A Generic Strategy for Dynamic Load Balancing of Distributed Memory Parallel Computational Mechanics using Unstructured Meshes

A. Arulananthan, S.P. Johnson, K. McManus, C. Walshaw, M. Cross
a.arulananthan@gre.ac.uk
Centre for Numerical Modelling and Process Analysis
University of Greenwich, London, SE18 6PF, UK

Abstract: A large class of computational problems are characterised by frequent synchronisation, and computational requirements which change as a function of time. When such a problem is solved on a message passing multiprocessor machine [5], the combination of these characteristics leads to system performance which deteriorate in time. As the communication performance of parallel hardware steadily improves so load balance becomes a dominant factor in obtaining high parallel efficiency. Performance can be improved with periodic redistribution of computational load; however, redistribution can sometimes be very costly. We study the issue of deciding when to invoke a global load re-balancing mechanism. Such a decision policy must effectively weigh the costs of remapping against the performance benefits, and should be general enough to apply automatically to a wide range of computations. This paper discusses a *generic* strategy for Dynamic Load Balancing (DLB) in unstructured mesh computational mechanics applications. The strategy is intended to handle varying levels of load changes throughout the run. The major issues involved in a generic dynamic load balancing scheme will be investigated together with techniques to automate the implementation of a dynamic load balancing mechanism within the Computer Aided Parallelisation Tools (CAPTools) environment, which is a semi-automatic tool for parallelisation of mesh based FORTRAN codes [2].

1. Introduction

Many computational problems assume a discrete model of a physical system, and calculate a set of values for every domain point in the model. These values are often functions of time, so that it is intuitive to think of the computation as marching through time. When such a problem is mapped onto message passing multiprocessor machine or shared memory machine with fast local memories, regions of the model domain are assigned to each processor. The running behaviour of such a system is often characterised as a sequence of steps, or iterations. During a step, a processor computes the appropriate values for its domain points. At the step's end, it communicates any newly computed results required by other processors. Finally, it waits for other processors to complete their computation step and send the data required for the computation of the next step [1,5].

A static mesh partition is unlikely to provide a good load balance when solving dynamic non-linear problems in parallel using an unstructured mesh. Prediction of the load associated with each mesh entity (grid point, face, element, etc.) is not simple. Even if we do predict the work load accurately, the computational work associated with each portion of a problem's subdomain may change over the course of solving the problem. This can occur when the behaviour of the modeled physical system changes with time. For example, during the course of solving a problem, more work may be required to resolve features of the emerging solution. Load variations due to differences, for example, in element shape or perhaps grid point degree may be anticipated but some effects such as changes in the discretisation or the physics associated with each entity may not be known until the code has run for some time. Cache effects and inhomogeneous architectures further complicate prediction. Adaptive meshing involving refinement and de-refinement will inevitably suffer from significant load imbalance. Even with a fixed mesh, multi physical simulations such as the modelling of phase changes such as melting or solidification [1], can lead to significant imbalance. Here the application of flow calculations are required only for the liquid portion of the problem and similarly the stress calculations are only required for the solid portion. Such load imbalance may only be determined at run time.

Because of the synchronisation between steps, the system execution time during a step is effectively determined by the execution time of the slowest, or most heavily loaded processor. We can then expect system performance to deteriorate in time, as the changing resource demand causes some processor to become proportionally overloaded. One way of dealing with this problem is to periodically redistribute, or remap load among processors. There are two fundamentally different approaches to remapping. The decentralised load balancing approach is usually studied in the context of a queuing network [9]. Decentralised balancing is the natural approach when jobs are independent, and a global view of balancing would not yield substancially better load distributions. A large class of computations are not well characterised by job arrival model, and it may be advantageous to take a global, or centralised perspective when balancing [8,7]. A centralised mapping mechanism can exploit full knowledge of the computation and its behaviour. Furthermore, dependencies between different parts of a computation can be complex, making it difficult to dynamically move small pieces of the computation from processor in a decentralised way. Global mapping is natural in a computational environment where other decisions are already made globally, e.g., convergence checking in an iterative numerical method. The presented algorithm monitors the work load at run time in order to predict the transfer of load between processors that will minimise the overall runtime of the computation.

