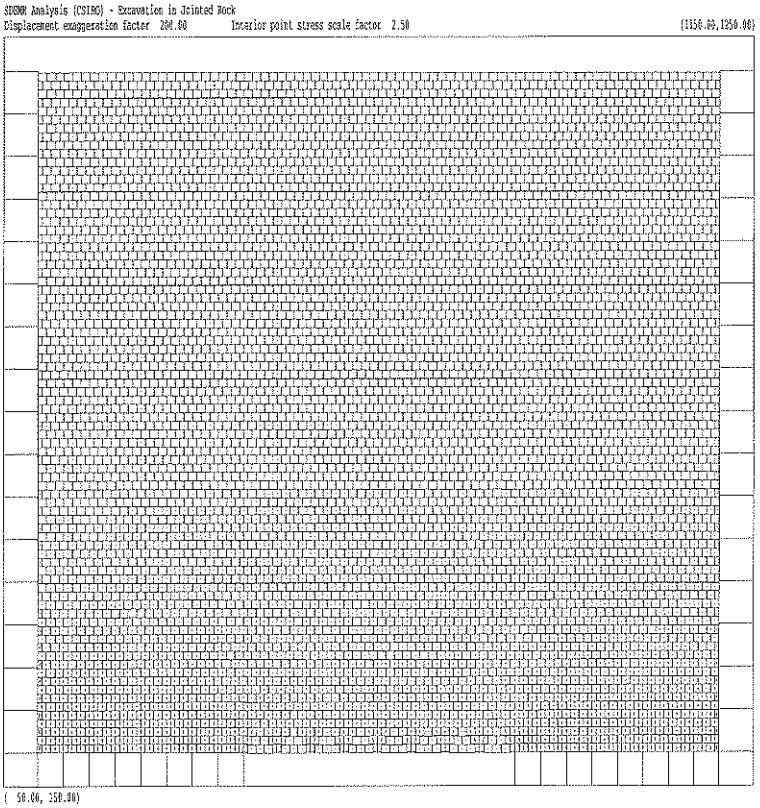

Figure 3: Block system after 80,000 cycles showing stress vectors

the KSR ALLCACHE search engine. The OS manages page migration and fault handling in units of 16 kbytes. The ALLCACHE engine manages movement of 128 byte subpages within the system. Movement of subpages is therefore cheap compared to the movement of pages. The implementation described in this work is for the 32 processor, single search group, KSR-1 installed at Manchester University[1].

Each cell is a 20 MHz, super-scalar, RISC chip with a peak 64-bit floating point performance of 40 Mflop/s (achieved with a multiply-add instruction) and 32 Mbyte of memory. Two instructions may be issued per cycle; the instruction pair consists of one load/store or i/o instruction and one floating point or integer instruction. The cells are connected by a uni-directional slotted ring network with a bandwidth of 1 Gbyte/s.

The ALLCACHE memory system is a directory-based system which supports full cache coherency in hardware. Data movement is request driven; a memory read operation which cannot be satisfied by a cell's own memory generates a request which traverses the ring and returns a copy of the data item to the requesting cell. A memory write request which cannot be satisfied by a cell's own memory results in that cell obtaining exclusive ownership of the data item—the data item moves to the requesting cell. In the process, as the request traverses the memory system, all other copies of the data item are invalidated, thus maintaining cache coherence through an invalidate-on-write policy.

[1] Running KSR OS version R1.1.4.1, Oct 20 1993 and compiler version 1.0, May 11 1993.

The machine has a Unix-compatible distributed operating system—the Mach-based OSF/1—allowing multi-user operation.

The programming model supported is primarily that of program directives placed in the user code (FORTRAN 77, and to some extent, 'C'). The directives may be placed manually or automatically (by a pre-processor, KAP). A run-time support system, PRESTO, and underlying Posix-based threads model support the user directives. The run-time system and threads are also directly accessible through a standard library interface.

