Projections: A Preliminary Performance Tool for Charm*

Amitabh B. Sinha
Department of Computer Science
University of Illinois
Urbana, IL 61801
email: sinha@cs.uiuc.edu

Laxmikant V. Kalé
Department of Computer Science
University of Illinois
Urbana, IL 61801
email: kale@cs.uiuc.edu

Abstract

The advent and acceptance of massively parallel machines has made it increasingly important to have tools to analyze the performance of programs running on these machines. Current day performance tools suffer from two drawbacks: they are not scalable and they lose specific information about the user program in their attempt for generality. In this paper, we present Projections, a scalable performance tool, for Charm that can provide program-specific information to help the users better understand the behavior of their programs.

1 Introduction

Performance is the reason for the existence of parallel computers. Machines with impressive *peak performances* - in the range of tens to even hundreds of gigaFLOPS - already exist today. However, 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 magnitudes 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 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. To improve the performance of a particular parallel algorithm, 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 are therefore crucial to improving the performance of parallel programs.

Currently, two issues are considered important in the design of performance tools for parallel programs: generality (the tool should be applicable to a wide range of paradigms) and scalability (the tool should provide comprehensible feedback for a large number of processors). We consider a third issue to be important in the design of a performance tool: specificity (the tool should provide information about specific features of the application program). A specific performance tool can provide the user with feedback in terms of attributes that the user can identify from the source program. The user can then identify and remedy the segments of the program responsible for poor performance. However a performance tool can achieve specificity only at the loss of generality; therefore it is important that a specific performance tool should be usable in conjunction with existing general-purpose tools.

Charm [4, 5] is a portable parallel programming language for MIMD machines. The system is described briefly in Section 2. In this paper, we will describe a performance feedback tool called Projections for Charm. We will describe the features provided by the tool, and illustrate with an example how the specificity of Projections helps in improving performance of Charm programs. Projections addresses the issue of scalability by providing aggregated summary displays.

There is a significant body of prior research on performance tools. In Section 3, we discuss some of the relevant work, and provide the motivation for Projections. In Section 4, we describe Projections. The use of Projections is illustrated with an example in Section 5. Projections is an evolving system. Some of the future enhancements are described in Section 6, including features that improve its scalability.

2 Charm

Charm is a machine independent parallel programming language. Programs written in Charm run unchanged on shared memory machines such as Encore Multimax and Sequent Symmetry, nonshared memory machines such as Intel i860 and NCUBE/2, UNIX based networks of workstations such as a network of IBM RISC workstations, and any UNIX based uniprocessor machine.

The basic unit of computation in Charm is a *chare*. A chare, created dynamically, has associated with it a data area, and some entry functions that can access this data area.

A new chare is created using the *CreateChare* system call. As a result of this system call, a *new-chare*

*This work was funded in part by NSF under grants CCR-90-07195 and CCR-91-06608.

message is created, which is at some later point in time picked up for execution by the system. New chare messages float among the available processors as the system moves them around in an attempt to balance load. Once picked up for execution, a new chare message results in the creation of a new chare, which is subsequently anchored to that processor. Messages can be addressed to existing chares using the *SendMsg* system call. This call generates *for-chare* messages.

The only other type of process in Charm is a *branch-office chare*. A *branch-office chare* has a representative chare on each processor. Chares can interact with the local representative of a branch-office chare by invoking entry functions in the branch chare using the *BranchCall* system call. The branches interact with each other using the *SendMsgBranch* and *BroadcastMsgBranch* system calls — as a result of these calls *for-boc* messages are generated.

Data is shared among different chares as well as branch office chares through messages and five types of specifically shared variables, namely: readonly, writeonce, accumulator, monotonic and distributed tables. The implementation of these five specifically shared variables is tuned to the architecture of individual parallel machines, hence they provide an efficient means to share data.

The description of the Charm runtime system indicates that there are numerous system-specific entities, and overlooking such specific knowledge about the system would result in an incomplete understanding of program behavior.

3 Previous work

There has been substantial work done previously on tools to analyze and understand the behavior and performance of parallel programs on parallel machines. In this section, we examine Paragraph and Upshot, two general-purpose performance tools.

ParaGraph [2] 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)[1].

Upshot [3] 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.

ParaGraph and Upshot, although useful in many contexts, suffer from two drawbacks: (1) they are not targeted for the execution model or the programming abstractions in Charm, and (2) they are not scalable. In the remainder of this section, we discuss these issues in greater detail.

