41

Dynamic Load Balancing for Multi-Physical Modelling using Unstructured Meshes

V.Aravinthan[1], S.P.Johnson, K.McManus, C.Walshaw, M.Cross

Introduction

As the complexity of parallel applications increase, the performance limitations resulting from computational load imbalance become dominant. A mapping that balances the workload of the processors in a parallel machine will typically increase the overall efficiency of a computation and so reduce the run-time. For many cases the computation times associated with a given task cannot be pre-determined even at run-time and so static load balancing returns poor performance. For many classes of problems, such as multi-physics problems involving fluid and solid mechanics with phase changes, the workload may 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 if reasonable efficiency is to be maintained. This paper examines the issues of dynamic load balancing in the context of PHYSICA, a substantial three dimensional unstructured mesh multi-physics continuum mechanics computational modelling code based on finite volumes methods [CCB+96]. PHYSICA is primarily intended for the modelling of the processes involved in metal casting. It provides a range of modelling facilities including compressible Navier Stokes flow, heat,

1 Parallel Processing Research Group, Centre for Numerical Modelling and Process Analysis, University of Greenwich, London, UK. email: V.Aravinthan@gre.ac.uk

Eleventh International Conference on Domain Decomposition Methods
Editors Choi-Hong Lai, Petter E. Bjørstad, Mark Cross and Olof B. Widlund ©1999 DDM.org

phase change, elastic-visco plastic, solid mechanics, chemical reaction and turbulence. The proposed *generic* strategy should be applicable to any unstructured mesh multi-physics parallel codes regardless of the application.

Parallel Processing

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 even moderate problem sizes. Parallel computation is required to satisfy this demand. In this respect, Single Program Multiple Data (SPMD) overlapping Domain Decomposition (DD) techniques have been used by authors to successfully parallelise unstructured mesh multi-physics applications e.g. [McM96].

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. DD is used to map such problems onto multiprocessor machines so that regions of the model domain are assigned to each processor. The operational 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 end of the step, 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 [CCB$^+$96, McM96]. As practical experience is accumulated the focus is directed to the improvement of scalability and consequently load balancing.

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. 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 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 modelled 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 the number of grid point adjacencies 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 coarsening will inevitably suffer from significant load imbalance. Even with a fixed mesh, multi physical simulations which include the modelling of phase changes such as melting or solidification [CCB$^+$96], 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. Such Dynamic Load Balancing (DLB) schemes for moderately dynamic load changes have been addressed by many workers [SB94, Wat95] but DLB schemes for large and/or rapid load swings and generic DLB schemes remain a challenge. The presented *generic* 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.

Methodology

A practical solution of the DLB problem involves [Wat95]:

- Load Evaluation: Some estimators of a processor's load must be required to first determine that a load imbalance exists.
- Profitability Determination: Once the loads of the processors have been measured, the presence of a load imbalance can be detected. If the cost of the imbalance exceeds the cost of load balancing, then load balancing should be initiated.
- Load Transfer Calculation: Based on the measurements taken in the first phase, the ideal work transfers necessary to balance the computation are calculated.
- Load Migration: Workloads are transferred from one processor to another.

By decomposing the DLB 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.

Load Evaluation

The effect of any load-balancing scheme is directly dependent on the quality of load evaluation. Good 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. One way to easily overcome the performance peculiarities of a particular architecture is to measure the load of a task directly. Typical machines provide clocks with millisecond to microsecond level accuracy. These timing facilities can be used to time each task, providing accurate measurements in the categories of execution time, idle time and communication overhead. In fact, the user need not manually time the

