A framework for intelligent performance feedback

Amitabh B. Sinha **Laxmikant V. Kalé**
Department of Computer Science,
University of Illinois, Urbana.
Urbana, IL 61801.
email: sinha|kale@cs.uiuc.edu

Abstract

The significant gap between peak and realized performance of parallel machines motivates the need for performance analysis. Contemporary tools provide only generic measurement, rather than program-specific information and analysis. An object-oriented and message-driven language, such as Charm, presents opportunities for both program-specific feedback and automatic performance analysis We present a framework in which specific and intelligent feedback can be given to the user about their parallel program. The framework will use information about the parallel program generated at compile-time and at run-time to analyze its performance using general expertise and specific algorithms in performance analysis.

Keywords: performance feedback, object-based, message-driven, intelligent analysis, post-mortem

1 Introduction

The need for parallel machines for solving large, computationally challenging problems has been clearly demonstrated. Today there already exist parallel machines with *peak performances* in the range of tens to even hundreds of gigaFLOPS. However writing parallel programs for these machines is not simple: it is not easy to reason about the correctness and efficiency of parallel programs. The primary source of difficulty is the multiplicity of threads of control in a parallel program. In order to ensure correctness, one needs to interleave and map these multiple threads correctly; and in order to ensure efficiency, one needs to interleave them in a manner which reduces the idle time. Therefore, techniques that help analyze the correctness and efficiency of parallel programs are necessary. In this paper, we introduce a framework for providing specific and intelligent performance feedback for parallel programs.

There exist a large and diverse collection of performance analysis tools for parallel programs. The primary emphasis of these tools has been to present information about different attributes of the execution of the parallel program. However, most often, this information is generic, e.g., utilization of processors, size of queues, number of messages, etc. Tools rarely provide more specific information about the program, such as information about different user-defined message types, execution of specific code blocks in the user program, etc. Further, most existing performance *analysis* tools do not *analyze* the information presented for possible performance problems and potential opportunities for improving performance. Analyzing performance information is a non-trivial task: applications programmers find it difficult to effectively analyze the information presented by these tools. The problem is further compounded on massively parallel machines because the amount of information presented is huge.

A good performance analysis tool should be able to (1) display program specific information, and (2) intelligently analyze the information to detect problems. In empirical studies, we have observed that the core of techniques applied in analyzing the efficiency of parallel programs utilizes information about characteristic attributes of the program, such as degree of parallelism, granularity of sub-tasks, load balance, etc. We believe that informative and intelligent display and analysis techniques can be designed for languages, whose primitives provide information about the characteristic attributes of the parallel program. In this paper, we outline the development of a framework that provides specific and intelligent feedback about the performance of a parallel program. The framework consists of three components: the acquisition component, the display component, and the analysis component.

The acquisition system is responsible for acquiring information about the identified characteristic attributes. In most current day parallel programming languages, information about characteristic attributes may be available; however, it may be difficult, if not impossible, to extract the information even with sophisticated compilers. It is essential to have a language with an adequately rich set of primitives, which provide the performance analysis system with a refined understanding of the events that occur during the execution of a program and their potential impact on performance. We have chosen Charm [1], a portable, object-based, and message-driven [1] parallel programming language, as the base language. The object-oriented and message-driven execution model of Charm provides a great deal of specific information about parallel programs written in Charm. In addition, Charm provides specific modes of information sharing which provide further

[1] The code block to be executed by a message is associated with it when it's sent. On arrival at the destination process the associated code-block is automatically executed by the system.

specific information about Charm programs.

The display and analysis components are part of the performance feedback tool called Projections. A preliminary version of Projections is described in [2, 3]. It had only a limited display capability: that of showing information about system attributes, such as utilization and number of parallel objects created. The display component added in the new version of Projections provides specific information about application program attributes. The analysis component uses information acquired about Charm program characteristics to automatically analyze some typical performance feedback problems. The analysis component is under development, and some of the features mentioned are still being implemented.

