AUTOMATING RUNTIME OPTIMIZATIONS FOR PARALLEL OBJECT-ORIENTED PROGRAMMING

BY

SANJEEV KRISHNAN

B.Tech., Indian Institute of Technology, Bombay, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

Abstract

Software development for parallel computers has been recognized as one of the bottlenecks preventing their widespread use. In this thesis we examine two complementary approaches for addressing the challenges of high performance and enhanced programmability in parallel programs: automated optimizations and object-orientation. We have developed the parallel object-oriented language Charm++ (an extension of C++), which enables the benefits of object-orientation to be applied to the problems of parallel programming. In order to improve parallel program performance without extra effort, we explore the use of automated optimizations. In particular, we have developed techniques for automating run-time optimizations for parallel object-oriented languages. These techniques have been embodied in the Paradise post-mortem analysis tool which automates several run-time optimizations without programmer intervention. Paradise builds a program representation from traces, analyzes characteristics, chooses and parameterizes optimizations, and generates hints to the Charm++ run-time libraries. The optimizations researched are for static and dynamic object placement, scheduling, granularity control and communication reduction. We also evaluate Charm++, Paradise and several run-time optimization techniques using real applications, including an N-body simulation program, a program from the NAS benchmark suite, and several other programs.

To my parents,

Table of Contents

Chapter

List of Tables

List of Figures

Chapter 1

Introduction

Parallel computers provide several advantages over traditional single-processor computers and mainframes. In particular, parallel computers are used because of their ability to provide better performance, reliability, spatial distribution and cost. Massively parallel computers constructed from commodity components hold the promise of large amounts of processing power at a low cost. They make possible the solution of computationally intensive problems in commercial and scientific application domains in a fraction of the time that would otherwise be taken.

In the last few years, several parallel computer platforms have become popular. Massively parallel computers such as the Intel Paragon, IBM SP, Cray T3D and TMC CM-5 provide hundreds or thousands of processors coupled by a low-latency, high-bandwidth interconnection network. Symmetric multiprocessors and high-end workstations provide more modest parallelism; they typically provide shared memory and a high-speed bus to connect processors. Clusters of workstations connected by networks based on Ethernet or ATM have also become very popular as a means of getting parallelism at low cost; they usually have higher communication latencies and lower bandwidths.

This thesis is focussed on issues in developing software for parallel computers. The next few sections present the key problems encountered in parallel software development and the approaches taken in this thesis for their solution.

1.1 Parallel software development is difficult

Software development for parallel computers has been recognized as one of the bottlenecks preventing their widespread use. This is because it is difficult for real parallel programs to attain the peak performance that parallel computers provide. Several new issues have to be tackled in a parallel program before peak performance can be achieved. The steps to be performed while designing a parallel program are:

- Decomposition: this is the process of breaking up a program into many tasks, each representing a piece of work which can be executed in parallel with the others. The amount of parallelism produced depends on the number of tasks that can execute independently. The two main issues that need to be kept in mind while decomposing a program are:
  - Task Granularity: this refers to the amount of work in a task (time taken to complete the task). For a given program, a small average task granularity leads to a large number of tasks, and a large task granularity leads to a small number of tasks. Task granularity thus affects the amount of parallelism in the program (a smaller granularity leads to a large number of tasks, and thus more parallelism), and the amount of overhead (each task is associated with a certain amount of creation and communication overhead, hence a large number of tasks leads to larger overhead).
  - Overlap with communication latencies: if there is one task per processor, and a task has to wait for remote data, its processor idles for the duration of the communication latency. This may be prevented by having more than one task per processor: when one task is waiting for remote data, its processor may context switch to other tasks,

thus overlapping the communication latency with useful work (assuming the context switch times are small compared to communication latencies).

- Mapping: after a program has been decomposed into tasks, the assignment of these tasks to processors must be determined. Task mapping or placement may be static (the placement of tasks is determined at the beginning of the program and not changed thereafter), or dynamic (tasks may be continuously created and mapped to processors, as well as migrated between processors after they have started execution). The two main aims of task mapping are:
  - Load balance: in order to increase processor utilization, tasks should be assigned to processors so that all processors have the same load.
  - Data locality: since remote data access latencies are significantly higher than local data access, tasks should be assigned to processors so as to reduce the amount of inter-processor communication. Groups of tasks which communicate heavily between themselves should thus be located on the same processor.
- Scheduling: this determines the order of execution of tasks on a processor, if there is more than one task. The issues affecting scheduling are:
  - Critical paths: if the program has one or more critical paths, tasks on the critical should preferably be scheduled ahead of other non-critical tasks.
  - Cache locality: tasks which reference the same data should be scheduled close to each other to increase the likelihood of reusing cache data.
- Machine-specific implementation: different parallel machines provide different architectural features, programming interfaces and tools, thus making it necessary to express the parallel program differently. For example, architectural support for shared memory

between processors makes it easier to write a parallel program using the shared memory programming model.

1.2 Problems: performance and programmability

From the preceding section it is clear that parallel programming requires the programmer to deal with many complex issues which are not encountered in the traditional sequential programming context. The process of designing a parallel program requires the programmer to create a strategy to deal with each of the four issues discussed above, after carefully evaluating choices.

The characteristics of parallel applications also crucially affect design decisions. As the application base for parallel computers has widened, programmers have to deal with a large space of possible program behaviors. Some of the characteristics that affect parallel program performance include communication patterns, the number and sizes of tasks created, the dependence structure between tasks, and the extent of irregularity or variance in these characteristics over the course of program execution. In order to develop a parallel program with good performance, the programmer should know about the application characteristics.

The parallel programming skills required to discover application characteristics and make good design decisions are not widespread among the application programmer community. This is because application designers are looking to parallel computers as a means of solving their problems faster, and may not be willing to invest the resources and time required to develop parallel programming skills. As a result, the user base for parallel computers has remained restricted.

From the preceding discussions, we can identify the following two key problems in parallel programming, which form the motivation for the work in this thesis:

- How to make parallel software development easier ?
- How to attain good performance for parallel programs ?

The above two problems are related: whereas simple parallel implementations often suffer from bad speedups and other symptoms of bad performance, a high programming cost is incurred for attaining good performance. The challenge is thus to attain good performance at low programming cost. The next two sections describe the two complementary approaches taken in this thesis.

1.3 Solution: object-oriented parallel programming

In this thesis we have used object-orientation as a means of addressing the first problem, viz. ease of parallel software development. Object-oriented programming has achieved considerable success in the sequential programming domain as a means of making software development easier. An object consists of a set of functions and their associated data. The key features provided by object-oriented languages are:

- Abstraction: an object provides a well-defined external interface which *encapsulates* implementation details, thereby isolating the complexity within the object from the external world. Thus an object provides a higher-level abstraction which allows us to separate *what* operation is done from *how* it is done.

- Code reuse: an object may *inherit* the functions and data of another object, as well as redefine some of them, thus allowing an already available object to be reused and customized. Moreover, *polymorphism* allows the same code to operate on different data types. Thus object-oriented languages encourage software reuse.

The introduction of object-orientation in parallel programming leads to easier parallel software development, and helps to address the problem of better programmability in parallel software:

- Object-orientation allows the programmer to encapsulate low-level mechanisms and manage the increased complexity of parallel programming. Programmers may program at a higher level of abstraction. E.g. the mechanism of message sending in a parallel program may be represented as a method invocation. Code reuse is even more important for parallel programs because of the higher cost of developing modules.

- The concept of a task or piece of work in parallel programming is elegantly represented by an object. Object-orientation thus provides an easier approach for decomposing a program into independent encapsulated concurrent objects.

Object-oriented parallel programming has been explored in several systems over the past few years. As part of this thesis we have designed and implemented **Charm++**, which was one of the first parallel object-oriented languages based on C++. Chapter 2 describes the language and its unique features in detail.

1.4 Solution: automated optimization tools

A typical parallel program development cycle includes the following steps (Figure 1.1a):

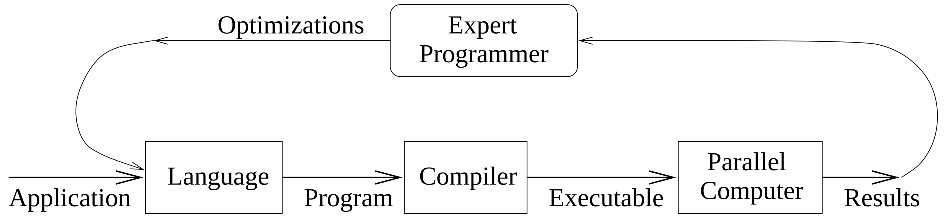
1. Program design and coding: the programmer develops an appropriate parallel algorithm, and codes the first prototype using a parallel language.

2. The program is run on a parallel computer, debugged, and typically the speedups for various problem sizes are observed. If the performance is acceptable, the process terminates here.

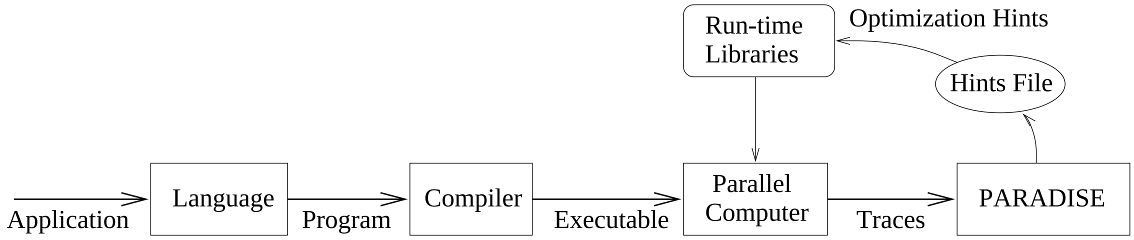
3. To identify the causes for performance loss, the programmer uses a performance analysis tool or writes custom code to generate performance data.

4. Optimizations to solve the performance problem are found ; techniques or algorithms to carry them out are identified or developed.

5. The parallel program is modified to implement the optimizations for solving the performance problem. Now the development cycle iterates back to step 2.



(a) Conventional program development cycle



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

Figure 1.1: Parallel program development cycles.

Ideally, an expert parallel programmer would be able to anticipate and respond to all potential performance problems in step 1 itself. 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 steps 3, 4 and 5. 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 in the compiler, run-time libraries or other tools can hence 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. In this thesis we have designed and implemented the **Paradise** (PARallel programming ADvISEr) post-mortem analysis tool [1] which can be used for automating run-time optimizations without programmer intervention (Figure 1.1b). It feeds optimization hints directly into the Charm++ run-time system, thus resulting in improved performance for the parallel program. Paradise is one of the first tools that automates a wide range of run-time optimization techniques for static object placement, dynamic load balancing, granularity control, scheduling, and communication reduction. Chapters 3 and 4 describe the design of Paradise.

1.5 Contributions of thesis

In this thesis, we have explored two complementary approaches for tackling the twin challenges of performance and programmability in parallel programs:

- Parallel object-oriented languages: We have developed Charm++, which was one of the first C++-based parallel object-oriented languages.

- Automatic optimization techniques: We have developed techniques for automating run-time optimizations for parallel object-oriented languages. These techniques have been embodied in the Paradise post-mortem analysis tool. Specific contributions here include:

  - a classification of known run-time optimizations and development of new ones
  - techniques for deriving program characteristics from a program representation
  - heuristics for selecting optimizations depending on program characteristics
  - techniques for generating concise optimization hints
  - development of run-time libraries which use optimization hints to improve program performance

The objective of this thesis is to verify the hypothesis that runtime optimizations lead to improved performance and that they can be automated.

The key features that distinguish our work on automating runtime optimizations from other work on performance analysis and optimizations are:

- Whereas most automated performance analysis tools only concentrate on visualization of data and reporting performance problems, Paradise goes one step further in the direction of automation by *solving performance problems through optimizations*, without programmer intervention.

- Most previous tools target loop-based or regular loosely-synchronous programming models. Paradise works for irregular, dynamic, asynchronous parallel object-oriented programs, for which there are significantly more opportunities and challenges for automatic optimization.

- Paradise automates *runtime* optimizations whereas most previous work in automatic optimizations has concentrated on compiler transformations.

We have also evaluated Charm++, Paradise and several run-time optimization techniques using real applications, including an N-body simulation program, a program from the NAS benchmark suite, and several other simple programs. The results of this evaluation, presented in Chapter 5, demonstrate that run-time optimizations can improve performance for specific applications and that they can be automatically applied using intelligent post-mortem analysis.

1.6 Dissertation outline

Chapter 2 describes the design and implementation of Charm++. In Chapter 3 we discuss the design of the framework for automating runtime optimizations and compare it with previous work. In Chapter 4 we discuss strategies and mechanisms for run-time optimizations, and specific techniques for automating them. Chapter 5 presents the application programs used to evaluate Charm++ and Paradise. Finally, in Chapter 6 we summarize the thesis and present directions for future work.

Chapter 2

Charm++

Charm++ [2] is a parallel object-oriented language based on C++. It was one of the first few C++-based parallel languages when developed in 1992–93. Charm++ enables the application of object orientation to the problems of parallel programming. Most of its features are based on those in the C-based parallel programming language Charm [3]. Charm++ programs can run today on most MIMD computers, including shared- and distributed-memory machines such as the Intel Paragon, TMC CM-5, IBM SP-2, nCUBE/2, Convex Exemplar, Sequent Symmetry, Encore Multimax, and workstation networks. This chapter describes its design philosophy, essential features, syntax and implementation.

2.1 Design philosophy

Charm++ aims to reduce the complexity of parallel program development by addressing the following issues:

Portability: Portable programs are ones which can run unchanged on different machines, while maintaining performance close to native implementations. Portability is necessary for applications that need to run on multiple platforms and for protecting the investment in parallel software when the underlying parallel machine is upgraded or changed.

Latency Tolerance: Remote data usually takes more time to access than local data on scalable parallel machines. Moreover, the arrival of remote data can be further delayed due to message contention, or to computations being done on remote processors. The magnitude and unpredictability of these latencies often cause significant performance loss; avoiding them often requires contorted program logic. A parallel programming system should make it possible to tolerate communication latency with minimal additional programming effort.

Support for Irregular and Dynamic Problem Structures: There has been much research into applications and languages based on the data-parallel and SPMD programming models, in which problem structures are regular and static (i.e., do not change over time). However, the best algorithms for many seemingly regular problems often involve irregular or asynchronous structure. Moreover, application domains such as irregular finite-element computations, adaptive grid refinement, discrete event and N-body simulation, AI search computations, and discrete optimization are inherently irregular. A general-purpose parallel language should therefore efficiently support irregular problem structures and asynchronous behavior. In particular, it is necessary to support dynamic creation of work and dynamic load balancing.

Modularity and Re-use: The inherent complexity of parallel programming implies that re-use of software modules will be even more cost-effective than it has proved for sequential software. However, re-use in a parallel context is fraught with challenges:

- Modularity should not lead to a loss of efficiency. In particular, latency tolerance by overlapping computations in one module with idle time in another should be possible.
- Modules written in a parallel language must be able to interoperate with modules written in other languages.

- Modules must be able to exchange data in a fully distributed fashion, without any centralization.

2.1.1 Automation of programming steps

As discussed in Section 1.1, developing a parallel application involves the four tasks of decomposition, mapping, scheduling and machine-specific implementation.

Various approaches to parallel programming can be characterized by the extent to which these tasks are automated by the programming system and by the generality of their approaches. A low-level portable layer such as PVM [4] only provides machine-independent implementation (thereby automating machine dependent implementation), but is sufficiently general to be useful in most problem domains. A programming system such as HPF [5] automates scheduling, but allows the programmer to specify the mapping of computations to processors and directly supports only data-parallel applications. Parallelizing compilers [6] attempt to automate all four tasks, but have proved effective only for regular problems having loop-based parallelism.

Charm++ is a general-purpose language that tries to automate the details of mapping, scheduling and portability. This enables a natural division of labor between the programmer and the system: programmers, with their understanding of the application and algorithm, specify the decomposition of the computation into parallel actions, while the system implements resource management and scheduling. In the object-oriented paradigm, decomposition arises as a natural consequence of creating object classes to encapsulate computations.

Implications of automation for tools:

The advantage of automation for programming tools such as compilers, run-time libraries and post-mortem analyzers is that they have more flexibility in influencing or guiding the automated steps, because they are given the responsibility and control over those functions. In a programming system with little automation (e.g. PVM or MPI [7]), the programmer explicitly specifies decomposition, mapping and scheduling; programming tools do not have access to information about those tasks and cannot influence them without help from the programmer. On the other hand, Charm++ is a language that is *designed for automation.* It automates mapping and scheduling while leaving decomposition to the programmer. Because of this design, the automatic optimization tool Paradise (Chapter 4) is able to incorporate several optimizations for mapping and scheduling to improve the performance of Charm++ programs without programmer intervention.

Although Charm++ automates mapping and scheduling, it allows programmers to override mapping explicitly and also lets them influence scheduling via prioritization.

2.1.2 Evolution

Several of the parallel programming concepts in Charm++ were first developed in the Charm language and the Chare Kernel runtime system between 1987 and 1991 [3, 8, 9, 10, 11]. Charm is an extension of C which supports parallel objects called *chares*. It was one of the first parallel languages to support a message-driven style of programming. The Chare Kernel runtime system supported dynamic object creation, dynamic load balancing, and prioritized message-driven scheduling in a portable manner. Charm++, which was developed in 1992-93 [12], fully incorporates object-oriented features such as inheritance and polymorphism into the Charm model and adds other abstractions such as parallel object arrays. Recently, the Chare Kernel was modified to operate on top of Converse [13], a machine-independent layer (described in section 2.3.1) which supports interoperability by allowing modules from multiple languages to co-exist in a single application.

2.2 The Charm++ language

In this section we discuss the essential features of Charm++. The first two features—message-driven execution and dynamic object creation—had been defined and explored in work on *actors* [14] done prior to Charm++. The contribution of this work is in developing one of the first pragmatic and portable implementations of object-oriented message-driven execution in the framework of C++, and the addition of higher level abstractions such as parallel object arrays.

2.2.1 Message-driven execution

In message-driven execution all computations are initiated in response to the availability of messages. In Charm++, messages are directed to a method inside an object. Messages received by a processor are stored in a pool; the system scheduler repeatedly chooses a message from this pool, then invokes the indicated method within the destination object. Sending a message to an object thus corresponds to an asynchronous remote method invocation. The scheduler allows the method to run to completion before selecting another message from the pool (this dictates that the code in the methods be non-blocking).

Message-driven execution, combined with an asynchronous (non-blocking) model of communication, enables latency tolerance by overlapping computation and communication adaptively and automatically, including across modules. Each processor has multiple active objects, some of which have messages waiting for them in the runtime system. Remote operations (such as fetching remote data) are done in a split-phase manner as follows: an object initiates the remote operation from code within a method by sending a request to the remote processor and returns control from the method to the scheduler. The scheduler can then schedule any of the other objects on the processor (including objects from another module), ensuring that the processor does not idle while there is still work to be done. When the requested remote data finally arrives (in the form of a message directed to a method), the runtime system can schedule the target method in the requesting object. Message-driven execution also allows a single server-like object to initiate several remote operations and process them in the order in which remote data becomes available, thus allowing the programmer to use careful non-determinism to improve efficiency. Message-driven execution thus has advantages over communication based on blocking receives and yields better performance by adaptively scheduling computations.

These issues, and empirical studies of the performance advantages of message-driven execution, are discussed in [15]. A comparison of message-driven execution with other concepts such as threads and active messages is also presented in [2].

2.2.2 Dynamic object creation: chares and messages

In order to support irregular computations in which the amount of work on a processor changes dynamically and unpredictably, Charm++ allows dynamic, asynchronous creation of parallel objects (chares), which can then be mapped to different processors to balance loads. Chare creation is a relatively low-cost operation: tens of thousands of chares can be created per second on a single current-generation processor, so chares may be thought of as medium-grained objects. A chare is identified by a *handle*, which is a global pointer; Charm++ therefore provides a shared object space for chares.

Chares communicate using messages. Sending a message to an object corresponds to an asynchronous method invocation. Message definitions have the form:

```
message class MessageType {
  // List of data and function members as in C++
};
```

Chare definitions have the form:

```
chare class ChareType {
  // Data and member functions as in C++.
  // One or more entry functions of the form:
```

```
 entry:
    void FunctionName(MessageType *MsgPointer)
    {
      ···C++ code block···
    }
};
```

In the above definitions, `message`, `chare`, and `entry` are keywords.

An entry function definition specifies code that is executed *without interruption* when a message is received and scheduled for processing. Only one message per chare is executed at a time. Thus chare objects define a boundary between sequential and parallel execution: actions within a chare are sequential, while actions across chares may happen in parallel.

Entry functions are defined exactly as sequential functions, except that they cannot return values and they must have exactly one parameter. This parameter must be a pointer to a message. The handle of a chare `ChareType` is of type `ChareType handle` (analogous to object pointers in C++, which have the type `ObjectType *`) and is unique across all processors. Just as C++ supports multiple inheritance, dynamic binding, and overloading for sequential objects, Charm++ supports these concepts for chares, thus permitting inheritance hierarchies of chare classes. Dynamic binding is also supported by allowing run-time determination of the type of a remote parallel object whose method is to be executed.

Every Charm++ program must have a chare type named `main`, which must have the function `main`. There can be only one instance of this chare type, which usually executes on processor 0. Execution of a Charm++ program begins with the system creating an instance of the `main` chare and invoking its `main` function. This function is typically used by programmers to create chares and branched chares (section 2.2.3) and to initialize shared objects.

Chares are created using the operator `newchare`, similar to C++'s `new` memory allocator. Its syntax is

```
newchare ChareType(MsgPointer)
```

where `ChareType` is the name of a chare class and `MsgPointer` is a message which can be accepted by the constructor entry-function of the chare class. The `newchare` operator deposits the *seed* for the new chare in a pool of seeds and returns immediately. The runtime system actually creates the chare at a later time on a processor selected by the dynamic load balancing strategy. When the chare is created, it is initialized by executing its constructor entry function with the message contained in the `MsgPointer` argument given to `newchare`. The user can also specify the processor on which to create the chare, through use of an optional argument, and thereby override the dynamic load balancing strategy.

`newchare` does not return any value. The user may, however, obtain a *virtual handle*—virtual because the chare has not yet been created. A chare can also obtain its own handle once it has been created. Chare handles may be passed to other objects in messages.