2. Constraints of Dynamic Load Balancing

The load balancing algorithm and consequent data movement must be very fast in comparison to the overall run-time. Dynamic load balancing is not merely a pre-processing step such as static partitioning since the algorithm and the consequent load migration may be performed frequently during the run time. Load rebalancing will only provide a performance gain if the time to rebalance is less than the decrease in run time consequent from rebalancing the code, so it is important to relate the overhead cost of remapping with the expected performance gain. In addition, computational mechanics codes are very demanding of memory and so the memory requirement for DLB must be small in comparison to the memory used by the application. Implementation of DLB algorithms can be highly application specific, decisions on granularity of monitoring and mesh migration are difficult to determine both statically and generically. The details of load evolution, of the remapping mechanism, and of various overhead costs are system and computation dependent, complicating the devising of a generic DLB algorithm.

3. Issues of Dynamic Load Balancing

A migration policy determines "which" (identification), "when" (decision) and "where" (location) processes should be migrated. In order to study general properties of remapping decision policies, it is necessary to *model* the behaviour of interest, and evaluate the performance of decision policies on those models. A number of major issues have been identified in the implementation of a dynamic load balancing scheme [8,7]:

3.1. Timing

The appropriate part(s) of the application code to time can vary widely between different codes. For example, one code may necessitate the timing of the top loop level such as the time step loop (Level 1 in Figure 1), but another code may require timing of the lower loop levels such as a loop within a conjugate gradient solver (Level 3 in Figure 1). For the DLB algorithm to be generic it will be more appropriate for the algorithm to automatically determine which levels of loop to time. Since the intention is to automatically generate DLB calls as a phase in the automatic parallelisation of unstructured mesh codes within the Computer Aided Parallelisation Tools (CAPTools) environment [4,3], the identification of all sensible loop levels and related code generation can resonnably be performed with dynamic loop level selection.

The presented model must not initiate the rebalancing mechanism too frequently, this will waste the time on moving the data around rather than the actual run. But, the load between the processors of the parallel machine can become badly balanced if the rebalancing mechanism is initiated too infrequently and hence the performance will deteriorate in time. So, it is important to correctly determine the criteria that will be appropriate to re-distribute the data. Three inter-linking factors are involved: the level of imbalance in each section of the code; the run time for each code section; and the time required for calculating and performing a redistribution. These factors must be measured dynamically from the code and used to predict if the reduction in imbalance (idle time) will compensate for the cost of the DLB algorithm.

3.2. Load migration

The key aspects of load migration are to determine how much of each subdomain to move, which entities to move and where to move them. These issues are addressed in the related work on the JOSTLE mesh partitioning tool [10,6] which describes how an existing mesh partition can be modified by a completely parallel algorithm. The load imbalance information is indicated by a weighted graph that is passed to JOSTLE, which will attempt to balance these weights in the resultant partition.

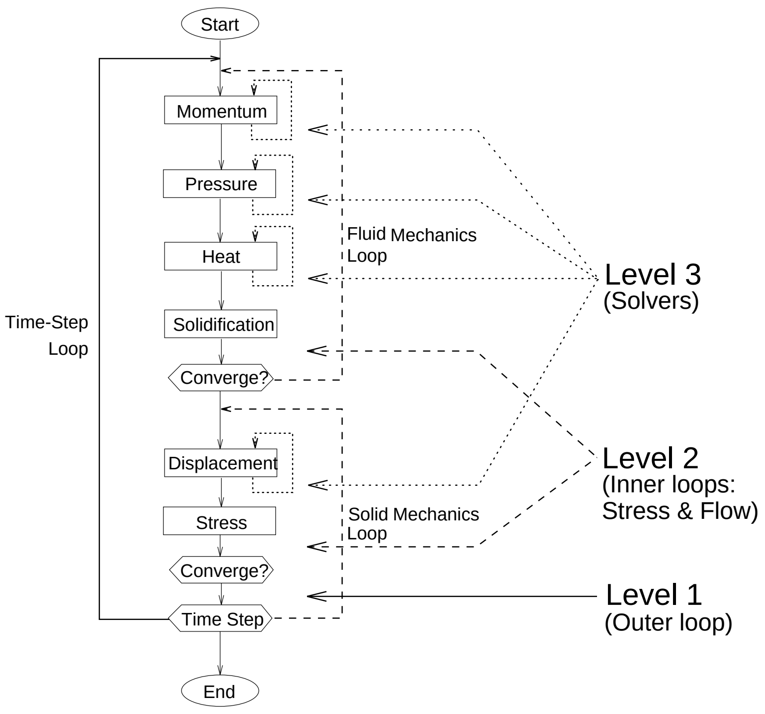

Figure 1. Different levels of loops that can be found in a typical CFD code.

