Runtime Support for Parallel Execution of Scientific and Engineering Applications

Satya R. Dash
Supervisor: Derek L. Eager

Abstract

This paper discusses the requirements of a desirable parallel runtime support system for scientific and engineering applications in general and computational fluid dynamics applications in particular. Existing runtime support systems are reviewed and the skeleton of a new proposed programming environment is formulated combining the best features of existing systems and incorporating some new ideas. Features of the runtime system critical to parallel performance but not of direct relevance to the end-user are proposed to be encapsulated at the lower echelons of abstraction. Load balancing issues in non-deterministic computational environments are analysed. The attempt is to identify some of the key issues of major importance to this field.

1 Introduction

In this report a review of various existing runtime support systems developed for parallel execution of scientific and engineering applications is discussed. The discussion is against the backdrop of MIMD multiprocessors and traditional imperative programming languages. The programming style is discussed in the context of SPMD (Single Program Multiple Data) style of programming where the same program body resides on each processor but with possibly different code fragments being executed at any point of time. Periodic synchronization events are built into the SPMD program in order to support cooperation among the various processors.

Many runtime support systems developed are tailored for particular types of scientific computations, capitalizing on the problem specific characteristics. The review begins with a general discussion on the key components of parallelism in scientific and engineering applications. Next the nature of the application domain and the resulting parallel programming issues to be considered while designing a parallel support system are discussed. After a comparison of the existing systems, features of our new proposed runtime system are put forward. Finally we conclude.

No attempt has been made to do a comprehensive review of all existing runtime support systems till date. Neither is any of the runtime support systems thoroughly described. Instead the attempt is to identify some of the key issues of major importance.

2 Program Parallelism

Parallel programming for scientific and engineering applications has been a subject of intense research in the past two decades. Different approaches to extract parallelism while developing new application software (or from existing sequential scientific programs) vary in the assumptions they make about the application program structure. These are discussed in the following pages.

2.1 Data Parallelism

Data parallelism is where the same program code runs over different sets of data in each processor. This has received the most attention because of its pervasiveness in scientific and engineering applications.

Scientific (and engineering) applications typically model a physical system and large arrays of data representing properties of interest at each point in space naturally arise which result in programs characterised by the presence of iterating loops over a large amount of data. This has been the focus of the compiler community and the study of parallelisation of such code has matured to a good extent [25]. Some of the successful programming languages in the Fortran community are based on this programming model [13], [4], and [20]. The tedium of parallelisation under such model is handled by and large by the compiler, with the scientific programmer restricted to annotizing the sequential code for supplying hints to the compiler on data distribution [10].

The greatest strength of data parallelism is that the overhead of programmer time incurred in exploiting data parallelism, whenever it is possible, is substantially reduced due to availability of compiler support [11]. The drawback of data parallelism approach is the inherent static nature of this programming model. Partitioning and scheduling issues, consequently, are determined by data partitioning at compile time.

However, as expertise in extracting parallelism from static loop based dependency analysis has matured researchers have increasingly looked at other forms in which parallelism is embedded in the sequential source code. Data parallelism is weak in handling all such issues.

2.2 Functional Parallelism

Functional parallelism appears when a potentially heterogeneous mix of tasks could be running simultaneously on possibly different sets of data. Task parallelism may be defined as parallelism not dictated by the distribution of data structures used for representation of a problem, such as the execution of different functions in parallel or parallelisation of loops in other than the *owner computes principle.* [1] Functional parallelism may decompose a problem domain into multiple SPMD programs.

Functional parallelism is appropriate in a number of situations where simulations are composed of different kinds of subsystems (FFT computation) or inter-disciplinary applications (climate modelling) [9], [17]. They also arise naturally in problems that can be reduced to graph traversals. Different discipline codes execute concurrently, interacting with each other only to share data.

2.3 Runtime Parallelism and Integration of Data and Functional Parallelism

Both data and task parallel programs may contain potential parallelism which is manifest only at run time. Data parallel programs could contain data dependencies that can only be resolved at execution time [18]. Functional parallel programs may similarly contain runtime resolvable tasks such as recursive data structures of unknown depth. Alternatively a program may have non-deterministic cpu execution time requirements which are resolvable only during program execution such as numerical convergence.