Messages are allocated using the C++ `new` operator. Messages are sent to chares using the notation

```
ChareHandle=>EF(MsgPointer)
```

which sends the message pointed to by `MsgPointer` to the entry function `EF` of the chare referenced by the handle `ChareHandle`. `EF` must be a valid entry function for that chare type. This method invocation is asynchronous: the caller continues with its computation while the callee may execute concurrently. Moreover, no value is returned from the method immediately. This syntax is intentionally different from the method invocation syntax for sequential objects in C++ so that programmers can clearly distinguish the semantics of parallel method invocation. The new token "`=>`" is used for sending messages instead of "`->`" to emphasize the difference between asynchronous message sending and synchronous sequential method invocation.

2.2.2.1 Supporting dynamic object creation and messaging

Dynamic load balancing: Dynamic object creation is supported by dynamic load balancing libraries, which help to map newly created chares to processors so that the work is balanced. Several load balancing strategies are provided [16], which may be selected at link time by the user depending on the demands of the application. In Chapter 4, we describe how the Paradise optimization tool automatically selects the load balancing strategy.

Prioritized Execution: Charm++ provides many user-selectable strategies for managing queues of messages waiting to be processed. Some of them, such as FIFO and LIFO, are based solely on the temporal order of arrival of messages. Priority based strategies allow the programmer to assign priorities to messages before they are sent. Charm++ supports integer priorities as well as bit-vector priorities that use a lexical comparison of bit-vectors to determine the order of processing. In Chapter 4, we describe how the Paradise optimization tool can automatically determine message priorities.

Conditional Message Packing Charm++ allows arbitrarily complex data structures in messages. On private memory systems, pointers are not valid across processors, hence it is necessary to pack pointer-linked structures into a contiguous block of memory before sending the message. However, packing is wasteful on shared-memory systems, or if the message is being sent to an object on the same processor as the sender. To allow optimal performance in these cases for messages involving pointers, the user is required to specify packing and unpacking methods called `pack` and `unpack`. These are called *by the system* just before sending and after receiving a message, respectively. Thus, only messages that are actually sent to other processors are packed.

2.2.2.2 Related work in concurrent objects

Dynamic object creation is provided in ICC++ [17], Mentat [18], μC++ [19], C++// [17], ABCL [20], and COOL [21], among others. Mentat supports coarse-grained objects, while

ICC++ and languages such as Cantor [22], HAL [23], and CA [24] support fine-grained objects. Compilers for fine-grained languages typically coalesce multiple objects and operations into coarser ones in order to achieve acceptable performance on available parallel machines.

While Charm++ does not allow parallelism within an object (i.e. all methods are atomic), CC++, ICC++ and others including actor-based languages allow parallelism within an object. These languages usually ensure consistency of shared variable access by supporting special program constructs such as locks, monitors, and single-assignment variables, or through data-dependence analysis in the compiler.

Support for mapping objects to processors is provided to some extent in Mentat, which provides random, round-robin and sender-initiated strategies, and also allows the programmer to specify object location hints. ICC++ supports randomized load balancing, and CC++, which uses a thread-based model along with logical processor objects for encapsulating address spaces, requires programmers to manage locality and load balance explicitly, and does not support dynamic mapping of objects to processors. Most other languages require programmers to specify object-to-processor mappings explicitly.

2.2.2.3 Example: a database query processing object

This section presents a simplified example of how message-driven execution and dynamic object creation can improve performance by doing work adaptively as remote data become available.

In parallel database applications, it is often beneficial to introduce intra-query concurrency to improve query response times. For example, consider an employee database having two relations: `Personal`, whose fields include `Name` and `Age`, and `Payroll`, whose fields include `Name` and `Salary`. Assume that the two relations are stored on separate disks, each served by an I/O processor, and that the query processor must request the two relations from the I/O processors before it can start processing them. The query processor is required to answer a

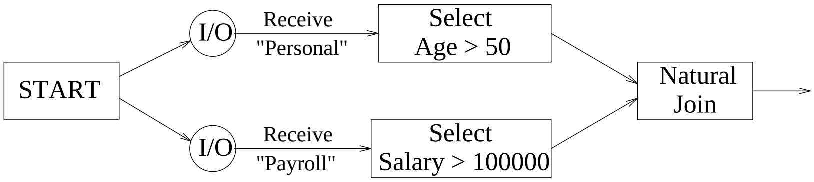


Figure 2.1: Macro-dataflow diagram for query object.

query of the form "Find all employees over the age of 50 who are earning more than $100,000." The macro-dataflow diagram for this query is given in Figure 2.1.

A program for this query using blocking receive calls implemented using standard message-passing libraries can be expressed as:

```
Query()
{
  RequestRelation(IOProcessor1, "Personal", ···);
  RequestRelation(IOProcessor2, "Payroll", ···);

  Relation *t1 = ReceiveData(IOProcessor1, ···);
  Relation *newt1 = t1->Select(AGE, GREATERTHAN, 50);

  Relation *t2 = ReceiveData(IOProcessor2, ···);
  Relation *newt2 = t2->Select(SALARY, GREATERTHAN, 100000);

  Relation *result = newt1->NaturalJoin(newt2);
  ··· return result relation to requester ···
}
```

The disadvantage of this program is that it will often have significant idle times. This is because the delays at the I/O processors are unpredictable, which means that the two relations may be received in any order. However, the code above enforces a fixed order of execution: if the

Payroll relation is received first, the processor will idle while it waits for the Personal relation, degrading the query response time. The following Charm++ code solves this problem:

```
chare class Query {
  int DonePersonal, DonePayroll;
  Relation *newt1, *newt2;
 entry:
  Query(QueryData *q)
  {
    IOChare1=>RequestRelation("Personal", thishandle,
                            &(Query::SelectPersonal));
    IOChare2=>RequestRelation("Payroll", thishandle,
                            &(Query::SelectPayroll));
    DonePersonal = DonePayroll = FALSE;
  }
  void SelectPersonal(Relation *t)
  {
    newt1 = t->Select(GREATERTHAN, AGE, 50);
    DonePersonal = TRUE;
    if (DonePayroll) JoinRelations();
  }
  void SelectPayroll(Relation *t)
  {
    newt2 = t->Select(GREATERTHAN, SALARY, 100000);
    DonePayroll = TRUE;
    if (DonePersonal) JoinRelations();
  }
 private:
  void JoinRelations()
  {
```

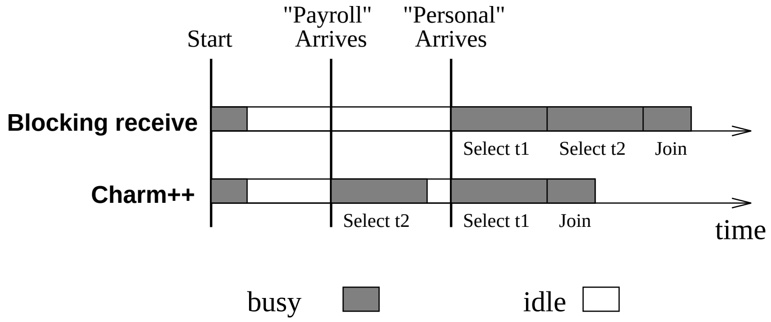


Figure 2.2: Performance of query object.

```
    Relation *result = newt1->NaturalJoin(newt2);
    ··· return result relation to requester ···
  }
};
```

In this Charm++ code, the `SelectPersonal` and `SelectPayroll` methods in the `Query` chare receive input from the I/O processors. These methods are invoked by the message-driven scheduler on the query processor, in the order the relations are received. If the `Payroll` relation arrives before the `Personal` relation, the blocking receive-based code will continue to wait for the `Personal` relation to arrive, while the message-driven code will execute the appropriate select operation automatically. This prevents the processor from idling while there is work to be done. Thus the Charm++ code above is likely to have a better query response time (Figure 2.2) if there is enough work in the select operations to amortize the overheads of message-driven execution. Note that `thishandle` specifies a chare's own handle and is similar to `this` in C++.

The advantages of message-driven execution are even more apparent when there are multiple objects active on a processor, e.g. when the processor is computing many complex queries simultaneously. Each query object is scheduled for execution as its data becomes available, which minimizes idle time and improves query response times.

The importance of other Charm++ features can also be seen in this example. Dynamic object creation allows independently-executing query objects to be created on demand as queries arrive in the system. Dynamic load balancing is necessary to ensure that new query objects are run on lightly-loaded processors, so that all processors have equal amounts of work. Message prioritization is also very useful when quick response times are needed for high priority queries, or to prevent large batch queries from delaying smaller interactive ones.

2.2.3 Parallel object arrays

A parallel object array is actually a group of chares with a single global name [25]. Parallel object arrays in Charm++ are a generalization of the branch-office chares of Charm, which allowed one group member per processor. A parallel object array is a multidimensional array of chares, with arbitrary mapping of array elements to processors (a default mapping is provided when mapping is not significant). An array element is identified by its coordinates. The array itself has a unique global group identifier.

2.2.3.1 Parallel array definition

A parallel array is defined as a normal parallel object (*chare*) class in Charm++, except that it must inherit from the system-defined base class *array*. This base class provides the following data fields:

- **thishandle** : this gives the unique handle (global pointer) of the array element.
- **thisgroup** : this gives the global id by which the whole array is known.
- **thisi, thisj, thisk** : these give the coordinates of the array element[1].

Messages that are sent between array elements must inherit from the system-defined message class *arraymsg*. The following code gives an example of an array definition.

[1] Currently, only 1, 2, or 3-dimensional arrays are supported, although this can be easily extended to higher dimensions. For brevity, all the examples in this section assume a 2-dimensional array.

```
message class MessageType : public arraymsg {
        // list of data fields to be sent
} ;

chare class MyArray : public array {
        // list of private and public data and function members
     entry:
        // list of "entry functions" where messages are received
        MyArray(MessageType *m) ;  // constructor
        void EntryFunction(MessageType *m) ;
} ;
```

2.2.3.2 Parallel array creation

A parallel array is created using the operator *newgroup*, which has the following syntax:

```
MapFunctionType mymapfn ;
MessageType *msgptr ;
MyArray  group arrayid1 =  newgroup MyArray[XSize][Ysize](msgptr) ;
MyArray  group arrayid2 =  newgroup (mymapfn) MyArray[XSize][Ysize](msgptr) ;
```

The code above creates two-dimensional parallel arrays. The **newgroup** operator causes all the array element objects to be created (and their constructors invoked) on their respective processors. The parameter **msgptr** is sent to all processors as the parameter to the constructor for each array element. The first array above uses the default mapping function. The second array has a user-specified mapping function **mymapfn**, which takes the coordinates of an element as input and returns the processor where the element is located. E.g. a mapping function which implements a cyclic mapping for a 1-dimensional array is defined as:

```
int CyclicMapping(int arrayid, int i)
{
```

```
        return i%CNumPes() ;
}
```

newgroup is a non-blocking operator that immediately returns the id of the newly created array, which has the type MyArray group, and is analogous to a global pointer to an array. Because of its non-blocking nature, the elements of an array might not have been created when *newgroup* returns the array id. If necessary, the programmer may explicitly synchronize after initialization of all array elements on all processors by using a suitable reduction or synchronization operation. Currently, parallel arrays may be created only from processor 0.

2.2.3.3 Asynchronous messaging: remote method invocation

The parallel array construct provides both point-to-point as well as multicast messaging. All messaging is asynchronous (no reply value is allowed), in keeping with the non-blocking communication paradigm of Charm++. If a reply is desired, the receiving object must send a reply message back to the sender object.

The syntax for point-to-point asynchronous messaging is:

```
arrayid[i][j]=>EntryFunction(msgptr) ;
```

where arrayid is the "global pointer" to the parallel array, i, j are the coordinates of the recipient array element, EntryFunction is the function to be invoked in the receiving object, and msgptr is the message to be sent across, which is passed as the sole parameter to the function.

The syntax for multicast asynchronous messaging is:

```
arrayid[i1..i2][j1..j2]=>EntryFunction(msgptr) ; // multicast to sub-array
arrayid[ALL][j]=>EntryFunction(msgptr) ;         // multicast to column
arrayid[i][ALL]=>EntryFunction(msgptr) ;         // multicast to row
arrayid[ALL][ALL]=>EntryFunction(msgptr) ;       // multicast to whole array
```

If an array element is known to be on the local processor, its data and function members may be accessed as usual (this feature is borrowed from the branch-office chares of Charm, and is very useful for inter-module interfaces):

```
arrayid[i][j]->datamember ;

arrayid[i][j]->function(....) ;
```

2.2.3.4 Array remapping

The parallel array construct supports both synchronous remapping and asynchronous object migration. Synchronous remapping must be initiated from processor 0 as follows:

```
arrayid->remap((MapFunctionType)newmapfn, return_chare_handle,
                                          &(ReturnChareType::ReturnFunction));
```

`newmapfn` is the new mapping function. All array elements will be moved from their original locations to their new locations as specified by the new mapping function. After all elements have been installed on their new locations, a message is sent to the function `ReturnFunction` in the chare object specified by `return_chare_handle`. This provides a synchronization point after remapping. The user program must ensure that during remapping no messages are sent to any elements of the array being re-mapped.

2.2.3.5 Asynchronous migration for parallel arrays

Sometimes it is not feasible to synchronize the whole computation before performing a remap. Moreover, the overhead of a synchronous remap is not overlapped with any computation, which may lead to large idle times for processors. Asynchronous remapping or "migration" solves these problems. It is activated by each array element independently. Each element to be moved calls the function `array::migrate((MapFunctionType)newmapfn)`. The `newmapfn` parameter specifies the new mapping function, which tells the run-time library the destination processor for the array element. The call results in only the specified object being moved to its destination processor. The actual steps performed by the runtime system while migrating an object are:

1. Before migrating an object, the runtime library calls a user-provided *pack* function on the object, which packs the object's data area into a contiguous message buffer. The programmer must provide a pack function for every object type that needs migration[2].

2. Send the message to the object's destination processor

3. Create the new object on the destination processor

4. Initialize the new object's data area using the message buffer. This is done by another user-provided *unpack* function. Note: the pack and unpack functions are virtual functions defined in the base class `array`.

5. Forward messages directed to the object from the old processor to the new processor.

Message forwarding for asynchronous migration:

Message forwarding is one of the critical issues in object migration, and has been extensively researched. The most common approach is for the migrating object to leave a forwarding address on its original processor. However, this may cause messages to incur significant forwarding overhead. In parallel computations where migrations occur frequently (e.g. in iterative computations having one ore more remapping operations every iteration), a message may have to travel many hops in order to find its object. To reduce overhead, some schemes have the object broadcast its new location to all processors so that messages can be sent directly to its new location. However, when there are many objects with frequent migrations, broadcasting is not practical.

[2]For simple object types it is possible for a compiler to automatically generate pack functions. However, for objects containing pointers this is more difficult.

We have designed an optimized protocol for message forwarding for parallel object arrays. We make use of the fact that an array has a mapping function associated with it, which specifies the location of all objects. Moreover, the normal behavior is for all array elements to migrate (i.e. the whole array gets remapped) at the end of a phase of computation. We store a table of mapping functions on each processor, which gives the mapping function for each phase of the computation (it is assumed that all objects have the same mapping function for a given phase). Because of the asynchronous nature of the migration, it is possible for objects from different phases to be simultaneously active on a processor. Hence we associate a phase number with every object, indicating the phase it is in at a given time. The phase number is incremented every time the object migrates. Also, every message is annotated with the phase number of the sending object. When an object sends a message, the mapping function corresponding to its phase is used to find the location of the destination object. When a processor receives a message, it checks to see if the destination object resides on its processor, and if so, delivers the message. If the destination object does not exist, the processor finds the last phase when the destination object was on the processor (thus taking into account cyclical migration). If the message's phase number is less than or equal to the last phase, it means the object was on the processor and has migrated away, so the message's phase number is incremented, and it is forwarded using the corresponding mapping function. If the message's phase number is greater than the last phase when the object was on the processor, it means the object will migrate to the processor in the future. So the message is temporarily buffered, and is delivered to the object when it arrives. Thus we have implemented message forwarding for parallel object arrays with low overhead.

2.2.3.6 Using parallel object arrays

Parallel object arrays are a versatile construct, with several modes of usage. Many of the advantages of parallel object arrays are derived because they provide a sequential local function call (like the branch-office chares of Charm):

- They can be used to implement distributed services, such as distributed data structures, global operations, and high-level information-sharing abstractions. Objects that interact with a parallel array need to communicate only with the local representative of the array, and need not be concerned with how the array elements collaborate among themselves to carry out the requested operation. Thus parallel object arrays provide a powerful mechanism for encapsulating complex concurrent operations.

- Multidimensional parallel object arrays can be used to represent a computational space in scientific applications. Each array element owns a subregion of the space (e.g. a set of grid points in a computational mesh), performs computations in that subregion, and communicates with other elements of the array (e.g. to transfer boundary grid points). The mapping of array elements to processors specifies the decomposition of the computational space. Section 5.2 describes how parallel object arrays were used to experiment with different mapping schemes for the NAS Scalar Pentadiagonal program, with almost no code changes. Asynchronous object migration overlaps computation with communication, hence can achieve much better performance than synchronous array transpose; Section 5.2 presents results from the NAS Scalar Pentadiagonal program to demonstrate this.

- They also provide a convenient mechanism for distributed data exchange between modules when there is exactly one array element per processor. The group identifier allows each element of a parallel array in one module to hand data over to the element in the other module on its own processor. For example, in a molecular dynamics application, elements of the main module might hand over the coordinates and velocities of all atoms to the corresponding elements of an "kinetic energy calculator" parallel array. In a pure actor model, finding the corresponding elements on each processor would be somewhat clumsy.

- Finally, parallel arrays having one array element per processor can also be used to encapsulate variables whose value is specific to every processor (e.g. a list of neighbors in a grid computation, or statistics of program execution for a performance measurement module).

2.2.3.7 Example: global sum using parallel object arrays

The following parallel array object forms the global sum of values resident in each processor, without creating a barrier.

```
message class IntegerMessage {
  int value;
};


chare class Sum : public array {
  int total, numResponses;
  ClientFnType clientfunction;
  Client group client;  // Client is the superclass for all clients
 entry:
  Sum(EmptyMessage *);  // Initialize local variables
  void FromChild(IntegerMessage *m)
  {
    total += m->value;
    if (++numResponses == CNumPes()) {
      // The library function CNumPes() returns the number of processors
      m->value = total;
      client[ALL]=>clientfunction(m);
    } else {
```

```
      delete m;
    }
  }
 public:
  void DepositNumber(int n, Client group c, ClientFnType f)
  {
    clientfunction = f;
    client = c;
    IntegerMessage *m = new IntegerMessage;
    m->value = n;
    thisgroup[0]=>FromChild(m);
  }
};
```

The client on each processor gives its value to the local representative of `Sum`, which forwards it to the array element on processor 0. When the number of elements that have sent their values equals the number of processors, the array element on processor 0 broadcasts the global sum result to all objects in the client. A scalable version of `Sum` avoids the bottleneck at processor 0 by using a spanning tree.

```
chare class AnyClient : public Client {
  Sum group sum;
  AnyFunction()
  {
    int number ;
    ···compute this processor's number···
    sum[CMyPe()]->DepositNumber(number, thisgroup, &(AnyClient::ReceiveSum));
    ···do other work···
  }
 entry:
  void ReceiveSum(IntegerMessage *m)
```

```
  {
    // Sum is in m->value
  }
};
```

Note the use of the group identifier and entry function pair in the parameters to `DepositNumber()`: this provides a "return address" which is used by the `FromChild` method in the `Sum` object to broadcast results back to the client, or possibly delegate another object to do the work and return results directly to the client.

If multiple concurrent client objects (i.e. with overlapping executions) need to compute global sums, they can do so by creating and using multiple instances of the `Sum` parallel array. A simple function call interface (via the `DepositNumber` function) to a global sum module without using parallel arrays does not provide the same flexibility. The ability of parallel arrays to link together a group of objects created via a single creation call is crucial for such applications involving multiple clients.

2.2.3.8 Related concepts

Parallel arrays have been used in various forms in several parallel programming systems. In data-parallel languages such as HPF [26] programmers have a shared-memory model in which arrays in the program are distributed over processors using compiler directives, and computations are assigned to processors using the "owner-computes" rule. Some parallel C++ systems such as pC++, LPARX, Amelia, CHAOS++ and POOMA [27, 17] support parallel arrays or collections of objects (LPARX and CHAOS++ support irregular arrays too) for scientific applications, also in the context of a loosely-synchronous data-parallel model. The advantages of our parallel object array abstraction are:

- in contrast to the lock-step operations on array elements in HPF, our array element objects may proceed independently of each other executing different methods as necessitated by the application.
- the ability to increase performance through asynchronous operations (e.g. method invocation and object migration) which overlap communication and computation.
- more flexibility in specifying arbitrary mappings of parallel object arrays (e.g. the multi-partition mapping scheme we used in the NAS SP benchmark cannot be specified using HPF's compiler directives).

Parallel object arrays in Charm++ are closest in spirit to the collections construct in ICC++ [17], which was developed around the same time. ICC++ does not support multicasts and remapping, though. Parallel object arrays are based on the notion of branch-office chares in Charm, which provided a group of message-driven objects with one element per processor, with asynchronous as well as local synchronous method invocation.

2.3 Implementation

Charm++ has been implemented using a translator and a runtime system. The translator converts Charm++ constructs into C++ constructs and calls to the runtime system, and also generates stubs for messaging. The runtime system consists of a language-independent portable layer called Converse [13], on top of which is the Chare Kernel layer [8, 11]. Figure 2.3 shows the structure of the runtime system.

2.3.1 Converse: portability and interoperability

The Converse layer provides a portable machine interface which supports essential parallel operations on MIMD machines. These include synchronous and asynchronous sends and receives, global operations such as broadcast, atomic terminal I/O, and other advanced features.