4.1 Synchronisation primitives—locks and barriers

Synchronisation is implemented at the subpage level in the KSR-1. A subpage may be requested by a thread in *atomic* state. No other thread may then take the same subpage in atomic state until it has been released. Get subpage (`gsp`) and release subpage (`rsp`) operations are supported in the instruction set and are available to the user through FORTRAN intrinsic calls. Other synchronisation primitives, such as barriers and pthread level mutex locks, are implemented using atomic subpages. Synchronisation is therefore implemented *through* the memory system. In contrast with systems which implement synchronisation via special buses, such as Sequent machines, locks on the KSR-1 are expensive—a request for a lock may have to traverse the network. Synchronisation on the KSR-1 is, however, scalable with the memory system, and therefore with the number of processors.

4.2 KSR-1 memory latencies

The KSR-1 processor has a level 1 cache, known as the subcache. The subcache is 0.5 Mbyte in size, split equally between instructions and data. The cache line of the subcache is 64 bytes (half a subpage).

There is a 2 cycle pipeline from the subcache to registers. A request satisfied within the main cache of a cell results in the transfer of half a subpage to the subcache with a latency of 18 cycles (0.9 μs). A request satisfied remotely from the main cache of another cell (one of the 32 in the single search group of the Manchester machine) results in the transfer of a whole subpage with a latency of around 150 clock cycles (7.5 μs). A request for data not currently cached in any cell's memory results in a traditional, high latency, page fault to disk.

Both taking a subpage into atomic state (`gsp`) and releasing a subpage (`rsp`) cost approximately 25 cycles, as both instructions must access the logic at the interface to the cell interconnect which is contained in other chips on the processor board.

4.3 Memory system behaviour—alignment and padding

In order for a thread to access data on a subpage, the page in which the subpage resides must be present in the cache of the processor on which the thread executes. If the page is not present, a page miss occurs and the operating system and ALLCACHE system combine to make the page present. If a new page causes an old page in the cache to be displaced, the old page is moved to the cache of another cell if possible. If no room can be found for the page in any cache, the page is displaced to disk. Moving a page to the cache of another cell is much cheaper than paging to disk.

Performance of applications in virtual memory systems can suffer from the phenomenon of *false sharing*; if two threads, running on different cells, request separate data items which reside on the same subpage, that subpage may continually thrash back and forth between cells. Most VM systems have to contend with false sharing at the OS page level, which is typically several kbytes in size. On the KSR-1 the unit of movement around the system is the relatively small 128 byte subpage. At this size, ensuring that data structures accessed by several threads do not cause thrashing can be achieved simply by ensuring that the structures are *padded* out to a subpage boundary and that they are aligned so as to begin on a subpage boundary. This is most simply achieved through suitable declaration of data structures; for example padding the inner dimension of multi-dimensional arrays.

An example of the effect of false sharing occurs when providing private arrays for threads; each thread will update its own copy of the array. If the shared array is small, then simply expanding the array by the number of threads can result in false sharing, manifested in poor load balance and impaired performance. If each local array is padded to a subpage boundary, false sharing is eliminated.

5 Parallelisation

The analysis and initial parallelisation of SDEM is made difficult by the complicated data structures. The major data structure is contained in a single integer/real vector with a number of consecutive elements of the vector constituting a record describing a block. Some of the elements point in turn to other records of contact blocks and all fields are referred to numerically rather than symbolically. It is not surprising that annotators and vectorisers have some difficulty extracting performance gains. The KAP parallel annotator [8], for example, was unable to identify any significant parallel regions in the main computational cycle.

Analysis of CYCLE [4][5] has shown that it may be parallelised as follows—

- The loop over blocks which contains calls to the MOTION and REBOX subroutines can be split into two loops by calling MOTION for each block and recording whether each block needs to be re-assigned to another bounding box. Instead of tracking the maximum velocity of any block, a shared flag is introduced which is set if the velocity of any block exceeds a certain threshold. This flag is then used to determine whether a call to UPDAT is necessary at the start of the next cycle. A second loop then calls REBOX on all blocks which require the reboxing procedure. The iterations of both resulting loops can be executed in parallel.

