Theor Chim Acta (1993) 84: 377–384

Theoretica Chimica Acta

Performance visualization for parallel programs

Ewing Lusk*
Argonne National Laboratory, Argonne, IL 60439, USA

Received October 1, 1991/Accepted December 5, 1991

Summary. We describe here a set of graphical performance visualization tools that have been developed at Argonne National Laboratory for increasing one's understanding of the behavior of parallel programs

Key words: Parallel programs – Graphical performance visualization

1. Introduction

Parallel programming is still a highly experimental science in which the designs of both user and system programs are undergoing simultaneous study. System implementors tune systems in response to experimental data from users, and users try to understand the effects of decisions made for them by system implementors as well as the effects of variations in the parallel algorithms being executed by their programs.

Collecting timing statistics and measuring speedups is often insufficient for understanding why the results are what they are. This is because in a sequential program, we know the sequence of events, whereas in a parallel program, not only is the precise sequence of events unknown to us, but it changes from one run to the next. In most cases, we have some expectation of the rough sequence of events, the overall "behavior" of the program, but in the parallel case it is very difficult to deduce from readily available statistics whether the program actually behaved according to our expectations or not. A typical situation is one in which our intuition has told us that near-linear speedups should occur, but execution times (easy to measure) tell us that we are not getting them. It is often not at all clear what to do next.

* This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38

2. Performance visualization tools

A family of tools for helping with this problem is just beginning to evolve. Unlike profiling tools such as Gauge [5], these tools try to capture the precise sequence of events occurring during program execution as opposed to counting those events. A minimal amount of data about each event is captured in a logfile of some kind, and the log is then examined in post-mortem fashion. Real-time display of events being logged is possible, but usually unproductive, because the subsequences one is interested in occur so rapidly.

Implementation of such tools raises a number of issues:

- The log must be captured with an absolutely minimal impact on the performance of the program. Otherwise the insights gathered by examining the log will not really apply to the production version of the program. This means buffering of events in memory, dumping to external storage without stopping execution, and little or no forced synchronization among multiple processes.

- The precise nature of the events to be logged is still very much a matter for debate. Different systems will have different "critical" system events, and of course different user programs will have different events altogether. Logging of all low-level events, such as all locking and unlocking operations, or all messages sent and received, while sometimes useful, may swamp the logging mechanism without providing real insight into parallel program behavior. Flexibility in specifying which events are to be logged is crucial.

- Given a mechanism for efficient logging and a decision about just what events should be logged, it remains to find a display mechanism that promotes reconstruction of the sequence of events and an understanding of how it was caused by the program specification. Both static and dynamic displays have been used, and each approach has its advantages. The fact that we are focusing here on parallel programs almost mandates a graphics rather than a text display. Of course implementation of such a program currently involves one in decisions about graphics languages, window systems, etc.

Many researchers are taking up these challenges. One of the most advanced systems in this category is Paragraph [6], a logfile display program developed at Oak Ridge National Laboratory. In general Paragraph provides more views of logfile information than the tools described here, although these systems provide more depth in the views they do provide.

3. Layers of parallel programs

Figure 1 shows some of the layers that may appear in the structure of parallel application programs. They provide a context in which to describe some of the tools.

At the bottom layer are the machines we are trying to utilize. Here we assume they are parallel machines, chosen to provide either current maximum performance or to provide an environment for the development of programs to run on the fastest machines of the future. They come in a variety of architectures. Those we are using at Argonne include a Sequent Symmetry, BBN TC-2000, Intel IPSC/860, and networks of workstations from Sun, Next, IBM, and Silicon Graphics. We are preparing our tools for use in the Intel Touchstone Delta, with 520 i860 nodes.