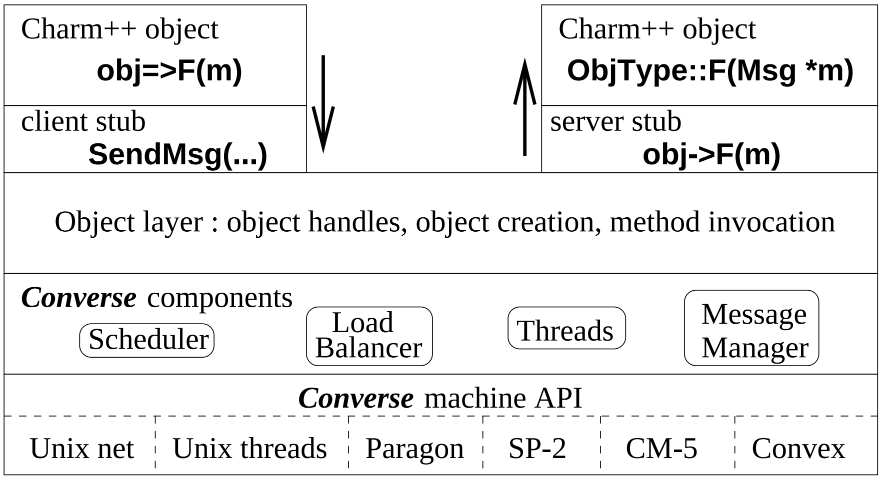


Figure 2.3: Structure of the runtime system for Charm++.

Converse is designed to help modules from different parallel programming paradigms to interoperate in a single application. In addition to the portable machine interface, it provides common paradigm-specific components, such as a scheduler, load balancer, message manager and user-level threads. These components can be customized and used to implement individual language runtime layers.

Some important principles that guided the development of Converse include:

need-based cost: Since Converse is intended to be an underlying interoperable layer, it should not require upper layers of software to incur a cost for functionality they do not need. E.g. Converse should not require all messages to go through a tag-based receive mechanism and its associated overhead (which exists in most message-passing libraries supported by vendors), since a message-driven system such as Charm++ does not need it. similarly, Converse should not require all messages to go through a scheduler with its corresponding overhead, since a simple message-passing library does not need it.

efficiency: the performance of programs developed on top of Converse should be comparable to native implementations; and

component-based design: the Converse layer is divided into components with well-defined interfaces, and possibly multiple implementations, which can be plugged in as required by higher layers.

Converse supports both SPMD style programs (which have explicit control flow as specified by the programmer and no concurrency within a processor) and message-driven objects and threads (which have concurrency within a processor and implicit, adaptive scheduling).

2.3.2 Chare kernel

The Chare Kernel layer was originally developed to support Charm, but was then modified to support the C++ interfaces required by Charm++. It implements functions such as system initialization, chare creation, message processing (including identification of target objects and message delivery) performance measurements, quiescence detection, and so on.

One important function of the Chare Kernel is to map parallel class and function names into consistent integer IDs, which can be passed to other processors. This is required because function and method pointers may not be identical across processors, especially in a heterogenous execution environment. The Charm++ translator cannot assign unique IDs to classes and methods during compilation because Charm++ supports separate compilation, and the translator cannot know about the existence of other modules. Also, this mapping must be implemented so as to support inheritance and dynamic binding while passing IDs for methods across processors. When a sender sends a message to a chare C at an entry function E defined in C's base class, C must call its own definition of E if it has been redefined, but its base class's definition of E otherwise.

To meet these requirements the Chare Kernel provides a function registration facility, which maintains the mapping from IDs to pointers. The translator-generated code uses this registration facility during run-time initialization to assign globally unique indices to chare and entry function names. These unique IDs can be passed in messages between modules. The translator also generates a stub function for every entry function in every chare class. When a message is received and scheduled for processing, the Chare Kernel uses this stub function to invoke the correct method in the correct chare object. To make dynamic binding work, the stub function invoked is the one corresponding to the static type of the chare handle at the call site; the C++ virtual function mechanism then invokes the correct method depending on the actual type of the chare object.

The Chare Kernel uses a scheduler (defined as a component of Converse) which picks up incoming messages from the Converse message buffer, enqueues them by priority according to a user-selected queueing strategy, and then chooses the highest-priority message from the queue for processing. The Chare Kernel also manages chare handles (which are essentially global pointers) and handles the mapping between local object pointers and chare handles.

Parallel object arrays are implemented using a hash table to map an array element's coordinates to its actual object pointer. For sending method invocations to an array element, the run-time first finds the processor on which the element resides using the mapping function, and then performs the usual send operation. Multicasts are implemented by sending one message to each processor; if there is more than one array element on a processor, the message is replicated after reception.

2.4 Summary

Charm++ provides a rich set of facilities that make it suitable for a broad range of applications. Some of the features that distinguish Charm++ from other approaches to parallelizing C++ include:

- its support for irregular (non data-parallel) computations through dynamically creatable parallel objects (chares), dynamic load balancing and message prioritization;

- its parallel object arrays which encapsulate complex parallel tasks as well as support multidimensional data-parallel computations, including multicasts and asynchronous remapping.
- its support for modularity without loss of efficiency through the message-driven execution model;
- its ability to coexist with other language modules in a single application due to its implementation on the Converse runtime system.

Chapter 3

A framework for automating runtime optimizations

This chapter describes the basic concepts in the framework for automating runtime optimizations, which has been embodied in the Paradise post-mortem analysis tool [1]. We first motivate runtime optimizations and the use of post-mortem analysis to automate them. We describe the framework for performance optimization, discuss issues in the program representation and present strategies for inferring optimizations and generating concise hints. Finally we present a detailed comparison with related work. The next chapter explores each optimization in detail.

3.1 Why run-time optimization

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 unpredictable factors.

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

- For parallel object-oriented programs based on C++, it is difficult to produce good compilers which infer program characteristics because of the difficulties of precise dependence and type analysis in the presence of pointers.

- 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.

- Many parallel programming environments are collections of libraries which implement a set of commonly needed functions (e.g. PVM, MPI). For such programs, run-time optimization is the only way to get better performance.

In short, 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.

Run-time optimizations are those for which mechanisms or policies need to be carried out at run-time, hence necessitating support from run-time libraries in order to perform the optimization. Optimizations performed at run-time can take care of many cases where compiler transformations are inadequate.

3.2 Implications of object-orientation for run-time optimization

As discussed in Section 2.1.1, Charm++ provides significantly greater challenges and opportunities for automated runtime optimization because it has been designed to automate object placement and scheduling. Further, object-orientation helps to support run-time optimizations by making it easier to specify information about the application which is necessary for optimization.

In most current systems, run-time libraries which need information from the application are restricted to *observing* application objects, and can collect only externally visible data such as the number of objects on a processor, which objects they sent messages to, etc. Thus application objects are a "black-box" as far as the run-time is concerned. In order to make application information available to the run-time in a truly flexible and general manner, the run-time should be able to "look inside" application objects in order to determine their properties. E.g. Instead of measuring processor load just by the number of active objects or messages in the scheduler queue, each application object can provide its own load as a internal (local) variable; the load manager can then sum this variable for all objects to get an accurate estimate of processor load. This can be done by defining an abstract class containing the load variable which is known to the load balancer, and deriving all objects requiring load balancing from this abstract class.

Also, whenever information from the application program has to be transferred to an optimization library, the information has to be rearranged into a data-structure as required by the library. This makes the task of integrating an optimization library into an application much more difficult. On the other hand, if we define an interface class which is known to the library module, then the application program can inherit from the interface class and use the functionality provided by it with little code modification. E.g. static load placement libraries usually partition an object interaction graph in order to achieve locality and load balance. Instead of explicitly creating such a graph and passing it to the library, the graph can be defined with abstract classes for vertex and edge, and application objects can inherit from these classes, so that the graph already exists in the application. Thus inheritance is a very useful mechanism for making information available to optimization libraries at run-time.

3.3 Why post-mortem analysis

In order to optimize a parallel program, we need information about the characteristics of the program to parameterize the optimization mechanisms and guide strategies for selecting them. As discussed in the previous section, compilers cannot provide the required information in many cases.

The next stage where information may be obtained is at run-time. Optimization libraries may dynamically collect information while the program is executing, and base optimization decisions on this information. This method has been used in several of the Charm++ runtime libraries: e.g. for load balancing. The advantage of this method is that input-dependent information is available at runtime. However, the major drawbacks of collecting information only at runtime are:

- Collecting information and executing optimization-selection strategies adds overhead which directly affects the program's performance.
- It is difficult to detect the overall problem structure such as communication patterns, phase structure, grainsize distribution, etc. Such global properties require extensive coordination and integration of runtime data from all processors, which can be very expensive.
- It is very difficult to make predictive decisions; most optimizations will be reactive in that they take action *after* a problem is detected. Further, the timeliness of optimizations may be affected, decreasing their potential for improving performance.

Hence we turn to post-mortem analysis to supply information for optimization that is not available from the compiler or runtime system.

It is commonly experienced that different executions of a parallel program are often similar, if not identical, in their structure and 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 hints for 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 heuristics that are known to expert parallel programmers; 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 one often encounters performance problems of the first category. It is these problems that we focus on.

3.4 Framework for automating run-time optimizations

In this thesis we develop a comprehensive framework for automating run-time optimizations. The framework involves an intelligent post-mortem analysis tool (Paradise) — which analyzes traces of program execution and finds program characteristics to suggest optimizations — working in close cooperation with a run-time system which carries out optimizations during program execution (Figure 3.1). The run-time libraries use hints from Paradise as well as information collected at runtime to parameterize optimizations and select between alternate optimization strategies.

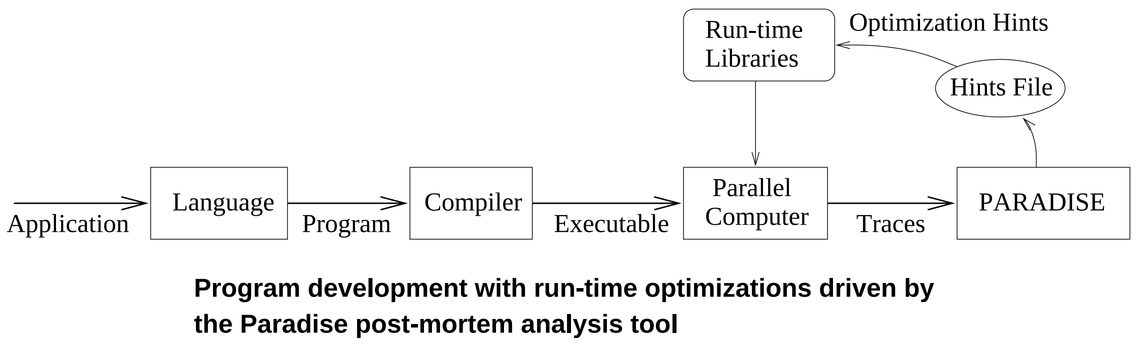


Figure 3.1: Framework for automating runtime optimizations.

Paradise builds a representation of the program's execution from traces, determines characteristics of the program, then uses several specific heuristics to find optimizations that would solve the performance problems, and generates concise hints which are communicated to the runtime libraries via a "hints file". The sophistication of optimizations frequently depends on the accuracy of the information that can be deduced by post-mortem analysis: thus the optimizations may have alternative strategies, one of which is chosen by Paradise depending on the accuracy of information available.

The optimizations in the framework are aimed at improving load balance and locality by better mapping of objects to processors, scheduling computations to optimize critical paths, reducing communication volume, and optimizing object granularity to reduce overheads. In most cases we try to introduce these optimizations without intervention from the compiler or programmer. Chapter 4 describes each optimization in detail. We first discuss some basic issues in the design of the framework for automating runtime optimizations.

3.5 Parallel program representation

The execution of a Charm++ program is represented as an *event graph*, which is constructed using traces collected at run-time. The basic version of the event graph was developed for the *Projections* [28] performance visualization and analysis tool. Issues in collecting trace data, reducing perturbation, and constructing the basic event graph are discussed in [29] and are beyond the scope of this thesis.

The basic event graph consists of vertices representing method executions within chare objects, and edges representing messages sent by a method and causing the execution of another method. Thus the basic event graph corresponds to a dynamic task graph of the program: the vertices specify computations and the edges specify communication between the computations. Vertices are labeled with the name of the method they represent, and the id of the object instance whose method was executed (for array element objects, the id includes the global id of the parallel object array and the coordinates of the element object). Since in Charm++ a message causes the invocation of a method, each method has exactly one predecessor, i.e. each method execution has exactly one *creation message*. Thus the basic event graph is actually a tree. There are three events corresponding to a vertex (each event is uniquely identified by an event number and a processor) : its *creation* event (which gives the time and processor from which the creation message was sent), its *start* event (which gives the time and processor when the method began executing) and the *finish* event (which gives the the time and processor when the method completed execution). A vertex also has a list of outgoing edges corresponding to the messages created by it. Finally, each processor also has a list of events that occurred on it, allowing idle times and other processor-specific metrics to be measured.

In the basic event graph all vertices had exactly one predecessor since a method is triggered by the arrival of exactly one message. Moreover, causal relations (intra-object dependences) and synchronization events were not modeled. Hence this simple event graph cannot model synchronizations such as in "join" operations: i.e. where a method executes only if several preceding methods have executed (Figure 3.2a). We now consider enhancements to the event graph to overcome these problems as well as the problems due to program unpredictability.

3.6 Problems due to unpredictability

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 predictable behavior (such as some SPMD or loop-based 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 unpredictable programs and for variations in the input parameters, the program execution may be different, so that fewer inferences may 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), or find higher-level input-independent patterns in the properties, because we cannot make suggestions at the level of individual instances.

The characteristics of a parallel program may be execution specific (they change from run to run even for the same input set), input specific (they change depending on the inputs, but remain consistent across runs), application specific (they are a property of the application and remain consistent across different input sets), or machine specific (they are a property of the machine the program was run on).

We now examine the causes for executions of the same program to differ. Whereas completely predictable programs always produce the same set of tasks and messages, most parallel programs contain unpredictability 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), or the number of processors. In some programs even the kind of computations performed may change depending on input.

- Scheduling: Some programming models (such as Charm++) adaptively schedule computations to overlap them with communication and also to adjust for varying run-time conditions. Thus even for executions with same inputs, unpredictability can arise due to different orderings of computations.

- 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 parallel program uses 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.

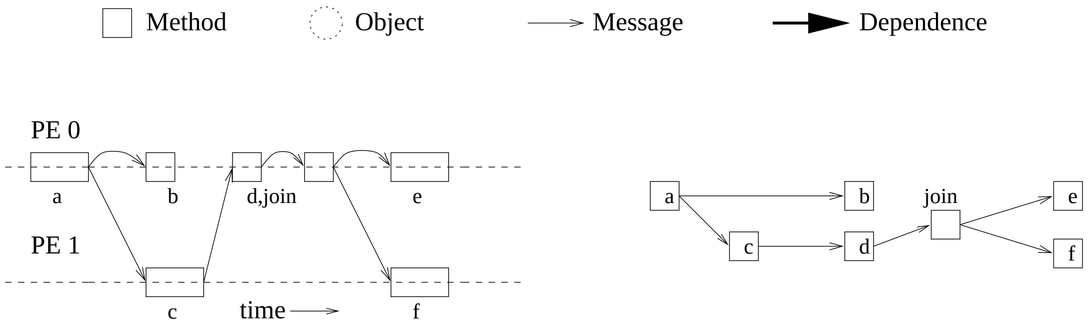
In this work we do not tackle the last two problems (unpredictability due to dynamic granularity control and speculative work). The next section describes techniques to handle the other types of unpredictability.

3.7 Handling unpredictability

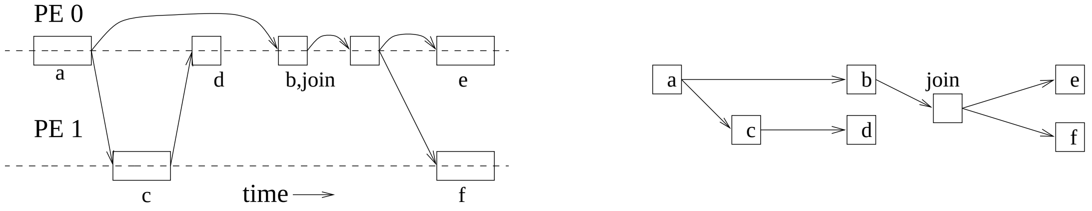
We target our techniques to handling different input sets as long as the basic computation steps do not change: i.e. only the numbers of objects and messages may change, not the structure of the parallel program or the kinds of computations performed. Paradise does not normally generate input-specific optimization hints which depend on individual message or object instances. Instead, as far as possible, we generate application-specific hints which do not depend on the particular input set: most of 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"), and application level properties and patterns (e.g. the program has tree-structured communication). However, for a few optimizations it is not possible to generate input-independent hints (e.g. message pipelining where the optimal degree of pipelining depends on the sizes of messages and computations). In such cases we assume that the input sets fed to the program cause it to have substantially similar characteristics as the run from which traces were collected, so that the optimization hints generated by Paradise will still be valid. Moreover, several of the runtime libraries (e.g. load balancing) collect input-dependent information at runtime, to complement the input-independent information given by Paradise.

To deal with the unpredictability caused by adaptive scheduling, we first need to develop a representation of parallel program execution which will remain the same for different schedules. The basic event graph can change in structure due to execution order reversal caused by adaptive, dynamic scheduling. E.g. consider a join operation (Figures 3.2a and 3.2b) where methods b and d must execute before the $join$ method. The basic event graph would only have one edge from the last method to the join, hence the two possible event graphs have different structures. To handle this, we need to make modifications so that *the event graph should remain isomorphic* even with changes in the temporal order of execution of computations.

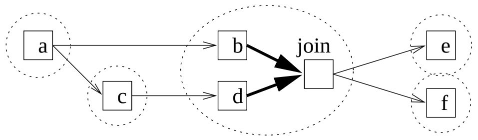
We have enhanced the basic event graph with precise information about intra-object synchronizations (internal dependences between methods), such as specified in the Dagger notation [30]. Dagger allows programmers to specify synchronizations between methods and messages: e.g. "method X can execute only if method Y has executed and message Z has arrived". We incorporate this information into the basic event graph. The *enhanced event graph* consists of vertices for method executions, edges for messages between methods and edges for internal dependences between method executions of the same object instance. Vertices in the event graph which were seemingly unrelated before will now be connected by dependences, yielding a



(a) Timeline and basic event graph when b executes before d



(b) Timeline and basic event graph when d executes before b



(c) Enhanced event graph with dependences and object grouping

Figure 3.2: A parallel operation such as "join" can result in different event graphs due to execution order reversal, which is prevented by adding dependences between method executions.

isomorphic representation even when there is adaptive scheduling. E.g. in Figure 3.2c, methods b and d are connected by dependences to the $join$ method, resulting in a graph which is isomorphic even when b and d execute in different orders. Thus methods can now have more than one predecessor (one message edge and one or more dependence edges), hence join operations can be modeled.

Finally, we handle unpredictability due to dynamic object placement. Paradise groups all event-graph vertices corresponding to the same object instance rather than grouping vertices by processors: this allows us to analyze the interactions between object instances (Figure 3.2c), instead of restricting ourselves to a less precise processor-level analysis. This is very useful when we try to analyze patterns of communication between objects. We now have an object-interaction graph which is homomorphic with respect to the enhanced event 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. This gives an execution representation that does not change even when objects are mapped to different processors.

3.8 Methodology

This section discusses several general strategies for inferring optimizations and generating concise hints, which played a key role in the design of Paradise.

3.8.1 Inferring optimizations

The process of discovering performance problems and selecting optimizations involves searching through a large space of possible alternatives. Paradise uses a combination of two approaches for this, which are well known in the context of expert systems:

- Forward reasoning: Determine the properties and characteristics of the program from the event graph. Use heuristic rules to match the characteristics to optimizations. This process is similar to the *heuristic classification* scheme used in Poirot [31], and the search-tree based analysis in Projections [29].

- Backward reasoning: For each type of optimization, determine if the program would benefit by its application, by extracting the relevant characteristics from the event graph. If so, determine the necessary detailed information to parameterize the optimization.

3.8.2 Generating concise hints using pattern matching and types

Another key problem that arises while automating optimizations is the necessity of generating a concise hint to the runtime libraries. The event graph contains a huge mass of information about each individual object and method invocation at run-time. As discussed in Section 3.7, the requirement of input-independence prevents us from generating hints in terms of individual instances of objects and messages. Also, it is infeasible to generate a large list of hints for each individual object or method: there may be hundreds or thousands of such hints, and reading, storing or searching them may cause substantial overhead at run-time.

We solve the problem of generating a concise hint using *pattern matching*. Given a list of entities (e.g. objects, methods, phases), each of which has a numerical property (e.g. the processor an object is mapped to, or the priority of an object, etc.), we want to be able to induce a pattern or function which maps the entity to the property. This function then provides a concise representation (closed form) of the information contained in the mapping, and can be easily specified in the hints file and used by the runtime libraries. Paradise constructs a list of entities, each identified by a unique number or set of coordinates. Each entity in the list has its associated value The mapping patterns used depend on the type of entity and value, and are given in more detail in the optimization sections. The expressions include linear, modulo, block-cyclic and other forms.

For each optimization, Paradise heuristically decides which of these expressions to evaluate. Then for a given expression, it tries to determine the constants in the expression that would best match the entity-value pair for all such pairs: the quality of the match is determined by the fraction of pairs that satisfy the pattern. The constants to evaluate are determined algebraically in some cases (e.g. by solving a set of linear equations) and otherwise by exploration of a space of possible constants[1].

When the properties of an entity are not quantitative or when individual entities cannot be identified uniquely, we use type information to generate concise qualitative hints. Essentially we would like to specify information for optimization in terms of sets of message instances which can be identified at the program level. We have defined a *lattice of message sets* ranging from most general (all message instances) to most specific. The kinds of sets in the lattice include: (a) all message instances (b) messages of a particular type (c) messages to/from an object type (d) messages to/from a particular method type (e) messages to/from a processor (f) messages in a particular communication pattern (g) messages in a particular phase of the program. Combinations of these sets give rise to additional sets in the lattice. Figure 3.3 shows an example of the kinds of sets in the lattice. The most general set is at the top, while the sets lower in the hierarchy are more specialized. E.g. The set "messages to a method" is a subset of "messages to an object type". Depending on the particular optimization, Paradise tries to specify properties for the most specific set of entities possible.

3.9 Related work

Our framework for automating runtime optimizations is different from but related to previous research in the areas described below.

3.9.1 Performance analysis tools

Performance analysis research has concentrated on visually displaying performance data and relating performance data to high level language constructs [32, 33, 28, 34, 35, 36, 37, 38, 39,

[1] In future, it may be useful to employ least-squares curve fitting or more sophisticated regression analysis.