- The iterations of the loop containing calls to STRESS can be executed in parallel without making any modifications.

- Recall that the FORD subroutine computes the accumulated forces acting on a block due to all other contacting blocks. The iterations of the loop containing calls to the FORD subroutine may also be executed in parallel, provided that the data structures for both interacting blocks are locked while accumulating the forces acting on each block. This is necessary to ensure correct execution, as otherwise it is possible for there to be more than one processor updating the variables in which the forces are being accumulated, and some updates may not occur.

  Locking means that no processor may execute a section of code guarded by a lock until it obtains the lock. Processors contend for a lock until they are successful in obtaining it. They then execute the critical section of code, releasing the lock on completion. The programmer must ensure that updates to a shared data structure which should only be accessed by one processor at a time occur within critical sections.

  Some care is required when it is necessary to lock two block records as it is possible for deadlock to occur. The solution we use is to immediately release the first lock if the second lock cannot be obtained.

- If major dislocation or collapse in the simulated system occurs (indicated by the flag tracked within MOTION) subroutine UPDAT is called to update the lists of contacting blocks. This results in release of list entries where blocks are no longer contacting, and addition of new list entries where new contacts are formed. The routine consists of a loop over blocks, whose iterations may be executed in parallel provided that access to the free list pointer (a variable 'pointing' to the next empty list item) is protected by a lock when creating new and discarding old contact records. Note that in a full run of SDEM, the frequency of calls to UPDAT will be sufficiently low that it contributes only a very small amount to the overall execution time.

Given the above observations it remains only to decide on a partitioning strategy for the parallel loops. All the parallel loops are over the number of blocks in the system. The strategy adopted is a simple partitioning of the iteration space of each loop into slices consisting of an equal number of blocks, so that each thread operates on a horizontal slice of the domain. Although a two-dimensional domain decomposition should be better in terms of the amount of communication required, the nature of the main data structure would make this difficult, requiring significant changes to the source code which we have precluded in this study.

The parallelisation strategy described above as implemented in the code written for the Cray Research machine formed the starting point for our experiments on the KSR-1. The remainder of this Section describes the refinement stages required to obtain an efficient implementation of the code on the KSR-1.

5.1 Explicit annotation

Since there was no benefit gained from using the KAP annotator, we proceeded immediately to explicit parallel annotation. The first step in this process was to directly translate the Cray Research directives to KSR directives. This was accomplished without difficulty as the notations used are very similar. The KSR FORTRAN compiler assumes that all variables not declared to be private are shared across the parallel loop. There is consequently the potential for accidental sharing of variables on the KSR. The Cray Research compiler requires all variables to be specified as either shared or private.

5.2 Block locking using pthread mutual exclusions

Initially the Cray Research calls to *lockon* and *lockoff* were replaced with calls to KSR *pthread_mutex_lock* and *pthread_mutex_unlock* [8]. The Cray Research locks [1] are implemented as hard instruction level polling of the lock variable. This can lead to runaway processor time under the Cray Research operating systems [5]. Although pthread locks permit comprehensive statistics gathering, including event time stamping, they were far too expensive in terms of time for this implementation, which requires one lock per block.

5.3 Block locking using get and release subpage

Calls to `gspnwt` (get subpage no wait) and `rsp` (release subpage) resolve to the equivalent of 'test and set' and 'clear' instructions on conventional architectures. The performance implications of these instructions are somewhat different from conventional shared memory multiprocessors as has already been explained in Section 4.1. The calls to `gspnwt` and `rsp` require the address of the subpage to be locked or unlocked. To implement this required the simple declaration of an array of subpages with the first word of each subpage being accessed as the lock. We refer to this version of the code as Version 1. The execution time and simulation performance (in iterations or cycles per second) for this and subsequent versions are presented in Section 6.

