PERFORMANCE ANALYSIS OF OBJECT-BASED AND MESSAGE-DRIVEN PROGRAMS

BY

AMITABH BHUVANGYAN SINHA

B.Tech., Indian Institute of Technology, Kanpur, 1988
M.S., Ohio State University, 1989

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, 1994

Urbana, Illinois

The significant gap between peak and realized performance of parallel machines motivates the need for performance analysis. Most existing performance analysis tools provide generic measurement and displays. It is the responsibility of the users to analyze the performance of their programs using the displayed information. This is a non-trivial task, because not only does one need to identify the information that is needed for such analysis, sometimes that information may not even be displayed by the tool. The task of analysis is even more difficult for massively parallel machines, where voluminous amounts of information can be generated. Therefore, a good performance analysis tool should be able to provide intelligent analysis about the performance of a parallel program. Such automatic performance analysis is feasible for programming paradigms that provide the system sufficient information about the behavior of its programs. We have built a framework for automatic analysis for one such paradigm called Charm, a portable, object-based, and message-driven parallel programming language. In this thesis, we describe the process of design and implementation of this framework, and show its utility with sample case studies.

To my family,

Acknowledgements

First, I would like to thank my advisor Sanjay. His unbounded enthusiasm as a guide and a researcher never ceases to amaze me. I have learnt much from him. I would also like to thank the members of my committee, Professors Skeel, Vaidya, and Reed for their helpful comments.

When I joined the Parallel Programming Laboratory in December 1989, Ram and Vikram were less than a year away from graduation, and initiated me into the world of parallel programming and Charm (or, Chare Kernel as it was known back then). Attila, Sanjeev, Ed, and Thorr provided lively company through the middle years of my Ph.D. when I was searching for a thesis topic. Josh, Milind, Narain, Robs (Brunner, Neely, and Zeh), and Terry were part of the laboratory in my final year. Their constant input has been responsible for improvements in the user-friendliness and correctness of Charm and Projections.

I lost Jayesh, a friend with whom I shared many 'laughing sessions' and many discussions on the meaning of life, and Ph.D. Dinesh, Subrata, Rashmi, Vijay, Sagarika, Ashish, Sarkar, Satyam, Samir, and Vineeta helped me retain a balance between school and life through the many years of graduate school. The last year, with the birth of our son Kairav, has been particularly busy for Jaya and I. Shanthi helped us immensely during this period by taking care of Kairav when the combined demands of our graduate school lives became overwhelming.

My parents, through all the years of my childhood, gave me two very important things: the freedom to do what I wanted, and the support when I needed it. I thank them for that.

Kairav and I were both formalizing our intuitions at the same time. He, about languages, three all at the same time, and I about event graphs and performance analysis. Perhaps, this last year has been the hardest on my wife, Jaya. She has patiently waited for me to finish my PhD, whilst she, only a few steps away from her own PhD proposal has taken care of the home and Kairav. The next few years are hers!

And, finally, this must be a coincidence: the fortnight before I got my thesis draft done, I suffered through the emergence of my final wisdom tooth. Do I feel any wiser having gone through the tortures of thesis writing and wisdom teeth in the same week? Only time will tell!

Table of Contents

Chapter

List of Tables

List of Figures

Chapter 1

Introduction

Even though there exist parallel machines today with *peak performances* in the range of tens to even hundreds of gigaflops, the actual performance obtained on realistic application programs on such machines varies dramatically, and is often much smaller than the peak performance. It is not uncommon to see variations of two orders of magnitude in performance for the same machine on different application programs. Even when we restrict attention to different implementations of the same algorithm, substantial variations in performance may exist on the same parallel computer. These variations arise due to a variety of factors. Some of the common factors, at least on distributed memory computers, are: presence and extent of sequential bottlenecks, load imbalance across processors, communication costs, I/O costs, and synchronization requirements. These factors are in addition to the usual uni-processor concerns, such as the cache performance of sequential segments of code. In order to improve the performance of a particular parallel program, one must identify the critical factor that is affecting the performance of the program negatively in the most significant way *and* the component of the algorithm that is responsible for this factor. Performance feedback and analysis tools which provide such feedback are therefore crucial to improving the performance of parallel programs.

The focus of this thesis is on techniques for analyzing the efficiency of parallel programs. More specifically, we aim at developing such techniques for message-driven and object-based programming languages. Our approach involves both language specific feedback, which attempts to show the user what happened during a run of their program, and automated performance analysis, which attempts to find potential causes of performance loss and suggest improvements automatically to the user. These concepts are elaborated in the sections below.

1.1 Language specific performance feedback

In order to understand the efficiency issues for a program, one needs to understand the behavior of the program itself. The most widely used technique for understanding program behavior is visualization. In program visualization, a number of attributes of the program's execution, such as processor utilization and network bandwidth, are displayed. The user can use the information conveyed through the displays, in conjunction with the knowledge of the application program, to determine possible performance problems.

A significant number of performance visualization tools, such as Paragraph [1] and Upshot [2], exist for the SPMD (Single Program Multiple Data) model of parallel computation. In the SPMD model, a single program executes on each processor, and programs on different processors communicate with each other using shared memory (lock/unlock) or message passing (send/receive) primitives. Message passing primitives can be loosely synchronous[1], tightly synchronous[2], or asynchronous[3]. For convenience, we refer to all the message passing variations of the single process model as SPMD.

Active research is being conducted on other execution models for parallel programs, such as on the message driven and object based execution models [4, 5, 6]. In a message-driven execution model, a message is addressed to a *method* of an object; the execution of the method is scheduled by the runtime after the message arrives at the destination processor. A message-driven execution model provides many advantages over the SPMD model of computation [7]. It permits the user to obtain a (possibly) greater degree of parallelism and efficiency through:

[1] A loosely synchronized send means that the program executing the send waits until the send is complete. This waiting is referred to as *blocking*. Completing the send, however, does not guarantee that the message has been received. A loosely synchronized receive means that the program executing the receive waits until the message arrives in the specified buffer.

[2] A *send* event in a process is synchronized with the corresponding *receive* event in another process, e.g., in CSP [3] a send blocks till the corresponding receive is executed, and vice versa.

[3] Unlike loosely synchronized sends and receives, asynchronous sends and receives do not block. Rather they return a unique message ID, which can be used to check for completion of the requested action. The ID is not reused until it has been released.

- Adaptive scheduling of the execution of a different method in the same object, when one method is waiting for a message.

- Adaptive scheduling of the execution of a different object, when an object is waiting for a message.

Since the native execution model of most distributed memory parallel machines is SPMD, messages and single processes on processors can be viewed as the lowest level of abstraction. The runtime support for more sophisticated and high-level execution models, such as the message driven execution model, is generally built on top of the SPMD layer on a distributed memory machine. Thus, even though visualizing the performance of program written for a high-level execution model in terms of processes and messages conveys an accurate picture of what happened on the machine, the level of abstraction is too low and not specific enough to the language. For example, existing performance tools for SPMD models do not provide any information on the creation of new objects, or the inter-leaving of the execution of many different objects on the same processor. Such information is critical to execution models which are message-driven and object based. Therefore, the first step towards a better understanding of the performance of high-level programs needs to be the development and identification of techniques to visualize information specific to an object-based and message-driven language, so that the displayed information can then easily be related to the program itself.

1.2 Automatic performance analysis

Program visualization tools often present volumes of visual feedback covering many different facets of program behavior. Performance analysis becomes non-trivial, because one needs to sift through large amounts of visual data to determine performance problems. Performance analysis becomes even more difficult on massively parallel machines, because the amount of information presented is much larger. Therefore it becomes necessary that performance tools provide some form of automatic support for performance analysis. For example, a performance analysis tool should be able to detect that the cause of poor turn-around time is the delay in scheduling of specific tasks on the critical path. Or, the performance tool should be able to detect that two objects (which could be located on the same processor, but are not) are communicating extensively, and suggest that they be mapped to the same processor.

Automatic performance analysis at first seems to be an intractable problem: one expects that there are thousands of possible performance problems and analysis techniques. This is probably true, however the analysis of many real applications has identified the reason for poor performance to be well recognized problems such as imbalance in load [8], the time taken for synchronization [8], or the small granularity with respect to communication latencies [9]. In fact, Fowler et al [10] have suggested a systematic decision tree based approach that a user can use to navigate through performance data and home onto typical performance problems. At intermediate points in the decision tree, one of the many views provided by their performance tool can be used to determine the course of action that needs to be taken next. The approach proposed by Fowler et al and other performance analysis examples suggest that a small core of techniques, which require information about specific behavioral characteristics of the program, can be used to analyze the performance of parallel programs.

Automatic analysis, therefore, seems tractable. How do we go about realizing automatic performance analysis? In building a road map towards automatic analysis, the first step is to identify the set of techniques that will be used for analysis based on experience and expertise attained by tuning many parallel programs. Such techniques embody our knowledge of the dominant and common reasons for performance loss, the set of tricks that can be used to improve performance, and an understanding of the circumstances (the 'symptoms') in which such tricks can be used. The second step is to identify the behavioral characteristics of the program needed to use the identified techniques. Some aspects of program behavior which we have found essential for automatic analysis are a knowledge of the sub-tasks in the program, the modes in which information is shared, the nature of global synchronization, and the mapping of tasks. The third and final step is to acquire information about the characteristics. Once all this can be accomplished, automatic analysis can be performed by automatically acquiring information and applying the analysis techniques.

One of the most difficult steps in the path towards automatic analysis is the acquisition of information about program behavior. How can the performance analysis tool acquire information about a program's behavioral characteristics? The user who writes a parallel program is often keenly aware of the behavioral characteristics of the program. This understanding can be lost in an attempt to code the program using the mechanisms available in the language. This happens because the language may not provide mechanisms which capture exactly the behaviors desired by the user. So, if there existed a parallel programming language which provided users with mechanisms to represent widely occurring program behaviors, the system could then readily acquire such information. There are three broad ways in which a parallel programming performance analysis system can know more about a user's program:

1. *Language constructs*: The easiest way of acquiring information about a program's behavioral characteristics is to provide language constructs, which in themselves embody information about a behavior. For example, the language can provide a construct which allows a user to specify a variable to be *read-only*, and permits only read operations on the variable. Such a construct is a simple way of letting the user convey to the system the information that the variable is read-only.

2. *System libraries*: The system can also acquire information about a program's behavior if known system libraries are used. For example, the use of the *barrier* construct provided by most message passing SPMD languages indicates a global synchronization in the program.

3. *Statis analysis or compiler support*: Compiler support is needed to acquire information not provided by language constructs and system libraries. A number of systems use compiler techniques to understand and optimize user code, both at the sequential and at the parallel programming levels. Such compiler effort often manifests itself in the form of high-level support in parallel programming. For example, in addition to providing common capabilities to the user, such as sending and receiving messages, parallel programming models often provide high-level features, such as automatic decomposition and load balancing. In the work on parallelizing compilers [11, 12, 13], the system attempts to automatically decompose a computation into tasks, map the tasks onto processors, and schedule them. In other models, such as PVM [14, 15, 16], Express [17], and Linda [18, 19] all three tasks, decomposition, mapping, and scheduling, are the user's responsibility. Implicit in the ability of a system to provide high-level programming support is the fact that the system knows more about the program's behavior, either through compile time analysis or through run time analysis. So, for example, if a system provides support for load balancing, it knows more about the computational nature of the tasks and the placement of the tasks.

The advantage of acquiring information through language constructs and system libraries is obvious: minimal compiler effort is required. However there is one drawback of acquiring information through language constructs: if too many language constructs are needed, the language can become cumbersome and difficult to use. The advantage of acquiring information through compiler support is that the user need not be concerned about numerous or complicated language constructs. However, it may not always be possible to acquire information through compile time analysis. In general, it is easier and more feasible to acquire information directly through language constructs or system libraries.

We define the *degree of specificity* of a parallel programming language to be the extent to which one can automatically determine information about an identified set of behavioral characteristics for programs written in that language. In a language with high degree of specificity, information about program behavior is easily available, and hence automatic analysis is feasible. What are the implications of a low degree of specificity on automatic performance analysis? Some amount of automatic analysis is still possible for language with a low degree of specificity. In this thesis, we acquire information about a Charm program using the first two mechanisms, namely language constructs and system libraries.

1.3 Performance analysis of Charm programs

In this thesis, we are concerned with techniques for the analysis of the performance of Charm programs. Charm [4, 20, 21, 22][4] is an object-based, message-driven, and portable parallel language. We have examined two complimentary approaches to performance analysis: program visualization and automatic analysis. In the eventuality that automatic analysis is unable to determine the cause of poor performance, program visualization can be used for analysis.

In terms of program visualization, we have identified attributes of the message driven and object based execution model of Charm and its high-level support for load balancing. We have also identified techniques to appropriately display such information.

Charm programs provide a wealth of information for automatic analysis through:

1. Language constructs, which provide more information about sub-task decomposition and nature of information sharing.

[4] The discussion in this thesis is applicable to the C++ extension of Charm called Charm++

2. Libraries and high-level support, such as quiescence detection and dynamic load balancing, which provide more information about phases of program execution and the nature of task mapping.

Given such information, performance analysis can be very specific. For example, instead of being told that a particular processor is overloaded during a certain phase of the computation, and constitutes a bottleneck, a tool can give more specific information. It may inform that the overloaded processor is busy because of the large number of new (small grained) processes being created on it, which suggests using a better dynamic load balancing strategy. Alternatively, it may state that the overloading is due to the large number of requests for a particular data-item stored on this processor, thus suggesting replicating that data item as a solution.

1.4 Contributions of thesis

In this thesis, we examine two complimentary approaches for better understanding the performance of Charm programs:

1. *Language specific visual information:* We have identified information specific to Charm's message-driven and object-based execution model, and provided visual access to such information. This allows the user to easily relate performance metrics, such as granularity of tasks, to specific portions in their Charm programs.

2. *Automatic performance analysis:* Visual information can provide the user with significant amount of information about a program's performance. However, such information can get difficult to understand as the number of processors increase, or as the complexity of the programs increase. We have designed a framework for automatic analysis, where the performance data is examined automatically for common performance problems.

A primary contribution of this thesis has been to provide a methodology for automatic analysis. In particular, we have examined the validity of the hypothesis that *automatic analysis is feasible for a language with a high degree of specificity, such as Charm.* We have achieved the goal of automatic analysis as follows:

1. *Identification of performance analysis techniques*: We have compiled a list of previously known techniques and a list of new techniques for performance analysis. Many new techniques become necessary and possible because of our choice of the base language.

2. *Identification of behavioral characteristics*: Based on set of the techniques for performance analysis that we identified, we proceeded to identify the types of information about program behavior that was needed for such techniques.

3. *Development of a language with high degree of specificity*: Charm, the language chosen as the base language for research in this thesis, has a high degree of specificity due to its object-based and message-driven execution model. We have increased the specificity of Charm by adding multiple specific modes of information sharing and system libraries for quiescence detection and load balancing. As a result, a considerable amount of information about the characteristics of Charm programs is available automatically.

The techniques for language-specific display and automatic analysis are embodied in a Motif based performance analysis tool for Charm called Projections. A preliminary version of the tool, with only limited visual capabilities and no capabilities for automatic analysis, was described in [23]. Projections also includes algorithms to automatically reconstruct approximate real time order from available orderings. Since the hypothesis is not a quantitative one, we have used Projections to determine the efficacy of these techniques on a test-suite of Charm programs.

1.5 Outline of dissertation

Figure 1.1 shows an outline of the various topics discussed in this thesis and their interrelationships. In Chapter 2, we provide a brief introduction to the Charm parallel programming language and its model of execution. In Chapter 3, we describe the language features, system libraries, and the high-level support provided in Charm, which allows us to extract information about program behavior. In Chapter 4, we describe the manner in which events in Charm programs are traced, and our solutions to the problems of asynchronous clocks and perturbation encountered during tracing. In Chapter 5, we describe a few key attributes of the execution graph which are useful for automatic analysis. In Chapter 6, we discuss various techniques needed for the performance analysis, and their integration into a framework for automatic

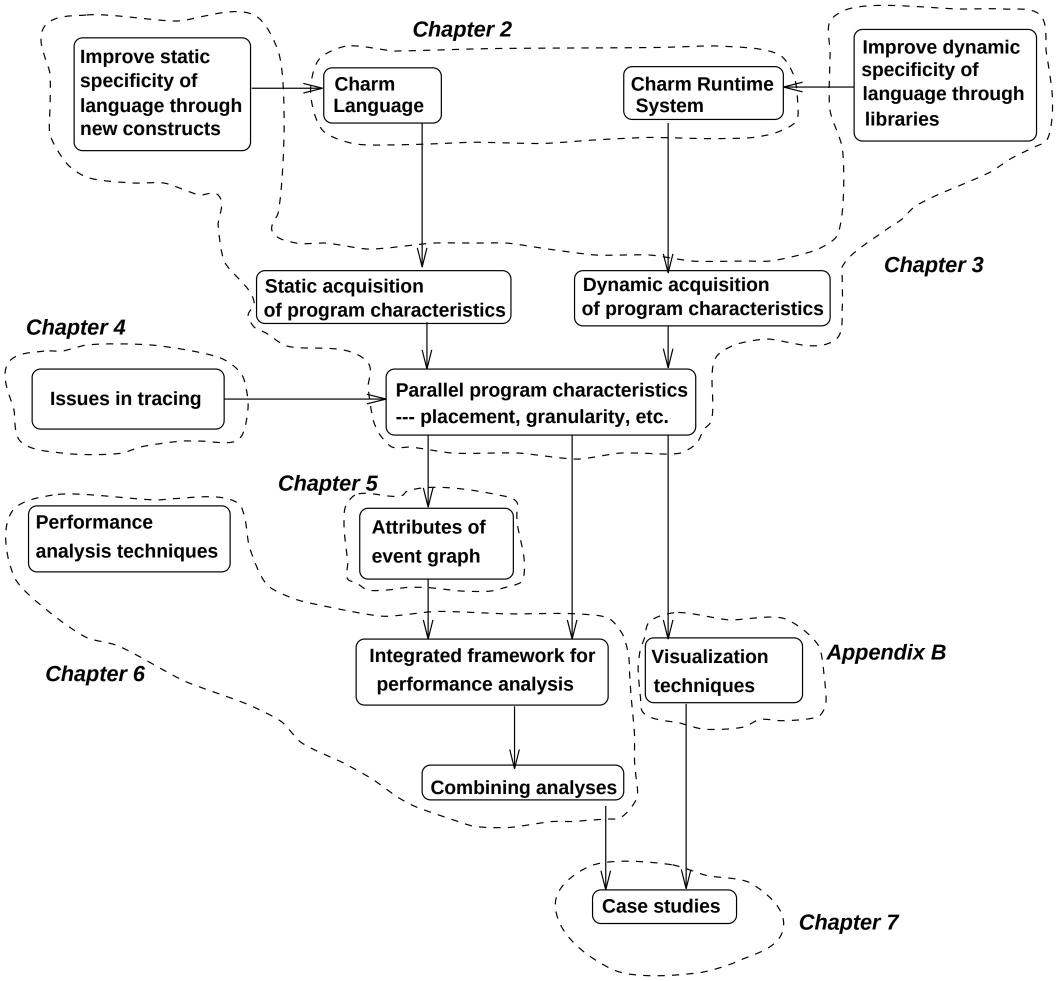

Figure 1.1: Outline of dissertation.

performance analysis of Charm programs. For the sake of completeness, we also describe in Appendix B how visual feedback of Charm-specific program parameters is provided. In Chapter 7, we present different case studies with which we have evaluated the techniques developed in this thesis. Finally, in Chapter 8, we summarize the thesis and present directions for future work.

Chapter 2

The Charm language substrate

In this chapter, we briefly describe the Charm programming language and its execution model.

2.1 The Charm programming language

The basic unit of computation in a Charm program is an entity called *chare*. The syntax of a *chare* is shown in Figure 2.1. A *chare* has its own data area, which is accessible to its *entry points*, and *private* or *public* functions. *Entry points* are blocks of C-code, which are sequentially executed when a message addressed to that entry point is delivered. *Private* functions are blocks of C-code, which are accessible only from within the chare. They provide a way for multiple entry points to share the same functionality without having to duplicate code. *Public* functions are also blocks of C-code which are accessible from outside the chare. A chare is similar to an object in that it provides data encapsulation. It is different from an object because it does not provide inheritance and polymorphism.

The basic mode of information sharing in Charm is a message. The syntax of a message declaration is the same as that of a *struct* declaration in C. Messages can be non-contiguous data, in which case the user needs to define *pack* and *unpack* functions (which are invoked automatically by the runtime system) to pack the message into a contiguous data format, and unpack from contiguous data into the old structure, respectively.

CreateChare (charename, entry, msg, [virtualID [, destPE]])

MyChareID(&chareID)

SendMsg(entry, msg, &chareid)

Chares are medium grained processes, and can be dynamically created using the *CreateChare* system call. A chare can determine its own address using the *MyChareID* system call. The chare can, after determining its address, send it to other chares in messages. Chares can send messages to other chares at known addresses using the *SendMsg* system call. There are other calls to allocate and free memory, and to destroy chares.

```
message Msg1 {
    Type1 field1;
    ...
}

chare Example1 {
    Local variable declarations

    /* Entry Point Definitions */

    entry EP1:  (message MESSAGE_TYPE1 *msgPtr)
        C-code-block
    ..
    entry EPn:  (message MESSAGE_TYPEn *msgPtr)
        C-code-block

    /* Local Function Definitions */

    private|public Function1(..)
        C-code-block
    ..
    private|public Functionm(..)
        C-code-block
}
```

Figure 2.1: Syntax of a chare.

Charm also provides another kind of process called a **b**ranch **o**ffice **c**hare (BOC). The syntax of a BOC is the same as that of a chare. A BOC is a replicated chare: there exists a branch or copy of the chare on each processor. All the branches of a BOC are referred to by a unique identifier. This identifier is assigned when a BOC instance is created, and may be passed in messages to other chares. Branch office chares can be created using the *CreateBoc* call. The definition of a BOC is similar to that of a chare. The *public* function of a BOC can be called by other chares running on the same processor, and provides a clean and easy-to-use interface for any process to access the local branch of a BOC. Branches of BOCs can interact with each other using the *SendMsgBranch* system call. A message can be sent to all the branches in a BOC using the *BroadcastMsgBranch* system call.

All Charm calls are asynchronous. For example, the *CreateChare* call takes a user-defined message to be delivered at an entry point of a chare that is to be created. The chare is not created immediately after the execution of the *CreateChare* call. Rather a *creation message* is generated, which consists of the user-defined message and some system information as header. The *creation* message is picked up for execution by the system at some later stage at which time the chare is created.

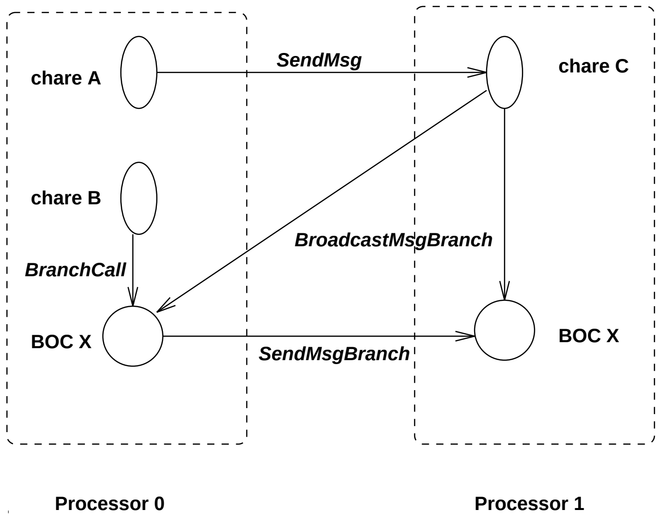

Figure 2.2: Execution model of a Charm program.

Figure 2.2 shows a picture of the basic programming model, as defined so far. Note that all communication across processors can occur only through messages.

2.2 The execution model of Charm

The execution model can be understood in terms of messages, message queues, and a message processing loop. *Creation* and *response* messages, generated as a result of the *CreateChare* and *SendMsg* calls, respectively, are queued up in message queues to be picked up at some later stage by the message processing loop. Message queues enables Charm to enforce various strategies in which *creation* and *response* messages are handled – fifo, lifo, or prioritized.

The message processing loop is essentially the same for shared and nonshared memory models. Inside the message processing loop, messages are picked up from the message queues and executed according to the message type – *creation* or *response*. For a *creation* message the system allocates space for the data area of the chare, and then makes a call to the specified entry point with the addresses of the data area and the user message as parameters. A *response* message contains the address of the chare, which is used to determine the chare's data area, and then the specified entry point is called with the addresses of the data area and the user message as parameters. In both shared and nonshared memory machines, a higher priority is given to *response* messages, under the assumption that these need to be processed faster because some chare is waiting for a response.

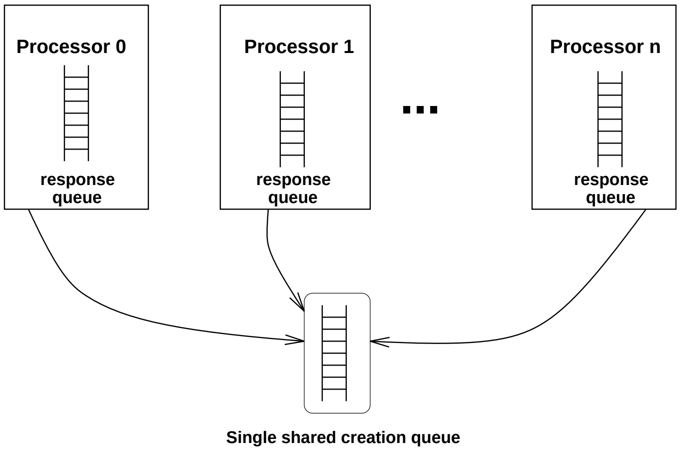

Figure 2.3: Charm runtime system for a small shared memory machine.

Current implementations of Charm do not support the migration of chares. This has two implications. First, since the chare does not migrate its address, after creation, remains fixed. Hence all *response* messages have a fixed destination. Second, the *creation* messages must be distributed amongst the available nodes to have a load balanced system. Response messages have a fixed destination, therefore they do not need to be balanced. The different natures of the *creation* and *response* queues necessitate different queues for the two types of messages. The implementation details of the message queues are different for shared and nonshared memory machines.

On shared memory machines (Figure 2.3), all processors share a common queue for *creation* messages, so that the work of creation can be balanced among the processors. However, each processor has its own local queue for *response* messages, since the response queues have a fixed

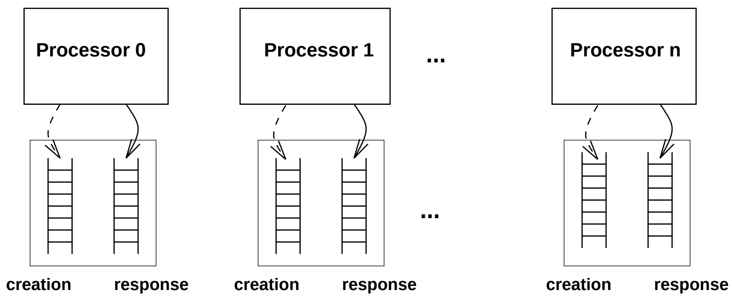

Figure 2.4: Charm runtime system for a nonshared memory machine.

destination processor, which is known at the time the *SendMsg* call was made. Saletore [24] has performed experiments with multiple queues and other scheduling strategies for shared memory implementations of Charm.

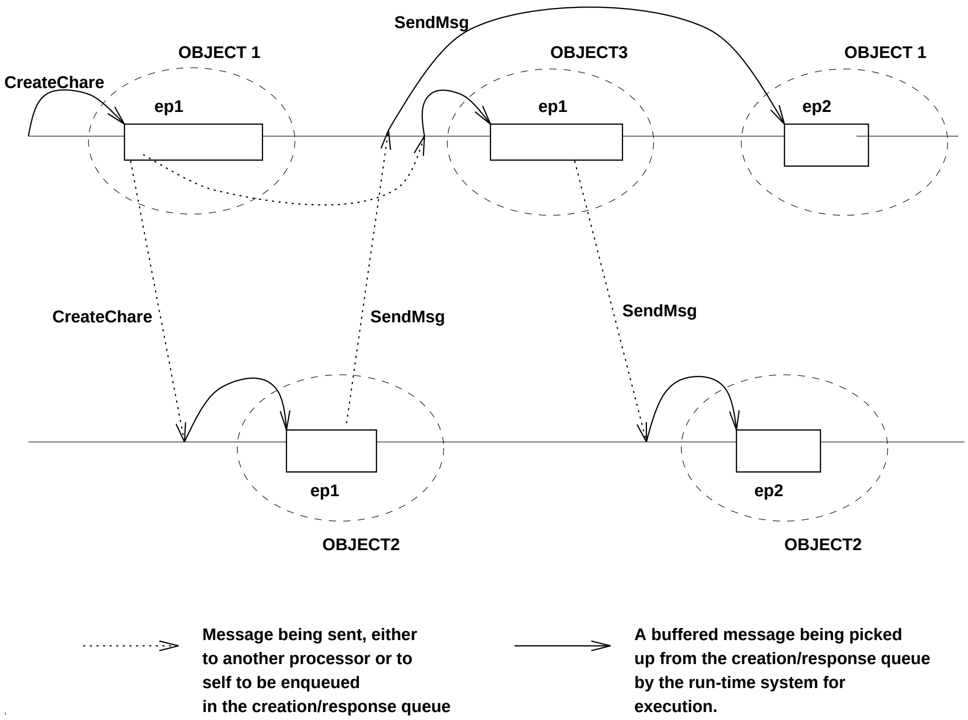

Figure 2.5: Events in the execution of a Charm program.

On nonshared memory machines (Figure 2.4), each processor has one local queue each for *creation* and *response* messages. Messages sent from other processors are enqueued in one of these queues according to the type of the message. The load balancing strategy then uses some scheme to balance the lengths of the *creation* message queues on the available processors.

Figure 2.5 shows the events in the execution of a Charm program. A message, created using the *SendMsg* system call for example, is first delivered to the destination processor, where it is enqueued in a *response* queue. At some later point in time, the runtime system, following a specified scheduling strategy dequeues the message, and executes the corresponding entry point. The scheduling strategy can be selected by the user. In addition priorities may be assigned to messages — the system executes the highest priority messages first.

Chapter 3

Specificity of Charm

A crucial step in automatic performance analysis is the automatic acquisition of information about a program's behavior. The first question that needs to be answered is: what sort of information about program behavior is needed? In performance analysis, the goal is to maximize the utilization of each processor [1]. This is achieved by maximizing the time a processor executes user code (*user time*), and minimizing the time a processor *idles* (*idle time*) or executes system related code (*overhead*). Idle time can be affected by the scheme with which tasks are placed on the processors (placement), the order of execution of messages, the extent to which tasks are balanced over the system (load balance), the number of tasks that can be scheduled independently at any given moment (the degree of parallelism), the amount of time spent in waiting for global synchronization, and the grainsize of tasks. The contribution of system overheads are affected by the grainsize of tasks and the time to access shared variables. Thus the characteristics of a parallel program that can affect its performance are: nature of sub-task decomposition, global synchronization, order of execution of messages, placement, grainsize of tasks, shared variable access time, and the degree of parallelism.

In Chapter 1, we had pointed out three methods to acquire information about a program's behavior, namely language constructs, system libraries, and high-level support. The basic language substrate of Charm, described in Chapter 2, has a high degree of specificity. Its

[1] In *speculative* computations, the amount of work that a processor does depends on the order in which it schedules their execution. Thus, a speculative computation can choose a poor schedule in which processors do a large amount of wasteful work, which keeps the the utilization of the processors high. Our primary goal is to maximize the utilization of the processors; however we attempt to do this by keeping the amount of speculative work low.

object-based and message-driven execution model is instrumental in providing us information about sub-task decomposition, placement, and granularity of tasks in Charm programs. We describe how this is possible in Section 3.1.

The execution model of Charm, described in Chapter 2, allows chares to share information with other chares only through messages. One can implement other modes of information sharing using messages. However, given the nature of message passing in a program, advanced compiler support, if at all possible, is needed to infer the specific information sharing mechanism being implemented. In order to increase the specificity of Charm, we augmented the existing language with multiple and specific information sharing mechanisms [25], which easily provide information about the nature of shared variables in a program, Note that information sharing abstractions also provide expressiveness for the programmer, in addition to their utility in performance analysis. In Section 3.2, we discuss how the specificity of Charm is enhanced with the addition of information sharing abstractions, which provide us with information about the nature of shared variables.

In Section 3.3, we discuss how the quiescence detection system library provides information about global synchronization. In Section 3.4 and 3.5, we discuss how the high-level support for load balancing and queuing in Charm provides more information about placement of tasks and their scheduling.

The basic Charm model, including chares and branch office chares, as well as various queuing and load balancing strategies, was developed independently of this thesis work and is used as a substrate. The information sharing abstractions, the quiescence detection algorithm, and the prioritized load balancing strategies 7.1 were developed specifically as part of this thesis.

3.1 Basic information

Charm programs can have two types of processes — chares and branch office chares. A branch office chare has a representative chare (branch) on each processor. Thus the placement of each branch of a branch office chare is known statically. A chare can be created in two modes — with or without specified placement. The nature of placement of a chare can be determined from the *CreateChare* call used; when no placement is specified, the exact processor on which the chare will be created is determined by the dynamic load balancing strategy. A chare that is created without any specified placement is automatically placed under the control of the dynamic load balancing strategy which specifies its placement.

The execution model of Charm is message-driven: every message is addressed to a particular *method* (also called the *entry point*) of some object; when the message is picked up for execution at its destination processor, it results in the invocation of the specified *method*. Further the execution model of Charm ensures that the code-block associated with a message is executed atomically, i.e., it cannot be pre-empted. In this programming model, the system can easily decompose the program into sub-tasks — the code-block associated with a message constitutes a sub-task.

We have instrumented the Charm run-time system to monitor various attributes of a message, such as sender and receiver objects, intermediate locations while being load balanced, and the times at which the message was created, enqueued, dequeued [2], and then processed. This information allows us to determine the granularity of tasks, the number of messages in a processor's message pool, and the utilization of each individual processor.

3.2 Information sharing abstractions

A parallel computation can be characterized as a collection of processes running on multiple processors. Depending on the programming model and language, it may have just one or many processes on each processor. As the processes are part of a single computation, they often have to exchange information with each other.

One of the most popular information sharing mechanisms is a shared variable. Two or more processes may exchange information by setting and reading the same shared variable. This model offers great simplicity as it appears to extend the sequential programming model in a natural manner. However information exchange through shared variables suffers from one major drawback: the difficulty of efficient implementation on large parallel machines. Shared variables can be implemented efficiently on small parallel machines, which physically share memory across a bus and can provide hardware support for a single global address space. However, many large scale machines available today, such as Intel iPSC/860 and Paragon, NCUBE/2, and CM-5,

[2] Note that in a dynamically load balanced system, such as the one provided by Charm, a message can potentially be enqueued and dequeued more than once.

include hundreds of processors. Implementing shared variables on such machines is difficult and inefficient.

Messages provide another important means of exchanging information between processes in systems such as PVM [14, 16], Express [17], and Actors [26]. Messages containing necessary information can be sent from a "sender" process to a known "receiver" process[3]. Most commercial distributed memory machines provide hardware support for message passing, so this mechanism to exchange information can be easily implemented. However message passing as the sole means of exchanging information may not be adequate, or may not be expressive enough to easily represent many different modes of information exchange. For example, in order to send a message, the sending process must know the identity of the receiving process. In many applications, such information may not be easily available. Message passing can also prove to be a cumbersome, if not an inefficient, mechanism to express information sharing between multiple processes. For example, read-only information[4] *can be* exchanged via messages in a language with message passing as the universal information sharing mechanism. But the cost of accessing the information is substantial. Access to the information can be optimized by replicating the read-only information on each processor. However the user needs to go into considerable effort in order to implement (with messages) a replicated variable, which is accessed through a unique identifier.

There exist other mechanisms to exchange information amongst parallel processes. The information sharing mechanisms provided by Linda [19] and Strand [29] suffer from the same problem: Each provides only a single information exchange mechanism. Compilers for languages with a universal information sharing mechanism often attempt to detect various modes of information sharing in order to produce more efficient object code. However the detection of a particular mode of information sharing can be imperfect and conservative at best. It would be more intuitive and convenient for the programmer to specify a mode of information exchange, rather than trying to fit all information sharing modes into the single mode of information exchange.

[3] In most current message-passing models, information can be exchanged only on a point-to-point basis. However, collective communication primitives are being designed by the message passing interface (MPI) standardization committee [27, 28].

[4] Read-only information is data that is initialized once and not altered thereafter.

In general, there are two problems with a single generic means of information exchange (whether it is a shared variable, a message, or some other mechanism):

1. Lack of expressiveness: A single generic means of information exchange may prove to be inadequate or cumbersome to express all possible information exchange modes in a program. The lack of specificity of a single mechanism also has negative impact on the understanding of program behavior.

2. Inefficiency: For any universal mechanism of information sharing, there will always exist modes of information sharing which cannot be efficiently implemented with the universal mechanism. Efficiency may be obtained for a limited set of modes by using sophisticated compilers to detect particular modes of information sharing. However there are limitations to what even a sophisticated compiler can do. In addition, universal information sharing mechanisms are usually not *efficiently portable*. Even though, a universal information sharing mechanism may be sufficient to express a wide variety of information sharing modes, it is unlikely that a particular method expressing an information sharing mode would be efficient across all parallel machine models. For example, an efficient implementation of a read-only variable on a shared memory machine would create a single shared variable, while an efficient implementation on a non-shared memory machine, would replicate the variable on all processors and refer to it by a single name. The single-shared variable method wouldn't be efficient on a large non-shared memory machine because each access would require messages. Similarly, it would be inefficient to install the replicated variable mechanism on a shared memory machine, where caches handle this automatically.

The problems with a single universal sharing mechanism suggest that a parallel language must provide multiple mechanisms to share information. Also, for portability, there must be a separation between the implementation of a particular mode of information sharing and its abstraction available to the user. Empirical observation of parallel programs suggests that processes share data in a few *distinct* and *specific* modes. We believe that such modes should be identified and explicitly supported in parallel languages and their associated models. We have identified and implemented five specific modes of information exchange in the parallel programming language Charm: *read only*, *write once*, *accumulator*, *monotonic*, and *distributed tables*. Read-only variables are the only true global variables in a Charm program — all other information sharing mechanisms are referred to by their unique identifiers. These abstractions have been implemented efficiently, and often differently, on different parallel machines in the context of Charm. However the abstraction is uniform to the user: it does not change from one machine to another. In this chapter we describe the syntax, semantics, usage, and implementation of these five specific information sharing mechanisms.

3.2.1 Read-only variables and messages

In many computations, many processes need *read access* to data that is initialized at the beginning of the computation, and is not updated thereafter. This mode of information sharing can be specified using the read-only mechanism. Charm provides two kinds of read-only information sharing: **read-only variables** and **read-only messages**. Read-only variables and messages can be declared in a Charm program as follows:

> **readonly** Type readname;
> **readonly** MsgType *readmsgname;

The essential difference between a read-only variable and a read-only message is that the latter is treated like any ordinary message: it can contain pointers to dynamically allocated memory, or variable sized arrays (which are automatically packed and unpacked to replicate the message on different processors).

Read-only variables and messages are initialized in the *CharmInit* entry point using the *ReadInit* and *ReadMsgInit* calls. Chares and branch-office chares can access read-only variables and messages via the *ReadValue* call. This call simply returns the (fixed) value of the read-only variable or message.

3.2.2 Write-once variable

In some computations, read-only information is available only after the parallel computation has proceeded for some time: the value is not available during the initialization phase of the program. In Charm, **write-once** variables support this mode of information sharing. Write-once variables are the dynamic counterpart of read-only variables. A **write-once** variable is created and initialized any time (and from any chare) during the parallel computation. Once created, its value cannot be changed. The creation is done via a non-blocking call *CreateWriteOnce* which returns immediately without any value. Eventually, the variable is "installed", and a message containing a unique global name assigned to the new variable is sent to the designated address. This unique name can be passed to other chares and branch office chares. They can access the variable by calling *DerefWriteOnce(name)*, which returns the value of the write-once variable. The *DerefWriteOnce* call is non-blocking, i.e., it returns a pointer to the write-once variable immediately. The cost of a *DerefWriteOnce* call is the cost of a local function call.

3.2.3 Accumulator variable

In many computations, a variable is needed to count the number of occurrences of an event, the number of processes of a certain type, etc. Such a variable is updated by a commutative and associative function. Charm provides this mode of information sharing through the **accumulator** data abstraction. Figure 3.1 shows the syntax of an accumulator definition. The accumulator abstract data type has associated with it a message containing the data area of the accumulator data type, an initialization function (init) and two user defined commutative-associative operators (add and combine).

```
accumulator acc_type {
    Message_Type *acc;
    Message_Type *init ()
        C-code-block
    add ()
        C-code-block
    combine ()
        C-code-block
} ACC_TYPE;
```

Figure 3.1: Syntax of an accumulator definition.

An instance of an accumulator can be created using the *CreateAcc* call. This call can be made statically (inside CharmInit) or dynamically — if it is created statically the identity of the variable is available immediately, but if it is created dynamically the identity is returned to a specified address. The identity of an accumulator can be passed in messages to other chares and branch-office chares. The identity of a statically created accumulator is available in the *CharmInit* entry point, and it can be more conveniently accessed if it is made into a read-only variable.

The system is free to maintain multiple copies of an accumulator variable: in some cases there may be one copy per processor, while in other cases a few processors might share a copy. The initialization function **init** is called, possibly on multiple processors, upon invocation of the *CreateAcc* call.

In the user program, an accumulator variable can be modified only via the *Accumulate* call. This call results in the first operator, **add**, being called, which *adds* to the accumulator variable in some user defined fashion, while maintaining commutativity and associativity.

A destructive read on an accumulator variable can be performed with the *CollectValue* call. This call is also non-blocking, and results in the eventual transmission of the value of the accumulator to a specified address. The second operator combine is called by the *CollectValue* call only if the system has maintained more than one copy of the accumulator variable. The operator takes two accumulator variables as operands and *combines* those variables element by element, again in a commutative-associative manner.

3.2.4 Monotonic variable

In some computations, processes need frequent access to a variable which changes monotonically. Such a variable is typically used in branch&bound computations. Charm provides this mode of specific information sharing with the **monotonic** abstract data type. Figure 3.2 shows the syntax of a monotonic definition. The monotonic data type has associated with it a message containing the data area of the accumulator data type, an initialization function (init) and a user defined monotonic operator (update).

```
monotonic mono_type {
    Message_Type *msg;
    Message_Type *init ()
        C-code-block
    update ()
        C-code-block
} MONO_TYPE;
```

Figure 3.2: Syntax of a monotonic variable declaration.

An instance of a monotonic variable can be created using the *CreateMono* call. This call results in the function **init** being called; **init** initializes and returns the initial value of the monotonic variable. Like accumulator variables, monotonic variables can be created either statically or dynamically.

Subsequent updates to the monotonic variable can be carried out through the *NewValue* call. This call results in the corresponding **update** function being called. The function **update** must be a monotonic and idempotent (multiple application of the function with the same parameters has the same result) function for the domain over which the monotonic variable is defined.

The (approximate) current value of a monotonic variable can be read by any chare at any time using the *MonoValue* call. The value returned by the *MonoValue* call will satisfy the following properties:

1. The value will be true, i.e., it will be either the value assigned during initialization, or provided thereafter by some *NewValue* call.

2. The value returned will be at least the best value provided by a *NewValue* call by the same process.

3. The system will do its best to provide the best value of the monotonic variable supplied by any *NewValue* call until that point in time.

3.2.5 Distributed table

In many applications, data can be split into many portions, and each portion can be accessed by a subset of processes in the system. Further, the subset of processes that access a portion of the table may not be pre-determinable. In other applications, processes that do not know each other's identity may need to exchange information. Charm provides **distributed tables** as a means of sharing information in these modes.

```
table table_type {
    Message_Type *msg;
    hash()
        C-code-block
} TABLE_TYPE;
```

Figure 3.3: Structure of a distributed table abstract data type.