In multi-disciplinary class of applications in addition to an outer level of functional parallelism, the individual subsystem often exhibit substantial internal data parallelism and therefore, the appropriate model is neither but a combination of both. Such is expected to be the future of parallel computation in the coming years where potential applications would tend to be more and more of a heterogeneous nature. The advent of heterogeneous mix of advanced multi-disciplinary scientific applications requiring multiple tasks do not fit into either the data or task parallel paradigms. These, therefore, require the development of languages which can express both data and task parallelism and permit interaction between them.

Under such conditions there is a motivation to build an underlying support system for resolving such *runtime issues* rather than burdening the scientific application programmer with the details.

[1] A principle in loop iteration distribution across processors where the processor which stores a given array element is responsible for all computation where a write operation is performed on that element.

3 Programming Approaches and Machine Architecture

3.1 Programming Models

In the context of traditional imperative programming languages there are two basic approaches to the problem of how data and functional coordination is achieved between processes. The distinction is based on how communication is achieved. In a shared-memory programming model interprocessor communication is performed implicitly through memory access instructions. On the other hand in message-passing programming model the communication is consciously performed by the programmer.

The shared-memory model is viewed as the one easiest for programmers to use because of the provision of a global name space. The programmer does not have to bother himself with communication other than interprocess synchronisation operations. Due to this reason most parallel programming systems tend to offer a global name space to the programmer irrespective of whether it is supported by the underlying physical machine.

However, for programs developed in a shared-memory programming environment (especially one virtually operating above a physically distributed memory), unless the programmer or compiler pays careful attention to data distribution, a higher communication overhead may be incurred as compared to those developed in a message-passing programming model. This is in a great part due to the programmer not sensing the cost of communication in a shared-memory model. Scalability of communication overhead in general is low under this model because with increasing problem size, data structures may exceed the local memory bounds and need a redistribution. The programmer (or compiler) must need to be aware of these scaling problems. Further, typical synchronisation constructs used in shared-memory programming such as locks, semaphores, and barriers have whose contention overhead can increase dramatically with larger number of processors.

On the other hand in message-passing programming model the communication costs are always visible up front and hence are generally the first target of optimisation.

3.2 Parallel Machine Memory Architecture

Orthogonal to the programming model is the memory architecture of the physical machine. Small scale single bus architectures support true physical centralised shared memory. Due to practical limitations on bandwidth and the number of processor interconnects single bus machine interconnects are restricted to about 30 [12]. On the other hand scalability is the driving force behind message passing distributed memory massively parallel systems. Consequently, all large scale systems possess physically distributed memory. But this comes at a cost of greater software complexity. The data has to be partitioned among the various nodes of a message-passing architecture bringing into forefront the disparity in local and global memory access costs. Thus there exists a classic hardware versus software complexity trade-off between the two type of machines.

3.3 Ideal Combination

It appears to be popular among the parallel computing community to synergize the benefits of both the single address space (shared memory programming model) and distributed memory architecture. Recent parallel support systems implement a virtual shared memory for programmer convenience whenever a physical shared-memory is not available (we will see these later when we review some application-specific Runtime Support Systems). These, therefore, may be visualised as instances of the shared-memory computing model in that they hide the underlying peculiarities of a distributed memory system.

The difference between physically supplied single shared memory (or architectures where the hardware supports a single address space even though the memory is physically distributed) and systems where this abstraction is provided only through software (operating system or compiler), is the issue of performance.

A software supplied single address space abstraction implies that one of the major assumptions of a shared-memory programming model (as found in small scale single bus machines), that all memory accesses have the same cost is no longer valid. Several subtle tuning requirements come into play such as efficient underlying communication, synchronisation, domain decomposition, and scheduling issues which should be handled efficiently but be transparent to the programmer who sees a single large memory. Furthermore, an efficient programmer transparent address translation mechanism from the global shared-memory space to individual processor local memories needs to be constructed. While saying this it must be acknowledged that one does need to distinguish architectures where the hardware supports a single address space even though the memory is physically distributed and systems where this abstraction is provided only through software (operating system) or compiler.

The issues with performance in principle can be attributed to cost of communication. This issue in large scale systems is always present irrespective of the memory view that is offered to the programmer because of the underlying distributed memory architecture. In large scale systems with physically distributed memory, data mapping to attain maximum locality, in the hierarchy of local and remote memories, is often one of the critical factors governing parallel efficiency. There is a parallel phenomenon in centralised memory systems where the hierarchy is formed by the cache and main memory. [16] have demonstrated that communication between local processor caches and main memory can frequently be a bottleneck in achieving high efficiency. [2]

In short what is being driven at, is that the issues involved in parallel computing are broadly similar in large scale systems with respect to the fundamental factors that affect the performance of parallel programs, in either shared-memory or message passing abstraction. Consequently, in this paper no distinction between the two memory architectures would be made, unless it is essential.