5.4 Improved block locking using get and release subpage

The cost of acquiring a lock can be 150 clock cycles if the requested subpage is not cached exclusively in the requesting cells memory, and is around 25 cycles if the subpage is cached exclusively. Releasing the lock also takes around 25 cycles. During this time the Cell Execution Unit issuing the instruction stalls. As this number of cycles is significant when compared to the useful code being executed it is obviously important to reduce the number of locks.

Inspection of the loop involving the call to FORD shows that for each block the interactions of all contacting blocks are considered before considering the next block. The approach taken on the Cray Research systems was to lock the two interacting blocks only when the forces were accumulated on the two blocks currently being considered; both locks were then released. The modified locking scheme locks the current block record under consideration and then considers, locking in turn, all contacting blocks before releasing it. This approach is against conventional wisdom which says that the critical region should be as small as possible. For this particular code, lock conflicts are so infrequent that it pays to increase the size of the critical section, thus reducing the number of locks. We refer to our new code as Version 2.

5.5 Data structure alignment

Monitoring of the KSR cache behaviour[2] indicated that the subpage cache miss rate was high. This was thought to be due to false sharing of subpages in the main data structure. To reduce this effect we forced all data structure allocations to subpage boundaries, and padded block records to occupy an exact number of subpages. We were able to accomplish this without re-organising the data-structures themselves, thus adhering to our minimum change guideline. The subpage cache hit rates were improved, but not nearly as much as anticipated. This formed Version 3 of the code.

[2]Each KSR cell has an event monitor chip. Monitoring information is available to the user through a library interface.

5.6 Conditional updating of shared variables

Further analysis and experiment showed that the high subpage cache miss rate was largely due to the updating of certain shared variables such as the flag which indicates that no blocks are moving, and counters of the number of lock conflicts. Although these updates are hardly ever executed, the optimising compiler issues an exclusive load (which invalidates all other copies) for these variables in advance of the conditional test which determines whether the update occurs or not. Thus the number of subpage cache misses is just the same as if the updates always take place. There are several possible work-arounds for this problem, such as placing the offending statements in a subroutine, and compiling these subroutines without optimisation. The resulting code, with this change, is Version 4.

5.7 Locality of contact records

Having significantly reduced the cache subpage miss rate, we still found that there were an unreasonably large number of page misses. The block contact data structure is assembled by the first call to UPDAT. This assembly is performed in parallel, and as a result, the block contacts accessed by one thread are scattered throughout the structure. This means that each thread has to have space allocated in its cache for every page of the structure, even though it only accesses a few of the subpages on each page. The effect of this is for there to be a high frequency of displacement of pages from the cache of each cell, as each cache can hold only a limited number of pages at a time. Our solution was to run the first call to UPDAT sequentially, which causes all the contacts associated with a loop chunk to be contiguous in memory, thus reducing the number of pages accessed by each thread. Having done this, we found that aligning and padding the block and contact records to subpage boundaries was no longer beneficial. The amount of false sharing of subpages is now negligible, and padding merely increases the cache miss rate, as redundant array elements are being communicated between cells. This resulted in Version 5 of the code.