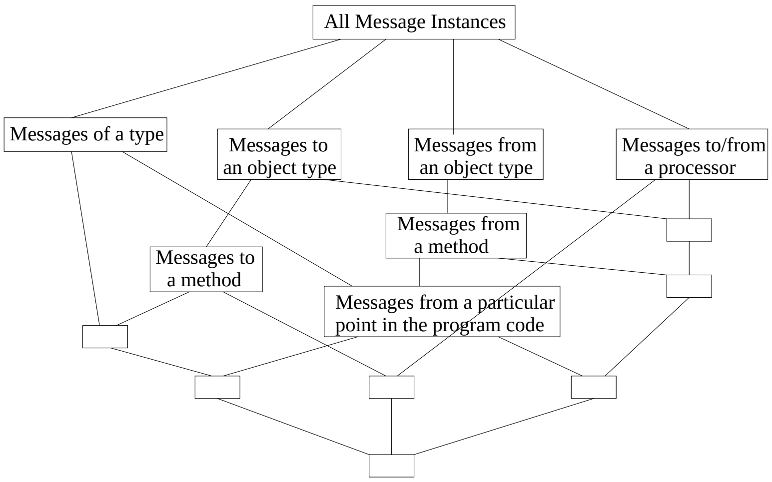


Figure 3.3: Example schema for lattice of message sets.

40, 41, 42, 31, 29, 43]. The features supported by some or all of these tools include graphical display of performance metrics such as utilizations and numbers of messages using histograms, animations of program execution, time-lines of activity on processors, flexible trace formats, dynamic instrumentation and perturbation reduction, and high-level language support by relating performance data to sections of source code.

A few of the above performance tools also go further than merely reporting performance data. These include MPP-Apprentice [43] and P^3T [39] for data-parallel programming models; Poirot [31] and Paradyn for message-passing / data-parallel programs [34]; and Projections for object-based message-driven programs [29]. These tools analyze the performance data to give the user insights into performance problems and diagnose their causes. Some of the common performance problems identified include under-utilized processors, load imbalance, excess communication overhead, long critical paths, system library overheads, synchronization bottlenecks, scalability problems, too small grainsize, poor cache usage, and others. Some of the tools also give a quantitative estimate of performance improvements when each performance problem is corrected, and prioritize the problems depending on their severity. In particular, the Projections performance analysis tool [29] developed earlier in the author's research group has an expert analysis component that incorporates several of the above features. However, all the above tools leave the task of interpreting the performance problem and choosing and incorporating optimizations to the programmer.

Our framework aims to go a step further in the direction of automation: the techniques embodied in Paradise not only identify performance problems, but also provide solutions in terms of optimizations for the problem areas, and in co-operation with the run-time libraries, incorporate the optimizations in the program without programmer intervention. Paradise can be profitably used in conjunction with a performance visualization tool, and complements the functionalities provided in the tools listed above.

3.9.2 Compiler and runtime optimizations

Automatic compiler optimizations have achieved success for automatic data partitioning and communication schedule generation in array-based Fortran programs [44, 45]. The use of profile information for compiler optimizations is well-known. Several sequential compilers use profile information to predict branch probabilities (e.g. [46]). Some compilers for data-parallel languages such as HPF [39] use profile information to accurately find the cost of various computation and communication operations. Parallelizing compilers for loop-based programs have explored the use of runtime dependence analysis and parallelism detection (e.g. using inspector and executor code) and speculative execution [47, 48, 49, 50], especially for irregular programs with input-dependent data access patterns. However, such work has been restricted to loop-based scientific programs, usually running on shared memory multiprocessors. To the best of our knowledge, our framework is one of the first efforts in *parallel object-oriented systems* towards using *post-mortem analysis* for automating *run-time* optimizations. 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.

Runtime systems for some parallel object-oriented languages incorporate several runtime optimizations. The Charm runtime system itself [51] provides several dynamic load balancing strategies for optimizing object placement, prioritized message-queueing strategies for optimizing scheduling, and optimization of remote object creation using "virtual object handles". CHAOS++ [17] optimizes irregular communication between distributed arrays of objects with pointer-based referencing. It also provides user-selectable partitioners for distributing irregular arrays across processors. CHAOS++ depends on a loosely-synchronous (separate phases of computation and communication) model, allowing communication to be performed by inspector and executor phases. COOL [52] is a shared-memory, thread-based parallel language which uses hints from the programmer to optimize scheduling and data locality. The programmer associates tasks with objects, objects with other objects, and objects with processors by specifying *affinity hints*: the runtime then tries to place tasks and objects on processors to satisfy the hints, as well as schedule tasks of an object back to back to preserve cache locality. ICC++ and the Concert system [17, 53] support grainsize optimizations through compiler analysis: the compiler increases the grainsize of objects by inlining method invocations statically, or by cloning methods and generating hints to the runtime via annotations. The actor-based language HAL [54, 55] optimizes request-reply communication by transforming them into asynchronous sends and extracting continuations, converts local messaging into function calls, and optimizes remote actor creation using alias addresses for actors. Mentat [17] supports dynamic placement through round-robin, random and sender-initiated strategies. However, the selection of strategy must be done by the user. Thus most of the above runtime systems merely provide optimization mechanisms and require the programmer to manually specify the optimization strategies to be used in a program, or assume a more restricted programming model, or only optimize low-level mechanisms.

In contrast, to the best of our knowledge, our framework is the first to automatically extract application-specific characteristics using post-mortem analysis, and use these characteristics to guide high-level optimization strategies and parameterize optimization mechanisms. Further, the optimizations are incorporated automatically into the runtime system by building runtime libraries which read in optimization hints from Paradise.

3.9.3 Other programming models

The unique aspect of our work is that it is in the context of a parallel object-oriented model which allows the run-time system the flexibility to choose strategies for placement (mapping) and scheduling of computations and managing communication between objects. Paradise automates optimizations for dynamic and static object placement, scheduling, grainsize control and communication reduction. Thus there are significantly greater opportunities and challenges for automatic optimizations in our framework. 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 or Fortran-D do provide opportunities for runtime optimizations, but only for loosely-synchronous data parallel programs for which optimizations are easier to perform, as compared to a parallel object-oriented model which involves irregular data-driven computations, dynamic creation of tasks and asynchronous communication.

3.10 Limitations of the framework

Our framework for automatic runtime optimization has limitations due to the heuristic nature of the techniques, the complexities of post-mortem analysis, the types of applications which can be automatically optimized, and the overheads of trace collection. While some of these limitations may prevent theoretical guarantees of performance, we are focussed in this work on practically useful techniques that work for many parallel programs. The set of optimizations currently in the framework is by no means exhaustive: new optimizations for different types of problems may be useful in the future (e.g. object placement for irregular graphs). Also, the runtime optimization techniques may be integrated with compiler techniques (e.g. for dynamic grainsize control).

3.10.1 Types of applications

As discussed in Section 3.7, the validity of post-mortem analysis is affected by the unpredictability of parallel application behavior. In particular, different input sets may cause the program characteristics to vary from run to run. We target our automated optimization 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. Thus we assume that the parallel program has application-specific characteristics that remain invariant across input sets.

3.10.2 Heuristic techniques

The framework for automatic optimization we have described in the previous sections makes use of heuristics for inferring program characteristics and finding optimizations. These heuristic rules embody the experience gained by expert programmers, and have a high probability of successfully improving a program's performance. However, the heuristics are *not guaranteed* to work in all cases, by their very definition; it is hence possible that for some parallel programs there will be no performance improvement, due to a shortcoming of the heuristic.

Again, this framework does not claim to incorporate an exhaustive set of heuristics or optimization techniques for all possible classes of parallel applications. In fact, as parallel programs are written for newer applications from different domain areas, new application-specific optimization techniques are often required, suggesting that an automated optimization tool must be constantly updated as parallel programming knowledge matures. However, as discussed in

Section 3.3, most performance problems can be solved using known optimization techniques, so that it is likely that an automatic optimization tool such as Paradise will be effective for many parallel programs. The next chapter describes a representative set of optimization techniques.

The extent of automation possible with purely runtime techniques varies, depending on the type of optimization. For some optimizations such as object placement (Sections 4.3 and 4.4), complete automation is possible. For others, such as granularity control and pipelining (Sections 4.6 and 4.7.2), even though the key decisions and parameters can be automated at runtime, the runtime still needs a *control point* (where the user program hands control to the runtime system) where the optimization can be incorporated. In order to provide a new control point, the compiler or programmer may need to modify the source program (e.g. by making a runtime system call at the appropriate place), and also ensure that the original semantics are retained. However, it is still a significant simplification of the optimization process if the key optimization decisions can be automatically made by the runtime system. Thus an ideal automatic optimization system would integrate runtime techniques with compiler transformations. In this framework, we focus on automating runtime optimizations.

3.10.3 Time and space complexity

The time and space requirements for Paradise were practically observed to be very reasonable: for all the programs described in Chapter 5 the time required for post-mortem analysis was less than 10 seconds on a current generation workstation such as the Sun Sparcstation 20 (excluding the time for disk I/O involved in event graph construction). E.g. for the irregular object-array based program in Section 5.1.3 there were about 28,000 entries in the log files, corresponding to over 9,000 messages in the program, and the analysis took about 2 seconds on the Sparcstation 20.

However, since our framework depends on post-mortem analysis, it is limited by the space requirements of the event graph program representation and the time requirements of the algorithms that traverse the event graph. Hence there is a practical limit on the size of programs that can be analyzed. Paradise constructs the event graph using an adjacency list representation, where each vertex of the graph maintains a list of edges. Thus if V is the number of vertices and E is the number of edges, the space complexity of the representation is $O(E)$. This corresponds to the number of messages sent in the program.

The time complexity of post-mortem analysis depends on the various algorithms used to traverse the event graph or object-interaction graph. Most of the algorithms consist of a depth-first traversal of the graph, whose complexity is $O(E)$. A few algorithms for gathering statistics (e.g. grainsize, or number of messages sent) about each object simply require a pass through all edges, hence their complexity is also $O(E)$.

In order to find a concise pattern for generating optimization hints (as described in Section 3.8.2), some optimizations require the solution of a set of equations, and others may require exhaustive search through a space of possible pattern-matching expressions (e.g. in Section 4.4.3). However, the size of the space is not usually large (e.g. all possible block sizes for partitioning a parallel array, which is 4 for the Jacobi program in Section 5.1.1). Practically, we have not currently found this to be a limitation for the example programs described in Chapter 5. However, it is possible that in the future, for large problem sizes, this type of search will be a limitation, which may necessitate better techniques to find concise optimization hints.

In addition, two factors mitigate the few cases where there is a possibly large time complexity of post-mortem analysis in Paradise:

- The input traces need not be for a large production run of the parallel program. It is sufficient to run a test case with a smaller input size, as long as the behavior of the program for the small and large inputs share the same characteristics. This is true for many parallel programs which have application-specific characteristics that are invariant across input sets.

- For off-line analysis tools (including compilers and post-mortem analyzers), it is usually better in practice to increase analysis time if it will result in better runtime performance of the actual program. While the tool is run just once, the performance benefits may be realized for several executions of the actual program. Since Paradise is an off-line analysis tool, its performance is not critical to the performance of the parallel program itself, and the latter is our main consideration.

3.10.4 Execution tracing overhead

The process of tracing the execution of a Charm++ program involves logging the time of creation for every message, and the times for start and completion of every method. The traces are stored in a finite-size memory buffer, and are written to disk when the buffer is full.

It is possible that the space and time overhead for tracing may perturb the execution of the program, and limit the maximum length of an execution that can be traced. Several techniques for reducing trace overhead and perturbation, reconstructing actual program execution from traces and dynamic instrumentation have been explored in the literature [29, 34]. In particular, the Projections performance analysis tool developed earlier in the author's research group incorporates several such techniques, which are used in Paradise too. These issues are beyond the scope of this thesis.

Since events are traced for message creation and start and completion of the method initiated by the message, the tracing overhead effectively serves to increase messaging overhead. The increase in execution time due to tracing would hence depend on the grainsize of computations in the programs. For programs with large grain computations we may expect a negligible increase in execution time. For programs with a smaller grainsize and large amounts of communication (e.g. the Gauss-Elimination program in Section 5.1.2 which has about 500,000 messages) the overhead almost doubled the program's execution time. For programs with moderate amounts of communication (e.g. the irregular object-array program in Section 5.1.3 which had about 9,000 messages) there is a 10%-20% increase in execution time.

Chapter 4

Techniques for automating runtime optimizations

Run-time optimizations are those optimizations for which mechanisms or policies need to be carried out at run-time, necessitating support from run-time libraries in order to perform the optimization. The set of possible run-time optimizations is very large, because of the variety of programming models, parallel algorithms and application behaviors. Moreover, as new applications are developed, the necessity for newer optimization techniques arises. Each type of optimization may also be applied in many different contexts depending on the programming model and application. In this thesis we discuss a representative set of commonly used optimizations. While many isolated optimizations have been previously researched for specific domains and applications, we conduct a broad analysis of run-time optimizations from the perspective of a generic concurrent object-oriented language. As discussed in Section 2.1.1, Charm++ has the advantage of being a language designed for automation of placement and scheduling, which gives the run-time system greater control over the mechanisms and strategies for optimization.

4.1 Analyzing runtime optimizations

In this chapter we first discuss the characteristics of parallel programs that need to be discovered. The next few sections then discuss the *mechanisms* and *strategies* for each run-time optimization, and how to automatically select them. The mechanisms embody knowledge about *how* to solve a performance problem, depending on the characteristics of the problem and the kind of information available to solve it. The strategies embody knowledge about *when* a mechanism is applicable, hence guide the *selection* of a suitable mechanism for solving a given performance problem.

We analyze each type of optimization in the following manner:

- Identify or create *control points* where the run-time system can intervene in the execution of application code, to affect its performance or behavior.
- Identify or design alternate mechanisms to be applied at the control points, which can influence and improve performance in specific situations.
- Develop the strategies used to make decisions and to select between alternative mechanisms at the control points.
- Identify the information required to parameterize the mechanisms and guide the strategies.
- Develop techniques and heuristics to automatically extract the information from the event graph.

4.2 Characteristics of parallel programs

In order to select a suitable optimization strategy, it is necessary to systematically discover the characteristics of the parallel program from the event graph. We describe some important characteristics, and discuss how they are inferred from the event graph.

4.2.1 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 [29]. The phase structure is also important for load balance 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, although the total load across all the phases may be balanced. The following types of phases are commonly encountered:

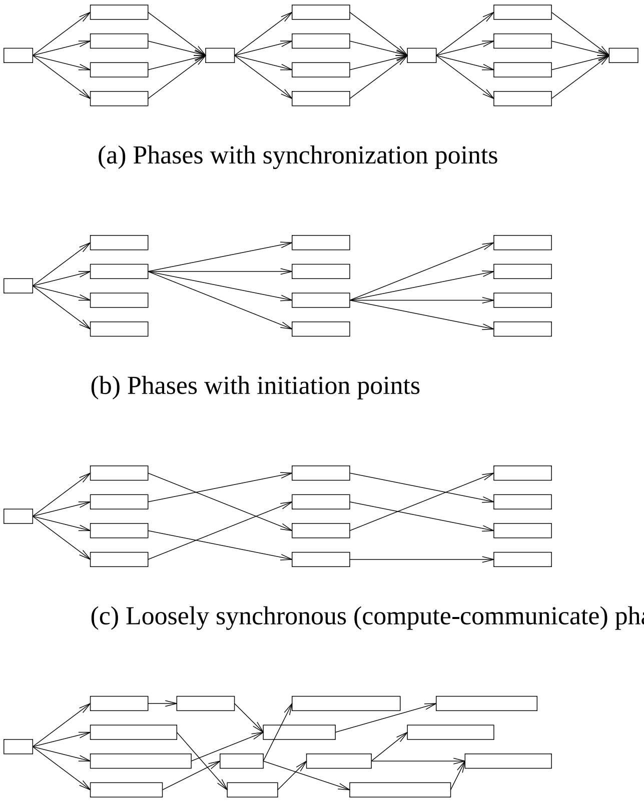
- Phases separated by synchronization points: here each phase proceeds only after all computations in the previous phase have finished (figure 4.1a).

- Phases separated by *initiation points*: here there are no synchronization points, instead each phase is initiated by a multicast from one object of the previous phase (figure 4.1b), which forms an initiation point.

- Computation phases alternating with communication phases: (figure 4.1c) 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 4.1d). 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 4.1: Different types of phases in parallel programs.

4.2.2 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, Paradise traverses the object-interaction graph (Section 3.7). 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.

- No locality exists: There are two cases when locality need not be taken into consideration while mapping objects to processors:
  - 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).
  - long-range interactions: 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: In graph-structured communication, objects communicate with neighboring objects. Usually this pattern of communication arises in spatially decomposed applications, where objects represent regions of space and communicate with adjacent regions. To differentiate between neighbor-communication and long-range communication, we use the average degree of a node in the object-interaction graph: if the degree is less than a threshold, it is worthwhile trying to partition the graph among processors to balance load and reduce inter-processor communication. We also differentiate the following two cases:

- Regular graph-structured communication: the communication pattern between objects is regular e.g. neighbor communication among objects in a multidimensional parallel object array (Section 2.2.3).
- Input dependent graph-structured communication: 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. Such communication patterns can be found in parallel discrete event simulation, and sparse matrix computations, for example.

4.2.3 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 characteristics of programs with respect to object creation are:

- Location of object creations: There are two patterns possible here. The first is *centralized creation* (one processor creates all the objects) and the second is *distributed creation* (many processors create objects). This characteristic can be easily inferred by counting the number of objects created on each processor. If object creation is distributed (e.g. tree-structured computations usually involve distributed creation), the load collection problem is more complicated, since ideally all processors require load information from all other processors. The heuristics that are commonly used include neighbor averaging (load information is sent only to a few neighbor processors), sender-initiated (load information is sent out by overloaded processors), receiver-initiated (load information is sent out by underloaded processors), and periodic (load is collected only in some phases). If object creation is centralized, the program fits into the "manager-worker" type of model, where a manager creates work and distributes it to workers. In this case, all load balancing decisions are taken at one processor, hence it may be enough to use a simpler strategy of sending load information from all processors to the creator processor.

- Time of object creations: There are two patterns possible here also. The first is *continuous creation* (objects are created continuously) and the second is *bursty creation* (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). A continuous creation pattern means that object mapping decisions need to be continuously made, so load information must be continuously available; however this also implies that sub-optimal object mapping may be sufficient, since any imbalances can always be corrected at a later time by suitably mapping newly created objects. A bursty creation pattern requires load-balancing decisions to be made at discrete intervals: load information collection can be done in a periodic manner (in the case when all objects are created just once at the beginning, no load collection is needed). However, in case of bursty creation the object mapping decisions need to be accurate; when this is not possible, object migration may be necessary to rebalance loads in the middle of the intervals.

4.2.4 Object grainsizes

The grainsize of a method is the amount of work done 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 grainsize. A large variation in grainsize complicates optimizations such as load balancing and scheduling. If all grainsizes are nearly the same, simpler strategies may suffice, or more accurate optimizations may be possible.

4.3 Optimizing dynamic object placement

Dynamic object placement is needed when objects are created dynamically throughout the execution of the program. There are two main aims of a dynamic object placement strategy: to balance load across processors; and to maintain communication locality by moving objects only when necessary and keeping heavily interacting objects on the same processor. Dynamic object placement has been extensively researched, and there are several general purpose as well as domain-specific schemes.

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.

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

4.3.1 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 are 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.

- 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. As part of this thesis we have designed a new parameterized load balancing scheme for tree-structured computations, which is described in Section 4.3.4.

4.3.2 Information required for dynamic object placement

In addition to the application-specific characteristics described in Section 4.2, 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), then 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 [52], before creating the object or after it has started execution.

4.3.3 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 to 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 [56, 57]. 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.

4.3.4 A parameterized dynamic object placement scheme for tree structured computations

The three load balancing schemes described above are general-purpose: they work for problems with any structure. However, if we know the problem structure, we can sometimes use better-tuned schemes. Recognizing this, Paradise uses a special load balancing scheme for tree-structured computations where there is some amount of uniformity in the tree.

The load balancing scheme 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 the general-purpose schemes. The types of trees which can be partitioned by this scheme are:

- Completely uniform trees: every internal node has the same branching factor, and all its child subtrees have the same load. The factorial program in 5.1.4 is an example. This kind of tree may also occur in exhaustive search applications.
- Uniform branchfactor: every internal node has the same branching factor, and child subtrees have a consistent ratio of loads. An example is the Fibonacci program in 5.1.4.
- 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.

This scheme 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 on that processor.

Paradise first determines the structure of the program (as described in Section 4.2.2). If it is tree structured, it systematically traverses the tree in a depth-first manner, calculating the loads and branching factors of all nodes. Then it analyses the branching factors and subtree loads to check if they are uniform. If so, it generates a hint including the average subtree load ratio and branchfactor. If the tree is too irregular, Paradise reverts to one of the three general purpose schemes described above.

4.4 Optimizing static object placement

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 [58, 59, 60] ; 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 (e.g. the NAS SP benchmark described in Section 5.2). 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.

4.4.1 Schemes for static object placement

The static placement strategy to be employed depends on the type of application:

1. For array-based programs (e.g. programs written using the parallel object array construct in Charm++), with regular patterns of communication, the static object placement strategy maps array elements to processors. The most popular schemes are *block* and *cyclic-k*. Other interesting schemes include the multi-partition schemes [61] for multi-dimensional arrays.

2. Applications having a large number of small grained objects with fixed positions in space: E.g. each object is a grid-point in a mesh, or each object corresponds to a particle in space. Such applications usually exhibit spatial locality (objects communicate with nearby objects), so they can be statically load balanced by assigning regions of the computational space to each processor. Since each object interacts with neighboring objects, communication between regions can be minimized by minimizing the length / area of the boundary between regions. Several techniques have been researched in the past, based on orthogonal bisection, spectral bisection, index-based mapping, genetic algorithms, linear programming, and multi-level methods [62, 63, 64]. The information required for these algorithms is the load per object and the positions of objects in space, both of which can be specified as local variables of the objects. We have developed a fast variation of an orthogonal recursive bisection algorithm, for partitioning particles in the N-body application (Chapter 5).