The syntax of the definition of a distributed table appears in Figure 3.3. A **distributed table** consists of a set of entries. Each entry consists of some data and a key (an integer) that uniquely identifies each distinct piece of data. The data in an entry in the table is a message. Like all other messages, data items in a table can have dynamically allocated areas either declared explicitly by the user or through Charm constructs. The function **hash** is used to define a mapping from an integer key to a specific processor to which each data must go. If these functions are not specified by the user, the system provides default hash functions.

There are various asynchronous access and update operations on entries in distributed tables. An entry can be added to the set using the *Insert* call. The user can search for a particular entry (using its key) using the *Find* call, and remove an entry from the set using the *Delete* call. The current implementation of distributed tables in Charm is a restricted version of this more general formulation: the hash function is specified by the system and the data is a string of characters.

A distributed table also provides a good distributed interface between two components of a parallel program. In sequential programming, data exchange between two phases of computation in an application is achieved through a sequential point of transfer, e.g., via parameters in a function call. Such a mechanism of exchanging data between two phases of a parallel program can create a bottleneck. Distributed tables are a suitable mechanism to exchange data in a distributed manner. The matrix multiplication example in Section 7 illustrates distributed data exchange — the result of a matrix multiplication is stored in a distributed table to be exchanged with the computation in a later phase of the application.

3.2.6 Choice of a specific sharing mechanism

Different application programs may need different modes of information sharing. The user must decide which specific mechanisms of information sharing to use in order to best represent a particular mode. Some of the guidelines that one follows in deciding which specifically shared variable to use in a program are:

- If the information needs to be passed to a process whose identity is known, then a message should be used.

- If the information needs to be passed to one or more processes whose identity is not known, then a distributed table should be used.

- If the information needs to be shared among most of the processes in the system, and it does not change after being initialized, then a read-only or a write-once variable should be used.

3.2.7 Implementation

On *shared memory machines* with a small number of processors, each information sharing abstraction is implemented as a shared variable. Read-only variables have no locks to control access, since they are accessed only in the read-only mode. Accumulators and monotonic variables have an associated lock, and operations on them are performed in a mutually exclusive manner using locks. Write-once variables have no lock to control access; however in order to establish a unique name for a write-once variable, processors need a lock for synchronization. A distributed table is managed as an array of chains of entries. A *hashed chaining* scheme is used. The key of an entry is used to map into an index in the array, which is a chain of entries whose keys map to the same index. A lock is associated with each index in the array to provide mutually exclusive access to chains. The same scheme is used for both small and large shared memory machines.

Since write-access to an accumulator might happen very often in some applications, a more efficient implementation is possible. In such an implementation, each processor would have a local copy of the variable. When the variable is finally read the processors use locks to add all the local copies. This scheme might be more suitable for larger shared memory machines also.

On a nonshared memory machine, a read-only variable or message is implemented as a replicated variable; each processor has its own local copy of the variable/message.

The remaining modes of information sharing are implemented as branch-office chares. Each branch maintains a local copy of the variable in the case of write-once, monotonic, and accumulator variables. In the case of distributed tables, the entries are divided amongst the branches of the BOC.

Write Once variables are initialized by the *CreateWriteOnce* call. A copy of the variable is first sent to the branch on processor 0 of the corresponding BOC. This branch assigns the variable a unique index, which serves as the identifier for the write-once variable, and then broadcasts the value and identifier of the variable to each of the branch nodes. Each node, after creating a local copy of the write once variable, sends a message to the branch on processor 0 (along a spanning tree in order to reduce bottlenecks) that it has created the variable. When it has received an acknowledgement message from all the nodes, the branch on processor 0 sends the identifier of the write once variable to the specified address. A write once variable can be read by the *DerefWriteOnce* call. This call returns the pointer to the local copy of the variable. The pointers to all the WriteOnce variables are stored in an array indexed by the identifier of the write-once variable.

The *Accumulate* call results in the application of the *add* function on the local value on the branch chare. The *CollectValue* call is used to (destructively) read the value of an accumulator variable. This call results in a broadcast to all branches. The branch chares then combine the values of the accumulator on their local processors. This is accomplished by each branch chare combining its value with the values of the accumulator on its children in the spanning tree, before sending the accumulated value up to its parent. At interior nodes of the spanning tree, the values are combined using combine. The branch on processor 0 communicates the final value to the supplied chare.

An update on a monotonic variable is performed by the *NewValue* call. The *NewValue* call can be implemented in two different ways: *combining via a spanning tree* or *flooding.* In the *spanning tree* implementation, the call results in the branch chare updating its local value (with the corresponding update), and sending a copy of the new value up to its parent branch chare in the spanning tree on the processors. Every branch combines values it receives from its children with its own by waiting for some fixed interval of time before sending its local value up to its parent branch chare. The root of the tree broadcasts the value to all branch chares. In the *flooding* implementation, the call results in the branch chare updating its local value, and sending a copy to its immediate neighbors (a dense graph on the processors is chosen). A processor which receives a new value from a neighbor, first checks if the value provided is better than what it currently owns. If the value is better, it propagates a copy of the value to its own neighbors. In both of the above implementations, the value of every update may not be simultaneously available to every branch, but shall be eventually available. Users may choose the monotonic implementation best suited to their application. A monotonic variable can be accessed using the *MonoValue* system call; this call returns the value of the local copy of the variable on that node.

Updates on entries in a distributed table can be carried out by calling the system calls *Insert* and *Delete*. Again, as in the case of shared memory systems a *hashed chaining* scheme is used. The key of an entry is hashed to obtain the processor number of the branch which stores the portion of the table to which this entry belongs, and the index in the table on that branch. An update message is sent to the required branch, which carries out the update operation and back-communication of update, if specified in the call options. The *Find* call is used to read entries in distributed tables. The key provided is used (as described above) to determine the branch and index. A message is sent to the corresponding branch chare to find the entry and reply back to the supplied address.

The implementation of many specifically shared variables using branch office chares and messages suggests that these two could be used to provide other necessary information sharing mechanisms in Charm.

3.2.8 Related work

A considerable amount of work has been done in achieving shared memory on distributed systems. In general, one difference in our approach from other approaches in distributed shared memory is that we do not provide all forms of sharing. Instead we have identified and implemented specific modes of information sharing, which are often used in parallel programs, and which have efficient and portable implementations. Note that the general-purpose shared variable can be implemented by the user with the branch office chare construct. However we have made no attempt to make such general-purpose implementations efficient through compiler optimizations.

The first work in distributed shared memory was done by Li [30] who showed that distributed shared memory provided the convenience of shared memory, while simultaneously being efficient. The first distributed shared memory system, called Ivy [30], provides for a strictly coherent shared memory on distributed systems. Our philosophy towards shared memory on distributed systems is substantially different, since we believe that the general purpose shared variable cannot be efficiently and scalably implemented on large distributed memory machines.

Therefore, we identify and provide only those limited forms of information sharing which can be efficiently and scalably implemented.

Emerald [31, 32, 33] provides a uniform object model. The user is responsible for both the distribution of data and objects, and the migration of objects, if necessary. An access to a remote object is done through an RPC, while local objects can interact through shared memory. Object migration is efficient, and can be performed often to ensure efficiency of access to objects. The work done in Amber [34] is derived from Emerald. Amber also provides strictly coherent shared memory across multiprocessors. This is done by migrating threads to the place where the accessed data exists: this resolves some problems with Emerald. However, this approach has one major drawback: if threads access non-local data very often threads could potentially migrate for every data access.

Clouds [35] provides sharing at an object level. The owner of an object can be changed dynamically causing it to move to the new owner. However, as an efficiency consideration, an object can be locked for owner. Thus if the user knows the access patterns of an object, he can lock it into the processor which accesses it the most: the remaining processors can then access it remotely without causing needless migration.

Orca [36, 37] supports shared objects. In Orca there are no global variables or pointers, and accesses to objects are made through well-defined functions. Only one operation can occur on an object at any given point in time. Orca utilizes advanced compiler optimization techniques to make object accesses efficient. One design consideration for Orca programs needs to be the definition of an object: if it is too coarse grained, then the execution is serialized, and if it is it too fine grained, considerable costs must be incurred in providing access to the object from everywhere.

The work done for Munin [38] is similar. They have identified different forms of coherence, as opposed to the single form of strict coherence provided by shared memory programming models. This allows programmers to specify less restricted versions of coherence in programs written for shared memory machines. Such programs can then be executed efficiently even on distributed memory machines. A difference in our approaches is that we provide specific modes of information sharing, while they provide looser (and less expressive, since each form of coherence can provide many modes of information sharing) forms of coherence. This increased specificity of the information sharing modes in Charm has been used in automatic performance analysis.

The work done for *PoliSim* [39] is also similar. In *Polisim*, data in the form of multidimensional arrays can be distributed on processsors in one of six possible ways: *Row, Column, RowSkew, ColSkew, Diag*, and *AntiDiag*. Further access policies to the data can be chosen from one of six possible policies: *StaticSeq, Seq, Opt, Remote, Migrate, Invalidate*, and *Update* The user has the freedom to mix and match to arrive at the optimal distribution and policy combination. As in the case of Munin, a different in the approach in *PoliSim* and ours is that our information sharing mechanisms are more specific.

3.3 Quiescence

One characteristic of program behavior that is important in performance analysis is global synchronization. In SPMD programs, a *barrier* operation results in a global synchronization. This follows because there is only one process on each processor, and so if it participates in a barrier operation, all processes participate, which implies that the system has a global synchronization.

Charm allows multiple processes on each processor. Further, in Charm, reductions are asynchronous, i.e., they block only the processes involved leaving the rest to proceed on with their computation. Therefore a reduction operation does not by itself signify a global synchronization among all objects. One method of achieving global synchronization is through quiescence detection. Loosely defined, quiescence is the state of execution in a system, when there are no messages left in the system. The absence of messages in the system signals that there is no activity. Further, if spontaneous activity is not permitted, then the absence of messages also indicates that no activity can occur in the future.

The user can request that the system detect a quiescent state with the system call *StartQuiescence(ep, chareid).* As a result of this call, the system initiates the quiescence detection algorithm, which sends a message to the specified entry point, *ep*, of the chare which is specified by *chareid*, when quiescence is detected. At this point, the user has the flexibility of either terminating computation, or beginning a new phase of computation by generating fresh messages, e.g., in IDA* (Iterative Deepening Algorithm) [24].

3.3.1 Implementation

Quiescence detection is implemented as a branch office chare which runs a two-phase distributed algorithm. All communication between the branches occur along a spanning tree covering the processors. In the description below all references to the *parent*, the *children*, the *root*, or the *sub-tree* of a processor are with respect to the corresponding entities in the spanning tree. We denote the first and second phases of the algorithm as Phase 1 and Phase 2, respectively. The algorithm use three kinds of control messages:

1. *Initialization* messages, which are broadcast to every processor, and result in the initialization of Phase 1 or Phase 2 on all the branches.

2. *Idle* messages, which are sent up to the parent during Phase 1. An idle message signifies that each processor in the sub-tree below has been idle at least once since the last *idle* message. It does not mean that all the processors were idle *simultaneously*.

3. *Activity* messages are sent up to the parent during Phase 2 and contain a report of activity (creation and processing) in the sub-tree rooted at the sending processor.

We use the construct — **wait until (condn)** — in the description of our algorithm. The process executing the **wait until** is suspended till such time as the **condn** becomes true. In the algorithm, each component maintains the following counts:

1. n_c: this is the sum of the number of *activation* messages **created** on this processor.

2. n_p: this is the sum of the number of *activation* messages **processed** on this processor.

3. N_c: the number of messages created in the sub-tree rooted at this component.

4. N_p: the number of messages processed in the sub-tree rooted at this component.

These are initialized to zero at the beginning of Phase 1 and Phase 2, and are sent up with *idle* and *activity* messages.

The algorithm appears in Figure 3.4. Phase 1 is called on each processor immediately before the user computation begins. Only one phase of the quiescence detection algorithm will be active at any time. In Phase 1, each leaf component waits until its processor is idle and then sends an idle message to its parent with the counts N_c and N_p initialized to n_c and

```
Phase 1() {
    N_c = 0; N_p = 0;
    wait until (RecdMsgsFromChildren()); /* wait until messages have */
                                          /* been received from all children */
    add to local N_c and N_p the values recd. from children;
    wait until (Idle()); /* wait until this processor has no activation messages */
    N_c = N_c + n_c; N_p = N_p + n_p;
    if (RootSpanTree()) /* check if this processor is the root of the spanning tree */
        if (N_c ≠ N_p) Broadcast message to begin Phase 1
        else {
            N^old = N_c /* N_c == N_p */
            Broadcast message to begin Phase 2 }
    else Send message with N_c and N_p to parent
}

Phase 2() {
    N_c = 0; N_p = 0;
    wait until (RecdMsgsFromChildren()); /* wait until messages have */
                                          /* been received from all children */
    add to local N_c and N_p the values recd. from children;
    wait until (Idle());
    N_c = N_c + n_c; N_p = N_p + n_p;
    if (RootSpanTree()) /* check if this processor is the root of the spanning tree */
        if (N^old == N_c) Report Quiescence
        else Broadcast message to begin Phase 1
    else Send message with N_c and N_p to parent
}
CreateMessage() { n_c + + }
ProcessMessage() { n_p + + }
```

Figure 3.4: Quiescence detection algorithm.

n_p, respectively. All other branches wait until they receive one idle message from each child, adding the values of N_c and N_p in these idle messages to their local values. Having received idle messages from all its children, the component waits until its processor is idle, and then it sends an idle message to its parent. The idle message contains the values of the counts N_c and N_p, which have been incremented with the values of n_c and n_p, respectively, on that component. When the root has received idle messages from all its children branches, it decides whether the system can be *idle* by comparing the values of N_c and N_p. If they are equal then there's a high probability (but not a certainty; see explanation of Figure 3.5 below) that all *activation* messages have been processed in the system. If the two counts are not equal then the root initiates Phase 1 again, otherwise the root initiates Phase 2 on all the branches.

In Phase 2, the branches send up their *activity* report messages containing the new values of N_c and N_p. *Activity* messages from branches are combined in the same way as in the first phase of the algorithm. When the root component has received one activity message from each of its children, it compares the old and the new values of N_c and N_p. If these values are the same it implies that there has been no new activity in the system, and the root reports *quiescence*; otherwise the root initiates Phase 1 again.

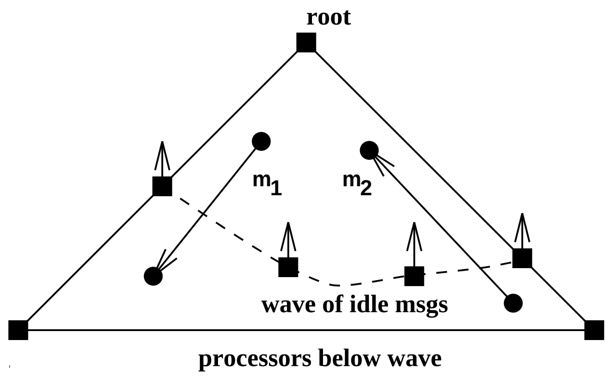

Figure 3.5: Wave of idle messages in Phase 1 of quiescence detection.

Note that a single phase is not sufficient to guarantee that the system was quiescent, because the counts N_c and N_p might match at the end of Phase 1 even though all messages were not processed. Figure 3.5 shows the 'wave' of *idle* messages being passed up the spanning tree in Phase 1 — the processors below the wave have already sent out their *idle* messages, while the processors above the wave haven't yet sent out their *idle* messages. Consider the following scenario in Figure 3.5: message m_1 is created on a processor above the wave of Phase 1 messages and sent to a processor below it, while message m_2 is created on a processor below the wave of Phase 1 messages and sent to a processor above it. Further, assume that the processing of m_2 did not create new activation messages. In this case, when the wave reaches the root, the following is true:

- Creation of m_1 is counted.
- Processing of m_1 is not counted.
- Creation of m_2 is not counted.

- Processing of m_2 is counted.

At the end of Phase 1, the counts may[5] match even though m_1 was processed after the idle message was sent. The processing of message m_1 could have generated more new activity, and therefore it is incorrect to infer from the counts matching at the end of Phase 1 that the system is quiescent.

A proof of the correctness of the algorithm and the performance of its implementation appears in Appendix A.

3.3.2 Related work

Much work has been done before on quiescence detection [40, 41, 42, 43, 44, 45, 46, 47], both for synchronous and asynchronous systems. We shall briefly discuss some previous work on quiescence detection in asynchronous computational models.

Lai [47] and Huang [46] present schemes which use distributed snapshots to detect quiescence in asynchronous distributed systems. In Lai's approach, a predefined process combines local snapshots (taken spontaneously by processes) into a global snapshot, which it then uses to determine if quiescence has occurred. Huang's method is essentially similar, the only difference being that any processor can initiate the collection of the global snapshot. Lai and Huang use different techniques to ensure that the global snapshot is *feasible*, i.e., it does not contain processing of messages if the corresponding creation is not also part of the snapshot. One drawback of both their schemes is that they do not extend to systems where processes can be created dynamically. In addition, Huang's scheme can become very expensive because each processor can initiate a global snapshot if it is idle. In the worst case, when all processes go idle everyone will initiate a broadcast. Lai's scheme does not suffer from this drawback, however the lack of coordination between the collection of local snapshots means that a considerable global snapshots may be collected.

Mattern[45] presents an elegant credit based scheme to detect quiescence in dynamic, asynchronous, distributed systems. In the worst case, the number of control messages needed by Mattern's algorithm is the number of activation messages. The scheme works by distributing

[5] If we assume that no other activity is occurring in the system then the counts will match; however if there is other activity in the system the counts may still match because there are more than one pair such as m_1 and m_2.

one unit of credit amongst active processes and activation messages. When a process creates an activation message, it divides its credits equally between itself and the message. A process returns its credits to the monitoring process (the quiescence detection algorithm) when it becomes passive. An activation messages' credit is passed on to its receiving process if the receiving process is passive; otherwise (if the receiving process is active) the credit is returned to the monitoring process. When the monitoring process has received the original one unit of credit, it reports quiescence.

Whenever processes split up their credits between themselves and activation messages they create, fractions are generated. Fractions cannot be accurately computed with the current representation of floating point numbers. Mattern presents an approach wherein fractions need not be explicitly computed. Notice that all fractions are of the form 2^{-n}, and hence can be represented by the negative of the logarithm (called *credits*), i.e., n. The scheme that Mattern outlines to solve the problem of having to compute fractions exactly involves computing the set of missing credits (credits possessed by activation messages, active processes and control messages at that time). The set is updated whenever credits are returned to the monitoring process. The set of missing credits is maintained by the monitoring process, which is a single process running on one processor. The cardinality of this set is bounded by the sum of the number of activation messages, active processes and control messages over the entire system. Assuming that the memory requirements of each activation message is constant, it would mean that the memory requirements of the monitoring process to maintain the set of missing credits is of the same order as the number of messages in the entire system. In addition, the monitoring process becomes a bottleneck in bigger distributed systems, since all credits are being returned to the one processor on which the monitoring process is located.

3.4 Dynamic load balancing

One key goal in a parallel program is to keep the amount of user work on different processors balanced. In a system, which does not provide dynamic load balancing, the user needs to explicitly implement the load balancing scheme. Often the only mechanism available to the user is a message, and the resulting implementation may obscure the strategy used to balance load. Further, the system is unaware of the specific purpose of a task and cannot determine whether or not the task can be moved around and performed at a different processor. An automatic and dynamic load balancing strategy provides us the following:

1. Information that the tasks being load balanced have no specific destination and can be moved around.

2. Information about the mapping of tasks and their computational requirements.

In the Charm execution model, all messages are deposited in a message-pool from where messages are picked up by processors whenever they become free. In the shared memory implementation of Charm, the pool of messages is shared by all processors; in the nonshared memory implementation, the message-pool is implemented in a distributed fashion with each processor having its own local message-pool. New-chare messages (message to create a new chare) are the only messages that may not have a fixed destination, and are therefore the only messages which can be load balanced. In nonshared memory implementations, load balancing strategies attempt to balance the sizes of the local message-pools on each processor. New chare messages may move among the available processors under the control of a load balancing strategy till they are scheduled for execution. Once picked up, a new chare message results in the creation of a new chare, which is subsequently anchored to that processor. Charm provides the user with the facility of multiple dynamic load balancing strategies, such as random, ACWN [48], and token [49]. Depending on the nature of the application, the user may choose to link in the one which is most ideal for their application.

Figure 3.6 shows the basic interface between the Charm runtime system and the load balancing strategies. Dynamic load balancing strategies are implemented as branch office chares in Charm. There are basic interface functions to initialize the BOC (*LdbInit*), and add information to (*AddStatus*) and extract (*ExtractStatus*) load and balance information from a message. Each strategy also has a interface function called *NewChare_From_Local*, which is called by the runtime to give to the load balancing strategy a new chare when it is created. It's the responsibility of the strategy to determine where the new chare is sent. The load balancing BOC also has an entry point called *NewChare_From_Net* at which it receives user messages from other branches of the load balancing BOC, and decides what happens to them: enqueue them locally or send them somewhere else.

```
BranchOffice LoadBalance {

    entry LdbInit: (message InitMsg *msg)
    { /* initialize the branch of the strategy */ }

    public NewChare_From_Local(msg)
    { /* do something with a locally created new chare */ }

    entry NewChare_From_Net: (message void *msg)
    { /* do something with a new chare sent from elsewhere */ }

    public ExtractStatus(msg)
    { /* receive status information of sending processor */ }

    public AddStatus(msg)
    { /* piggyback status information for destination processor */ }
}
```

Figure 3.6: Shell of a dynamic load balancing strategy in Charm.

3.5 Queuing strategies

In Chapter 2, we described the runtime execution model for Charm. Messages arriving at a processor are enqueued in either the *creation* or *response* queues, and are scheduled for execution under the control of a queuing strategy. Charm allows the user to select the scheduling strategy from a number of available strategies, such as lifo, fifo, fifolifo, etc. The user can exercise greater control over the scheduling of messages by attaching priorities to them. The system has various prioritized scheduling strategies, again user-selectable, that schedule messages according to their priorities [50]. The choice of a particular scheduling strategy is made at link-time, so the runtime system has information about the strategy chosen.

From the perspective of performance analysis, queuing strategies provide information about the nature of scheduling in the program, such as when messages were queued and dequeued, and the order in which messages were executed by the runtime system.

3.6 Summary

In this chapter, we have shown how information about the behavioral characteristics of Charm program can be acquired. We have shown how language constructs, such as chares, and messages provide useful information about program behavior. We have also shown how the specificity of Charm can be increased with the addition of five specific means of information sharing. These mechanisms add to the expressive power of any language. The abstraction presented to the user for any of these five mechanisms is identical on all machines. However, for efficiency, the implementation of each mechanism has been specifically tuned for different architectures. For example, a monotonic variable is implemented as a shared variable on shared memory machines and a branch office chare on nonshared memory machines. We have also shown how high-level support, such as quiescence detection, dynamic load balancing, and queuing strategies provide greater information about Charm programs.

Chapter 4

Issues in tracing: asynchronous clocks and perturbation

In Chapter 3, we have broadly discussed the type of information available about the behavior of Charm programs. In Section 4.1, we describe the specific information acquired for Charm programs and the methodology for acquiring such information.

Analysis is carried out by comparing actions on different processors at the "*same time*". This would make sense only if the meaning of "same time" across processors was clear. Often, in parallel systems, the values of the local clocks are not synchronized, so that "same time" is not a well defined concept. In such cases, it becomes difficult and error-prone, to reason about events across processors. In Section 4.2, we discuss the problem of asynchronous clocks and our solution. In this thesis, all performance analysis is done with the assumption that clocks are synchronized. Traces with synchronized clocks can be obtained either directly if the parallel machine provides them, or through runtime clock synchronization schemes, or using the schemes we discuss in this chapter. The performance analysis techniques discussed in this thesis do not depend on the clock synchronization schemes we outline in this chapter; rather they will work with any trace which uses synchronous clocks.

Tracing perturbs the execution of the program, and consequently alters the timing patterns and information in the program. Since, accurate timings are necessary to make any correct inferences in performance analysis, the perturbation due to tracing must be reduced as much as possible. In Section 4.3, we describe our methodology to reduce perturbation due to tracing.

4.1 The nature of raw performance data

A Charm program can be executed in two different modes. In the first, the *normal* mode, execution proceeds without any events or activities being recorded. In the second, the *projections* mode, the system records information about defined activity types. A program can be executed in any one of the two modes by linking with the appropriate libraries — there is no need to add instrumentation, or to recompile the user program in order to generate the trace information. Trace is collected through a combination of two mechanisms: *static* program information and *runtime* program information. Each processor has its own local buffer to record trace data. When a buffer overflows during an execution run, it is written out to a log-file corresponding to that processor. At the end of the execution of the program, the buffers are written out onto each processor's log-file. We shall now briefly describe the nature of information collected statically and at runtime.

4.1.1 Static information

The static information in a Charm program consists of the structure of the program itself. This includes the following:

1. *Chares and branch office chares:* At the highest level, the information consists of the names of the chares and branch office chares that constitute the program.

2. *Message types:* For each message type, information about its size.

3. *Entry points:* For each chare (BOC), static information exists for the entry points that make up the chare (BOC) and the type of message received by each entry point.

4. *Specifically shared variables*: Other information recorded statically includes names and types (accumulator, etc.) of various specifically shared variables in the program.

Note that some other static information follows implicitly from the information recorded above. For example, if an object is a branch office chare, then there is static information about the placement of that object: there is one copy of the object on each processor.

4.1.2 Runtime information

Various parameters of a message are recorded when it is created, enqueued, and dequeued, and when the system starts and finishes processing it. In most cases, the information recorded includes the following:

1. *event-type*: the type of the event, i.e., one of *create*, *enqueue*, *dequeue*, *begin processing*, and *finish processing*.
2. *message-id*: a number that uniquely identifies the message on the processor on which it was created
3. *processor-id*: the processor on which the message was created.
4. *entry name*: the name of the entry point to which the message is addressed.
5. *message-type*: the type of the message, i.e., one of *NewChareMsg*, *ForChareMsg*, *BocMsg*, or *BroadcastBocMsg*.
6. *chare-id*: the unique number identifying the chare that created the message.
7. *time:* the time at which the event occurred.
8. *priority*: the priority of the message, if any.

Even though, specifically shared variables are implemented using messages at the lowest level, additional information is needed. For example, for an *Insert* request on a distributed table other useful information needed includes the value of the *key* provided.

4.1.3 Event graph

An *event*, as used in this thesis, is the execution of an entry point. Since an entry point can be activated multiple times by different messages during the execution of a program (not simultaneously, however), there can be many events in the trace corresponding to the execution of the same entry point. Notice that since entry points execute over a period of time, events do not represent point-objects. Let, V denote the set of all *user* events in the execution of the program, and let V_p denote the set of user events on processor p. An event $v \in V$ has:

1. a time v_c at which the call to create it was made,

2. a time v_s at which the system picked up the message in order to create the event, and

3. a time v_f at which the system finished creating it.

Definition 1 *An event $y \in V$ is said to be created by an event $x \in V$, if the message for the entry point corresponding to y was created in the entry point corresponding to x.*

Let $x \rightarrow y$ denote the fact that x created y. Further, since each message can be created only while one entry point is executing, we have the following observation:

Observation 1 *An event can have only one creator, i.e., for an event y, there can exist only one event x, such that $x \rightarrow y$.*

Let $E = \{(x, y) \mid x \rightarrow y\}$ be the set of edges on the vertices defined by the set V. Now (V, E) defines a graph of the events in the execution of a Charm program. We denote (V, E) as the *event graph*. The following notations for an event e are used in describing algorithms throughout this thesis:

1. *CreationTime*: the time at which the user requested for e to be created, i.e., e_c,

2. *BeginProcessingTime*: the time at which the system started to create e, i.e., e_s,

3. *TransmissionTime*: the time taken by the system system to transmit the message corresponding to e.

4. *EndProcessingTime*: the time at which the system finished creating e, i.e., e_f.

4.2 Asynchronous clocks

A predicate for correct analysis is the availability of synchronized clocks on all processors. In the absence of such synchronization, analysis can be potentially erroneous, because two events that did not occur simultaneously in real time may appear as if they did and vice versa. Consider the example in Figure 4.1. It is a plot over time of the creation of *StartComputation* and of the processing of *NextComputation* events that occurred in the execution of a parallel molecular dynamics program EGO on a network of five workstations. The basic computational structure of EGO is iterative. In each iteration, the same number of *NextComputation* events occur, followed by a global synchronization on all processors, which is followed by the creation of *StartComputation* events to start the next iteration. The *StartComputation* events are all created on processor 0, and broadcast to all processors to start the next iteration.

The plots in the figure seem to indicate that the last iteration was very long, and a much larger number of *NextComputation* events happened. However, we know from the program that these statements cannot be true, as only an identical number of *NextComputation* events occur in each iteration. The reason for this anomalous behavior is the asynchrony in clocks, which makes it appear as if the *StartComputation* events were created earlier than what actually happened.

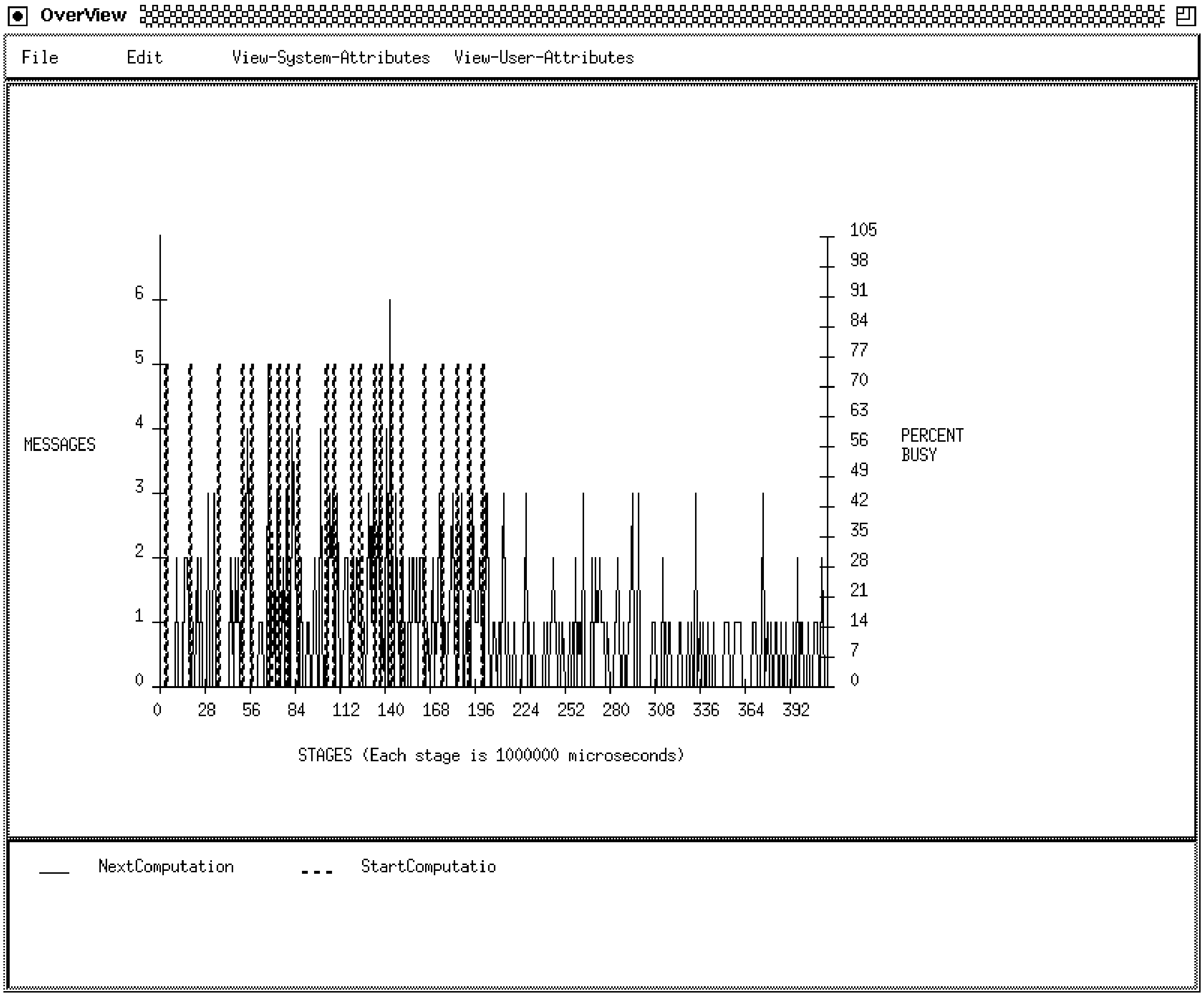

Figure 4.1: Performance data for EGO without any real-time reconstruction.

A more severe aspect of asynchronous clocks can be the presence of messages which violate causality: their processing time on one processor precedes their creation time on another. Following Heath, we call such messages tachyons [1]. Figure 4.2 shows an example of a tachyon.

There are two popular approaches to obtain synchronous clocks in asynchronous systems:

1. *Software mechanisms* treat *local* time as a data value. Processors exchange local values periodically, and achieve synchronization by updating local clock values depending on the corresponding values on the neighbors. However, software synchronization methods are at best expensive and inaccurate. In order to achieve tight synchronization between processors, time values must be exchanged sufficiently often. But since the overheads of frequently exchanging information can be substantial, there is a limit to how often clock information can be exchanged. Since one cannot incur excessive overheads for clock synchronization, this necessarily means one must compromise the accuracy of synchronization.

   In special cases, such as on hypercubes with known clock shifts and in the absence of faulty clocks, inexpensive and reasonably accurate software synchronization methods can be implemented, such as the one implemented for PICL by Dunigan [51]. The clock-skew approach is not general purpose enough, however, since it does not work on distributed machines with unknown clock skews, or on a distributed system where individual machines might have different clock speeds. Mills [52, 53] has reported that software synchronization methods can take many tens of milliseconds on a network.

2. *Hardware mechanisms* for clock synchronization are inexpensive and accurate. The basic mechanism is through phase-locked clocks: the clock signal generated by each processor's clock is a vector combination of the signals generated by all the other clocks in the system. One potential problem for hardware mechanisms is the time lag in transmitting clock signals to other processors. Shin [54] proposed a mechanism which can tolerate both faulty clocks and time lags for clock signal transmission. However, his mechanism requires $O(n)$ inputs for each clocks, where n is the number of clocks to be synchronized. Some reduction in the requirement of the number of inputs to each clock can be achieved by synchronizing small groups of clocks: however the synchronization is not accurate then.

[1] The term tachyon originates in the physical sciences. A tachyon is a particle that travels faster than light.

A second hardware mechanism is to have a single clock drive all other clocks. This has been implemented for the NCUBE-II.

A third hardware mechanism is to have a single hardware monitor which receives all tracing events and provides a timestamp for them. There is no potential for asynchrony because the monitor can concurrently accept inputs from all processors, and it uses a single clock to determine a timestamp for an event. Such a mechanism, called HYPERMON, was implemented for the Intel parallel machines by Malony et. al. [55, 56].

As we mentioned above, software methods tend to be expensive and inaccurate. Further, it is expected that most vendors of parallel machines would provide synchronous clocks at the hardware level in the near future. For these reasons, we have chosen not to implement runtime software schemes for clock synchronization.

Some synchronization mechanism is still necessary because hardware synchronization mechanisms are not feasible for an increasingly popular "parallel" computer: a network or a cluster of workstations. Most often, each workstation in the cluster is used in a time-sliced mode between multiple users. Different processes in a user's parallel program running on different workstations can therefore be scheduled at different times. Traditional software synchronization mechanisms need an estimate for the time it would take for a message to travel from one processor to another. In the case of a network, it is determined by the network load and the type of network: ethernet, FDDI, ATM, etc. One additional complication is the non-deterministic nature of time-slicing: the process to which the message is addressed may be time-sliced out when it reaches the destination node. Under these circumstances, software synchronization methods are not very effective.

Typically networks of workstations are spread across a wider area. The time lag to transmit clocks signals under such conditions can be large and unpredictable. This increases the complexity of traditional hardware synchronization methods, such as phase locked clocks, substantially, and thereby make them unfeasible. Even if some inexpensive hardware mechanism could be designed for clock synchronization, it would only synchronize the absolute value of the local clocks. In a time-sliced workstation environment, the absolute value of the local clock does not provide a fair indication of the user-time, because, in addition to the process time, it also includes the time for which the process was time sliced out. There is an additional problem with using wall clock time. In a workstation environment, many processes of the program can be executed on the same processor. In such a case, the workstation is time sliced between two processes of the same program. Therefore, a wall clock time measurement for each process will include the times for which both processes were time-sliced out and the time for which only the other process was time-sliced out. The actual time for a process is again an indeterminate fraction of the wall clock time. Networks therefore provide a unique environment where no traditional clock synchronization schemes work well.

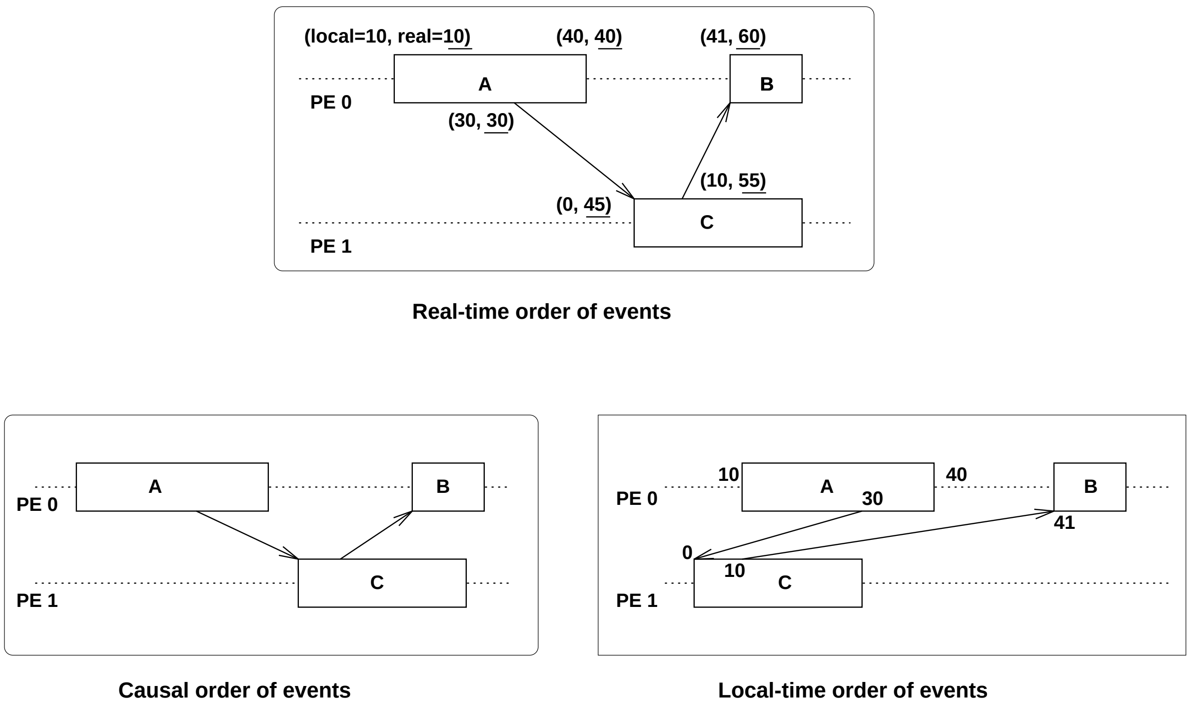

Figure 4.2: Various orderings of actions in a parallel program; the numbers indicate the local and real times for each event.

It seems that we need some method to reconstruct real-time order. What ordering of events is available from which real time ordering can be reconstructed? Consider the actions A, B, and C in Figure 4.2. The first ordering diagram shows the *real-time* ordering of actions (the two numbers indicate the local and the real times for each event). *Real-time* is the time determined using a wall-clock or some global external timing device. The real time ordering is not available to us in a system without synchronous clocks. The orderings shown in the second and third diagrams, however, are available to us. The second ordering diagram shows the *causal* ordering of actions, which consists of the partial order on the actions due to the *creation* relationship.

The third ordering diagram in the same figure shows the *local-time* ordering of actions A, B, and C. *Local time* is the time recorded by a processor's clock. Notice that the local-time ordering can introduce inconsistencies, e.g., C seems to happen before A, even though A created event C. If the clocks of all processors were synchronized, one would be able to obtain the real-time ordering, which maintains the partial orders dictated by local-time and causal ordering. In the absence of a synchronous clock, one only has the local-time and the causal orderings. Note that neither of them are by themselves sufficient and there is no way to get back the actual real-time ordering. However, we can achieve some degree of synchronicity by using the causal and local-time orderings available to us to reconstruct an approximate real-time ordering. In this section, we shall briefly describe two post-mortem real-time reconstruction algorithms that we have implemented.

4.2.1 First real-time reconstruction algorithm

Our first algorithm derives from Lamport's [57] work on logical clocks in distributed systems. Lamport's original scheme was meant for logical synchronization of clocks in a distributed system. In Lamport's scheme, each member of the distributed system has a local clock, which is incremented by one for every local event. Every message before being sent to a remote processor is timestamped with the current value of the local clock. When the message is received at the other end, that local clock's value is first set to the greater of the timestamp of the message and the value of the clock, and is subsequently incremented by one.

One fundamental difference in our approach arises because we attempt to recreate real time, and not just logically synchronous time as in Lamport's work. Further, our algorithm is post-mortem: real-time is reconstructed from (possibly) asynchronous traces. The basic idea behind our algorithm is to identify events that violate causality, and delay the processing time of each such event so that their new processing time is greater than the time at which they were created, thereby satisfying causality. Further to better simulate the parallel machine on which we are running the program, the processing time is delayed (with respect to the time of creation) by the time necessary to communicate the event to a remote processor. This is another difference with Lamport's scheme, where the logical clock ticks were incremented by one for any event.

Figure 4.3 shows the first algorithm. A list (*being_processed*) of the earliest processing event on each processor and the earliest task (*min_task*) among these are maintained. The algorithm

```
TASK *task, *min_task;
#define IsTachyon(task) (CreationTime(task) ≥ ProcessingTime())

SkipToProcessing(task) { while (!IsProcessing(task)) task=task->next;}

RightShift(TASK *min_task) {
    shift = ProcessingTime(min_task) - CreationTime(min_task)
            + TransmissionTime(min_task);
    for (task=min_task; task!=NULL; task=task->next)
        EventTime(task) += shift;
}

DetermineTachyons() {
    for (i=0; i<maxpe; i++) {
        task = transaction_list[i].head;
        AddList(being_processed, SkipToProcessing(task));
    }
    min_task = LeastList(being_processed);
    while (min_task) {
        if (IsTachyon(min_task)) RightShift(min_task);
        SkipToProcessing(min_task->next);
        DeleteList(being_processed, min_task);
        min_task = LeastList(being_processed);
    }
}
```

Figure 4.3: First algorithm for real-time reconstruction.

proceeds by checking if the current minimum, *min_task*, is a tachyon. If it is not, then the least processing task on that processor is updated. If it is a tachyon, then its processing time is set to a time after its creation, which takes into account the transmission delay for the event. In order to maintain relative local ordering, the timestamp of events after the current minimum on the same processor are also increased by the same amount. The minimum, *min_task*, is then removed from the *being_processed* list, and a new minimum is computed, and the process repeats until *min_task* is NULL. We will refer to the process of increasing an event's timestamp as *right-shifting*.

Figure 4.4 shows an example of the various stages in the working of the algorithm for a sample trace. Initially events A and C are in the list *being_processed*, and C is *min_task*. C is also a tachyon, so all the events on processor 1 are shifted to the right, so that C does not

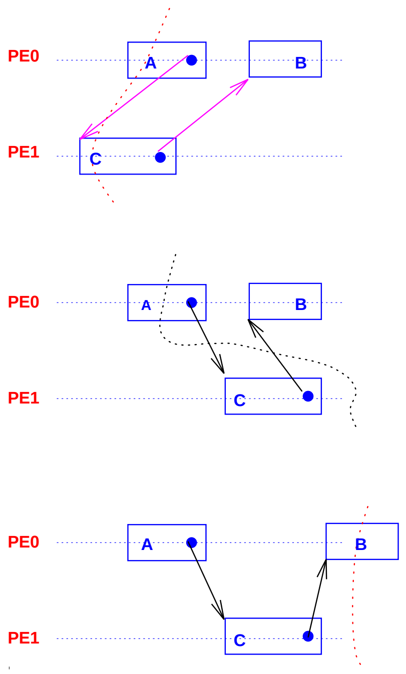

Figure 4.4: Stages in the execution of the first real-time reconstruction algorithm.

violate causality. Once that is done, C is removed from the *being_processed* list. Now, event B becomes a tachyon, so it is right-shifted, and all three events satisfy causality.