5.8 Load imbalance

Analysis of the performance of individual subroutines showed that there was considerable computational load imbalance in the loops containing calls to MOTION, REBOX and FORD. The SETUP subroutine, which generates the data structures from a parameterised description of the block system, assigns the fixed boundary blocks to the early block indices. As they are fixed, MOTION needs perform no computation for them, leading to load imbalance in the loop with calls to MOTION. Since in each cycle only a few blocks require reboxing, the loop with calls to REBOX is, of course, poorly load balanced. The large fixed boundary blocks have more contacts than internal blocks, and so the loop containing calls to FORD is also unbalanced. Furthermore, during the course of the computation contacts may be formed and broken as the system of blocks settles, and substantial variations in the number of contacts associated with blocks assigned to KSR cells can occur. As the major part of the computation is associated with FORD—in computing forces due to the contacts—it is not possible to load balance both MOTION and FORD without significantly affecting data locality and thus increasing the number of non-local data accesses. As FORD costs approximately three times as much as MOTION for this problem, we decided to load balance FORD. The solution used is to monitor the time taken by each thread in FORD and to adjust the loop slice boundaries to compensate. We find that after a few tens of cycles, FORD becomes well load balanced. We now have the final version, Version 6, of the code.

6 Results

To test the various versions of the parallel code developed in Section 5 we run each version for 500 settlement cycles, followed by the removal of eight blocks, and a further 500 cycles. Since calls to UPDAT are infrequent in a full-length run, we record the elapsed time for the 1000 cycles only, ignoring initialisation and calls to UPDAT. An optimised sequential version of the code run on one cell requires 787.4 seconds for these 1000 cycles. Table 6 shows the elapsed times in seconds on varying numbers of cells for each version of the parallel code. The results presented are the best times out of three runs.

Figure 4 shows the performance in cycles per second for each version of the code. The Naive Ideal performance is computed by simply multiplying the performance of the optimised sequential code (1.2 cycles per second) by the number of cells. The Realistic Ideal performance is computed by taking into

	Number of cells								
Version	1	2	4	8	12	16	20	24	28
1	986	733	502	348	294	266	249	245	245
2	913	691	481	342	287	259	242	244	243
3	959	587	392	277	230	226	210	217	236
4	955	593	308	180	135	111	106	94.8	110
5	920	486	257	143	106	89.9	81.7	77.4	76.6
6	913	478	250	135	97.5	80.4	68.9	63.4	59.0

Table 1: Elapsed times in seconds for 500+500 cycles of SDEMH (6496 blocks)

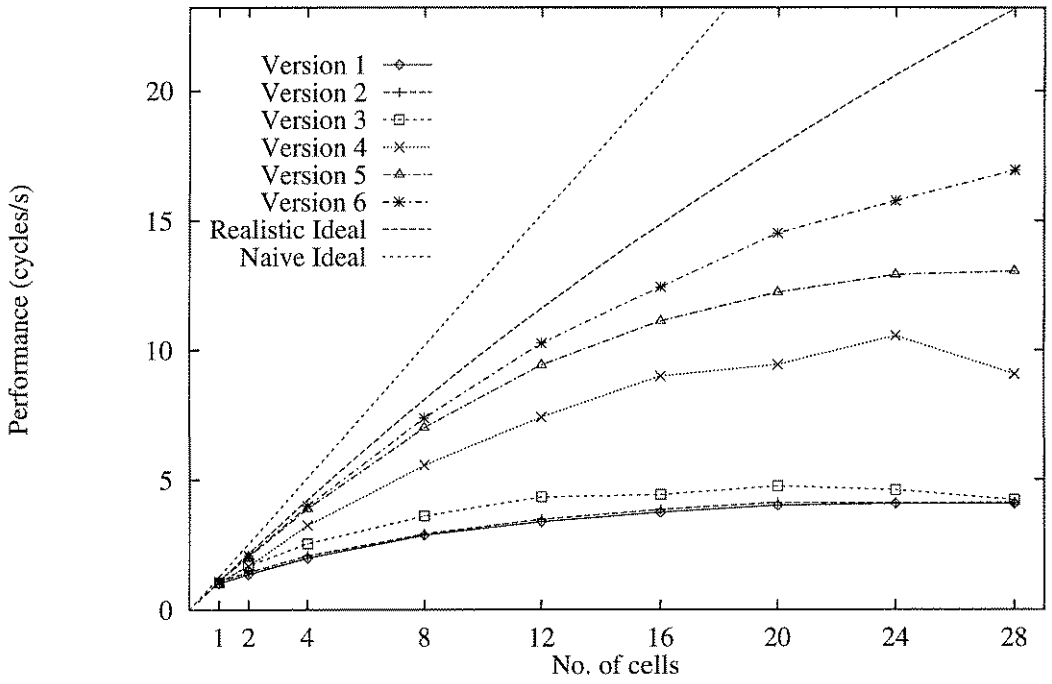

Figure 4: Performance of 500+500 cycles of SDEMH (6496 blocks)