3. Applications which have fewer, medium-grained objects: In addition to the above strategies, the load balancing problem can be mapped to a graph partitioning problem. Each vertex of the graph represents an object, and edges represent communication between objects. Both vertices and edges have weights corresponding to their computational and/or communication loads. The aim is then to partition the vertices so that all processors have equal weights, and also to ensure that the weight of cross-processor edges is minimized. While graph partitioning is an NP-complete problem, several approximate algorithms for solving it exist. The information required for these algorithms includes the load per object and the interactions with other objects, both of which can be specified as local variables of the objects. We have developed a variant of recursive bisection for edge-load partitioning for objects in the N-body application.

For our discussion of automation techniques in the next two sections, we concentrate on static object placement for the first category of array-based programs. In the next three sections, we describe how to automate array object placement for three categories of programs: regular programs without phases, regular programs with phases, and irregular programs.

4.4.2 Automating static object placement in programs without phases

Paradise currently chooses static object placement strategies for programs written using Charm++'s parallel object array construct. Essentially, Paradise chooses a scheme for assigning array element objects to processors which partitions the array among processors so as to balance load as well as reduce inter-processor communication. In this section we consider programs without significant phases (Section 4.2.1) : i.e. programs which do not have phases or for which all objects are active in all phases, which implies that we can ignore the phase structure for the purpose of placement.

For the case where the object load and communication patterns are uniform, (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 run-time). In such cases it is possible for Paradise to automatically infer a precise pattern for mapping objects to processors. In particular, the uniformity of loads and communication implies that a block-structured decomposition will perform well[2]. Thus the main task is to find an appropriate aspect ratio for the block of elements assigned to a processor, such that communication overhead is minimized.

For regular array-based programs, Paradise partitions arrays using a (block, block, ...) pattern, where processors are arranged in a grid and a contiguous rectangular block of array element objects is placed on each processor. For each array, the amount of communication along each dimension is calculated by creating a plane perpendicular to the dimension which bisects

[2] A cyclic decomposition is usually good for distributing nonuniform loads across processors.

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 block. At run-time, the object mapping library uses this aspect ratio, the array size, and the number of processors to determine the actual sizes of the block and assign array elements to processors.

When there is more than one object array, it may be necessary to align the arrays such that objects from different arrays which communicate heavily are mapped to the same processor. We use the strategy developed by Li & Chen [65] for compilers. Each dimension of each array is represented as the node of a dimension-graph, and edges represent communication between dimensions. A graph partitioning heuristic partitions nodes in this dimension-graph so that inter-partition communication is minimized. Each resulting partition represents a dimension of a template grid (as in HPF [5]), and each object in each array is assigned to a grid point. Finally the template grid is partitioned across processors as described in the previous paragraph.

Chapter 5 gives an example of automatic array partitioning for regular programs. E.g. for a Jacobi relaxation program (Section 5.1.1) where each object communicates with its two adjacent objects in each dimension, Paradise suggests square regions for each processor. E.g. for a program where communication occurs only along one dimension, processors will get objects belonging to long slabs oriented along the direction of communication so that there is no inter-processor communication.

4.4.3 Automating static object placement in programs with phases

As described in Section 4.2.1, the phase structure of a program is important if the object activity patterns vary from phase to phase. 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, although the total load across all the phases may be balanced[3]. Currently the following load balance analysis applies only for Charm++ programs with parallel object arrays.

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[4].

A load balance constraint is specified between a pair of objects. For each phase constraints are attached to every pair of objects that are active within that phase. The existence of a constraint between a pair of objects indicates that both objects should preferably not be assigned to the same processor because they are active at the same time. Thus satisfying as many constraints as possible would ensure that the objects are evenly distributed across processors.

Figure 4.2 gives an example of a program which has a two dimensional array, with a global dependence along each column of the array. Each row of the array forms a phase consisting of element objects which are simultaneously active. Hence all objects are not active in all phases. A load balance constraint is generated between every pair of objects in the same row. If we consider three candidate mappings: row-major, column-major and block-block, the column-major mapping satisfies all constraints, hence it is the best.

Paradise uses backward reasoning (Section 3.8.1) to infer the best mapping pattern. For each known mapping pattern it generates an assignment of array element objects to phases and then evaluates how many load-balance constraints are satisfied by the mapping (a constraint is satisfied if the objects connected by it are mapped to different processors). Finally a hint containing the best mapping pattern and its parameters is generated.

The mapping patterns currently evaluated by Paradise are:

[3] If all objects are active in all phases, then the program effectively does not have phases for the purpose of object mapping, and the techniques in 4.4.2 can be applied.

[4] Currently, Paradise tries to satisfy constraints across all phases as far as possible by selecting a single best mapping. It does not remap objects between phases.

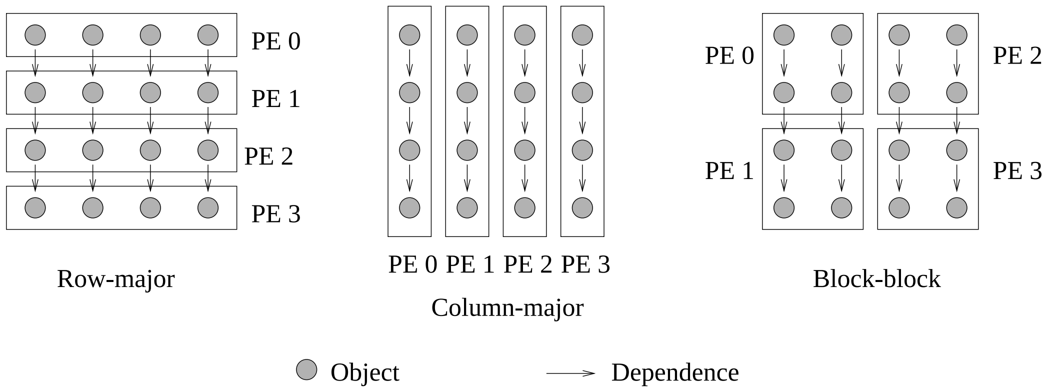


Figure 4.2: Candidate mappings for an object-array based program which has dependences along a column, causing each row to form a phase of objects that are simultaneously active.

- all block-cyclic mappings. E.g. for a 2-D array, this has the form $Map(i,j) = \frac{j}{a} MOD\ b + b * (\frac{i}{c} MOD\ d)$
- the multi-partition (Bruno-Capello) mapping scheme [61]. 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 Paradise, which parameterize the pattern. For this optimization, different values of the constants are generated by exhaustively exploring a space of possible values. We intend to add more patterns as the need arises.

Chapter 5 gives two examples of programs with phases for which Paradise automatically generated mappings: the NAS Scalar-Pentadiagonal benchmark (Section 5.2) and a Gauss-Elimination program (Section 5.1.2).

4.4.4 Automating static object placement for irregular programs

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)[5]. 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 5.1.3 gives results for an example irregular program. Note that the use of the inherited "load" variable gives an example of the advantages of object-orientation for runtime optimization (Section 3.2).

4.5 Scheduling optimizations

The scheduler in the run-time system of an object-oriented language determines the sequence of execution of method invocations for 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[6] of the scheduling strategy is to ensure that multiple independent computation paths[7] in the parallel program finish at about the same time. In programs where

[5] 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.

[6] Other aims include cache-locality enhancement by scheduling methods of the same object back-to-back [52].

[7] 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.

some computational paths are longer than others, there is likely to be a critical (longest) path. We can optimize the scheduling by executing methods on the critical path with minimal delay.

4.5.1 Mechanisms for optimizing scheduling

Two mechanisms for optimizing scheduling may be used in parallel object-oriented programs:

- 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 (Section 2.3.1). The message send operation also provides a control point to the run-time system when the priority can be set and hence scheduling can be influenced. To implement prioritization, the run-time needs to know what priorities need to be assigned to messages, and at what times in the program they should be assigned.

- Intentional idling: since computations in Charm++ are not pre-emptable, a high-priority message on the critical path may wait for a long low-priority computation to complete. Hence it is sometimes beneficial to idle a processor for a short period of time, until the critical path message arrives. To implement this optimization, the scheduler needs to know what message to wait for, and at what point in the computation.

Identifying which methods must be scheduled early is a difficult problem because no global information is usually available at runtime regarding which of the paths is the critical path. The scheduler does not know which of the methods to be invoked lies on the critical path. Scheduling mechanisms hence depend on heuristics which can use information about the critical path from the application program.

4.5.2 Heuristics for optimizing scheduling

Paradise automates optimizations for scheduling with the help of the prioritized scheduling mechanism. The heuristics used by Paradise to determine scheduling optimizations are:

1. Determine if the program has a significant critical path. The algorithm used to determine the critical path is based on a longest-path heuristic: Paradise performs a depth-first traversal of the event graph, annotating each node with the longest path from that node to the end of the program.

2. Determine the method types or message types lying on the critical path. Determine priority values to be assigned to method or message types on the critical path. The priority of a type is higher if it occurs more often on the critical path, and lower if it occurs more often on non-critical paths. Also, determine the amount of critical path delay for each of the message types. The priority of a type is higher if its messages are delayed for longer periods of time.

3. If objects can be uniquely identified (e.g. by a coordinate in a parallel object array), find an expression to relate the object's priority to its id (coordinate). First Paradise assigns priorities to objects on the critical path. The priority of an object is higher if it occurs earlier on the critical path. Then Paradise tries to find a pattern to fit all the (priority, id) pairs, i.e. for relating the object-id to its priority. Currently a linear function of the form $Priority = a * id + b$ is used to determine an object's priority from its id. The constants in the linear function are determined by generating candidate constants from two (priority, id) pairs and counting the number of pairs which fit the expression for those constants[8]. If 90% of the pairs satisfy that expression, it is chosen. A hint containing the expression is generated.

[8] A more sophisticated scheme would be to use least-squares fitting.

Note that this optimization provides an example of the use of a set of messages (as in the lattice described in Section 3.8.2) of the same type, as well as an example of pattern matching. Section 5.1.6 gives an example of automatic scheduling optimizations for the Gauss-Elimination program.

4.6 Optimizations for grainsize control

When there is dynamic creation of work in a parallel program, the grainsize of each object created should be large enough to amortize object-creation overheads, yet small enough that there is enough parallelism, i.e. there are sufficient objects for load balancing. Requiring the programmer to explicitly specify grainsize is cumbersome, although it usually provides acceptable performance [66]. There has been research into compiler techniques for automatically determining the grainsize of computations (e.g. by merging fine-grained pieces of work [53]), however, the grainsize decision often needs to be made differently depending on run-time load conditions. For example, in a tree-structured computation, a small initial grain size helps to distribute work among processors quickly; when all processors are working on a sub-tree, a large grain size helps to reduce object creation overheads and maintain locality; finally, if sub-trees have unequal loads, creating objects for sending to underloaded processors helps to maintain load balance.

The granularity optimization strategy needs to be closely tied to the dynamic load balancing strategy in order to detect when new work should be created. Thus the information needed is the same as for load balancing: (a) the load on different processors, and (b) the granularity of objects, which allows the utility (work v/s overheads) of the object to be evaluated.

4.6.1 Mechanisms for runtime grainsize control

We now analyze two mechanisms for grainsize control. For objects of fixed granularity, the run-time can be allowed to decide whether an object should create a child object or do the work within itself as sequential function. The control point for this mechanism can be provided to the run-time in the form of a "ShouldCreateObject()" function, which is implemented by the run-time and called by an object to determine whether it should create a new child object[9]. One strategy for the run-time would be to create new objects only when other processors have relatively low loads, to ensure sufficient parallelism. Thus we can develop a granularity control mechanism in which the application performs chunks of sequential computation before checking with the run-time system whether a new object should be created. The code below illustrates the use of this optimization.

```
ChareType::DoWork()
{       ···Do some work···
        MsgType *m = new MsgType(···) ;
        if ( ShouldCreateObject() )
                newchare ChareType(m) ; // create a new parallel object
        else
                new ChareType(m) ; // do work sequentially
}
```

Note that for tree structured computations, it is better to send nodes at shallower depths in the tree to other processors, because they have larger grainsizes [67, 68]. However, in order to do this in the most general and efficient manner, the runtime should be able to dynamically switch from sequential work through local function calls (on stack) to parallel work via remote objects. Currently the Charm++ runtime does not support this functionality. Hence the

[9]Note that this control point only allows the runtime to automatically determine the object creation decision; the compiler or programmer must still modify the source program and ensure that the original semantics are retained.

grainsize optimization strategy currently does not support decisions based on the depth of the object in the tree.

The second mechanism for grainsize control is based on parameterized granularity, where the amount of work done in a method can be controlled by the run-time system. Parameterized methods can be either provided by the programmer or generated by the compiler, and are useful in other contexts too (e.g. to set the degree of pipelining as explained in Section 4.7). If the total load of an object is known, the run-time can either request the object to execute a chunk of its load sequentially, or create parallel objects by requesting the object to split itself into child objects. The chunk-size can be specified as an extra argument to the method in the object or as a "global" parameter which is set by the run-time strategy. One strategy is to set the chunk size to half the remaining work of an object, when all processors are busy. If some processors become idle later, this leaves some load for redistribution[10]. Parameterized code-blocks can be used for both iterative and recursive computations. Adaptive grainsize control has been manually programmed before with good results [57, 68].

4.6.2 Automating grainsize optimization

Paradise automates the two optimizations for grainsize control described in the previous subsection. The first, for objects of fixed granularity (i.e. whose granularity cannot be affected by the run-time system), is made possible by having the user program call a "ShouldCreateObject" function in the run-time library. This function allows the run-time system to determine whether an object should be created or not by using load information from other processors, with the help of the dynamic object placement strategy. If other processors have sufficient work, the runtime system prevents the creation of a new object. Otherwise a new parallel object is created which can possibly be sent to other processors.

[10] Because the Charm++ scheduler is not pre-emptive, objects should not be so large that the run-time system cannot respond to changes in processor loads.

Paradise determines whether the grainsizes of objects are small enough to cause concern about overheads. If so, it generates hints to enable the above grainsize optimization mechanism in the run-time system, for the particular object type. (In future, it may be possible to give a hint to a compiler to automatically transform selected objects for increasing granularity or for invoking run-time functions to select granularity).

The parameterized grainsize control mechanism is described in conjunction with the pipelining optimization in Section 4.7.2.

4.7 Communication optimizations

Communication optimizations are useful in many parallel programs to reduce communication volume and overheads, as well as reduce the latency of remote data access.

4.7.1 Mechanisms for optimizing communication

There has been a lot of research in optimizing communication for data-parallel applications written in HPF/Fortran90D [69, 45], in which the compiler generates a good communication schedule or generates calls to run-time communication libraries. The run-time optimizations described below have been commonly observed to be useful for diverse applications, in a concurrent object-oriented context. Some of them can also be profitably done together for the same set of messages. For communication optimizations, the control points for the run-time are from the time a message is sent out, through the time the scheduler dispatches the message for processing.

- *Message combining or aggregation* : 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 the overhead due to message transmission latency. In order to do message combining, the run-time needs information about precisely which messages to combine. Essentially, when a message-send call is made, the run-time needs to know whether to send the message right away or delay the message so that it can be combined with another message to the same processor.

- *Pipelining* : this is the opposite of combining; when a large message is sent 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, reducing idle time by overlapping communication and computation. The problem here is similar to the grainsize optimization problem: the run-time should have information about how many pieces the computation should be divided into (the degree of pipelining), or equivalently, the grainsize of a piece, depending on communication latency. The run-time should also have information about when to do pipelining, depending on whether the receiving processor is waiting for a message.

- *Identical message combining* : in some applications, an object on one processor may need to multicast data to many remote objects, some of which may be on the same remote processor. Instead of sending multiple identical messages to the same processor, it is possible to send just one message and replicate it on the receiving processor, and thus reduce messaging overhead. Identical message combining has been implemented in the multicast operation for parallel object arrays.

- *Message caching* : this is a very useful optimization for private-memory machines. A lot of communication in applications is due to requests for remote data, much of which does not change over time. Such read-only data can be broadcast from the processor creating it, and cached on all processors. Such a construct already exists in Charm++. When only few processors need a (possibly large) read-only datum, a request-reply protocol is usually followed. Here too, caching can help by preventing multiple requests from a processor for the same datum. In order to do caching, the run-time needs to identify data so that it can match requests with existing cached data. Also, it needs to know whether to cache a message or not, and when to update/invalidate it.

- *Advance messages* : In most applications, remote data is accessed by a request-reply protocol, which requires two messages. Not only does this increase overhead, it may also result in the requesting processor idling while the data is fetched. Instead, if the producer of a datum knows which processor is the consumer, the data can be sent in advance to the consumer, eliminating a message. At the requesting processor, the advance data is effectively cached: a request message can retrieve advance data from the cache. Advance messages have been profitably used in the N-body application described in Chapter 5.

- *All-to-all personalized messages*: this is a commonly needed function (for example, while doing a matrix transpose operation), in which each processor sends a message to every other processor. For P processors, this would normally require P^2 messages, which could cause a lot of network contention and resulting slowdowns. Instead, the messages can be transferred using variations of a dimensional exchange protocol. To do this, the run-time needs to identify an all-to-all send operation, collect all P messages on a processor, and then implement the faster protocol.

4.7.2 Automating communication optimizations

In this work, we chose to focus on automating the the message pipelining and message combining optimizations described in the previous subsection.

4.7.2.1 Message pipelining

For this optimization, we need to determine the degree of pipelining, i.e. the number of pieces a computation should be broken into so that its results can reach consumers earlier. The degree of pipelining depends 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. Thus the optimal degree of pipelining is input-dependent. In order to enable this pipelining optimization, the program must be written in a parametric form so as to give the run-time system a control point where the degree of pipelining can be determined. This control point is provided in the form of the function "GetPipelineDegree()" which is called by the user method and returns the proper degree of pipelining for the method.

The analysis performed for automating pipelining is similar to the one described in [29]. Paradise looks along the critical path of the program for methods which execute after a significant idle period on their processor. Often there is a chain of methods of the same type (e.g. corresponding to a sweep over the computation space, in spatially decomposed applications), all of which execute after a long idle period ; they can usually be pipelined by splitting the first method in the chain into smaller pieces. Paradise finds a predecessor method which may be pipelined, and generates a hint to pipeline the chosen methods, specifying the optimal degree of pipelining. The optimal degree of pipelining is computed using the formula described in [29] (Figure 4.3), presented here for completeness:

$k = \sqrt{\frac{\beta l+g}{s}}$, where k is the degree of pipelining, β is the time to transfer one byte across the network (i.e. $\beta = 1/bandwidth$), l is the length of the message, g is the grainsize of the computation to be pipelined, and s is the scheduling overhead on the receiving processor.

4.7.2.2 Message combining or aggregation

Paradise determines the message types in the parallel program whose instances may be combined. It also gives a hint regarding the number of messages to combine.

Paradise identifies a set of messages as a candidate for combining when they are sent from the same source processor to the same destination processor within a short period of time.

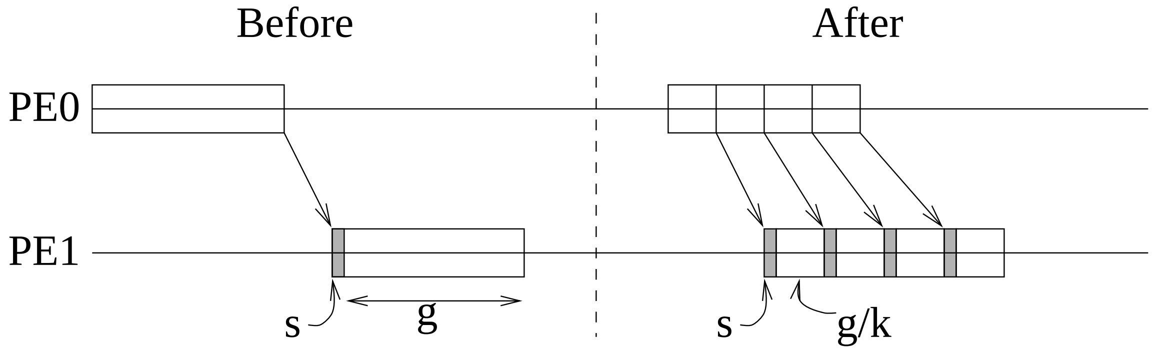


Figure 4.3: Message pipelining.

It then checks if the destination processor consumes the messages immediately. If not, the messages may be safely combined, and a hint to that effect is generated. 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 destination processor.

A special case where combining is easy to apply is when the program has a phase structure with synchronization points, as is common in data-parallel scientific applications. A synchronization operation among all elements of a parallel object array usually results in each element sending a message to one processor. In such a case, the number of messages to combine is set to the number of array elements on a processor. Often, the computation does not provide opportunities for message combining in every phase, but in periodically recurring ones (e.g. corresponding to outer loops). The phases in which the combining should occur are also found in the following manner: from the event graph Paradise determines the list of phases after which there is a synchronization. Then it tries to fit the phase numbers into a linear pattern of the form $phasenum\%a + b = 0$, and finds the constants a and b. Finally a hint including all this information is generated. Section 5.1.7 gives an example of this optimization being performed automatically.

Chapter 5

Applications

In this chapter we give examples of programs written using Charm++ which demonstrate the benefits of object-orientation in parallel programming, and the increase in performance due to run-time optimizations. Most of them were also automatically optimized using Paradise.

5.1 Evaluation using simple benchmarks

In this section we present several simple example programs which are used to demonstrate the ability of Paradise to automatically infer and enable runtime optimizations, thereby improving performance without extra programming cost. For each program in this section we first obtained the performance of the original unoptimized program (without tracing overhead), then executed the program with the event tracing switched on, collected the traces and fed them into Paradise, and finally used the hints generated by Paradise for re-executing the program to get the automatically optimized results.

5.1.1 Programs without significant phases

For programs which do not have phases or whose phases are not important for load balance, Paradise applies the techniques described in Section 4.4.2. Here we demonstrate automatic object placement for such a program. The example program performs Jacobi relaxation using a 5-point stencil computation. The Charm++ program consists of chare objects organized in a two dimensional parallel array, with each object being responsible for a sub-domain of the computational grid. Objects communicate with four neighboring objects (*north, south, east and west*) in order to exchange boundary rows. After completing one step of relaxation, all objects participate in a reduction to determine if convergence has been reached.

Paradise was able to deduce that the program has phases which are not significant for object placement. However, there is significant inter-object communication. Paradise then analyzed the communication patterns in terms of the communication cost along each dimension of the array. The ratio of costs along the two dimensions gives the aspect ratio of the block of objects to be assigned to each processor. For the Jacobi program, there is equal communication along the two dimensions, hence the ratio is 1.0. Finally, Paradise generated a hint for mapping the parallel object array using block partitioning, with the computed block aspect ratio.

