Dynamic Load Balancing of Distributed Memory Parallel Computational Mechanics using Unstructured Meshes for Multi-Physical Modelling

V.Aravinthan, S.P.Johnson, K.McManus, C.Walshaw, M.Cross
`V.Aravinthan@gre.ac.uk`

Parallel Processing Research Group
Centre for Numerical Modelling and Process Analysis
University of Greenwich, London, UK

Abstract

As the complexity of parallel applications increase, the performance limitations resulting from computational load imbalance become dominant. Mapping the problem space to the processors in a parallel machine in a manner that balances the workload of each processors will typically reduce the run-time. In many cases the computation time required for a given calculation cannot be pre-determined even at run-time and so static partition of the problem returns poor performance. For problems in which the computational load across the discretisation is dynamic and inhomogeneous, for example multi-physics problems involving fluid and solid mechanics with phase changes, the workload for a static subdomain will change over the course of a computation and cannot be estimated beforehand. For such applications the mapping of loads to processors is required to change dynamically, at run-time in order to maintain reasonable efficiency. The issues of dynamic load balancing are examined in the context of PHYSICA, a three dimensional unstructured mesh multi-physics continuum mechanics computational modelling code [2].

1 Introduction

Multi-Physics modelling on a continuum scale brings together established techniques for structural mechanics and Computational Fluid Dynamics (CFD) to address problems which involve many physical phenomena. The significant non-linearity of the differential equations involved leads to a high computational demand from moderate problem sizes. Parallel computation is required to satisfy this demand. Single Program Multi-Data (SPMD) overlapping Domain Decomposition (DD) techniques established for structured mesh CFD codes have been used by the authors to successfully parallelise unstructured mesh muli-physics applications. As practical experience is accumulated the focus is directed to the improvement of scalability and consequently load balancing with a view to developing techniques that not only improve current performance but provide a foundation for further automation of parallelisation and implementation of Dynamic Load Balancing (DLB).

The cost of high performance parallel computers forces efficient utilisation to justify their purchase but the investment and expertise required to fully convert a conventional sequential code into SPMD DD parallel can be enormous. To relieve this problem, tools are urgently required to assist in the parallelisation process and dramatically reduce the time required for parallelisation but without sacrificing the performance or quality of the resulting parallel software. The Computer Aided Parallelisation Tools (CAPTools) have been developed to handle complex application codes (both new and dusty deck) in a structured (regular) grid context using grid decomposition parallelisation techniques [5, 3]. Exploitation of advanced interprocedural symbolic dependence analysis[4] within a sophisticated environment allows a user to interact with the tools, to direct and enhance the resulting parallel code. These techniques have demonstrated transformation of serial source code into message passing parallel source code that is not only efficient, scalable and portable but is also recognizable and maintainable by the code originators. Extension of the CAPTools SPMD-DD-MP paradigms in the manual parallelisation of a number of unstructured mesh CFD and FEA codes has culminated in parallel codes that exhibit high efficiency on a wide range of parallel machines. Abstraction of the techniques into a generic form has extended the scope of CAPTools to automation of the parallelisation of unstructured mesh based codes.

The performance of parallel software has, for some time, been a focus of attention and so the parameters that govern performance are now well understood. Many components in a parallel application, the iterative solvers for example, can now demonstrate excellent performance on current generation hardware. This is not yet the case for complete packages, especially those that address complex applications such as multiphysics modelling. Many factors impact on the performance of such large scale applications, two important issues for many unstructured mesh multi-physics applications, are scalability and load balancing. Load imbalance, appears in may forms from small, often avoidable, static imbalances to dynamic inhomogeneous load balancing problems. The focus of this paper is directed towards the discussion of strategies that address the important scalability and load balancing issues that arise in unstructured mesh applications and more specifically in unstructured mesh multi-physics applications. Of particular interest is to develop strategies that can assist in the development of dynamic load balancing in a generic automatable framework.

A static mesh partition is unlikely to provide a good load balance when solving dynamic nonlinear 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 the work-load is predicted 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 [2], 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.

