# Automating Runtime Optimizations for Load Balancing in Irregular Problems

Sanjeev Krishnan and Laxmikant V. Kale
Department of Computer Science,
University of Illinois, Urbana-Champaign.
Email : {sanjeev,kale}@cs.uiuc.edu
Phone : (217) 244-0094

**Abstract**

In order to reduce the effort required for attaining good performance for parallel programs, it is necessary to use automated performance optimizing techniques. In this paper we describe run-time optimizations for load balancing, and techniques to automate them without programmer intervention using post-mortem analysis of parallel program execution. These techniques are very useful for applications having irregular, non-uniform or dynamic load patterns. We classify the characteristics of parallel programs with respect to object placement (which determines load balance), then describe techniques to discover these characteristics by post-mortem analysis, and present heuristics to choose appropriate load balancing schemes based on these characteristics. Our ideas have been developed in the framework of the *Paradise* post-mortem analysis tool for the parallel object-oriented language Charm++. We also present results for optimizing simple parallel programs running on the Thinking Machines CM-5.

## 1 Introduction

Parallel program development cost has been recognized as one of the bottlenecks preventing more widespread use of parallel computers. The increased cost of parallel software is due to several new issues which have to be tackled before parallel programs can attain the peak performance that parallel computers provide. The goals of high performance and better programmability are difficult to achieve simultaneously. The challenge is to attain good performance at low programming cost.

Parallel programming skills are also not widespread among application programmers, and moreover, the programmer may not know enough about the characteristics of the application and implementation without investing considerable effort. E.g. the choice of a load balancing scheme for an application requires knowledge of the application's load characteristics. Thus it is difficult to anticipate and respond to all potential performance problems during the design of the first prototype of a parallel program. The first prototype is hence likely to have several serious performance flaws such as load imbalance, leading to active research for identifying

performance problems and solving them using optimization techniques. These steps lead to a significant portion of the programming cost in the parallel program development cycle. Since most parallel programmers are not skilled in identifying and solving performance problems, *expert parallel programming knowledge must be embodied in tools which are available to the parallel programmer.*

In particular, using expert knowledge either in the compiler or the run-time system to automating program optimizations can significantly help to reduce the parallel software development effort. When most performance problems are solved in this manner, there will be fewer iterations of the development cycle. Even in cases where completely automatic techniques are not possible, it is beneficial to automate the optimization steps to the extent possible. This paper is concerned with precise post-mortem analysis which can be used for automating run-time optimizations for load balancing, especially for irregular, non-uniform or dynamic problems.

## 2 The Optimization Framework

This work is part of a larger project to develop a framework for automatic runtime optimizations [1, 2].

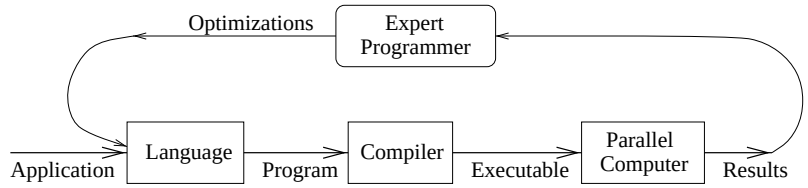
The need for runtime optimizations arises because compiler transformations alone are not sufficient to optimize all parallel programs. In particular, compiler optimizations cannot take into account unpredictable run-time execution environments and unknown application characteristics. Static compiler analysis is also restricted by the difficulties of precise dependence and type analysis in the presence of pointers, and by separate compilation of program modules. Finally, for library based systems such as PVM and MPI, runtime optimization is the only means to get better performance.

