Automating Parallel Runtime Optimizations Using Post-Mortem Analysis

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

Attaining good performance for parallel programs frequently requires substantial expertise and effort, which can be reduced by automated optimizations. In this paper we concentrate on run-time optimizations and techniques to automate them without programmer intervention, using post-mortem analysis of parallel program execution. We classify the characteristics of parallel programs with respect to object placement (mapping), scheduling and communication, then describe techniques to discover these characteristics by post-mortem analysis, present heuristics to choose appropriate optimizations based on these characteristics, and describe techniques to generate concise hints to runtime optimization libraries. 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

Developing efficient software for parallel computers has been recognized as one of the bottlenecks preventing more widespread use of parallel computers. This is because several new issues have to be tackled before parallel programs can attain the peak performance that parallel computers provide. Whereas simple parallel implementations often suffer from bad speedups and other symptoms of low performance, good performance can usually be attained only at a high programming cost. The challenge is thus to attain good performance at low programming cost.

Ideally, an expert parallel programmer would be able to anticipate and respond to all potential performance problems during the design of the first prototype. However, parallel programming skills are not widespread, and moreover, the programmer may not know enough about the characteristics of the application and implementation without investing considerable effort. The first prototype of a parallel program is hence likely to have several serious performance flaws, leading to active research for identifying performance problems and 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.*

Automating program optimizations using expert knowledge either in the compiler or the run-time system 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 without programmer intervention.

To appear in the 10th ACM International Conference on Supercomputing, May 1996, Philadelphia.

2 The Optimization Framework

This section motivates run-time optimizations, the use of post-mortem analysis for automating them, describes our framework, and relates it to previous work.

2.1 Need for run-time optimizations

Traditionally, research in optimization techniques has concentrated on compiler transformations of parallel programs for exposing parallelism, automatic grainsize control, determining data distribution, improving locality, reducing communication overhead, etc. The disadvantages of optimizations performed only at compile time are :

Compiler optimizations are static: they cannot take into account run-time conditions. On the other hand, a parallel computer presents an inherently variable environment: resource availability and message transmission have non-deterministic factors.

Unpredictability: Many parallel applications have unpredictable computational needs and communication patterns which cannot be inferred from a static analysis of the parallel program.

Static analysis is complicated: for parallel object-oriented programs based on C++ because of the difficulties of precise dependence and type analysis in the presence of pointers.

Separate compilation: If a parallel program is composed from separately compiled modules, the compiler does not have enough global information to optimize the program; e.g. load balancing decisions cannot be made by a module in isolation, since the load on a processor is affected by computations in all modules.

Library based parallel programming environments: (e.g. PVM and MPI) are collections of run-time libraries. For such programs, run-time optimization is the only way to get better performance.

Thus the compiler may have insufficient information to decide whether an optimization is necessary, how to do it, and when (at what point in the program) to do it. Optimizations performed at run-time can take care of many cases where compiler transformations are inadequate.

2.2 Paradise: Automatic Optimization

Run-time optimizations 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 *manually* analyse 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.

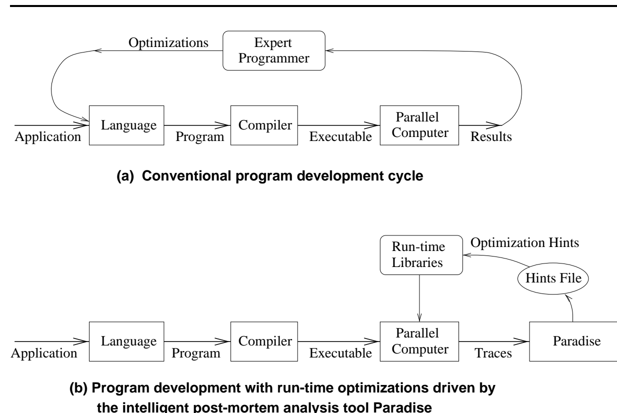


Figure 1: Parallel program development cycles.