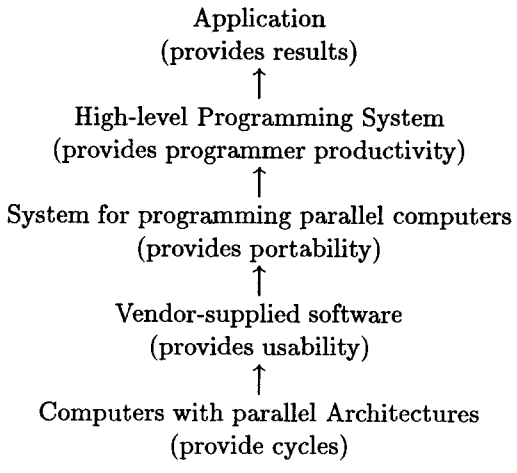

Fig. 1. Layers of a parallel application

The next level up consists of the vendor software to express and control parallelism, necessary in order to use the machines. Since standards for such primitives do not exist, and the vendors compete with each other in offering different programming models and primitive operations, this software is portable neither syntactically nor sematically. That is, not only do vendors offer different names for similar functions, but in many cases radically different models of computation and slightly different semantics for very similar operations. This is not a criticism; at this stage in the development of parallel programming paradigms it is necessary to explore alternatives competitively.

Nonetheless, with software lifetimes increasing and hardware lifetimes decreasing, it is more clear than ever that one is liable to be developing software on one machine which will be run on another. Thus portability becomes an important concern in any project, and a number of systems for writing portable programs by hiding the vendor software beneath a layer of machine-independent primitives that implement a portable computational model have evolved. At the lower level we are discussing here, such systems are typically subroutine libraries for low-level languages such as C and Fortran. One such system developed at Argonne is described in [1]. It has served as the foundation for several other related systems, and for the current such system under development at Argonne, called **p4**.

The next layer up represents the attempt to bring to parallel programming the benefits of high-level languages. Such languages provide the usual programmer productivity benefits and when combined with chunks of sequential code written in low-level languages for efficiency, need not have a negative impact on performance. At Argonne we are using two systems based on logic, PCN [2] and the Aurora Parallel Prolog system [7].

At the highest level there are application-specific systems, which hide all of the layers below from the end user. We will not be concerned with this layer here.

4. Some program visualization systems

In this section we describe some of the systems that have been developed and used at Argonne National Laboratory to better understand the behavior of

parallel algorithms. They have been developed in conjunction with real systems and applications, in order to understand the results of design decisions in those systems. In every case they have had a major impact on the algorithms and ultimately on the efficiency of these systems. They occur at both the low-level portability layer and the high-level parallel language layer in Fig. 1. The tools are **upshot**, which displays events and states in parallel time lines, **ravel**, which animates message-passing programs, and **wamtrace**, an animation system for parallel Prolog.

4.1. Collecting log information

There is no need for a log collection mechanism to be very tightly integrated with the log display mechanism. Both **upshot** and **ravel** use the **alog** package for creating logfiles. It consists of a set of C macros and Fortran-callable subroutines for logging arbitrary events, with event types arbitrarily by the programmer. An event consists of an event type, a process id, a timestamp, and one integer and one string of data. Events are collected in local memory by each process and only written to disk when the run is complete. After the run is over the separate logfiles are merged on the timestamps to produce a single properly sorted logfile. Optionally, the user prepares a second file defining process states by specifying an entry and exit event for each state. These two files are the input to both **upshot** and **ravel**.

4.2. Upshot

Upshot (Fig. 2) shows a horizontal time line for each process, with colored bars to represent different states. It is possible to scroll smoothly through time with scrollbars at the top of the display.

One can also mouse-click on specific events to pop up data boxes that show the rest of the data logged in the event. We have found this type of display particularly valuable when the parallel program contained work units of widely varying grain size and changed its behavior during the run. Our parallel automated reasoning program **Roo** [8] is an excellent example of this type of program. **Upshot** has been used to trace programs written in Strand [4], PCN [2], Prolog, and Fortran, as well as C.

4.3. Ravel

Ravel is somewhat specialized to message-passing programs, and is used for showing the patterns of message flow as well as the lengths of message queues, in addition to the states of processes.