### *2.1 Paradise: Automating Runtime Optimizations*

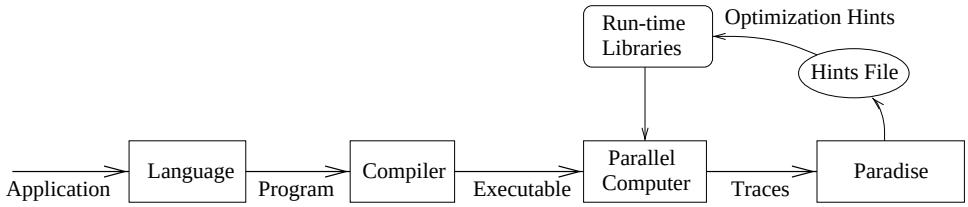
Run-time optimizations such as for load balancing need information about characteristics of the parallel program in order to enable optimization mechanisms and guide strategies for selecting them. Since compilers cannot statically deduce the required information in many cases, programmers often *manually* analyze the execution of a program in order to discover program characteristics and optimizations (Figure 1a). Automatic post-mortem analysis may thus reduce the effort required for performance optimization.

Most parallel programs needing optimizations such as for load balancing fall into one of two categories: the first consists of those for which good optimization strategies can be derived from well known heuristics; for such programs an automated expert system for guiding optimization can potentially achieve good results. The second category consists of programs which need new algorithms and techniques for optimizing them; for such programs the intervention of a human programmer is obviously needed. However, experience with parallel applications has shown that many of them have well-recognized performance problems for which optimizations can be found using a few good heuristics.

This paper presents current results from an ongoing project to develop a framework for automating run-time optimizations (Figure 1b). The framework involves the expert post-mortem analyzer PARADISE (PARallel programming ADvISEr) which analyses traces of program execution, finds program characteristics and suggests optimization hints. Paradise works in close cooperation with a run-time system which uses the hints to parameterize optimizations and




**(a) Conventional program development cycle**




**(b) Program development with run-time optimizations driven by the intelligent post-mortem analysis tool Paradise**

Figure 1: Parallel program development cycles (reproduced from [2]).

select between alternate optimization strategies. Paradise builds a representation of the program's execution from traces, determines characteristics of the program, uses heuristics to find optimizations that will solve the performance problems, and generates concise hints which are communicated to runtime libraries by a "hints file".

### *2.2 Previous work*

Automatic post-mortem analysis has been used in performance analysis tools for parallel programs, to give the user insights into performance problems such as load imbalance using expert analysis [3, 4, 5, 6]. Our framework aims to go a step further in the direction of automation: Paradise not only finds performance problems, but also solutions in terms of optimizations for the problem areas, and in co-operation with the run-time libraries, incorporates the optimizations in the program without programmer intervention. Some compilers for data-parallel languages such as HPF [7] use profile information to accurately find the cost of various computation and communication operations. In [8], profile information is used to manually calculate weights to be used in weighted graph decomposition, for irregular data-parallel applications. To the best of our knowledge, our framework is one of the first efforts towards using post-mortem analysis for automating selection of load balancing strategies, and incorporating them into the program without programmer intervention. Also, our optimizations apply to dynamic, non data-parallel and irregular applications as well as regular ones.

Another unique aspect of our work is that it is in the context of a parallel object-oriented language which allows the run-time system the flexibility to choose strategies for placement (mapping) of computations. Thus there are significantly greater opportunities and challenges for automatic load balancing strategies. In contrast, message-passing layers such as PVM or MPI require the programmer to explicitly specify the placement of computations, and also do not provide facilities for dynamic creation of tasks, thus restricting the extent of automatic load balancing. Again, data-parallel languages such as HPF present a much simpler

regular computational model for which optimizations are easier to perform, as compared to a parallel object-oriented model which involves dynamic creation of tasks and asynchronous communication.

# 3 Program model

For completeness, we include a brief description of the programming model and its post-mortem representation, taken from [2].

This work is based on the parallel object-oriented language *Charm++*, [9, 10] which is an extension of C++. The basic unit of work in Charm++ is a *chare*, which is a medium-grained concurrent C++ object. Chares are dynamically created; there may be thousands of chares per processor. A chare type is a C++ class containing data and functions which may be triggered by the arrival of messages. Functions inside chares are atomic (they may not be pre-empted).