Different executions of a parallel program are often very similar, if not identical, in their structure, communication patterns and performance characteristics. This leads us to observe that analyzing one or more executions of a program can give us information about performance problems and possible optimizations for the program. In a traditional program development cycle, all the tasks of analyzing performance data, identifying performance problems, designing optimizations, and incorporating them in the parallel program need to be done by the programmer. By incorporating expert knowledge in a post-mortem analysis tool, it is possible to do some of these tasks automatically and generate information for automatic compile-time as well as run-time optimization.

Most parallel programs needing optimization 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 analyzes 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 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".

2.3 Previous work

Performance analysis research has concentrated on visually displaying performance data [1, 2], relating performance data to high level language constructs [3, 4], or giving the user insights into performance problems using expert analysis [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.

Automatic compiler optimizations have achieved success for automatic data partitioning and communication schedule generation in array-based Fortran programs [7, 8]. The use of profile information for compiler optimizations is well-known. Several sequential compilers use profile information to predict branch probabilities (e.g. [9]). Some compilers for data-parallel languages such as HPF [10]) use profile information to accurately find the cost of various computation and communication operations. To the best of our knowledge, our framework is one of the first efforts towards using post-mortem analysis for automating run-time optimizations. Also, the scope of our optimizations is much broader than in traditional compiler optimizations, and moreover applies to dynamic 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) and scheduling of computations and managing communication between objects. Thus there are significantly greater opportunities and chal-

lenges for automatic optimizations. In contrast, message-passing layers such as PVM or MPI require the programmer to explicitly specify the placement and scheduling of computations and the communication between them, and also do not provide facilities for dynamic creation of tasks, thus restricting the extent of automatic optimizations. 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

This work is based on the parallel object-oriented language *Charm++*, [11, 12] 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 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* [2] 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.

3.1 Problems due to non-determinism

The validity of inferences drawn from post-mortem analysis of an execution depends on how similar subsequent executions are to that execution. For programs with completely deterministic behavior (such as some SPMD programs), or for executions with the same inputs, the inferences will be very accurate. Suggestions for optimization may be made in terms of individual message and object instances. For irregular or non-deterministic programs, variations in the input parameters can affect program execution, so that fewer inferences will be valid. In such cases we need techniques to specify properties of a set of instances (e.g. all messages of a particular type in the source program), because we cannot make suggestions at the level of individual instances. Essentially, we need to infer program-level characteristics (e.g. whether the program has graph-structured or tree-structured communication) by analyzing an execution.

Completely deterministic programs always produce the same set of tasks and messages. However, most parallel programs contain non-determinism because of the following factors:

- Inputs: Many parallel programs have different numbers of tasks and messages depending on the input set, e.g. when the size of the input problem changes. Sometimes even the kind of computations performed change depending on input : such cases are extremely difficult to handle.
- Scheduling: Some programming models adaptively schedule computations to overlap them with communication and also to adjust for varying run-time conditions. Thus the order of execution of computations on a processor may change from run to run.
- Placement: Dynamic object placement strategies place objects on processors depending on run-time conditions, hence the location of computations may change from run to run.
- Granularity: If the run-time system provides dynamic granularity control, the number of parallel objects created may differ from run to run, although the total amount of computation remains the same.
- Speculative computations: Some applications (such as those involving search) create speculative work, the amount of which depends on run-time conditions.

In this paper we do not tackle the last two problems (non-determinism due to dynamic granularity control and speculative work). We target our techniques to handling different input sets as long as the basic computation steps do not change : i.e. only the numbers of objects and messages may change, not the structure of the parallel program or the kinds of computations performed. This is achieved by not generating any execution-specific optimization hints which depend on individual message or object instances. Instead, the optimization hints are in terms of object and message types (e.g. "all messages of type T are to be given priority level 1").