1.1 Load balance

Data distribution in an unstructured mesh DD parallel application is ordinarily based on a decomposition of the mesh into P subdomains calculated to balance the computational load on each processor. It is inevitable that the data dependence in a DD parallel application will require punctuation by frequent synchronisation points [6]. Any one processor allocated a greater amount of work than it's peers will cause all other processors to idle while it reaches each synchronisation point.

For a complex application the prediction of computational load is seldom accurate. It follows that a static mesh decomposition, and hence a static load balance may, in practice, be imbalanced. Load imbalance can arise from a number of effects associated with data partitioning for inhomogeneous systems. Static inhomogeneity is evident in many forms:

- In the parallel hardware, different processors may have differing calculation or communication rates.
- In an unstructured mesh application code, calculation may be based on loops over dif-

fering entity types, such as mesh vertices, elements, element faces, coefficients in a matrix system or others.

- In a multi-physics problem case, the calculation may be divided into a number of computational domains, for example, flow in the fluid portion, stress-strain in the solid portion and heat transfer over the entire problem.

To develop a system for improving a static load balance under these conditions would require at run-time; measuring the calculation and communication performance of each processor, measuring the degree of connectivity of each mesh entity and modification of each entity weight in anticipation of which of the loops in the program will include each entity. Such a calculation is fraught with difficulties and presents a significant overhead in itself. The challenges for static load balancing in response to static inhomogeneity are eclipsed by the difficulties presented by dynamic inhomogeneity:

- In the parallel hardware, the processors (and communication network) may be subject to workloads from other users.
- In an adaptive unstructured mesh application code the mesh, and hence the partition may change at each time step.
- In a multi-physics problem case, the computational domains may change at each time step, e.g. solidification, moving boundaries.

Dynamic Load Balancing (DLB) schemes for moderately dynamic load changes have been addressed by many workers but DLB schemes for large and rapid load swings remain a challenge. Load balancing for multi-physics is still in it's nascency. The development of strategies to improve static load balancing is therefore considered with a view to providing a foundation for the development of DLB for multi-physics.

2 Methodology

A practical solution of the dynamic load-balancing problem involves [7]:

- Load Evaluation: An estimate of each processor's load is required to determine the extent of load imbalance.
- Profitability Determination: Does the cost of load imbalance exceed the cost of load migration.?
- Load Transfer Calculation: Based on the measurements taken in the first phase, calculate the work transfers necessary to balance the computation.
- Load Migration: The mesh is repartitioned and the appropriate data transferred between processors.

By decomposing the dynamic load balancing process into distinct phases, experiments can be performed with different strategies for each of the above steps, allowing the impact of differing techniques to be investigated.

2.1 Load Evaluation

Accurate load measurement is necessary both to determine that a load imbalance exists and to calculate how much work should be transferred to relieve that imbalance. This requires accurate measurement of processor execution time, idle time and communication time. Many machines provide clocks with nanosecond level accuracy although standardisation of timer routines is weak. A message-passing library can be instrumented to accumulate such timings into appropriate categories. Any time between communication operations would be labelled as execution time (run time or CPU time), any time actually sending or receiving data would be tagged as communication time and any time waiting to receive a message would be accumulated as idle time. However the accuracy of timing short operations is affected by the overhead of calling the timer in a manner analogous to the Heisenberg uncertainty principle. For many communication calls the overall performance of the code requires that the communication start-up latency is absolutely minimal. Adding a timer to such calls adds to the latency and may cause performance deterioration. As such the accuracy and impact of idle timing is somewhat platform dependent.

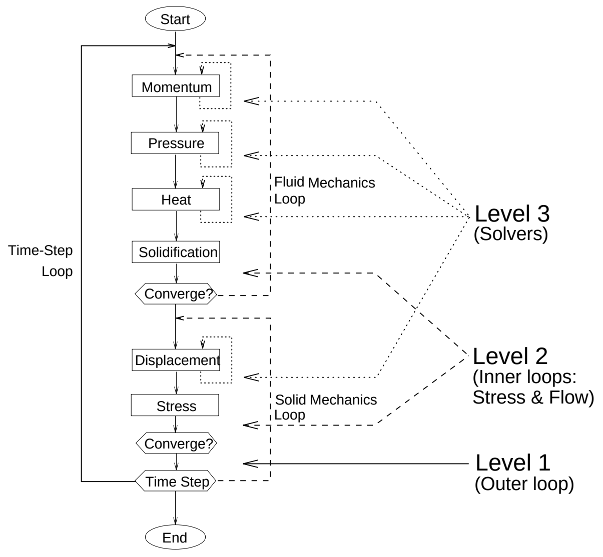

Figure 1: Different levels of loops that can be found in a typical CFD code.