Chares communicate by sending messages to functions (invoking methods) in other chares asynchronously. An essential feature of the Charm++ parallel programming model is *asynchronous message driven execution*, which helps latency tolerance by overlapping communication and computation. All calls to the run-time are non-blocking, and there are no "receive" calls. Remote accesses are performed in a split-phase manner. Each processor has multiple chares, and a pool of messages targeted to methods in the chares. The scheduler picks messages from the pool one by one, and "processes" them by invoking their target methods in the proper chare objects.

Charm++ also provides multidimensional parallel arrays of objects which are distributed over processors using a user-specified mapping function. Array elements may communicate with each other in a point-to-point manner or using multicast communication primitives.

**Post-mortem representation**:
The execution of a Charm++ program is represented as an *event graph*, which is essentially a dynamic task graph constructed using traces collected at run-time. Issues in collecting trace data, reducing perturbation, and constructing the basic event graph are discussed in [6] and are beyond the scope of this paper. A simple version of the event graph was originally used for the *Projections* [11] performance visualization and analysis tool. The event graph constructed by Paradise consists of vertices representing entry-function executions, edges representing messages between entry functions and edges for dependences between methods (these dependences must be specified in the language or generated by the compiler). Also, all vertices belonging to the same object instance are grouped together.

# 4 Parallel program characteristics affecting object placement

A load balancing strategy determines the placement of objects onto processors. In order to select a suitable object placement strategy which balances load, Paradise attempts to systematically discover the characteristics of the parallel program from the event graph. We first describe four important characteristics which affect object placement, and discuss how they are inferred from the event graph.

### *4.1 Object grainsizes*

The grainsize of a method is the amount of work done (load) in the method execution, and is computed as the difference in time from the start to the completion of the method (since all methods in Charm++ are atomic and all operations are non-blocking). The grainsize of an object instance is the sum of the grainsizes of all method executions for that object. The grainsize of an object type is the average over all objects of that type. The average grainsize of a program is computed by taking into account all parallel objects on all processors. If the average grainsize of a program is too small (e.g. comparable to the message latency on the machine it was run on), it indicates that the program might suffer from large overheads, for which corrective optimizations may be needed. If the average grainsize is too large, it indicates less parallelism leading to significant idle times on processors.

A useful characteristic of a program is the amount of variation in object loads. A large variation in loads requires more complex dynamic load balancing strategies. If all grainsizes are nearly the same, simpler strategies may suffice, or more accurate optimizations may be possible.

### *4.2 Patterns of object creation*

Object creation patterns tell us which processors create objects, and at what times in the program execution they are created. The patterns of object creation determine the times and locations where object mapping decisions have to be made, and thus determine the strategy for collection and distribution of load information which is needed for making the object mapping decisions at run-time. E.g. For a data-parallel program where arrays of objects are created at the beginning, no load information needs to be collected; on the other hand, for state-space search where the search tree is expanded in the course of the program, load information must be continuously updated so that new objects can be sent to underloaded processors.

The *locations of object creation* may be centralized (one processor creates all the objects) or distributed (many processors create objects). This characteristic can be easily inferred by counting the number of objects created on each processor. In general, distributed object creation requires more complex load balancing strategies. The *times of object creation* may be continuous (objects are created continuously) or bursty (objects are created in bursts, with intervening periods when no objects are created; a special case of this is when objects are created just once, at the beginning of the computation).

### *4.3 Data locality*

The behavior of a program with respect to data locality tells us the extent of access to non-local data. It is desirable to increase data locality in a program so that the amount of data accessed from remote processors decreases. Data locality can be increased by taking into account interactions between objects while making object mapping decisions: closely interacting objects should be mapped to the same processor. In order to deduce patterns of inter-object interactions, we construct an object-interaction graph in which the nodes are objects and edges represent communication between objects. The weight of an edge represents the amount of communication (e.g. number of messages) between the pair of objects it connects.