To deal with non-determinism due to adaptive scheduling and dynamic object placement, we need to ensure that the program representation remains invariant even if there is such non-determinism. As described in the previous subsection, the event graph contains edges between methods for dependences : this ensures that the graph for operations such as a "join" — where a method executes only if several preceding methods have executed — does

not change even if the order of execution of the preceding methods changes. Also, by grouping vertices belonging to the same object instance, we get an *object-interaction graph* which is independent of the exact assignment of objects to processors, thus removing the effects of non-deterministic placement. Moreover, this allows us to analyze the interactions between object instances, instead of restricting ourselves to a less precise processor-level. This is very useful when we try to analyze patterns of communication between objects.

4 Optimizing 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.

4.1 Characteristics of parallel programs affecting object placement

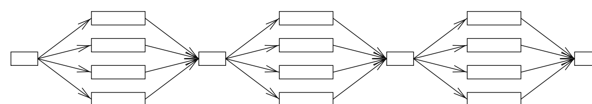
In order to select a suitable object placement strategy, Paradise attempts to systematically discover the characteristics of the parallel program from the event graph. We first describe some important characteristics which affect object placement, and discuss how they are inferred from the event graph.

4.1.1 Phase structure of program

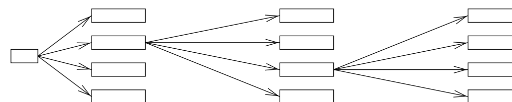
This characteristic tells us if there are repeatedly occurring phases in the program. Separating the program into independent phases helps to focus analysis [6], because each phase can be analyzed independently. The phase structure is also important if all objects are not active in all phases: ignoring the phase structure may result in load imbalance within a phase in which only part of the objects are active. The following types of phases are commonly found in parallel programs:

- Phases separated by synchronization points: here each phase proceeds only after all computations in the previous phase have finished (figure 2a).
- Phases separated by *initiation points*: here there are no synchronization points, instead each phase is initiated by a multicast from one object (figure 2b), which forms an initiation point.
- Computation phases alternating with communication phases: (figure 2c) this is typical in a data-parallel, loosely synchronous program, where all processors perform the same computation (of same size) on their own data, then send and receive data from other processors.
- No phases: the program does not exhibit a repeating communication pattern (figure 2d). This may arise because phases in the program overlap with each other so that separate phases cannot be discerned.

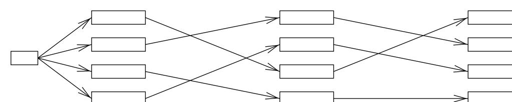
For a program which has phases we need to make sure that the load in each phase is balanced; this usually leads to additional constraints on assignment of objects to processors which need to be considered by the object mapping strategy.



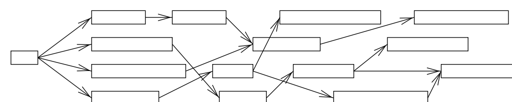
(a) Phases with synchronization points



(b) Phases with initiation points



(c) Loosely synchronous (compute-communicate) phases



(d) No phases

Figure 2: Different types of phases in parallel programs.

4.1.2 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. Whether data locality can be exploited for a particular program or not depends on the inherent communication patterns for the program. Most programs fall into one of the categories described below.

Locality is unimportant: There are two cases when locality need not be taken into consideration while mapping objects to processors. The first is when all objects execute independently, without communicating with each other. This is detected in the object-interaction graph when the total number of edges is close to the total number of nodes

(since each object must have at least one edge connecting it with its creator object). The second is when each object communicates with almost every other object (e.g. when all objects do a broadcast), so that there is no locality in the communication pattern. This is detected when the average degree of a node in the object-interaction graph is close to the total number of nodes. If no locality exists in the program, object mapping needs to only take care of load balance, as determined by the other characteristics.

Tree-structured communication : Here each object communicates only with its parent (creator) object or its child objects. This property is detected in the object-interaction graph by a simple graph traversal.