The paper is organized as follows. In Section 2, we have identified several characteristics of parallel programs. In Section 3, we motivate the need for richer programming primitives, such as those in the Charm parallel programming language. In Section 4, we outline mechanisms to acquire information about program characteristics in the context of Charm programs. In Section 5, we present Projections, a tool that displays information about various program characteristics, and also provides an analysis of the performance of the program. An example is presented in Section 6 which illustrates the use of the performance tool. Finally, in Section 7, we discuss other related performance analysis mechanisms and directions for future work.

2 Identification of program characteristics

Through empirical studies, we have been able to determine a small number of program characteristics which are often used in analyzing the performance of a parallel program. In performance analysis, the goal is to maximize the utilization of each processor. This can be achieved by maximizing the fraction of time a processor executes user code (*user time*), and minimizing the fractions of time a processor *idles* (*idle time*) or executes system related code (*overhead*).

In most cases, it's useful to maximize the amount of user computation. However, in some cases, part of the user computation is wasteful — the performance loss due to such waste is known as speculative loss. The degree of wasteful work in a program must be minimized. Idle time can be affected by the scheme with which tasks are placed on the processors (placement), dependence between tasks and the synchronization requirements of processes, the order of execution of messages (scheduling strategy), the number of tasks that can be scheduled independently at any given moment (the degree of parallelism), and the computational requirements of tasks (grainsize). The contribution of system overheads is 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: scheduling strategy, synchronization, placement, grainsize of tasks, shared variable access time, degree of parallelism, and speculative loss. We shall now discuss each of the characteristics we have identified above in greater detail:

1. **Dependence between tasks and synchronization:** The nature of dependences between tasks and the nature of synchronization amongst them, is perhaps the most important characteristic of a parallel program. The most common causes of bottlenecks in parallel programs is processors/tasks waiting for other tasks to complete. For example, in a divide and conquer program, each interior node of the divide and conquer tree must wait for results from its children. The user/system must minimize such waiting periods, because a waiting task can cause a processor to idle.

2. **Scheduling strategy:** The scheme with which tasks are scheduled for execution in a parallel system can also affect a program's performance. For example, in a program where tasks on the critical path are not scheduled early might delay the completion of the critical path.

3. **Placement and load balance:** This describes how tasks are placed on the various parallel processing elements, e.g., in the SPMD model of execution one task is placed on each processor. Closely related to the placement of tasks in a parallel program is the degree of load balance. One metric for measuring load balance is the balance in the number of tasks. However, that is not sufficient, since even though the number of tasks on different processors may be balanced, their loads may not be balanced. Therefore, a second metric is essential: it is the balance of computational load in the program. The first metric is still necessary because tasks have other attributes such as creation, processing, and memory overheads which also need to be balanced.

4. **Shared variable access:** In different programs, processes must share information amongst each other in some form or the other. 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. For example, in a program where there is some information which is initialized once and thereafter accessed only in the read mode, the cost of data access could be minimized by replicating the information.

5. **Degree of parallelism:** The degree of parallelism measures the number of ready parallel tasks that exist at any given time. It is an important factor in analyzing the performance of programs, because when programs are executed on larger machines there may not be enough parallel tasks to keep the machine busy. Further, if the number of ready parallel tasks on a parallel processor are large, it also indicates a high degree of overlap, which can lead to faster completion times.

6. **Granularity of tasks:** The granularity of a task in a program is the average computational time needed by a task. 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, shared memory access time, context switch time, 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. It is therefore necessary to carefully choose an appropriate granularity for tasks in a parallel program.

7. **Degree of speculative work:** In a number of parallel programs, the total amount of computational work is dependent on the order in which parallel tasks are executed. In such programs, one can generally define the minimum amount of computational work that must be done; any additional work is termed as speculative. It is necessary to keep the amount of speculative work to a minimum, otherwise speedup anomalies can be observed. The amount of speculative work can be reduced using different scheduling strategies for the tasks, which may depend on priorities attached to them.

3 The need for a richer parallel language

Currently, most parallel applications are based on the SPMD model, and are written using C/Fortran parallel libraries. However applications based on the SPMD model do not provide much specific

information about the program. In particular, it is difficult for the system to acquire information about sub-tasks in a program or about the nature of information sharing.