However this algorithm does not eliminate all tachyons. Figure 4.5 shows a counter example. In the first step, task B is a tachyon, so the algorithm right-shifts tasks on processor 1 to rectify that. Note that after this first right-shift, task B is no longer considered by the algorithm. Next the algorithm consider event C, which is also a tachyon, so it is also right-shifted. However, right-shifting C increases the creation time of task B causing it to become a tachyon again. Since B will no longer be considered it will be left behind as a tachyon by the algorithm.

A tachyon can be left behind by this algorithm *only* if the timestamp of its creator (on another processor) is greater than the event in the *being_processed_list* for that processor. For in such a case, the creator event could be moved to the right at some later time thereby possibly causing all events it creates to become a tachyon. One simple solution to this problem is to retain the *min_task* in the *being_processed* list if its creator event has still not been examined.

The complexity of this algorithm is $O(n^2)$, where n is the number of tasks: In the worst case, one could have to right-shift all following events for every event, which is $\sum_{i=0}^{n-1} i = O(n^2)$

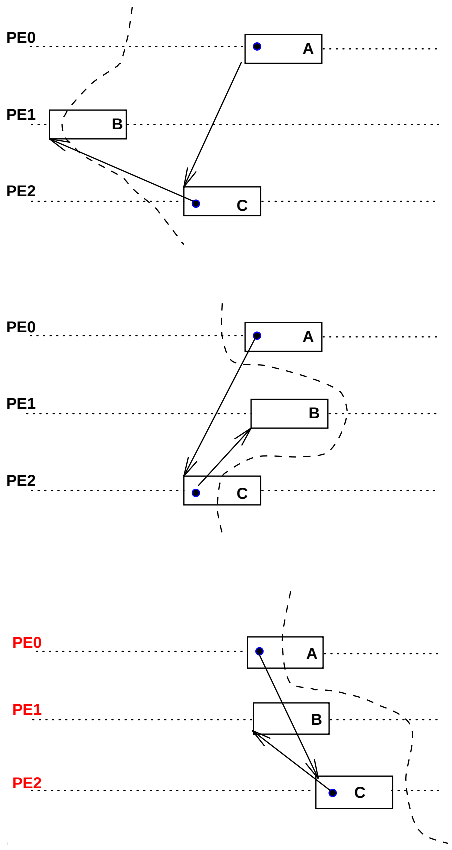

Figure 4.5: A tachyon can remain even after a pass by the first real-time reconstruction algorithm.

operations. Figure 4.6 shows the effects of the algorithm on the traces in Figure 4.1. A total of 472 tachyons were detected before the application of the algorithm.

4.2.2 Second real-time reconstruction algorithm

One problem with the first algorithm is the "unreal" jumps in real-time with the arrival of a message. If a message arriving at a processor is a tachyon, then the algorithm increases the local value of the clock on that processor in order to preserve causality for the arriving message. In general, the increase in the clock's value can be substantial enough to give the perception of a rough clock. The second algorithm attempts to resolve this problem by "simulating" the execution of the program, under the following constraints:

1. The local ordering of events on a processor is maintained.

2. Causality is preserved.

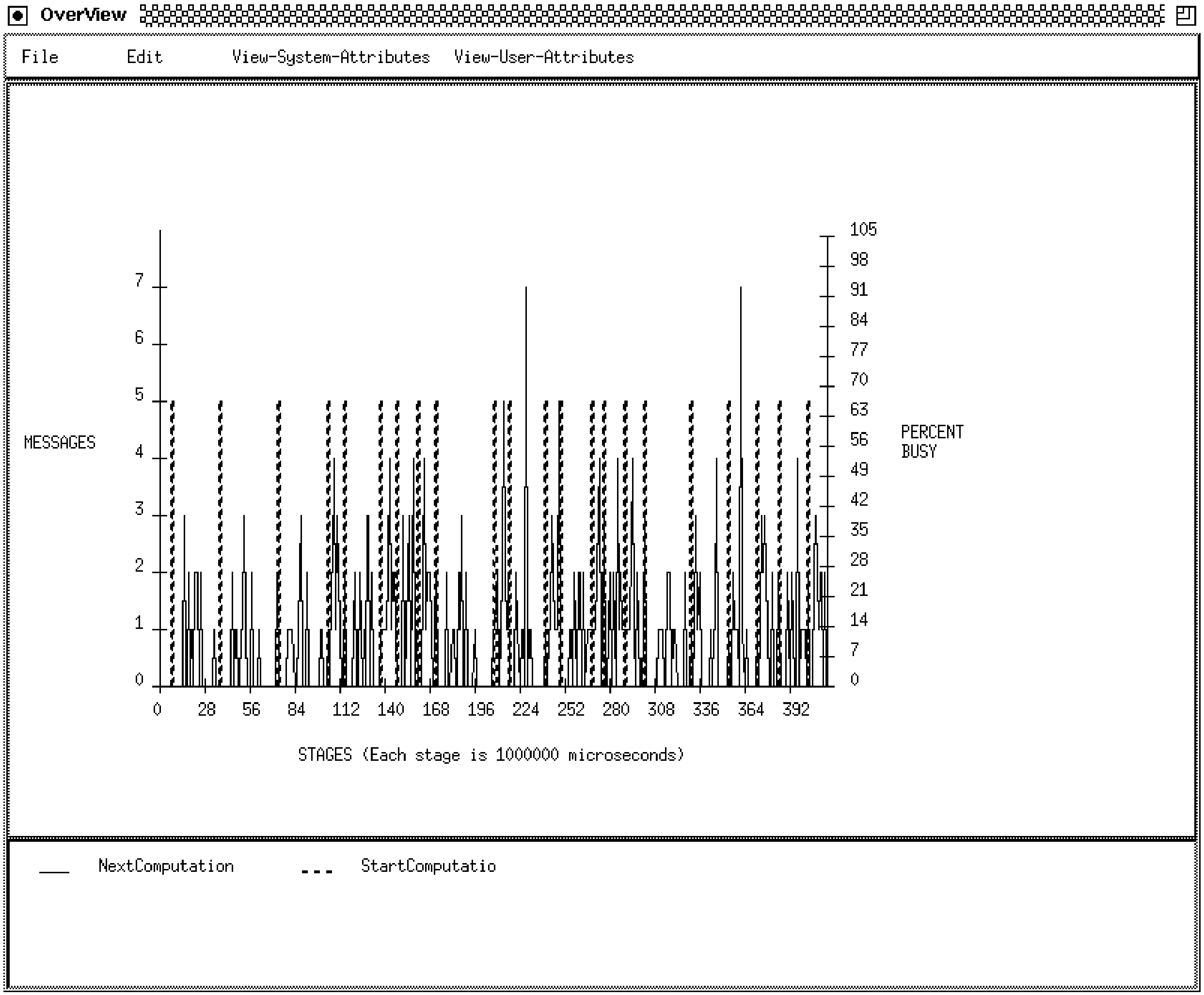

Figure 4.6: Performance data for EGO after the first real-time reconstruction algorithm.

3. The time taken by a message is determined by its size and the latency parameters of the network.

The second algorithm appears in Figure 4.7. The basic idea is to maintain a list of "ready" events and schedule the earliest event from amongst them. An event is created in one of two modes: *waiting* or *ready*. An event is *waiting* if the preceding events on the same processor have not yet been processed. The event becomes *ready* once the preceding events on that processor have been processed. *Ready* events are inserted into the *ready_list*, while *waiting* events are inserted into the *creation_list*. The algorithm schedules the execution of the earliest event (in time) from the *ready_list*, and updates its timestamp if the event is a tachyon. Notice that

```
unsigned int *shifts;
TASK *task, *min_task;
#define IsTachyon(task) (CreationTime(task) ≥ ProcessingTime())

ReadyList(TASK_LIST *task_list) {
    TASK_LIST *ready_list = NULL;
    for (;task_list; task_list=task_list→next)
        if (CreationTime(task_list→task) > CurrentTime(task_list→task→pe))
            AddList(ready_list, task_list→task);
    return ready_list;
}

SkipToProcessing(TASK *task) {
    for (;!IsProcessing(task);task=task→next) {
        if (IsCreationTask(task)) AddList(creation_list, task);
        EventTime(task) += shifts[task→pe];
    }
    if (task) EventTime(task) += shifts[task→pe];
}

RightShift(TASK *t) {
    shift = ProcessingTime(t) - CreationTime(t) + TransmissionTime(t);
    shifts[t→pe] += shift;
    EventTime(t) += shift;
}

DetermineTachyons() {
    shifts = (unsigned int *) malloc(sizeof(unsigned int)*number_pe);
    for (i=0; i<number_pe; i++) shifts[i] = 0;
    for (i=0; i<number_pe; i++) SkipToProcessing(transaction_list[i].head);
    min_task = LeastList(ReadyList(creation_list));
    while (min_task) {
        if (IsTachyon(min_task)) RightShift(min_task);
        DeleteList(creation_list, min_task);
        SkipToProcessing(min_task->next);
        min_task = LeastList(ReadyList(creation_list));
    }
}
```

Figure 4.7: Second real-time reconstruction algorithm.

only the timestamp of the offending event is updated; the timestamps of the remaining events on that processor are appropriately altered when the algorithm examines those events. After the earliest event has been scheduled, the next event on that processor is inserted into the *ready_list*, and all events created in between are inserted into the *creation_list*. Figure 4.8 shows the effect of the second algorithm on the events in Figure 4.1. Note that approximate real-time orderings generated by the two algorithms is qualitatively similar.

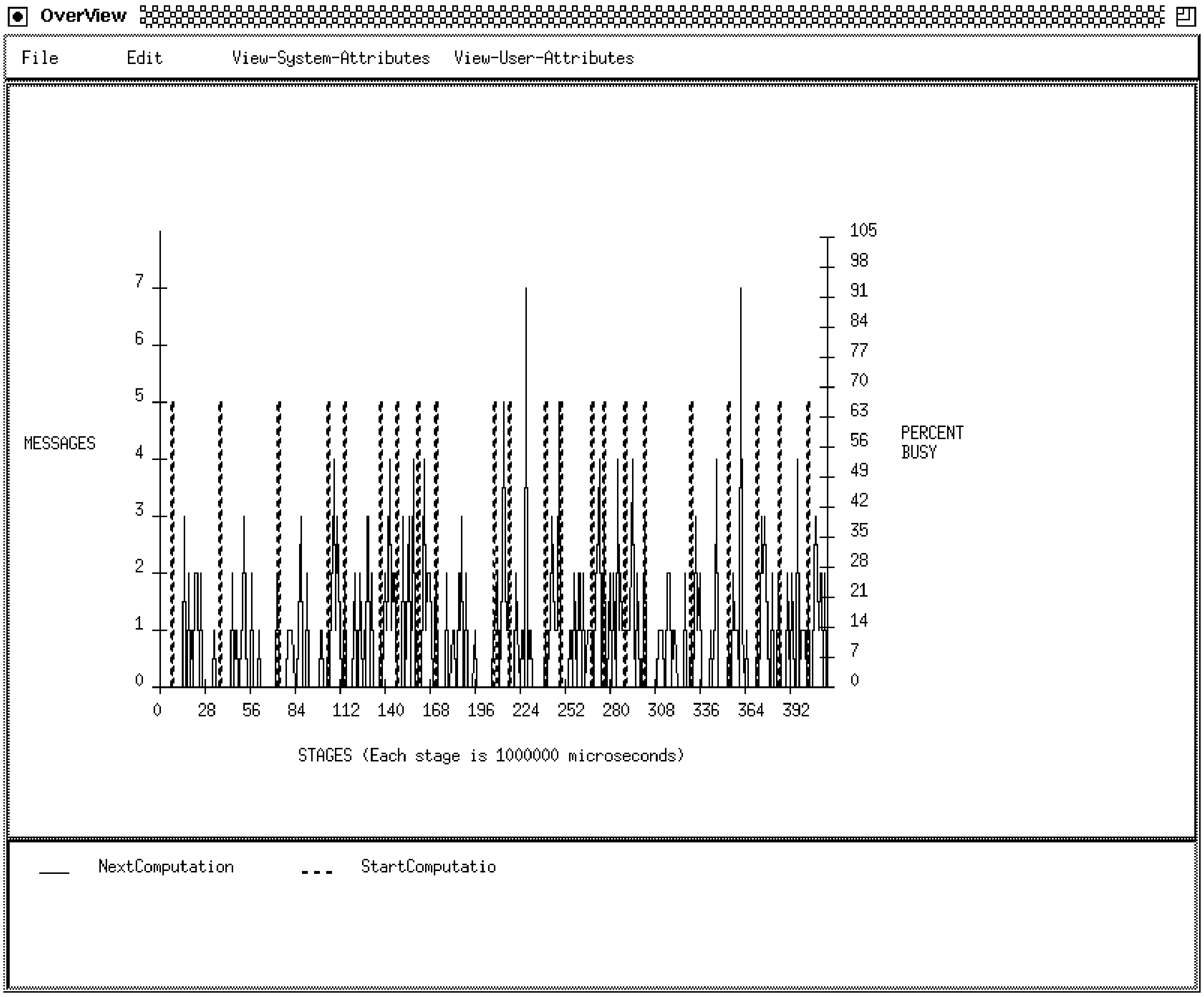

Figure 4.8: Performance data for EGO after second real-time reconstruction algorithm.

The complexity of this algorithm is $O(n * log(p))$, where p is the number of processors. Each event is examined only once to determine whether it is a tachyon, and is shifted only once as a result (the single shift takes into account all previous shifts on that processor). However to update the minimum event from the p events in the list of ready events takes $O(log(p))$

operations, and this happens after each event is examined, therefore the overall complexity of the algorithm is $O(n * log(p))$.

4.2.3 Related work

Our approach has similarities with the schemes proposed by Neiger [58] and Welch [59] for logical synchronization of clocks in distributed systems. In their approach, each processor's clock ticked naturally, and events were processed when their scheduled execution time (one greater than the creation time of the event) matched the local clock value on the destination processor. Again, as with the Lamport's scheme, our approach is different because it reconstructs approximate real-time, and not just logical synchronous clocks. Further, unlike these schemes, our approach is post-mortem.

Malony [56] has developed a method for explicit reassignment of timings measured in tracing to account for perturbation due to tracing. He defines an approximation due to trace manipulation to be a feasible execution if the total ordering of events in the measured time is preserved. He notes that what is really desired is not just a feasible execution, but a *likely execution.* A likely execution is a subset of all feasible executions that are most probable. He points out that determining likely executions is a difficult problem because complete information about loop scheduling algorithms and data dependence information is needed. He describes methods for conservative approximations which take into account known data dependence information and attempt to account for *hidden dependences* (dependences which have not been monitored). In re-ordering traces, Malony achieves the goal of maintaining total ordering by accounting for known dependences and making only pessimistic assumptions about hidden dependences.

A similarity in his work on perturbation analysis and our work in approximate real time reconstruction is the maintenance of all known dependences in the program. His work is different because it assumes that the measured traces are generated using a synchronous clock, so that total ordering imposed by time is consistent with the partial ordering imposed by dependences. Further, his method attempts to maintain this initial total ordering. In our work, we do not have the initial consistent total ordering, and attempt to reconstruct it from avaialable partial orderings. The differences in our approaches stem from this basic difference in the nature of available information.

4.3 Perturbation of program execution

In tracing a parallel program, one will always introduce additional operations which will perturb its execution. Trace gathering, therefore, has two conflicting goals: one to gather as much information about the program as possible, and the other to perturb the execution of the program as little as possible. The amount of perturbation for each trace event depends largely on the amount of data being traced, but is generally small. The execution of the program is perturbed to a much larger extent when the traces are output to disk. This becomes necessary eventually, because arbitrarily large traces cannot be stored on a processor. One way to address both problems simultaneously is to keep the size of the information stored for each trace event small. Since lesser data is being stored, the perturbation per call is lesser. In addition, since lesser data is being stored per call, more trace calls can be made before disk i/o becomes necessary. Thus the perturbation due to disk i/o is postponed.

Charm is a medium grained programming language. The recommended granularity of entry points, which receive small messages, is about 2-3 milliseconds. We trace five events for each message on the average: creation, queuing and dequeuing, and start and finish of processing. The perturbation due to tracing five events is substantially less than the recommended least granularity for most entry points, therefore the overall perturbation is not substantial. In most programs, the overhead of tracing is less than 5% of the execution time. Thus, the size of the traces, and hence the consequent perturbation, is a lesser problem in Charm programs. Nevertheless tracing is still a problem, and we have mitigated its effects by using the *replay* mechanism.

Instant replay was introduced by Blanc and Mellor-Crummey [60] as a mechanism to debug the asynchronous behavior of a parallel program. In a parallel program's execution, a bug can manifest itself because of an unusual ordering of events. The bug may not recur if the experiment is repeated under the control of a debugger, because the debugger may alter the original ordering of events. In such cases, it would be helpful to be able to deterministically reproduce the bug. The instant replay mechanism allows a user to reproduce a program's execution.

The key idea in a replay is to identify atomic events and record their order of occurrence in the execution. The execution can be replayed by re-executing the atomic events in their recorded order of occurrence. A replay cannot take into account spontaneous events, e.g., events which occur periodically with no other causal events.

In our strategy, the first execution run is used to collect minimal trace data about entry points. Since entry points are atomic events in a Charm program, this information is sufficient for replay. Subsequently, the program is re-played, at which time extensive trace data is gathered. This trace is used to replay the program execution by re-executing each entry-point on each processor as recorded on the trace.

How much does tracing save? In order to be able to replay a Charm program, one needs to replay the order of execution of messages. The only information needed to replay a Charm program is the order of processing of events on each processor (no information is needed about creation, enqueue, and dequeue events); an event can be uniquely identified by the following pair:

1. *message-id*: the number which uniquely identifies a message on the processor on which it was created.

2. *processor-id*: the number of the processor on which the message was created.

The trace data necessary for replay is an ordered set of such pairs, where the ordering is imposed by the order in which the events occurred on that processor. Since the time at which events, such as creation, occur in the replay can be different from the original run, it becomes necessary to also record information about the creation, enqueue, and dequeue of the message. For each message, the following additional information is recorded:

1. *time*: one needs to record the times for various events because they would change in the replayed version.

2. *event-type:* the type of the event, namely, creation, enqueue, dequeue, or processing.

As we have mentioned before in Section 4.1, other information that is needed for performance analysis includes the name of the entry point, the unique id of the chare, the priority of the message, and the the type of message. Without any compression, the size of the trace recorded has thus been reduced by half! Table 4.1 shows the time for three different runs of a Jacobi program on 4 Sun workstations. The first run was done without any tracing, the second with minimal tracing for replay, and the third with extensive tracing for Projections. Note that the tracing for replay adds about 4% to the execution time of the case when there is no tracing, while extensive tracing adds about 10%. The time for disk i/o is also about 33% less in the case of minimal replay tracing as compared to extensive tracing.

Execution time/Disk io (milliseconds)		
No tracing	Minimal tracing	Extensive tracing
10580/0	10980/600	11590/800

Table 4.1: Execution and disk i/o times for three versions of a Jacobi program.

There are some unique issues for replay in the context of Charm, because it provides high-level support for dynamic load balancing, quiescence detection, and information sharing. In the remainder of this section we examine these issues.

4.3.1 Dynamic load balancing

Many of the load balancing strategies in Charm have a spontaneous component. The strategy may periodically check the sizes of queues on the local processor, compare it to the sizes of queues on neighbors, and redistribute work. Such periodic activity cannot be replayed because of its dependence on time.

The trace recorded for replaying the program contains information about the request and subsequent creation of a task. In the case when the task is a response message to BOC or chare, the destination is fixed. However in the case of a message to create a chare, the destination has been determined through some load balancing strategy. This must be preserved in the replay. In our current solution, a sequential script looks at all the trace data, determines the destination of every new chare message, and creates a destination log file for each processor, which contains the destination of all new chares generated on that processor.

A replay load balancing strategy, shown in Figure 4.9, implements the known load redistribution. Like all other load balancing strategies it is also implemented as a BOC. Each branch of the BOC keeps a map for the local processor, which tells it where a new chare message needs to go: this information is read in from the destination log for the local processor. Whenever a new chare message is generated locally, the load balancing strategy determines its

```
BranchOffice Replay {

    entry LdbInit: (message InitMsg *msg) {
        /* initialize the branch of the strategy */
        last_map = ReadMapData(MAX_MAPS);
    }

    public NewChare_From_Local(msg) {
        int event = GetEvent(msg);
        int map = FindMap(event);
        SendMsgBranch(Replay@NewChare_From_Net, msg, map++);
        if (map > last_map) last_map = ReadMapData(MAX_MAP);
    }

    entry NewChare_From_Net: (message void *msg) {
        QsEnqMsg(msg);
    }

    public ExtractStatus(msg) { }

    public AddStatus(msg) { }
}
```

Figure 4.9: Replay dynamic load balancing strategy.

destination and sends it there. The behavior of the old load balancing strategy is therefore not replayed, only its effect is.

4.3.2 Quiescence detection

In Section 3.3, we described the implementation of the quiescence detection algorithm. The algorithm checks for local quiescence by examining the state of local queues. Therefore, a replay strategy would be able to replicate the behavior of the quiescence detection algorithm only if it replicated the state of the queues at all points of time. Replicating the state of the queues adds considerable complexity to the replay, because one must also replay the queuing and dequeuing events on each processor. Instead, there are two simpler strategies for dealing with quiescence:

1. Ignore the behavior of the old quiescence detection algorithm, and let the system detect quiescence for the replayed program.

2. Do not even detect quiescence; rather send the quiescence message, whose creation is known from the trace, when the system is quiescent.

The advantage of the first scheme is that it works without any modification to any code, and there is no trace information needed. The advantage of the second scheme is that there is no overhead of detecting quiescence. The disadvantage is that the quiescence message must be so identified and sent at the appropriate time. We have chosen the first scheme for its convenience.

4.3.3 Specifically shared variables

Specifically shared variables are implemented as BOCs and their properties are ensured using messages to communicate between different branches of the BOC. Any time-dependent behavior in the exchange of messages could pose problems in replicating the behavior of the shared variable.

In our first implementation of a monotonic variable, the values of the variable on all processors were combined by propagating values up a spanning tree on all processors; combination occurred by making each processor in the spanning tree wait for a pre-determined interval of time for other values from its children. In some of our applications, we found that monotonic variables were rarely updated, and therefore a different implementation was more efficient: broadcast the value of the variable to nearest neighbors, recursively, until everyone had the value. In this implementation, there are no time-dependent features, and therefore it can be replayed. However, the behavior of the spanning tree implementation cannot be replicated in a replay.

4.3.4 Related work

Malony [56] reasons that tracing will always cause perturbation, and has suggested methods to compensate for perturbation. His scheme involves determining the perturbation associated with every trace event. Equipped with this knowledge, one can apply the requisite compensation for every trace event in the program to obtain unperturbed performance information.

Hollingsworth et. al. [61] have developed an approach in which instrumentation for tracing is done dynamically and on demand: if information about a particular quantity is desired, the corresponding instrumentation is added by modifying the core image of the program. This approach has its merits, but for automatic performance analysis, we need all trace information before analysis can be done. In such cases, all instrumentation is necessary, and therefore dynamic instrumentation has no utility.

Recently, Hollingsworth and Miller [62], have developed an approach called the W^3 model, which attempts to reduce the amount of data traced for parallel program performance analysis by intelligently activating the trace dynamically when and where it's needed. Their model attempts to make such decisions based on low level architecture/language characteristics, such as lock-usage, semaphores, and barriers, and some generic high level characteristics, such as an object's wait-time for messages. Their approach is not complete, because the tool can provide trace reduction only for the set of problems it knows about.

4.4 Summary

In this chapter, we discussed how trace information is collected for a Charm program. We also talked about our strategies to deal with two problems in tracing, namely, asynchronous clocks and perturbation. We attempt to resolve the problem of asynchronous clocks by recreating approximate real time from the local and logical times available in the trace. We attempt to reduce the amount of perturbation due to tracing by using the replay mechanism. The replay mechanism was previously proposed as debugging support to replay asynchronous message arrival orders.

Chapter 5

Important attributes of the event graph

In Section 5.1, we describe some basic attributes of the event graph. Performance analysis of all events in the event graph can be expensive and does not provide suitable focus. In Section 5.2, we describe our choices for a set of events on which performance analysis can be conducted. In Section 5.3, we describe logical separation events. These events are used to partition the execution of a program into independent units, such that each of the units can then be analyzed separately. In Section 5.4, we discuss the utility and the nature of patterns in Charm programs. The type of analysis and the behavior of the program can be better understood if it is viewed as a composition of smaller, easily understood patterns.

5.1 Basic attributes of the event graph

Once tachyons have been eliminated in the event graph (V, E) using the algorithms in Chapter 4, the following theorem is true:

Theorem 1 (Creation) *For any event* $v \in V$, $v_c < v_s \leq v_f$.

Note that the relation between v_c and v_s is strict inequality because we assume that there is some overhead associated with creating an object, and so, in practice, the request must happen before the event is created. □

Observation 2 (Causality) *Causality guarantees that* $x \to y \Rightarrow x_s \leq y_c$.

Justification: The system must begin to process an event, before the event itself can create other events. Therefore if an event x creates an event y, the execution time of x must precede the creation time of y, i.e., $x_s \leq y_c$. □

Theorem 2 (Acyclic) *(V, E) is a directed acyclic graph.*

Proof: Assume that the graph had a cycle $x^1, ..., x^n$, such that $x^1 = x^n$ and $(\forall_{i=1}^{n-1})((x^i, x^{i+1}) \in E)$. Such a cycle is not possible because it violates Theorem 1:

$$\begin{aligned}
&(\forall_{i=1}^{n-1})((x^i, x^{i+1}) \in E) && \\
&\Rightarrow (\forall_{i=1}^{n-1})(x^i \rightarrow x^{i+1}) && (definition)\\
&\Rightarrow (\forall_{i=1}^{n-1})(x_s^i \leq x_c^{i+1}) && (causality)\\
&\Rightarrow (\forall_{i=1}^{n-1})(x_s^i \leq x_c^{i+1} < x_s^{i+1}) && (creation; v \in V, v_c < v_s)\\
&\Rightarrow (x_c^1 < x_s^1 \leq x_c^2 < x_s^2 ... x_c^n < x_s^n) && \\
&\Rightarrow x_s^1 < x_c^n && (transitivity)\\
&\Rightarrow x_s^1 < x_c^1 && (x^1 = x^n)
\end{aligned} \tag{5.1}$$

The last equation violates the *Creation Theorem.* □

Note that a directed acyclic graph doesn't have to be a tree, even though the underlying undirected graph is connected, e.g., consider the directed acyclic graph in Figure 5.1. In fact, the directed acyclic graph for Charm events is a tree, as we prove later on in this chapter.

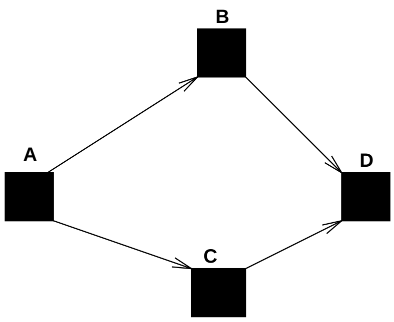

Figure 5.1: A directed acyclic graph, whose underlying graph is not a tree.

In a Charm program, there are two computations proceeding concurrently: the user and the system. System events correspond to the activities of the load balancing, the quiescence detection, and other system modules. The two computations interact very infrequently. For example, the system creates the CharmInit entry point, which initiates the user computation. Subsequently, the only user event that is ever created by a system event is the message sent by the quiescence detection module to the user program when the algorithm detects no activity in the system. The quiescence detection algorithm is activated only if the user makes a call to the *StartQuiescence* system call.

Since (V, E) includes only user computation, and the CharmInit and quiescence detection entry points are created by system events, we can treat these as events without a creator. (One would assume that if all computation — system and user — were included in V, all events would have a creator. However this is not true. The quiescence detection and some load balancing algorithm have events, which are created based on a timer value; such events have no creator.)

Observation 3 *Only the CharmInit entry point and quiescence detection entry points have no creator.*

The following definitions are used later in this thesis.

Definition 2 *An event x is said to causally precede an event y, if $x = y$, or $x \rightarrow y$, or there exist z^1,...,z^k, such that $((x \rightarrow z^1) \wedge (\forall_{i=1}^{k-1})(z^i \rightarrow z^{i+1}) \wedge (z^k \rightarrow y))$.*

Let $x \Rightarrow y$ denote the fact that x causally precedes y.

Definition 3 *We define the precedent set of an event x to be the ordered set of events $\{x^1, ..., x^k = x\}$, such that $(\forall_{i=1}^{n-1})(x^i \rightarrow x^{i+1}) \wedge \neg(\exists t)(t \rightarrow x^1)$.*

Let $x_{precede}$ denote the precedent set for x.

Definition 4 *We define the first event of a precedent set, $x_{precede}$, to be the event $y \in x_{precede}$, such that, $\forall_{v \in x_{precede}}(y \Rightarrow v)$. Let λ_x denote the first event of the precedent set for event x.*

λ_x is well defined only if the set is finite. The set is finite because:

1. There are only finitely many events in time before any event in the event-time diagram.

2. All events that causally precede an event must have happened before it in time (causality and transitivity).

3. The graph is acyclic: therefore there cannot be an infinite chain of causal predecessors of any event.

Because there are only finitely many events before an event, and no infinite chains, hence the set of events that causally precede an event must be finite.

Theorem 3 *The first event in the precedent set for event x, occurs earlier (in time) than all other events in the precedent set, i.e.,* $\forall_{v \in x_{precede}}(\lambda_{x_s} \leq v_s)$.

Proof: Since $\forall_{v \in x_{precede}}(\lambda_x \Rightarrow v)$, the proof follows from Theorem 1 and transitivity. □

Theorem 4 *For any precedent set, $x_{precede}$, its first event λ_x can have no creator.*

Proof: The proof is by contradiction. Support the first event did have a creator. Then by the definition of a precedent set, the creator, say v, must be in the precedent set. From causality, it follows that $v_s \leq \lambda_{x_c}$. And from the *Creation Theorem*, it follows that $\lambda_{x_c} \leq \lambda_{x_s}$. Therefore, by transitivity, $v_s \leq \lambda_{x_s}$. From Theorem 3, it follows that the first event occurs earlier than all other events. Since the creator of the first event must occur before it, so the creator must occur before all events in the precedent set. Hence the creator is the first event of the precedent set. This contradicts the assumption that λ_x is the first event. □

Theorem 5 *The first event of any precedent set is either the* CharmInit *or a quiescence detection entry point.*

Proof: The proof follows from Observation 3 and Theorem 4. □

Definition 5 *We define the antecedent set of an event x to be the set of events* $\{x = x^1, x^2, ..., x^n\}$, *such that* $(\forall_{i=2}^{n})(x \Rightarrow x^i)$.

Let $x_{antecede}$ denote the antecedent set for x.

Definition 6 *We define the last event of an antecedent set, $x_{antecede}$, to be the event $y \in x_{antecede}$, such that, $\forall_{v \in x_{antecede}}(y_f \geq v_f)$. Let μ_x denote the last event of the antecedent set for event x.*

Theorem 6 *Let, x^1 be the CharmInit and $x^2, ..., x^k$ be the quiescence detection entry points. The underlying graph defined by the antecedent set of x^i is a tree.*

Proof: A connected graph with n nodes is a tree if it has $n-1$ edges. Since $x^i \Rightarrow v$, where $v \in x^i_{antecede}$, so if one ignores the direction of the arcs in the underlying graph, for any $v \in x^i_{antecede}$, there is a path from v to x^i, and vice versa. The underlying graph is then connected by transitivity: since there is a path from v_1 to x^i and a path from x^i to v_2, where $v_1, v_2 \in x^i_{antecede}$, then there is a path from v_1 to v_2 through x^i. Let the cardinality of $x^i_{antecede}$ be n. Then, because every event in $x^i_{antecede}$, except x^i has exactly one creator (Observations 1 and 3), there are $n-1$ edges. Therefore, the underlying graph for the antecedent set is a tree. $\square$

Theorem 7 *Let, x^1 be the CharmInit and $x^2, ..., x^k$ be the quiescence detection entry points. Then, $V = x^1_{antecede} \cup x^2_{antecede} ... \cup x^k_{antecede}$.*

Proof: Obviously, every event in the sets on the right hand side is in V. It remains to show the converse. By Theorem 5, if $v \in V$, then $\lambda_v \in \{x^1, ..., x^k\}$. Since λ_v is in the precedent set of v, therefore by definition, $\lambda_v \Rightarrow v$. By the definition of the antecedent set, $v \in \lambda_{v_{antecede}}$. So every event in V is in the antecedent set of one of $x^1, ..., x^k$. So $V = x^1_{antecede} \cup x^2_{antecede}, ..., \cup x^k_{antecede}$. $\square$

Theorem 8 *Let, x^1 be the CharmInit and $x^2, ..., x^k$ be the quiescence detection entry points. Then, $x^i_{antecede} \cup x^j_{antecede} = \Phi$.*

Proof: The proof is by contradiction. Let there exist i, j such that $y \in x^i_{antecede}$ and $y \in x^j_{antecede}$. Then, by definition, $x^i \Rightarrow y$ and $x^j \Rightarrow y$. But this implies that the event corresponding to y had two messages that created it. This is not possible since the underlying graph of a Charm event diagram is a tree. Thus, it is not possible for any i, j that any event lie in both $x^i_{antecede}$ and $x^j_{antecede}$ $\square$

Theorems 7 and 8 tell us that the event diagram of a Charm program is a union of disjoint graphs.

Definition 7 *If, x^1 is the CharmInit entry point, and $x^2, ..., x^k$ are the quiescence detection entry points, we define the antecedent set $x^i_{antecede}$ as the i^{th} disjoint phase.*

5.2 Critical path

Performance analysis of all the events in the event graph can be very time consuming. Analysis can be made more efficient and focused by choosing only a subset of events which have a significant impact on the performance of the program. One choice for such a set is the critical path in the program. The critical path in a program, as defined by Yang and Miller [71], is the longest computational chain in the event graph of the program (an example appears in Figure 5.2(a)). In this definition, communication latencies and scheduling delays are ignored; only computational times and dependences between processes are taken into account. Since such a set ignores parallel components of program behavior, such as communication latencies, it does not adequately represent the performance of the parallel program. However, critical path analysis is still useful in the detection of sequential performance problems in the program. A critical path also provides a lower bound on the completion time of a parallel program with an infinite number of processors.

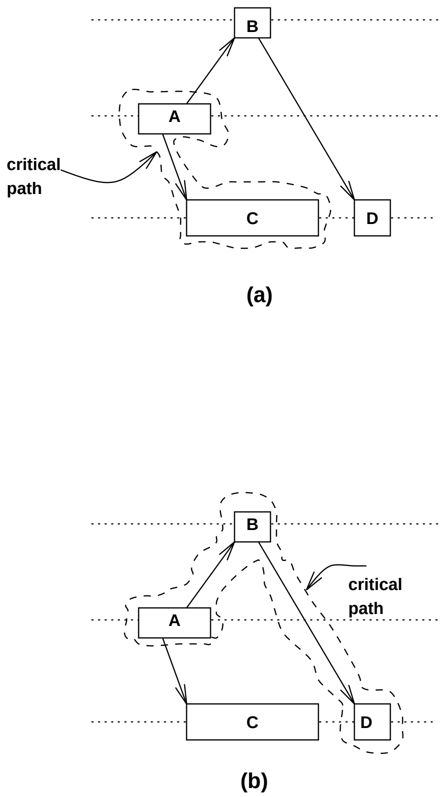

Figure 5.2: Alternate definitions of critical path

An alternate definition of the critical path has also included communication latencies in the computation of the longest chain in the event graph (an example appears in Figure 5.2(b))[1]. Such a critical path models information about parallel program behavior better than the simple computational time based critical path. In addition, if all delays, such as waiting for a message to arrive and delays in scheduling of messages, are accounted for in the computation of the critical path, the critical path will include the event that finished last in the execution. However, if delays are not accounted for in the computation, the critical path may not include the last event in the program. We consider the last event to be an important event in the program because it directly represents the turnaround time of the program — any improvements in its execution time would translate into improvement of the turnaround time of the program. Therefore a definition of the critical path which include communication latencies and scheduling delays provides a good model of parallel program behavior. Further, since this definition includes communication latencies and scheduling delays, it provides a very useful component of the program behavior on which automatic analysis can focus.

We use an extension of this definition of a critical path to determine the the set of events, called the *last event chain*, on which performance analysis techniques need to focus. An extension of this definition is needed for Charm programs because the presence of quiescence detection events cause the corresponding event graph of a Charm program to be split into unconnected components (Theorems 7 and 8). In such a graph, the notion of a longest chain in the graph is not sufficient, because the events in the longest chain will lie in only one of the connected. In the following theorem, we prove that the disjoint phases are also disjoint in time. This theorem makes it possible to extend the second definition of critical path.

Theorem 9 *Let, x^1 be the CharmInit and $x^2, ..., x^k$ be the quiescence detection entry points. Further, let, $x^1_s \leq x^2_s ... \leq x^k_s$. Then, the trees defined by $x^1_{antecede}, ..., x^k_{antecede}$ are all disjoint in time, i.e., for $1 \leq i \leq k$, $(\forall_{t \in x^i_{antecede}})(\forall_{v \in x^{i+1}_{antecede}})(t_f \leq v_s)$.*

Proof: Since $x^2, ..., x^k$ are quiescence detection entry points, choosing two consecutive events from $x^1, ..., x^k$ implies that at least one of them is a quiescence event. Since only the first event is CharmInit, so irrespective of what x^i is, x^{i+1} must be a quiescence detection event. A

[1] Note that this definition of critical path is equivalent to starting from the last event in the computation and tracing back through its creators to the beginning of the program.

quiescence detection event, by definition, occurs only when there is no activity in the system. Therefore all activity initiated by the event x^i, which occurs before x^{i+1} must have terminated before x^{i+1}. And since, x^{i+1} precedes all events in its antecedent set, therefore all events in the antecedent set of x^i will occur before all events in the antecedent set of x^{i+1}. $\Box$

Since each disjoint phase is disjoint not only in terms of events that constitute the phase but also in terms of time, therefore, we can define a last event chain for each phase as follows:

Definition 8 *We define the last event chain of the i^{th} disjoint phase as the precedent chain of the last event of the i^{th} phase.*

Note that this definition of the last event chain corresponds to the second definition of critical path in the case when the graph has only one component.

The last event chains of each phase is then independent and disjoint. Hence the last event chain of the computation can be defined as follows:

Definition 9 *The last event chain for the computation is the union of the last event chains for each disjoint phase of the event graph of a Charm program.*

For example, in Figure 5.3, the last event chain for the computation is a union of the two disjoint chains $g - f - e$ and $c - b - a$.

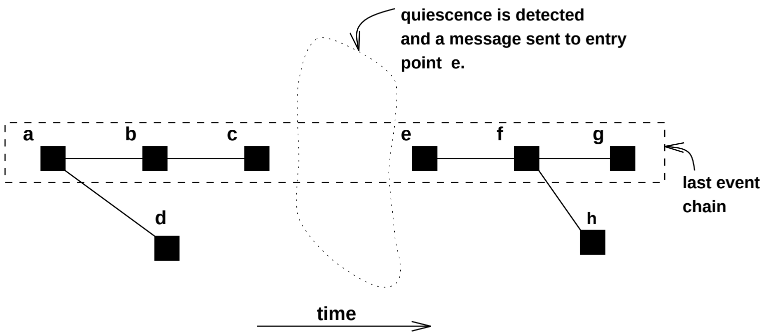

Figure 5.3: Last event chain.

5.3 Logical separation events

One critical issue that needs to be examined is the range of time over which a program's trace should be analyzed. For example, in order to compare the loads on many processors, one would need to identify the time from which to start counting events and the time at which to finish counting events on all the processors. There are many possibilities. The analysis could be carried over the duration of the entire program or over fixed periodic intervals of time (the interval can be user-specified or can be some heuristically chosen value). However, a program often goes through different natural phases in its execution, each one different from the other. For example, in an iterative solver each iteration can be thought of as a phase. In such a situation, the impact of different performance criterion may be different on different iterations, and hence the analysis could be different for each iteration. Therefore an analysis carried out over the entire program or any pre-defined interval may provide incorrect feedback, because they may not coincide with the phases in execution. The best ranges of time would be those which correspond naturally to different phases of computation in the user program.

How does one go about defining and determining such phases? Our objective that each phase be separately analyzable leads to the definition of the boundary between two phases as a point in time such that the relative times of the events before that point in time have no impact on the performance (i.e., the relative timings) of the events after it. We denote the events which separate a program into such natural phases as logical separation points. In the remainder of this section, we motivate and provide a formal definition of logical separation points.

Empirically speaking, parallel programs are naturally repeating. One type of events that naturally separate the execution of a program into phases, are those at which the program goes through a global synchronization. At such an event, all activity in the program ceases, and activity is subsequently resumed from that event. Since the user execution begins at the CharmInit entry point, and all subsequent activity is initiated there, we make the following observation:

Observation 4 *The CharmInit entry point is always a logical separation point.*

When the user requests for quiescence to be detected in a program, a quiescence detection algorithm monitors the state of computation, and sends a message to a pre-specified entry point when there is no possibility of further activity in the system. A quiescence detection event therefore splits the execution of the program into time-disjoint phases. Hence the following observation:

Observation 5 *An event that is the result of a quiescence detection message is a logical separation point.*

We have already proved that the antecedent sets for CharmInit and quiescence detection entry points are disjoint, acyclic directed graphs. Consider any one of these antecedent sets. Are there other events in the antecedent set which divide the program into its natural phases?

By our definition, any event such that the performance of events before it has no impact on the performance or timings of the events after it should also be logical separation point. A natural way of viewing such points is as an articulation point in a graph. In a connected graph, a node is defined to be an *articulation point* if the removal of the node and its incident edges causes the graph to be split into multiple connected components. Figure 5.4 shows the articulations points in two different graphs. In the top graph, point c is an articulation point, while in the bottom graph, points a, b and d are articulation points.

If you consider the graphs to be event diagrams, where each node represents an event and the edges (in the forward time direction) represent creation of an event, then the articulation point in the top graph separates two phases of the event diagram: one consisting of events a, b, and d, and the other consisting of events e, f, and g. Notice that the first set of events does not affect the relative performance of the second set of events.

However as the bottom figure shows, an articulation point need not necessarily split the graph into phases of program execution. The second graph could be thought of as the event diagram of a tree-structured computation; thus, even though each interior node of such an event diagram would be an articulation point (in this case b and d), neither of the articulation points separates the events in the diagram into distinct phases.

One key factor seems to be the separation of phases in time, so we must include temporal conditions for an event to be a logical separation event. From the intuitive notion of logical separation points as global synchronization events, an event can be a logical separation point if there is no other event which can occur concurrently with it. This would intuitively define a logical separation point. With this motivation, we define a logical separation point as follows.