If there is very little inter-object communication or if there is "all-to-all" communication, it

is not worthwhile trying to increase locality. However, if the communication is tree-structured (each object communicates only with its creator or child objects) or graph-structured (e.g. neighbor communication) it is worthwhile trying to increase locality by mapping closely interacting objects to the same processor.

# 5 Optimizing dynamic object placement

There are two main aims of an object placement strategy: to balance load across processors; and to maintain locality by moving objects only when necessary and keeping heavily interacting objects on the same processor.

The control points for dynamic object placement are from the time of seed[1] creation through the time the seed is dispatched by the run-time for creating the new object.

There are three components of a load balancing scheme. The *load collection* component determines how load information from different processors is collected. The *initial mapping* component determines the processor to which a newly created seed is sent. Finally the *rebalancing* component is responsible for redistributing seeds and objects after they have been initially assigned to a processor.

## *5.1 Schemes for dynamic object placement*

Several schemes may be used for dynamic object placement, with different levels of sophistication, overheads and for different types of load balancing problems. Some of the schemes commonly used are:

- Randomized: when a new object is created, it is placed on a random processor, in the hope that the randomization will eventually even out the number of objects per processor. Each processor creating objects generates a different random number sequence to prevent all processors from placing load on the same destination processor.
- Round-robin: processors are ordered in a round-robin sequence and a new object is placed on the next processor in the sequence. (Each processor maintains its own round-robin sequence of destination processors). This is a simple, low-overhead scheme, which works well if all objects have approximately the same grain-size. The drawbacks are that it will not work well if there are a few large-grained objects (there are not enough objects to go around the sequence) or if objects have variable grainsizes, or if there is a need to maintain locality.
- Neighbor averaging: each processor exchanges load information with its neighboring processors, and if the difference in loads is greater than a threshold, the overloaded processor sends work to the underloaded processor. Otherwise processors keep newly created seeds with themselves. Thus neighbor averaging works by smoothing out differences in load. Load gradually *diffuses* from highly loaded neighborhoods to underloaded neighborhoods. The advantage of this scheme is that it maintains locality because new objects are kept local as much as possible. However, there is some overhead associated with collection of load.

[1] A seed is the initialization message for a new object.

- Centralized manager: one processor is designated as the manager, and the rest of the processors are workers. Workers send new seeds to the manager, which distributes them equally among the workers.

- Distributed manager: this is a more scalable version of the previous scheme. The processor set is divided into clusters, each of which has is own manager which balances load within a cluster. Periodically, managers also balance load between themselves. This scheme can quickly adapt to unpredictable changes in processor loads, hence is useful when object grainsizes are variable. However, locality is not maintained because objects may be sent to arbitrary processors. There is also some overhead for sending seeds to the manager, hence this strategy will work well only if the grainsize of objects is large enough to amortize messaging overhead for seeds.

The Charm run-time system already provides some load balancing schemes such as randomized, neighbor averaging, and distributed manager. We have also designed a new parameterized load balancing scheme for tree-structured computations which exploits information about the tree-structure to optimize load balance [2]. All these schemes are automatically enabled at runtime based on the hints suggested by Paradise in the hints file.

### *5.2 Information required for dynamic object placement*

In addition to the application-specific characteristics described in Section 4, the following input-specific information is useful for dynamic object placement:

- Processor load information: Since load patterns can vary widely across applications, no single scheme for collecting processor loads may be good for all applications. For example, load information may be sent out by lightly loaded processors which need to receive work, or by heavily loaded processors which need to give away work, or in a periodic manner, or in particular stages of an application, etc.