4 Techniques Used for CFD Applications

Before we discuss runtime systems it may be worthwhile to categorise the different kind of problem domains that are typically found in CFD applications. We will also study the parallel programming issues that are thrown up by the nature of the application domain.

It is typical of scientific and engineering applications that the most frequent requirement for computation is to solve some differential equation over a given spatial region. Problems in fluid flows, electromagnetism, quantum mechanics, structural mechanics, statistical mechanics, quantum physics, etc. all involve solutions to partial differential equations. The differential equation in CFD problems arises from analysis of one or more fluid variables which is a continuous function over the defined spatial region. The numerical solution tries to approximate the behaviour of the continuous fluid variable. There exist two general classes of approaches for the numerical approximate solution :

- Grid Based Techniques.
- Quasi-Particle Techniques.

4.1 Grid Based Techniques

4.1.1 The General Method

In this class of methods the spatial region of interest is subdivided into a *grid or mesh.* The mathematical representation of the continuous fluid variable is now considered as a series of discrete values at the mesh points.

There are two generic grid based solution techniques : the *Finite Difference method* and the *Finite Element Method.* Only a brief discussion of the two would be presented in the following paragraphs.

In finite difference scheme the governing differential equation is approximated at each grid point in the discretized region. The result is an equation containing the unknown variables' values at each grid

[2] Although this study was conducted with data sets which entirely fit into the cache, the results intuitively have a more general validity.

point. There are *implicit methods* which lead to a system of linear equations to be solved and which comprise the computation intensive portion of the problem. *Explicit methods*, however, can be solved directly without the need for generating system of equations. The reason we are trying to distinguish between implicit and explicit methods is due to the fact that in implicit methods, equation system solvers employ iterative methods, while explicit methods instead are a direct solution march over the spatial or time domain. Iterative methods are repeated until convergence and in the most general case are non-deterministic in terms of the computational load they offer.

Finite element methods implicitly couple the solution at all grid points by approximating the differential equation over the discretized domain by a piecewise continuous mathematical function (as opposed to point values in finite differences) with each grid point contributing as boundary conditions. Finite element methods possess two important characteristics. One, there exists two phases of computation. In one phase the application of the finite element approximation leads to a system of linear equations (as in the case of finite differences) which are solved in the next phase. Two, the computational load of the equation solver (the second phase) is far greater than building the equations themselves.

By far the most common grids are rectangular, especially so with finite difference technique. But triangular, quadrilateral, tetrahedral, and other unstructured shapes arise with finite element techniques because these grids possess more powerful to handle completely general geometries and complex behaviour. With triangular grids the generally accepted way of triangulation is that of Delaunay Triangulation (Voronoi meshes) because it leads to the generation of diagonally dominant matrices which are mathematically better conditioned [23].

Unstructured meshes lead to problems which are termed as irregular. The reason behind this terminology is apparently based on the irregular representation in terms of arrays or irregular data access patterns. While with rectangular meshes it is easy to partition the data among processors (often by a quick recursive bisection scheme [5]), with unstructured meshes the problem becomes much more difficult [24] because of need to optimise the load balance and communication overhead. The problem becomes more urgent with time-dependent solution-adaptive grid refinement procedures where one needs to react quickly to changes in data structure. Irregular problems also arise in particle computations where data access patterns change rapidly between iterations as particles move.

4.1.2 Multilevel Methods

It is obvious that finer the resolution of the grid the more accurate the approximate solution would be. [6] reports that advances in algorithms, numerical solution techniques, problem models, appear to be reaching asymptotic limits in terms of trading off computational cost for accuracy. It is now more effective to increase the number of grid points to improve spatial resolution and hence accuracy. However, the finer the mesh the greater is the computational load. The computational load increases as, at least, of the order of k^d with grid refinement, where, 'k' is the degree of refinement and 'd' is the dimension of the grid. As a result it is not possible to go to very fine grid sizes in the entire problem domain with the available computing power in the conventional solution methods.