In the SPMD model, a copy of the program executes on each processor — there is only one task on each processor, and it is is the entire program. Further, the SPMD model does not permit the dynamic creation of new tasks on a processor. This all-encompassing concept of a task does not permit the analysis tool to easily extract specific information about sub-computations in the program. We could consider the actions associated with a message to constitute a task. However even though the actions associated with receiving a message generally follow the *receive* statement in the program, the exact actions associated with a message are not very clear without some dependence analysis.

Further, most widely used SPMD programming languages do not provide any specific modes of information sharing — the user needs to explicitly implement the types of information sharing needed by the program. However it is difficult (again, it may require sophisticated dependence analysis techniques) for the system to identify the particular information sharing mechanism implemented by the user. For example, a user might have implemented a variable that is shared only in the read-mode by the sub-tasks in the parallel program. However the compiler may not be able to detect even such a trivial information sharing mode. In order to acquire more specific information about a parallel program, it therefore, becomes necessary to use a different, and more specific programming paradigm.

We have chosen an object-based portable parallel programming language called Charm. The execution model of Charm is message-driven, i.e., the arrival of a message at a particular processor results in the invocation of the code-block associated with it. There is no explicit receive statement in Charm. 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.

The basic unit of computation in Charm is a *chare* (similar to an object). A chare's *definition* consists of an encapsulated data area and entry functions that can access the data area. A chare *instance* can be created dynamically using the *CreateChare* system call. As a result of this system call, a *new-chare* message is created. Each chare instance has a unique address. Entry functions in a particular chare instance can be executed by addressing a message to the desired entry function of the chare. Messages can be addressed to existing chares using the *SendMsg* system call. This call generates *for-chare* messages.

Charm provides a second type of process called a branch office chare, which is essentially a chare replicated on each processor. A branch office chare has the same syntax as that of a chare Branch office chares provide a convenient abstraction for the implementation of various distributed strategies, e.g., load balancing, quiescence detection etc.

The object-based and message-driven paradigm of Charm already provides us with some information about Charm programs. In this section, we shall illustrate how one can obtain information about various characteristics of Charm programs through static and dynamic acquisition techniques. Information about some characteristics are not available through either of these mechanisms. For some of these, we have been able to acquire information through new language constructs. We also discuss how information about the other remaining characteristics can be obtained by adding new features, such as known libraries or annotations.

An interesting outcome of the decision to select Charm as the base language has been that its object-oriented nature and message-driven model of execution make many distinctly different

performance optimization decisions possible. In most SPMD programs, the outcome of an analysis is often confined to generic advice, such as *move the sends up, and move the receives down.* In such a model, it is difficult, if not impossible, to answer questions such as *"Which messages, up to where, and down to where?"* A message-driven and object-oriented paradigm makes it possible to answer such questions and provide even more avenues for analysis.

4 Acquiring information about the characteristics of Charm programs

In this section, we present mechanisms to acquire information about those characteristics of Charm programs which are useful in analyzing their efficiency. Information about the characteristics of a parallel program can be acquired either *statically* (through language constructs) or *dynamically* (through the run-time system). For Charm programs information about placement, shared variables, and synchronization can be acquired statically, while information about other characteristics, such as grainsize of tasks, scheduling, etc., can be acquired dynamically.

Placement:

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.

Shared variables and their access:

In previous work [4], we have discussed the motivation and details for providing a limited number of specific information sharing mechanisms in Charm. The reason Charm does not provide any general-purpose shared variable is that in experiments with parallel programs, we have observed that they often share data in only a few *distinct* and *specific* modes; the 'completely general' shared variable is rarely used. Currently, Charm provides five different kinds of shared variables: *read only*, *write once*, *accumulator*, *monotonic* and *distributed tables*. These specific modes were implemented as efficiently as possible on a particular architecture.

A **read-only** variable is initialized at the *CharmInit* entry point and can only be accessed via the call *ReadValue* from any other chare. This call simply returns the (fixed) value of the variable. A read only variable may be a scalar (e.g. integer), array or a structure.