account certain minimum overheads which are known to be present in the final version of the code. These are

- The cost of implementing locks. We assume that each `gspnwt` and each `rsp` require 25 cycles, and that no lock conflicts occur.
- The cost of remote accesses required to access contacting blocks which may 'belong' to another thread. We assume that the cost of remote accesses is 150 cycles per subpage.
- The cost of barrier synchronisation. We assume that each parallel loop incurs an overhead of 0.75ms.[3]
- The cost of unparallelised sections within CYCLE. This is very small—around 0.06% of the sequential execution time.

We see that the first three versions of the code give very poor performance, never exceeding 5 cycles per second. Subsequent improvements in the parallelisation strategy which resulted in Versions 4–6

[3] This figure is based on experimental data.

each give a substantial improvement in performance. The final version achieves 17 cycles per second on 28 cells. Most of the difference between the measured performance of Version 6 and the realistic ideal performance is due to the remaining load imbalance, particularly in MOTION and REBOX.

7 Further enhancements

The results presented in Section 6 show that while we have achieved reasonable performance, we are still falling some way short of ideal performance. In this section we consider what else might be done to further improve performance, by examining the sources of the overheads identified. The overheads we observe are due to lock operations, remote data accesses, load imbalance and barrier synchronisation.

7.1 Two-dimensional data partitioning

Our algorithm for partitioning the computational load consists of slicing the iteration space of the parallel loops. Without altering the original scheme for the numbering of the blocks, this corresponds (except for the large fixed boundary blocks) to partitioning the domain in horizontal slices. This means that the number of remote accesses required for data associated with blocks is essentially constant as more processors are used. Indeed once there are less than 3 rows of blocks per slice (for the problem size we consider here this occurs when using more than 26 cells) the number of remote accesses starts to *increase* with the number of processors). A better solution might be to decompose the domain two-dimensionally, so that each cell operates on a rectangular region of the domain. This would ensure that the number of remote accesses required would decrease as the number of processors was increased. To do this would involve a major rewriting of the initialisation phase of the code, to change the way in which blocks are numbered. It would also mean that the dynamic load balancing scheme described in Section 5.8 would no longer be applicable.

7.2 Removal of reboxing

The reboxing procedure takes no part in the actual computations being performed; it merely serves to maintain an interface to a boundary element code. In the sequential code REBOX only accounts for around 2% of the execution time for a cycle. REBOX does not parallelise very well, however, as the number of blocks requiring reboxing on each cycle is often in single figures. Parallelisation of REBOX inevitably results in significant load imbalance, and on 28 cells it accounts for over 5% of the execution time. It therefore would seem worthwhile to rewrite the code so that the bounding box information is computed only once at the end of the run, rather than being maintained on every cycle as at present.

7.3 Local force accumulation

The computed overhead due to the `gspnwt` and `rsp` instructions is significant. Monitoring of the code shows that the number of lock conflicts is minimal, but as the number of KSR cells used increases lock conflicts become more likely. An alternative scheme is to use a local data structure to accumulate the forces on the block due to the blocks 'assigned' to a particular KSR cell and then perform a reduction of the global forces acting on the block outside the FORD subroutine; there is no need to lock the block records and the global reduction may be performed in parallel. To minimise the changes to data structures the local data structure has to be almost the same size as the major block data structure. As a consequence the number of cache subpage misses is significantly increased and the improvement in performance is not as good as might be expected. Some preliminary tests indicate that the overheads associated with this method are in fact higher than those associated with locking. However, further investigation is required.