Definition 10 *An event, x, is a* logical separation point *if it satisfies the following conditions:*

1. *There are no events which occur concurrently (in real-time) with it:*

$$(\neg\exists t)(((t_s \leq x_f) \wedge (t_f \geq x_s)) \wedge \neg(x \rightarrow t)) \tag{5.2}$$

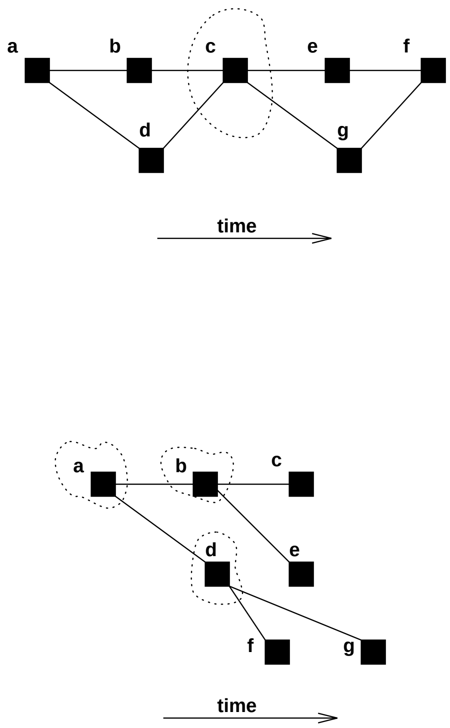

Figure 5.4: Event c is an articulation point in the top figure, and events a, b, and d are each articulation points in the bottom figure.

2. *There is no event that crosses over x, i.e., there is no event (excluding ones created by x), which is created before the completion of x, and is processed after it:*

$$(\neg \exists t)((t_c \leq x_f) \wedge (t_s \geq x_f) \wedge \neg(x \rightarrow t)) \tag{5.3}$$

These conditions determine all logical separations points[2]; however some of them are not necessary for analysis. Consider the event diagram in Figure 5.5: a chain of events constitutes a sequence of logical separation points. For the purpose of partitioning the execution of the program into phases, choosing each one as a logical separation point means that the phases in between consist of no events. The amount of analysis that needs to be done can be simplified by collapsing the chain of logical separation points into either the first or the last event as the

[2] Since the conditions used in determining logical separation points are temporal, it is possible that a different trace ordering would produce a different set of logical separation points? With the current definition, this is indeed possible. If all synchronization information about the program was available, a better definition is possible. In such a definition, a logical separation point would be an event which logically succeeds every event before it in time, and logically precedes all events after it in time.

only logical separation point: we have arbitrarily chosen the last event of the chain as the only logical separation point. Hence the following observation:

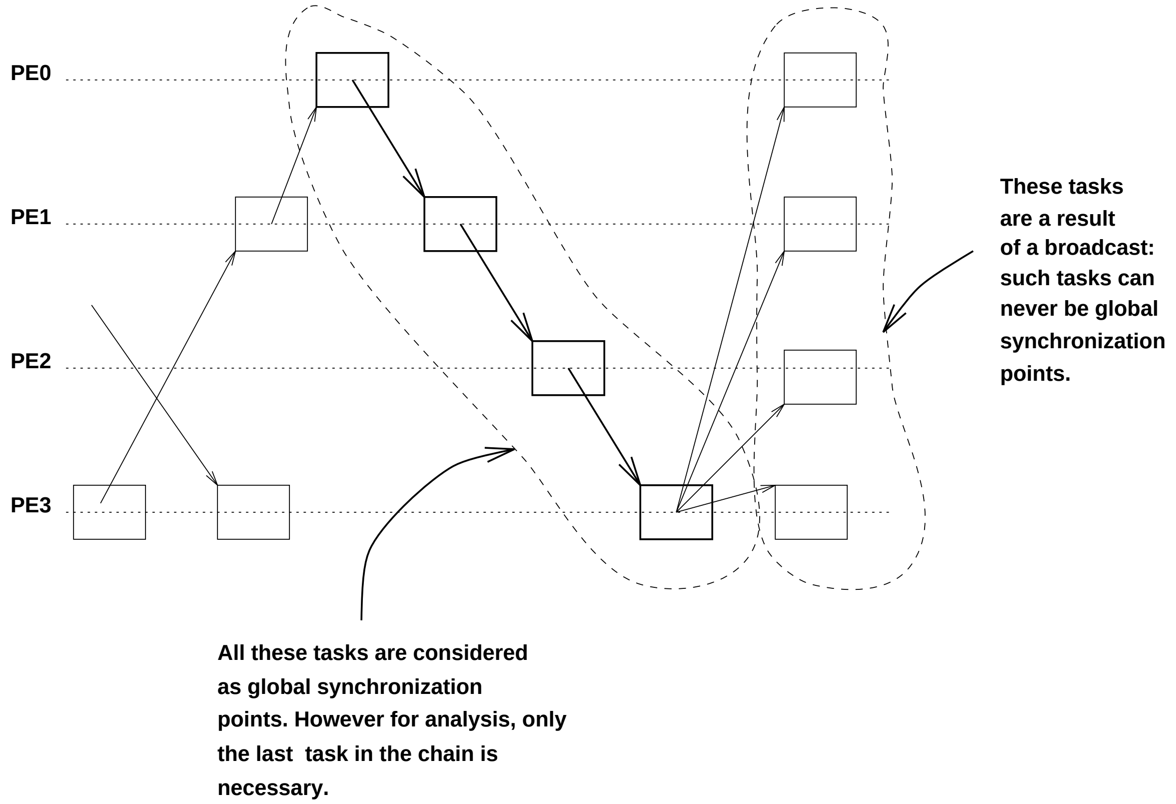

Figure 5.5: A chain of logical separation points and some events which can never be logical separation events.

Observation 6 *In a chain of logical separation events, the (logically) last event is sufficient.*

There is one note we should make to the above observation. Our initial definition attempted to collapse chains in order to eliminate phases with zero events. Collapsing long chains of logical separation points however causes another problem, which is collecting too many logical separation points into a pre-existing phase: this could potentially affect the analysis. Therefore, for long chains of logical separation events, both the first and the last need to be considered as logical separation points.

Once the logical separation points of an execution are determined, the program's execution can be partitioned into phases: each phase is the set of events that occurs between consecutive logical separation points. We call each such phase *logically independent*. A logically independent phase is a period in the execution of the program, which is independent from the rest of the execution, except for an event which triggers off all the events in the period under consideration. In an iterative solver with global reduction, for example, each iteration would be a logically independent phase.

Now we must develop an algorithm to determine logical separation points and logically independent phases. The most straightforward $O(n^2)$ and $\Omega(n^2)$ algorithm is one in which the algorithm checks every pair of events to see whether they could have occurred concurrently or crossed over. An optimization which reduces the best case complexity would be to traverse the events in the order in which they occurred. The first algorithm does just that.

5.3.1 First algorithm to determine logical separation points

Figure 5.6 shows the first algorithm to determine the logical separation events. The algorithm determines the set of events, which satisfy the two equations 5.2 and 5.3. The logical functions, $NoConcurrentEvents$ and $HasPredecessor$, implement the two equations, respectively. An event that has been created can be in one of three states:

1. *Passive*: An event that has been created already is first designated as passive.
2. *Ready*: A passive event becomes ready if it is the next event (as defined in the event traces) to be processed on a particular processor.
3. *Active*: An event becomes active if it is the earliest event amongst the ready ones.

The lists $created$, $ready$, and *being_processed* maintain the list of passive, ready, and active events in the program. The algorithm starts by inserting the earliest event from each processor into the $ready$ list; all events created before the ready event are inserted into the $created$ list. In every step, the least event from the $ready$ list is inserted into the *being_processed* list. An event could cross over the current event only if it exists in either the $created$ or the $ready$ list at this time. Similarly an event could occur concurrently with the current event only if it exists in the *being_processed_list* list. If no event is determined in any of these lists which either crosses over or which occurs concurrently with the current event, the current event is designated a logical separation event, and the algorithm proceeds to the next event on the $ready$ list.

This algorithm has a worst case complexity of $O(n^2)$, where n is the number of events. The algorithm looks at each event to see if it satisfies the Equations 5.2 and 5.3: this involves checking

```
TASK *task, *min_task;
TASK_LIST *created, *ready, *being_processed, *sync_list;

SkipToProcessing(TASK *task) {
    while (!IsProcessing(task)) {
        if (IsCreation(task)) AddList(created, task);
        task = task->next;
    }
    AddList(ready, task);
}

UpdateBeingProcessedList() {
    for (task=being_processed; task; task=task→next)
        if (EndProcessingTime(task) < BeginProcessingTime(min_task))
            DeleteList(being_processed, task);
}

DetermineSyncPoints() {
    created=ready=being_processed=sync_list=NULL;
    for (i=0; i<maxpe; i++) SkipToProcessing(transaction_list[i].head);
    min_task = LeastList(ready);
    while (min_task) {
        AddList(being_processed, min_task);
        DeleteList(ready, min_task);
        if (!NoConcurrentEvents(being_processed, min_task) &&
            !HasPredecessor(created, min_task))
            /* min_task is a global synchronization point */
                AddList(sync_list, min_task);
        SkipToProcessing(min_task->next);
        min_task = LeastList(ready);
        UpdateBeingProcessedList();
    }
}
```

Figure 5.6: Algorithm to determine logical separation events in an execution.

to see if the lists *created* and *ready* are empty, and the only event in the *being_processed* list is the current minimum event. These operations can be done in constant time. However, in order to determine the next ready event one needs to look at all the events in the *created* list. In the worst case, when all the events are created at the beginning and on one processor, this part of the computation can take $O(n)$ steps, hence the worst case complexity is $O(n^2)$. However, on an average the algorithm does much better than $O(n^2)$, because the *created* list does not have all events in it.

One observation allows us to reduce the number of times the two checks for logical separation need to be done:

Theorem 10 *Every logical separation event must lie on the last event chain.*

Proof: The proof follows by contradiction. Let v be a logical separation point that does not lie on the chain, and $\{p^1, ..., p^k\}$ constitute the last event chain for the disjoint phase in which v lies. Since p^k is the last event for the disjoint phase, therefore $v_f \leq p^k_f$. Further, since the last event chain is also a precedent chain by definition, therefore, it follows from Theorem 5 that p^1 is either the CharmInit or a quiescence detection entry point. Since v belongs to a disjoint phase defined by p^1, therefore, $p^1 \Rightarrow v$, which implies that $p^1_s \leq v_c$. From definition, $v_c < v_s \leq v_f$. Combining the three equations ($p^1_s \leq v_c$, $v_c < v_s \leq v_f$, and $v_f \leq p^k_f$), we arrive at the following equation:

$$p^1_s < v_s \leq v_f \leq p^k_f \tag{5.4}$$

Because $p^1 ... p^k$ is the precedent chain for p^k, therefore, $p^1 \rightarrow p^2 ... \rightarrow p^k$, which implies that:

$$p^1_s \leq p^2_s ... \leq p^k_s \tag{5.5}$$

Now one of the following must be true:

a. $(\exists i, j)(p^i_s \leq v_s \leq p^j_s \leq v_f)$

b. $(\exists i)(p^i_s \leq v_s \leq v_f \leq p^{i+1}_s)$

The first condition (a) violates condition (1) for a logical separation point that no other event occur concurrently with v. The second condition (b) violates condition (2) for a logical separation point that no events cross over. Since one of these conditions is true, it contradicts the fact that v be a logical separation point. Therefore, by contradiction, all logical separation points must lie on the last event chain. □

Intuitively, a logical separation point provides the only connection between the set of events before and the set of events after it (see Figure 5.7). Therefore, if the last event chain crosses from an events after a logical separation point to an event before it, the only way it can do

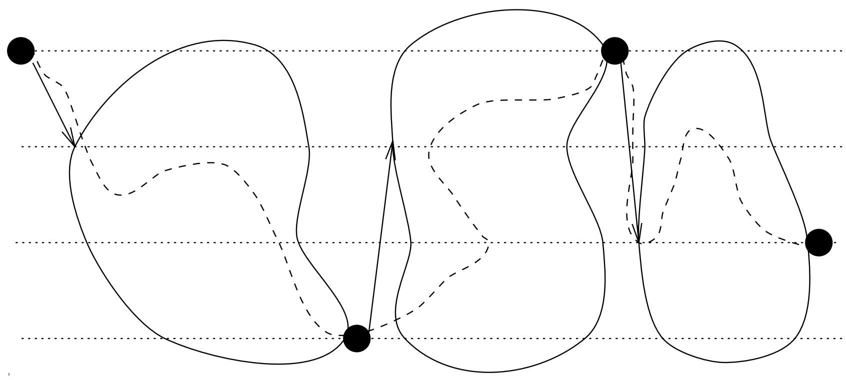

Figure 5.7: The last event chain and the logical separation points.

is by passing through the logical separation point. This result allows us to prune the search for a logical separation points to those events that lie on the last event chain. Except in the unusual case when the computation is a chain, the last event chain would have considerably fewer events than there are in the system. Therefore, in the average case, the complexity of the program would be much better.

5.3.2 Second algorithm to determine logical separation points

We have also designed and implemented a second algorithm which does not try and verify the two conditions for every event. Instead it attempts to determine the number of "active" events at a particular time. If this number is 1, then it indicates that there is only one event in the system at that point in time. This indicates that the event is a logical separation point because it satisfies the two conditions: no concurrently occurring event and no event that crosses over.

Figure 5.8 shows the second algorithm to determine logical separation events. The algorithm maintains a heap of the earliest event ready to be processed on each processor, and the count *active* of the number of active messages. A message is potentially active if it is created and not processed or if it is being processed in the system. Whenever a creation event is encountered, the count of active messages is incremented, and whenever an event is completed, the count is decremented. When the count is 1 at the time the system begins processing an event, it means that there is only one event that could possibly be active at that time, inclusive of the current event. Thus, the current event must be a logical separation point.

The complexity of the algorithm is $O(n * log(p))$, where n is the number of events and p the number of processors. Each event is examined only once, and for each event, it is added

```
TASK *task, *min_task;
TASK_LIST *sync_list;

SkipToProcessing(TASK *task) {
    while (!IsProcessing(task)) task = task->next;
}

DetermineSyncPoints() {
    for (i=0; i<maxpe; i++) SkipToProcessing(transaction_list[i].head);
    min_task = LeastList(ready);
    while (min_task) {
        switch(min_task→type) {
        case CREATION:
            actual++;
            break;
        case END_PROCESSING:
            actual- -;
            break;
        case BEGIN_PROCESSING:
            if (actual==1) AddToSyncList(min_task);
            break;
        }
        SkipToProcessing(min_task->next);
        min_task = LeastList(ready);
    }
}
```

Figure 5.8: Second algorithm to determine logical separation events in an execution.

and deleted from a sorted heap of p events. Therefore, there are $O(log(p))$ operations for every event, hence the complexity is $O(n * log(p))$.

Why are logical separation points and logically independent phases a good idea? Logical separation points demarcate the execution of the program into phases which are logically independent. Therefore, performance analysis of each logically independent phase is independent of that for any other phase. Further, any steps taken to tune performance of one will not worsen the performance of other phases. Another important feature of logical separation points is that they identify a "band" of time in which no activity crosses over. As a result, they can be relatively accurately identified even with approximate real time reconstruction. A possible generalization of a logical separation point that may be useful is a set of k events, at most one from each processor, such that no other event crosses over them.

5.4 Patterns of communication

A parallel computation, based on a message passing model, is at the lowest level a bunch of processes communicating through messages. Often the messages form easily recognizable patterns, such as those of a reduction or a broadcast. There are two different ways in which it would be useful to know the nature of message passing patterns in a program:

1. It can help us detect potential problem patterns in a user program, e.g., a bottleneck event.

2. Different considerations are necessary in terms of performance analysis for different patterns. For example, in a spanning tree reduction, the balance in the computation across processors is not as important as the depth of the spanning tree and its branching factor. Conversely, in a jacobi iterative solver, where processors exchange messages with each other, the balance in the computation involved in the exchange is very important.

There are a large number of message passing patterns possible. We have identified the following patterns of message passing to be of interest to us (they are discussed in detail later on):

1. *Bottleneck*: If all processors send messages to one processor, it can become a bottleneck if it is a server processor or if a global synchronization occurs on it.

2. *Broadcast*: One processor broadcasts a message to all other processors, possibly including itself.

3. *Exchange*: Each processor sends (and therefore by symmetry receives) messages to (from) its neighbors.

4. *Reduction*: All processors contribute a value which is combined to produce a global value; often the value may be zero, in which case the reduction is a synchronization operation. A reduction may be implemented by a spanning tree.

5. *Cyclic*: Each processor sends out a message in a ring through all or some of the other processors.

6. *Chain*: Each processor sends out a message in a chain through some or all other processors. It differs from a cyclic pattern, because the message does not return back to the processor from which it originated.

7. *Sequential chain*: One processor sends out a message, which goes in a chain through all processors.

The patterns described above are quite complex. However, they can be constituted from the six basic conceptual patterns, shown in Figure 5.9. The circle in the figure represents an entry point and the arrows represent messages. An entry point can receive one or multiple messages, as a result of which it may send out none, one, or multiple messages. We assume that an entry point does not spontaneously generate activity.

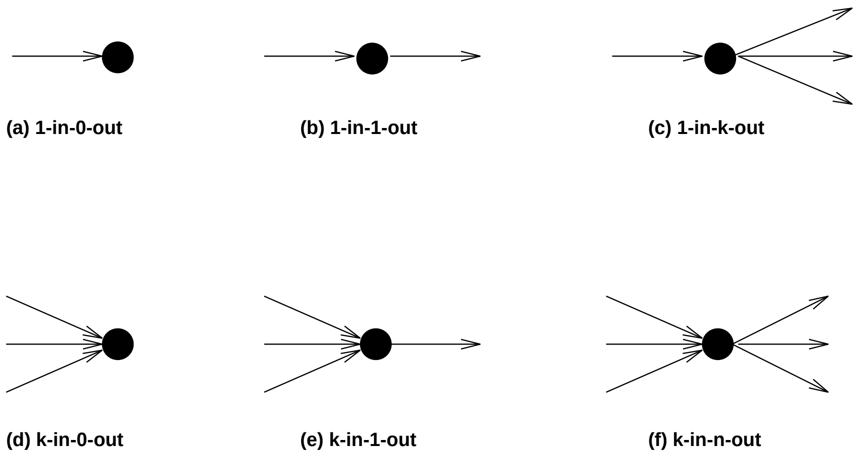

Figure 5.9: Building blocks of message passing patterns.

Consider the example of a spanning tree reduction in Figure 5.10(A). Each process receives values from children in a spanning tree, which it combines and sends up to its parent. The messages arriving in and leaving out of each processor can be characterized as a k-in-1-out pattern, where k is 2.

Consider another example of a Jacobi iterative solver. In each iteration, each processor gets a broadcast message. Upon receipt of the broadcast message, a processor sends out its local value to its four (processors on the edge of the mesh send out to two or three neighbors)

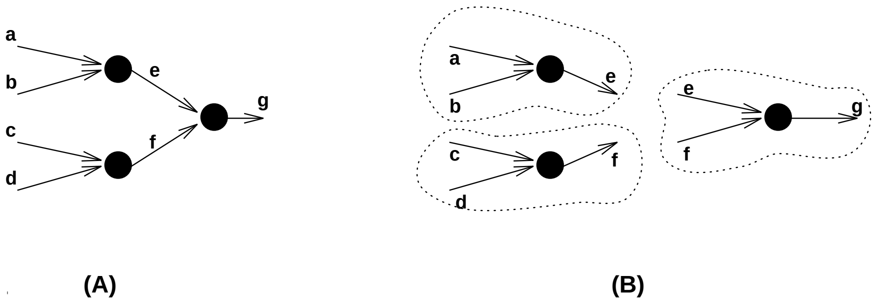

Figure 5.10: Partition of message patterns for a spanning tree reduction.

neighbors. Subsequently each processor receives messages from its neighbors, which it uses to recompute its value, and then it sends out the value for a global reduction. Figure 5.11 shows how the computation in a Jacobi iterative solver can be composed from a 1-in-k-out and a k-in-1-out pattern. The first block represents the receipt of the broadcast and the sends to the neighbors, while the second block represents the receipt of values from neighbors and the participation in a reduction.

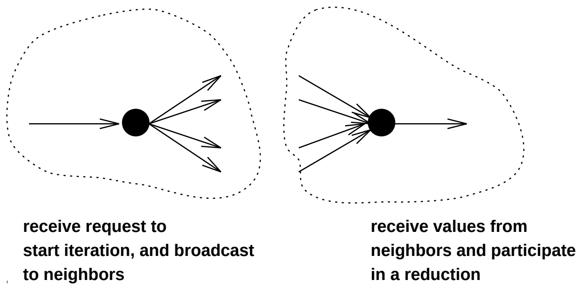

Figure 5.11: Partition of message patterns in a process of a jacobi iterative solver into combinations of possible patterns.

Let us try and relate the patterns in Figure 5.9 to events in Charm programs. One immediate problem is the fact that entry points cannot *simultaneously* receive multiple messages. So a k-in-1-out or a k-in-n-out pattern would look very different in the event-time diagram of a Charm program's execution. Figure 5.12 shows how these two patterns look for Charm programs.

In event-diagrams for Charm programs, there can be no k-in-n-out patterns, where $k > 1$ and $n \geq 0$. In fact, there are only three basic building blocks for patterns in the event-time diagrams for Charm programs; these are the first three, a, b, and c in Figure 5.9. One must now be able to compose these basic patterns into one of the six conceptual patterns, and subsequently into more complex message passing patterns.

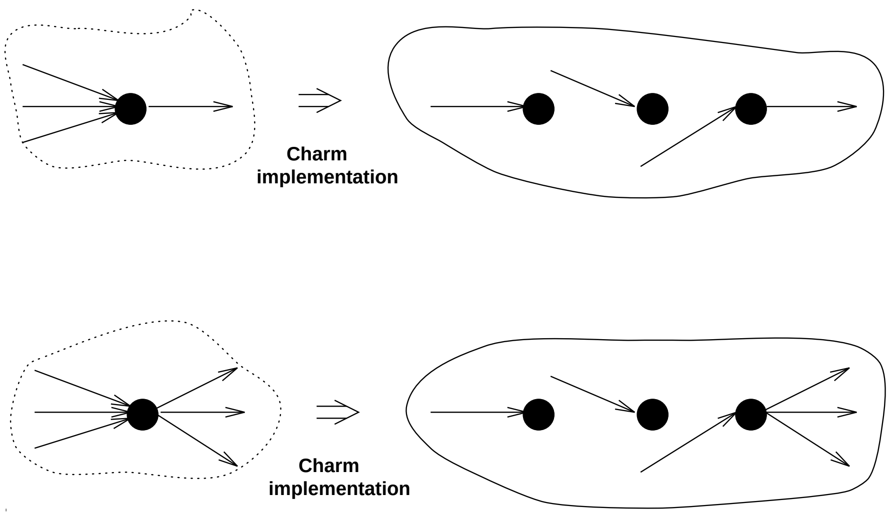

Figure 5.12: Patterns of message passing in a Charm implementation of the spanning tree reduction.

In order to use the three basic building blocks in Charm to understand the more complex message passing patterns, we first observe that a logical separation event precedes all events in the corresponding logically independent phase. Therefore, we can isolate our examination of how message passing patterns are composed to each logically independent phase separately. We also observe the following characteristics of an entry point:

1. An entry point is a code-block, which performs some specific function. Often, the code in an entry point (even on different processors) performs the same set of operations.

2. All instances of an entry point are triggered by a message of the same type. Further, it is likely that all messages are created from entry points which are of the same type.

As a consequence of these observations, it is likely that the entry points and the messages that constitute a pattern, are of the same type. For example, all the interior events in a spanning tree reduction are likely to correspond to the same entry point. Of course, more complex patterns could be generated with a combination of multiple message types and multiple entry point types. However even looking at patterns formed by a single entry point provides us with a vast amount of information about program behavior.

We describe the scheme with which patterns are determined for events; the scheme is valid for branch-office chares. It will work for chares with some modifications. We begin with the following definition:

Definition 11 *The logical depth of an event t is the number of events in the ordered set $t_{precede}$ which correspond to the same entry point as t.*

Figure 5.13 shows the logical depth of the events in the execution of a spanning tree on 6 processors.

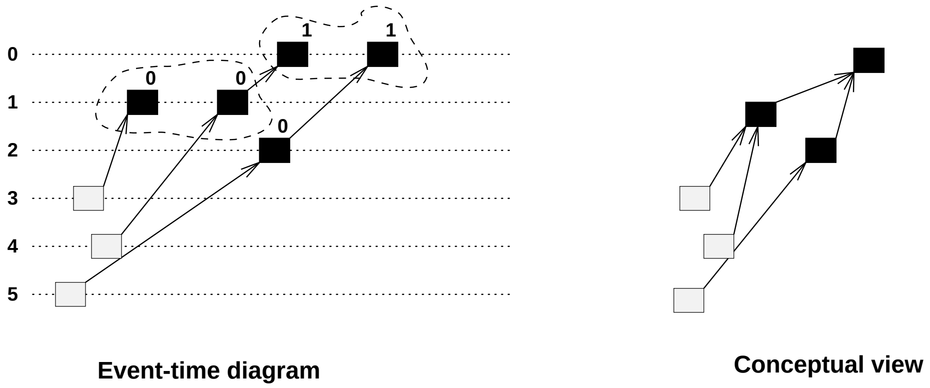

Figure 5.13: Logical depth of events in a spanning tree reduction.

Intuitively, we attempt to determine the logical order in which events corresponding to an entry point occur. Once the logical order has been established, one can see "phases" of an event occurring on different processors: in the first phase the events at logical depth 0 are executed, in the next the events at logical depth 1 are executed, and so on.

Figure 5.14 shows the algorithm to determine logical depth for each entry point inside a logically independent phase. The basic algorithm is similar to a depth first search, except that the depth assigned to a node in the graph is not the depth in the depth first search tree, but the logical depth with respect to an entry point.

Once the logical depth for the events corresponding to each entry point has been determined, one creates the logical matrix for each entry point. The logical matrix, M_e, for an entry point e is constructed as follows:

- The matrix is of dimension pXk, where p is the number of processors and k is one more than the maximum logical depth of any event corresponding to e.
- $M_e[i][j] = -1$, if there is no event corresponding to e with logical depth j on processor i.
- $M_e[i][j] = (l_1, ..., l_n)$, if $l_1, ..., l_n$ are processors which created the events with logical depth j on processor i.

```
DFS(TASK *v, int ep, int depth) {
    TASK *w;
    if (v → ep == ep) v → depth = depth + 1;
    for (w ∈ v → children)
        DFS(w, ep, depth+1);
}

LogicalOrdering(TASK *separator) {
    for (ep ∈ UserEntryPoints)
        DFS(separator, ep, 0);
}
```

Figure 5.14: Algorithm to determine logical depth of entry points.

Once the logical matrix, M_e, has been computed, one can determine different patterns by examining its entries. As an example consider the logical matrix in Table 5.1 for the spanning tree shown in Figure 5.13. Given a logical matrix M_e, how does one tell that it corresponds to a spanning tree? Here are the signs for which one looks:

Processor	Logical depth	
	0	1
0	-1	(1, 2)
1	(3, 4)	-1
2	5	-1
3	-1	-1
4	-1	-1
5	-1	-1

Table 5.1: Logical matrix for a spanning tree reduction on 6 processors.

1. Each processor creates at most one event of type e.

2. Only one processor creates no event of type e.

3. The events of type e at the greatest logical depth occur on only one processor, and it is the same processor which creates no other events of type e.

In Chapter 6, we discuss how logical matrices for other patterns look, and how they can be detected.

5.4.1 Related work

Cuny and Hough [68, 69, 70] have used patterns to provide a better visual understanding of the order of occurrence of nonatomic and concurrent events. Their work is in the context of a parallel debugger called Belvedere. Their approach consists of reordering events into logically equivalent event graphs, in which the patterns of message passing are obvious. In most cases, reordering can be achieved by determining the connected components of the event graph, and then displaying the graph so that all components at the same level start at the same time. However in some cases, reordering is not possible because of circular dependences. In such cases, the user can construct partially consistent views, which they call *perspective views*. The principal focus of their approach is to provide a visual picture of the execution of the program, which makes it easy to identify anomalies (and hence bugs) in message passing patterns. Our approach aims at detecting the pattern itself so that it can be used to better analyze the performance of a parallel program.

Islam [72] has classified application using patterns of synchronization and communication. Information about the nature and pattern of synchronization and communication in a user program, can be used to design a better message passing protocol specifically for that program, as compared to the generic one available.

Chapter 6

Integrated automatic performance analysis

In Chapter 3, we described the type of information needed about a Charm program's behavioral characteristics and the methods used for acquiring it. In Chapter 5, we defined various attributes of the event graph and suggested algorithms to determine them. In this chapter, we describe the framework in which the event graph, which represents information about a Charm program's behavioral characteristics, can be analyzed automatically for performance problems.

Some of the techniques we have developed in automatic analysis are well known in performance analysis literature. However there is a difference in the foci of our analysis and previous work. We examine this difference in Section 6.1. In Section 6.2, we introduce the integrated framework for analysis and motivate some performance analysis techniques that become necessary. These techniques and others are described in in Section 6.4, along with methods to evaluate the severity of a performance problem. In Section 6.5, we describe how the application of the analysis can be optimized by using the last event chain. In Section 6.6, we describe how different analyses are combined.

We introduce some notation that will be needed in this chapter. Let,

m^e denote the message type corresponding to entry point e,

$e_1, ..., e_n$ be the entry points in the Charm program,

P be the number of processors, and

α and β be the latency parameters for message transfer.

6.1 Nature of analysis

In previous work, the focal point of analysis has been either the processor or a message treated as a single generic type. For example, performance data is displayed in terms of number of messages sent and received by a processor, or the utilization on a processor. Two performance analysis tools that provide some degree of program specific feedback are:

1. *Gprof* [73] was one of the first tools to provide call-graph profiling. It measures time spent in executing each procedure, and the fractions of that time spent in executing other procedures called from it. However, call-graph profiling does not provide any information about messages and their association with procedures. Since Gprof was never meant for performance analysis of parallel programs, this is reasonable to expect.

2. *Upshot* [63] allows one to view a timeline of user defined events that occurred on each processor. However, Upshot does not provide any specific performance information about message types in the program.

In our approach, the focus of the analysis is sharper: our analysis is done with respect to an entry point or a specific message type. We examine this contrast in more detail in this section.

At the lowest level, a Charm program consists of message and the execution of entry points. Because the execution model of Charm is message driven there is a clear association between work (execution of an entry point) and the type of message that caused it. Further, the different types of work and messages, and their association are all statically determined, because:

1. A message can trigger the execution of only the defined entry points in the program; therefore all work must correspond to one of the defined entry points.

2. The *CkAllocMsg* call used to allocate Charm messages, allocates only messages of types defined in the program; therefore a message must correspond to one of the defined message types.

3. An entry point receives a message whose type is statically determined; therefore the association between an entry point type and a message type is statically determined.

In the SPMD model, a single process executes continuously on each processor, and sends and receives messages periodically. There is no direct association between work and messages. One can associate with a message the computation that immediately follows its receipt and until the receipt of the next message. However, in general, such a partition of the user computation will not separate the code into its natural partitions, such as functions. Therefore, identifying pieces of work to the user can be cumbersome. Identifying message types and their association with work is an even more severe problem.

A message can be a string of characters, or an array of some sophisticated data types; a *send* call in a typical SPMD model simply takes an address and the number of bytes starting from that address that need to be sent. Therefore, the messages in the system can be of any arbitrary type. Are there other ways in which the type of a message can be determined? Can a message be distinguished using the tag which the user attaches, or the line numbers of the *send* or *receive* statement corresponding to it?

```
send(msg1, B, size1, tag1);
barrier();
send(msg2, B, size2, tag1);
```

(A)

```
recv(msg1, &pe, &size, &tag);
f1(msg1);
barrier();
recv(msg2, &pe, &size, &tag);
f2(msg2);
```

(B)

Figure 6.1: Messages with identical tags can do different pieces of work.

If the tags of two messages are the same, it does not necessarily mean that the message types (and their intended functionality) are the same. A user may re-use the same tag for two messages in two different portions of his code, if he knows that they cannot arrive at the same time. Figure 6.1 shows an example in which process A sends two messages to process B which have different functionalities, but have the same tag for the messages; the same tag can be used for both messages without any possibility of conflict because the two processes participate in a barrier synchronization between the sends for the two messages. This example illustrates that a tag cannot always be used by itself to distinguish between messages.

Can the originating line number of the corresponding *send* for the message be used as a distinguishing factor? The line number is also not sufficient, because messages doing identical work can be sent from different points in the program. Consider the example in Figure 6.2. Both (A) and (B) show examples of how a spanning tree reduction could be implemented in the

```
if  (NumberMyChildren==0)
     send ( msg, parent, size, tag1);
else  {
      received=0; tag=tag1;
      while (received != NumberMyChildren) {
             recv (msg, &pe, &size, &tag);
             compute(msg);
             received++;
      }
      if (parent)    send (msg, parent, size, tag1);
}
```

(A)

```
received=0; tag=tag1;
while (received != NumberMyChildren) {
       recv (msg, &pe, &size, &tag);
       compute(msg);
       received++;
}
if (parent)    send (msg, parent, size, tag1);
```

(B)

Figure 6.2: Messages doing identical work can be sent from different portions of the user program.

SPMD paradigm. Each message is combined with values at an interior node in the spanning tree using the *compute* call, and therefore they do identical work. In the implementation in Figure 6.2(A), since the two sends originate from different line numbers, it is not possible to determine whether or not their intended actions are the same. Note that the code could be written as in Figure 6.2(B), in which case the line number of the send could be used to distinguish this type of message.

```
ReceiveLoop() {
     done=0;
     while (!done) {
           tag=-1;    pe=-1;
           recv(msg, &size, &pe, &tag);
           switch (tag) {
           case SERVICE:
                   compute(msg); break;
           case CLIENT:
                   store(msg); break;
           case TERMINATE:
                 done=1;
            }
     }
}
```

Figure 6.3: Messages which do different things can be received at the same place in the program.

Can the line number at which the message was received be used as a distinguishing factor? This is also not always possible. Consider the example of the receive loop in Figure 6.3. Such a loop is common in cases when the user is trying to maximize the number of outstanding receives that are acceptable at any point in time, e.g., we are using such a receive loop in a parallel molecular dynamics programs called NAMD (Not Another Molecular Dynamics Program) being written in PVM. The basic idea is that messages of all types can be received using a wild card receive statement, and the appropriate processing function can be invoked according to the tag or the contents of the message. Therefore, the line number of the corresponding receive cannot be used by itself to distinguish different types of messages.

In most cases it is possible to classify messages into types using a combination of tags and line numbers of sends and receives; however this approach can be cumbersome. However, in the worst case, the system may end up identifying the number of different types of messages in the system to be as large as the number of messages sent in an execution run. The increased complexity of creating displays and analysis for a large number of message types forces performance tools for SPMD program to present information about messages as a whole, with no distinction between different types of messages. Further analysis is often done with respect to a processor, and not with respect to specific work on a processor. We elaborate on the consequences of this difference in the foci of previous work and our approach with examples throughout this chapter.

6.2 Integrated automatic performance analysis

In performance analysis, a user typically examines performance data for potential performance problems, such as load imbalance and poor granularity of tasks. Our objective in automatic analysis is to automatically provide the user with feedback about the components of a parallel program responsible for poor performance, along with information about the performance loss due to each such component. The process of automatic analysis is meant to be iterative: the user can choose the component affecting the performance most severely, make appropriate modifications, invoke the analysis component again, and repeat the analysis-modification process until it results in an efficient parallel program.

After the event graph has been constructed by reading in all the program traces, automatic analysis can be carried out. Figure 6.4 summarizes the basic algorithm for automatic analysis. We first determine attributes of the event graph, such as the last event chain, logical separation points, and the pattern of message passing for each entry point using the functions DetermineLastEventChain, DetermineLogicalSeparationPoints, and DeterminePatterns, respectively. Once the logical separation points have been determined, the execution of the program can be partitioned into logically independent phases using adjacent pairs of logical separation points. The performance of events within each logically independent phase is analyzed separately.

```
TASK_LIST *separation_list, *last_event_list;

Expert() {
    /* determine logically independent phases and patterns */
    DetermineLastEventChain(last_event_list);
    DetermineLogicalSeparationPoints(separation_list);
    DeterminePatterns();

    /* for each logical phase */
    for (current=separation_list; current; current=next) {
        next = current→next;
        utilization = ComputeTaskCounts();
        if (utilization < 75) {
            SystemIdiosyncrasy(current, next);
            PhaseByPhaseAnalysis(current, next);
        }
        CombineAnalyses();
        PhaseReport();
    }
    WastefulWorkAnalysis();
    EvaluateLDB();
    SharedVariableAnalysis();
    CriticalPathAnalysis();
    SummaryReport();
}
```

Figure 6.4: The performance analysis expert.

Before any analysis is done, the function ComputeTaskCounts computes the following quantities for each logically independent phase:

N_e^p, the number of instances of execution of the entry point e on processor p,

G_e^p, the average granularity for the entry point e on processor p,

N_e, the number of instances of execution of the entry point e on all processors (i.e., $\sum_p N_e^p$),

$T_e = \sum_p (N_e^p G_e^p)$, the total time spent executing entry point e across all processors, and

G_e, the average granularity for the entry point e on all processors (T_e/N_e).

The function ComputeTaskCounts also returns the processor utilization[1] for the current logically independent phase. Analysis techniques are invoked for a phase if the processor utilization during that phase is less than 75%[2].

The first analysis technique invoked ascertains whether the performance loss is actually due some system idiosyncrasy. Once it has been determined that the performance problem is because of user code, PhaseByPhaseAnalysis is invoked. The inputs to PhaseByPhaseAnalysis are the event graph, its attributes, and information about counts and granularities of events. The broad goal of the function PhaseByPhaseAnalysis is to provide feedback to increase the utilization of processors with the following caveat: Utilization must be increased, while keeping the amount of speculative work low. In *speculative* computations, the amount of work that a processor does depends on the order in which the execution of sub-tasks in the computation is scheduled. If a poor schedule is chosen, processors can end up doing a large amount of wasteful work. Though this will keep the processors busy, the turnaround time of the program will increase.

PhaseByPhaseAnalysis, shown in Figure 6.5, first identifies all those points in time in each logically independent phase when the utilization of processors is poor. If processor utilization is poor even though there is work in the processor's queues, then one possibility is that the overheads of task creation dominate useful computation. Utility analysis can be carried out to determine the parts of code that are responsible for such performance loss.

Processor utilization may be poor because there is no work in the processor's queue, i.e., the processor idles. The analysis techniques that can be invoked at this point depend on the nature of tasks involved: If the tasks involved are those which are dynamically mappable under the control of a load balancing strategy, then it becomes necessary to analyze the performance of the strategy itself. However, if the tasks involved are statically mapped, then other techniques become relevant:

[1] We define *processor utilization* to be the average percent of time spent by each processor in doing user work.

[2] The choice of 75% as good processor utilization is a heuristic. One way to tune performance would be to allow the user to set the heuristic value at lower levels, such as 50%, at the beginning of performance analysis, and move it to higher levels, such as 85%, later on. This would allow them to focus on severe performance problems early on in the analysis. Later, the user could move the threshold higher, thereby getting an idea of the less severe factors affecting performance.

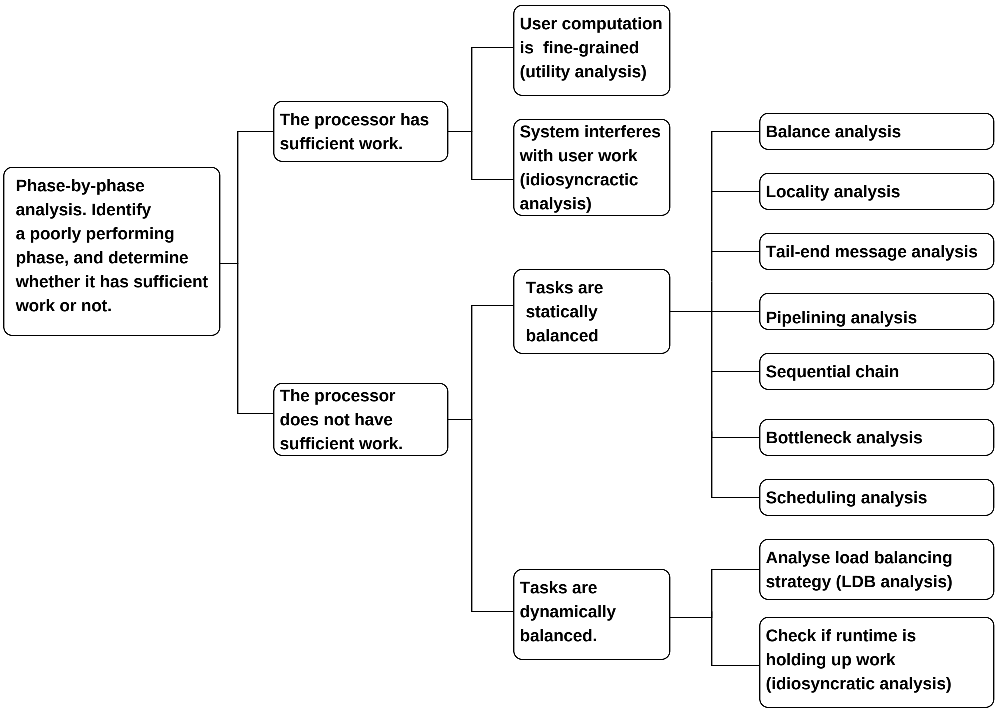

Figure 6.5: Phase-by-phase analysis.

1. A processor could idle while waiting for other processors to complete, if the load on that processor was less than the load on other processors (balance analysis).

2. Or, a processor could idle if the degree of parallelism was insufficient, so that there wasn't enough work to give to every processor (degree of parallelism analysis).

3. Or, a processor could wait if the next message it was supposed to receive was a large message (latency analysis).

We have briefly motivated some techniques for performance analysis. The difference in the analysis for dynamically and statically mapped tasks appears large. However, a number of techniques used for statically mapped tasks are also included in a modified manner in the load balancing strategy analysis. In Section 6.4, we discuss these and other relevant performance analysis techniques in greater detail.

6.3 Evaluation of severity of a performance problem

The applications of performance analysis techniques in the automatic performance analysis framework will often result in a litany of performance problems. In most cases, there are only a few severe performance problems. In order to make performance analysis worthwhile, one should be able to identify the worst offenders so that users can then concentrate their efforts on solving those problems first whose solution would yield the maximum performance benefits. We define the *severity* of a performance problem as follows:

Definition 12 *The* severity *of a performance problem is the amount of reduction in the program's execution time if the problem is fixed.*

The first heuristic is not to report any trivial problems. For example, the CharmInit entry point lies on the critical path and the last event chain. Therefore, it can be identified as a performance problem by many analyses. As an application of the first rule, we do not report any problems with CharmInit, except under special circumstances.

The severity of problems with some entry points are minor. How does one choose a cutoff for the severity? Instead of choosing a minimum level of severity of a problem before it is reported, we have chosen to ignore a problem, if its normalized severity is not too large. In formal terms, the heuristic we use is:

Heuristic 1 *If a problem is determined to have a severity s, it is not deemed sufficiently severe, unless $\frac{s}{S} > f$, where S is the combined severity of all problems in the logically independent phase, and f is a heuristically determined fractional value. In our case we have chosen it to be 0.2.*

We define the function *overlap* which takes two arguments as follows:

Definition 13 *$overlap(t_1, t_2) = max\{\sum_{v \in V_p^{t_1,t_2}} (min(t_2, v_f) - max(t_1, v_s)) \mid p \in P\}$, where P is the number of processors, $t_1 \leq t_2$, and $V_p^{t_1,t_2} = \{v \mid (v \in V_p) \wedge (v_s \leq t_2) \wedge (v_f \geq t_1)\}$ is the set containing the events that occur within a time interval (t_1, t_2) on processor p.*

Then, overlap(t_1, t_2), defines the maximum time spent in user computation on any processor between time t_1 and t_2 as shown in Figure 6.6. Why is the overlap function useful? Suppose there is a performance problem, which if solved could eliminate the time interval (t_1, t_2) from the program's execution time on one of the processors. Because at least one processor has

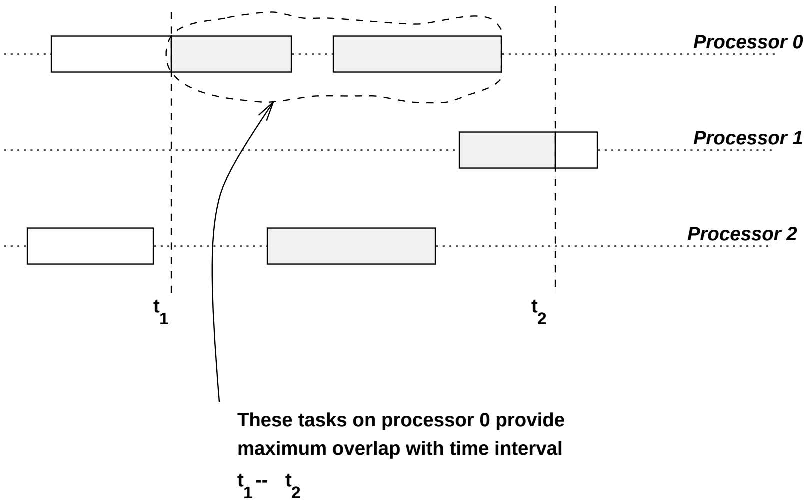

Figure 6.6: The overlap function defines the amount of computation that overlaps with a specified period of time.

computation equal to *overlap*(t_1, t_2) in that same interval of time, so the best case improvement in the program's performance could only be $t_1 - t_2 - overlap(t_1, t_2)$.