A **write-once** variable is created and initialized any time (and from any chare) during the parallel computation. Once created, its value can only be read. The creation is done via a non-blocking call *WriteOnce(dataptr, datasize, entryPoint, ChareID)* which immediately returns without any value. Eventually, the variable is "installed", and a message containing a unique name assigned to the new variable is sent to the designated *entryPoint* of the designated chare. This unique name can be passed to other chares, which can access the variable by calling *DerefWriteOnce*.

Accumulator variables are counters, with one difference. The initial value of an accumulator

must be *zero*. Accumulator variables have associated with them two functions - an *accumulating* function that adds to the counter, and a *combining* function that combines two counter variables. An accumulator variable is initialized during initialization of the main chare, and can be read only once, destructively. It can be modified only via a function *Accumulate*, which adds a given value to the accumulator. The destructive read is performed via the (non-blocking) call *CollectValue*, which results in eventual transmission of a message containing the final value of the accumulator to the named chare. It is easy to think of the accumulator as an integer to which we want to add other integers from time to time, although the language allows it to be any type, with any user-defined commutative associative operation.

A **monotonic** variable is global variable that "increases" monotonically in some metric by the application of an idempotent function. It is used typically in branch-and-bound computations. Its (approximate) current value can be read by any chare at any time using the call *MonoValue*, and a potential new value for it can also be provided by any chare using the call *NewValue*. The supplied value replaces the old value if it is "better" than the old value, using a user-supplied comparison function. It is only guaranteed that the value read from any other chare will be *eventually* better or equal to the new value supplied.

A **distributed table** consists of a set of entries, each with a key part and a data part. Various asynchronous access and update operations on entries in the table are provided. For example, one may call *Find(tbl, key, entryPoint, chareID)*. The call immediately returns, and eventually a message containing the data associated with the given key is sent to the specified chare at the specified *entryPoint*.

Synchronization characteristics:

The synchronization requirements of a Charm program are not easily available either statically or dynamically. However, information about object-level synchronization can be acquired automatically in Dagger [5], which is a high-level notation on top of Charm. Dagger allows the user to easily express synchronization even in asynchronous message-driven execution models. Further, system-level (across all objects) synchronization can be identified if a known library such as the reduction library is used.

Granularity, scheduling, load balance:

In the Charm execution model, all *new-chare* and *for-chare* 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 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.

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

[2]Note that in a dynamically load balanced system, such as the one provided by Charm, a message can potentially

allows us to determine the grain-size of tasks, the number of messages in a processor's message pool, and the utilization of each individual processor. The last two quantities provide information about the balance of message-load and the balance of processor-utilization.

The Charm runtime system allows the user to choose from a wide variety of scheduling strategies, such as lifo, fifo, fifolifo, etc. It allows the user to add further control on the scheduling of messages on a processor by attaching priorities to messages. The system has various prioritized scheduling strategies, again user-selectable, that schedule messages according to their priorities. The choice of a particular scheduling strategy is made at link-time, so the runtime system has information about the strategy chosen.

5 Intelligent performance feedback: Projections

In this section, we outline the design of Projections, the display and analysis component of a framework for providing intelligent feedback about the performance of Charm programs. The Projections performance analysis environment has two components: display and analysis. Both the components are post-mortem: they use traces of the program execution. Projections traces can be obtained automatically by linking with the projection binaries: no change is necessary to the user program. The first component is a tool to display Charm specific data. The second component of the performance analysis environment analyzes information about the various characteristics of a parallel program and provides the user with feedback about the performance of the program. We describe these components in more detail in this section.

5.1 The display component

A preliminary version of the display component of Projections was presented in [2, 3]. 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, and creation of new tasks. Charm is based on the MPMD (Multiple Program Multiple Data), while most currently available programming tools target the SPMD (Single Program Multiple Data) model. Thus features unique to the MPMD model, such as the creation of new processes need to be displayed in Projections.

2. *Program specific performance information.* Projections allows the user to view information about the creation, processing, granularity of messages to entry points. Since messages in Charm are addressed to specific entry methods in chares, providing information about messages using the reference point of the corresponding chare and entry method will provide the user with information which can be related to relevant portions of the parallel program.

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 most basic Projections views treat program attributes

be enqueued and dequeued more than once.