3.1 Specific feedback

Many different models for parallel computation have been proposed. Figure 1 shows a classification of execution models. Parallel execution models can be broadly classified either as SIMD or MIMD. The class of MIMD algorithms can be separated into two different paradigms: SPMD (Single Program Multiple Data) and MPMD (Multiple Program Multiple Data). SPMD and MPMD can be further separated into synchronous message passing[1] and asynchronous message passing[2] paradigms respectively. Synchronous and asynchronous paradigms can be further classified as static (processes are created statically) or dynamic (processes can be created dynamically).

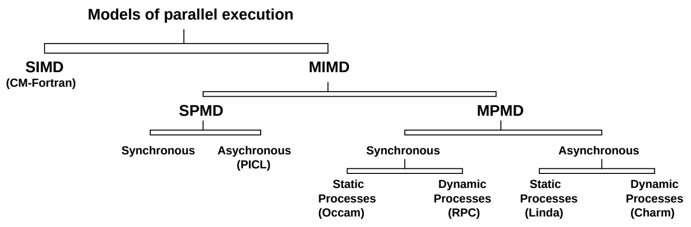


Figure 1: **The paradigms of parallel computation.**

Most of today's parallel applications are written in the SPMD model of parallel computation. Other models of parallel computation offer advantages over the SPMD model of computation. The MPMD programming model allows for multiple processes and their dynamic creation, and permits the user to obtain a (possibly) greater degree of parallelism. The downside of having multiple and dynamically created processes is that MPMD models are even more difficult to program and understand than SPMD models Existing programming tools, such as ParaGraph and Upshot, are geared towards the SPMD model of computation — they do not provide any information on the creation of new processes, or the inter-leaving of the execution of many different processes on the same processor, etc.

In addition, neither Paragraph nor Upshot are specific to any language. This generality permits them to be used flexibly with many different languages. However with this generality comes the loss in specific information about a program's execution model. These performance tools are based on a very general message passing model — processes executing on different processors and communicating through messages. The runtime system for Charm programs provides features

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

[2]A *send* event in a process can happen without the corresponding *receive* event being executed on another process.

to share data and automatically balance load. General purpose performance tools would lose such program-system specific information, and therefore not give the user sufficient information to understand the behavior of the program. 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 should 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.

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.

3.2 Scalability

Upshot and ParaGraph cannot be used effectively, or are at best awkward to use for analyzing performance of programs running on large parallel machines with hundreds or thousands of processors. Upshot provides a microscopic view of the computation by displaying each event as it occurs on the processor's time-line (users can "zoom" out of this microscopic view, but only to a limited extent). ParaGraph, also, focuses on details by animating each message passed in the system, either as an animation view, or as a spacetime diagram. These views are useful when there are few processors and the communication pattern is regular (e.g. and FFT on a grid algorithm). However, when the number of processors are many, the user is unable to absorb the details in these dynamic, detailed views. Projections provides both summary and detailed views of program execution, allowing the user to examine the entire program in summary form, and then examine specific (interesting) areas of program execution in more detail.

4 Projections

Every parallel application program has its own characteristics. A knowledge of these characteristics can make debugging and performance analysis more accurate, e.g., for a parallel application which has no synchronization requirements, the performance analysis tool should look for good load balance and adequate grainsize of tasks, and not for patterns of message passing. Ideally the characteristics of application programs would be implicit in the language features used in the programming of the application. The current parallel languages provide only limited information about the characteristics of the parallel application.

Charm provides a great deal of information about the characteristics of a program, and we believe that this information would enable us to make more specific analysis of the program. A long-term of our research (see Section 6) is to develop an intelligent and automated performance analysis tool for Charm. As a first step, we have identified that the type of messages (new-chare, for-chare, or for-boc) and the distinction between the creation and processing of messages, can provide the user with more information about the program execution.

In this section, we discuss how data about the type of messages and their creation and processing is collected and displayed. We also discuss how trace data for Projections can be transformed to obtain trace data for Upshot.

4.1 Trace data collection

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 *record* 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. By re-linking a program with the appropriate libraries, the user can generate trace information for the program on any machine on which Charm has been installed.

Each processor has its own local buffer to record trace data for messages created and processed on it. Data is recorded in these local buffers while the program is executing. If a buffer is about to overflow during an execution run, then 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.