6.4 Performance analysis techniques

In this section, we describe some performance analysis techniques and the methods with which we evaluate the severity of the performance problems they determine. Performance analysis techniques can diagnose some well known problems, such as imbalance in load [8], the time taken for synchronization [8], and small granularity with respect to communication latencies [9]. We describe techniques that diagnose common place problems such as above, and other more sophisticated problems. The techniques described in this section have been arrived at by asking questions, such as *"Why is a processor not doing useful work at this time?"*. The techniques range from questions about the benefits of creating a task to questions about the balance of shared variable access across processors.

6.4.1 Scheduling analysis

The message-driven execution model of Charm adds a new dimension to performance analysis. We have previously noted that Charm provides greater possibilities of concurrency because any message that arrives at a processor has the potential of getting scheduled for execution. Could this adaptiveness in scheduling lead to performance problems? If a message that creates a task on the last event chain is delayed because other arriving messages which are not on the last event chain are scheduled for execution, there is a possibility that the turnaround time for the application could be increased. Charm provides two different schemes to counter such a problem:

1. *Queuing strategies*: Charm permits the user to link in one of many different queuing strategies. The user can exercise coarse control over the execution order of messages by choosing an appropriate strategy. Thus, for example, by choosing a *lifo* strategy, the user can schedule the execution of later arriving tasks first.

2. *Prioritization:* A better mechanism to exercise control over message scheduling is by assigning messages priorities. Prioritized queuing strategies in Charm select the message with highest priority available in the queues for execution. So, for example, it is possible to shorten the length of the last event chain by giving preference to the tasks that lie on it.

In this section, we examine a technique to analyze the execution of a program in order to be able to suggest which tasks should be prioritized to improve turnaround time. The last event chain contains the last events in each logically independent phase. Therefore, it contains the set of tasks which affect the execution time most severely, and improving which would improve the turnaround time. We define tasks on the last event chain of the execution of a program as *critical tasks*, and others as *non-critical tasks*.

Figure 6.7 shows a situation, where a different schedule based on priorities could improve turnaround time. The tasks on the last event chain that need to be completed on processor 1 are delayed because the task C on processor 0 that triggered them is scheduled later than other tasks. Scheduling analysis would identify such occurrences and suggest that C be given higher priority than A and B.

Scheduling analysis examines each event on the last event chain, and proceeds to determine a better schedule if it finds a task which was scheduled on an idle processor. Our experience shows that the above technique works only if the following conditions are met:

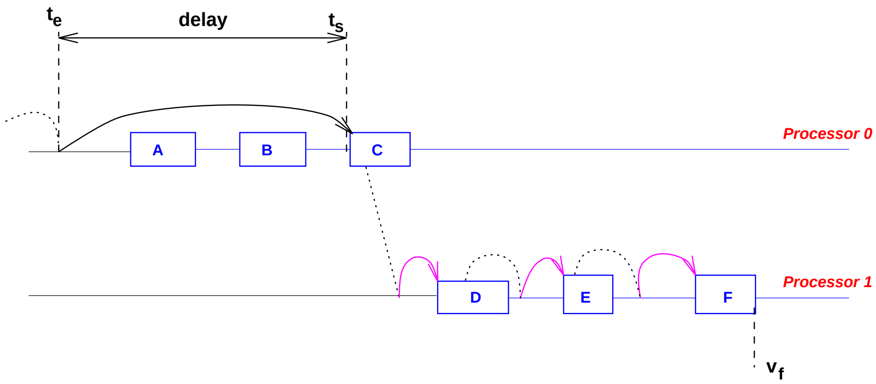

Figure 6.7: Scheduling analysis.

1. There are critical tasks, such as C in Figure 6.7, which wait to be scheduled while other non-critical tasks, such as A and B in Figure 6.7, are executed. This criterion determines non-critical tasks whose execution could be scheduled later.

2. There are critical tasks, such as D in Figure 6.7, whose execution is preceded by idle periods. This criterion establishes the possibility of improvement, because critical tasks could be scheduled during these idle periods (of course, without causing other idle periods), thereby increasing processor utilization.

Scheduling analysis in Charm consists of three components: the first component determines the last event chain in the program's execution, the second component checks whether there are any critical tasks whose execution was preceded by idle periods, and the third component checks for non-critical tasks which were executed while critical tasks were not scheduled for execution.

Let t be a task on the last event chain which is delayed. Let v be the last delayed task on the last event chain, such that $t \Rightarrow v$.

Heuristic 2 *The severity of this scheduling problem is $t_s - t_e - overlap(v_f - (t_s - t_e), v_f)$, where t_e is the time at which the message for task t was enqueued for scheduling.*

Justification: If a task on the critical path is scheduled later than other tasks not on the critical path, then the delay is the elapsed time between the time the message was enqueued, t_e, and the time at which the system started processing the message, t_s. Therefore, v could have finished as much earlier as the delay in scheduling task t, which is $t_s - t_e$. Subtracting *overlap*$(v_f - (t_s - t_e))$, the amount of overlap within the period $v_f - (t_s - t_e), v_f$, we get the maximum possible performance improvement, and hence the severity of the problem as $t_s - t_e - overlap(v_f - (t_s - t_e), v_f)$. □

Since priorities are effective only for tasks which reside in the queue at the same time, there can be still be a problem if long non-critical tasks are scheduled for execution before the arrival of the message for a critical task. In this case, even though the critical task may have higher priority, it will not be selected because it was not even in the queue. One possible solution to this problem is to provide constructs which force a message to wait. In the scenario we just described, non-critical tasks could be forced to wait, even though the processor was idle, until the message for the critical task arrives and is scheduled for execution.

6.4.2 Tail end message analysis

Quite often it is the case that the message to a remote processor is sent at the end of an entry point. If the user computation has a sufficient degree of parallelism, then it is likely that the wait time necessary for the communication of the message will be overlapped with something else on the receiving processor. However if that is not the case, the receiving processor could idle until the message arrives. The solution is obvious, but may not be possible in some cases: move the send earlier in the entry point. It may not be possible if the information necessary for the message is available only at the end of the entry point.

Heuristic 3 *Let x be the tail end event, and let y be its creator. Then the severity of the problem is $(x_c - y_s - overlap(y_s, x_c))$.*

Justification: Since the problem is because the message is sent at the tail-end of the creator entry point, the execution time could be speeded by moving the corresponding send earlier. The earliest it could be moved to is the beginning of the creator's entry point, i.e., an improvement of $(x_c - y_s)$. Subtracting from it the overlap in that period of time, one gets the greatest possible improvement, and hence the severity of the problem, as $(x_c - y_s - overlap(y_s, x_c))$. □

6.4.3 Pipelining analysis

In a computation, sometimes large messages may need to be sent to a remote processor for processing. Since the latency involved in sending a large message is considerably longer, if the transmission time of the communication is not overlapped with other work, the receiving processor can be idle for some time before the message arrives. Figures 6.8(a) and 6.9(a) show the case of a large message that arrives at an idle processor[3]. For the remainder of this section, we shall refer to events in Figure 6.8 and Figure 6.9 as Case 1 and Case 2, respectively.

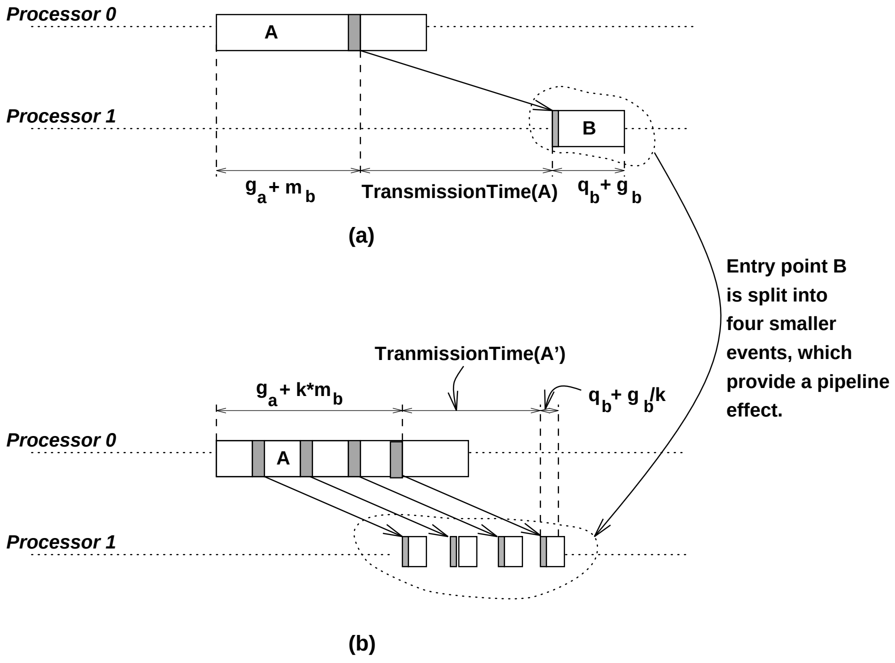

Figure 6.8: Pipelining when $g_a > g_b$.

The severity of the problem can be reduced if the message is pipelined: the message is broken into smaller fragments, and they are sent out individually. The transmission times of later fragments can then be overlapped with the computation time for the earlier fragments. Figures 6.8(b) and 6.9(b) show the effect of pipelining on the same examples.

In both Figure 6.8(a) and 6.9(a), when the large message is sent without pipelining, the duration of time from the beginning of task A to the finish of task B is:

$$\underbrace{g_a + m_b} + \underbrace{(\alpha + \beta s_b)} + \underbrace{g_b + q_b}$$

where, the first term is the combined granularity of A and the time to allocate the message, the second term is the transmission time for the message to entry point B of size s_b, and the third term is the granularity of B and the time to schedule the message for execution.

We make the following assumptions for pipelining:

[3] Note that the difference in the two cases is the relative granularities of tasks A and B.

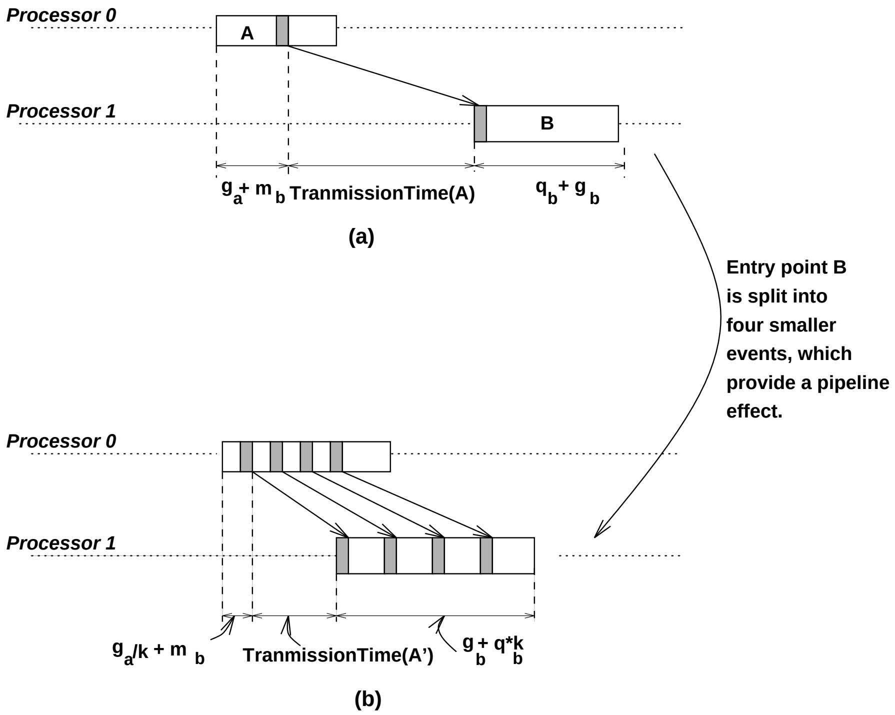

Figure 6.9: Pipelining when $g_b > g_a$.

1. If the large message is split into k smaller ones, then each fractional message would need to be computed by $\frac{1}{k}^{th}$ of the computation associated with entry point A,

2. Each of the k small fractions of the large message would result in computation equal to $\frac{1}{k}^{th}$ of the computation associated with entry point B.

Thus, in Figure 6.8(b), a message of size $\frac{m_b}{k}$ is sent at the end of $\frac{g_a}{k}$ computation and causes $\frac{g_b}{k}$ computation at the other end. In addition, each new message requires m_b for allocation. Therefore the new computational time of A is $g_a + m_b k$. The duration of time from the beginning of task A to the finish of the last fraction of task B is:

$$\underbrace{g_a + m_b k} + \underbrace{(\alpha + \beta(\frac{s_b}{k}))} + \underbrace{q_b + \frac{g_b}{k}}$$

where, the first term is the increased granularity of A, the second term is the transmission time for each fractional message, and the third term is the granularity of $\frac{1}{k}^{th}$ fraction of B and the scheduling time.

Similarly, in Figure 6.9(b), a message of size $\frac{m_b}{k}$ is sent at the end of $\frac{g_a}{k}$ computation and causes $\frac{g_b}{k}$ computation at the other end. In addition, each new message requires q_b for scheduling on processor 1. Therefore the duration of time from the beginning of task A to the finish of the last fraction of task B is:

$$\underbrace{\frac{g_a}{k} + m_b} + \underbrace{(\alpha + \beta(\frac{s_b}{k}))} + \underbrace{g_b + kq_b}$$

where, the first term is the granularity of $\frac{1}{k}^{th}$ fraction of A and allocation of a message, the second term is the transmission time for each fractional message, and the third term is the granularity of B and the time necessary for scheduling.

The reduction in time duration for Cases 1 and 2 is:

$$Case\ 1: \quad (\beta s_b + g_b)(1 - \tfrac{1}{k}) + m_b(k-1) \tag{6.1}$$

$$Case\ 2: \quad (\beta s_b + g_a)(1 - \tfrac{1}{k}) + q_b(k-1) \tag{6.2}$$

Notice that the reduction depends on the degree of pipelining k. What is the value of k for the maximum possible reduction in Case 1? Taking the first derivative of Equation 6.1, we get:

$$-(\beta s_b + g_b)\frac{1}{k^2} + m_b = 0$$

$$\Rightarrow \quad k^2 = \tfrac{\beta s_b + g_b}{m_b}$$

$$\Rightarrow \quad k = \sqrt{(\tfrac{\beta s_b + g_b}{m_b})} \tag{6.3}$$

The value of k determined by Equation 6.3 provides the maximum reduction in Case 1, because the second derivative is positive. The possible performance improvement in Case 1, is therefore:

$$k_1 = \quad m_b(\sqrt{(\tfrac{\beta s_b + g_b}{m_b})} - 1) + (\beta s_b + g_b)(1 - \tfrac{1}{\sqrt{(\frac{\beta s_b + g_b}{m_b})}})$$

Similarly, the maximum performance improvement in Case 2 will be:

$$k_2 = \quad q_b(\sqrt{(\tfrac{\beta s_b + g_a}{q_b})} - 1) + (\beta s_b + g_a)(1 - \tfrac{1}{\sqrt{(\frac{\beta s_b + g_a}{q_b})}})$$

Subtracting possible overlap that exists in that period, we determine that the severity of the pipelining problem in Cases 1 and 2 are:

$$\begin{array}{ll} Case\ 1 & k_1 - overlap(B_f - k_1, B_f) \\ Case\ 2 & k_2 - overlap(B_f - k_2, B_f) \end{array} \tag{6.4}$$

6.4.4 Message combining analysis

In some cases, there can be a performance loss because there are a number of small messages going from one processor to another. In such a situation, the exact opposite of pipelining is necessary: performance can be improved if the small messages are combined into one larger message. Consider the example in Figure 6.10(a). The entry point A is executed more than

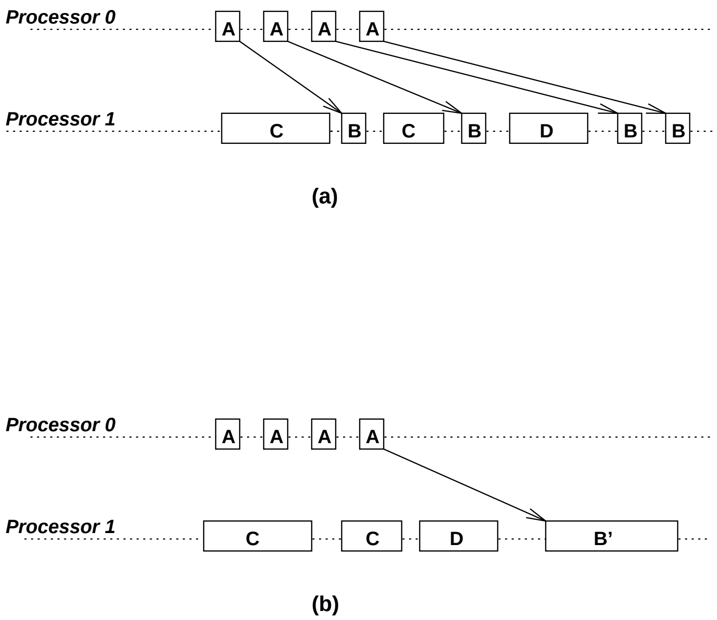

Figure 6.10: Combining messages.

once on processor 0, and each time a small message is sent to entry point B on processor 1. If the messages from the multiple executions of entry point A are collected into one large message which is sent at the end resulting in a longer entry point B, as shown in Figure 6.10(b), turnaround time can be improved, because one saves on the initial latencies of creating multiple small messages. Of course, such a combination will improve performance only if there is sufficient work to do on the other processor (processor 1 in this example), so that it will not idle while the larger message is in transit.

The cost of allocating and sending k small messages to entry point B is $k(m_b+\alpha+\beta s_b)$. The cost of sending one large message is $(m_b + \alpha + \beta k s_b)$. The maximum reduction in turnaround time is $(k-1)(m_b+\alpha)$, which is possible if the destination processor has sufficient work to overlap the transit time of the larger message.

Thus, the criterion for combining and pipelining are different:

1. Combining is a useful technique when the latency of sending a larger message can be overlapped with computation.

2. On the other hand, pipelining is useful when the latency of sending a large message cannot be overlapped with computation.

6.4.5 Sequential chain analysis

One particularly damaging set of events can be a sequential chain: it is a set of events, $x^1, ..., x^n$, such that $\forall_{i=1,n-1}(x^i \rightarrow x^{i+1})$ and no two events of the chain exist on the same processor. Figure 6.11 shows an example of a sequential chain.

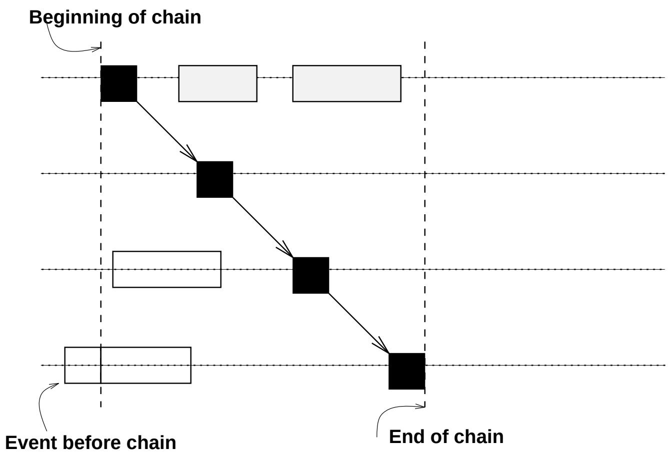

Figure 6.11: An example of a sequential chain.

We identify sequential chains in the event graph by examining the logical matrix (defined in Section 5.4) for entry points. The properties of the logical matrix for a sequential chain are:

1. The maximum logical depth $\leq p-1$, where p is the number of processors,

2. Each processor executes only one event.

3. There is only one event on all processors for any logical depth.

Processor	Logical depth			
	0	1	2	3
0	3	-1	-1	-1
1	-1	0	-1	-1
2	-1	-1	1	-1
3	-1	-1	-1	2

Table 6.1: The logical matrix for a sequential chain.

Table 6.1 shows the logical matrix for a sequential chain.

Heuristic 4 *Let $x^1, ..., x^n$ be the sequential chain of events. Then the severity of the problem is $\frac{(P-1)(x_f^n - x_s^1)}{P} - overlap(x_s^1, x_f^n)$, where P is the number of processors.*

Justification: The duration of the sequential chain is from the beginning of the first event till the end of the last event in the chain, which is $(x_f^n - x_s^1)$. If the sequential chain were distributed across all processors, then each processor would do only $\frac{1}{P}^{th}$ of the work. Hence the total improvement in performance, or the severity of the problem is $\frac{(P-1)(x_f^n - x_s^1)}{P}$. Subtracting from it the overlap in that same period of time, one gets the total time added because of the sequential chain, which is $\frac{(P-1)(x_f^n - x_s^1)}{P} - overlap(x_s^1, x_f^n)$. □

6.4.6 Bottleneck analysis

An entry point e can become a bottleneck on one processor, if it becomes the recipient for messages from all processors. Figure 6.12 shows event diagrams for two different types of bottleneck:

1. In Figure 6.12(a), entry point A becomes a bottleneck, because it waits for messages from all other processors before creating subsequent activity in the program.

2. In Figure 6.12(b), entry point A becomes a bottleneck because it acts as a server for requests from all other processors.

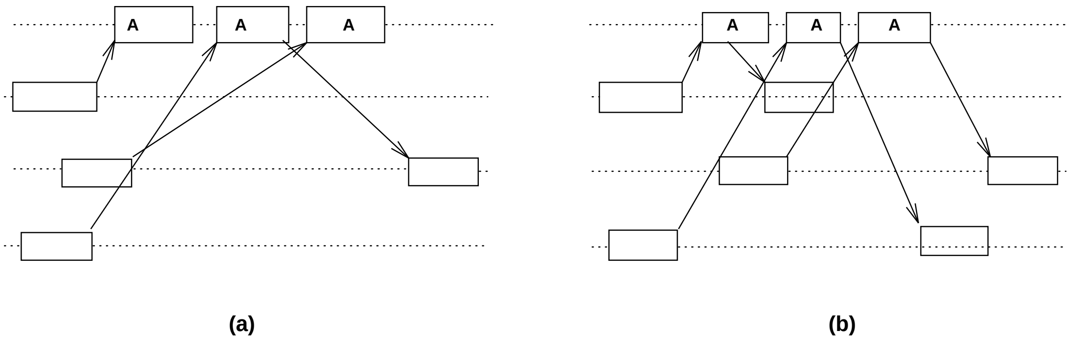

Figure 6.12: Two examples of bottleneck.

We can determine if an entry point is a bottleneck by examining its logical matrix. The properties of the logical matrix for an entry point which is a bottleneck are:

1. The maximum logical depth of events of type e is 0.

2. All events of type e are processed at a single processor.

3. Each processor creates at least one event; the processor which processes all events may be excluded.

Table 6.2 shows the logical matrix for a bottleneck on four processors.

Processor	Logical depth 0
0	(1, 2, 3)
1	-1
2	-1
3	-1

Table 6.2: The logical matrix for a bottleneck.

6.4.7 Task utility analysis

An activity, such as the creation of a task has an associated cost and utility. Utility analysis attempts to measure the cost/utility ratio of an activity to decide whether the activity is really useful. For Charm programs, utility analysis analyzes the effectiveness of creating a new task.

How can one determine whether it is useful to create a task? We make the following observations:

1. In the Charm execution model, a message non-preemptively invokes an entry point inside a chare. Therefore an entry point e can be considered to be a task and the associated message m^e its creator.

2. The utility or *granularity* of a task in a program is the average computational time needed by the task.

3. The cost of creating the task is the cost of creating the associated message. This includes the cost of allocating the message, the latency involved in sending it to a remote processor[4], and the cost of scheduling the message on that processor.

The utility of a task can be determined by comparing the granularity of the task and the cost of creating the associated message.

The granularity of tasks in an application is an important factor in the performance of an application. If the tasks are too fine grained, then the system overheads (communication latency time, message processing time, shared memory access time, context switch time, scheduling, etc.) can adversely dominate the execution of the program. Conversely, if the tasks are too large-grained, then there would be too few tasks to effectively parallelize. Therefore it is necessary to carefully choose an appropriate granularity for tasks in a parallel program.

Note that the choice of granularity may need to be machine-dependent. For example, the cost of creating, transmitting, and scheduling a small Charm message is about 1500 microseconds on a network of workstations, while it may take as little as 200 microseconds on the CM-5. So the utility/cost ratio for a program, whose messages are small and the average granularity of tasks is 1-2 milliseconds, is favorable for the CM-5, but not favorable for a network of workstations. Of course, if the messages were large, then the granularity of tasks would need to be greater.

The analysis corresponding to task utility analysis in previous work [10] is the *communication to computation* ratio analysis, which looks at the ratio of the amount of time spent in

[4] The message driven execution model of Charm provides multiple mechanisms to tolerate latency. Therefore, it seems contradictory to include this in the analysis for cost of creating a message. In reality, when the processors have high utilization (indicating good overlap) this analysis will not be invoked. This analysis is invoked only when the processor utilization is low and there are sufficient tasks, which often indicates high system overheads.

communication to the amount of time spent in computation *by a processor*. Fowler et al [10] have proposed the following systematic approach to use this ratio in order to improve a program:

1. If the ratio is small, it indicates that a lot of time is spent in computing. One needs to perform critical path analysis to identify sequential portions of the code that affect performance most.

2. If the ratio is large, it indicates that the processor is communicating excessively, and the user program needs to be examined to reduce communication traffic.

All analysis is performed manually by the user with the assistance of multiple views of the program data. One problem with this strategy is that even though it may identify the problem and the corresponding processor, it cannot identify more specifically the block of code or message in the program that needs to examined. Our analysis will identify the offending entry point; thereby attention can be focussed on improving the utility/cost ratio for that entry point.

Heuristic 5 *If an event x is determined to have poor granularity, then the severity of the problem is $(I_x \sum_p N_x^p - T_x)/P$, where I_x is the ideal granularity for entry point x.*

Justification: Assume that T_x, the total computational time for entry point e cannot be altered. Since the entry point has poor granularity, performance can be improved if the overheads of executing the entry point is reduced. This is possible if the entry point is executed fewer times with larger granularity; the easiest choice for granularity is the smallest acceptable granularity I_x[5]. The current overheads are $I_x \sum_p N_x^p$. If each execution of the entry point has at least the smallest acceptable granularity, then the entry point e will be executed T_x/I_x times, and the total overheads will therefore be $(T_x/I_x)I_x$, which is T_x. Therefore the decrease in total overheads will be $(I_x \sum_p N_x^p - T_x)$, and so each processor's execution time can potentially be reduced by $(I_x \sum_p N_x^p - T_x)/P$. □

The implications of utility analysis is different for different types of messages. For a message which creates a chare, poor granularity might suggest that the computation intended for the chare be subsumed, i.e., be carried out by the creator chare itself. In contrast, it may not be possible to subsume the computation in a message which is a request or a response to an

[5] A good choice for granularity is about ten times larger than the cost of creating the task.

existing chare, for the necessary information may lie or be needed at the destination. In such a case, the user may need to increase the granularity of the message by combining many messages to the same chare into one larger message, if possible.

6.4.8 Runtime idiosyncrasies analysis

In high-level parallel languages, there are often idiosyncrasies in their implementation. In most cases, the implementation details do not, and should not, matter. However in some cases, knowledge of the implementation details can help solve mysterious performance problems. A low-level, but illustrative example of a system idiosyncrasy was reported by Saini [74]: On the Paragon a program executed twice over ran faster the second time than it did the first time. It is necessary for an automatic performance tool to carry a list of such idiosyncrasies so that it can inform users if their program is affected by one of them. In a system such as Charm, which also provides many advanced features, such as automatic load balancing, quiescence, etc., the user must be willing to accept certain overheads. However, the system must inform the user if the overheads of these underlying mechanisms outweigh their utility, and the user should be advised against their use if at all possible. The following analyses are carried out in Projections to determine Charm runtime idiosyncrasies:

1. *Are there tasks which are too large grained?* We saw earlier that tasks needed to have a minimum granularity so that their utility exceeded the overheads of creation. However, tasks which are of large granularity can also cause problems in ways that may be unexpected to the user. Since the execution of an entry method is atomic (it cannot be interrupted until it is executed completely), large grained tasks may occupy processor resources, thereby causing delays in processing of small grained requests, possibly critical, by other processors. For example, in some load balancing libraries each processor sends out messages to request work from other processors. If these messages are delayed by a large grained task on the destination processor, the requesting processor may not get work. Consequently the load balance may suffer critically. Similarly, requests for data from distributed table are sent in the form of small grained messages; the delay in servicing such requests will delay computation waiting for the reply.

Some system tasks may be carried out through system interrupts (without violating atomicity for the user code). However network interrupts are generally expensive on current parallel machines. Therefore in programs where there are large grained entry methods which cause such problems, the user must be informed that they be split into smaller tasks.

2. *Are there large sequential code portions?* The execution of a Charm program starts at the CharmInit entry point, where the user can create specifically shared variables. Charm semantics specifies that the value of all variables created inside CharmInit are available to all other entry points. Since the user can create other computation inside the CharmInit entry point, such computation needs to be delayed until the values of all specifically shared variables created inside CharmInit are available on all processors. This is implemented by having processor 0 broadcast an initialization message after the completion of the CharmInit entry point. User computation begins on processors only after the initialization message has been processed.

   Since all processors wait until the completion of CharmInit, if the user performs a lot of work inside CharmInit, it will result in the sequentialization of a large portion of their code. The performance analysis tool should be able to detect this and inform the user.

   Heuristic 6 *If system idiosyncrasy analysis determines that the CharmInit entry point was taking too long, then the severity of the problem is* $G^0_{CharmInit}(1 - \frac{1}{P})$.

   Justification: One could obtain better parallel performance, if the computation inside CharmInit entry point were distributed across all processors, resulting in $\frac{G^0_{CharmInit}}{P}$ computation. The savings in sequential execution time would be $G^0_{CharmInit} - \frac{G^0_{CharmInit}}{P}$, and therefore the benefit across P processors would be $G^0_{CharmInit}(1 - \frac{1}{P})$. □

3. *Is load balancing strategy useful?* A user who does not need a load balancing strategy (because the program does not have any dynamically created chares that need to be dynamically mapped) may still be paying for the overhead of a sophisticated load balancing scheme that periodically exchanges load information, for example. By examining the types of objects created (dynamically mapped vs. statically mapped), we can recommend against using a particular strategy in a particular program.

4. *Is quiescence useful?* The overheads associated with quiescence include a periodic function call and messages when processors go idle. The quiescence detection algorithm works efficiently with little overhead if the user program has few idle periods. However if the user program does not need quiescence detection or if quiescence detection is active for a large period of time, the user could be advised to omit quiescence detection or start it late.

6.4.9 Balance analysis

One of the most common performance problems in parallel programming is the imbalance in the utilization of processors. This is specially true in the case of irregular applications where the amount of work associated with a task is not easily determined. A processor's time can be divided into three types of activities: user work, overhead, and idle time. Balance analysis attempts to evaluate whether each type of activity is balanced across processors. Note that balancing any two of them will balance all three activities.

In previous work, balance analysis has been done by comparing the total amount of computational time and overheads on different processors. Our approach focuses on identifying the specific task which is the cause of imbalance.

6.4.9.1 User work

We have shown earlier in this chapter how the computation on a processor can be partitioned according to the type of the message that caused it and the corresponding entry point that is executed. An imbalance in user computation on a processor can occur only as the result of an imbalance in computation corresponding to one or more entry points. Balance analysis can now be more specific because it can compare the balance of work for different types of messages, and provide feedback about the sort of message that has imbalance in work, rather than just the processor number. The computation associated with an entry point type e can be imbalanced because of any of the following reasons:

1. The number of messages of type m^e are different on two processors, or

2. The granularity of messages of type m^e are different on two processors.

In theory the load imbalance can be caused by any combination of the above reasons. However in practice it is impacted most by one the above. In any case, it helps the user most if we can identify the worst offenders among the above.

There is one more aspect to comparing loads, because a straight-forward comparison of the loads may not always produce useful analysis. Consider a program, where all processors do an equal amount of computation, and subsequently participate in a spanning tree reduction. In this case, the interior nodes of the spanning tree would process k messages, where k is the branching factor, while the leaf nodes would process no messages. The analysis would then needlessly identify these messages as the source of imbalance and delay.

However, we can eliminate such unnecessary feedback by determining from the pattern of message passing whether balance analysis needs to be done for that particular message type. In our analysis, we do not conduct balance analysis for messages addressed to an entry point whose message passing pattern is that of a spanning tree reduction. On the other hand, we always perform balance analysis for messages whose pattern is that of a ring or a chain (these are defined in Chapter 5). A ring can be distinguished from the following message passing characteristics:

1. In every logical step, each processor receives exactly one message, and
2. Each message travels through some or all of the available processors before arriving at the processor from which it was originally sent.

An example of a logical matrix (defined in Chapter 5) for a chain on 4 processors is shown in Table 6.3.

Processor	Logical depth		
	0	1	2
0	3	2	1
1	0	3	2
2	1	0	3
3	2	1	0

Table 6.3: The logical matrix for a cycle.

Heuristic 7 *If balance analysis determines the number of occurrences of an entry point x to be unbalanced across processors, then the severity of the problem is* $(max(N_x^p) - \sum_p N_x^p/P)G_x$.

Justification: Assume that the total time of execution for entry point x, T_x, or the number of messages for entry point x, N_x, cannot be altered. Performance can be improved if the number of occurrences of the entry point are balanced, i.e., each processor has $\sum_p N_x^p/P$. In that case the processor with the maximum number of occurrences, $max(N_x^p)$ of the entry point x would have $(max(N_x^p) - \sum_p N_x^p/P)$ fewer messages to process, and hence the benefit would be $(max(N_x^p) - \sum_p N_x^p/P)G_x$. $\Box$

Heuristic 8 *If balance analysis determines the granularity of an entry point x to be unbalanced across processors, then the severity of the problem is* $(max(G_x^p) - \sum_p \frac{G_x^p}{P})\frac{N_x}{P}$.

Justification: Assume that the total time of execution for entry point x, T_x, or the number of messages for entry point x, N_x, cannot be altered. Performance could be improved if the granularity of x were balanced, i.e., each processor had $\sum_p G_x^p/P$. In that case the processor with the largest granularity $max(G_x^p)$ of the entry point x would have $(max(G_x^p) - \sum_p G_x^p/P)$ less work to do, and hence the benefit would be $(max(G_x^p) - \sum_p \frac{G_x^p}{P})\frac{N_x}{P}$. $\Box$

6.4.9.2 Overheads

The overheads of system activity can be classified into two categories:

1. *System overhead:* The overheads of background system activity due to libraries for load balancing, quiescence detection, and specifically shared variables are classified under library overhead.

2. *Per message overhead:* The overheads associated with a message for its creation, transmission, and scheduling are classified under per message overhead.

In addition to the balance desired for user work, it's also essential that the overheads of user work also be balanced. Thus, the overheads of a load balancing or quiescence detection strategy should be balanced.

Heuristic 9 *If balance analysis determines the system overheads to be unbalanced across processors, then the severity of the problem is* $(max(\sum_{e \in System} N_e^p G_e^p) - \sum_{e \in System}(\frac{N_e G_e}{P}))$.

Justification: All Charm libraries are implemented as branch office chares, and their work can also be partitioned into entry points belonging to these branch office chares. The analysis of severity for imbalance of system library overhead, then, becomes very similar to the analysis for user work imbalance. Performance can be improved if the library overheads were balanced, i.e., each processor has $\frac{\sum_{e \in System}(N_e G_e)}{P}$. In that case the processor with the maximum overhead, $max(\sum_{e \in System} N_e^p G_e^p)$ would have lesser overhead, and hence benefit, of $(max(\sum_{e \in System} N_e^p G_e^p) - \sum_{e \in System} \frac{N_e G_e}{P})$.

6.4.9.3 User work and overheads

Often the user partitions the set of available processors to perform heterogeneous tasks, so that some processors may do user work, while others perform tasks involving large overheads. For example, we were designing a hierarchical load balancing strategy, which used a multi-rooted tree on the processors. The roots of the tree were responsible for the load balancing while the other processors were responsible for user work: thus even though the roots did no user work while the other processors had very little overhead, the sum of their overheads and user work was balanced. In such cases, it's necessary to determine whether the sum of the user work and overheads of the processors are balanced.

6.4.9.4 Memory usage

Another analysis which is useful for determining scalability (in terms of size of inputs that can be used) is determined by the balance in memory usage. For example, in the hierarchical load balancing strategy discussed above, some large data structures necessary for the strategy were located only on the roots of the tree. Since the branching factor of each tree was 16 and the leaves were at the first level, so in the worst case the strategy did not use as much as $(15/16)^{th}$ of the available memory. Consequently the size of the programs we could run were limited.

As a user, without a performance analysis tool, one will notice this problem only when the program fails on a large size application. With performance analysis one would get advance notice while running smaller problems that a memory imbalance problem exists.

6.4.10 Locality analysis

It is well understood in parallel programming that local requests are faster than remote requests. Remote requests take longer because they have to contend for system resources, such as network or bus, and wait for remote processors to respond to their request. The performance of a parallel program can be improved if a majority of requests are satisfied locally. In locality analysis, the system attempts to analyze how well the program maintains locality of requests during the execution of the program.

One component of locality is to maintain processes that need to communicate on the same processor. In general, this means that the system must examine the communication patterns between all the processes and look for alternative placement of tasks which would reduce inter-process communication. Charm provides the user with many different load balancing schemes, each with different set of trade-offs: the analysis must determine whether the mapping induced by the dynamic load balancing scheme is appropriate to the requirements of inter-process communication in the application. This is discussed in more detail in Section 6.4.11.

6.4.11 Load balancing analysis

Charm permits chares to be created dynamically, and also provides runtime assistance for load balancing of chares. Many different load balancing strategies, such as, random, ACWN (Adaptive Contracting Within Neighborhoods), Manager, Token, and Receiver are available in Charm. The user can choose to link their program with any one of these strategies.

The various load balancing strategies in Charm employ different protocols to balance load. Further, no single strategy is uniformly the best for an application, therefore it becomes necessary to analyze a strategy's performance. In addition to a metric for evaluating the effectiveness of the load balancing strategy, other metrics to evaluate how well it maintains locality and its overhead are also necessary. In this section, we discuss the metrics used to evaluate a load balancing strategy.

6.4.11.1 Load distribution effectiveness

The most obvious metric would appear to be the average utilization of processors. Let $\gamma(t)$ be the function which defines the number of concurrently executing tasks during the execution of the program at time t. Then the utilization of processors during a time interval (t_1, t_2) is given by the expression:

$$\frac{1}{t_2 - t_1} \int_{t_1}^{t_2} \frac{\gamma(t)}{P} dt$$

We have described before that a Charm program can consist of two types of tasks, one of which is dynamically created and mapped, and the other which is statically mapped. Since both these tasks contribute to the processor utilization, this metric is not sufficient to understand the effectiveness of the load balancing strategy. We therefore separate $\gamma(t)$ into $\gamma_s(t)$ and $\gamma_d(t)$, where the subscript s and d denote the static and dynamic components. Therefore, the contribution to the utilization due to dynamically mappable tasks is given by the expression:

$$\frac{1}{t_2 - t_1} \int_{t_1}^{t_2} \frac{\gamma_d(t)}{P} dt$$

Now, consider the following scenario for two strategies:

1. $\gamma_d(t) = 2$ for the first strategy and $\gamma_d(t) = 3$ for the second strategy,

2. $\gamma_s(t) = 3$ for the first strategy and $\gamma_s(t) = 0$ for the second strategy, and

3. Both programs are executed on 5 processors.

The contribution to utilization is 0.40 for the first strategy and 0.80 for the second strategy. Notice that even though the ratio is better for the second strategy, it is not necessarily the better strategy, because the overall utilization is better for the first strategy. Thus the metric must take into account the statically mapped tasks. If there are $\gamma_s(t)$ tasks being executed in the system at any time t, then only $P - \gamma_s(t)$ processors are available for dynamical load balancing. The following metric provides a better estimate of the contribution towards processor utilization by dynamically mappable tasks:

$$\frac{1}{t_2 - t_1} \int_{t_1}^{t_2} \frac{\gamma_d(t)}{P - \gamma_s(t)} dt \tag{6.5}$$

However this metric is also not sufficient. Let δ_s and δ_d denote the number of available static and dynamic tasks during the execution of the program, respectively. Now, consider the following scenario for two strategies:

1. $\gamma_d(t) = 2$ for the first strategy and $\gamma_d(t) = 3$ for the second strategy,

2. $\gamma_s(t) = 0$ for both strategies,

3. $\delta_d(t) = 2$ for the first strategy and $\delta_d(t) = 10$ for the second strategy, and

4. Both programs are executed on 5 processors.

The contribution to utilization is 0.40 for the first strategy and 0.80 for the second strategy. Notice that even though the ratio is better for the second strategy, it is not necessarily the better strategy. The reason is fairly simple: the ratio does not account for the degree of parallelism in the program. Thus, even though the first strategy utilizes the processors poorly, the problem is due to the low degree of parallelism, and not necessarily because of the ineffectiveness of the load balancing strategy.

The following metric provides an indication of the effectiveness of load distribution:

$$\frac{1}{t_2 - t_1} \int_{t_1}^{t_2} \frac{\gamma_d(t)}{\delta_d(t)} dt \tag{6.6}$$

One problem with the metric for effectiveness of load distribution is that it ignores the fact that the maximum number of tasks that can execute at any given point in time cannot exceed the available number of processors. It does not really matter if there are 1000 available tasks, because if there are only 4 processors, the maximum number of concurrent tasks can only be 4. Therefore a second indication of the effectiveness of load distribution is the modified metric:

$$\frac{1}{t_2 - t_1} \int_{t_1}^{t_2} \frac{\gamma_d(t)}{min(P, \delta_d(t))} dt$$

Note that we have cut off the number of available tasks at the number of processors. However, all processors may not be available to dynamically mapped tasks, since statically mapped tasks execute on them. Thus, taking into account statically mapped tasks, the modified second metric for effectiveness of load distribution is the following expression:

$$\frac{1}{t_2 - t_1} \int_{t_1}^{t_2} \frac{\gamma_d(t)}{min(P - \gamma_s, \delta_d(t))} dt \tag{6.7}$$

Note that Equation 6.7 reduces to Equation 6.6 when $\delta_d < P - \gamma_s$, and reduces to Equation 6.5, otherwise. Thus Equation 6.7 models both processor utilization and the effectiveness of the load balancing strategy.

Note that the metrics do not make much sense at the singularity points, e.g., when $\delta_d(t_1) = 0$ at some time t_1. We partition the program using such singular points, and compute the load balancing effectiveness according to Equation 6.7. The value of load balancing effectiveness is chosen to be 1 when the function being integrated has a singularity.

6.4.11.2 Locality induced by strategy

Every time a task is sent to a remote processor, the strategy incurs the overhead of the send and the latency of the transfer. Another important measure of the performance of a load balancing strategy is the extent to which it schedules tasks locally. A good measure of locality is the average number of hops that a piece of work travels under the control of the load balancing strategy. The best scenario is one in which tasks do not make any hops, because the amount of communication required is minimum. The random load balancing strategy in Charm always sends out work to a randomly chosen processor: in this case the average number of hops traveled by a task is always $\frac{P-1}{P}$, because if P chares are created the randomness ensures that one chare will remain locally. In the ACWN strategy, a processor sends out work only if the load on one of its neighbors is less than a threshold, which is dynamically determined according to the loads on neighbors. In this case the number of hops can range from 0 to 3 (the maximum number of hops that a task is allowed to travel), with the average expected to be below 1 because chares are not sent out when the system is in saturation.