Table 5.1 presents times in seconds for the default placement strategy (cyclic-cyclic) and automatically inferred placement strategy (block-block) generated by Paradise for the Jacobi program on 16 processors of the CM-5. There were 256 objects in the parallel object array, with each object having a 4x4 sub-domain. The relaxation took 528 iterations to converge. The execution time on 1 processor was 129.5 seconds.

Strategy	Default	Automatic
Time (sec)	29.55	24.54

Table 5.1: Time for Jacobi program, for the original (default placement) run and the automatically optimized run.

5.1.2 Programs with significant phases

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. The simplified pseudo-code for the row object class is:

```
chare class row : public array {
        int myrow ;                 // Number of my object's row
        double *elements ;          // Actual matrix elements for my row
    entry:
        row()
        {       ... constructor ...
                if ( myrow == 1 ) BroadcastMyRow() ;  // start elimination
        }
        BroadcastMyRow()
        {       ... make elements[myrow] = 1.0 by scaling my row ...
                ... send my row to ReceiveRow function in rows myrow+1 to N ...
        }
        ReceiveRow(RowData *r)
        {       ... eliminate next column in my row ...
                if ( rows 1 through myrow-1 are done )
                        BroadcastMyRow() ;
        }
}
```

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 loosely synchronous 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. Load balance constraints are generated between objects in a phase, and each of the candidate mappings (block/cyclic for different values of the block size) is evaluated on the number of constraints matched, as described in Section 4.4.3. The mapping pattern which matches the load-balance constraints best corresponds to a block size of 1, i.e. a cyclic mapping (any block size greater than 1 will cause more constraints to be left unsatisfied). 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 5.2 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, the default (cyclic) mapping, and the automatically optimized (cyclic again) mapping. The execution time on 1 processor was 534.7 seconds. In this case the default mapping fortuitously happens to be the best, and Paradise cannot do better. The important point here is not the relative gain in performance but the fact that Paradise deduced a mapping that we know from experience is good for the Gauss-Elimination program.

Strategy	Random	Default	Automatic
Time (sec)	38.72	34.90	34.90

Table 5.2: Time for Gauss-Elimination for the original (random placement), default (cyclic placement) and automatically optimized (cyclic placement) versions. Automatic prioritization (Section 5.1.6) was used for all runs.

5.1.3 Irregular object-array based programs

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 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 5.3: 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. The program was also modified to set the load variable in the base class "loadarray" in the constructor of each object. During the next run of the program, the ORB library was initiated at the specified phase, which 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 5.3 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.

5.1.4 Programs requiring dynamic object placement

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 4.3.1). Table 5.4 presents results from running the program on 16 processors of the CM-5, with 137 objects being created.

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

Table 5.4: Time (in milliseconds) with the default and automatically chosen load balancing strategies for the variable-grainsize program.

Heavily communicating objects:

This is an artificial program which creates a number of objects of the same grain-size, but 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 5.5 presents results from running the program on 16 processors of the CM-5, with about 5200 objects being created.

Regular tree:

This is a naive doubly recursive program to compute the N^{th} Fibonacci number. It creates a binary tree of objects. Paradise was able to deduce the following characteristics with respect to placement:

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

Table 5.5: Time (in milliseconds) with the default and automatically chosen load balancing strategies for the heavily-communicating objects program.

1. the program has dynamic object creation
2. there are no phases
3. the communication overhead is not significant
4. the program has a regular tree structure with a branching factor of 2 at all internal nodes. The ratio of loads of the two sub-trees is 0.62 on average.

Hence it suggested the special uniform-tree placement strategy. Table 5.6 presents results from running the program on 16 processors of the CM-5, for computing the 20^{th} Fibonacci number with a grain-size of 10 (i.e. all leaf nodes in the tree compute the 9^{th} or 8^{th} Fibonacci number). The total number of objects created was 465.

Strategy	Roundrobin (default)	Uniform-tree (automatic)
Time	69	29

Table 5.6: Time (in milliseconds) with the default and automatically chosen load balancing strategies for the Fibonacci program.

5.1.5 Programs with many small objects

We used Paradise to automatically optimize the grain-size of the Fibonacci program described in the previous subsection. Paradise detected that the average grainsize of objects for the program was not sufficiently large, and generated a hint for optimizing grainsize. The load balancing strategy for trees read in this hint, and accordingly disabled object creation after the tree had been expanded sufficiently deep, and there was at least one internal node (a subtree) assigned to each processor. The number of objects created was only 31 (as opposed to 465 earlier). Table 5.7 shows results from running the default and automatically grainsize-optimized versions of the Fibonacci program on 16 processors of the CM-5.

Strategy	Default	Grainsize optimized
Time	29	19

Table 5.7: Time (in milliseconds) for the default (no grainsize control) and automatically grainsize-optimized versions of the Fibonacci program.

5.1.6 Programs with critical paths

The Gauss-Elimination program described in section 5.1.2 has a critical path consisting of the initiation points when a row broadcasts itself to all rows below it in the matrix. (Note: it is important to optimize the critical path scheduling because there are several objects on a processor, each of which may have messages from other rows waiting for it). Each object occurs exactly once on the critical path, and the object with index (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 indices in the parallel object array. The priority expression matching routine now tries to relate the object-index to its priority. Since Charm++ assigns higher priority to lower numbers, the priority of an object can be set to its row-number. 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 running on 32 processors of the CM-5 (for a 1000x1000 matrix) with and without prioritization is given in table 5.8.

Strategy	No Priorities	Automatic Priorities
Time (sec)	43.58	34.90

Table 5.8: Time for Gauss-Elimination for the original (without priorities) and automatically prioritized versions (cyclic placement was used for both runs).

5.1.7 Communication-limited programs

Message Combining (Aggregation):

We use the Jacobi program described in Section 5.1.1 to demonstrate automatic message combining. The program consists of phases of neighbor communication alternating with reduction operations. Our simple implementation of the program performed the reduction by having each chare object send a message to the *main* chare on processor 0. Thus processor 0 receives a large number of small messages.

Paradise detected that the program had synchronization points, and that all array elements send a message to the synchronization points. It then determined which phases send messages that could be combined by finding the values of the phase selection constants p_a and p_b (as described in Section 4.7.2). Finally it generated a hint to combine messages to processor 0 which originated from the specified phases. The number of messages to combine is the number of elements of the parallel array on a processor.

Table 5.9 presents times in seconds for the initial and automatically optimized versions of the Jacobi program on 16 processors of the CM-5. There were 256 objects in the parallel object array, with each object having a 4x4 sub-domain.

Message pipelining: We demonstrate message pipelining using a Charm++ program for computing the overlay of two sets of non-overlapping polygons. The aim is to generate a new set of polygons consisting of the geometric intersection of the two input sets. This is an

Strategy	Before combining	After automatic combining
Time (sec)	50.57	24.54

Table 5.9: Time for Jacobi program without message combining and with automatic message combining.

application that frequently arises in geographical information systems. The program reads in the first input polygon list and distributes it equally to all processors. It then reads in the second list and pipelines it through all the processors, with each processor computing the intersection of the second list with the polygons of the first list owned by it. Thus the performance of the program crucially depends on the degree of pipelining.

Paradise detected that the program had a long critical path traversing all the processors, in which the chain of "overlay" methods executed after a long idle time. Hence it decided to pipeline overlay methods and their predecessor "start" method, which started the pipe. The degree of pipelining was determined using the formula from [29], and a hint to that effect was generated. At runtime, the "start" method called the "GetPipelineDegree" function to get the degree of pipelining (which was computed by the runtime library using the hint from Paradise), and broke up the second polygon list into small chunks before pipelining them through the processors. Table 5.10 presents results for the polygon overlay program for the non-pipelined (i.e. sequential) version, the automatically pipelined version, and the best-case manually pipelined version. The program ran on 32 processors of the CM-5, and computed the overlay of two lists of 15,000 polygons each.

Strategy	No pipelining	Automatic pipelining	Best (manual) pipelining
Time (sec)	526.4	15.37	15.09

Table 5.10: Time for polygon overlay program without pipelining, with automatic pipelining and with manual pipelining.

5.2 NAS scalar-pentadiagonal benchmark

The NAS Scalar Pentadiagonal (SP) benchmark [71] is one of three Computational Fluid Dynamics benchmarks in the NAS benchmark suite. It is intended to represent the principal computation and communication requirements of CFD applications in use today.

The SP benchmark involves the solution of multiple independent systems of scalar pentadiagonal equations which are not diagonally dominant. The computational space is a three-dimensional structured mesh consisting of 64 x 64 x 64 grid points. The method used is an iterative Alternating Direction Implicit (ADI) method. In each iteration there are three "sweeps" successively along each of the three coordinate axes. Thus the method involves global spatial data dependences.

Our main objective in implementing the NAS SP benchmark was to develop a code that could be used to easily experiment with different domain decomposition strategies. We also wished to leverage the object-orientation provided by Charm++ to develop reusable abstractions that would simplify the process of developing parallel applications.

5.2.1 Parallelization schemes

The steps in the the numerical algorithm [72] which are significant for parallelization are:

- Computation of the RHS vector of the partial differential equation. Each grid point in the cubical mesh needs values of the U matrix from two neighboring grid points on either side, in each of the three dimensions. This corresponds to six "parallel-shift" operations.
- Solution of a system of linear equations in the x-direction. Each grid point initially needs values from two succeeding grid points in the x-direction (corresponding to a shift operation in the negative-x direction). Then there is a sweep along the positive-x direction in which each grid point computes values that are needed by the next two points.
- Solution of a system of linear equations in the y-direction. This is similar to the previous step, except that communication is along the y-direction.
- Solution of a system of linear equations in the z-direction. This is similar to the previous step, except that communication is along the z-direction.

Parallelizing these steps requires decomposition of the three-dimensional computational array among processors. This decomposition must be done so as to balance computational load across processors as well as reduce inter-processor data communication.

Three of the most common methods used to parallelize ADI methods are [73]:

- Pipelined static block decomposition: each processor is statically allocated a contiguous three-dimensional block of grid points for the entire length of the computation. The block is made as close to cubical as possible to minimize the amount of communication (which is proportional to surface-area of the block). During the sweeps, each processor receives boundary data from the previous processor in the sweep direction, computes its data, and sends its boundary data on to the next processor. In order to reduce idle times while processors wait for data from previous processors, the computation is pipelined: each processor works on a slice of its grid points, sends the resulting boundary on to the next processor, and then goes on to the next slice. The disadvantage of this decomposition is that many processors idle at the beginning and end of the sweeps; moreover, there are many small messages sent between processors corresponding to the boundary data for each slice, which could cause significant overhead on machines with large message latencies.

- Transpose-based dynamic block decomposition: the three-dimensional mesh is divided into slabs (Figure 5.1) oriented along the X direction first. After the X-direction sweep completes a transpose operation is done to orient the slabs along the Y-direction, in preparation for the Y-sweep. Finally, a third transpose operation is needed before the X-sweep of the next iteration. Thus there are a total of three transpose operations needed per iteration. The advantage of this method is that computations within each sweep are completely local to a processor. However, the transpose operations between sweeps can result in significant overhead on bandwidth-limited machines.

- The multi-partition or Bruno-Capello decomposition [61, 74]: this is a static decomposition where the computational mesh is divided into cubes, and each cube is assigned to a processor such that all processors are active at all stages in each of the three sweeps. In other words, each coordinate plane in the computational space contains cubes on all processors. Thus processor loads are balanced during all stages of all sweeps, and also no transpose operations are needed. The minimum number of cubes needed for this decomposition is $P^{3/2}$ (where P is the number of processors), so that each processor has $\sqrt{P}$ cubes (Figure 5.1). The cube with coordinate (i, j, k) is allocated to processor $(i-k)\%s + s((j-k)\%s) + 1$, where $s = \sqrt[3]{P}$ and $1 <= i, j, k <= s$. The tradeoff in the multi-partition method is that computations within a sweep involve cross-processor messaging.

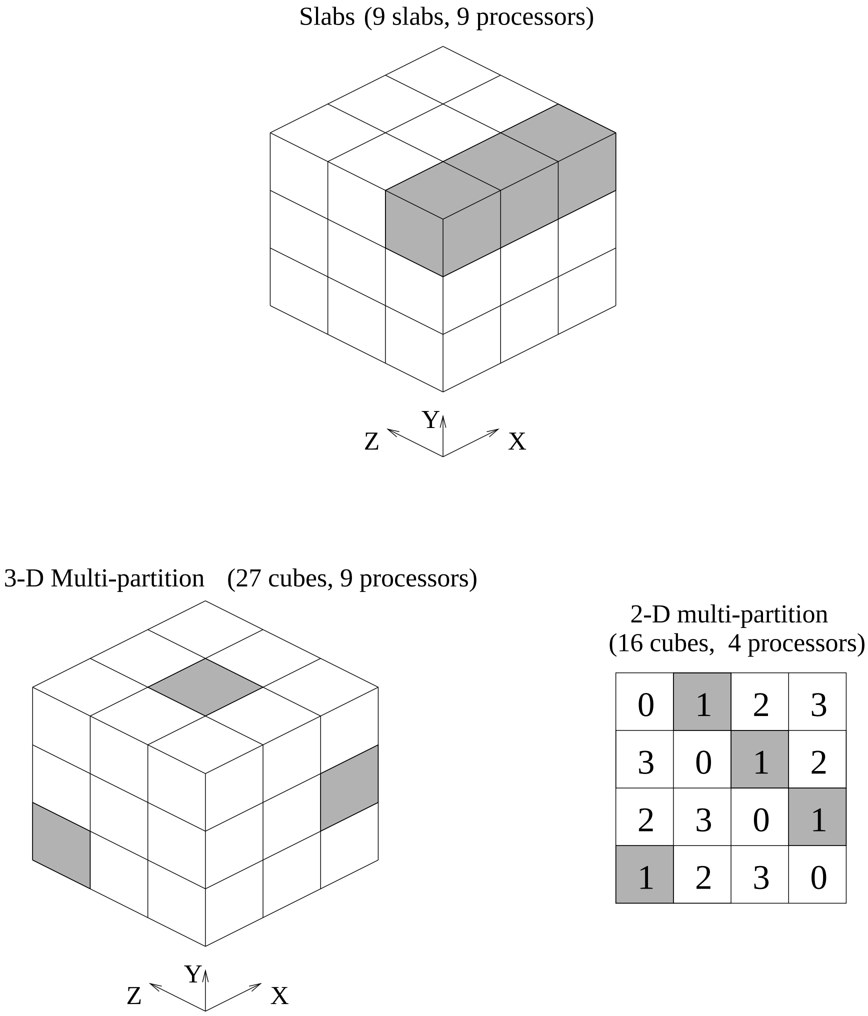


Figure 5.1: Different mapping schemes for the NAS scalar pentadiagonal program. The shaded areas represent regions that are assigned to the same processor.

5.2.2 Implementation using parallel object arrays

We developed the following abstraction for the NAS SP benchmark code: The computational space is represented as a three-dimensional array of cubical sub-spaces. Each cube is represented by a parallel object in Charm++, which communicates with other cubes by sending and receiving messages. Thus the parallel program consists of a network of communicating objects.

The different decomposition/mapping strategies are expressed by simply specifying a different mapping function for the parallel array. E.g. the mapping function for the multipartition (Bruno-Capello) decomposition is:

```
int MultiPartitionMapFn(int arrayid, int i, int j, int k)
{       // return processor number owning object (i,j,k)
        return ( XArraySize*((i-k+XArraySize)%XArraySize) +
                     (j-k+YArraySize)%YArraySize ) ;
}
```

For the transpose method, all adjacent cubes along the direction of the sweep are mapped to the same processor. E.g. the mapping function for the sweep along the X-axis is:

```
int XSweepMapFn(int arrayid, int i, int j, int k)
{
        return ( ZArraySize * j + k ) ;
}
```

The transpose is effected by simply doing a remap operation on the parallel array between sweeps, with the mapping function corresponding to the orientation of the next sweep. Thus we have a very flexible, elegant code which allows us to concentrate on experiments with the application, instead of getting involved in the details of implementing the decomposition.

The asynchronous migration facility provided with parallel arrays allows us to further optimize the transpose method by overlapping communication and computation. Each cube object migrates itself as soon as it has completed its work along one sweep. Thus the communication overhead of transferring its data to another processor is overlapped with the computation performed by other cubes. This overlap gives significant performance advantages over the traditional loosely-synchronous (separate phases of computation and communication) implementations.

5.2.3 Automatic mapping using Paradise

When the NAS SP program's traces were fed into Paradise, the placement analysis heuristics recognized that the program had loosely synchronous phases. Since a sweep starts from one face of the 3-dimensional computational space and proceeds along one dimension, a phase consists of all cube objects lying on a face of the computational space, i.e. all cubes lying on a plane perpendicular to the direction of the sweep. Since each phase does not involve all the cubes, Paradise constructed a set of constraints indicating that cubes within a phase should be evenly distributed among processors. On evaluating the known mapping patterns with respect to these constraints, the multi-partition (Bruno-Capello) pattern was found to best fit these constraints, so a hint to that effect was generated.

5.2.4 Performance results

Table 5.11 presents performance results for the different object-mapping schemes in the Charm++ implementation of the NAS SP benchmark on the Intel Paragon. The results are for problem size A as specified in the NAS benchmark documents [71]: the computational space is a 3-dimensional array of 64 x 64 x 64 grid points. *Sync-Transpose* is the block mapping which does a transpose using the parallel object array synchronous remap operation. *Async-Transpose* is the block mapping with asynchronous transpose (migration) of parallel objects for moving data between sweeps. *Multipartition* is the multi-partition mapping which was automatically generated by Paradise. The user program in this case did not specify a mapping function, which was automatically read in from the optimization hints file generated by Paradise. Timings on 1 processor were not available due to memory restrictions.

The results show that the Multipartition mapping is the best overall, with the Async-Transpose and Sync-Transpose decompositions being successively worse. The important point here is that Paradise chose the best (Multipartition) mapping, which has also been established as the best scheme in the literature [73].

Processors	4	16	64	256
Sync-Transpose	-	8.08	3.01	1.98
Async-Transpose	-	7.81	2.54	1.40
Multipartition	24.63	7.60	1.98	1.00

Table 5.11: Time (in milliseconds) for different decompositions for the NAS SP benchmark (size A) on the Intel Paragon.

5.3 Adaptive fast multipole algorithm

In this section we describe the design and implementation of a parallel adaptive fast multipole algorithm (AFMA) for N-body problems [75]. Our work is one of the first parallel implementations of the AFMA on distributed memory computers. This work on this AFMA application provided the motivation for many of the optimization ideas in this thesis, and preceded the development of Paradise. The objectives for this work were :

- Development of a large real application using Charm++. Several Charm++ features such as object-orientation, message-driven execution, etc were found to be useful in the development of the AFMA code.
- Design of several runtime optimizations to validate our hypothesis that runtime optimizations are required for achieving good performance, especially for irregular and dynamic applications.
- Evaluation of the performance benefits of several runtime optimizations, including overlap of communication with computation, communication reduction, overlap across stages of the program, static load balancing, and dynamic redistribution of load using runtime information.

- Research in parallel N-body algorithms : our code does not just implement existing ideas but includes new techniques for improving performance.

Our work has demonstrated that the AFMA can be efficiently implemented on parallel computers, including distributed memory ones. We have achieved this by designing a version of the AFMA that is more easily parallelized, by using better strategies for parallel operations, and by implementing our algorithm in an object-oriented, message-driven manner.

5.3.1 The N-body problem

The N-body problem is an important core problem arising in several simulation applications in astrophysics, molecular dynamics and fluid dynamics. The computation in the N-body problem consists of calculating interactions between all pairs of bodies (particles) in the system. The order of complexity for a simple formulation of this problem would thus be $O(N^2)$, for a system consisting of N particles. A large amount of unstructured, fine-grained parallelism is present in this problem. however, exploiting this parallelism efficiently on massively parallel computers is a challenging task.

There exist good algorithms for solving the N-body problem, which make use of a key insight into the physics of the problem : a group of particles can be approximated by a single particle (analogous to their center-of-mass) if they are sufficiently far away from the particle at which their force is being evaluated. For N particles, the Fast Multipole Algorithm due to Greengard and Rokhlin [76] claims to have an $O(N)$ time complexity with a rigorous bound on error. Other methods include the $O(NlogN)$ algorithms due to Appel [77] and Barnes-Hut [78]), the particle-in-cell methods [79], and the simple $O(N^2)$ all-pairs algorithm. Some implementations also use a Distance Class method [80] which has larger time steps for computing interactions between atoms at greater distances. Although the problem sizes at which the Fast Multipole method would outperform the Barnes-Hut method is not clear, there has recently been a lot of interest in implementing both algorithms on parallel machines.

The FMA exploits the "center-of-mass" concept by approximating the effect due to a *well-separated* (sufficiently far away) group of particles by a *multipole expansion* [81], which is a refined formalization of the center-of-mass, leading to provable error bounds. Interactions are computed between particles and groups of particles, as well as between different groups of particles. In order to partition the particle set into groups, the FMA recursively divides the computational space into cubical "cells", which are ordered hierarchically in a tree. For the three dimensional problem an octtree is generated. In the *non-adaptive FMA*, a uniform grid is imposed on the computational space, resulting in a complete tree whose leaves all have the same depth. However, this is unsuitable for non-uniform particle distributions. Hence the *adaptive FMA* (AFMA) [82] divides cells until the number of particles in a leaf cell is less than some specified *grain-size*, leading to an irregular tree which is deeper in regions corresponding to greater particle densities.

Typical N-body systems that are sought to be simulated in molecular dynamics and astrophysics have $10^4 - 10^7$ particles, and simulation is needed for $10^4 - 10^6$ timesteps. Thus N-body simulations can be computationally very demanding. Parallel computers are hence very useful for simulating larger systems over longer time durations. Several researchers have explored the issues in parallelizing N-body algorithms. However, the adaptive FMA is difficult to efficiently program on parallel computers, especially for private memory ones. This is because of the difficulty of achieving good load balance and spatial data locality simultaneously, distributing shared data structures such as the tree among processors, overcoming communication latencies arising due to the inherent irregularity and unpredictability of the data access patterns, and reducing communication volume. There are parallel implementations of the non-adaptive FMA on shared and distributed memory computers, including the work by Board and others [83, 84], and of the adaptive FMA on shared memory computers [85]. Our work is one of the first parallel implementations of the AFMA on message passing computers.