4.2 Trace data format

There are currently five types of events which are recorded in trace files when a Charm program is executed in the *record* mode:

1. Initiation of the program on a processor.

2. Termination of the program on a processor.

3. Creation of a message. Messages are processed at some later time. A message can be one of a *new-chare*, a *for-chare* or a *for-boc* message.

4. Start of processing of message on a processor. Since all the system calls in Charm are non-blocking, no other message can be picked up for execution on this processor before this message is completely processed.

5. Finish of processing of message.

The following three fields are recorded for each event:

1. *activity type*: The activity type of this event from among the above seven activity types.

2. *processor*: The processor number on which the event occurred.

3. *time*: The local time (in microseconds) when the event occurred.

The following two additional fields are recorded for the creation and processing of each message:

1. *entry*: The entry function for which this message is intended.

2. *msg type:* The type of the message, i.e., new-chare, for-chare, etc.

4.3 Trace data display using Projections

The nature of execution of a Charm program can be well understood by examining the values of the number of creations and processings of *new-chare*, *for-chare* and *for-boc* messages, and the percentage busy time of processors. These values provide an indication for when, where and what type of work is created during a program execution, and how this corresponds to the overall performance of the program in terms of the percent busy time. A processor is determined to be busy during the period of time it executes an entry function corresponding to a chare or a branch-office chare. 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 data obtained by running the program in *record* mode can be displayed in different ways to provide the user with many views of the performance of the program. The most basic views treat program attributes, e.g., creation of new-chare messages, 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: stages and processors. 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 first set of views are called the *overview* views. In these views, the user can select program attributes to be viewed. The values of the selected program attributes over the selected *stage sets* and *processor sets* are shown as one screenful of data for an *overview* of the program execution. There are two types of overviews. In the first, the chosen program attribute is summed up over all processors in the processor set for each stage, and this aggregate is displayed as a function of the stage number (so, the function displayed is $D_a(s) = \sum_p F_a(s,p)$). In the second, the chosen program attribute is summed up over all stages in the stage set for each processor, and this aggregate is displayed as a function of the processor number (so, the function displayed is $D_a(p) = \sum_s F_a(s,p)$). These views are very useful in selecting interesting stages and processors for a more in-depth view. In either of these two overviews, the user can interactively change the program attributes to be selected, and the stage and processor sets for the views. By changing the stage and processor sets the user can selectively look at specific periods of program execution, or only at specific processors. Since the views are limited to a single screen full of information there is a consequent limit on the detail that can be presented in the view. This is the motivation for the second type of views.

The second type of views are called *in-depth* views. There are four in-depth views:

1. View for a particular stage, s_0, ranging over the selected processor set, $F_a(s_0,p)$.

2. View for a particular processor, p_0, ranging over the selected stage set, $F_a(s,p_0)$.

3. View for the aggregate over all stages in the particular stage set, ranging over the selected processor set, $\sum_s F_a(s,p)$.

4. View for the aggregate over all processors in the particular processor set, ranging over the selected stage set, $\sum_p F_a(s,p)$.

The last two types of in-depth views are very similar to the overviews, the major difference being that the display is not limited to one screenful of information — information is displayed over several "pages", and the user can browse through these pages as desired. In all these views the user can interactively choose stage sets and processor sets over which the program attributes can be viewed. In addition the length of a stage can also be interactively changed, so that the views can be made more detailed, or more coarse. The limit of the amount of detail depends on the granularity of the program whose performance is being measured. These views are completely scalable — they are not limited by the number of processors or the length of execution time of a program to be viewed.

4.4 Links with other display tools

The trace data obtained for Projections is fairly general, and can be transformed in a simple way to obtain trace data in the format required by other more general strategies. We have done this transformation for Upshot, so that the trace data obtained for Projections can be transformed to obtain the trace data format for Upshot. The data format for Upshot includes an initial header file specifying number of processors, number of different event types, and other information, and a tail part which contains a record of events occurring on various processors at various times. The transformation consists in making the beginning and finishing of processing of an entry point as two different events, and then using this pair of events to denote the state corresponding to the execution of the entry point.

5 Understanding program behavior

In this section, we have chosen an example application to illustrate the use of Projections in understanding Charm program behavior. The analysis is carried out with *overviews* for this example, even though more detailed information can be (and was) obtained using the in-depth views, it hasn't been used here, because of space limitations. However, *overviews* are sufficient for the analysis we will present.