- Load per object; also, load for the entire sub-tree of objects which are created by an object. If each processor can estimate the sum of future loads due to all its objects (assuming all newly created objects are kept local), it is easier for a scheme to balance loads accurately. This load information can be specified by the programmer as object-local variables or as parameters while creating an object.

- Information about interactions between objects: this is necessary for maintaining locality. This information can be specified by the programmer in the form of object-affinity hints.

### *5.3 Heuristics for automating dynamic object placement*

For programs which create objects dynamically throughout the execution of a program, Paradise chooses a load balancing scheme depending on the program's characteristics. The hint generated contains the chosen scheme and any necessary parameters. At execution time, this hint is read in by the load balancing module in the Charm runtime system and is used to activate the chosen scheme. Currently Paradise chooses one of three schemes: round-robin, neighbor averaging, and distributed manager. The heuristics used by Paradise to automatically choose a scheme are:

```
if ( object creation is centralized )
        if ( all objects have the same grainsize )
                Choose the round-robin scheme.
        else
                Choose the distributed-manager scheme (the processor
                creating all objects is the manager).
        endif
else
        if ( there is significant inter-object communication )
                Choose the neighbor-averaging scheme (it maintains locality
                by only moving objects when necessary to balance load).
        else if ( the average grainsize is sufficiently large )
                Choose the distributed-manager scheme (grainsize is large
                enough, so there are not too many objects; overhead of sending
                seeds to the manager will not be significant).
        else if ( all objects have the same grainsize )
                Choose the round-robin scheme.
        else
                Choose the neighbor-averaging scheme (large number of objects
                with varying grainsize: none of the other two will work).
        endif
endif
```

These rules embody some of the expertise we have accumulated while optimizing several applications requiring dynamic load balancing schemes [12, 13]. Note that this expertise can be brought to bear on this problem only because the post-mortem analysis based on the enhanced event graph is able to identify the relevant characteristics of the parallel computation.

### *5.4 Example programs*

In order to test Paradise's ability to correctly choose a dynamic object placement strategy for programs which dynamically create work, we used the following test programs.

**Variable grainsize objects:**
This is an artificial program which creates a number of objects of varying grain-sizes, in the form of an irregular, large-branchfactor tree. Paradise was able to deduce the following characteristics with respect to placement:

1. the program has dynamic object creation
2. there are no phases
3. the communication overhead is not significant
4. the average grain-size of objects is sufficiently large.

Hence it suggested the distributed-manager strategy (Section 5.1). This hint was written into the hints file and read in by the load balancing libraries at runtime, thus enabling the chosen strategy. Table 1 presents results from running the program on 16 processors of the CM-5, with 137 objects being created.

**Heavily communicating objects:**
This is an artificial program which creates a number of objects of the same grain-size, but

| Strategy | Roundrobin (default) | Dist-Manager (automatic) |
|---|---|---|
| Time | 2624 | 2290 |

Table 1: Time (in milliseconds) with the default and automatically chosen load balancing strategies for the variable-grainsize program.

which communicate heavily with each other. The structure of the program is an irregular tree, with large messages being sent from children to parents in the tree. Paradise was able to deduce the following characteristics with respect to placement:

1. the program has dynamic object creation
2. there are no phases
3. the program has an irregular tree structure
4. the communication overhead is significant

Hence it suggested the neighbor-averaging strategy, which keeps objects local to the processor which created them as far as possible, thus reducing the number of messages that need to go across processors. Table 2 presents results from running the program on 16 processors of the CM-5, with about 5200 objects being created.

| Strategy | Roundrobin (default) | Neighbor-Avg (automatic) |
|---|---|---|
| Time | 7685 | 6326 |

Table 2: Time (in milliseconds) with the default and automatically chosen load balancing strategies for the heavily-communicating objects program.