4. Model to predict when to re-partition

We propose a simple rebalancing decision heuristic which assumes that the rate of change of imbalance between processors is always linear, that the rebalancing time is constant (rebalancing time includes JOSTLE time, renumbering time [5] and the re-distribution time), and that JOSTLE removes all imbalance. Response to large changes in load have the potential to over compensate and lead to instability in the algorithm. Stability of the algorithm also becomes an issue when the communication latency is high compared to the speed of variation of the load, unnecessary migrations should be avoided. It is imperative to avoid oscillation or cycling of the load across the processors and so a damping coefficient is incorporated into the algorithm to relax the movement of entities. The damping coefficient is calculated in response to the rate of change of work and consequently limits the speed of response to load changes.

The presented algorithm forms a cost function, t_{cost}, that models the time for re-distribution and the predicted application code run time in relation to the rate of increase of imbalance (see equation 1). The model is based on an instance in time and predicts what would happen under the model assumptions. It uses the number of iterations (n) between DLB redistributions to predict runtime. t_{cost} explicitly embodies two of the costs a remapping policy must manage: delay cost of rebalancing and idle time costs of not rebalancing.

$$t_{cost} = \int_0^t \left(\frac{n.i \times B}{2} + \frac{J}{n.i} \right) dt = \frac{Bn.i.t}{2} + \frac{J.t}{n.i} \qquad (1)$$

Where i is the time taken for each iteration, B is the rate of increase of imbalance across

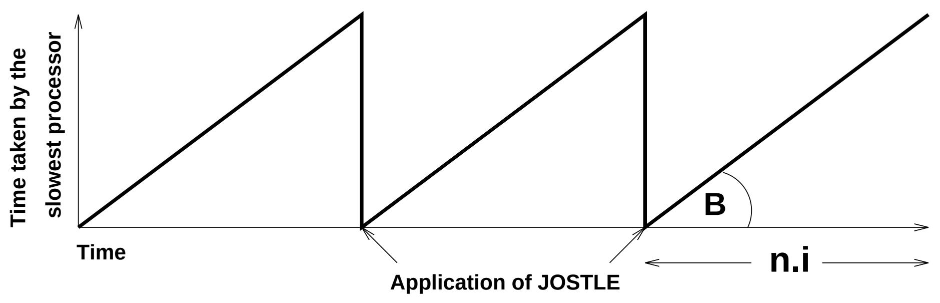

Figure 2. Model to predict when to re-partition

the processors and J is the JOSTLE time. We then minimise the re-balancing time with respect to n (see equation 2). The model (equation 3) predicts an optimal value for n that minimises the runtime prediction function. Redistribution will be performed n iteration after the previous redistribution.

$$\frac{dt}{dn} = \frac{Bit}{2} - \frac{Jt}{n^2 i} = 0 \tag{2}$$

$$n = \sqrt{\frac{2J}{Bi^2}} \tag{3}$$

5. Load Migration Algorithm

When it is predicted that it is worth performing a re-balancing operation, the following algorithm is used to move and update the data:

• Time code section on each processor. We can either time CPU-time or idle-time.
• Calculate weight on each processor using monitored time, e.g. high CPU-time implies high weight or high idle-time implies low weight.
• Call JOSTLE with 'weighted' graph (or mesh). JOSTLE then balances these weights in the resultant partition, returning a new processor ownership (primary partition) array.
• Update secondary partition (e.g. nodal) using the primary partition (e.g. element).
• Identify and update moved entities data (i.e. an elements temperature, U-velocity, V-velocity etc). The entities are locally numbered, which reduce the memory size by removing the need for any globally sized arrays to be stored, this maximises scalability of memory. The communication is implemented using two routines, one to construct a movement (within a processor) and communication (between processors) set listing where entities are to be sent/received to/from which processor, and the other to perform the communication using that communication set for a particular variable:

CALL COMMSET	takes JOSTLE send-sets as input and calculates movement and communication sets, listing entity numbers and the processor numbers to where they are moved to.
CALL SWAPCORE	uses a movement and communication set calculated in COMMSET and moves/communicates input array data.