6.4.11.3 Overhead of load balancing messages

This quantity is the ratio of the number of messages used by the load balancing strategy to the number of user messages. The random load balancing strategy uses no messages to communicate amongst the branches of the strategy, so this number is 0. The ACWN strategy uses messages to communicate the load status on processors to their branches: these are exchanged periodically, so their number is determined by the length of execution of the program and by the load on processors themselves.

6.4.11.4 Priority balancing

Another metric which is useful in the context of speculative computations is the effectiveness of the load balancing strategy in maintaining priority globally. This metric is computed by taking an average of the global priority rank (amongst all the available messages at that time) of each message when it is picked up for execution. Obviously, the smaller the rank, the more effective is the priority balancing.

6.4.11.5 Use of metrics

How does one use all these metrics to determine the best strategy? There is no straightforward answer even if we consider computations with no speculative component. The least important metric is the overhead of the strategy itself, for if the strategy distributes load effectively, provides better processor utilization, and maintains locality, the overheads do not really matter. Locality is an important metric because the strategy balances only messages to create new chares. Subsequntly, all response messages to the chares are sent to the location where the chare was created. Thus, even though the load balancing strategy may do an effective job, substantial overheads may be incurred because each response message travels to a destination processor. The most important metric in the analysis of a load balancing strategy is Equation 6.7.

6.4.12 Degree of parallelism analysis

One critical efficiency issue for a Charm program is providing sufficient number of parallel tasks. The message driven execution model encourages a style of programming in which the user creates an environment where each processor has more than one task to schedule. Since any task can be scheduled in a message driven strategy, such a style will keep processors busy. Thus a critical measure of program's parallel potential is the degree of parallelism, or the number of tasks available at any point in time. We use the following metric to evaluate a program's degree of parallelism:

$$\frac{1}{t_2 - t_1} \int_{t_1}^{t_2} \frac{\delta_d(t)}{P} dt$$

6.4.13 Shared variable analysis

The nature of information sharing and the methods of access of the shared information often affects the performance of the parallel program. It is therefore important to know about the nature of *shared variable access* in parallel programs. The usage of the five information sharing mechanisms in an application program provides some insight into the nature of information exchange in the program. This insight can be utilized to provide a more accurate analysis of the performance of programs. Performance concerns that can be addressed when one knows the nature of information sharing (through specifically shared variables) in a Charm program are the utility of creating a shared variable and the nature of its accesses.

6.4.13.1 Is it useful to create a shared variable?

An important analysis for a shared variable is to assess whether the cost of it is justified by its use?

1. If some information is represented as a read-only variable or a write-once variable, and is not accessed often, the cost of replicating the variable on nonshared memory machines might exceed the savings in access time. In such cases, it might be better to make the variable into an entry in a distributed table, or to replicate it only on the processors that need it.

2. If a monotonic variable is updated frequently, the *spanning tree* implementation should be chosen. However, if it is updated rarely then the *flooding* implementation should be chosen.

6.4.13.2 Are shared variable accesses local?

Another aspect of locality is to keep accesses to shared variables local as far as possible. In Charm, only the distributed table mechanism is subject to this analysis. The operations on the remaining specific modes of information sharing have been chosen carefully so that most accesses can be implemented with local accesses. The following analyses could be carried to determine if the shared variable accesses are local:

1. If some information is represented as an entry in a distributed table, and it is accessed very frequently by many different processors, analysis could suggest that the data be made write-once.

2. If a large number of entries in the distributed table are accessed only once, it would be most efficient to locate their insertion and access on the same processor (the *locality* principle), where possible.

3. If an entry of a distributed table is accessed repeatedly on the same processor, then it should be cached (the *caching* principle).

6.4.14 Critical path analysis

The critical path in a program's execution is the longest computational chain in the execution of the program. Since the events that lie on the critical path all add up to realize the longest computational chain, it stands to reason that improving their computational performance will shorten the length of the chain. In a parallel program, where the order of execution of events is more indeterminate, it is not always the case that making the critical path shorter by making the tasks that lie on it more efficient will always mean a reduction in the execution of the program. However, Hollingsworth et al [75] have shown that critical path analysis provides much better results than some other sophisticated metrics.

Critical path analysis in Charm determines the longest computational chain (communication costs are considered to be zero) in terms of the entry points which executed. This, then, exactly determines the sequential portions of the code that need to be made more efficient.

6.5 Wait event chain and its utility in analysis

A number of the techniques we defined in Section 6.4, such as balance analysis, tail end analysis, or pipelining analysis, can be applied for every event in the execution of the program. However, in practice, we have found it useful to apply the analysis for the small subset of events which lie on the last event chain.

Figure 6.13 shows the scheme with which different analysis techniques, discussed in Section 6.4, are applied for the events in the last event chain. The broad idea behind last event

```
ChainAnalysis(TASK_LIST *list) {
    int idle;
    TASK *current, *previous;
    for (current=list; current; current=previous) {
        previous = current→previous;
        idle = DetermineProcessorIdleTime(current);
        if (Acceptable(idle)) continue;

        /* we have identified an event with unacceptable wait */
        /* check if its message passing has a problem pattern */
        if (SequentialChain(current))
            previous = SkipToBeginningOfChain(current);
        if (Bottleneck(current)) continue;

        switch (current→type) {
        case NewChare:
            /* dynamically mapped task, hence evaluate ldb strategy */
            evaluate_ldb = TRUE;
            break;

        case ForChare:
        case BocMsg:
            overlap = DetermineOverlap(CreationTime(current),
                BeginProcessingTime(current));
            TailEndMessage(current);
            Pipelining(current);
            SchedulingAnalysis(current);
            BalanceAnalysis(current);
            break;
        }
    }
}
```

Figure 6.13: Last event chain analysis.

chain analysis is to start the analysis with the last event in the last event chain, i.e., the event that finished last in the program's execution. Starting from the current event we trace back along the chain, until we arrive at an event which is scheduled on a previously idle processor. Such an event identifies an idle period on a processor, which when reduced will (likely) have impact on the overall performance. There are two possibilities at this point:

1. If the creator of the current event is also scheduled on an idle processor, it may have added some delay on its own, in addition to propagating any prior delay.

2. If, the creator of the current event is not scheduled on an idle processor, then the delay is either due to scheduling (Scheduling analysis) or other problems, such as large messages (Pipelining analysis), etc.

Using results from pattern analysis, we first determine whether the message passing for an event has the pattern of a bottleneck or a sequential chain. If it does, then this is recorded, and the analysis continues with the previous event on the last event chain. For sequential chain, analysis is simultaneously carried out on all events in the chain. Therefore, analysis for the previous event is unnecessary if it is part of the sequential chain; therefore we skip to the beginning of the chain, and continue the analysis.

If the event's message passing pattern does not match that of a bottleneck or a sequential chain, we proceed to analyze the event according to the type of message that created it. If it is a new chare message, a boolean variable is set to indicate that load balancing analysis should be carried. The analysis is more interesting if the event is a response message to an existing chare or branch office chare. In that case, tail end, pipelining, scheduling, and balance analysis are applied to determine the performance problem.

6.6 Combination of different analyses

The algorithm for automatic performance analysis uses many different techniques to analyze problems. Often analysis will suggest many different strategies to improve performance. For example, the analysis might determine a problem of low granularity with gain analysis and simultaneously determine a problem of load imbalance with balance analysis. Sometimes the analysis reports from two different techniques will have a correlation, which indicates a more severe problem. One must be able to automatically combine such analyses and provide a more realistic feedback. We briefly describe some combinations we have explored:

1. *Balance and Tail end*: Balance analysis may determine that the reason for poor performance is that two processors have imbalanced loads. Tail end analysis may determine that a processor idles because it received a message sent at the tail end of an entry point.

The two analyses can be combined if the wait for a tail end message is increased because the processor sending the message has considerably greater load than the destination processor.

2. *Balance and Pipelining*: Again, balance analysis may determine that the reason for poor performance is that two processors have imbalanced loads. Pipelining analysis may determine that a processor idles because it receives a large message. The two analyses can be combined if the wait for a large message is increased because the processor sending the message has considerably greater load than the destination processor.

3. *Runtime and Load balancing:* A poor load balancing strategy will not only lead to idle processors, but it could also cause quiescence detection to become active and interfering. In such a case, one should report that the excessive overheads of quiescence detection are due to a poor load balancing strategy.

6.7 Related work

Gprof [73] was one of the first tools to provide call-graph profiling. It measures time spent in executing each procedure, and the fractions of that time spent in executing other procedures called from it. This was proposed as a sequential tool, but is useful for analyzing the performance of sequential code-blocks.

Critical Path [76] is the longest computational chain in a program. In the computation of this metric, inter-process communication time is ignored. It provides a good idea of the procedures that take the longest to execute, suggesting thereby that they must be improved in order to improve the performance of the program.

Logical Zeroing [77] attempts to ascertain the improvement in the length of the critical path by zeroing out the effects of a specific procedure. The scheme is simple: each procedure is zeroed out, and the critical path is computed, the difference in the lengths between the old and the new paths is the cost of improving that procedure. One should therefore target the procedure which has the greated benefit.

Quartz NPT [78] measures time spent in doing useful work (ignoring inter-process arcs). For each procedure, the metric is computed by adding the ratios of time spent in the procedure and the degree of parallelism at that point. Essentially, the procedure is divided into periods with the same degree of parallelism, and then the ratio is computed for each period, and added up to get the metric for the procedure. NPT uses elapsed time, and not the time spent in executing the user process. One advantage of this method is that it realistically models events, such as spinning on a lock or waiting for disk i/o. The disadvantage is that one cannot distinguish between the time spent doing useful work and the time the application was not executing.

Slack [75] is a metric used to compute the benefits associated with improving a task on the critical path. Often in addition to the critical path, there exists another secondary path, only slightly smaller than the critical path. Because of this secondary path, fixing the procedures on the critical path, may not greatly reduce the program execution time. Slack attempts to measure the inter-relationship between the critical path and other secondary paths.

In previous work in analysis, Jamieson [79] has used the characteristics of parallel algorithms, in conjunction with the characteristics of parallel architectures, to provide an understanding of how well the algorithm is suited to different architectures.

Recently, Hollingsworth and Miller [62], have developed an approach called the W^3 model, which attempts to reduce the amount of data traced for parallel program performance analysis by intelligently activating the trace dynamically when and where it's needed. Their model attempts to make such decisions based on low level architecture/language characteristics, such as lock-usage, semaphores, and barriers, and some generic high level characteristics, such as an object's wait-time for messages. Our approach deals with more program-specific characteristics of the program, and will provide more language level suggestions for performance improvement.

Fahringer [80] has developed *Weight Finder* and P^3T (Parameter based Performance Prediction Tool) to predict the performance of Vienna Fortran programs. The tool works as follows:

1. First, a sequential profiling run of the program is performed. With this the tool collects concrete and characteristic values for sequential program parameters, such as loop iteration counts, true ratios, and frequencies.

2. Next, based on the sequential parameters, the tool computes parallel performance parameters, such as work distribution, number of transfers, transfer times, amount of data transferred, network contention, and number of cache misses.

3. Finally, the information about the parallel program parameters are used to improve performance of parallelizing compiler by predicting the impact of different data distribution

strategies, and the impact of various program transformation strategies, such as loop fusion and interchange.

Their current approach has two limitations:

1. The performance prediction is based on the six parallel program parameters described above. The relative importance of these parameters need to be identified for each machine by running a set of training programs. This identifies the parameter one should look at in the analysis. However, their strategy does not currently include techniques to choose between conflicting results for parameters. For example, in comparing two strategies if the amount of data transfer is less for the first, but the cache misses are more for the first, then the strategy which needs to be selected is not clear.

2. There are always a number of optimization strategies that can be applied at any point in time. Their work can accurately determine the effect of each individual optimization strategy. However, they do not provide any information on the optimal order of application of multiple strategies. This makes the information about the magnitude of performance improvement less useful: it is easy to know that an optimization is useful, however it is more difficult to determine the order in which optimizations should be tried. Fahringer [81] suggests that determining local minima is not feasible, and believes that a backtracking strategy is sufficient.

6.8 Summary

In this chapter, we have presented a framework for automatic performance analysis. The framework uses the event graph, its attributes and various performance analysis techniques. We have also presented schemes to evaluate the severity of a performance problem. This helps identify the most important performance problems. Performance analysis in Projections is meant to assist the user by identifying significant performance problems automatically and suggesting possible improvements directly, such as the use of a different load balancing strategy, or a prioritization strategy, or to suggest remedies that may or not be possible depending on the user's application characteristics (e.g., pipelining analysis).

Chapter 7

Case studies

In this chapter, we report our experience in using Projections to analyze the performance of four applications:

1. *Traveling salesman problem:* A large portion of the analysis for the first application, a traveling salesman problem, was conducted using only the visualization component of Projections. The analysis motivated the development of semi-distributed load balancing strategies for balancing priorities. Recently, we evaluated the strategies we developed earlier using the automatic analysis component of Projections.

2. *Matrix multiplication:* The second application is a matrix multiplication problem, which was designed to illustrate the use of distributed tables. The analysis allowed us to improve the performance of the algorithm considerably. It also motivated the development of a caching library for distributed tables.

3. *Multiple linear solvers*: The third application is a multiple linear solver. This application illustrates how incomplete information about a program can cause automatic analysis to fail. It also illustrates the usefulness of the display component of Projections.

4. *EGO*: The fourth application is EGO, a parallel molecular dynamics program. Automatic analysis suggested how the performance of the program could be improved. The analysis was verified using a model; however since the basic algorithm for EGO has been determined to be non-scalable in terms of its memory requirements, the future development and integration of the analysis has been suspended.

The first two applications are chare based applications and use load balancing libraries, while the last two applications are branch office chare based applications. Together, they demonstrate the usefulness of the performance analysis tool for irregular computations, such as traveling salesman problem and for regular computations, such as molecular dynamics. In the remainder of this chapter, we discuss our experiences with Projections for each one of the above applications.

7.1 Traveling salesman problem

The initial analysis for this problem was carried out with the earliest version of Projections, which only provided visual feedback about performance information specific to the Charm execution model. Subsequently, the automatic analysis component was used on the final version of the load balancing algorithm that we developed.

The Traveling Salesman Problem (TSP) [82] is a typical example of an optimization problem solved using branch&bound techniques. In this problem, a salesman must visit n cities, returning to the starting point, and is required to minimize the total cost of the trip. Every pair of cities i and j has a cost C_{ij} associated with them.

We have implemented the branch&bound scheme proposed by Little, et al [83]. In Little's approach, one starts with an initial partial solution, a cost function (C), and an infinite upper bound. A partial solution comprises a set of edges (pairs of cities) that have been included in the circuit, and a set of edges that have been excluded from the circuit. The cost function provides for each partial solution a lower bound on the cost of any solution found by extending the partial solution. The cost function is monotonic, i.e., if S_1 and S_2 are partial solutions and S_2 is obtained by extending S_1, then $C(S_1) \leq C(S_2)$. Two new partial solutions are obtained from the current partial solution by including and excluding the "best" edge (determined using some selection criterion) not in the partial solution. A partial solution is discarded (pruned) if its lower bound is larger than the current upper bound. The upper bound is updated whenever a solution is reached.

In the Charm implementation of the branch&bound solution of TSP, each partial solution is represented by a chare and the cost of the partial solution is the priority of the new-chare message. A monotonic variable is used to maintain the upper bound. We term as *useful* messages all new-chare messages with cost less than the cost of the best solution, and as *useless* messages all new-chare messages with cost greater than the cost of the best solution.

Figure 7.1 shows results of execution runs of a 40-city TSP on a shared memory machine, the Sequent Symmetry. The information is presented in terms of speedups and the number of nodes (of the branch&bound tree) that are generated during the computation. A look at the figure shows that the number of branch&bound nodes generated remains almost constant in all the runs, and the speedups are close to linear.

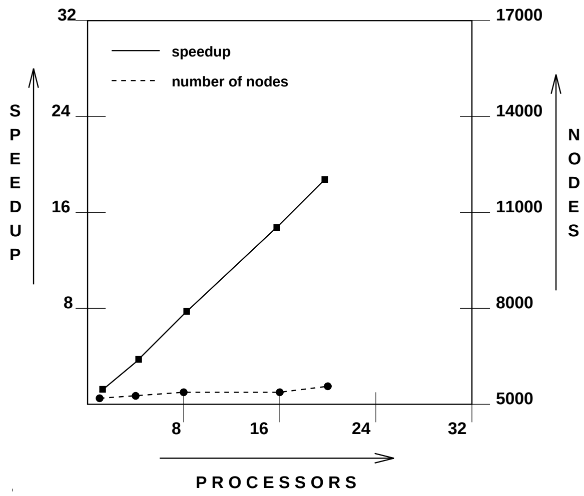

Figure 7.1: Speedups and the number of nodes generated for executions of an asymmetric 40 city TSP on the Sequent Symmetry.

The TSP application was executed on 16 processors of an NCUBE/2 with the ACWN [84] (adaptive contracting within neighborhoods) load balancing strategy. Figure 7.2 shows the result of the execution of a 40-city asymmetric TSP problem on an NCUBE/2 with the ACWN load balancing strategy.

Notice that we get nearly linear speedups in the case of the shared memory machine runs, while in the case of the nonshared memory machine runs (with either load balancing strategy) the speedups seem to saturate after 8 processors. Figure 7.3 show overviews of new chare creation and processing over stages for the ACWN.

In Figure 7.3, note that even after the solution was found at about 14.4 seconds, many new chare messages were still created. In our implementation, we prune at creation all *useless* messages. Therefore these new-chare messages could be created only if there remained in the

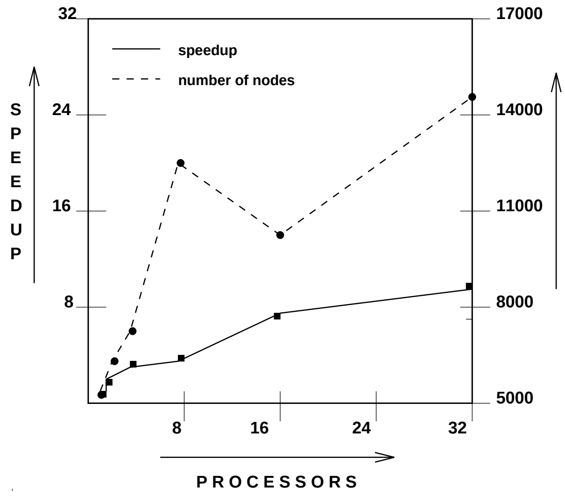

Figure 7.2: Speedups and number of nodes generated for an asymmetric TSP problem on the NCUBE/2 using the ACWN load balancing strategy.

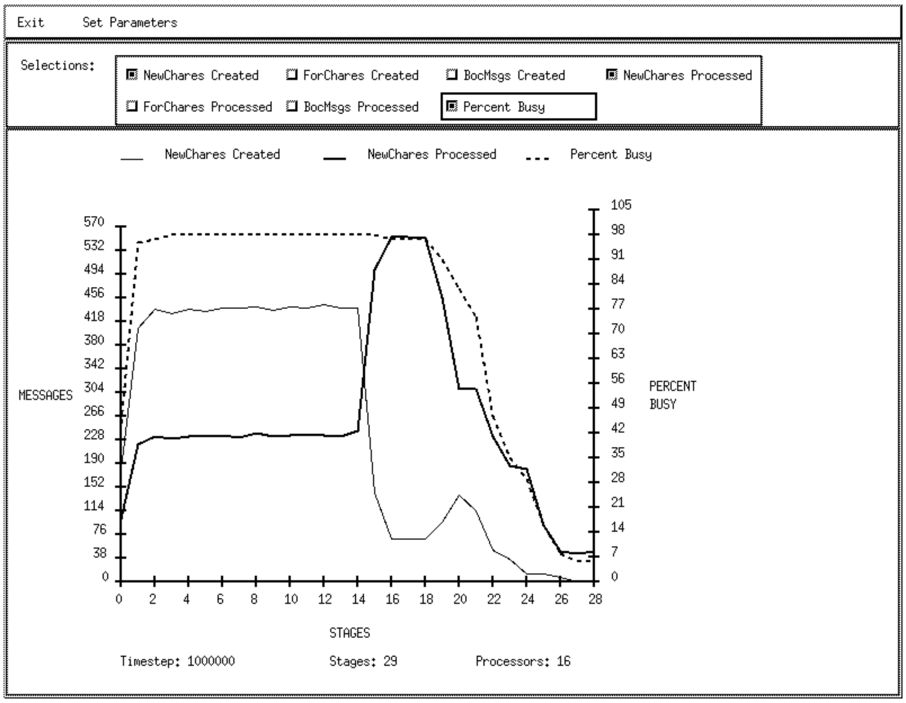

Figure 7.3: Overall efficiency and number of chares created and processed over the execution of a branch&bound implementation of TSP on an NCUBE/2 with the ACWN strategy.

system *useful* messages even after the best solution was found [1]. As the processor utilization is

[1] Note that in a traditional performance tool which shows sends and receives for messages, the plots would have looked considerably different, because they do not distinguish between the receipt of a message and its scheduling. In this case, the plots shows scheduling of messages, and therefore we are able to make this analysis.

close to 100% prior to this time, this must have happened because many *useless* messages were processed before the solution was found.

Why are the speedups good in the shared memory implementation? In the shared memory implementation, all processors share one priority queue of tasks. Therefore tasks are processed in the order of their priorities; consequently very little *useless* work is done, and the amount of speculative work is low. Since the total amount of work remains fairly constant even as the number of processors increase, and since all processors are busy 95% of the time the completion time is much faster.

Why are the speedups not good in the nonshared memory implementation? Since the average busy time for each processor is 80% we can eliminate longer idle times (as the number of processors grow) as a reason for poor speedups in the case of nonshared memory runs. In the nonshared memory implementation, tasks are distributed across all processors. Non-prioritized load balancing strategies do not balance priorities so that a lot of low priority messages (which may be pruned in an optimal execution) may get processed on some processors, while there are still high priority messages to be processed on other processors. This leads to a great deal of speculative work which manifests itself in the increase in the number of branch&bound nodes. In the case of both the random and the ACWN load balancing strategies, the number of nodes increases by almost 300%, and speedups were not good — even though there are more processors, there is more work (indicated by increase in number of nodes) to be done, hence the completion time does not decrease in proportion to the increase in the number of processors.

The above results indicate that in such speculative computations it is important that nodes be processed in the order of their priorities. Any efficient prioritized load balancing strategy should be able to ensure, as far as possible, that the processing of tasks occurs in the global order of their priorities. A simple measure of how well the load balancing strategy follows the above criterion is the variance in the number of nodes created with the number of processors — the lesser the variance, better is the criterion being adhered to, and vice versa.

What should be the nature of a load balancing strategy so that tasks are processed in their global order of priorities? Our experience with the centralized queue for tasks in the shared memory model versus the completely distributed queues for tasks in the case of nonshared memory models (using random and ACWN load balancing strategies) suggests that a prioritized load balancing strategy would perform better if it balanced load and priorities between *partially distributed queues*.

We outline the development of a prioritized load balancing strategy assisted by Projections. The first step towards the development of a good prioritized load balancing scheme was a centralized load manager strategy. Clearly this strategy would not scale well — the load manager would be a bottleneck. However, implementing and experimenting with this strategy allowed us to confirm the validity of the criterion mentioned earlier, and to determine the modes in which the bottleneck occurs.

7.1.1 First Step: Load Manager Strategy

In this strategy one processor is chosen as the load manager, the remaining processors are its managees. Managees send all new work to the load manager. The load manager is responsible for the buffering of new work in a prioritized queue and assigning loads to each of its managees. A managee keeps the load manager informed about its load status in two ways — first, by periodically sending load information to the load manager, and second, by piggybacking load information onto each piece of new work sent to the load manager. The strategy that the load manager adopts in distributing load among its managees is to maintain the load on every managee within a minimum and maximum allowable load range. Whenever a load manager receives new load information about a managee, it sends it work only if the current load on the managee is less than the minimum load — we define the minimum acceptable load on a processor as the *leash* size. The leash is used to keep managees busy with work, while the manager sends it more work. We had expected that varying the leash size would make a difference. However in all our experiments any leash size of greater than 1 performed equally well. One of the reasons for this might be that the average granularity of work for the 40-city case is about 0.8 seconds, and this might be sufficient to mask the idle time needed for a managee to request work from the manager.

Figure 7.4 shows the results of the execution of a 40-city TSP with the Load Manager strategy. Notice that the number of nodes have remained fairly constant, and the speedup is almost linear. The execution with the Manager strategy took 21 seconds, and a total of 5785 partial solutions were generated. The optimal solution was found at 9.6 seconds. Figure 7.5 show overviews of new chare creation and processing over stages for the Manager case. Note

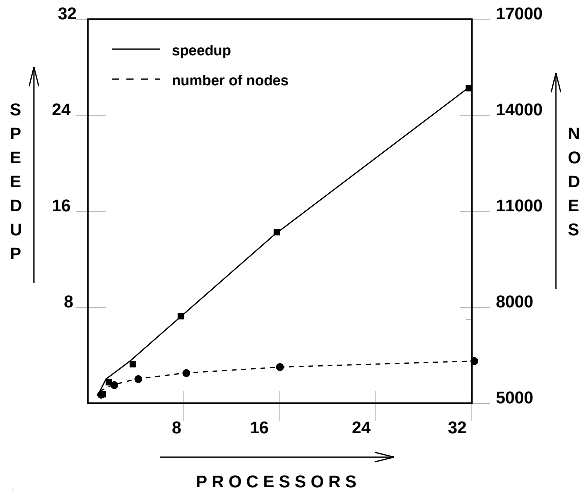

Figure 7.4: Speedups and the number of nodes generated for executions of a 40 city asymmetric TSP on the NCUBE/2 using the load manager strategy.

that very few new-chare messages are *created* after the best solution is found indicating that the load balancing strategy did a good job of balancing both the load and priorities of new-chare messages. The Load Manager strategy works well up to 32 processors, but its primary drawback is that it is not scalable to many more processors. In fact, it failed to run for the problem at hand for 64 processors. The failures were due to too many messages per unit time, which lead to an overflow of the system message buffer.

This motivated the next stage in the development of a multi-level prioritized load balancing strategy: the multiple managers strategy. Multi-level strategies have been studied before. Furuichi et al [85] present a strategy to partition the search of an OR-parallel graph in a distributed and hierarchical fashion among various processors — some processors function as sub-task generators and distribute the tasks among the remaining processors. One critical issue in their strategy is the generation of sub-tasks of reasonable granularity — if the grainsize is small, then the overheads of distributing would be substantial, if the grainsize is too large, then there might not be enough work to distribute. The model of computation in [85] is different from ours: in their model tasks are generated at and divided by only the task-generators, while in ours tasks can be generated at any managee. Ahmad and Ghafoor [86] have presented a semi-distributed strategy for task allocation for regular topologies, such as hypercubes, as an alternative to completely centralized and completely distributed task allocation strategies.

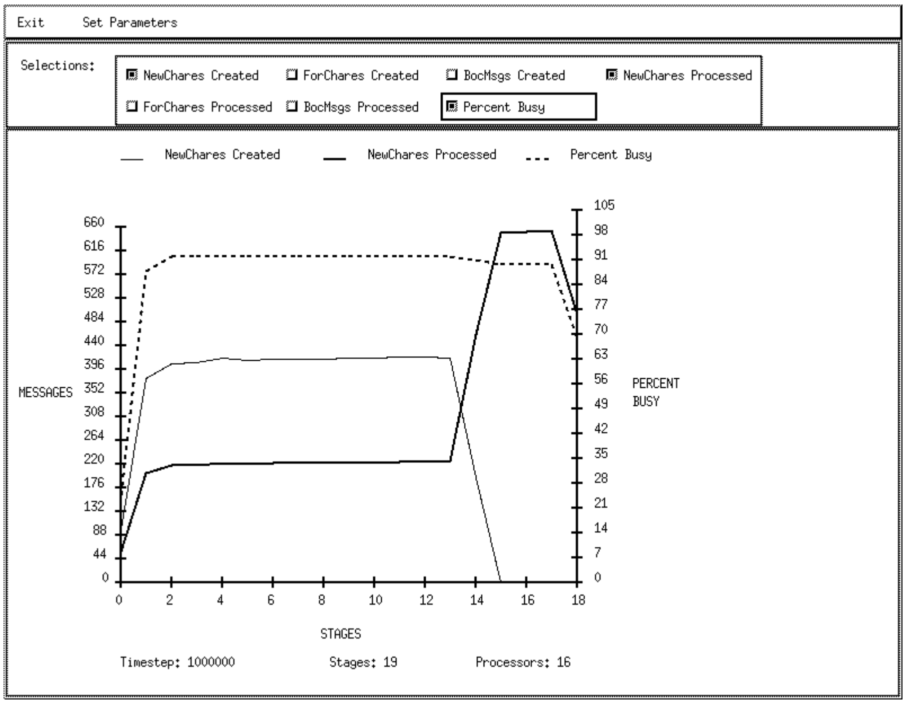

Figure 7.5: Overall efficiency and number of chares created and processed over the execution of a branch&bound implementation of TSP on an NCUBE/2 with the Manager strategy.

Neither of the above strategies take into account the additional factor of balancing priorities of tasks over processors.

7.1.2 Second Step: Multiple Managers Strategy

In the multiple managers strategy, the processors in the system are partitioned into clusters. One processor in each cluster is chosen as the load manager, the remaining processors in the cluster being its managees. Managees send all new work created on themselves to their corresponding load manager. Each load manager has two responsibilities:

1. It must distribute the work among its managees. As in the load manager strategy, the managees inform their load managers of their current work load by sending periodic load information and piggybacking load information with every piece of new work they send to the manager. The manager uses load information from its managees to maintain the load level within a certain range for all its managees.

2. It must balance both load and priorities over all the load managers in the system. This is accomplished by an exchange of high priority tasks between pairs of managers. Each manager communicates with a defined set of neighboring managers — in our implementation the managers were assigned positions in an n-dimensional cube, and the neighbor relation was defined as neighbors in the cube. An exchange of tasks between a pair of managers occurs in two steps. In the first step, the managers exchange their load information. In the second step each manager sends over some tasks to the other manager. Even if the loads are balanced, the managers exchange a fixed number of high priority tasks — this does the priority balancing. Further, if the loads are unbalanced, the manager with greater load sends to the manger with the lesser load additional tasks — this does the task-load balancing. Note that the tasks exchanged are the highest priority tasks on each manager. We have experimented with a strategy in which one half of the top priority tasks were exchanged, but this resulted in a degradation in performance, perhaps because of the cost of determining the top half elements. We can intuitively explain why exchanging the top priority tasks might be sufficient: the managees of each manager are already working on the top priority elements on their load managers, which are likely to generate new high priority tasks. Therefore an exchange of work between managers causes a distribution of the top priorities between two managers and their managees.

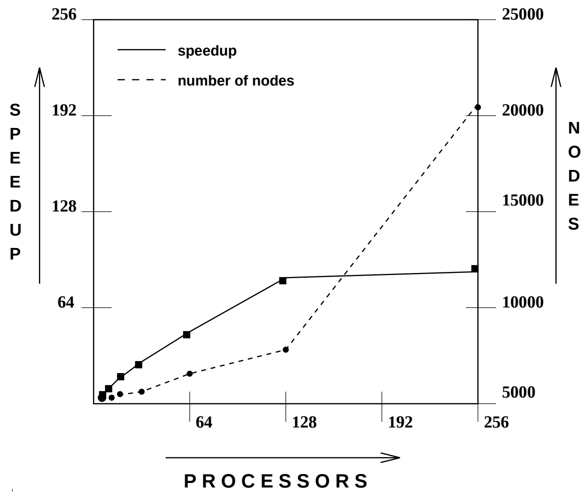

Figure 7.6: Speedups and the number of nodes generated for executions of a 40 city asymmetric TSP on the NCUBE/2 using the multiple managers strategy. In this case the cluster size is 8 processors.

Figure 7.6 shows the results of runs of a 40-city TSP with a multiple managers load balancing strategy. The speedups were good for 128 processors, but thereafter the number of nodes increased sharply, and the speedup remained unchanged. One of the reasons might be that there was not enough new work at the managers [2]. In that case the priority balancing would not be effective causing expensive nodes to be processed early. In the example in Figure 7.6 approximately 7500 nodes were expanded for 128 processors, which works out to 50 nodes per processor for the 128 processor case and only 25 nodes per processor for the 256 processor case for the entire duration of the computation, which does seem to be a small number of nodes on each processor. In order to confirm our analysis we ran the TSP program for a larger problem size. The results for the 50-city run of the TSP are shown in Figure 7.7. Actual times are provided, instead of speedups, because the program did not run on a single processor due to insufficient memory. The results show better speedups till 256 processors and confirm our analysis.

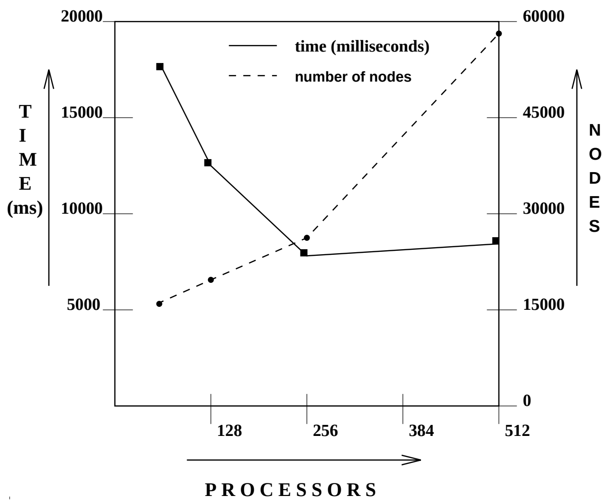

Figure 7.7: Speedups and the number of nodes generated for executions of a 50 city asymmetric TSP on the NCUBE/2 using the multiple managers strategy. In this case the cluster size is 16 processors.

The multiple managers strategy scales up reasonably well to 256 processors. Could we have used larger problems and obtained speed-ups for even more processors? Probably, yes. However, we ran out of memory for larger problem sizes. The reason for this is that there is an

[2] Note that the automatic analysis component of Projections would have identified this problem using the degree of parallelism analysis.

imbalance in the memory requirements of the load managers and the managees in the multiple managers strategy. The imbalance arises because all newly created work is queued up at the load managers. This poses problems because the amount of new work that can be created becomes limited by the number of managers and their available memory, even though there is a larger amount of memory available on the managees (assuming all processors in the system have equal amount of memory, and that there is more than one managee for each manager). Our final load balancing strategy attempts to balance the memory requirements of the load manager and the managees.

7.1.3 Third Step: Token Strategy

The token strategy is very similar to the multiple managers strategy. The processing elements in the system are split up into clusters — one processor in each cluster is chosen as the load manager, the remaining processors are its managees. New work created on managees is stored in hash-tables on the processor itself, while a token containing the priority of the new work is sent to the load managers. The load managers balance tokens and priorities among themselves by exchanging their high priority tokens — a fixed number of tokens is always exchanged to accomplish priority balancing, while some more tokens may by exchanged to balance the number of tokens on the load managers. Each managee informs its manager of its load by (1) piggybacking load information with each token it sends to the manager, and (2) periodically sending load information. When a manager decides that one of its managees (say M) needs work, it selects the highest priority token on it, and sends a request to the processor that generated (and stored) the work corresponding to the token asking for the work to be sent to M.

There were two problems that we anticipated with the token strategy:

1. In the manager and the multiple managers strategies, new work traveled two hops — one hop for the work to be sent to the manager and another hop for the work to be sent from the manager to a managee (ignoring the number of hops a message might take because of load balancing). In the token strategy, each piece of new work causes three hops — one hop for the token to be sent to the manager, one hop for the manager to send managee a request to send work, and a third hop for the work to be sent.

2. There may be a delay in a processor responding to a request to forward work it owns because it may be busy executing user work, which is non-preemptible in the Charm execution model [3].

We had hoped that the second problem could be mitigated if we managed the leash size so that the managees had some work to do, while work was being sent to them — however, experimental results indicate that any leash size of greater than 1 performed equally well.

Figure 7.8 shows the execution times and the number of nodes generated for runs of a 50-city asymmetric TSP on the NCUBE/2 with the token strategy. The results are comparable to those obtained with the multiple managers strategy. The advantage of using tokens in the token strategy case was countered by the disadvantage of each message traveling three hops, compared to two hops for the multiple managers strategy. The token strategy, however, is superior when it comes to solving larger problems because it utilizes the available memory much more efficiently.

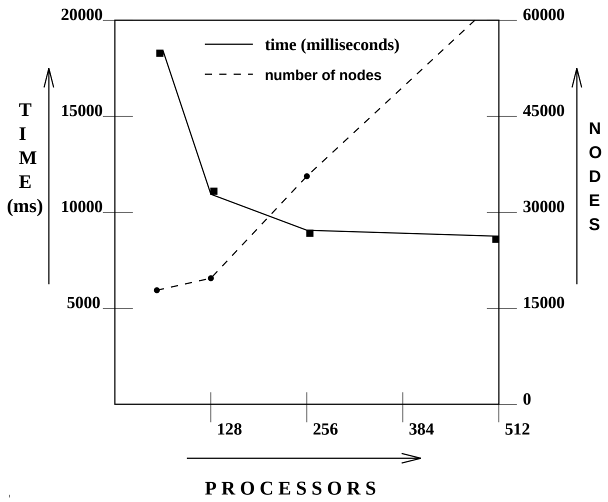

Figure 7.8: Execution times and the number of nodes generated for executions of a 50 city asymmetric TSP on the NCUBE/2 using the tokens strategy to balance load. In this case the cluster size is 16 processors.

Figure 7.9 shows the execution times and the number of nodes generated for runs of a 60-city asymmetric TSP on the NCUBE/2 with the token strategy. Notice that the number of nodes generated in this case are fairly constant for up to 512 processors and the speedups are good.

[3] Newer versions of Charm circumvent this problem on some machines by using a system level interrupt.

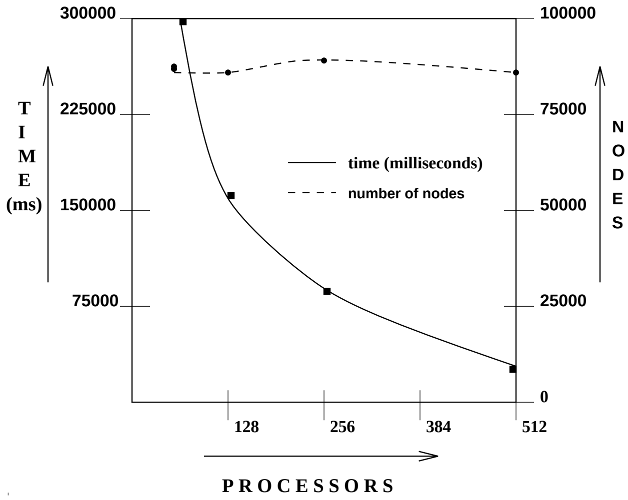

Figure 7.9: Execution times and the number of nodes generated for executions of a 60 city asymmetric TSP on the NCUBE/2 using the tokens strategy to balance load. In this case the cluster size is 16 processors.

How good is the token strategy? The results of execution runs of the 60 city asymmetric TSP seem to indicate that the number of nodes created remains nearly constant with the number of processors. Is there any room for further improvement? In order to answer these questions we need to examine two quantities: the amount of wasteful work done and the fraction of time spent waiting for new work by each processor.

We have attempted to estimate the amount of wasteful work done by determining the distribution of nodes in terms of cost over time of the nodes created and processed in the execution runs of the 60-city TSP problem. Table 7.1 shows the number of *useful* and *useless* nodes created and processed for various stages of the program execution for two separate runs of the 60 city asymmetric TSP. In the first run, A, the initial upper bound is selected to be infinite, while in the second run, B, the initial upper bound is selected to be one unit greater than the cost of the best solution. In our implementation, we prune at creation all nodes with cost greater than the current upper bound. Since the initial upper bound is one unit greater than the cost of the best solution, no *useless* nodes are created in run B. Note that in Table 7.1 the number of *useless* nodes created in Run B are more than zero. This is because we have counted nodes whose cost equals the cost of the best solution as *useless* nodes.

An examination of the distribution of the number of nodes processed before the best solution in case of Run A shows that the number of *useless* nodes processed are very few, and the

Nodes Created				Nodes Processed			
Before Soln.		**After Soln.**		**Before Soln.**		**After Soln.**	
Useful	**Useless**	**Useful**	**Useless**	**Useful**	**Useless**	**Useful**	**Useless**
43259	40659	546	1236	41788	141	2094	41748

Initial Upper Bound: INFINITY (999999)
Time first solution was found: 118424 ms
Time finished: 155809 ms

(A)

Nodes Created				Nodes Processed			
Before Soln.		**After Soln.**		**Before Soln.**		**After Soln.**	
Useful	**Useless**	**Useful**	**Useless**	**Useful**	**Useless**	**Useful**	**Useless**
43585	6025	371	242	42081	3	1880	6260

Initial Upper Bound: 826 (optimal)
Time first solution was found: 117806 ms
Time finished: 124831 ms

(B)

Table 7.1: Number of useful/useless nodes created/processed, before/after the best solution was found.

number of *useful* nodes are substantially more. This means that very little wasteful work is done before the best solution is found. However, the number of *useless* nodes processed after the best solution is found is considerable — is this wasteful work? The answer is no, because these nodes were created before the solution was found when the upper bound in this case was infinite, and they are simply being picked up and discarded after the best solution was found. This analysis is confirmed if we repeat the above run with an initial upper bound which is one greater than the cost of the best solution. In this case (Run B), the distribution of costs of nodes processed before the solution was found look very similar to the results in Run A. However, the number of *useless* nodes processed after the solution was found are substantially less than the corresponding number in Run A — much fewer *useless* nodes are created in Run B, because our implementation prunes all *useless* nodes at creation time. This reduction manifests itself in a faster finish time, though the time to find the first solution remains virtually unchanged. The time spent in wasteful work before the best solution is found is a small fraction ($< 1\%$) of the time spent in doing useful work. The next quantity — fraction of time spent in waiting for work by processors — was determined to be an average of less than 6% for each one of the managees. These two quantities indicate that the token strategy performs fairly well, though some small improvement might be possible.

7.1.4 Fourth Step: Aperiodic Strategy

Later, when the automatic analysis component of Projections was ready we ran it on the token strategy for a 20 city TSP problem on a network of 5 workstations. The analysis, shown in Figure 7.10, indicated the following problems existed with the load balancing strategy:

1. The load balancing strategy was not performing effectively, i.e., even though there was work available, processors were still idle.

2. As a result of poor load balance, many processors were idle, which resulted in the quiescence detection algorithm being turned on excessively. This happens because the quiescence detection algorithm is adaptive, and becomes active only when processors become idle. Since the load balance was not adequate, it left many processors idle. Therefore, the quiescence detection algorithm became unnecessarily active.