7.4 Asynchronous version

In view of the very low incidence of lock conflicts, it may be possible to run the code asynchronously (that is, with no locks) and still obtain physically reasonable results. Removing the locks might result in the occasional update of the forces on a block being missed. In a real system this may occur as the corner of a block crumbles or fractures. Thus as long as the missed updates occur infrequently and at random, the solution may be just as physically meaningful as that given by the code with locks.

8 Conclusions

We have presented the procedures required to produce an efficient implementation of SDEM on the Kendall Square Research KSR-1. Our initial version of the code, obtained by taking a code run on the Cray Research YMP and translating Cray FORTRAN to KSR FORTRAN, resulted in rather poor performance. Subsequent revisions of the code, however, gave significant improvements. These revisions were concentrated in three main areas—

- Reducing the cost of using locks. On the KSR-1 locks are implemented via the memory system, which although ensuring scalability with number of processors, implies that locking updates to fine-grained data structures can be costly.
- Ensuring maximum locality of data. The KSR-1 has a physically distributed memory, and therefore it is critical to reduce the number of remote data accesses made during the execution of the program.
- Reducing load imbalance. The somewhat irregular nature of the physical problem can result in significant load imbalance unless steps are taken to address this. We have implemented a dynamic load balancing scheme which we believe will be effective for a wide range of systems of interacting blocks, and which will react if a major collapse occurs in the course of modelling a system.

For the size of system we consider, a full run consisting of 200,000 cycles could be executed in less than 3.5 hours on 28 processors. We have also suggested some further enhancements which may possibly lead to even better performance. Some of these, however, would be in conflict with our aim of making minimal changes to the source code. We also believe that many of the techniques developed here would be applicable to the fully deformable distinct element codes UDEC and 3DEC.

9 Acknowledgements

The authors thank Dr. M.A. Coulthard of the CSIRO Division of Geomechanics for his support of this research and the provision of the SDEM code studied. This research was supported by a grant of time on the KSR-1 from the Centre for Novel Computing.

References

[1] Cray Research, *CF77 Compiling System Volume 4: Parallel Processing Guide*, Cray Research, Incorporated, SG-3074 5.0, 1991.

[2] Cundall, P.A., *A computer model for simulating progressive large-scale movements in blocky rock systems*, Proceedings ISRM Symposium on Rock Fracture, Nancy, Vol. 1, Paper II-8. 1971.

[3] Cundall, P.A. et al., *Computer modelling of jointed rock masses*, Technical Report N-78-4, U.S. Army Engineer Waterways Experiment Station, Vicksburg, Mississippi, 1978.

[4] Egan G.K. and M.A. Coulthard, *Parallel processing for the Distinct Element Method of Stress Analysis*, 3rd Australian Supercomputing Conference, Melbourne, December, 1990.

[5] Egan, G.K., *Parallelisation of the SDEM Distinct Element Stress Analysis Code*, Applications of Supercomputing in Engineering III, Computational Mechanics Publications, Elsevier Applied Science, London New York, pp 297–312, 1993.

[6] Itasca, *UDEC - Universal distinct element code*, Version ICG1.6; User's manual. Itasca Consulting Group, Incorporated, Minneapolis, 1990.

[7] Itasca *3DEC - 3-D distinct element code*, Version 1.2; User's Manual, Itasca Consulting Group, Incorporated, Minneapolis, 1990.

[8] K.S.R., *KSR Fortran Programming*, Kendall Square Research, 170 Tracer Lane, Waltham, MA, 15 Feb. 1992.

[9] Lemos, J.V. *A hybrid distinct element - boundary element computational model for the half-plane*, M.S. Thesis, Department of Civil and Mineral Engineering, University of Minnesota, Minneapolis, 1983.