The Traveling Salesman Problem (TSP) [9] 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. [6]. 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.

The TSP application was executed on 16 processors of an NCUBE/2 with the ACWN [10] (adaptive contracting within neighborhoods) load balancing strategy. The execution with ACWN took about 29 seconds with a total of 7131 partial solutions being generated. The optimal solution was found at 14.4 seconds. Figure 2 show overviews of new chare creation and processing over stages for the ACWN.

In Figure 2, 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 system *useful* messages even after the best solution was found. As the processor utilization is close to 100% prior to this time, this must have happened because many *useless* messages were processed before the solution was found. In a sequential implementation, messages are processed in ascending order of their costs, and after the best solution is found no *useful* messages remain. In our execution run with the ACWN load balancing strategy, *useful* messages exist even after the best solution is found, because new chare messages are not processed in order of their costs. This suggests that the load balancing strategy being used with this application did not balance the priorities of new chare messages while balancing their loads.

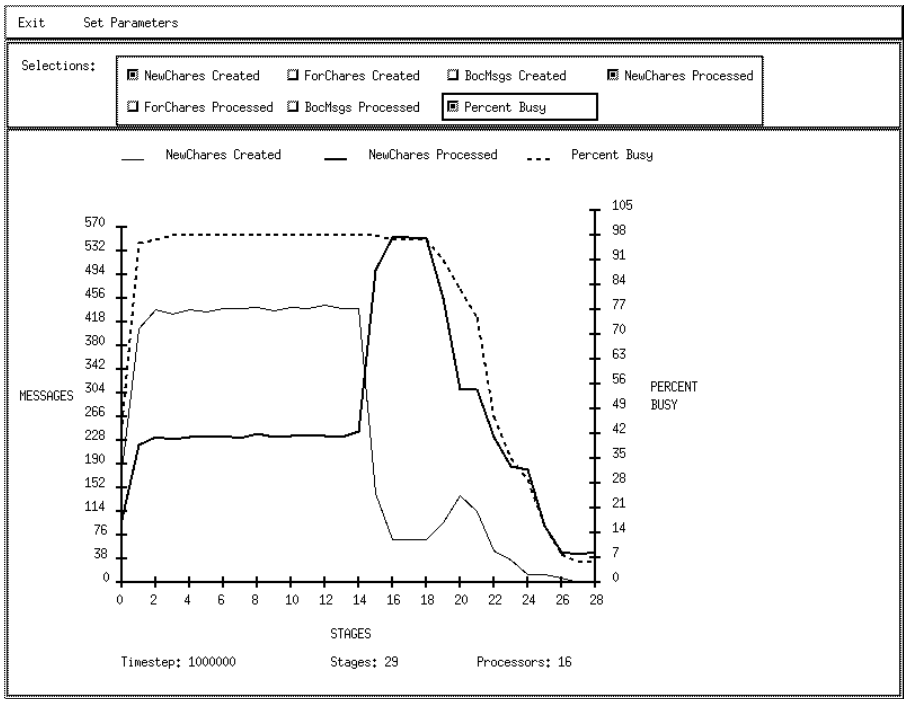
So we ran the TSP application on 16 processors with another load balancing strategy called the Manager strategy [11], which balances both load and priorities across processors. The execution with the Manager strategy took 21 seconds, and a total of 5785 partial solutions were generated. The optimal solution was found at 9.6 seconds. Figure 3 show overviews of new chare creation and processing over stages for the Manager case. Note 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.

6 Discussion and future work

We have presented Projections, a scalable performance tool for a machine independent parallel programming language, Charm. We have illustrated with an example how Projections can be used to make specific analysis of program behavior. We were able to analyze the suitability of different dynamic load balancing strategies, provided by the runtime system, for a particular application. We have also shown how the trace information for our performance tool can be transformed with ease into trace information for Upshot, thus allowing the user to access the views provided by Upshot. Hence we can, with the same trace information, provide access to general purpose tools such as Upshot and very specific tools such as Projections. These other general purpose tools do not provide the views that our performance tool does, and vice versa, so they complement each other.

Pablo [7, 8] 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. 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.