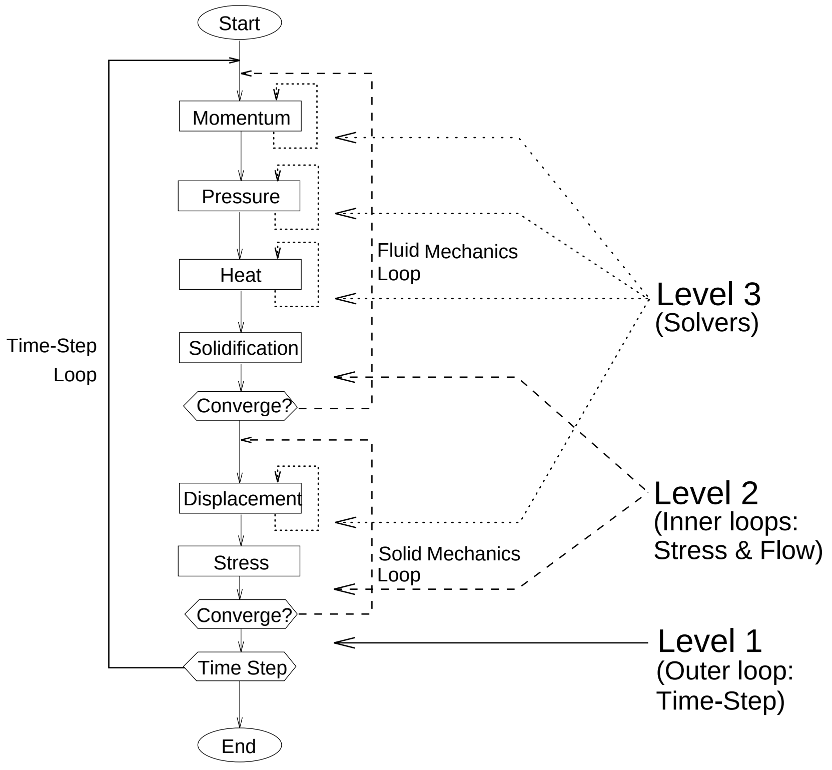

Figure 1 Different levels of loops that can be found in a typical CFD code.

code at all. These timings can be easily taken at the library level. A message-passing library could certainly be instrumented to accumulate time into various categories. Any time between communication operations would be labelled as runtime (execution 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.

Most scientific codes have different levels of loops within the code, for example a top level loop such as the time step loop and lower level loops such as the solver loops (see Figure 1). Thus 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 the lower loop levels. These application codes usually have inherent synchronisation points within the loops. In particular, global norm calculations and other termination 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 a natural, clean point at which to initiate load balancing.

Profitability Determination

The DLB model is a run-time overhead and so must not initiate the rebalancing mechanism too frequently as this will waste time on moving the data around. On the other hand, if the rebalancing mechanism is initiated too infrequently, the load between the processors of the parallel machine can become badly balanced and hence the performance will deteriorate in time. Thus, it is important to correctly determine the criteria that will be used to decide when 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 will compensate for the cost of the DLB algorithm.

When a barrier is initiated, the average load of all of the processors is determined. If the aggregate efficiency is below some user-specified limit, the workload is considered to be imbalanced. Even if a load imbalance exists, it may be better not to load balance, simply because the cost of load balancing would exceed the benefits of a better work distribution. The time required to load balance can be predicted directly by keeping a record of time taken in previous load balance(s). The expected reduction in run time due to load balancing can be estimated loosely by assuming efficiency will be increased to 100 percent or more precisely by maintaining a history of the improvement in past load balancing steps. If the expected improvement exceeds the cost of load balancing, the next stage in the load balancing process should begin.

A re-balancing decision heuristic is proposed here 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) [AJM$^+$98]. 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 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 incurred by not re-balancing.

t_{cost} is given by:

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

where i is the time taken for each iteration, B is the rate of increase of imbalance across 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 iterations after the previous re-distribution.

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

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

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. Here, the JOSTLE mesh-partitioning tool is used [WCE97]. JOSTLE can be used to rebalance an existing partition in parallel whilst minimising the amount of data migrated. The load balance information is indicated by a weighted graph of the data (e.g. elements) that is passed to JOSTLE together with the current processor ownership array of the data. JOSTLE will then attempt to balance these weights in the resultant partition.

JOSTLE returns the new processor ownership for the local core data, indicating where the data should move. Using this array, processor ownership arrays for the secondary data are updated (e.g. faces and nodes based on an element's partition).

Data Migration

A DLB framework must also provide mechanism for actually moving the data from one processor to another. This includes identifying and updating moved entities data such as an element's temperature, pressure, U-velocity, V-velocity etc. The load balancing algorithm and consequent data movement must be very fast in comparison to the overall run-time. DLB 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. Distributed memory systems require that all data are distributed and so data on each processor is locally numbered. Thus the framework must be able to handle entities that 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 although locally numbered entities make the moving of data between processors more difficult.