A look at the creation and processing of new chare messages in the overview, shown in Figure 7.11, showed the cause of the problem. The priority balance seemed to work, because no new chares (with low priorities) were created after the initial burst. However, the performance of the load balancing scheme is poor after the solution is found. The reason for this poor performance is the periodic component built into the strategy. A managee has two ways of letting the manager know of its status:

1. Each managee periodically sends its status to the manager,

2. Load status is piggybacked onto each message carrying a token sent to the manager.

Upon receiving status update from a node, the manager sends the managee work if its load is poor. In the final stages of the algorithm, when the solution has been determined, no new chares are generated. Therefore the only mechanism available for a managee to inform its manager of its status is the periodic message. After the solution has been determined, work on the managees is of small granularity, because it consists mostly of examining and discarding messages. Thus a managee finishes its work quickly and waits until its periodic message is sent

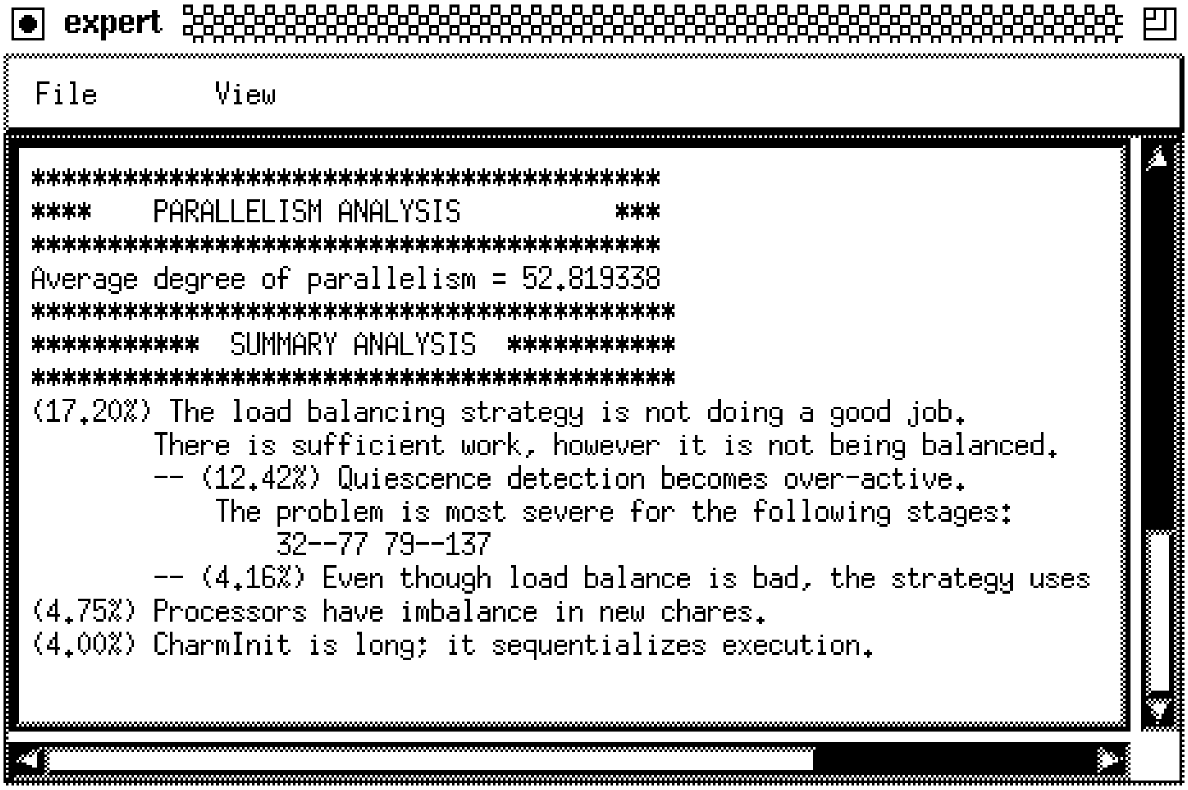

Figure 7.10: Analysis for the execution of a 20-city TSP problem on a network of 5 workstations using the token strategy.

to the manager to receive more work. This explains the peaks of processing after the solution has been found.

In the new *aperiodic* strategy, we eliminated periodic status updates. Instead, now status information is sent to the manager whenever the load becomes poor on a managee. As a result, the best solution in the 20 city case is determined in 1930 milliseconds and the entire branch&bound tree is searched in 3100 milliseconds. In comparison, for the periodic version of the token strategy the best solution in the 20 city case was found in 2360 milliseconds, and the entire branch&bound tree was searched in 5060 milliseconds.

Figure 7.12 shows the creation and processing of the entry point for branch&bound. Notice that now the execution finishes soon after the best solution is found (approximately where the creation graph dips sharply). Also notice that the branch&bound part of the computation seems to occur more at the end now. The reason for this, as confirmed by the analysis in

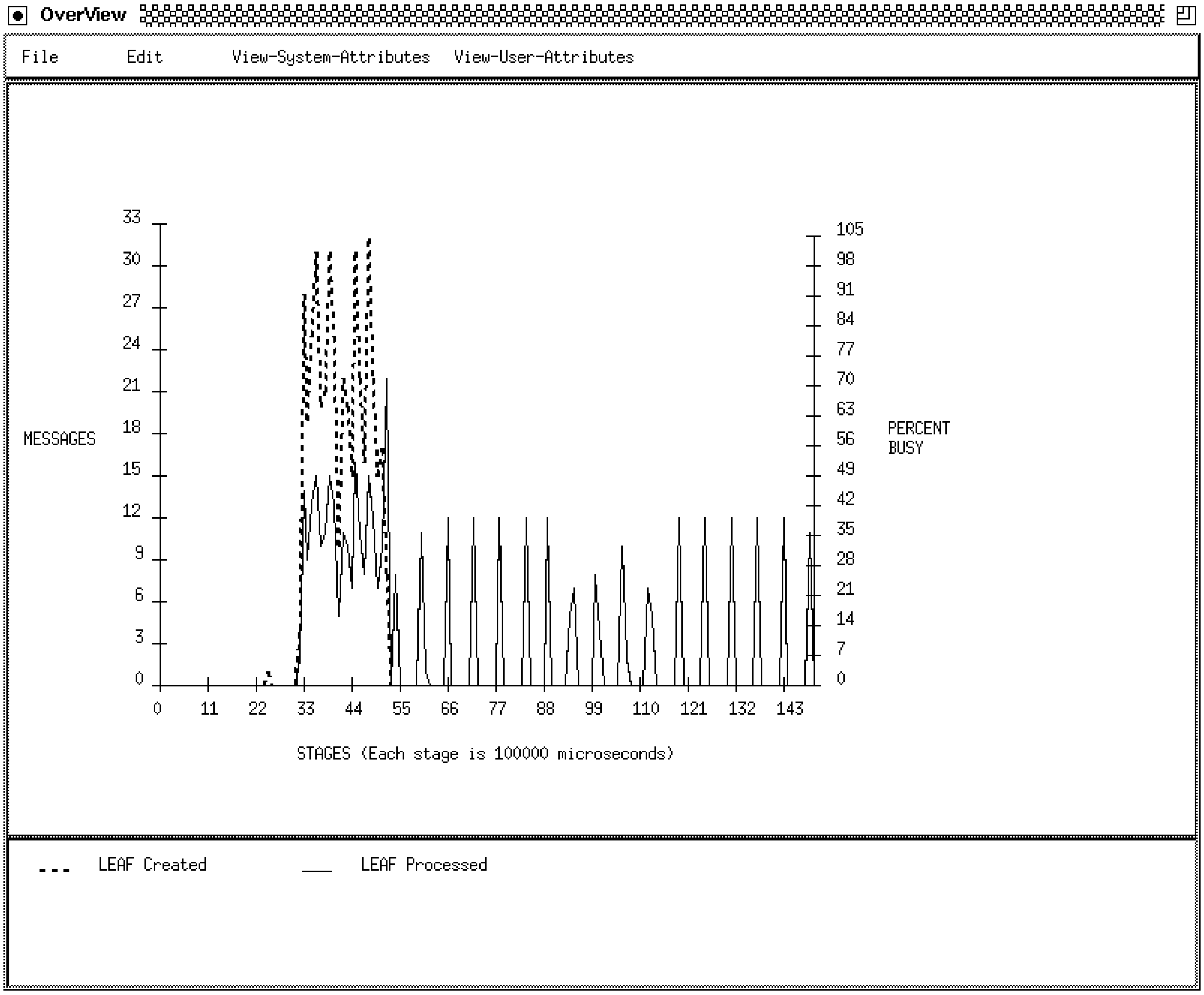

Figure 7.11: Number of branch&bound nodes created and processed in the execution of a 20-city TSP problem on a network of 5 workstations using the token strategy.

Figure 7.13, is the *CharmInit* entry point. The input data is read in the *CharmInit* entry point, and as pointed in the analysis in Figure 7.13 that accounts for a large performance loss.

In Section 7.1.3, we saw that the multiple managers strategy out-performs the token strategy for certain problems. We attribute the superior performance of the multiple managers strategy to the fewer hops taken by each message. Some of the delays due to extra message hop in the token strategy can be reduced by caching work corresponding to the best tokens on the managers. We would to like to investigate the effect of caching high priority nodes on the managers on the performance of the token strategy.

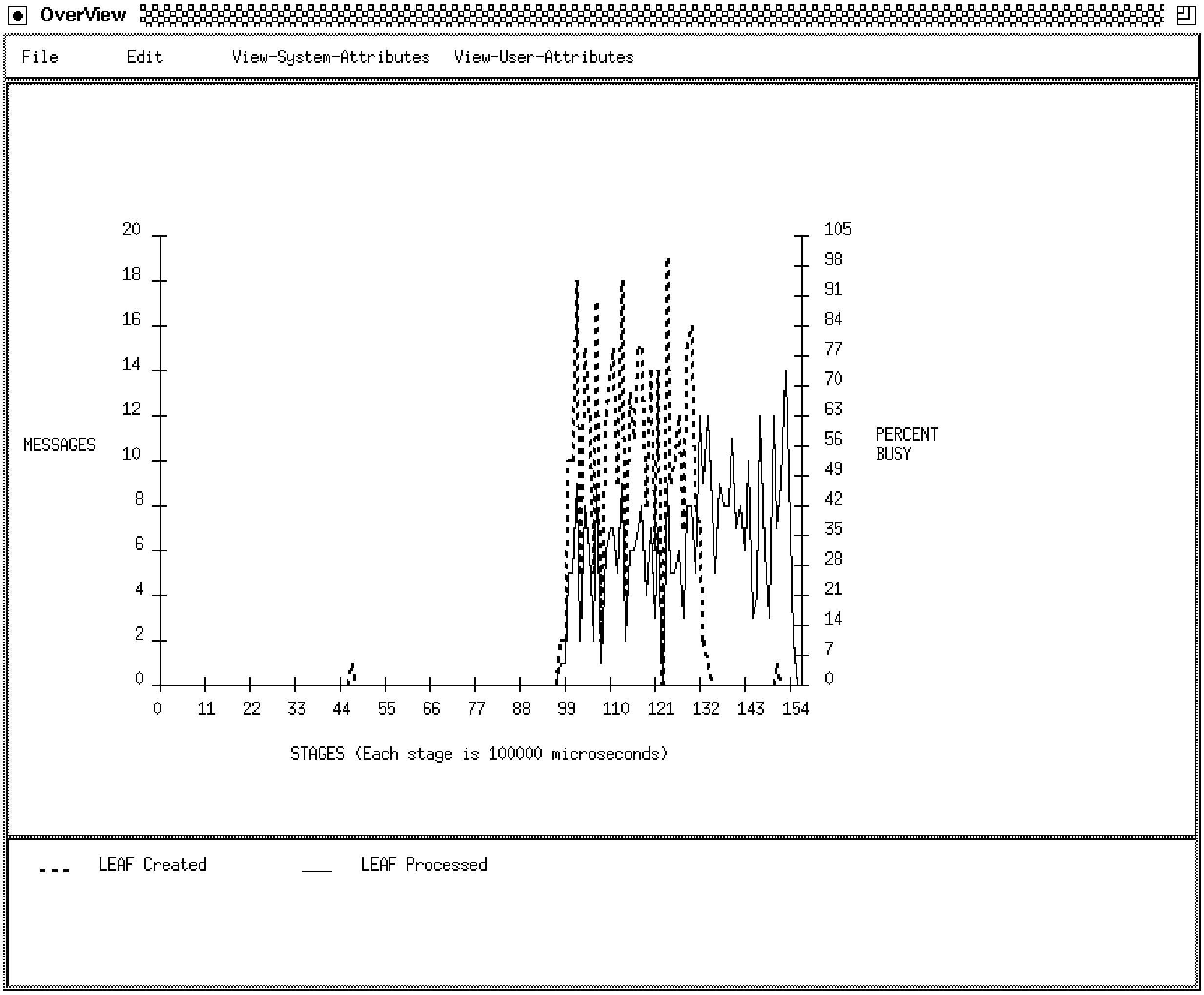

Figure 7.12: Number of branch&bound nodes created and processed in the execution of a 20-city TSP problem on a network of 4 workstations using the aperiodic token strategy.

7.2 Multiple linear solver

We considered an application which tries to solve n independent sparse penta-diagonal systems using the Gauss-Siedel (red-black) iterative method [7]. Such computations arise in unsteady fluid flow calculations. Since the systems are different (and have different boundary conditions), they converge at different rates. In the Charm implementation, the solutions of all the n systems were carried out simultaneously — this was done so as to exploit the possibilities of overlap provided by message driven execution. The solution of each system goes through multiple iterations. In each iteration, a processor exchanges data with its neighbors, computes upon the

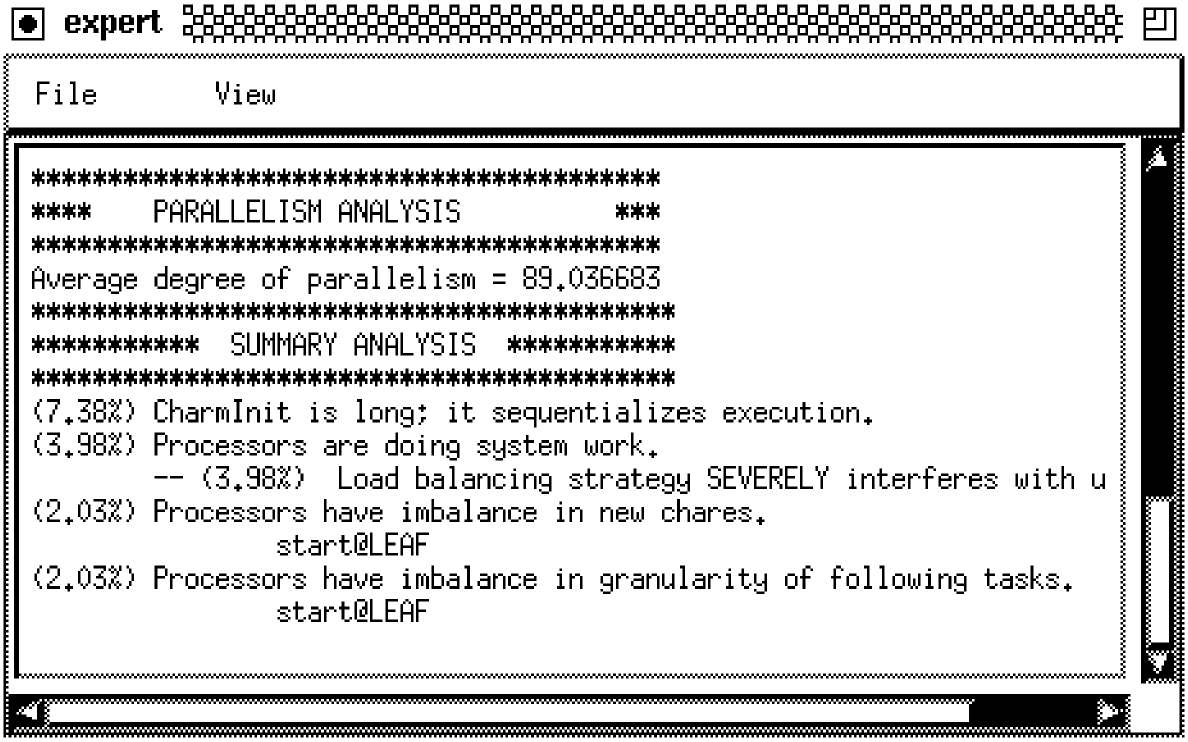

Figure 7.13: Analysis for the execution of a 20-city TSP problem on a network of 4 workstations using the aperiodic token strategy.

data it has received, participates in a global reduction on the new values, and then starts the next iteration on receiving the result of the reduction.

Figure B.1 shows a picture of some of the Projections views of the program's execution trace. The expert analysis informed us that the critical path analysis indicated improvement if certain entry points lying on the critical path were prioritized. However even though many entry points were listed on the critical path, only two were listed as potentials for prioritization. The timelines for this trace provide an explanation for this analysis.

Figure 7.14 shows timelines for stage 7 (one of the first stages of the execution of the program) and stage 35 (one of the last stages of the execution of the program). The timeline on the left corresponds to the former, and the timeline on the right corresponds to the latter. The dark-blocks are reduction phases in the execution. The timeline on the left shows that the processors continue solving other systems, while reduction is being carried out for one system. Consequently, there is high degree of overlap in stage 7 (and most of the stages during the beginning stages of execution). In the timeline on the right, one notices that processors idle

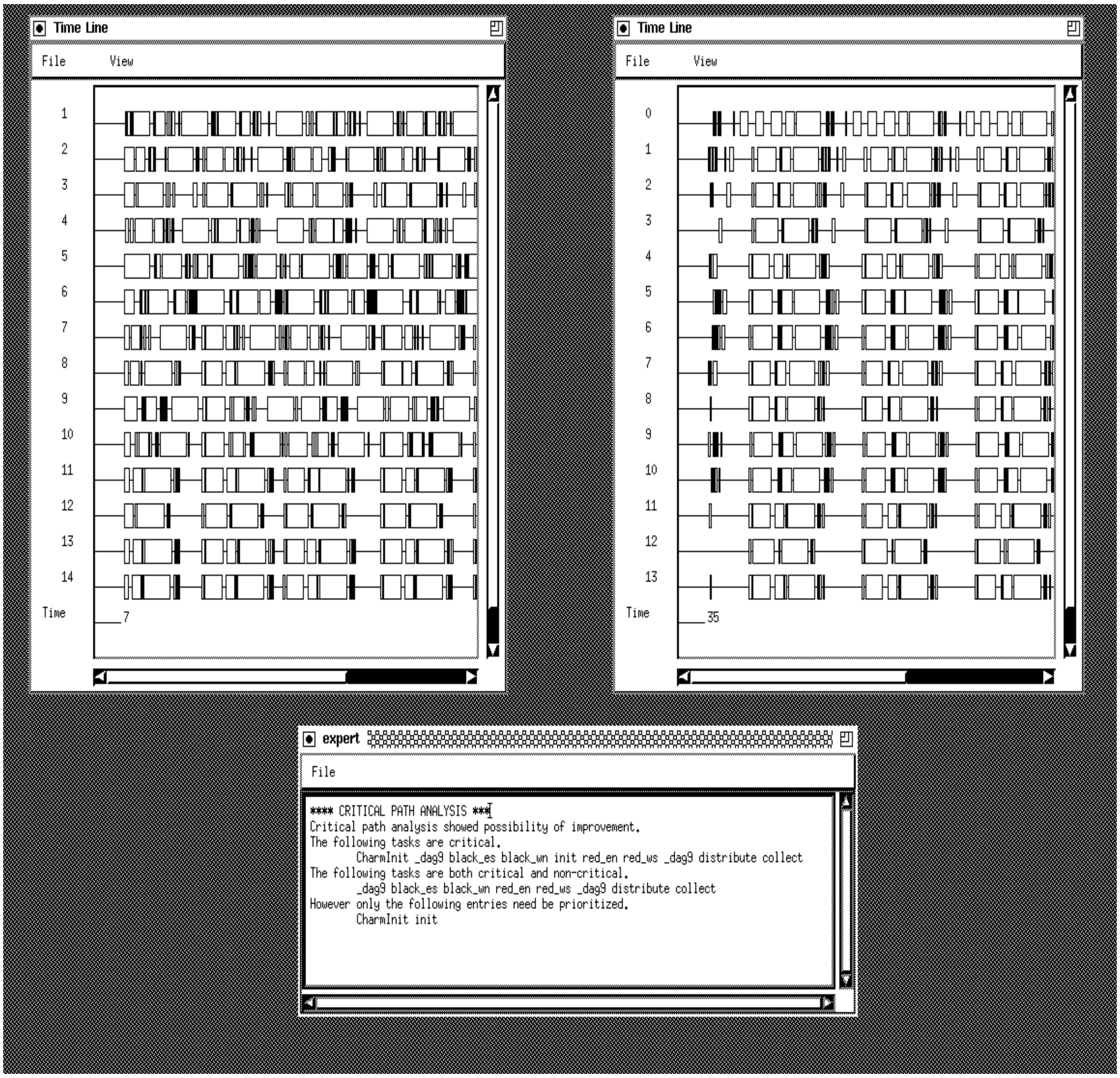

Figure 7.14: Timelines for some processors for stages 7 and 35 of the program execution.

while the reduction is being carried out. This occurs because only one system remains to be solved at the very end, and therefore there is no possibility of overlapping the reduction with the solution of other systems.

Critical path analysis, therefore, correctly identified the critical tasks (the ones solving the last system); however the same tasks were also identified as non-critical. This occurs because the program uses the same object and the same entry points to solve all the systems. Therefore the analysis could not suggest an effective prioritization scheme on its own — *CharmInit* and *Init* are initialization entry methods and are executed only once.

This example illustrates the usefulness of critical path analysis. It also illustrates that the limitations of any expert analysis comes primarily because the user often re-uses code blocks for different portions of the computation, which the analysis tool needs to comprehend in order to be able to completely analyze the program. This could be overcome if the tool had additional information of the structure of the object itself.

Of course, automatic analysis assists manual analysis in such cases, too. In this case, it informed the user that the critical path was the problem and identified the specific entry points on the critical path.

7.3 Matrix multiplication

In this section, we will illustrate the use of automatic analysis with a simple example: matrix multiplication. There are many different and sophisticated matrix multiplication algorithms. For simplicity, we have chosen the basic $O(n^3)$ algorithm. In the parallel implementation, the result matrix (AxB) is divided into rectangular blocks, and each block is computed in parallel. The computation of each block will need multiple rows of A and multiple columns of B.

Figure 7.15 shows a Charm implementation of a dynamic formulation of matrix multiplication. The chare mult computes a rectangular block of size row_grain x col_grain. Since the size of each rectangular block is the same, the number of floating point operations computed by each chare is exactly the same. However, a Charm program can be executed on a network of workstations, where the differences in the speeds of the individual workstations can cause even uniform tasks to take non-uniform amounts of time. Therefore this program employs dynamic load balancing. But, a load balancing strategy that dynamically places chares on processors as the program executes makes it impossible to estimate the sets of rows of A and columns of B that would be needed on a particular processor. A trivial solution would be to replicate matrices A and B on all processors: the obvious problem with this approach is that the program is not scalable to large input matrices. The distributed table abstraction can be used to solve this problem. The rows of A and the columns of B are stored, using the Insert call, as entries in distributed tables, row_table and col_table, respectively. Figure 7.15 shows the

```
table row_table, col_table;
readonly int rows, cols, row_grain, col_grain;

chare main {
    entry CharmInit: {
    for (i=0; i<rows/row_grain; i++) {
        get_row(i, row_grain, row);
        Insert(row_table, i, row); }
    for (i=0; i<cols/col_grain; i++) {
        get_column(i, col_grain, col);
        Insert(col_table, i, col); }
    for (i=0; i<rows; i+=row_grain)
        for (j=0; j<cols; j+=col_grain) {
            msg = (MSG *) CkAllocMsg(MSG);
            msg->row_index = i;
            msg->col_index = j;
            CreateChare(mult, mult@start, msg);
} } }

chare mult {
    entry start: (message MSG *msg) {
        recd_rows = recd_cols = 0;
        index = msg->row_index;
        Find(row_table, index, row_cont, me);
        index = msg->col_index;
        Find(col_table, index, col_cont, me); }
    entry row_cont: (message TBL_MSG *msg) {
        recd_row=1; store_row(msg->data, row); PrivateCall(continue()); }
    entry col_cont: (message TBL_MSG *msg) {
        recd_col++; store_col(msg->data, col); PrivateCall(continue()); }
    private continue() {
        if ((recd_row && recd_col)) {
            dot_product(result, row, col);
            index = row_index*cols+col_index;
            Insert(result_table, index, result);
} } }
```

Figure 7.15: A matrix multiplication implementation.

matrices A and B being inserted in the CharmInit entry method of the main chare. Each entry of the distributed table is a user-specified number of rows/columns; in this case each entry is either row_grain number of rows or column_grain number of columns. The chare mult is used to compute a block of the result matrix. When it is first created (start entry point), the chare uses the *Find* call to get the necessary rows and columns. Computation of entries in the particular block of the result matrix is done when both the rows of A and the columns of B have arrived. Once the result is available, it is inserted into a distributed table result_table from where it can be accessed in a subsequent phase of computation.

Machine	Time (ms)
CM-5 (64 pes)	2425
Network (4 pes)	47730

Analysis: poor granularity.
Solution: increase granularity.

(A)

Machine	Time (ms)
CM-5 (64 pes)	660
Network (4 pes)	20750

Analysis: imbalance of table entries, duplicate accesses, imbalance of table inserts.
Solution: Use better hash function for balance

(B)

Machine	Time (ms)
CM-5 (64 pes)	220
Network (4 pes)	15600

Analysis: imbalance of table inserts.
Reason: Inserts are imbalanced because they all occur in *CharmInit*.

(C)

Table 7.2: Sequence of results for different versions of the matrix multiplication implementation as analysis is applied and the program is changed to account for various analyses. All times are in milliseconds for multiplying two 256x256 matrices.

Table 7.2 shows the performance of different versions of our algorithm as the analysis was used to improve its performance. The initial results are shown in Table 7.2(a). The analysis informed us that the main problem was that the granularity of tasks (in particular, for the entry point that did the matrix multiplication) were too small. Our first modification was to run the program with larger granularity: this was done by letting each chare compute a bigger block of the result matrix. In this case the size of the block that each chare computed was increased from 4x4 to 16x16 (by changing row_table and col_table from 4 to 16). Table 7.2(b) shows the timings with this change. One interesting note to add is that the granularity of tasks for best performance was about the same on the CM-5 and the network of SUN workstations. This is not surprising because even though the CM-5's processor was about 8 times faster than the SUN workstation processor we were using, it was balanced by the CM-5 communication network which was 10 times faster than the SUN's ethernet connection.

The automatic analysis for the modified program showed that the main problems were imbalance of table entries, duplicate accesses, and imbalance of table inserts. In the next round of changes, we were able to make the distribution of table entries balanced across processors by choosing a better hash function for the tables. Further simultaneously enabling caching (using the caching system library provided in Charm) of shared variable accesses (to reduce duplicate accesses) resulted in the timings shown in Table 7.2(c). The times for the same program without enabling caching were almost the same: caching did not provide any real performance benefit in this program. However caching reduced the amount of redundant storage because of concurrent retrievals of the same shared variable on each processor; consequently larger problem sizes could be run.

The final analysis still showed some problems about the imbalance of table requests. This is so because all our inserts into the distributed tables, row_table and col_table, occurred in CharmInit, which is executed only on processor 0. For these small machine configurations this is not a significant problem. However in larger problems, it would be useful to distribute the inserts across all processors.

7.4 EGO: A parallel molecular dynamics algorithm

In molecular dynamics simulations, a large fraction of the computing time is spent in the evaluation of Coulomb and van der Waals forces, both involving $O(N^2)$ floating point operations, where N is the number of atoms in the molecule. The Coulomb forces, which describe the electrostatic interactions in a homogeneous dielectric environment depend on the charges q_i

and q_j of pairs (i,j) of atoms and on the corresponding vector $\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$ joining the atoms at positions $\vec{r}_i$ and $\vec{r}_j$. The force between atoms i and j acting on atom i is

$$\vec{F}_{ij} = \frac{q_i q_j \vec{r}_{ij}}{4\pi\varepsilon r_{ij}^3} \quad . \tag{7.1}$$

The evaluation of the remaining interactions, namely bonded, in case of large polymers, consumes only a small fraction of computer time. Further the van der Waals forces are computationally very similar to the Coulomb forces. Therefore, we examine only the evaluation of Coulomb forces.

Since the evaluation of long-range pair interactions is the computationally most demanding part of a simulation, this task needs to be accelerated. EGO employs two techniques to reduce the $O(n^2)$ operations needed to compute the electrostatic forces:

1. From Newton's law, the force between atoms i and j acting on atom j is

$$\vec{F}_{ji} = -\vec{F}_{ij} \quad . \tag{7.2}$$

   Using Equation 7.2, it is not necessary to compute both $\vec{F}_{ij}$ and $\vec{F}_{ji}$; therefore it is possible to cut down the computation time to about one half by avoiding redundant computations.

2. In order to gain a substantial speedup, EGO uses a new distance classing algorithm [87] to account for long-distance interactions. The algorithm classifies each pair of atoms into one of eight distance classes according to the distance between the atoms in the pair. It then evaluates the Coulomb forces for atom-pairs in the closer distance classes more frequently than the forces for atom-pairs in farther distance classes, thus avoiding unnecessarily frequent computations of interactions between particles which are far apart. The algorithm is closely related to the multiple-time scale (MTS) algorithm suggested in [88].

The interactions between any pair of atoms are mutually independent. Hence, the atoms of a biopolymer are distributed over the available set of processors, each processor computing the interactions corresponding to the set of atoms local to it. The atoms assigned to a specific processor will be referred to as the 'own' atoms of that processor, all other atoms are the 'external' atoms. For a discussion of the computational task of a processor we separate the

Coulomb force F_i on atom i into two contributions

$$\vec{F_i} = \sum_{\substack{\text{'own'} \\ \text{atoms } j}} \frac{q_i q_j \vec{r}_{ij}}{4\pi\varepsilon r_{ij}^3} + \sum_{\substack{\text{'external'} \\ \text{atoms } k}} \frac{q_i q_k \vec{r}_{ik}}{4\pi\varepsilon r_{ik}^3} \quad . \tag{7.3}$$

In order to evaluate the first contribution the processor needs to know only the coordinates of its 'own' atoms j. The second contribution, however, requires knowledge of the coordinates of all 'external' atoms k. Each individual term of the second sum is referred to as a partial force. These coordinates are passed around the processors in such a way that any time coordinates pass by, a processor uses them to complete computation of the total force $\vec{F_i}$ that acts on its 'own' atoms.

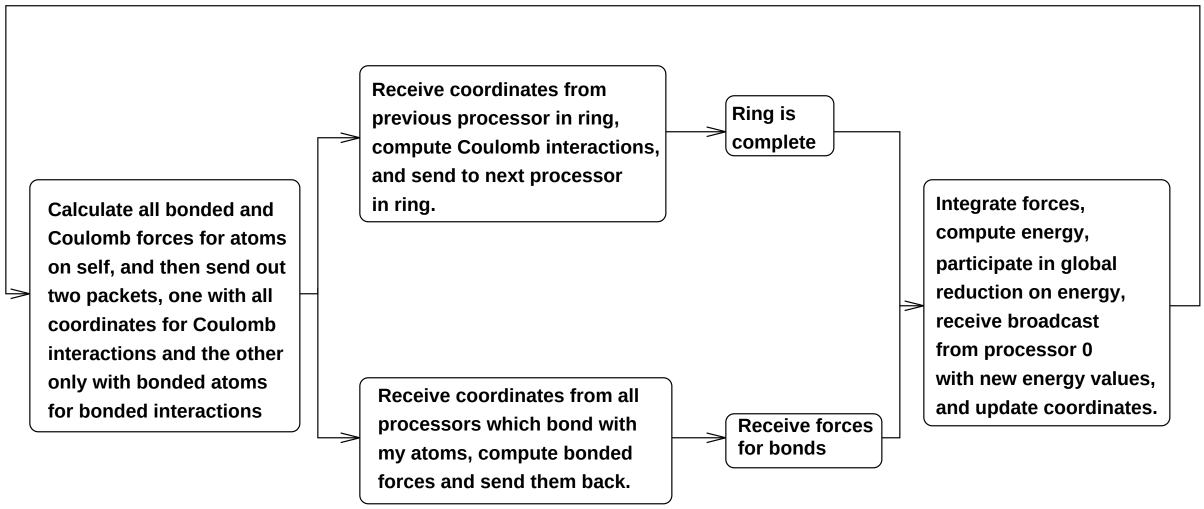

Figure 7.16: Program flow in EGO.

Figure 7.16 shows the computational structure of the EGO program. An integration step begins with each processor computing the electrostatic forces for atoms it owns. Subsequently, it sends out two messages: one containing coordinates of all its 'own' atoms to the next processor on the ring, and the other containing coordinates of its 'own' atoms for which their bonded partners lie on other processors. The first message goes around in a ring, such that each processor computes the pairwise interactions between its atoms and the atoms in the message. This computation ceases when the message returns to the originating processor. For the second message, each processor waits until it has all information about the bonds for its atoms. It then computes the bonded forces and sends them back to the processors whose atoms participate in the bonds. Once the bonded forces and the pairwise message arrive, each processor does a local integration to determine the local energies, followed with a reduction to determine the global energy, and finally receives the new energy in a broadcast from processor 0. The next integration step begins after each processor uses the new energy values to compute new positions.

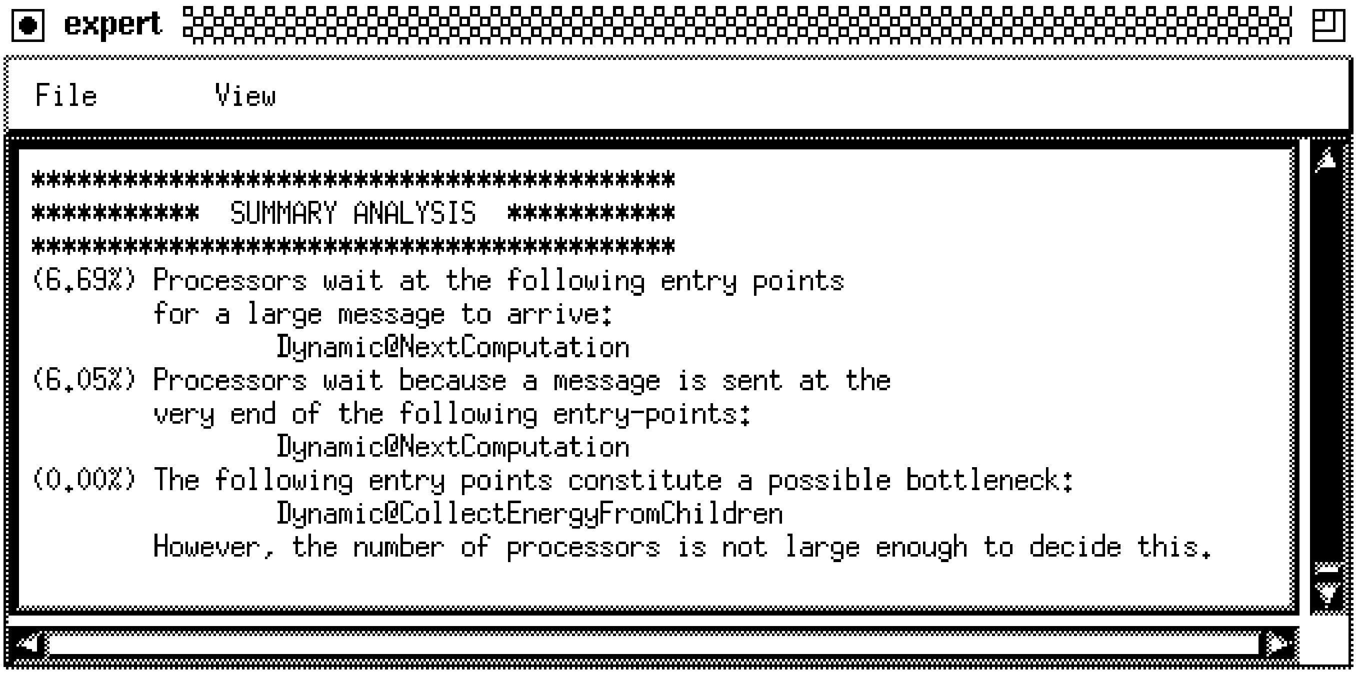

Figure 7.17: Analysis report for experimental model of EGO.

Our initial performance studies confirmed that most of work involved computing the electrostatic forces. In order to easily study the effects of various techniques on performance, we conducted our studies on an experimental model. In the experimental model, we stripped out all i/o and bonded computations, and simulated the computation and message sizes for only the Coulomb interactions. The analysis given by Projections for the model appears in Figure 7.17. Projection's analysis showed that the biggest performance problem was the large message being sent at the end of the entry point that computed the Coulomb interactions (*NextComputation*), and performance could be improved if the message could be sent earlier.

The message which arrives at the *NextComputation* point contains coordinates and forces for all the atoms of a processor earlier in the ring. Since the coordinates do not change, they can be sent to the next processor in the ring right away. However, the values of the forces in the message need to be updated to account for the forces due to the atoms on that processor. Therefore, the forces need to be sent out only after they are updated.

The first optimization involved breaking up the message that arrives at the *NextComputation* entry point. The smaller part of the message containing the coordinates is sent out immediately to the neighbor, while the larger packet containing the forces computed in that entry point are sent out only at the end. The performance of the program improved from 660 seconds to 600 seconds, an improvement of about 10%.

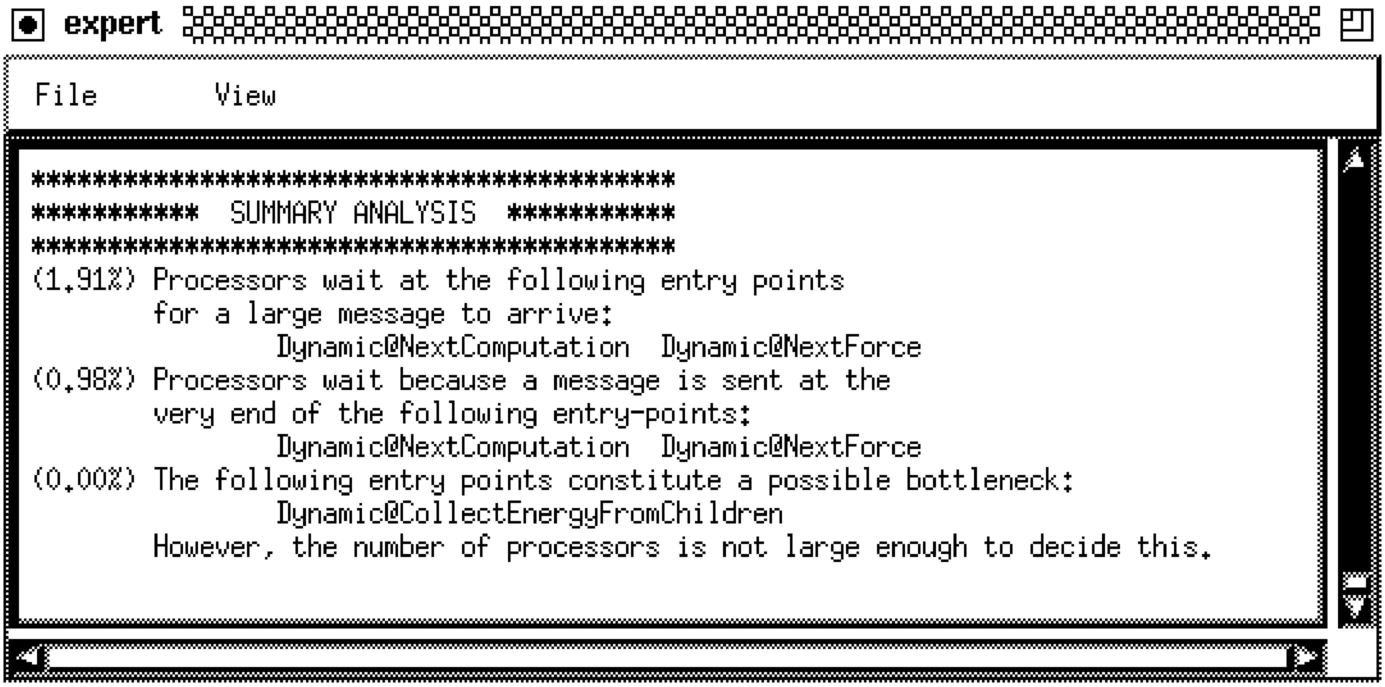

Figure 7.18: Analysis report for experimental model after first optimization.

We had expected the large message containing the forces sent at the end of the *NextComputation* entry point would also pose a performance problem. However, the analysis, shown in Figure 7.18, indicates little possibility of further improvement in the program. Why doesn't this large message sent at the end of the *NextComputation* entry point pose a problem? The amount of computation dependent on this message containing forces is substantially less than the amount of computation depending on the message containing coordinates. Therefore, delaying the force message does not delay a large amount of computation; it only adds some latency, which is effectively overlapped by the large computation triggered by the coordinate message.

Chapter 8

Conclusion

Performance tuning of parallel programs is a complex problem. It is clearly desirable to provide the programmer with system support for this purpose. This is particularly true if parallel programming has to become a broad based activity involving a large number of programmers whose primary area of expertise is likely to be in an application domain and not in performance tuning of parallel programs. This thesis has explored a possible approach in this direction.

Most current performance tools provide visual or aural feedback about generic performance parameters, such as utilization and network traffic. Further, most tools target the SPMD model of computation. In this thesis, we focused on the performance of parallel programs written in an object-based and message-driven paradigm. New language-specific visual feedback techniques become necessary for this purpose, because the execution model of a message-driven program is significantly different from that of the SPMD model of execution. Further, automatic performance analysis becomes feasible for such a model because it provides considerable information about a program's behavior.

One of the essential ingredients for automatic performance analysis is a language with high degree of specificity (a language for which information about program behavior is readily acquired). The Charm language was used for this purpose in this thesis. As illustrated in Chapter 3, a performance analysis system can acquire information about the behavior of a Charm program through one of the following mechanisms:

1. Language constructs, such as messages and chares, provide information about placement and granularity. Further, as described in Chapter 3, we developed specific information sharing mechanisms that provide more information about shared data.

2. System libraries, such as quiescence, load balancing, and queuing strategies, provide more information about points of global synchronization, placement, and scheduling of tasks.

Based on these mechanisms, detailed information about a program's behavior can be collected. In Chapter 4, we described the construction of the event graph by combining compile time information and run time traces. The event graph is a rich data structure containing substantial amount of information about program behavior. We also discussed two problems in the obtaining traces: asynchronous clocks and perturbation due to tracing. We solved the problem of asynchronous clocks with a restricted post-mortem simulation of the program's execution, which generates approximate real time order. We developed a replay-based technique which reduced the amount of perturbation in a program by as much as half in some cases: with this strategy the first run of the program is used to collect timing information and minimum traces needed for replay, and the second run is used to collect extensive performance data. The tracing for replay perturbs the program by about 5% in most cases. Once the event graph has been constructed, it is used to provide program and language specific performance feedback and automatic analysis.

The visual component of the performance analysis tool for Charm, described in Chapter B, provides feedback specific to the Charm execution model, such as creation of chares and overheads of quiescence detection and load balancing. Visual feedback also includes program specific information, such as the numbers and granularities of various methods in chares.

Automatic analysis is carried out by splitting the event graph into logically independent phases. The performance of a logically independent phase can be analyzed and improved independent of the rest of the computation, because it has no effect on the computation that follows it. We described algorithms to compute logically independent phases in Chapter 5. Performance analysis carried out for each logically independent phase is diagnostic: various performance analysis techniques are triggered off on the basis of different program behavior. The analysis is focused by restricting attention to a small subset of events in the whole computation. The subset of events we consider are those included in the last event chain in the computation. The last event chain of the computation is the union of the last event chains in all logically independent phases. The last event chain in a logically independent phase is constructed by starting with the event that completed last in the phase and tracing through its chain of creators to the first event in the phase. The last event chain is a representative component of program behavior, and improving the performance of events lying on that chain will normally lead to improvement in the turnaround time of the program. The algorithm for the construction of the last event chain is described in Chapter 5. The performance analysis techniques we developed diagnose well-known problems, such as load imbalance, and other problems, such as scheduling analysis and pattern analysis. Various performance analysis techniques are described in Chapter 6. The automatic analysis infrastructure thus consists of the following: an algorithm to partition the execution into logically independent phases, an algorithm to determine the last event chain, a decision-tree based approach which applies different performance analysis techniques to determine performance problems for each phase, and techniques to combine different analyses from the same and multiple phases.

The effectiveness of the techniques we developed are examined and demonstrated on four representative applications: traveling salesman problem, multiple linear solver, matrix multiplication, and parallel molecular dynamics. We found that our techniques were able to identify the program's performance problems in most cases. In some cases, analysis techniques could not isolate the problem precisely because specific information was not available. Even in such cases, the analysis proved useful by providing partial information which the programmer could use to solve the problem.