Most scientific codes have different levels of loops within the code, high level loops such as the time step loop surround sweeps to converge each time step and low level loops such as the linear solver iterations. The appropriate part(s) of the application code to time can vary widely between different codes. One code may necessitate the timing of the top loop level, but another code may require timing of a lower level loop. Parallel application codes have inherent synchronisation points within these loops. In particular, global norm calculations and other determination detection mechanisms typically involve a global sum, checking of convergence or some other reduction operation, the results of which are checked by each processor involved. These barrier operations provide an obvious point at which to initiate load balancing.

2.2 Profitability Determination

Clearly the dynamic load balancing mechanism must attempt to optimise the rate at which re-balancing is carried out. The re-balancing process is itself a parallel overhead. Re-balancing too frequently will waste time in unprofitable data migration. If re-balancing is left too long then time is wasted as the load imbalance increases. 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
- 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 dynamic load balancing algorithm. Determination of when to load balance requires two phases: detecting that a load imbalance exists and determining if the cost of load balancing exceeds its possible benefits.

A re-balancing decision heuristic is proposed which assumes that the rate of change of imbalance between processors is always linear, that the re-balancing time is constant, and that re-balancing removes all imbalances. Response to large changes in load has 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) [1]. The model is based on an instance in time and predicts what would happen under the model assumptions. It uses the number of iteration (n) between re-distributions to predict run-time. t_{cost} explicitly embodies two of the costs a re-mapping policy must manage: delay cost of re-balancing and idle-time costs of not re-balancing.

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

Where i is the time taken for each iteration, B is the rate of increase of imbalance across the processors and J is the re-balancing time. The re-balancing time is then minimised with respect to n (see equation 2). The model (equation 3) predicts an optimal value for n that minimises the run-time prediction function. Re-distribution will be performed n iteration after the previous re-distribution.

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

$$n = \sqrt{\frac{2J}{Bi^2}} \qquad (3)$$

2.3 Work Transfer Calculation

After determining that it is advantageous to load balance, the amount of work-load that must be transferred from one processor to another must be calculated. The parallel mesh-partitioning tool JOSTLE provides a highly localised optimisation algorithm with graph reduction to both accelerate the optimisation and, perhaps more importantly, provide a more global perception in a manner analogous to multigrid techniques. Most significantly for this work, JOSTLE iteratively optimises and, if necessary, load balances an The load balance information is indicated by a weighted graph of the primary mesh entity type that is passed to JOSTLE together with the current processor ownership array of the primary entity type. JOSTLE will then attempt to optimise the balance of these weights in a partition that minimises the amount of communication and minimises data migration in returning a new processor ownership array indicating where the data should move.

Using the ownership array for primary entity types the ownership arrays for the secondary entity types are updated (eg. faces and nodes based on an element's partition). The only concern here, which will limit the amount of workload being transferred, is when the amount of data transferred exceeds the available memory on the processor involved. Only a few current parallel architectures provide support for virtual memory, so there is fixed limit on the amount of available memory at each computer. Even if virtual memory is provided, the cost of exceeding the amount of physical memory may well surpass the cost of the load imbalance.

2.4 Data Migration

A dynamic load-balancing framework must also provide a mechanism for actually moving the data from one processor to another. This includes identifying and updating the data associated with each moved entity, such as an element's vertices, temperature, pressure, etc. The load balancing algorithm and consequent data movement must be very fast in comparison to the overall run-time 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. It is therefore important to relate the overhead cost of remapping with the expected performance gain.