- Recalculate the new overlaps (or halo elements) and update communication set.

So a set of subroutine calls in PHYSICA looks like this:

```
C     Call the dynamic JOSTLE with the weighted element graph.
      CALL DJOSTLE(graph,numELEM,ELEMset)
C     Update the node partition using the element partition.
      CALL SUBSIDUARY(ELEMset,NODEset)
C     Calculate the movement and communication sets for elements and nodes.
      CALL COMMSET(ELEMset,commELEM)
      CALL COMMSET(NODEset,commNODE)
C     Communicate and update elements data using the elements comm set.
      CALL SWAPCORE(UELE,commELEM)
      CALL SWAPCORE(VELE,commELEM)
      CALL SWAPCORE(WELE,commELEM)
C     Communicate and update nodes data using the nodes comm set.
      CALL SWAPCORE(XNODE,commNODE)
      CALL SWAPCORE(YNODE,commNODE)
      CALL SWAPCORE(ZNODE,commNODE)
```

6. Results

The DLB algorithm has been implemented in a simple test code which simulates dynamically changing workloads using an unstructured mesh. The implementation involved the development of a set of utilities to migrate mesh entities in parallel while only ever requiring access to local data. Results from the test code in Figure 3 show that the DLB overhead is sufficiently small to provide a worthwhile performance improvement. It significantly improves the performance achieved by static balancing. The algorithm was tested on a network of 5 SUN workstations each of which have differing performance. Initially, all 5 processors were given the same number of elements, but as Figure 4 shows, the DLB algorithm migrated the elements from the slow workstations to the fast ones to balance the load across the processors as the code ran. As you can see from Figure 4, the DLB model decides to do the first remapping after 2 iterations because the work-load is very badly balanced. But as the balance improves across the processors, the re-balancing interval (n) increases.

The DLB algorithm is currently being incorporated into the multi-physics modelling environment PHYSICA [1] where all aspects of the algorithm will be investigated in a realistic test cases. In particular, the criteria for the selection of optimal granularity (i.e. the loop level that maximises performance) will be identified.

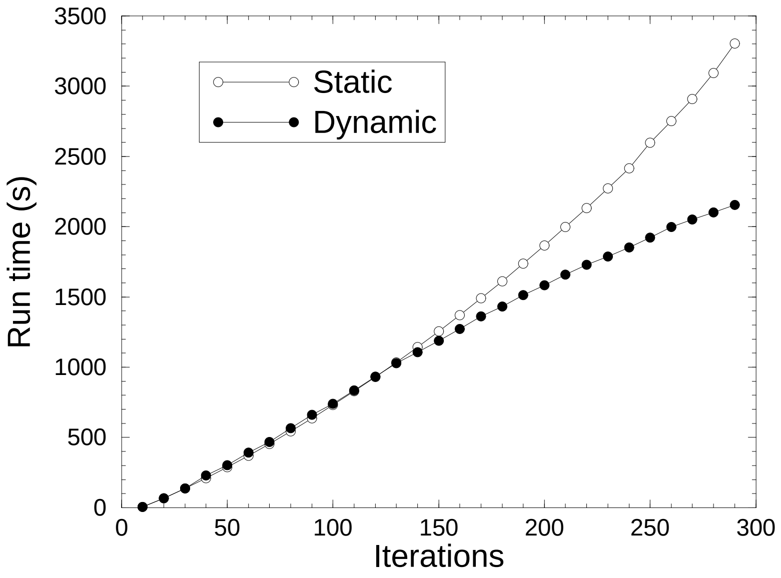

Figure 3. Run time of solidification case on 4 processors on a Transtech Paramid.