Graph-structured communication : arises when objects communicate with neighboring objects. Usually this pattern of communication is found in spatially decomposed applications, where objects represent regions of space and communicate with adjacent regions. Regular graph structures arise when the communication pattern between objects is regular (e.g. neighbor communication among objects in a parallel object array). Input dependent graph-structured communication occurs when the exact nature of the object-interaction graph depends on the input data (e.g. when each object constructs a list of interacting objects using input data).

4.1.3 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).

4.2 Optimizing static object placement for programs without phases

In static object placement, objects are created and placed only at the beginning of the program. Paradise currently chooses static object placement strategies for programs written using Charm++'s parallel object array construct. Essentially, Paradise chooses a strategy which partitions the array among processors so as to balance load as well as reduce inter-processor communication. When the communication pattern matches with commonly observed regular communication patterns (e.g. a grid-based decomposition where each object communicates with the two adjacent objects in each dimension), the partitioning strategy need not be dynamic (i.e. it need not execute at runtime). In such cases it is possible for Paradise to automatically infer a precise pattern for mapping objects to processors.

The first task to be done for array partitioning is to align interacting object arrays of the same dimensions. The strategy here is the one used by Li & Chen [13] and Gupta & Banerjee [7] for compilers, involving construction and partitioning of a graph whose vertices represent array dimensions. Each set of aligned arrays represents a template grid (as in HPF), and each object in each array is assigned to a grid point.

Next, Paradise partitions the template grids across processors. For regular array-based programs without phases, Paradise uses a (block, block, ...) pattern, where processors are arranged in a grid and a contiguous block of array elements is placed on each processor. For each template grid, the amount of communication along each dimension is calculated by creating a plane perpendicular to the dimension which bisects the grid, and finding the number of messages crossing the plane. The amount of communication is used to determine the block size in that dimension : the more the communication, the larger the block size. The final hint generated specifies the aspect ratio of the processor grid. At run-time, the object mapping library uses this aspect ratio and the array size to determine the block size and assign array elements to processors. E.g. for a Jacobi relaxation program where each object communicates with its two adjacent objects in each dimension, Paradise suggests square regions for each processor. E.g. for an array where communication occurs only along one dimension, processors will get long slabs oriented along the direction of communication so that there is no inter-processor communication.

4.3 Optimizing static object placement for computations with phases

The phase structure of a program is important only if object activity patterns vary from phase to phase. Paradise first constructs a list of phases in which a significant number of objects are not active. It then generates a set of load balance constraints on object mapping, and evaluates the suitability of well-known mapping patterns with respect to the constraints.

A constraint is specified between a pair of objects to indicate that both objects should preferably not be assigned to the same processor. Within each phase, constraints are attached to every pair of objects. Thus satisfying as many constraints as possible would ensure that the objects are evenly distributed across processors. In future we intend to automate object migration to allow remapping of objects between phases, if there are conflicting constraints on object placement across phases.

Finding patterns of object placement:
It is desirable to express the mapping of objects to processors concisely so that it can be communicated to the runtime libraries as a hint. Ideally, the mapping should

be an expression which maps the object's id to a processor number. For parallel object arrays in Charm++, the expression uses the coordinates of an object in the 1, 2, or 3-dimensional object array to map it to a processor. Currently, the mapping patterns supported include :

- all block-cyclic mappings. E.g. for a 2-D array, this has the form
  $Map(i,j) = \frac{i}{a} MOD\ b + b * (\frac{i}{c} MOD\ d)$
- the multi-partition mapping scheme [14]. E.g. for a 3-D array this has the form
  $Map(i,j,k) = \frac{j-k}{a} MOD\ b + b * (\frac{i-k}{c} MOD\ d)$

In the above expressions i, j, k are the coordinates of an object, and a, b, c, d are the constants that need to be found by the analyzer. We intend to add more patterns as the need arises.