8.1 Future work

The approach we have pursued has identified many new opportunities for performance analysis and research issues that need to be investigated. So far, we have identified a few issues for future work. They can be broadly classified under work for language specificity improvements, reduction in perturbation due to tracing, and new performance analysis techniques.

8.1.1 Specificity improvement

Our current definitions and operations do not provide a method to "destroy" a shared variable. In the experience we have had so far with parallel programs such an operation has not been needed. One consequence of not having a destroy operation on automatic analysis is that the scope (in terms of execution time) of a variable is the entire program. This can sometimes lead to broad, rather than focused, analysis. Further, one can conceive of parallel programs where a destroy operation may become necessary because of memory usage. Therefore, we plan to support this operation for specifically shared variables in future versions of Charm.

In future work, the acquisition component of Projections will be further developed to acquire information about a program's synchronization characteristics. We plan to use the Dagger [89] notation to acquire information about task-level synchronization in the program. We are also investigating mechanisms to acquire additional information about speculative computations in a program.

8.1.2 Perturbation reduction

In future, we plan to examine techniques to collect data from hundreds and thousands of processors without requiring prohibitively large memory space. This would need development of minimal data formats and compression techniques for the log files. A solution adopted at times has been to display data on a real-time basis (i.e. displaying attributes as they are generated during the run of the program, thus obviating the need to store them in files). Such displays constrain user-analysis in obvious ways. Another aspect of scalable displays is the ability to select groups of processors for further review. Currently we allow the user to select only contiguous sub-ranges. In future, the user can select groups of processors which have some unifying thread — e.g. all the processors in the same position in all the planes of a three-dimensional mesh.

8.1.3 New analysis techniques

The event graph data structure has considerable amount of information about Charm programs. We haven't exploited all the information in the event graph data structure. For example, one aspect that has been virtually left out in the performance analysis techniques we developed so far has been the information about objects in Charm. Data encapsulation and ordering of execution of methods inside an object (available through Dagger) makes it possible to isolate the cause of an event to a limited set of methods in other objects. Figure 8.1 shows an example in which only messages m_0, m_1, and m_3 in object O_1 affect message m_7 in object O_4. We plan to acquire such information and use it for analysis.

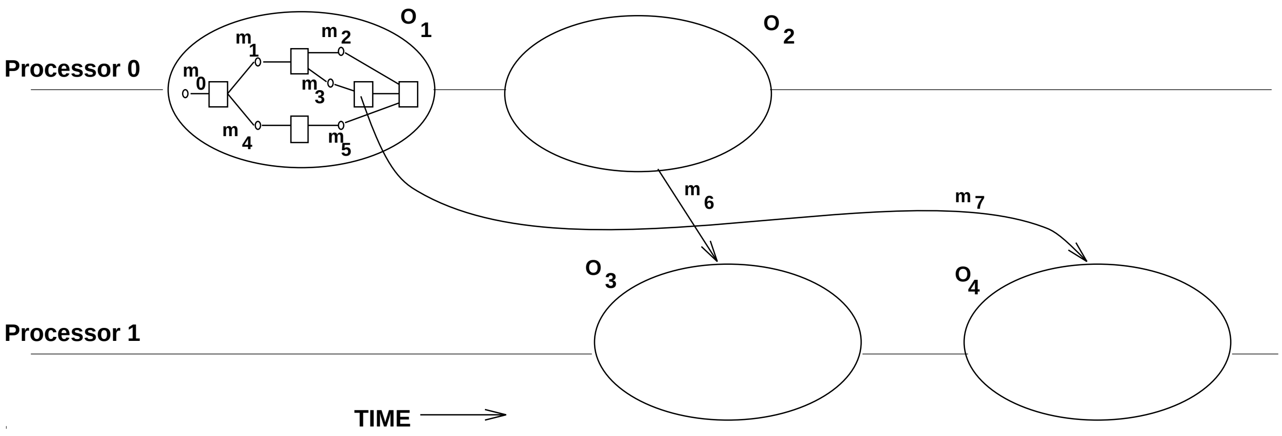

Figure 8.1: Causal analysis of events in a program execution.

Our experience has been that the capabilities of the analysis tool has increased with every application program. We believe that the current techniques identify many severe and hard to detect problems, but we anticipate that many more techniques will be needed as the performance tool is used for new applications. One area of active research in future will thus be the addition of new techniques.

Appendix A

Quiescence detection

A.1 Correctness proof for quiescence detection algorithm

In this section, we offer an informal proof for the quiescence detection algorithm. The proof is in two parts: first, we prove that if the system is quiescent the algorithm will detect it, and, next we prove that the algorithm detects quiescence *only* when the system is indeed quiescent.

Theorem 11 If quiescence has occurred then the algorithm will report it.

Proof: If quiescence has occurred, then there are no activation messages remaining to be processed. Since messages are not created spontaneously or lost, the counts for the creations and processings of activation messages must match, and the algorithm will detect quiescence in at most two iterations of Phase 1 followed by one iteration of Phase 2. □

Theorem 12 The algorithm will not report quiescence unless the system has been quiescent.

Proof: The proof is by contradiction. Assume that even though the algorithm has reported quiescence, the system is not quiescent. Since the system is not quiescent, at least one of the following must be true (from the conditions for quiescence) : there are unprocessed activation messages or there is atleast one busy processor.

We introduce some notation first. Let τ_i and υ_i denote the time at which the last instances of Phase 1 and Phase 2, respectively, were completed on processor i. Let Υ denote the time at which processor 0 made the broadcast to initiate Phase 2. And for a message m, let c_m denote its time of creation and p_m its time of processing. Since the broadcast to begin Phase 2 occurred *after* all processors had completed Phase 1 and *before* any processor began Phase 2, therefore:

$$(\forall i, j)(\tau_i < \Upsilon < \upsilon_j) \tag{A.1}$$

Can there be any unprocessed activation messages in the system after Phase 1 if quiescence has been reported? Let us assume that there are unprocessed messages in the system after Phase 1. Since each processor was idle at the end of Phase 1 on that processor, and no messages can be created spontaneously, atleast one of the unprocessed messages, say a, would have to have been created before Phase 1 on some processor. At the end of Phase 1 the counts for number of messages created and processed were the same (otherwise Phase 2 would not have been started). Therefore, for every message whose creation, but not processing, occurred before Phase 1, there would be a corresponding message for which the processing, but not the creation, occurred before Phase 1; otherwise the counts would not match. Let b be a message whose processing, but not creation, occurred before Phase 1. Then the following is true:

$$(\exists i)(c_a < \tau_i) \tag{A.2}$$

$$(\exists i)(\tau_i < p_a) \tag{A.3}$$

$$(\exists i)(p_b < \tau_i) \tag{A.4}$$

$$(\exists i)(\tau_i < c_b) \tag{A.5}$$

At the end of Phase 2, the counts for total number of messages created and processed in the system, N_c and N_p, have the same values as they had after the last iteration of Phase 1. Since n_c and n_p are monotonically non-decreasing, their values must have remained unchanged on each processor, implying that no messages were created or processed on any processor between Phase 1 and Phase 2. Therefore a and b must have been created and processed, respectively, after Phase 2.

$$(\exists i)(\upsilon_i < c_b) \tag{A.6}$$

But b could not have been created after Phase 2, for by combining Equations A.1, A.4, and A.6, we get:

$$(\exists i, j)(p_b < \tau_i < \Upsilon < \upsilon_j < c_b) \tag{A.7}$$

Since a message could not have been created after it had been processed, no message such as b could exist. But the messages a and b exist in pairs; therefore there can be no messages which are unprocessed after Phase 1 if quiescence has been reported.

Can any processor be busy after Phase 1? The end of Phase 1 on a processor implies that the processor is idle (and has received messages from its children), therefore if it is busy after that it must be because it created or processed some activation message. However no messages could have been created or processed after Phase 1, otherwise there would have been an increase in the counts n_c or n_p, and quiescence would not have been detected. Therefore no processor could have been busy after Phase 1.

We have proved that after quiescence has been reported there are no unprocessed activation messages and no busy processors in the system. Therefore when quiescence is reported, the system is indeed quiescent. □

A.2 Performance of quiescence detection in Charm

The quiescence detection feature in Charm has been used in the implementation of a wide variety of real-life applications including parallel algorithms for logic synthesis [90] and for test pattern generation of sequential circuits[91]. In order to measure the performance of the quiescence detection algorithm in varying program contexts, we tested its performance for the following four synthetic benchmark problems[1] on a nonshared memory machine:

1. Problem A is a parallel divide and conquer application. Computation starts with an initial problem, which is recursively divided to create sub-problems which are executed in parallel. The solutions from sub-problems are then combined. In the initial phases of the computation when sub-problems are being created there isn't enough work for all processors. The situation is similar at the very end when solutions are being combined and sub-problems finish executing.

2. Problem B is a multi-phase application, where each phase is itself a divide and conquer application. There are six phases of the divide and conquer application. Parallelism and

[1] We do not need to use a quiescence detection algorithm to detect quiescence for any one of the benchmark problems; the problems are only used as a controlled experiment, where the onset of quiescence can be independently ascertained.

processor utilization varies between near-idleness to total-utilization several times before termination. Thus, Problem B is good benchmark to test the efficacy and correctness of the quiescence detection algorithm.

3. The task graph for Problem C is a finite length chain of processes with the property that each process in the chain is created by the process preceding it in the chain (the first process in the chain is created by the main process), and only one process in the chain is active at any instant of time. Each new process is created on a randomly chosen processor.

4. In Problem D, each processor has one process, and the processes communicate along a directed cycle on the processes. The computation consists of a pre-determined number of iterations of sends and receives. In an iteration each process sends a message to the next process in the cycle, and receives a message from the previous process in the cycle.

Table A.1 shows the performance results of the quiescence detection algorithm for program runs on the NCUBE/2, a nonshared memory machine. Column 1 shows number of processors. Column 2 shows the number of control messages that were used to detect quiescence. Column 3 shows the number of iterations of Phase 1 and Phase 2 performed by the algorithm for that execution run. Column 4 shows the time in milliseconds that elapsed between the onset and detection of quiescence by the algorithm. A comparison of the number of control messages needed to detect quiescence with the number of activation messages in that execution run shows that in all cases, except Problem C, the number of control messages used are substantially lower than the number of activation messages. Problem C is a 'hard' problem for the algorithm, for there is only one active process on one processor at any time, and all other processors are idle. The number of control messages generated and processed in Problem C occupy computational resources of idle processors, and they shouldn't be considered as an indication of the overhead of the quiescence detection algorithm. This claim is substantiated by the results in Table A.2. For Problem C, the number of control messages used by the quiescence detection algorithm is substantially more than the number of activation messages (in most cases); however the average overhead of the quiescence detection algorithm for Problem C is only about 12%.

Table A.2 shows the execution times for Problems A, B, C and D with and without the quiescence detection algorithm on a nonshared memory machine, NCUBE/2. For problems A and B, the application performs better in some cases with the quiescence detection algorithm

#PE	Msgs	Iterations Phase1+Phase2	Time (ms)
2	20	8+2	5
4	40	9+1	6
8	88	10+1	7
16	144	8+1	11
32	256	7+1	16
64	384	5+1	25
128	996	6+1	41
256	1992	6+1	73

Problem A
Activation Msgs: 6389

#PE	Msgs	Iterations Phase1+Phase2	Time (ms)
2	90	44+1	8
4	172	42+1	16
8	264	32+1	20
16	416	25+1	62
32	608	17+2	23
64	1024	15+1	30
128	2304	16+2	23
256	3328	12+1	29

Problem B
Activation Msgs: 9036

#PE	Msgs	Iterations Phase1+Phase2	Time (ms)
2	26	12+1	14
4	92	21+1	15
8	232	28+1	17
16	496	30+1	28
32	960	29+1	23
64	2112	32+1	20
128	2688	20+1	29
256	4096	15+1	45

Problem C
Activation Msgs: 42

#PE	Msgs	Iterations Phase1+Phase2	Time (ms)
2	6	2+1	15
4	12	2+1	16
8	24	2+1	18
16	48	2+1	24
32	96	2+1	32
64	192	2+1	30
128	384	2+1	29
256	768	2+1	30

Problem D
Activation Msgs: 2000/processor

Table A.1: The performance results of the algorithm for four problems A-D, on the NCUBE/2, a nonshared memory machine.

running than without it. This is not as surprising as it may seem, because Charm provides dynamic load balancing strategies whose behavior may change because of minor delays in processing other messages due to the quiescence detection messages. For Problem C, the overhead of the quiescence detection algorithm ranges from 10% to 17%. For Problem D, the overhead of the quiescence detection algorithm ranges from 3% to 4%.

Our quiescence detection algorithm adapts automatically to system loads. When the system is heavily loaded very few control messages are generated thus not interfering with the user computation. More control messages are generated and processed when the system is lightly loaded, but this time there isn't enough user work with which the control messages can interfere.

#PE	Problems			
	A	B	C	D
2	8982/9128	55234/53293	43/39	983/944
4	4775/4861	29618/27068	56/51	984/945
8	2547/2618	16850/16173	57/53	988/948
16	1452/1417	9512/9689	68/61	993/954
32	889/861	5758/5731	73/67	1007/967
64	593/646	3732/3610	84/75	1034/995
128	511/429	2710/2525	108/91	1088/1048

Table A.2: The execution times for Problems A, B, C and D with/without the quiescence detection algorithm on a nonshared memory machine, NCUBE/2. All the times are in milliseconds.

Appendix B

Performance feedback and visualization

A preliminary version of the display component of Projections was presented in [49, 23]. It provided information about generic program parameters, such as utilization of processors, and some Charm specific information, such as creation and processing of messages of different system types, i.e., *NewChareMsg*, *ForChareMsg*, *BocMsg*, or *BroadcastBocMsg*. The basic philosophy of how information is visualized has remained essentially the same. However, we now display much more Charm related and program specific information. In this chapter, we outline the scheme with which the performance data for Charm programs is displayed.

In Section B.1, we describe the types of information displayed. In Section B.2, we describe the different visualization strategies in Projections. And in Section B.3, we discuss related work in program visualization.

B.1 What is visualized?

The message driven execution model of Charm allows us to decompose Charm programs into messages and the entry points to which they are addressed. Therefore, for Charm programs not only can we display generic system and processor-based information, such as utilization of processors and idle time, but it is also possible to display Charm specific information about messages classified using either Charm-specific types or using program-specific types. The display component provides the user with a mechanism to view:

1. *System specific performance information.* This includes properties of system, such as busy time, queue lengths, creation and processing of messages (or tasks), where messages are classified according to the types defined by the system, such as *NewChareMsg*, *ForChareMsg*, *BocMsg*, or *BroadcastBocMsg.* In addition, one can view information about the load balancing and the quiescence detection strategy.

2. *Program specific performance information.* Projections allows the user to view information about the creation, processing, and granularity of messages to entry points.

B.2 How is data visualized?

Henceforth, we will collectively refer to system and program specific performance information as program attributes. Projections displays data about program attributes which allows the user to identify when, where and what type of work occurred during the execution of the program, and how that corresponds to the processor utilization.

The execution of the user program is divided into equal-length periods of time called *stages*. The length of the time period, called *timestep*, used to cut up the execution time into stages is user-defined and can be changed interactively by the user to define finer and coarser stages, as desired.

The most basic Projections views treat program attributes as a function of two variables: *stage* and *processor index.* Each program attribute can be thought of as a three-dimensional object, and the views are merely projections of this object onto the coordinate axes defining the object space. One set of views provide different projections of this two-variable function. We can represent the function, F_a, for the program parameter, a, as

$$a = F_a(s, p)$$

In the above equation s is the stage of program execution and p is the processor index. The stage, s, and the processor index, p, range over a *stage-set* and a *processor-set*, respectively. In the default case the *stage-set* ranges over the stages for the period of execution of the program, and the *processor-set* ranges over the processors used for execution.

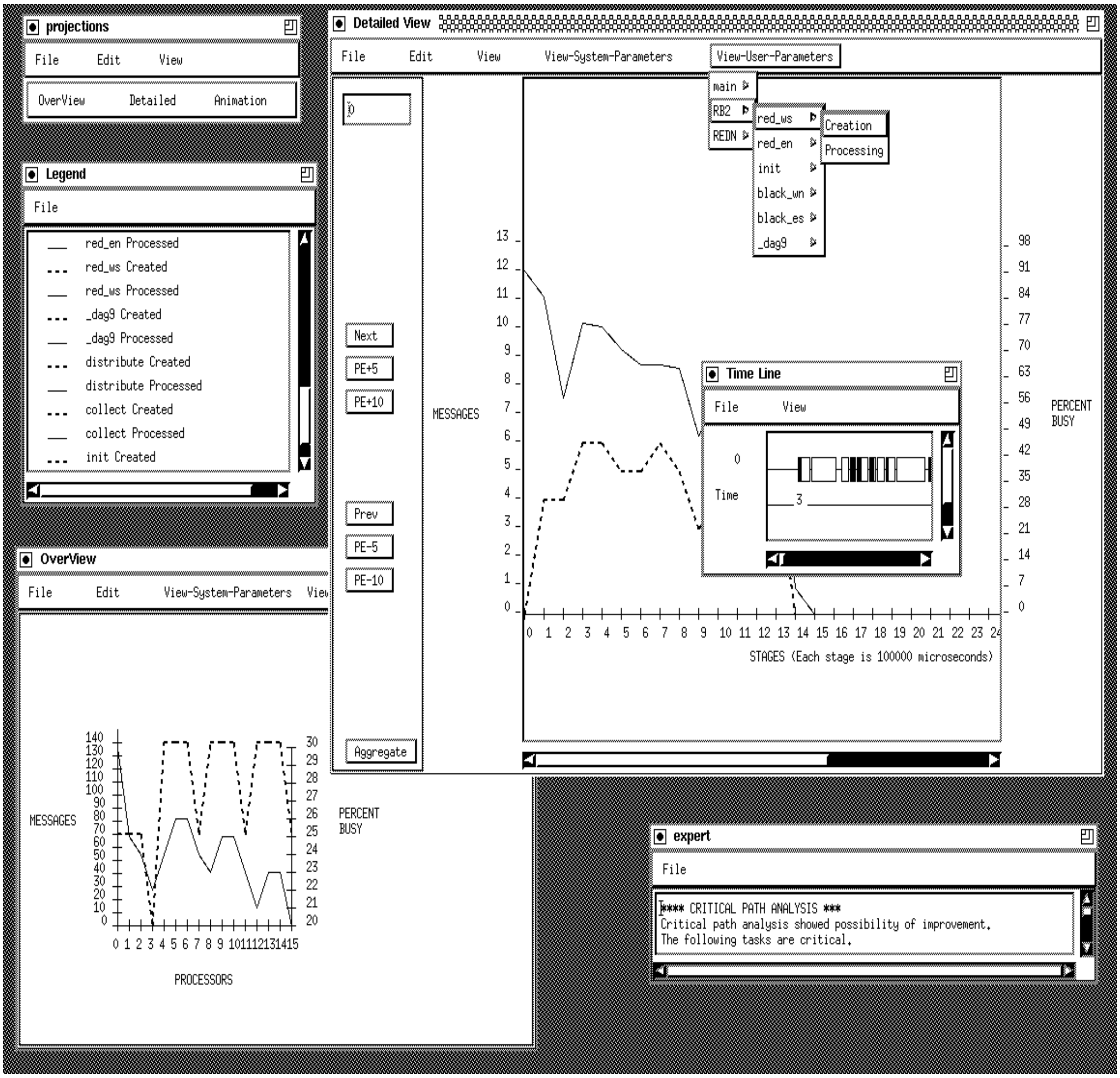

Figure B.1: A sampling of Projection views.

Figure B.1 shows a sample of views available in Projections. The top-level window for Projections appears at the far-left corner. There are there types of views — *overview*, *detailed*, and *animation*.

Figure B.2 shows a Projections overview. An *overview* shows an aggregated (added across all processors or across time) summary of the values of various program attributes. A *detailed* view shows a complete view of all attributes either across processors or across time. An example of a detailed view appears in Figure B.3.

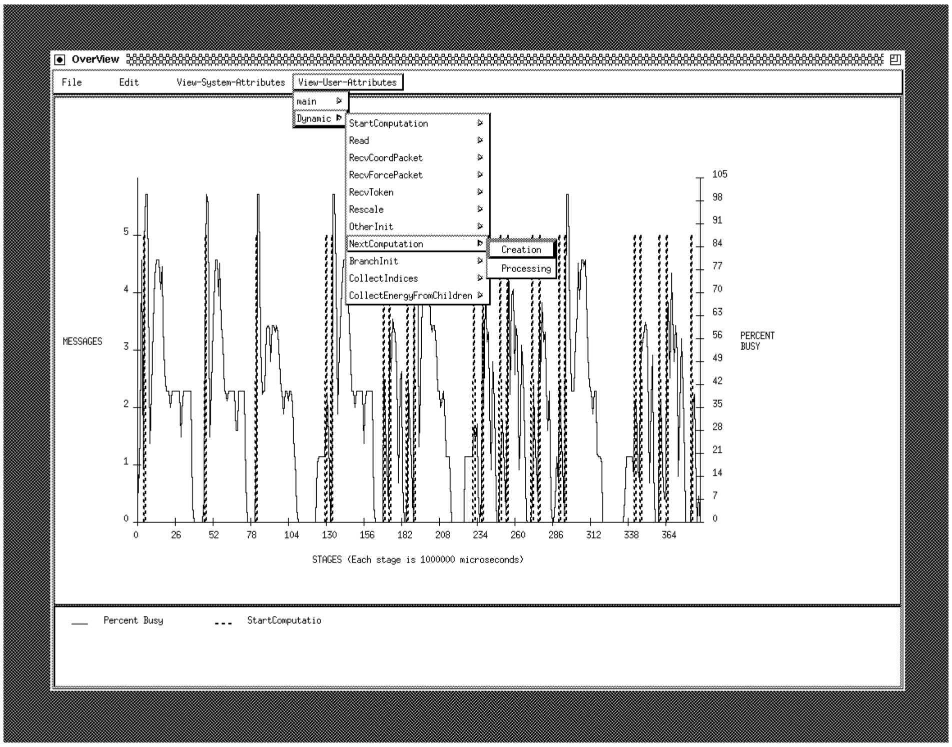

Figure B.2: An overview (projected along processors).

Note that both the overview in Figure B.2 and the detailed view in Figure B.3 have an *View-User-Attributes* button in the menu. This menu item allows the user to select user-defined attributes. This menu is different for each program, and reflects the structure of the program: the chares and the entry-functions that compose the chare. For example, in this program, there are two chares: *main* and *Dynamic*. The *main* chare has three entry points: *CharmInit*, *Quiescence1*, and *Quiescence2*. The *Dynamic* chare has the following entry points: *CollectEnergyFromChildren*, *CollectIndices*, *BranchInit*, *NextComputation*, *OtherInit*, *Rescale* , *RecvToken* , *RecvForcePacket*, *RecvCoordPacket*, *Read* , and *StartComputation*. Using these menus, the user can display information about the creation and processing of messages to any of the entry points.

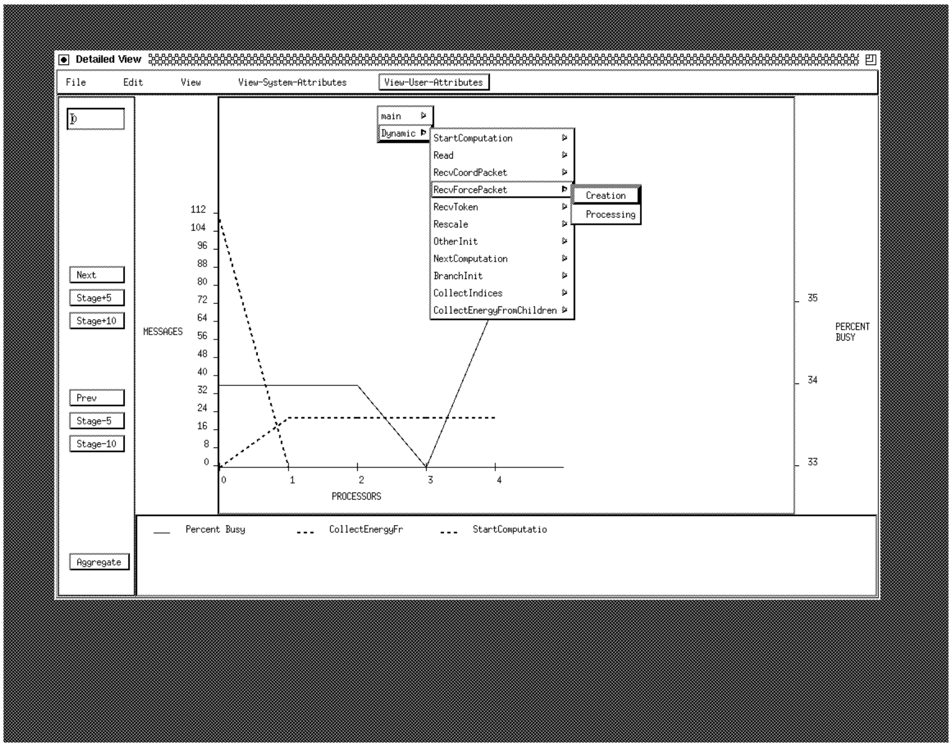

Figure B.3: A detailed view (projected along time).

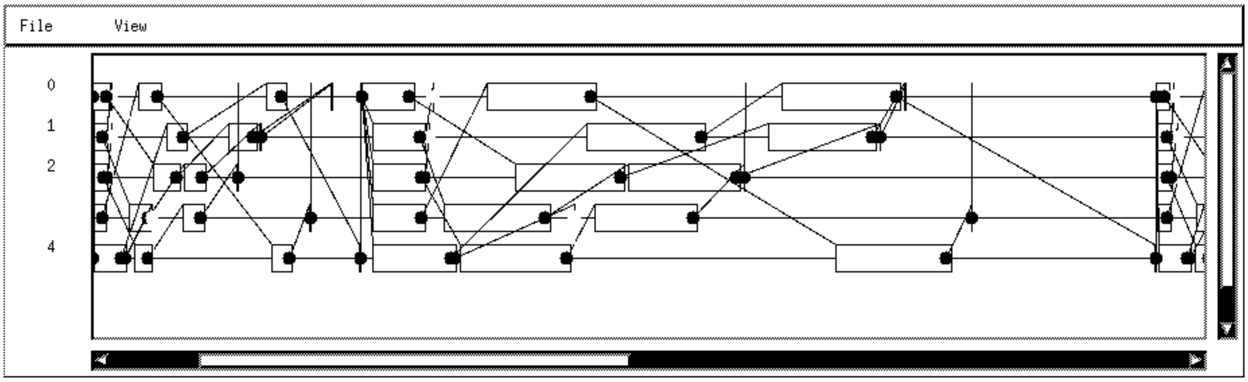

Figure B.4: A timeline.

One can also query inside the detailed view to get a *timeline* [1] of events occurring on the chosen set of processors for the chosen period of time. This view is useful in understanding

[1] This view is inspired by the performance tool developed at the Argonne National Laboratory called Upshot [63].

what happened on which processor at what time. Figure B.4 shows an example of a timeline view.

The third type of views are animation views. In these views each processor's state is represented as a color — blue for busy and red for idle, and intermediate shades between blue and red representing intermediate levels of busy and idle. The processors can be arranged in different topologies, and the user can understand from the colors of processors through the various stages of program execution the structure of the program. Currently, there are four available topologies — spanning tree, square mesh, ring and three-dimensional mesh. Here again, the user can interactively alter the topology of processor connections, length of a stage, and the stage and processor periods. In the case of the three dimensional mesh view, the user can specify the lengths and orientations of the three dimensions.

B.3 Related work

There has been substantial work done previously on tools to visualize the behavior and performance of parallel programs on parallel machines.

ParaGraph [1] aims to provide the user with a dynamic depiction of the behavior of the parallel program by offering a re-enactment of the program's trace through many different views. The views fall under four broad categories: utilization displays (e.g., Gantt chart, concurrency profile, etc.), communication displays (e.g., message queues, animation, etc.), task displays (e.g. task count, etc.), and other displays (e.g. critical path, phase portrait, etc.). ParaGraph provides multiple views of the same attribute, e.g. to study utilization there are Gantt charts, concurrency profiles, Kiviat diagrams etc., so that the user may be able to see different aspects of the attribute in different views, and this may aid the user's understanding of program behavior. Trace data for ParaGraph can be generated by instrumenting the user program with primitives from PICL (Portable Instrumentation Communication Library)[64].

Upshot [2] displays the logfile information as a time-line for each processor. A time-line for a processor contains either (or both) an event trace of the program on that processor or a trace of states of the program on that processor. An event is defined to have a beginning and a closing state. Different events may be displayed in different (user-chosen) colors.

Our approach in the display component of Projections combines ideas in ParaGraph and Upshot. It allows the user to view generic system information, such as utilization and sizes of queues. It also allows the user to view program specific information about various objects and their corresponding methods, such as in Upshot. In addition it also provides information specific to the Charm paradigm, which ParaGraph and Upshot do not provide. One key difference is that our trace generation is automatic. The user does not need to instrument their code in order to generate trace information: a link time option chooses the correct libraries.

In addition, neither Paragraph nor Upshot are specific to any language. Though this generality makes it possible to use them flexibly with many different languages, along with it comes the loss in information about a specific execution model. Projections is a performance tool geared towards the programming language Charm, providing the user information about the program in terms of language features. It is not proposed as a tool to replace all existing performance tools. Rather, Projections is a performance tool that complements the general-purpose nature of tools, such as Paragraph and Upshot, with information specific to Charm.

Pablo [55, 65] is a *portable, scalable, and extensible* performance environment being developed at the University of Illinois, Urbana. Pablo consists of two components: software instrumentation and performance data analysis. The latter consists of performance data transformation modules that can be graphically interconnected to form an acyclic, directed data analysis graph. Performance data flows through the nodes of this graph, and is transformed to yield various performance metrics. Pablo is a more general displaying environment, which permits a tool builder to display different attributes of a program using many different techniques: in some sense it is a meta-tool. The work being done for Pablo was done concurrently with this project. Tools such as Projections can be developed on top of Pablo with relative ease in the future.

Balsa [66] was a program animation system developed as part of the electronic classroom project at Brown University. In addition, to the capabilities of animation, it provides other features such as being able to execute the program in the system itself. The user can generate trace events by selecting points of interest in the program, which can then be displayed using a *renderer*. It permits views to be reused between programs which produce the same traces. It has one problem: it does not allow one to easily manipulate views.

Voyeur [67] was developed with the idea that different programs require different animation views, and therefore the user should be easily able to create specific views for their program. It provides a class hierarchy of views, which can be combined to generate application specific views.

Belvedere [68, 69, 70] attempts to solve the problems of complexity and concurrency in parallel programs. Visualization tools often present incomprehensible diagrams of nonatomic and concurrent events. Belvedere attempts to solve this problem by reordering the events to produce more comprehensible views. In most cases, reordering produces logically equivalent event graphs. However in some cases, reordering is not possible because of circular dependences. In such cases, the user can construct partially consistent views, which they call *perspective views*. Reordering for perspective views is achieved by determining the connected components of the event graph, and then displaying the graph so that all components at the same level start at the same time. They also use the notion of a phase of program execution. However, their notion of a phase is substantially different from our notion of logically independent phases. In an iterative solver, where each processor exchanges elements with all neighbors, they denote each exchange as a phase, while we denote one iteration of the solver as a phase.

Moviola [10] is an execution history browser. Its basic display is the execution graph of a program: the displays can use either physical or logical time. The browser allows capabilities to zoom, pan, and scroll through the graph in either direction. It also provides options for the user to select subsets of processes, synchronization, and events.

B.4 Summary

In this chapter, we described how visual feedback about information specific to the Charm programming model is provided. Information is broadly classified into generic system attributes, such as utilization, and user attributes, such as creation of a specific entry point. Each attribute of the information can be thought of as a function of the time and processor number. The primary feedback mechanism provides projections of each attribute onto either the time or the processor number.

Bibliography

[1] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, pages 29–39, Sept. 1991.

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

[3] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21, no. 8:666–677, 1978.

[4] L. V. Kalé. The Chare kernel parallel programming language and system. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II, pages 17–25, St. Charles, IL, August 1990.

[5] T. vonEicken, D. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. *ACM*, pages 256–266, 1992.

[6] A. S. Grimshaw. *Mentat: an object-oriented macro data flow system.* PhD thesis, University of Illinois, Urbana-Champaign, June 1988.

[7] A. Gursoy. *Performance benefits of message driven execution.* PhD thesis, University of Illinois, March 1994.

[8] C. B. Stunkel, D. C. Rudolph, W. K. Fuchs, and D. A. Reed. Linear optimization: a case study in performance analysis. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications.* Association for Computing Machinery, March 1989.

[9] D. W. Jensen and D. A. Reed. A performance analysis exemplar: parallel ray tracing. In *Concurrency: practice and experience*, volume 4(2), pages 119–141, April 1991.

[10] R. Fowler and Ivan Bello. The programmer's guide to Moviola: An interactive execution history browser. Technical Report 269, University of Rochester, Department of Computer Science, February 1989.

[11] D. J. Kuck et al. The effects of program restructuring, algorithm change, and archictecture choice on program performance. In *International Conference on Parallel Processing*, pages 129–138, August 1984.

[12] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. In *Communications of the ACM*, volume Volume 29, No. 12C, December 1986.

[13] J. R. Allen and K. Kennedy. Automatic transformation of fortran programs to vector form. In *ACM Transactions on Programming Languages and Systems*, volume Volume 9, No. 4, pages 491–542, October 1987.

[14] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2, 4:315–339, December 1990.

[15] G. A. Geist and V. S. Sunderam. The PVM system: Supercomputing level concurrent computations on a heterogeneous network of workstations. *Sixth Distributed Memory Computing Conference Proceedings*, pages 258–261, 1991.

[16] J. Dongarra et al. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, 7, No. 2:166–175, 1993.

[17] J. Flower, A. Kolawa, and S. Bharadwaj. The express way to distributed processing. In *Supercomputing Review*, pages 54–55, May 1991.

[18] D. H. Gelernter . Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[19] N. Carriero and D. Gelernter . How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, pages 323–357, September 1989.

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

[21] L. V. Kale, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM Parallel Programming Language and System: Part I – Description of Language Features. *IEEE Transactions on Parallel and Distributed Systems*, 1994. (submitted).

[22] L. V. Kale, B. Ramkumar, A. B. Sinha, and V. A. Saletore. The CHARM Parallel Programming Language and System: Part II – The Runtime system. *IEEE Transactions on Parallel and Distributed Systems*, 1994. (submitted).

[23] L. V. Kalé and A. B. Sinha. Projections: A scalable performance tool, April 1993. Parallel Systems Fair, International Parallel Processing Symposium.

[24] V. A. Saletore . *Machine Independent Parallel Execution of Speculative Computations.* PhD thesis, Dept. of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1990.

[25] L. V. Kalé and A. B. Sinha. Information sharing in parallel programs. In *International Parallel Processing Symposium*, April 1994.

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

[27] M. Snir, W. Gropp, and E. Lusk. Document for a standard message-passing interface: point to point communication. draft, 1993.

[28] A. Geist and M. Snir. Document for a standard message-passing interface: collective communication. draft, 1993.

[29] I. Foster and S. Taylor . *Strand: New Concepts in Parallel Programming.* Prentice Hall, 1990.

[30] Kai Li. *Shared virtual memory on loosely coupled multiprocessors.* PhD thesis, Yale University, September 1986.

[31] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in emerald. *IEEE Transactions on Software Engineering*, 13 (1):65–74, January 1987.

[32] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the emerald system. *Proceedings of the ACM conference on Object Oriented Programming Systems, Languages and Applications*, pages 78–86, October 1986.

[33] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1989.

[34] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The amber system: parallel programming on a network of multiprocessors. In *In Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[35] P. Dasgupta, R. C. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahmad, R. LeBlanc Jr., W. Appelbe, J. M. Bernabeu-Auban, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wileknloh. The design and implementation of the Clouds distributed operating system. *Computing Systems Journal*, 3, Winter 1990.

[36] H. E. Bal and A. S. Tanenbaum. Distributed programming with shared data. *Proceedings of the 1988 International Conference on Computer Languages*, pages 82–91, October 1988.

[37] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: a language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, pages 190–205, March 1992.

[38] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: distributed shared memory based on type-specific memory coherence. In *Second ACM Symposium on principles and practice of parallel programming*, volume Volume 25, Number 3, March 1990.

[39] B. K. Totty and D. A. Reed. Dynamic object management for distributed data structures. *Proceedings of Supercomputing*, pages 692–701, November 1992.

[40] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11, 1, August 1980.

[41] N. Francez. Distributed termination. *ACM TOPLAS*, pages 42–55, Vol. 2, No. 1, January 1980.

[42] J. Misra and K. M. Chandy. Termination detection of diffusing computations in communicating sequential processes. *ACM TOPLAS*, pages 37–34, Vol. 4, No. 1, January 1982.

[43] C. Hazari and H. Zedan. A distributed algorithm for distributed termination. *Information Processing Letters*, pages 293–297, 24, 1987.

[44] S. P. Rana. A distributed solution of the distributed termination problem. *Information Processing Letters*, pages 43–46, 17, 1 (July 1983).

[45] F. Mattern. Global quiescence detection based on credit distribution and recovery. *Information Processing Letters*, pages 195–200, 30, 1989.

[46] S. T. Huang. Termination detection by using distributed snapshots. *Information Processing Letters*, pages 113–119, 32, 1989.

[47] T. H. Lai and T. H. Yang. On distributed snapshots. *Information Processing Letters*, pages 153–158, 25 (1987).

[48] W. W. Shu and L. V. Kalé. A dynamic load balancing strategy for the Chare Kernel system. In *Proceedings of Supercomputing '89*, pages 389–398, November 1989.

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

[50] L . V. Kalé, B. Ramkumar, V. Saletore, and A. B. Sinha. Prioritization in parallel symbolic computing. In T. Ito and R. Halstead, editors, *Lecture Notes in Comp. Science*, pages 146–181. Springer-Verlag, 1993.

[51] T. H. Dunigan. Hypercube clock synchronization. *Concurrency: Practice and Experience*, 4(3):257–268, May 1992.

[52] D. L. Mills. Network time protocol (version 2) specification and implementation. Technical Report RFC-1119, DARPA Networking Group, September 1989.

[53] D. L. Mills. Network time protocol (version 3) specification and implementation. Technical Report Draft RFC, DARPA Networking Group, September 1990.

[54] K. G. Shin and P. Ramanathan. Transmission delays in hardware clock synchronization. *IEEE Transactions*, C-37:1465–1471, 1988.

[55] A. D. Malony, D. A. Reed, J. W. Arendt, R. A. Aydt, D. Grabas, and B. K. Totty. An integrated performance data collection, analysis, and visualization system. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications.* Association for Computing Machinery, March 1989.

[56] A. Malony. *Performance observability.* PhD thesis, University of Illinois, October 1990.

[57] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[58] G. Neiger and S. Toueg. Substituting for real time and common knowledge in distributed systems. *Proceedings of the sixth symposium on the principles of distributed computing*, pages 281–293, August 1987.

[59] J. L. Welch. Simulating synchronous processors. *Information and Computation*, 74(2):159–171, August 1987.

[60] T. J. LeBlanc and J. M. Mellor-Crumney. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36, No. 4:471–482, April, 1987.

[61] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Scalable High Performance Computing Conference*, May 1994.

[62] J. K. Hollingsworth and B. P. Miller. Dynamic control of performance monitoring on large scale parallel systems. In *International Conference on Supercomputing*, July 19-23 1993.

[63] T. Disz and E. Lusk. A graphical tool for observing the behavior of parallel logic programs. Technical Report CSRD 746, Argonne National Laboratory, February 1988.

[64] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. Picl: a portable instrumented communication library, c reference manual. Technical Report ORNL/TM-11130, Oak Ridge National Laboratory, 1990.

[65] D. A. Reed, R. D. Olson, R. A. Aydt, T. M. Madhyastha, T. Birkett, D. W. Jensen, B. A. A. Nazief, and B. K. Totty. Scalable performance environments for parallel systems. Technical report, University of Illinois, Urbana, 1991.

[66] M. H. Brown. *Algorithm animation.* PhD thesis, Brown University, 1988.

[67] D. Socha, M. L. Bailey, and D. Notkin. Voyeur: Graphical views of parallel programs. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN NOTICES*, 24(1):206–215, January 1989.

[68] A. A. Hough and J. E. Cuny. Belvedere: Prototype of a pattern-oriented debugger for highly parallel computation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 735–738, University Park PA, 1987.

[69] A. A. Hough and J. E. Cuny. Initial experiences with a pattern-oriented parallel debugger. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN NOTICES*, 24(1):195–205, January 1989.

[70] A. A. Hough and J. E. Cuny. Perspective views: A technique for enhancing parallel program visualization. In *Proceedings of 1990 International Conference on Parallel Processing*, pages II.124–II.132, University Park PA, August 1990.

[71] Cui-Qing Yang and B. P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 366–373, 1988.

[72] N. Islam. *Customized message passing and scheduling for parallel and distributed applications.* PhD thesis, University of Illinois, Urbana-Champaign, June 1994.

[73] S. L . Graham, P. B. Kessler, and M. K. McKusick. GPROF: a call graph execution profiler. *SIGPLAN 1982 Symposium on Compiler Construction*, pages 120–126, June 1982.

[74] S. Saini and H. Simon. Enhancing applications performance on the intel paragon through dynamic memory allocation. *Intel Supercomputer User's Group Meeting, St. Louis*, pages 185–192, October 1993.

[75] J. K. Hollingsworth and B. P. Miller. Slack: a performance metric for parallel programs. Technical report, University of Wisconsin, Madison, Computer Sciences Technical Report, March 1992.

[76] C. Yang and B. P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, JUNE 1988.

[77] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski. Ips-2: The second generation of a parallel program measurement system. *IEEE Transactions on parallel and distributed systems*, 1 (2):206–217, April 1988.

[78] T. E. Anderson and E. D. Lazowska. Quartz: a tool for tuning parallel program performance. *Proceedings of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 115–125, May 1990.

[79] L. H. Jamieson. Characterizing parallel algorithms. In Leah H. Jamieson, D. Gannon, and R. J. Douglass, editors, *The characteristics of parallel algorithms*. The MIT Press, 1987.

[80] T. Fahringer. *Automatic performance prediction for parallel programs on massively parallel computers*. PhD thesis, University of Vienna, 1993.

[81] T. Fahringer. Personal communication. August 1994.

[82] E. W. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

[83] J. D. C. Little, K. G. Murty, D. W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11:972–989, 1963.

[84] W. Shu and L. V. Kalé. Dynamic scheduling of medium-grained processes on multicomputers. Technical report, University of Illinois, Urbana, 1989.

[85] M. Furuichi, K. Taki, and N. Ichiyoshi. A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-psi. *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990.

[86] I. Ahmad and A. Ghafoor. A semi distributed allocation strategy for large hypercube supercomputers. *Supercomputing*, 1990.

[87] H. Grubmüller, H. Heller, A. Windemuth, and K. Schulten. Generalized verlet algorithm for efficient molecular dynamics simulations with long-range interactions. *Molecular Simulation*, 6(1–3):121–142, 1991. Pub.# 141.

[88] W. B. Streett, D. J. Tildesley, and G. Saville. Multiple time-step methods in molecular dynamics. 35(3):639–648, 1978.

[89] A. Gursoy and L. V. Kalé. Dagger: Combining the benefits of synchronous and asynchronous communication styles. In *International Parallel Processing Symposium*, April 1994.

[90] K. De and B. Ramkumar and P. Banerjee. ProperSYN: A Portable Parallel Algorithm for Logic Synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, November 1992.

[91] B. Ramkumar and P. Banerjee. Portable Parallel Test Generation for Sequential Circuits. In *Proceedings of the International Conference on Computer-Aided Design*, November 1992.

Vita

Amitabh Sinha was born in Patna, India. He got his B. Tech. in Computer Science and Engineering from Indian Institute of Technology, Kanpur in May 1988. He got an M.S. in Computer and Information Science from Ohio State University, Columbus in December 1989. After completing his PhD, he will be joining Oracle Corporation in Redwood Shores, California, as a Technical Staff Member.