## 6 Optimizing static object placement for irregular programs

Many applications, especially array-based applications in science and engineering, do not create objects dynamically: all objects are created at the beginning of the program. In such a case it is possible to place objects at the beginning of execution using a static placement strategy. As for dynamic placement, the two considerations for a static placement strategy are to balance load and maintain communication locality. The control point for allowing the run-time to determine static object placement can be provided by a function call to a partitioning or placement library, at the beginning of the program.

Work on compiler techniques for automatic data partitioning in array-based Fortran and HPF programs has achieved considerable success; block and cyclic mappings of regular arrays can be generated by compilers. However, there are many other types of irregular/dynamic applications for which static placement cannot be done by only compile-time analysis. Even for array-based scientific programs, block/cyclic mappings are not sufficient for many applications. Finally, since the number of processors and the size of the array and other parameters can vary from run to run of a parallel program, we need run-time decisions about the processor on which a particular object should be placed.

When the amount of work done in the element objects of a parallel object array varies significantly, these load variations must be taken into account while assigning objects to processors, in order to balance load across processors. Often variations in load arise due to input-dependent load patterns. E.g. In a particle simulation, the simulation space is divided into regions and each region is assigned to an object. The load of an object is proportional to the number of particles in its region. When the input particle distribution is non-uniform, there may be a large variation in the number of particles (hence load) per object.

Paradise detects a variation in load by comparing the grainsizes of the array element objects and checking to see if they vary significantly. If there is significant variation in load, the program is classified as an irregular program. For such programs, an *input-dependent runtime object placement scheme* must be used. Currently the scheme used for partitioning a parallel object array is Orthogonal Recursive Bisection (ORB).

ORB is a well known low-overhead scheme which recursively bisects a multidimensional space into two partitions by planes orthogonal to the coordinate axes, such that the load in each partition is approximately equal. The bisection process stops when the number of partitions is equal to the number of processors. Since the partition assigned to each processor is a rectangular (convex) subspace of the original multidimensional space, the communication (which is proportional to the boundary of the region) volume is also reduced. Thus at the end of ORB each processor gets a set of objects corresponding to a rectangular subarray of the original parallel object array, such that all processors have equal loads.

Thus for irregular programs Paradise generates a hint to the runtime libraries to use ORB for partitioning the parallel object array. Additionally, the runtime system needs information about when the partitioning must be applied. Since ORB partitioning requires information about the load of each object, it can only be applied after the object array has been initialized. An appropriate point at which ORB can be applied is the first synchronization point in the program. Accordingly, Paradise finds the phase number corresponding to the first synchronization point, and includes this phase number in the optimization hint.

Another important issue is the problem of conveying load information for each object to the ORB library. This is solved using inheritance. All parallel array objects are required to inherit from the "loadarray" class which contains a "load" variable. Each object is required to set this variable in its constructor (e.g. the load may simply be the number of particles in the object's region)[2]. When the ORB algorithm is initiated, each processor can find the load of all objects it contains by reading the "load" variable in each object. All load values are collected on processor 0, which then applies the ORB algorithm (thus the actual ORB algorithm itself is just a sequential algorithm), and broadcasts the resulting mapping to all processors.

Thus Paradise enables object placement for input-dependent irregular programs without programmer intervention. Section 6.1 gives results for an example irregular program.

### *6.1 Example irregular object-array based program*

We demonstrate automatic static object placement for irregular programs using an idealized particle simulation application. This type of program occurs in several scientific applications, including molecular dynamics and gas flow simulations. The program uses a two dimensional parallel object array to represent a computational space in which particles are distributed in

[2] Alternatively, in iterative programs it may be possible for the runtime system to automatically store the load of an object in its "load" variable during one iteration, and use the load information for optimizing mapping in subsequent iterations.

a non-uniform manner. Thus each object (which represents a region of the space) contains a variable input-dependent number of particles. The computation consists of each object exchanging particles with neighboring objects, and thereafter performing some computation. This continues for several iterations.

| Strategy | Default | Automatic |
|---|---|---|
| Non-Uniform | 7.93 | 2.51 |
| Uniform | 2.21 | 2.04 |

Table 3: Time in seconds for particle simulation program for the default (cyclic-cyclic) placement and automatically optimized (using runtime ORB) versions for a uniform and a non-uniform distribution of particles.