Multigrid [7] and *multiblock* methods are the answers to this dilemma of accuracy versus computational feasibility trade-off. These are the recent popular methods for accelerating the solution of PDEs (Partial Differential Equations) in CFD problems.

- **Multigrid Method :** In multigrid methods [7] the problem is discretized on several nested grids of widely different mesh sizes. Interactions between these levels enable to solve PDEs with a fine discretization step in a much shorter time than otherwise possible and to adapt the discretization to the evolving solution in an optimal way, even when singularities are present.

  The philosophy behind multigrid methods is based on the nature of the relaxation (solution) process on a grid. Solution errors of the scale of the grid size are corrected in a few iterations; however, larger scale errors are corrected very slowly. Thus in the first few iterations relaxation produces a rapid convergence rate, but soon after the convergence rate levels off. Further computational effort produces vastly diminished returns. This is precisely where the relaxation sweeps at a single level may be discontinued and relaxation on coarser grids may be employed.

In the algorithm for multigrid method initially relaxation is employed to obtain an approximate solution on the finest grid and the errors smoothened out. Then corrections to this approximation are performed on a coarser grid recursively. Optionally, an approximate solution on the coarser grid is used as an input to find a solution of greater precision with a more refined mesh. Due to this movement from one level to at various stages, solutions from one grid are required to be projected onto another.

A number of numerical issues arise in any multigrid implementation - mesh ratio (generally 1:2), error smoothening criteria, convergence criteria.

- **Multiblock Methods :** These [22] are a generalisation of multigrid methods. The computation is split into several simpler coupled problems in disjoint sub-domains whose union forms the original domain. Therefore, a multiblock calculation is essentially a collection of separate computations in different sub-domains *loosely coupled* together by boundary conditions. The decomposition into sub-domains is usually motivated by one of the following reasons [22]:

  - The mesh generation is simpler individually in the sub-regions while in the overall domain it is of great geometrical complexity.
  - It is necessary to isolate computationally expensive domains for adaptive grid refinement.
  - There is a need to carry out computations on different physical phenomena occurring in each sub-region or to employ different numerical methods depending on the nature of the sub-domain.
  - Need for separation into sub-domains to treat discontinuities.
  - And finally, of recent, multiblock decomposition may be motivated for ease of coding a CFD problem on parallel machines.

  In many instances a multiblock problem can and does comprises of several nested multigrid sub-problems.

  The key idea in multiblock is the loose coupling of blocks through interblock boundary contact points. For a typical application, the main body of the program consists of an outer sequential loop, and an inner parallel loop. The inner loop iterates over the blocks of the problem, after applying boundary conditions to each block. These boundary data may be stored in specially designated cells referred to as *ghost cells*, since the data is secondary to that in the interior of the blocks or real cells [1]. The computation steps of all blocks are synchronised at discrete intervals (usually time steps) before a fresh cycle begins.

  The principal advantages of the multiblock approach stem from the loose coupling between blocks. Since the code deals with only individual blocks it is modular. *In passing it may be pointed out that there exists a strong parallel of the multiblock approach with multidisciplinary problems, where operations within a block comprise a data parallel computational chunk and the block itself representing a task. It may be further emphasised that multigrid and multiblock concepts are merely general approaches to solving a discretized region and are orthogonal to the underlying grid structure being solved - rectangular, irregular, etc.*

4.2 Quasi-Particle Methods

Quasi-particle [19] methods attempt to solve CFD problems by simulating approximately (statistically), the behaviour of the particles making up the fluid. The collective behaviour of the fluid is extracted as an average of the quasi-particles behaviour. Since the true fluid particles are molecules, the approximation involved is in treating a collection of molecules of some larger size as quasi-particles.

It may be pointed out that quasi-particle methods refers to a class of fluid dynamics numerical approaches, with several individual methods such as the Grid-and-Particle method, the Vortex method, a Particle-In-Cell method, and several other variations. The common thread in these is a long-range interaction calculation among the particles and the dynamic nature of the spatial distribution of the particles. In addition, particle methods may utilise grid techniques to solve differential equations to obtain several secondary quantities.

Quasi-particle methods possess the advantage that they are generally numerically stable. Their drawback is a relative lack of efficiency and accuracy because of the dependence of the solution accuracy on the number of particles approximated due to the statistical nature of computation . The smoothness of the computed solution depends on the number of quasi-particles one can afford. The computational requirement increases in magnitude much greater than the increase in number of quasi-particles due to multiple particle-particle interactions.