In Fig. 3 we see **ravel** displaying the very same logfile as displayed by **upshot** in Fig. 2. The display shows sixteen processes working on a solution of a Dirichlet problem in which each process carries out an update of its region of the grid, exchanges boundary information with its neighbors, and then carries out the next iteration. At the moment, process 6 is sending boundary information to process 5, processes 1, 7, and 18 can be seen by their color to be computing, and most of the other processes are waiting to receive messages. The small numbers

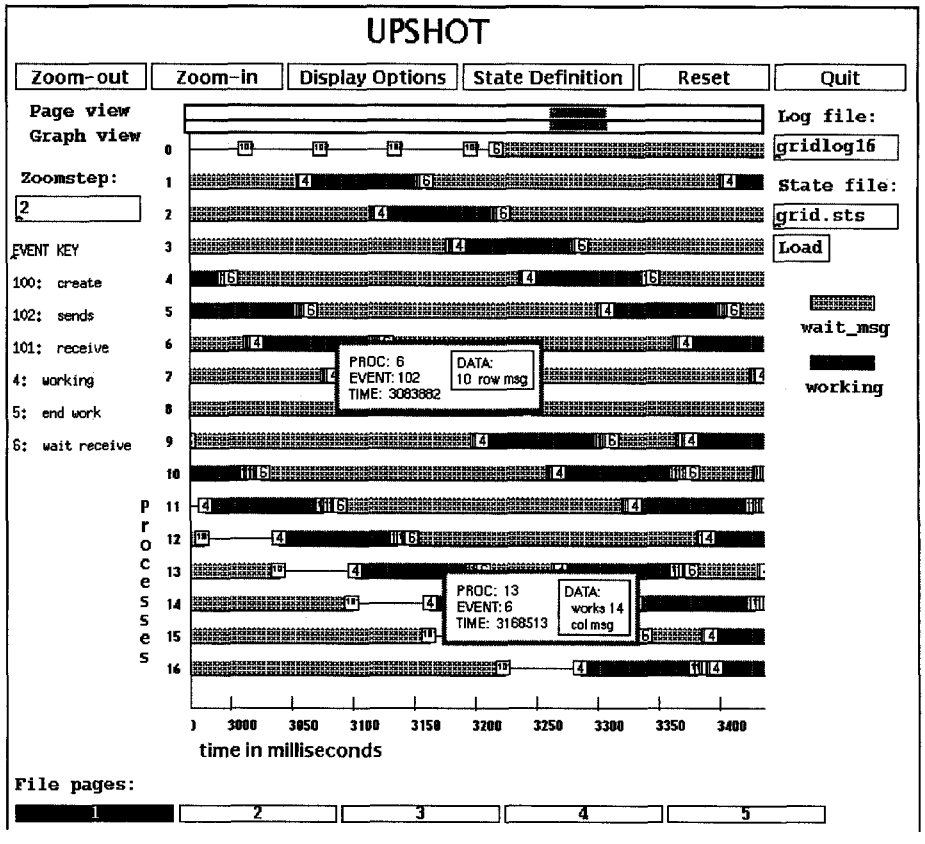

Fig. 2. Upshot view with events and popup data boxes

beneath the processes give the length of the message queues. In some problems these message queue lengths provide valuable clues about bottlenecks in the computation.

It is possible with **ravel** to move the processes around with the mouse to achieve a pattern of processes that is particularly useful for understanding the message flow. Here the processes have been arranged into a grid that reflects the portions of the grid they are responsible for.

4.4. Wamtrace

Wamtrace [3] is at the high-level programming system layer, since it is tied to a particular programming system.