4.3.1 Placement for Gaussian Elimination

We demonstrate automatic object placement for a simple program which performs Gaussian Elimination (only the triangularization step, without pivoting). The Charm++ program for this has an object for each row in the matrix, and a parallel array of row objects which constitutes the matrix. When a row becomes the pivot row, it multicasts its elements to all rows below it in the matrix.

From the event graph, Paradise was able to deduce that there is no locality in the communication pattern (since each row object needs to communicate with all other rows). Further, the program has phases separated by initiation points corresponding to the multicasts made by the pivot rows. The number of objects active in each phase varies, thus it is necessary to balance load in each phase separately. Hence Paradise maps objects to processors phase by phase. The mapping pattern which matches the load-balance constraints best corresponds to a cyclic mapping. Paradise now generates a hint (consisting of the values of the constants in the mapping expression) to the object-mapping module in the run-time through the optimization hints file. Table 1 presents times in seconds for running the Gauss-Elimination program on 32 processors of the CM-5 for a 1000x1000 matrix, with a random object mapping strategy and the automatically optimized strategy.

Strategy	Random	Automatic
Time (sec)	38.72	34.90

Table 1: Object mapping for Gauss-Elimination.

4.4 Optimizing dynamic object placement for tree-structured programs

Dynamic load balancing is an extensively researched area, and there are several general-purpose as well as application-specific strategies. If we know the problem structure, we can sometimes use better-tuned strategies. Recognizing this, Paradise uses a special load balancing strategy for tree-structured computations where there is some amount of uniformity in the tree. In tree-structured programs, each object communicates only with its parent (creator) object or its child objects.

The load balancing strategy for tree-structured computations essentially tries to partition the tree equally across all processors by assigning a subtree to each processor. This maintains locality, results in better load balance, distributes the tree across processors quickly, as well as avoids other overheads of general-purpose load balancing strategies. The types of trees which can be partitioned by this strategy are :

- Completely uniform trees : every internal node has the same branching factor, and all its child subtrees have the same load. This kind of tree may occur in exhaustive search applications.
- Uniform branchfactor : every internal node has the same branching factor, and child subtrees have a consistent ratio of loads.
- Uniform subtree load : all subtrees at an internal node have the same load, and the branching factor varies as a linear function of the depth.

The strategy assigns a set of processors to work on each internal node of the tree (the root is initially assigned the whole processor set). Each internal node divides its processors among its child subtrees, based on their loads as predicted by Paradise. This process continues until a subtree has just one processor assigned to it. Thereafter all nodes in the subtree are kept local.

Paradise first does a traversal over the object-interaction graph to determine if the program is tree-structured. It then systematically traverses the tree in a depth-first manner, calculating the loads of all leaf and internal nodes. Then it analyses the branching factors and subtree loads to check if they are uniform. If so, it generates an appropriate hint.

If the tree is too irregular, Paradise reverts to one of the general purpose load balancing strategies provided with the Charm++ run-time system: currently it chooses between a round-robin strategy (objects are placed on processors in a round-robin manner), neighbor averaging (load is balanced among processors in a neighborhood) and hierarchical manager (a cluster of processors is controlled by a manager). The choice is made depending on the grain-sizes of objects (e.g. round-robin will not work well if the grain-sizes are widely varying), the amount of locality needed (e.g. neigbor-averaging works better when locality is needed), and the number of objects to be balanced.

4.4.1 A uniform tree structured problem

We demonstrate automatic dynamic object mapping for tree-structured programs using a simple parallel program for computing the factorial of a large number. The program has essentially a divide-and-conquer structure. Each node in the divide-and-conquer tree is a Charm++ object, which creates two child nodes for computing partial products. The child nodes reply back to the parent when the result is found. This program was compiled and run on the TMC CM-5, the traces of program execution were collected and fed into Paradise. Paradise was able to deduce that the program has a uniform uniform tree structure (all leaves are at equal depth), and a branch factor of 2 at

every internal node. Moreover, the object creation is contінous and there is no phase-structure. Hence Paradise chose the object mapping strategy for trees, and also provided the additional information that the tree was regular and binary.