5 Parallel Programming Issues in CFD Applications

In order to achieve desired speedup upon parallelisation several aspects have to be considered. These are outlined very briefly. We will be concentrating on the communication and load balancing issues. Data layout management to assist locality and reduce non-remote communication would also be discussed here.

CFD (and engineering and scientific applications) by nature - because of their problem domain which essentially - model a spatial region which in turn is discretized into cells. Because of this characteristic there is a significant amount of inbuilt data parallelism which can be statically allocated. Domain decomposition is concerned with this partitioning of the problem domain to balance the workload among the several processors.

5.1 Communicating Data

The nature of communicating data are intricately linked to the data structure employed for representing the physical domain of the problem being solved. Whenever the mesh points are uniformly spaced it is fairly obvious to choose an array based representation. Complications immediately arise with multigrid since the refinement of the grid and therefore, the number of mesh points is only determinable at runtime. Thus provision for dynamic creation of arrays is required. Communication of data between grids of different resolutions implies that data exchange between arrays of different dimensions take place. This, in addition to mapping from one array space to another, it may also involve interpolation of data associated with some mesh points, in line with multigrid philosophy.

The above scheme is further compounded when one deals with multiblock applications. In accordance, with the fact of coexistence of blocks with different resolution grids in them, there is a need for dynamic creation and movement of varying dimensional coexisting arrays.

On the other hand with irregular grids and non-uniform meshes it is not obvious that an array is desirable, considering the highly irregular distribution of particles/mesh points. While it is tempting to use linked lists to represent non-uniform arrays of data, there is a problem of handling pointers in a distributed multiprocessor environment subsequent to data migration. Consequently, the common approach has been to fall back on arrays by employing a sparse matrix representation for those cases where mapping of the complete spatial domain (irregular) would lead to bloated matrices.

The bottom line of the preceding discussion is that the most common form of communicating data in scientific and engineering applications appear as arrays and mapping of data from one dimension to another of different dimension is important in CFD applications.

Over and above the magnitude of communicating data, by virtue of the fact that these are in the form of arrays, several optimisations may be performed to reduce communication overhead. The major problem, of course, in a multiprocessor is communication latency. As a result it is always more beneficial to send longer messages. Message coalescion and aggregation,collective communication and communication prescheduling are general techniques which have been implemented to overcome the communication latency cost both for information known at compile time [13], [15] as well as those identifiable at runtime [18]. An equally important issue is data replication wherein some data which occupy very little memory but are heavily shared may be replicated in each processor's closest memory, instead of being partitioned and thereby imposing costly inter-processor communication.

5.2 Dynamic Load Balancing

Suppose domain decomposition of some domain has been performed. Each processor computes within its assigned region until some preset milestone is reached. Then it waits until the other processors finish their assigned work. At this point the load allocation balance among processors is reevaluated and reallocated equally among all processors if necessary. In consequence the parallel program executes in two phases - a computational phase followed by a synchronisation phase. Thus effectively their exists a barrier where the children are synchronised. Load balance may be performed during or after the synchronisation phase based on measurement of the non-uniformity of the workload distribution in the just completed computation phase.

Most existing parallel programming environments adopt this system of a "computational phase" followed by a "load balance phase". Usually recursive bisection method of decomposition is employed for this purpose [5]. Here the original domain is split into two equal workload halves which are recursively halved until the number of sub-domains match the number of processors. Normally this works fine as long at each load rebalance step the work is reallocated equally among all processors and hot spots [3] do not develop. However, this process does not take into account the fact that in completely reallocating work the "affinity" [4] that some data may have for particular processor could be completely destroyed. In a heterogeneous multiprocessor environment where the availability of machines is not dedicated to a single program and consequently available processing power is unpredictable, it is not even possible to implement recursive bisection and yet attain balanced domain decomposition. Under such circumstances incrementally rebalancing the load among processors based on the previous partition and the most recent performance measure rather than starting from *ab initio* could be perhaps more profitable. The overhead of not preserving "affinity" has been estimated to be as high as 30-60 % of the total execution time [16] in a different context.