Our design for the parallel AFMA program consists of the following stages :

1. Partitioning and exchange of particles among processors.
2. Construction of the local AFMA tree, and exchange of subtrees with neighboring processors.
3. Partitioning of pair-wise load to ensure load balance.
4. Computation of the different types of interactions, including the upward and downward passes in the AFMA tree.

The next few subsections describe the run-time optimizations we have incorporated in each of these steps.

5.3.2 Partitioning and tree construction

The aim of parallel partitioning is to assign a region of particles to each processor which balances loads and at the same time minimizes inter-processor communication. We optimize the partitioning by obtaining the computational load for each particle from the previous iteration of the FMA since it is likely that the load does not change significantly from one iteration to the next. The technique used is similar to [85] where the measured load of a cell is distributed among its particles. This provides a very effective means of adaptively redistributing load, and results in must better load balance (Section 5.3.6 presents results for this optimization).

Since particles interact more closely with nearby particles, the spatial relationship between particles should guide the partitioning. Thus a convex region with low boundary surface area is likely to require less communication with other regions. In our implementation, we have used a fast implementation of the Orthogonal Recursive Bisection (ORB) algorithm.

Orthogonal recursive bisection is a well known technique which recursively partitions the computational space by planes parallel to the coordinate axes [86]. The partitions resulting from ORB are rectangular and thus convex, resulting in less communication with neighboring partitions. The direction of bisection is chosen such that every cell is bisected along its longest dimension, ensuring that the ratio of surface area to volume is as low as possible. At the end of the partitioning particles are physically moved to the processors owning the partitions they belong to. If there are P processors, the number of parallel bisection operations needed is $P-1$. However, all these bisections need not be serialized. In fact, the critical path involves only $log_2 P$ bisections, corresponding to the height of the binary tree formed by hierarchically ordering the partitions. Hence we overlap the parallel bisection operations to significantly reduce idle times of all processors. As soon as one partition has been bisected, the bisections for its child partitions are started concurrently. This is done with little programming effort because of the asynchronous message-driven model of our design.

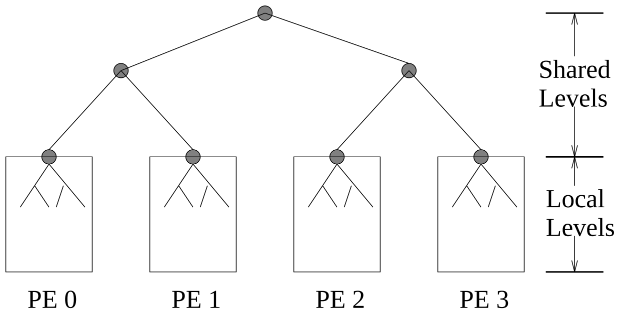


Figure 5.2: Local and shared levels in the AFMA tree.

At the end of partitioning, processors exchange particles so that each processor has its own set of particles. A copy of the top $log_2 P$ levels of the AFMA tree (which is the tree formed by ORB) is also made on all processors, so that all processors have the root cells of all other processors' local trees. Now each processor proceeds to construct its own local part of the tree by recursively dividing the space assigned to it by ORB. The division continues till the load on each leaf cell is less than some pre-specified grainsize. Figure 5.2 describes the relation of the shared and local trees.

After each processor builds its local tree, it needs to get parts of other processors' trees in order to compute the remote members of the cell-cell interactions required for the AFMA. The purpose of this step is similar to the Locally Essential Tree (LET) construction step in [87], however, we avoid the overheads associated with their implementation by two optimizations. First, we assemble only the structure of the LET and the coordinates for each cell; the particles and multipoles for each cell are not transferred because they are not required for finding the list of interactions for each cell. Second, instead of fine-grained receiver initiated communication for expanding the tree one level at a time [87], each processor sends to its neighbors the exact part of its own local tree that they will require, resulting in just one message per neighbor (details of this step are described in [75]). Thus we use large grained, sender initiated communication to reduce data transfer overheads.

Once each processor has received parts of the LET from other processors (no exchange of the LET is needed with well-separated processors), and attached them to its copy of the shared top levels of the tree, it has the entire LET, and can start computing members of the interaction lists for each of its cells. Each cell C has four types of interaction lists (details are described in [75]) :

- U list : this contains cells which are adjacent to C or are "close enough" that the interaction needs to be computed between every pair of their particles.

- V list : this contains cells which are sufficiently "well separated" from C such that their multipole expansion can be converted to a "local expansion" at the center of C.

- W list : this contains cells which are smaller than C, and neither sufficiently well separated nor close enough. Their multipole expansions need to be directly evaluated at C's particles.

- V list : this contains cells which are larger than C, and neither sufficiently well separated nor close enough. Their particle fields need to be added to the local expansion at the center of C.

The computation of interactions essentially involves an upward traversal in the AFMA tree during which multipole expansions are evaluated for successively larger cells, followed by a downward traversal during which local expansions are computed for successively smaller cells. Finally the pair-wise U list interactions are computed.

5.3.3 Balancing pair-wise interactions

Pair-wise (U list) interactions take up a major portion of the time for computing interactions between cells. To eliminate redundant pair-wise interactions, Newton's third law is usually used. For a pair of cells A and B, particle data is sent from A to B, where the forces are computed and sent back to A. Although this insight for removing redundant computation is well known, in the parallel context we are faced with the problem of deciding, for every U list computation, which processor it should be computed on. If U list computations are not assigned to processors carefully, it is possible that some processors will get overloaded. We solve this load balancing problem by a combination of static and dynamic methods. The static load balancing method assigns each U list computation to a processor before the computation phase begins. The dynamic load balancing smooths any variations in load because of inaccuracies in the static method.

We can arrive at a heuristic solution to the static load balancing problem by formulating it as an *edge-partitioning problem* on a graph. The vertices of the graph are processors. An edge is drawn between a pair of processors P and Q if there is a U list interaction between a cell on P and a cell on Q. The *weight* of this edge is the total computation cost of all cross-processor U list interactions between cells on P and cells on Q. The problem now reduces to that of partitioning the load of each edge among the processors it connects such that all processors have approximately equal loads after all edges have been assigned. Although this edge-partitioning problem can be solved optimally by mapping it to a flow problem on the graph, or by linear programming, these techniques can often take a significant amount of time because their complexity is greater than $O(V * E)$, for V vertices and E edges. Moreover, it is difficult to parallelize them efficiently. We require a fast, easily parallelizable algorithm.

We have developed a heuristic algorithm by making use of *spatial information* which is already present in the AFMA tree. This spatial information is in the form of the repeated bisections performed while constructing the shared levels of the tree. The main idea in our edge-partitioning algorithm is hence to recursively divide the edge loads between sibling cells (which represent adjacent partitions) in the hierarchical tree. The core of the algorithm is described in pseudocode in Figure 5.3. The heuristic for dividing cross-partition edges between two partitions takes into account local as well as global criteria. The global requirement is that the loads of the two partitions should be balanced. The local requirement is that the edge should not be assigned to a processor which is already heavily loaded (close to the average load per processor). We have obtained excellent results by using the local criterion when processor loads are close to the average load (as would be the case towards the end of this algorithm), and using the global criterion otherwise.

To run this algorithm in parallel, we replicate the graph on all processors. This is not expensive because the number of vertices in the graph is just the number of processors, and edges are mostly between spatially adjacent processors. Each processor recursively bisects *only the subtrees (partitions) it belongs to.* Thus the number of edges that have to be examined by a processor effectively halves as the algorithm traverses each level down the tree. If E is the total number of edges and V is the number of processors, then in the best case each processor has to examine $E + E/2 + E/4 + ...$ edges, which means that the algorithm has a best case complexity of $O(E)$ and a worst case complexity of $O(E * logV)$. In practice the running time is close to the best case because the initial ORB partitioning of particles ensures that the number of edges across partitions is balanced to some extent. Together with the ORB partitioning, this edge-partitioning heuristic results in good load balance for pair-wise computations.

```
edge_partition(cell *c)

{

/* divide edges within the partition corresponding to cell c among c's 2 children */


        child1_load = load due to edges between processors within child1

        child2_load = load due to edges between processors within child2

        cross_edges = list of edges between processors in child1 and processors in child2


        for ( each edge in cross_edges ) {

                if ( either of the processors connected by the edge is close

                                                    to the average load ) {

                        assign the edge to the least loaded processor

                        remove the edge from cross_edges

                }

        }


        for ( each edge in cross_edges ) {

                assign the edge to the lesser loaded of child1 and child2

                increment child1_load or child2_load correspondingly

        }


        if ( my processor is in child1 )

                edge_partition(child1)

        else

                edge_partition(child2)

}
```

Figure 5.3: Edge partitioning algorithm used to eliminate redundant U list computations.

The dynamic load balancing heuristic balances load while the computation is in progress. By default, pair-wise computations are performed by the processor to which they were assigned by the static load balancing heuristic. As soon as a processor finds the resulting forces, it sends the results over to the partner processor. When a processor has finished all its work and becomes idle, it starts performing those cross-processor U list computations which were assigned to other processors and whose results it has not received. It then sends the resulting forces to the partner processor, in the hope that the partner processor has not done the computation already : if so, the partner's work is decreased, and if not, the partner simply discards the forces. The extra performance benefits of dynamic load balancing are presented in Section 5.3.6.

5.3.4 Overlapping communication latency

Since we have implemented our algorithm using Charm++, cells in the AFMA tree are modeled as objects having functions for computing the different interactions. Thus the design consists of concurrently executing objects which interact by asynchronous function invocations. This is a very natural and elegant model which makes it very easy to overlap communication and computation and tolerate communication latency with little extra programming effort.

As discussed in subsection 5.3.2, the partitioning by ORB can be optimized by performing bisections concurrently. This allows idle times during one bisection to be overlapped with computation from the other. Again, during the construction of the Locally Essential Tree, it is not necessary for a processor to wait for the parts of the tree from other processors ; it can continue with the construction of its own local tree. When a message containing tree-data from another processor arrives, the code for processing it is scheduled and that part of the tree is attached to the local tree.

Each of the four types of interactions (corresponding to the four lists) involve highly parallel and irregular interactions between cells in the AFMA tree. When the interacting cells are on different processors, U-list interactions require the transfer of particle data, while V-, W- and

X- list interactions require transfer of multipole expansions or local expansions. For each of the interactions, all processors iterate over their cells, computing local interactions and sending/receiving messages for remote ones. Interactions with remote processors are computed completely asynchronously: when a message containing remote data arrives, the object to which it is directed is automatically scheduled by the Charm runtime system, and the proper function for computing the interaction is invoked. Thus no time is wasted in waiting for remote data, and an almost ideal overlap of communication and computation is achieved.

Although there is much parallelism within each stage of the AFMA, executing them sequentially (as has been done in almost all previous implementations) can lead to serious imbalances and processor idling. This is because it is difficult to balance the load in each stage by itself. In [88] an attempt has been made to explicitly overlap two stages (in the context of the Barnes-Hut method) using a "non-synchronizing" global communication protocol. However, this requires some programming effort, and is difficult to achieve when there are many stages with complex dependences.

Figure 5.4 shows the dependences across the various stages of the AFMA. It can be seen that there is a lot of scope for overlapping the idle times in one stage with computation from other stages. This is achieved naturally, and with *no extra programming effort*, because of the asynchronous, message driven nature of our design. Each processor just starts off all the interactions, and then dependences between stages are enforced *at the granularity of the cell objects*, thus avoiding any need for global synchronization between stages. This overlap across stages enables us to prevent processors from idling when there is work in other stages left. Moreover, because processors do not wait for remote data, locality of data is less important; we have found that the ORB scheme used for partitioning provides sufficient locality.

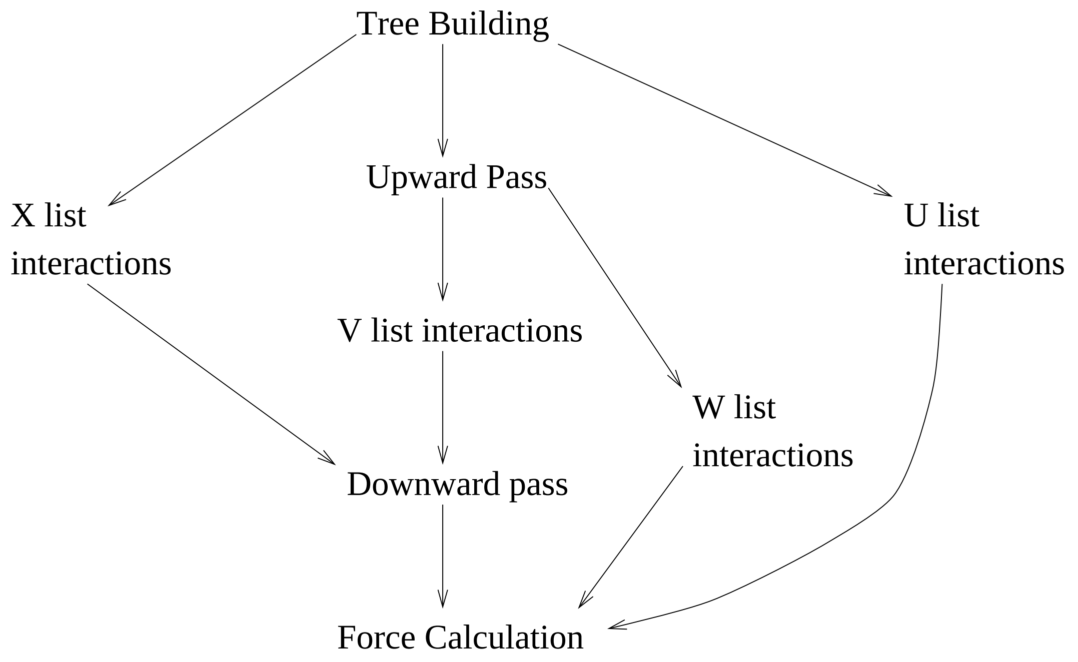


Figure 5.4: Dependences between stages of the AFMA.

5.3.5 Communication optimizations

The communication patterns in the parallel AFMA are unpredictable and irregular. This is especially a problem for distributed memory computers which have high message latencies. Thus it is not possible to get good performance with fine-grained, receiver initiated communication, as is possible for shared memory machines [89]. We have extensively used a sender-initiated *advance-send* protocol to reduce communication overhead in our implementation. In [88, 90] a form of advance-send is used in the context of the Barnes-Hut method by sending particle data to remote nodes instead of requesting for their particles. For the AFMA, the basic problem to be solved before using advance-send is for each processor to determine which other processors will be consumers of its particle and multipole data. The key insight is to use, for each interaction list, the *dual interaction list*. From the definition of the four lists, it is known that the U list is its own dual (i.e. if cell A is in cell B's U list, then cell B has to be in cell A's U list), the

V list is its own dual, and the W and X lists are duals of each other (i.e. if cell A is in cell B's W list, then cell B has to be in cell A's X list). Thus each cell simply has to advance-send its particles to all cells in its U list, its multipole expansions to all V and X list cells, and a converted local expansion to all W list cells. In the AFMA, sending multipole expansions instead of particles has the advantage that a single multipole expansion is usually much less voluminous than the particle data itself, for cells containing tens or hundreds of particles. As described earlier, when a message carrying this data arrives at a processor, the Charm++ run-time automatically schedules the correct interactions in the appropriate cell objects. Thus advance-send is achieved with very little programming effort.

We have developed a library class for doing advance-sends which also prevents duplicate data from being sent to the same destination processor. Programmers simply need to inherit the message classes in their Charm++ program from this advance-send class to use the advance-send protocol.

5.3.6 Performance results

We have implemented our parallel AFMA algorithm using Charm++. The implementation can thus run on virtually any MIMD parallel computer. This subsection presents performance results for our implementation.

Tables 5.12 and 5.13 present results on the IBM SP and TMC CM-5 for our 3-D AFMA implementation for a uniform random distribution and a nonuniform distribution[1] of 20,000 particles each, for one time-step. The number of multipole expansion terms is 8, corresponding to the high-accuracy simulations in the work by Board [83]. The results in Tables 5.12 and 5.13 do not include the parallel tree construction step.

[1]Each coordinate value in this distribution was generated by doing a bitwise AND of two random integers, and then normalizing it within the computational box. Thus most particles are concentrated at one of the corners of the box: 42% of the particles are in 12% of the space.

Processors	1	2	4	8	16	32	64
Uniform	168.4	100.7	70.14	36.39	22.31	12.86	6.37
Non-uniform	261.9	193.6	100.3	52.08	26.66	15.24	8.25

Table 5.12: Time (seconds) to simulate one time-step for 20,000 particles on the IBM SP.

Processors	4	8	16	32	64	128
Uniform	229.4	116.0	64.55	34.24	18.45	10.90
Non-uniform	384.3	163.8	85.50	42.24	21.09	11.04

Table 5.13: Time (seconds) to simulate one time-step for 20,000 particles on the CM-5.

The times taken for tree construction are presented in Table 5.14, for the same distribution of particles on the IBM SP. As can be seen, the tree construction does not scale very well with increasing numbers of processors, in spite of a highly optimized tree construction code. This is an inherent limitation because of the communication latencies on the parallel machine. However, tree construction is only about 7.2 % of the total execution time even in the worst case, hence this limitation is less significant. Moreover, several computation steps can be carried out before a new tree formation is required due to load imbalances; a new tree-partitioning every time-step is not required for the correctness of the algorithm, as long as particles are moved to their new cells after a computation step.

Figure 5.5 graphically shows the speedups for the non-uniform distribution of 20,000 particles on the IBM SP as a function of the number of processors.

Processors	1	2	4	8	16	32	64
Tree construction	2.36	2.10	1.41	0.818	0.497	0.427	0.596

Table 5.14: Time (seconds) to construct the parallel AFMA tree for the non-uniform distribution of 20,000 particles on the IBM SP.

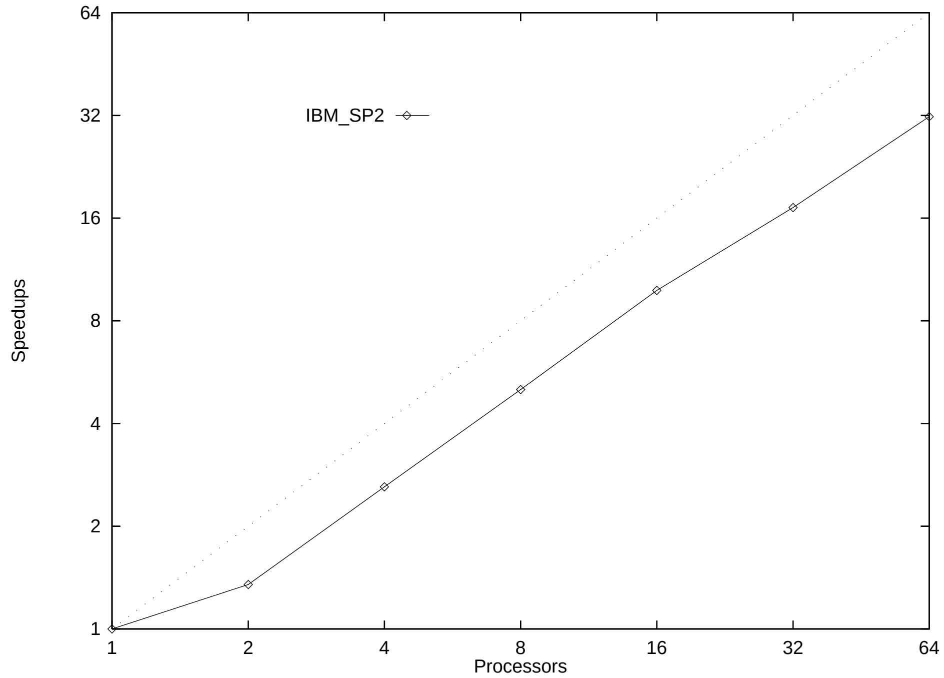


Figure 5.5: Speedups for parallel AFMA as a function of number of processors, on the IBM SP.

5.3.7 Analysis of performance

We now look deeper into the performance of the AFMA code in order to understand the benefits of the various optimizations.

Figure 5.6 is a picture of the percentage utilization over time (averaged over all processors) for the nonuniform distribution of 10,000 particles on 32 processors of the CM-5. The picture

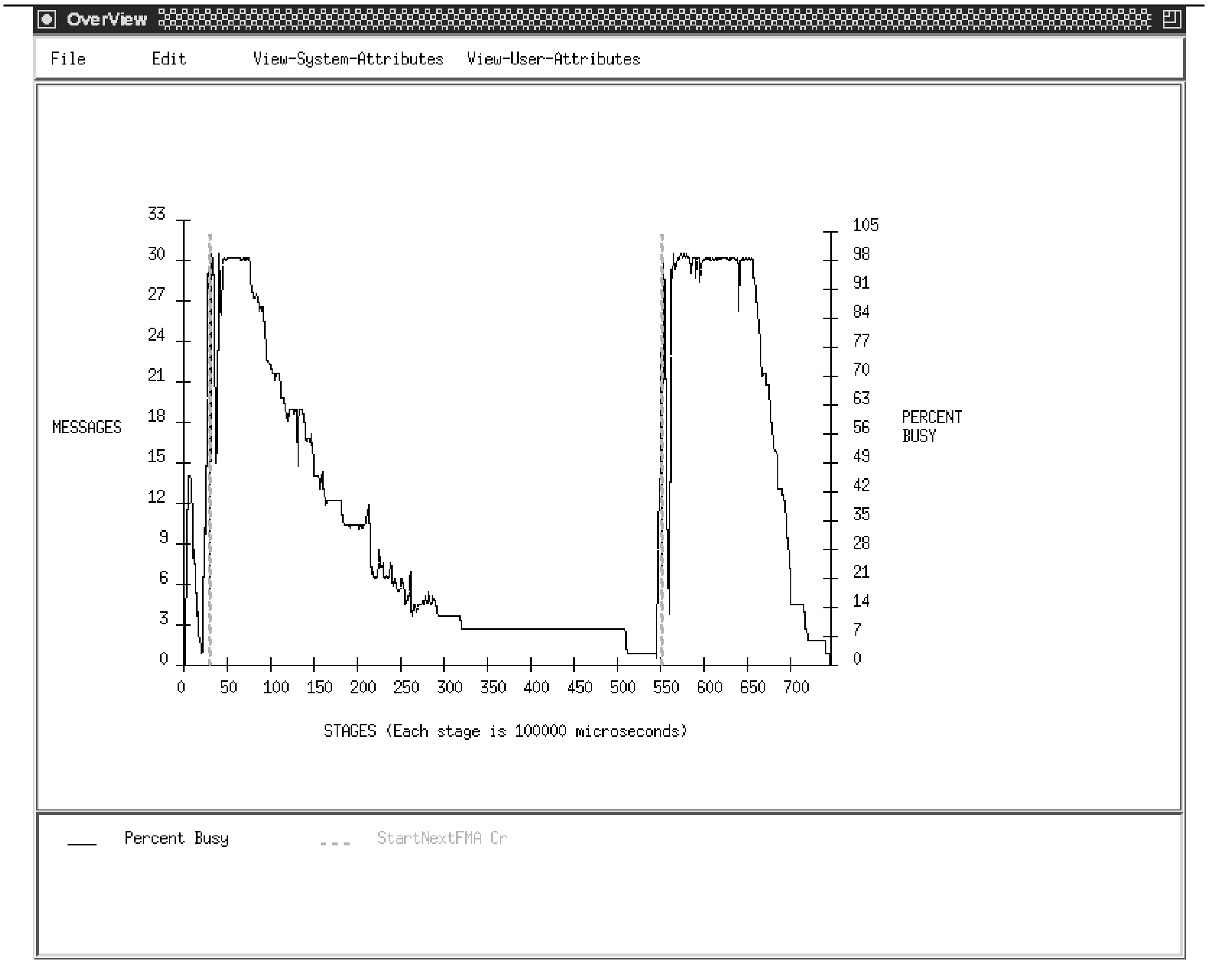


Figure 5.6: Percentage utilization over time for two time-steps with 10,000 particles on 32 processors of the CM-5.

was obtained using the Projections performance analysis tool [28]. The picture shows two iterations of the AFMA (the vertical dashed lines separate the iterations).