The hints generated by Paradise were fed back to the object-mapping module using an optimization hint file. The factorial program was again run on the CM-5 to get the optimized results. Table 2 gives times (in milliseconds) for computing the factorial of 4096 on 16 processors (which results in 8191 objects in the tree). Results are presented for a random object mapping strategy and the automatically optimized tree strategy.

Strategy	Random	Automatic
Time (mS)	684	163

Table 2: Dynamic object mapping for factorial program.

5 Scheduling Optimizations

The scheduler in the run-time system determines the order of execution of method invocations in objects on a processor. This order becomes important when there are multiple concurrent objects on a processor, any of which can be allowed to execute. The main aim of the scheduling strategy is to ensure that all computation paths[1] in the parallel program finish at about the same time. In programs where some computational paths are longer than others, there are likely to be one or more critical paths. We can optimize the scheduling by executing tasks on critical paths with minimal delay. This is a difficult problem because no global information is normally available: the scheduler does not know which of the methods to be invoked lies on the critical path.

Scheduling may be affected through prioritization: objects or messages can be assigned a priority value, which is used to select a message for processing when there are many messages in the scheduler queue. Charm++ already allows the programmer to assign a priority to a message before sending it. To implement prioritization automatically, the scheduler needs to know what priorities need to be assigned to messages, and at what times in the program.

Paradise first determines if the program has one or more significant critical paths. The algorithm used to determine critical paths is based on a longest-path heuristic: a depth-first traversal of the event graph reveals the longest path from each node to the end of the program. Paradise then determines the method types or message types lying on the critical paths and the priority values to be assigned to them. The priority of a type is higher if it occurs more often on critical paths, and lower if it occurs more often on non-critical paths. Further, Paradise determines which objects are on critical paths and assigns priorities to objects. The priority of an object is higher if it occurs more often on the critical path, and also higher if it occurs earlier on the critical path. Then Paradise tries to find a pattern for relating the object-id to the priority, using a linear priority expression.

[1] A path through a parallel program corresponds to a path in the event graph for the program. The length of a path includes the computations and communication delays on that path.

5.1 Prioritizing Gauss-Elimination

The Gauss-Elimination program described in section 4.3.1 has a critical path consisting of the initiation points when a row broadcasts itself to all rows below it in the matrix. Each object occurs exactly once on the critical path, and object id (row number) i occurs before object $i+1$ on the critical path. Thus Paradise deduces that objects should be assigned a priority based on their object ids. The priority expression matching routine now tries to relate the object-id to its priority. Since Charm++ assigns higher priority to lower numbers, the priority of an object can be set to its row-id. This hint is generated by Paradise and read in by the run-time system during the next run of the program. The performance of the program with and without prioritization is given in table 3.

Strategy	No Priorities	Automatic Priorities
Time (sec)	43.58	34.90

Table 3: Prioritization for Gauss-Elimination

6 Communication Optimizations

Communication optimizations are necessary to reduce communication volume and overheads, as well as tolerate latency. Here we discuss two of them that are commonly used in parallel programs.

Message pipelining

Instead of sending a large message at the end of a large computation to an idle processor, it is possible that by sending pieces of the message earlier, the idle processor could start processing the pieces earlier, thus reducing idle time by overlapping communication and computation. However, in general, the choice of the number of pieces the computation should be divided into (the degree of pipelining) must be made at run-time, depending on the communication latencies and bandwidth of the target machine, as well as the sizes of the messages and computations in different phases of the program. In order to enable this pipelining optimization, the methods to be pipelined must be parameterized so as to give the run-time system a control point where the degree of pipelining parameter can be set. Paradise determines the degree of pipelining for different message types using heuristics which reduce idle time and overheads [6].