Barrier synchronised load rebalance also displays an weakness when the computational requirements for each partition is skewed and is not known in advance. True. techniques such as self-scheduling, scattered decomposition, wrapped allocation would alleviate this problem. However, multi-block applications where the computations in different sub-domains could be widely different, is not susceptible to any of these treatments. Thus there is a clearly defined need for reducing load imbalances by means of load sharing between processors at points other than barriers at the end of each computational phase. An equally likely source of non-uniformity, in spite of the equal work allocation, could be that may be the number of mesh points in each sub-region is not the same, possibly because the mesh point spatial density is not uniform(such as adaptive grid refinement). As related in the previous paragraph another form in which non-uniformity may arise, even when the initial subdivision of work among processors is equal, is when the processors are heterogeneous in terms of available cpu power.

In conclusion it is easily apparent that neither of these conditions will exist in general. There appears to be an identified need to study load sharing of work among processors as well as affinity accounted incremental load balancing in non-uniform and non-predictable applications as well as in the presence of non-coherent multiprocessor architectures. The question that arises is whether such applications exist? The answer is in the affirmative in the CFD domain where frequently adaptive grid refinement is used which leads to tightly coupled multiblock applications. More generally, in the future with the advent of greater use of parallel machines for solution of multi-disciplinary applications need for load sharing and maintaining data locality (affinity) would become a pressing need.

---

[3] Sub-domains where the work content is highly imbalanced with respect to its neighbours

[4] Units of work have "Affinity" for a particular processor due to the data locations the processor accessed in its last execution period and which are still available in the local cache at the end of execution period and could be potentially be reused in the next execution step.

6 Application Specific Runtime Systems

The application specific runtime systems that would be covered in this section are :

- PARTI (Parallel Automated Runtime Toolkit at ICASE)
- DIME (Distributed Irregular Mesh Environment)
- GenMP (GENeric MultiProcessor)
- LPARX (Lattice PARallelism)
- Adhara

Table 1 summarises the features of the above runtime systems. The terminology used in the table is clarified now. The level of communication encapsulation is highly optimised when the user only calls a primitive but is not responsible for managing the communicated data. Coarse grained parallelism is analogous to task decomposition except that the atomicity of each task is large enough to contain substantial amount of data parallelism at lower granularity.

PARTI [1] geared for multigrid/multiblock applications demonstrated that while precomputing communication is essential, it is necessary to recover the cost through amortisation over many iterations. It was also projected that communication overhead is substantially reduced if each block is not subdivided across a large number of processors (too fine grained parallelism). Therefore, provision for distribution of blocks (or arrays) into a subset of the multiprocessors was made available.

DIME [23] is a programming environment for computing on a triangular mesh (Delaunay Triangulation) in 1-D or 2-D in a distributed memory network of processors. During the course of computation DIME does load rebalance and adaptive grid refinement at the end of every computational phase.

LPARX [3] and Adhara [14] are two application specific runtime systems which provide the application programmer with a high level abstract machine which completely releases them from any burden of parallel programming. In Adhara data distribution unlike generalised Fortran versions is completely encapsulated; however, LPARX leaves it the programmer. The user in these systems sees a single large virtual uniprocessor. These are two support systems which one has in mind while suggesting a new application specific programming environment.

6.1 Summary and Comparison of Features of Existing Systems

The survey conducted above portray many of our ideas and comments presented during the discussions in Sections 3 to 6. A summary is :

- General acceptance of the need for provision of a global name space for end-user.
- Increasing need for extraction of coarse (functional) and fine-grained (data) parallelism.
- Transparency of communication of data and optimisation of all such data movements.
- Abstract machine offered to the user : Adhara and LPARX emphasise the need for offering very high level abstraction for the end-user and completely relieve him from the presence of a multiprocessor architecture. The others require the end-user to distribute the data.
- We also note that affinity retaining load balancing is not performed although it does not prevent the user from actually implementing it himself. Therefore effectively load balancing and domain decomposition are identical.

The requirements of the programming environment that we aim, in addition to those listed above, would be to provide an abstract machine which requires the application programmer to only specify his physical computation and identification of the tasks in a multiblock application. Out of all we have surveyed only Adhara and LPARX do this.