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. 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 program to define finer and coarser stages, as desired. The 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.

The attached plate 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*. An *overview* shows an aggregated (added across all processors, or across time) summary of the values of various attributes. A *detailed* view shows a complete view of all attributes either across processors or across time. One can also query inside the detailed view to get a *timeline* [3] of events occurring on the chosen set of processors for the chosen period of time. This view is useful in understanding what happened on which processor at what time. Note that both the overview and the detailed view has an *View-User-Parameters* button in the menu. This menu item allows the user to select user-defined attributes. It 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 three chares: *main*, *RB2*, and *REDN*. And the chare *RB2* has entry points: *red_ws*, *red_en*, *init*, *black_wn*, *black_es*, and *dag9*. Using these, the user can display information about the creation and processing of messages to any of the entry points.

5.2 The analysis component

Based on our experience in analyzing the performance of several Charm programs, we have identified a decision-tree based approach for performance analysis. Figure 2 shows part of the decision-tree. Notice how different nodes of tree correspond to acquiring information about the program characteristics that we have talked about in Section 4. The first two levels of the tree are used to identify when and where the program performed poorly: this identifies periods in time and sets of processors for which the program did not perform well. The lower levels of the tree are used to determine the exact nature of the cause of poor performance. A portion of the lower levels of the tree appears in Figure 2: the triangles indicate positions in the tree which have further branches.

A performance analysis tool should embody this expertise and automate its application to the execution of a particular program. It should have the ability to perform individual analysis required at the various nodes of the tree. Projections is intended to be such a tool. In Section 4, we have discussed how one can acquire information about the characteristics of a Charm program. Based on such information and the decision-tree in Figure 2, Projections can analyze the execution of Charm programs. Our current analysis component consists of critical path analysis and some granularity and placement analysis. We are in the process of implementing other portions of the decision-tree.

[3] This view is inspired by the performance tool developed at the Argonne National Laboratory called Upshot [6].