From the event graph for this program, Paradise was able to find that the grainsizes of objects varied significantly, and hence the program was classified as an irregular one. The phase number corresponding to the first synchronization point in the program was found, and a hint to perform Orthogonal Recursive Bisection (ORB) at that phase was generated by Paradise. The program is required to be modified by the programmer to set the load variable of the system base class "loadarray" in the constructor of each object. During the next run of the program, the runtime system automatically initiated the ORB library at the phase specified by Paradise. The ORB library accessed the load for each object and generated a partition of the parallel object array. This partition was encoded as a new mapping function. The ORB was followed by a remap operation using the new mapping function, after which the rest of the program was allowed to continue. Table 3 presents results for running the program on 16 processors of the CM-5. The simulation involved 1000 particles, distributed over a two dimensional parallel object array of size 16x16. The default mapping of the parallel object array was cyclic-cyclic. Results are presented for a uniform and a non-uniform input distribution. From the results it is clear that the ORB optimization which was automatically enabled by Paradise significantly improves performance for the nonuniform distribution, without introducing overhead for the uniform case.

# 7 Summary

In this paper we have described techniques for automating load balancing especially for irregular or dynamic applications. The specific contributions of this work are:

- Use of post-mortem analysis to infer program characteristics relevant for load balancing
- Development of heuristics for choosing load balancing schemes
- Evaluation of techniques on simple programs

Our results have shown that load balancing schemes can be automatically selected and incorporated into the application program, thus decreasing the effort required from application programmers for developing parallel programs with good performance. In future work, we intend to evaluate the optimizations using real applications, as well as broaden the set of optimizations and techniques in the automatic optimization framework.

## References


[1] Sanjeev Krishnan. *Automating Runtime Optimizations for Parallel Object-Oriented Programming.* PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, June 1996.

[2] Sanjeev Krishnan and L. V. Kale. Automating Runtime Optimizations Using Post-Mortem Analysis. In *Proceedings of the 10th ACM International Conference on Supercomputing, Philadelphia*, May 1996.

[3] W. Williams, T. Hoel, and D. Pase. The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D. In K. M. Decker and R. M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems.* Birkaeuser Verlag, Basel, Switzerland, 1994.

[4] B. Robert Helm and Allen Malony. Automating Performance Diagnosis : A Theory and Architecture. In *Proceedings of the International Workshop on Computer Performance Measurement and Analysis, Beppu, Japan*, August 1995.

[5] Barton P. Miller et al. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11), November 1995.

[6] Amitabh B. Sinha. *Performance Analysis of Object Based and Message Driven Programs.* PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, January 1995.

[7] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook.* MIT Press, 1994.

[8] K. Tomko and E. Davidson. Profile Driven Weighted Decomposition. In *Proceedings of the 10th ACM International Conference on Supercomputing, Philadelphia*, May 1996.

[9] L.V. Kale and Sanjeev Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, September 1993.

[10] L. V. Kale and Sanjeev Krishnan. Charm++ : Parallel Programming with Message-Driven Objects. *in Parallel Programming using C++*, MIT Press, 1996. To be published.

[11] L.V. Kale and Amitabh Sinha. Projections : A scalable performance tool. In *Parallel Systems Fair, International Parallel Processing Sympo sium*, April 1993.

[12] A. B. Sinha and L.V. Kale. A load balancing strategy for prioritized execution of tasks. In *Proceedings of the International Parallel Processing Symposium*, April 1993.

[13] L. V. Kale, Ben Richards, and Terry Allen. Efficient parallel graph coloring with prioritization. *Lecture Notes in Computer Science*, 1996. To be published.