Runtime System	Issue	Features
PARTI [1]	Application Environment Parallelism Programming Environment Communication Data Layout Management Load Balancing	Regular Array Problems - specifically Multigrid/Multiblock Integration of Coarse and Fine Grain Parallelism Global Name Space Provided Encapsulated - Highly Optimised Distribution by User Barrier Synchronised, Managed by User
DIME [23]	Application Environment Parallelism Programming Environment Communication Data Layout Management Load Balance	Computations on Unstructured Triangular Meshes Coarse Grained - SPMD Global Name Space Provided Encapsulated but no indication of optimisations Distribution by User Encapsulated but Internally Barrier synchronised
GenMP [2]	Application Environment Parallelism Programming Environment Communication Data Layout Management Load Balance	Spatially localised mesh computations Coarse Grained Global Name Space Provided Encapsulated but no indication of optimisations Specified by user Internally Barrier synchronised
LPARX [3]	Application Environment Parallelism Programming Environment Communication Data Layout Management Load Balancing	Non-Uniform Computations Coarse Grained Global Name Space Provided Encapsulated but no indication of optimisations Specified by user Barrier Synchronised, Managed by User
Adhara [14]	Application Environment Parallelism Programming Environment Communication Data Layout Management Load Balancing	Non-Uniform Computations Coarse Grained Global Name Space Encapsulated and optimisations for this class of application Encapsulated Encapsulated, Barrier Synchronised

Table 1: Salient Features of Existing Application Specific Runtime Support Systems

7 Proposed New Runtime System

A new runtime support system is described in this section. First a justification on why an application specific system is needed is presented and then a summary of the useful features that should be present in the new system are discussed. Then a layered overview of the system organization is presented. Clearly at the moment only the skeleton of the system has been sketched.

7.1 Why an Application Specific Runtime System

Why not a general purpose runtime system? The clue to the rationale behind application specificity lies in the ease-of-user-interface. The researcher involved in developing applications needs performance not parallelism. He has a conceptual model of a problem and a proposed algorithm for solving it. The key to end-user friendliness is to enable him to develop big applications in a short period of time.

[8] in COOL have also summarised the features that a parallel programming support system should provide for performance in a fashion analogous to the present one. However, they go in for programmer annotations and other assistance for efficiency. The reason for this could possibly be due to lack of domain knowledge and this reinforces the need for application specific support systems.

Ease-of-use can be achieved by abstracting away the computer science part from the application programmer and allowing him to concentrate on the science of his computation. It is proposed that the functionality of parallel programming be provided in the form of software templates. The generality of the software environment is achieved by providing a large number of templates whose ensemble encompasses a wide target audience. While each template would be geared for a specific application (in our case multigrid/multiblock) it can be tailored to individual needs within a short time frame.

The reason that Adhara is able to completely encapsulate all details of computation is because of its restriction to a particular application. All communication optimisations and load distributions that are needed are performed by the runtime support system for the application programmer transparently. As a result an user in Adhara merely specifies the physics of his computation.

Application specificity in the present context implies multigrid/multiblock applications. In case of multiblock, the application programmer would be asked to specify the following details of computation.

1. The spatial domain.
2. Sub-domains of each blocks.
3. The differential equation system being solved in each block, the discretization scheme adopted, and the stencil depth.
4. Local convergence criterion and global convergence criterion.
5. Grid refinement criterion.
6. Number of levels for a multigrid method, error smoothening criterion.

Needless to say the underlying runtime system should perform all the tuning necessary to obtain the maximum performance.

7.2 Load Sharing

In Section 5.2 need for load sharing in non-dedicated multiprocessor systems was highlighted. It is planned to automatically perform load rebalance whenever imbalance arises in the course of computation. And this rebalance should be done in an incremental way so that the affinity of data already accessed is preserved. One possible approach is to rebalance the load locally within each processor's neighbours and let it percolate down eventually to all processors. This would have the benefit of preserving data affinity to the maximum extent. [16] and [21] have done this within the context of distributing iterations on multiprocessors for maintaining affinity. This needs to be carried forward in the context of more complex programs such as those resulting from solution of differential equations in multigrid applications (overloaded with different grid density in various sub-domains) not only from affinity considerations but from load sharing standpoint.

8 Concluding Remarks

In this report a survey of characteristics of scientific and engineering applications with emphasis on computational fluid dynamics applications was done. Issues important to parallel programming were summarised. A few existing systems were reviewed. It was found that in order to effectively encapsulate low level programming details from the application programmer - but which are necessary from efficiency considerations - application specificity of the support system was desirable. It was concluded that load balancing/sharing issues in heterogeneous multiprocessor environments needed to be addressed.