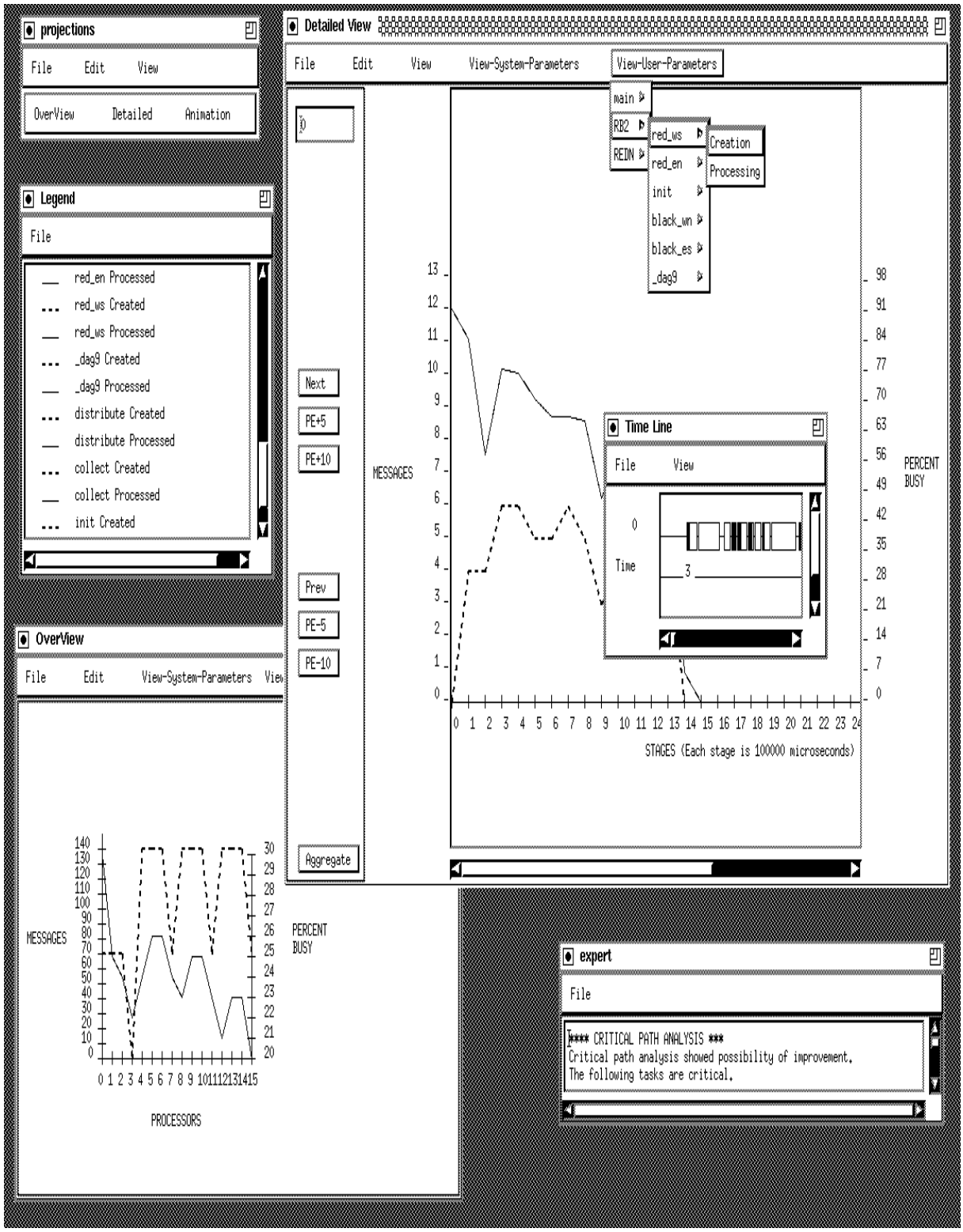

Figure 1: **A sampling of Projection views**

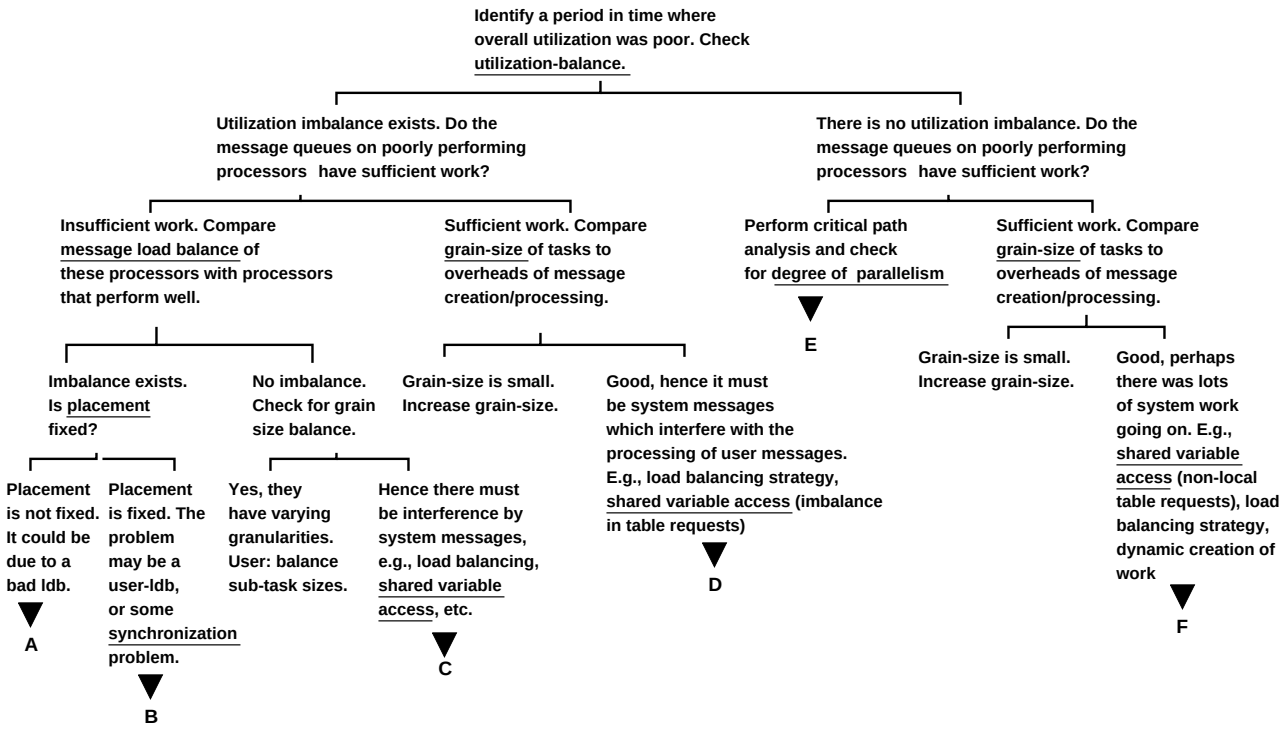

Figure 2: **Framework in which all performance data is to be analyzed.**

The process of analysis is designed to be iterative. A pass of the analysis component will provide the user with feedback on possible modifications to improve the performance of the program; the user after making appropriate modifications will invoke the analysis component again to get more suggestions, and the process of analysis-modification continues. We believe that the end-result of this iterative analysis-modification process will be an efficient parallel program. To illustrate the kinds of analyses that can be performed, we shall discuss in greater detail two of the branches of the decision-tree.

Branch C (Figure 2)

One of the concerns in branch C is whether shared variable access is a source of problem for the performance of the program. The usage of one of the five information sharing mechanisms available in Charm, 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. Some of the performance concerns that could be addressed when one knows the nature of information sharing (through specifically shared variables) in a Charm program are [4]:

- If distributed tables were being used, then it would be necessary to check whether the distribution of keys and access of table entries were uniform over all processors.
- Monotonic variables are often used in speculative computations, where the speculative component could depend to a large extent on the value of the monotonic variable. In such cases, it would be useful to provide the user with information about the speculative component of computation and when updates were made to the monotonic variable.

[4] Note that all the concerns listed do not correspond to load imbalance. This is because we are describing the module that's called for *all* shared variable access analysis, and only some of those cause load imbalance.

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

- 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.

- If a monotonic variable is updated frequently, the *spanning tree* [4] implementation should be chosen. However, if it is updated rarely then the *flooding* [4] implementation should be chosen.