Message aggregation or combining

When a processor sends many small messages to the same remote processor and the messages are not immediately processed, the messages can be combined into one large message before being sent out, and then broken up after being received. This reduces message transmission overhead. When such a message is sent by a processor, the run-time at the sender needs to know whether to send the message to the receiving processor right away or delay the

message so that it can be combined with another message to the same receiver. In this situation Paradise provides valuable hints regarding which messages to combine, and at what times in the program.

We give an example of a simple parallel data-collection operation which benefits from message combining. The operation involves a parallel object array with a large number of elements distributed over all processors, each of which sends messages to one object on processor 0. Paradise detects this operation, and generates a hint to the runtime for enabling message combining for the particular message type. The number of messages to combine is set to the number of objects of the array on a processor. Note that the runtime uses this number only as a hint: a time-out value is used to limit the maximum amount of time a message may be delayed in order to wait for another message to the same processor. Table 4 gives results for the total time taken for broadcast to and data-collection from 512 objects on 32 processors of the CM-5, with each object sending a 4 byte message to processor 0.

Strategy	No Combining	Automatic Combining
Time	182	88

Table 4: Time (in milliseconds) with and without message combining for broadcast and data-collection over 512 objects on 32 processors.

7 Summary

In this paper we have described techniques for automating parallel run-time optimizations using post-mortem analysis. The specific contributions of this work are:

- automation of a broad set of run-time optimizations for object placement, load balancing, scheduling and communication, without user involvement.
- systematic classification of the characteristics of parallel object-oriented programs and run-time techniques for optimizing programs with those characteristics.
- development of heuristics for automatic post-mortem analysis of program traces for (a) discovering program characteristics, (b) choosing optimizations, and (c) generating concise hints to runtime optimization libraries.
- evaluation of heuristics on simple programs.

Our results have shown that run-time optimizations can be automated, 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 as the need arises.

References

[1] D. A. Reed et al. Scalable Performance Analysis : The Pablo Performance Analysis Environment. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, 1993.

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

[3] V. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, J. Wang, and D. A. Reed. An integrated compilation and performance analysis environment for data-parallel programs. In *Proceedings of Supercomputing 1995*, December 1995.

[4] B. Mohr, D. Brown, and A. Malony. TAU: A Portable Parallel Program Analysis Environment for pC++. In *Proceedings of the 3rd Joint Conference on Parallel Processing: CONPAR 94 - VAPP VI*, September 1994.

[5] J. Kohn and W. Williams. ATExpert. *Journal of Parallel and Distributed Computing*, 18:205–222, 1993.

[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] P. Banerjee et al. The PARADIGM Compiler for Distributed-Memory Multicomputers. *IEEE Computer*, October 1995.

[8] S. Sharma, R. Ponnusamy, B. Moon, Y. Hwang, R. Das, and J. Saltz. Run-time and compile-time support for adaptive irregular problems. In *Proceedings of Supercomputing 1994*, November 1994.

[9] P. P. Chang, S. Mahlke, W. Chen, N. Warter, and W. Hwu. IMPACT : An Architectural Framework for Multiple-Instruction Issue Processors. In *Proceedings of the 18th International Symposium on Computer Architecture*, May 1991.

[10] 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.

[11] 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.

[12] L. V. Kale and Sanjeev Krishnan. Charm++ : Parallel Programming with Message-Driven Objects. *in Parallel Programming using C++*, MIT Press, 1995. To be published.

[13] J. Li and M. Chen. Index domain alignment : Minimizing the cost of cross-referencing between distributed arrays. In *3rd Symposium on the Frontiers of Massively Parallel Computation*, October 1990.

[14] N. H. Naik, V. K. Naik, and M. Nicoules. Parallelization of a class of implicit finite difference schemes in computational fluid dynamics. *International Journal of High Speed Computing*, 5(1), 1993.