References

[1] Agarwal, G., Sussman, A., and Saltz, J. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. Tech. Rep. 93-77, ICASE, NASA Langley Research Center, October 1993.

[2] Baden, S. B., and Kohn, S. R. The reference guide to GenMP - the generic multiprocessor. Tech. Rep. 92-243, Dept. of Computer Science and Engg., Univ. of California, San Diego, 1992.

[3] Baden, S. B., Kohn, S. R., and Fink, S. J. Programming with LPARX. Tech. Rep. 377, Department of Computer Science and Engg., University of California, San Diego, 1994.

[4] Banerjee, U., Eigenmann, R., Nicolau, A., , and A., P. D. Automatic program parallelization. *Proceeedings of the IEEE 81*, 2 (February 1993).

[5] Berger, M. J., and Bokhari, S. H. A partitioning strategy for non-uniform problems on multiprocessors. *IEEE Transactions on Computers C-36*, 5 (1987).

[6] Boris, J. P. *New Directions in Computational Fluid Dynamics in Annual Review of Fluid Mechanics*, vol. 21. Annual Reviews Inc., 1989, ch. 4.

[7] Brandt, A. Multi-level adaptive solutions to boundary value problems. *Mathematics of Computation 31*, 138 (April 1977).

[8] Chandra, R., Gupta, A., and Hennessy, J. L. Data locality and load balancing in COOL. In *4th ACM Symposium on Principles and Practice of Parallel Programming* (1993).

[9] Chandy, K. M., Manohar, R., Massingill, B. L., and Meiron, D. I. Integrating task and data parallelism with the collective communication archetype. Tech. Rep. CS-TR-94-08, Department of Computer Science, California Institute of Technology, August 1994.

[10] Gupta, M., and Banerjee, P. PARADIGM: A compiler for automatic data distribution on multicomputers,. In *International Conference on Supercomputing* (July 1993).

[11] Gupta, S. K. S., Kaushik, S. D., Mufti, S., Sharma, S., Huang, C.-H., and Sadayappan, P. On compiling array expressions for efficient execution on distributed-memory machines. In *International Conference on Parallel Processing* (1993), vol. II.

[12] Hennessy, J. L., and Patterson, D. A. *Computer Organization and Design : the Hardware/Software Interface.* Morgan Kaufmann, 1994.

[13] Hiranandani, S., Kennedy, K., and Tsang, C.-W. Compiling Fortran D for MIMD distributed memory machines. *Communications of the ACM 35*, 8 (August 1992).

[14] Immaneni, A. *Runtime Support for Dynamic Space-Based Applications on Distributed Memory Machines.* PhD thesis, Dept. of Computer Science and Engg., Univ. of Washington, Seattle, 1994.

[15] Li, J., and Chen, M. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems 2*, 3 (July 1991).

[16] Markatos, E. P., and LeBlanc, T. J. Using processor affinity loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 5* (April 1994).

[17] Mehrotra, P., and Haines, M. An overview of the Opus language and runtime system. Tech. Rep. 94-39, ICASE, NASA Langley Research Center, May 1994.

[18] Mirchandaney, R., Saltz, J. H., Smith, R. M., and Nicol, D. M. Principles of runtime support for parallel processors. In *ACM International Conference on Supercomputing* (July 1988).

[19] Oran, E. S., and Boris, J. P. *Numerical Simulation of Reactive Flow.* Elsevier, 1987.

[20] Rogers, A. M., and Pingali, K. Compiling for distributed memory architectures. *IEEE Transactions on Parallel and Distributed Systems 5*, 3 (March 1994).

[21] Subramaniam, S., and Eager, D. L. Affinity scheduling of unbalanced workloads. In *Supercomputing 94* (November 1994).

[22] Veuillot, J.-P., and Cambier, L. *A Sub-Domain Approach for Computation of Compressible Inviscid Flows in Numerical Methods in Euler Equations of Fluid Dynamics*, vol. 21. SIAM, 1985.

[23] Williams, R. *DIME : Distributed Irregular Mesh Environment.* Caltech Concurrent Computation Project Technical Report No. C3P 861, California Institute of Technology,, February 1990.

[24] Williams, R. Performance of dynamic load balancing algorithms for unstructured mesh calculations. Tech. Rep. C3P 913, Caltech Concurrent Computation Project, California Institute of Technology, June 1990.

[25] Wolfe, M. J. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, Massachussets, 1989.