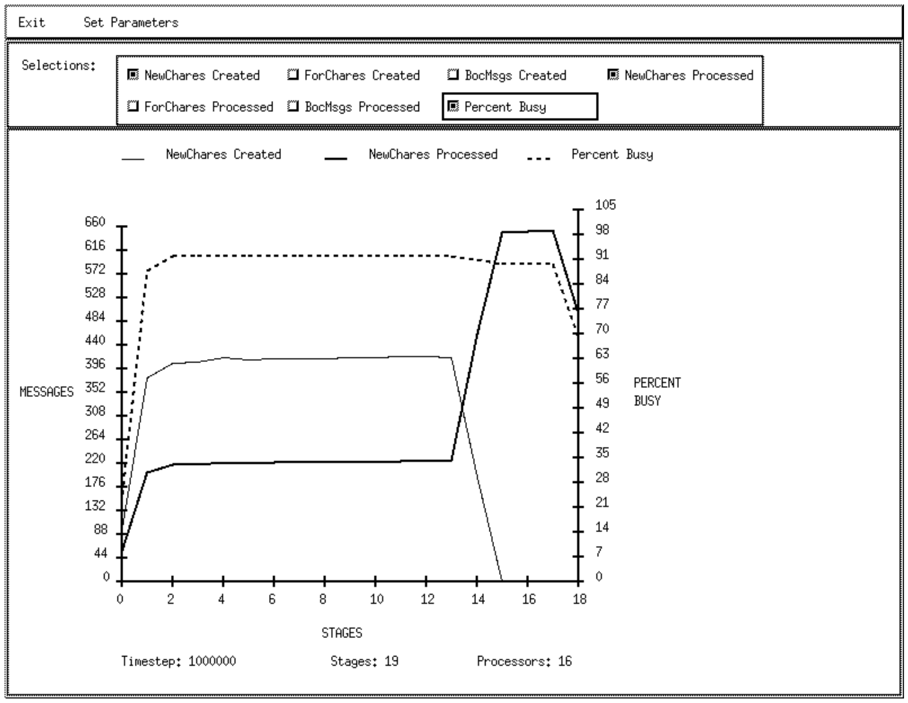
In future, we plan to deal with performance attributes relating to specifically shared objects, message queues and load balancing strategies. For example, an examination of the access patterns of items stored in distributed tables would help us understand whether or not the items were distributed uniformly (if not a different key may be necessary), and whether or not accesses from different processors were uniform. A knowledge of the distribution of the values of the



Selections: ✓ NewChares Created ☐ ForChares Created ☐ BocMsgs Created ✓ NewChares Processed ☐ ForChares Processed ☐ BocMsgs Processed ✓ Percent Busy

Timestep: 1000000 Stages: 29 Processors: 16

Figure 2: **This figure shows the overall efficiency, number of chares created and number of chares processed over the various stages of the execution of a branch&bound implementation of the TSP program on an NCUBE/2 with the ACWN load balancing strategy.**



Selections: ✓ NewChares Created ☐ ForChares Created ☐ BocMsgs Created ✓ NewChares Processed ☐ ForChares Processed ☐ BocMsgs Processed ✓ Percent Busy

Timestep: 1000000 Stages: 19 Processors: 16

Figure 3: **This figure shows the overall efficiency, number of chares created and number of chares processed over the various stages of the execution of a branch&bound implementation of the TSP program on an NCUBE/2 with the Manager load balancing strategy.**

priority of the top element in the case of prioritized queuing strategies would be useful in understanding how the execution of an algorithm was proceeding over a prioritized search tree. The richness and specificity of programming constructs in Charm will help us in developing Projections into a more informative performance analysis tool.

In the long-term, we believe that a much more intelligent and automated performance analysis tool is possible if the language constructs provide enough information about the program behavior. Such a tool will be built for Charm in the following steps:

- By identifying parallel application characteristics, whose knowledge would help in the process of debugging and performance analysis
- By providing mechanisms which allow the user to specify the program characteristics identified above either implicitly or explicitly
- By incorporating commonly used program characteristics into Charm

We are also examining 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). This sort of display constrains 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 subranges. 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.

References

[1] 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.

[2] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, pages 29–39, Sept. 1991.

[3] Virginia Herrarte and Rusty Lusk. Studying parallel program behaviour with upshot. User Manual for Upshot.

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

[5] L. V. Kale, *et al.* The Chare Kernel Programming Language Manual. Internal report.

[6] 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.

[7] 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, 1989.

[8] 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.

[9] Edward W. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

[10] W. Shu and L. V. Kale. A dynamic load balancing strategy for small-grained processes. In *Supercomputing*, November 1989.

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