One of the important goals of distributed memory parallelisation is to allow scalability of memory so that, as processor numbers are increased, the size of mesh that can be handled also increases. This implies the use of locally numbered entities which compounds the difficulty of moving data between processors. The communication is implemented in two phases, one constructs a movement set (within a processor) and a communication set (between processors) listing where entity numbers are to be sent/received to/from which processor, and other performs the communication using that communication set for a particular variable. Some arrays (particularly geometry pointer arrays themselves) have to be converted to global numbers before any movement can take place. The data structures required for the parallel execution and domain decomposition consist of the local to global numbering array which stores the original global number of locally numbered entity. Hence, this local to global numbering array can be used to convert all the pointer arrays from local to global numbers. Following the construction of the new partition, the pointer arrays must be renumbered to the new local numbering scheme.

3 Results

The dynamic load balancing algorithm has been successfully implemented in PHYSICA at the time-step loop level (see figure 1). It has been tested with a solidification test case which models a cooling metal bar. The bar begins all liquid at a temperature just above solidification and is cooled from one end so that a solidification front moves along the bar. After 30 time-steps, the bar is almost completely solid (Figure 2).

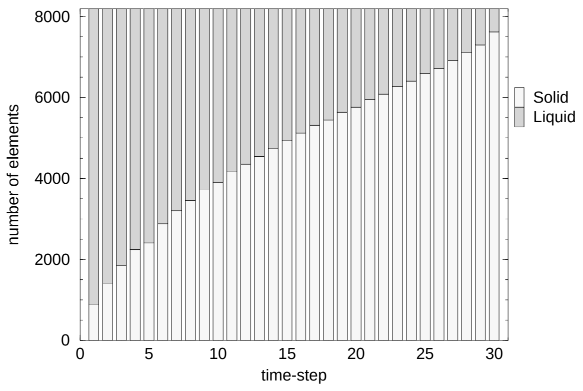

Figure 2: Solidification of a cooling bar

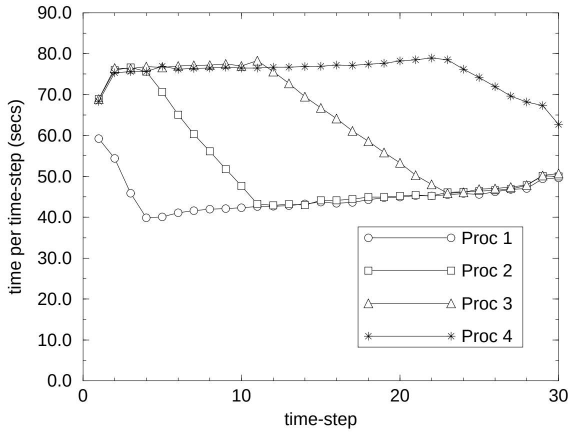

Figure 3: Time-step time without dynamic load balancing

Figure 3 shows the wall-clock time per time-step for each processor without DLB and Figure 4 shows the times with DLB. It can be seen that in Figure 3 the overall run-time is restricted by the load imbalance leading to a time per time-step of around 80 seconds. Figure 4 shows the initial time per time-step correspondingly at almost 80 seconds being steadily reduced to a final figure of less than 60 seconds. The migration of

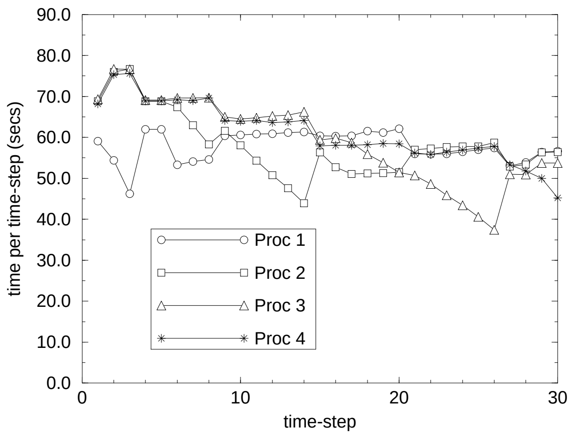

Figure 4: Time-step time with dynamic load balancing

mesh elements that produces this improvement is shown in Figure 5. Here the steps in the curves are the points at which the DLB algorithm has chosen to migrate work. A 20% reduction in the overall run time produced by DLB is shown in Figure 6