In the first iteration (stage 30 through stage 550) the load partitioning (ORB) algorithm assumes that all particles have the same load, so that all processors are assigned equal numbers of particles. From the picture we observe that the first iteration took considerably longer than the second. The reason is obvious : in the first iteration the processor utilization was very bad for most of the time. Moreover, the utilization decreases only gradually and remains around 10 % after stage 320, suggesting that three or four processors were very highly overloaded.

In the second iteration the ORB partitioning algorithm makes use of particle load measures from the first iteration. Hence it can now make an accurate estimate of loads and assign processors particles appropriately. From the picture, we see that the second iteration is much shorter (stage 550 through the end), and has very good utilization : the curve drops sharply at the end of the program execution showing that all processors finished their work at nearly the same time, hence they had equal loads. Thus we have demonstrated the significant benefits that can be obtained from adaptive runtime optimizations.

The next few subsections quantitatively analyze the benefits due to each of the optimizations designed for the AFMA. All these results are for 10,000 particles of the non-uniform distribution, on 32 processors of the TMC CM-5.

5.3.7.1 Adaptive load partitioning

Iteration	First	Second
Time (sec)	50.56	18.27

Table 5.15: Time (seconds) taken for the first and second iterations, showing the improvements due to adaptive load partitioning.

We first analyze the benefits of the adaptive Orthogonal Recursive Bisection component. Table 5.15 shows the time taken for the first iteration (with all particles having equal loads by default), and the second iteration (with particle loads from the first iteration guiding the adaptive repartitioning). Note that these results are *inspite* of the best efforts of the edge-load partitioning component: i.e. the first iteration has bad load balance even after underloaded processors are assigned all their edge loads. This shows that the initial particle distribution caused a highly non-uniform load distribution.

5.3.7.2 Overlapped ORB

Table 5.16 shows the time taken for the Orthogonal Recursive Bisection component (Section 5.3.2), for the non-overlapped case (all bisections occur one after the other) and the overlapped case (independent bisections are overlapped as much as possible).

Strategy	Non-overlapped ORB	Overlapped ORB
First iteration	3.15	2.13
Second iteration	1.32	0.324

Table 5.16: Time (seconds) taken for orthogonal recursive bisection for the first and second iterations, showing the advantages of overlap.

We can draw two conclusions from table 5.16. First, overlapping the independent bisections of the ORB component gives significant improvements. Second, the improvement is even greater in the second iteration. The second conclusion can be explained by observing that particles have already been spatially distributed at the beginning of the second ORB : hence fewer processors need to participate in bisecting each region of space. The non-overlapped ORB forces all processors to wait for bisecting regions in which only a few processors have particles.

5.3.7.3 Edge-load balancing

Table 5.17 presents the time taken for the AFMA with and without the edge-load balancing (Section 5.3.3) for carefully assigning pairwise load to processors. In the non-balanced case, edges are assigned to processors using a naive scheme which divides the edges between two processors equally between them.

Strategy	Naive edge-assignment	Balanced edge-loads
Time (sec)	23.51	18.27

Table 5.17: Time taken for AFMA with and without edge-load balancing.

5.3.7.4 Communication optimizations

Table 5.18 shows the performance of the AFMA with and without the advance-message and message-reuse optimizations. The benefits for this do not seem to be significant, which can be explained from the organization of the advance-sends : all of them are done together at the beginning of the computation, which may lead to network congestion. We expect to achieve greater benefits from advance-sends and message reuse by spreading the communication over the course of the computation.

Strategy	Request-reply	Advance-send & reuse
Time (sec)	19.04	18.27

Table 5.18: Time taken for AFMA with and without communication optimizations.

5.3.7.5 Overlapping across stages

As described in Section 5.3.4, the object-oriented data-driven programming model provided by Charm++ allows the different list computations in the AFMA algorithm to be overlapped, by allowing objects from different computations to interleave their execution, and enforcing dependences at the individual object level instead of requiring global synchronizations. Table 5.19 shows the time taken for one iteration with and without this overlap.

Strategy	Non-overlapped AFMA	Overlapped AFMA
Time (sec)	24.72	18.27

Table 5.19: Time taken for overlapped and non-overlapped AFMA stages.

Chapter 6

Conclusion

In this thesis we have explored issues in languages, optimizations and automated tools for parallel object oriented programming.

We have discussed object-orientation as a means of dealing with the inherent complexity of parallel software development. We leveraged the abstraction and code-reuse facilities provided by the widely used language C++, by developing a parallel extension called Charm++. Charm++ provides support for dynamically creatable parallel objects, asynchronous method invocation, message-driven execution, parallel object arrays with arbitrary mapping and asynchronous migration, dynamic load balancing, and several other useful features. Charm++ is a working system, running on a wide range of parallel machines, and has been used for writing several benchmark programs and applications.

We have also explored the use of automated tools for incorporating optimizations in parallel programs without user intervention. We have concentrated on run-time optimizations, which are especially useful for irregular or dynamic programs in the parallel object-oriented domain. We have developed several run-time optimizations, and analyzed their mechanisms, policies and the information required for carrying them out. We then presented the Paradise automatic post-mortem optimizer which generates optimization hints. Working in close cooperation with the

Charm++ runtime libraries, Paradise is able to automatically carry out several optimizations. We discussed the internal program representation built by Paradise. The choice of optimization strategy is guided by classifying programs based on their characteristics such as phase structure, data locality, and object creation patterns. Paradise optimizes static object placement for programs without phases and with phases. It also optimizes dynamic object placement by choosing a suitable load balancing strategy. Automated optimizations for scheduling, grainsize control and communication reduction were also presented.

The effectiveness of Charm++ and Paradise are demonstrated on several applications : the Fast Multipole Algorithm for N-body problems, the NAS Scalar Pentadiagonal program, and several other benchmark programs with a wide variety of opportunities for optimization.

6.1 Future work

Our work on the Charm++ parallel object-oriented language has demonstrated that parallelism can be profitably incorporated in a practical, widely used sequential language. In the future, we may introduce a few other features such as multi-threaded objects, parameter marshaling for methods and event synchronization or selective method acceptance, if they are found to be useful. Since the development of Charm++ in 1992-93, parallel object-oriented languages have become very popular in the parallel computing community. Today there are more than fifteen different dialects of parallel C++ [17], with a wide variety of programming models and approaches to combining object-orientation and parallelism. Thus it is hoped that the field of parallel object-oriented programming is maturing, as experience with various features grows. In fact, there is currently a standardization effort under way towards developing a standard parallel C++ called *HPC++* which incorporates many of the features available today in parallel C++ dialects.

The field of automated performance optimization tools is only beginning to be explored. The ideal of a completely automatic optimization system (which works as well as sequential optimizers in compilers for sequential programs) is still some distance away. The set of parallel programming models, application areas, performance problems and optimization techniques is so large that significant research is needed in all those areas before we can have a truly general purpose tool that automatically optimizes most performance problems for most applications. In particular, the following specific areas need to be researched :

- Integration with compiler techniques : Paradise currently requires some support from the compiler/programmer for incorporating some runtime optimizations like pipelining. It would be useful to automatically perform compiler as well as runtime optimizations. While we have concentrated on runtime optimizations in this work, several researchers have explored compiler optimizations for different programming models. A truly automated optimization system will use information from static analysis, run-time measurement and post-mortem analysis to automatically choose and parameterize compile time transformations as well as runtime optimization libraries.

- Language/Tool design driven by optimizations : One of the problems we have faced during this work is the difficulty of obtaining information about the characteristics of the parallel program that are required for optimization. Hence it is important to design parallel languages that make it easy for the programmer to provide this information, and compilers and tools that can provide this information automatically.

- Optimizing Speedups : In this work we perform post-mortem analysis on a single trace of program execution. The analysis may hence not be valid when the program characteristics radically change, as will be the case when the processor set or input size is increased by orders of magnitude. To solve this problem, Paradise needs to use traces from several runs of the program on different processor sets, and model speedups either by deriving expressions for predicting the execution time of the program, or using trace-driven simulation.

- Integration with other tools : Paradise is one of several tools developed in the author's research group including: Visual Dagger, a GUI-based editor for depicting macro-dataflow and dependences within and across objects; Projections [28], a performance visualization and expert analysis tool; and a trace-driven simulator [91]. Integrating these tools into a full program development environment will greatly simplify Charm and Charm++ program development.

- More optimizations and heuristics : An expert system is only as good as the amount of expert knowledge given to it. This is true of Paradise too : its capabilities can be greatly enhanced by adding new optimization techniques and better heuristics. As mentioned above, the list of possible performance optimizations is large and ever increasing. Some of the runtime optimizations that may be automatically incorporated include object placement for irregular graphs (this would involve choosing and finding weights for graph partitioning libraries); automatic remapping (migration) for parallel object arrays when there is a conflict between the communication patterns in different phases of a program; detection and optimization of communication operations such as all-to-all messaging.

- Evaluation : While we have found Paradise to be very useful for optimizing the set of application programs described in this thesis, more experience is needed with a broad range of parallel programs and optimization techniques, in order to refine the heuristics incorporated in Paradise, and to motivate new optimizations.

Bibliography

[1] 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.

[2] L. V. Kale and Sanjeev Krishnan. Charm++. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*, chapter 5. MIT Press, 1996.

[3] L.V. Kalé. The Chare Kernel Parallel Programming Language and System. In *Proceedings of the 19th International Conference on Parallel Processing*, pages II–17–II–25, August 1990.

[4] V.S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

[5] 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.

[6] D.A. Padua and M.J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):829–842, dec 1986.

[7] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, May 1994.

[8] W. Fenton, B. Ramkumar, V. Saletore, A.B. Sinha, and L.V. Kalé. Supporting Machine-Independent Parallel Programming on Diverse Architectures. In *Proceedings of the 1991 International Conference on Parallel Processing*, 1991.

[9] L.V. Kalé. A Tutorial Introduction to CHARM, December 1992.

[10] L.V. Kalé, B. Ramkumar, A. Sinha, and A. Gursoy. The Charm Parallel Programming Language and System: Part I – Description of Language Features. Technical Report 95–2, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Urbana-Champaign, 1995.

[11] B. Ramkumar, A. Sinha, V. Saletore, and L.V. Kalé. The Charm Parallel Programming Language and System: Part II – The Runtime System. Technical Report 95–3, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Urbana-Champaign, 1995.

[12] L.V. Kale and Sanjeev Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications*, September 1993. (Also: Technical Report UIUCDCS-R-93-1796, March 1993, University of Illinois, Urbana, IL).

[13] L.V. Kalé, M. Bhandarkar, N. Jagathesan, and S. Krishnan. Converse: An Interoperable Framework for Parallel Programming. Technical Report 95–12, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Urbana-Champaign, March 1995.

[14] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[15] A. Gursoy. *Simplified Expression of Message-Driven Programs and Quantification of Their Impact on Performance.* PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, June 1994.

[16] A. Sinha and L.V. Kalé. A Load Balancing Strategy for Prioritized Execution of Tasks. In *Proceedings of the 7th International Parallel Processing Symposium*, pages 230–237, April 1993.

[17] Gregory V. Wilson and Paul Lu, editors. *Parallel Programming Using C++.* MIT Press, 1996.

[18] A.S. Grimshaw. Easy to Use Object-Oriented Parallel Programming with Mentat. *IEEE Computer*, pages 39–51, May 1993.

[19] P.A. Buhr, G. Ditchfield, R.A. Stroobosscher, B.M. Younger, and C.R. Zarnke. uC++: Concurrency in the Object-Oriented Language C++. *Software Practice and Experience*, 22(2):137–172, 1992.

[20] A. Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System.* MIT Press, 1990.

[21] R. Chandra, A. Gupta, and J.L. Hennessy. COOL: An Object-Based Language for Parallel Programming. *IEEE Computer*, 27(8):14–26, August 1994.

[22] W. Athas and N. Boden. Cantor: An Actor Programming System for Scientific Computing. In *Proceedings of the ACM SIGPLAN Workshop on Object Based Concurrent Programming, ACM SIGPLAN Notices*, pages 66–68, April 1989.

[23] C. Houck and G. Agha. HAL: A High Level Actor Language and its Distributed Implementation. In *Proceedings of the International Conference on Parallel Processing*, August 1992.

[24] A.A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs.* MIT Press, 1993.

[25] Sanjeev Krishnan and L. V. Kale. A parallel array abstraction for data-driven objects. In *Proceedings of the Parallel Object-Oriented Methods and Applications Conference*, February 1996.

[26] High Performance Fortran Forum. *High Performance Fortran Language Specification (Draft)*, 1.0 edition, January 1993.

[27] S. R. Kohn and S. B. Baden. Irregular coarse-grain data parallelism under lparx. *Scientific Programming*, 5(3), 1996. To appear.

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

[29] 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.

[30] A. Gursoy and L. V. Kale. Dagger: Combining the benefits of synchronous and asynchronous communication styles. Technical Report 93-3, Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana-Champaign, March 1993.

[31] 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.

[32] 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.

[33] 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.

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

[35] R. Bruce Irvin and Barton P. Miller. Mapping Performance Data for High-Level and Data Views of Parallel Program Performance. In *Proceedings of the ACM International Conference on Supercomputing*, May 1996.

[36] J. C. Yan, S. Sarukkai, and P. Mehra. Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit. *Software Practice and Experience*, April 1995.

[37] 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.

[38] M. Crovella and T. J. LeBlanc. The Search for Lost Cycles: A New Approach to Performance Tuning of Parallel Programs. In *Proceedings of Supercomputing 1994*, November 1994.

[39] Thomas Fahringer. Estimating and Optimizing Performance for Parallel Programs. *IEEE Computer*, 28(11), November 1995.

[40] M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, September 1991.

[41] V. Herrarte and E. W. Lusk. Studying parallel program behavior with upshot. User Manual for Upshot.

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

[43] 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.

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

[45] 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.

[46] 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.

[47] L. Rauchwerger, N. Amato, and D. Padua. Run-time methods for parallelizing partially parallel loops. In *Proceedings of the 9th ACM International Conference on Supercomputing*, July 1995.

[48] J. Wu, J. Saltz, S. Hiranandani, and J. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the International Conference on Parallel Processing*, August 1991.

[49] D. K. Chen, J. Torrellas, and P. C. Yew. An efficient algorithm for the run-time parallelization of doacross loops. In *Proceedings of Supercomputing 1994*, November 1994.

[50] A. Lain and P. Banerjee. Exploiting spatial regularity in irregular iterative applications. In *Proceedings of the International Parallel Processing Symposium*, April 1995.

[51] W. Fenton, B. Ramkumar, V.A. Saletore, A.B. Sinha, and L.V. Kale. Supporting machine independent programming on diverse parallel architectures. In *Proceedings of the International Conference on Parallel Processing*, August 1991.

[52] R. Chandra, A. Gupta, and J. Hennessy. Integrating concurrency and data abstraction in the COOL parallel programming language. *IEEE Computer*, February 1994.

[53] A. Chien, V. Karamcheti, and J. Plevyak. The concert system: Compiler and runtime support for fine-grained concurrent object-oriented languages. Technical Report R-93-1815, Department of Computer Science, University of Illinois, Urbana-Champaign, June 1993.

[54] C. Houck and G. Agha. Hal: A high level actor language and its distributed implementation. In *Proceedings of the International Conference on Parallel Processing*, August 1992.

[55] W. Kim and G. Agha. Efficient support of location transparency in concurrent object-oriented programming languages. In *Proceedings of Supercomputing 1995*, December 1995.

[56] 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.

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

[58] M. Gupta and P. Banerjee. Paradigm : A compiler for automatic data distribution on multicomputers. In *ACM International Conference on Supercomputing*, July 1993.

[59] B. Chapman, T. Fahringer, and H. Zima. Automatic support for data distribution on distributed memory multiprocessor systems. In *6th Conference on Languages and Compilers for Parallel Computing*, August 1993.

[60] U. Kremer and K. Kennedy. Automatic data layout for high performance fortran. In *Proceedings of Supercomputing 1995*, December 1995.

[61] 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.

[62] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of Supercomputing 95*, December 1995.

[63] B. Hendrickson and R. Leland. The Chaco user's guide. Technical Report SAND 93-2339, Sandia National Laboratories, Albuquerque, NM, October 1993.

[64] Chao-Wei Ou and Sanjay Ranka. Parallel incremental graph partitioning using linear programming. In *Proceedings of the Supercomputing 94*, November 1994.

[65] 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.

[66] L.V. Kale and Sanjeev Krishnan. Medium grained execution in concurrent object-oriented systems. In *Workshop on Efficient Implementation of Concurrent Object Oriented Languages, at OOPSLA 1993*, September 1993.

[67] M. Wu and W. Shu. A dynamic program partitioning strategy on distributed memory systems. In *Proceedings of the International Conference on Parallel Processing*, August 1990.

[68] E. Mohr, D. Kranz, and R. Halstead. Lazy task creation : A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, July 1991.

[69] S. Hiranandani, Ken Kennedy, and C-W Tseng. Compiler Optimizations for Fortran-D on MIMD Distributed memory machines. In *Proceedings of Supercomputing 1991*, November 1991.

[70] L. V. Kale and Sanjeev Krishnan. A comparison based parallel sorting algorithm. In *Proceedings of the International Conference on Parallel Processing*, August 1993.

[71] D.H. Bailey et al. The NAS Parallel Benchmarks. *Intl. Journal of Supercomputer Applications*, 5(3), 1996.

[72] D.H. Bailey et al. The NAS Parallel Benchmarks. Technical Report NASA Technical Memorandum 103863, NASA Ames Research Center, July 1993.

[73] R. Van der Wijngaart. Efficient implementation of a 3-dimensional adi method on the ipsc/860. In *Proceedings of Supercomputing 1993*, November 1993.

[74] J. Bruno and P. Capello. Implementing the Beam and Warming method on the hypercube. In *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications*, January 1988.

[75] Sanjeev Krishnan and L. V. Kale. A parallel adaptive fast multipole algorithm for n-body problems. In *Proceedings of the International Conference on Parallel Processing*, August 1995.

[76] L. Greengard and V. I. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73, 1987.

[77] A. W. Appel. An efficient program for many-body simulation. *SIAM Journal of Scientific and Statistical Computing*, 6, January 1985.

[78] J. E. Barnes and P. Hut. A hierarchical O(NlogN) force calculation algorithm. *Nature*, 324, 1986.

[79] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. McGraw Hill International, 1981.

[80] H. Heller, H. GrubMuller, and K. Schulten. Molecular dynamics simulation on a parallel computer. *Molecular Simulation*, 5, 1990.

[81] Leslie Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems*. MIT Press, 1988.

[82] J. Carrier, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm for particle simulations. *SIAM Journal of Scientific and Statistical Computing*, 9, July 1988.

[83] John A. Board et al. Scalable implementations of multipole accelerated algorithms for molecular dynamics. In *Proceedings of the Scalable High Performance Computing Conference*, May 1994.

[84] L. Greengard and W. D. Gropp. A parallel version of the fast multipole method. *Computers Math Applications*, 20(7), 1990.

[85] J. Singh, C. Holt, J. Hennessy, and A. Gupta. A parallel adaptive fast multipole method. In *Proceedings of Supercomputing 93*, November 1993.

[86] M. S. Warren and J. K. Salmon. Astrophysical n-body simulations using hierarchical tree data structures. In *Proceedings of Supercomputing 92*, November 1992.

[87] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of Supercomputing 93*, November 1993.

[88] A. Grama, V. Kumar, and A. Sameh. Scalable parallel formulations of the Barnes-Hut method for n-body simulations. In *Proceedings of Supercomputing 94*, November 1994.

[89] J. P. Singh, J. L. Hennessy, and A. Gupta. Implications of hierarchical n-body methods for multiprocessor arc hitecture. *ACM Transactions on Computer Systems*, May 1995.

[90] P. Liu and S. Bhatt. Experiences with parallel n-body simulation. In *6th Annual ACM Symposium on Parallel Algorithms and Architectu res*, 1994.

[91] Attila Gursoy and L. V. Kale. Simulating message-driven programs. In *Proceedings of the International Conference on Parallel Processing*, August 1996. To Appear.

Vita

Sanjeev Krishnan was born in Akola, Maharashtra, India. He obtained his B. Tech. in Computer Science and Engineering from the Indian Institute of Technology, Bombay, India, in July 1991. From fall 1991 through spring 1996 he was a research assistant at the Department of Computer Science, University of Illinois, Urbana-Champaign. After completing his Ph.D. he will be working at Sun Microsystems Inc in Menlo Park, California, as a Member of Technical Staff.