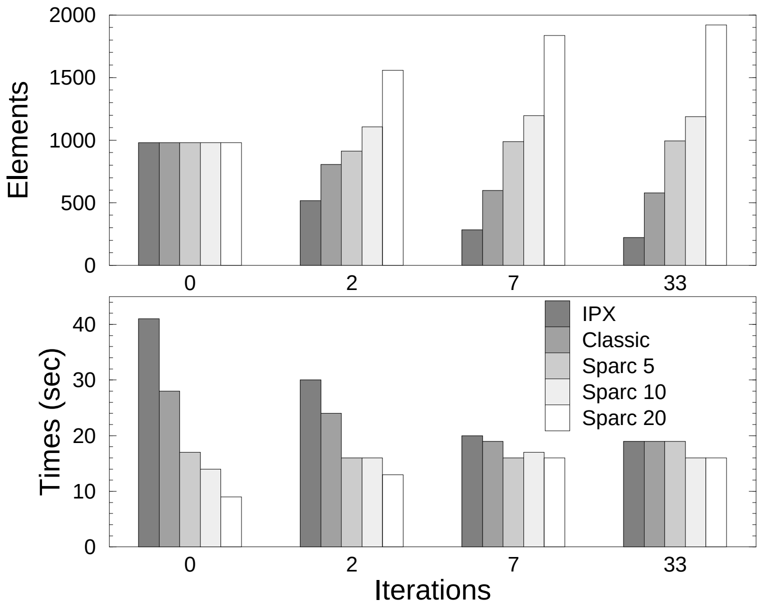

Figure 4. Workstation times and their loads.

7. Conclusions and Related Work

The presented dynamic load balancing scheme has been developed and tested on small test cases that successfully addresses the important issues relating to the maximisation of speedup through the minimisation of load imbalance. The algorithm uses information extracted at runtime to continuously monitor and adjust the work load. The scalability of the algorithm can be questioned as the processors on a parallel machine increase. In this algorithm processors only communicate with their neighbours, this allows the algorithm to be scalable. Ideal policies should be general enough to apply automatically, and be good enough to significantly improve performance. The application of this scheme to complex application codes such as PHYSICA [1] is expected to provide further improvements and justify inclusion of the DLB scheme as an automated phase of the parallelisation process in CAPTools [3]. David M. Nicol and Joel H. Saltz model in [7] look at a similar model to the one presented here to predict how often to do the remapping, but other issues such as how much of the data to migrate when re-balancing, which entities to move and where about in the code to move are not considered.

REFERENCES

1. M. Cross, P. Chow, C. Bailey, N. Croft, J. Ewer, P. Leggett, K. McManus, and K. A. Pericleous. PHYSICA - a software environment for the modelling of multi-physics phenomena. In *Proc ICIAM 1995*, 1996.
2. C. Ierotheou, S. P. Johnson, M. Cross, and P. F. Leggett. Computer aided parallelisation tools (CAPTools) - conceptual overview and performance on the parallelisation of structured mesh codes. *Parallel Computing*, 22:163–195, March 1996.
3. C. Ierotheou, S. P. Johnson, K. McManus, P. F. Leggett, and M. Cross. Semi-automatic parallelisation of unstructured mesh codes. May 1997. In this volume.
4. S. P. Johnson, C. Ierotheou, and M. Cross. Computer aided parallelisation of unstructured mesh codes. In *Proc PDPTA 1997*, volume 1, pages 344–353, July 1997.
5. K. McManus. *A strategy for mapping unstructured mesh computational mechanics programs onto distributed mesh parallel architectures.* PhD thesis, Computing and Mathematical Science, University of Greenwich, 1996.
6. K. McManus, C. Walshaw, M. Cross, P. Leggett, and S. Johnson. Evaluation of the JOSTLE mesh partitioning code for practical multiphysica applications. In *Parallel Computational Fluid Dynamics, implementations and results using parallel computers*, pages 673–680, 1996. Proc Parallel CFD 1995.
7. D.M. Nicol and J.H. Saltz. Dynamic remapping of parallel computations with varying resource demands. *IEEE Trans. Computers*, 37(4):1073–1087, Sept 1988.
8. Thierry Le Sergent and Bernard Berthomieu. Balancing load under large and fast load changes in distributed computing systems - a case study. In *Parallel Processing: CONPAR 94 - VPP VI*, pages 854–865. Springer Verlag, 1994.
9. J.A. Stankovic. An application of bayesian decision theory to decentralised control of job scheduling. *IEEE Trans. Computers*, C-34:117–130, Feb. 1985.
10. C. Walshaw, M. Cross, and M. Everett. Mesh partitioning and load-balancing for distributed memory parallel systems. In *Proc. Parallel and Distributed Computing for Computational Mechanics, Lochinver, Scotland, 1997*, 1997.