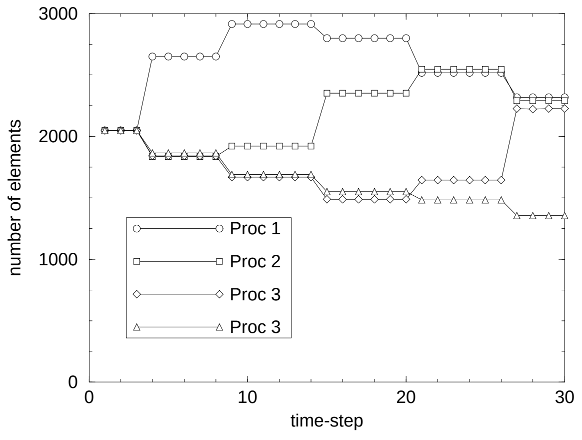

Figure 5: Number of elements on processors

4 Conclusion

For a given problem then as P increases the importance of load balance in measured parallel performance becomes increasingly significant. For dynamic inhomogeneous load imbalances characteristic of multi-physics problems it may not be possible to obtain a *good* load balance. Nevertheless it is clear that DLB can reduce the a load imbalance in an initial partition and so provide a worthwhile performance improvement.

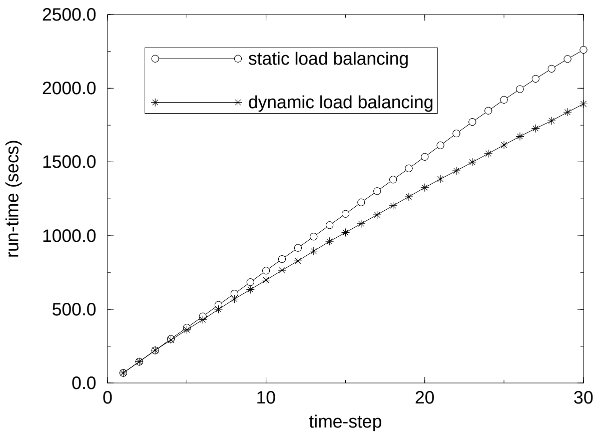

Figure 6: Overall run-time

The development of parallel JOSTLE has provided an opportunity to advance the state of the art in practical unstructured mesh parallel applications and in particular dynamic load balancing. This development has been driven by, amongst other things, the demands of both CAPTools and PHYSICA, which fill the role of user to provided JOSTLE with a requirement definition, user feedback and test vehicles to both verify and validate development. The strength of this synergy is reinforced by continual input from CAPTools and JOSTLE to the development within parallel PHYSICA of strategies for generic, automatable and robust dynamic load balancing.

The DLB scheme in this paper has consequently been developed and tested in PHYSICA using test cases that illustrate the challenging issues in load balancing for dynamic inhomogeneous problems. Information extracted at run-time is used to continuously monitor and migrate the work load as the developing solution causes the work load to move across the problem space. The resulting methodology is not only successful in reducing run-time but is also sufficiently generic to be amenable to automatic parallelisation in CAPTools.

References

[1] A. Arulananthan, S.P. Johnson, K. McManus, C. Walshaw, and M. Cross. A generic strategy for dynamic load balancing of distributed memory parallel computational mechanics using unstructured meshes. In *Parallel Computational Fluid Dynamics, recent developments and advances using parallel computers*, pages 43–50, 1998. Proc Parallel CFD 1997.

[2] 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.

[3] E. W. Evans, S. P. Johnson, P. F. Leggett, and M. Cross. Automatic generation of multi-dimensionally partitioned parallel CFD code in a parallelisation tool. In *Proc. PCFD'97*. Elsevier Science Publishers B.V., 1997.

[4] S. P. Johnson, M. Cross, and M. G. Everett. Exploitation of symbolic information in interprocedural dependence analysis. *Parallel Computing*, 22:197–226, 1996.

[5] S. P. Johnson, C. S. Ierotheou, and M. Cross. Automatic parallel code generation for message passing on distributed memory systems. *Parallel Computing*, 22:227–258, 1996.

[6] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, pages 103–111, August 1990.

[7] J. Watts. A practical approach to dynamic load balancing. Master's thesis, Scalable Concurrent Programming Laboratory, California Institute of Technology, 1995.