- If a large number of entries in the distributed table are accessed only once, and if the entries were located on a processor distinct from the one that inserted it into the table, or the one that accessed it, the cost of insert is the cost of 1 message and the cost of an access is the cost of 2 messages. The cumulative costs of insert and access could be reduced if the entry was inserted into the table for the processor which made the insert request.

A performance analyzer that has an inventory of such concerns, would check programs for the existence of one or more of these concerns. If one of them did exist, the performance analysis could suggest a method to improve performance.

Branch E (Figure 2)

One possible analysis in branch E is critical path analysis. The critical path is the longest chain of computation in a program. It determines the set of tasks which affect the execution time most severely, and hence need to performed more efficiently.

We define tasks on the critical path of the execution of a program as *critical tasks*, and others as *non-critical tasks*. In the context of Charm, where messages drive execution and messages are scheduled for execution on the basis of a user-selected strategy, it is possible to shorten the length of the critical path by giving preference to the tasks on the critical path, i.e., if critical tasks are allotted higher priority than non-critical tasks in the scheduling scheme. Our experience shows that the above technique works only if the following conditions are met:

1. There are critical tasks whose execution is preceded by idle periods. This criterion establishes the possibility of improvement, because one could schedule the execution of non-critical tasks during these idle periods (of course, without causing other idle periods), thereby increasing processor utilization.

2. There are critical tasks which wait to be scheduled while other non-critical tasks are executed. This criterion determines non-critical tasks whose execution could be scheduled later when processors are waiting for critical tasks to exeecute.

Critical path analysis in Charm consists of three components: the first component determine the critical path in the program's execution, the second component checks whether there are any critical tasks which were scheduled for execution on idle processors, and the third component checks for non-critical tasks which were executed while critical tasks were non scheduled for execution. In the next section, we present a sample of critical path analysis for a program.