The communication is implemented in two phases. The first phase constructs a movement set (within a processor) and a communication set (between processors) listing the entity numbers to be communicated and the processor to communicate with. The second phase performs the communication using the communication set calculated in the first phase for a particular variable.

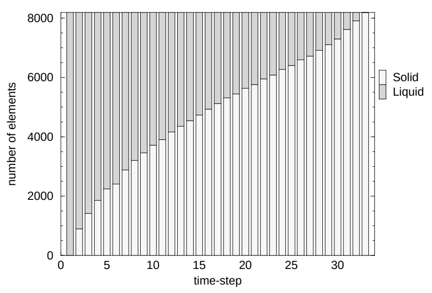

Figure 2 Solidification of a cooling bar

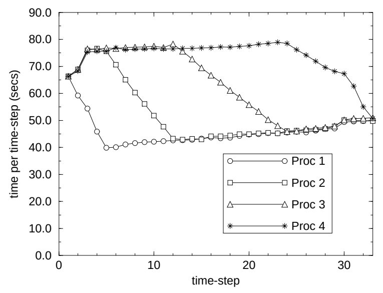

Figure 3 Time-step time without DLB

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 DD 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.

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 (see Figure 2).

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 in Figure 3 that the overall run-time is restricted by the load imbalance leading to the execution time of around 80 seconds per time-step for most of the calculation. Figure 4 shows the initial execution time correspondingly at almost 80 seconds per time-step being steadily reduced to a final figure of less than 60 seconds. A 20% reduction in the overall run-time produced by DLB is shown in Figure 5.

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

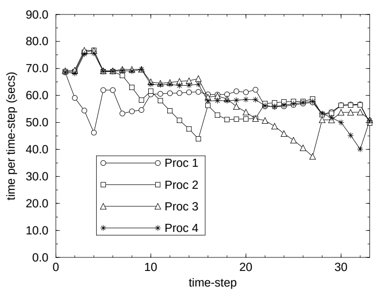

Figure 4 Time-step time with DLB

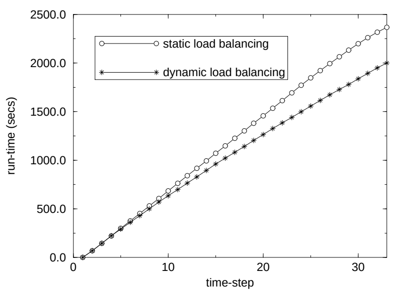

Figure 5 Overall run-time

load imbalance in an initial partition and so provide a worthwhile performance improvement.

The development of parallel JOSTLE has provided an opportunity to advance the state of the art in practical unstructured mesh parallel application and in particular DLB. The DLB scheme in this paper has been developed and tested in PHYSICA using test cases that illustrate the challenging issues in load balancing for dynamic inhomogeneous problems. Information extracted at runtime is used to continuously monitor and migrate the workload as the developing solution causes the workload to move across the problem space. The resulting methodology is not only successful in reducing run-time but should also be sufficiently *generic* to be applicable to a diversity of unstructured mesh based codes.

REFERENCES

[AJM$^+$98] Arulananthan A., Johnson S., McManus K., Walshaw C., and Cross. M. (1998) A generic strategy for dynamic load balancing of distributed memory parallel computational mechanics using unstructured meshes. In *Proc Parallel CFD 1997*, pages 43–50.

[CCB$^+$96] Cross M., Chow P., Bailey C., Croft N., Ewer J., Leggett P., McManus K., and Pericleous K. A. (1996) PHYSICA - a software environment for the modelling of multi-physics phenomena. In *Proc ICIAM 1995*.

[McM96] McManus K. (1996) *A strategy for mapping unstructured mesh computational mechanics programs onto distributed mesh parallel architectures.* PhD thesis, Computing and Mathematical Science, University of Greenwich.

[SB94] Sergent T. L. and Berthomieu B. (1994) 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.

[Wat95] Watts J. (1995) A practical approach to dynamic load balancing. Master's thesis, Scalable Concurrent Programming Laboratory, California Institute of Technology.

[WCE97] Walshaw C., Cross M., and Everett M. (1997) Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *J. Par. Dist. Comput.* 47(2): 102–108.