6 Example: Gauss-Siedel application

We considered an application which tries to solve n independent sparse penta-diagonal systems using the Gauss-Siedel (red-black) iterative method. 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 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 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 3 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 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 most of the same tasks were also non-critical because the program used the same object to solve all the systems. Therefore the analysis could not suggest an effective prioritization scheme — *CharmInit* and *Init* are initialization entry methods and are executed only once. Of course, there is no prioritization scheme applicable for this program, because it would be necessary to prioritize that system which takes the longest to be solved; however this cannot be done without knowing beforehand the system that would take the longest to solve.

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, such as that provided by the Dagger notation.

7 Discussion and future work

In previous work, Jamieson [7] 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 [8], have developed an approach called the W^3 model, which attempts to reduce the amount of data traced for parallel

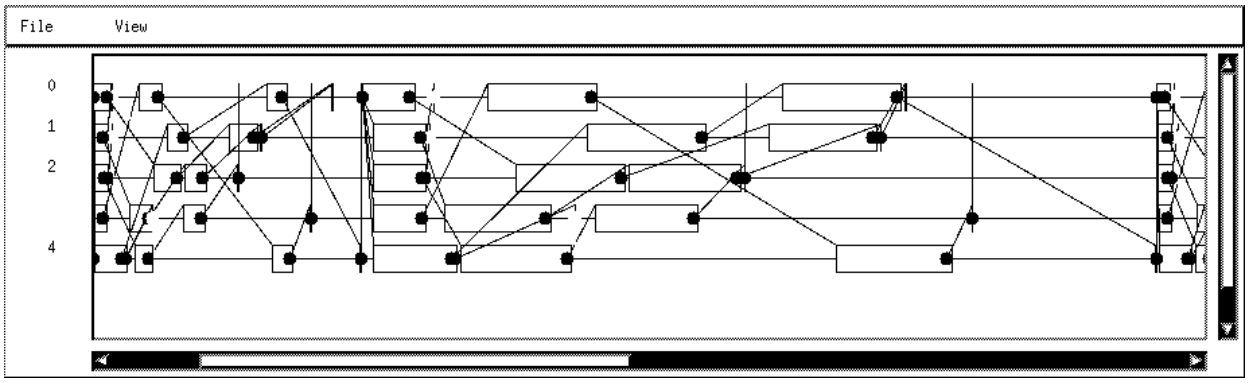

Figure 3: **The figure shows timelines for some of the processors for stages 7 and 35 of the program execution**

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.

Even though the size of the trace information needed by us is significantly smaller than those necessary for shared memory programs, it is still large enough for concern. In future, we plan to explore avenues for reducing the amount of trace information needed. One possible technique is to acquire information necessary for replaying a Charm program, and then progressively acquire more detailed trace information about the program as needed by replaying the execution of the program, and tracing only the events needed.

In future work, we need to develop acquisition mechanisms further to acquire information about synchronization characteristics and the degree of speculative computation. We plan to use the Dagger notation to acquire information about task-level synchronization in the program. We are still developing mechanisms to acquire information about speculative computations in a program. Once information about synchronization and degree of speculative computation is available, more analysis of the program will be possible.

References

[1] Kale L.V. The Chare Kernel Parallel Programming System Programming System. In *International Conference on Parallel Processing*, August 1990.

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

[3] L. V. Kale and A. B. Sinha. Projections: a preliminary performance tool for charm. In *International Parallel Processing Symposium*, April 1993.

[4] L. V. Kale and A. B. Sinha. Information sharing in parallel programs. In *International Parallel Processing Symposium*, April, 1994.

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

[6] Terry Disz and Ewing Lusk. A graphical tool for observing the behavior of parallel logic programs. Technical Report CSRD 746, Argonne National Laboratory, February 1988.

[7] Leah H. Jamieson. Characterizing parallel algorithms. In Leah H. Jamieson, Dennis Gannon, and Robert J. Douglass, editors, *The characteristics of parallel algorithms*. The MIT Press, 1987.

[8] Jeffrey K. Hollingsworth and Barton P. Miller. Dynamic control of performance monitoring on large scale parallel systems. In *International Conference on Supercomputing*, July 19-23 1994.