It is portable across machines, but specialized to one particular parallel programming mechanism, the Aurora parallel Prolog system [7]. Its logging mechanism does not depend on a microsecond timer, but rather uses shared memory to sequence events in a pair of buffers. Events are buffered by a separate process, and written to a file concurrently with program execution as the internal buffers fill up. Thus there is no strict limit to the number of events that can be

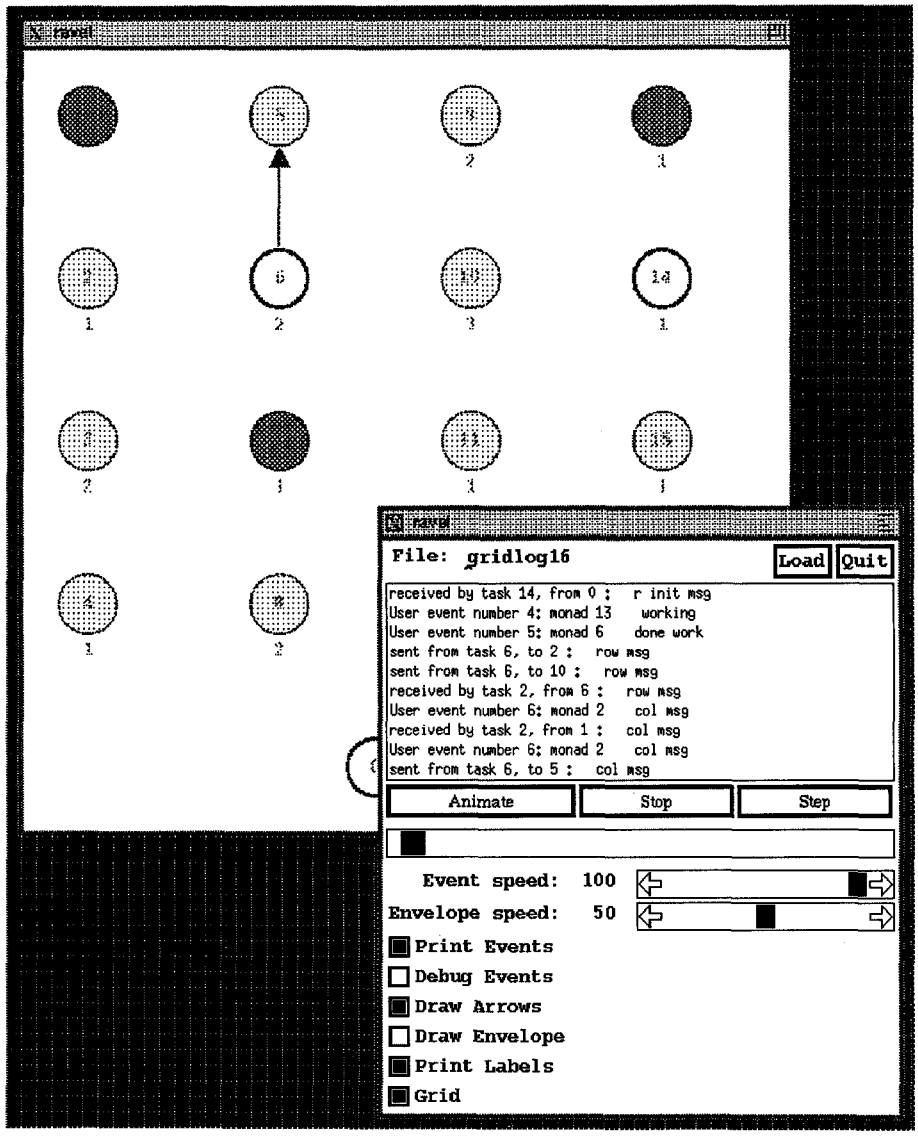

Fig. 3. Ravel

logged. The events logged are "system" events of importance to the behavior of parallel Prolog programs: the creation and elimination of parallel choice points, and the execution of parallel alternatives emanating from those choice points. (It is also possible for the user to log events from the Prolog program being executed, but graphical interpretation of them requires modifying the wamtrace program itself.) These events are displayed dynamically, with a dynamically changing tree-shaped representation of the Prolog computation, with representations of the processes exploring it moving around on the tree in an illustration

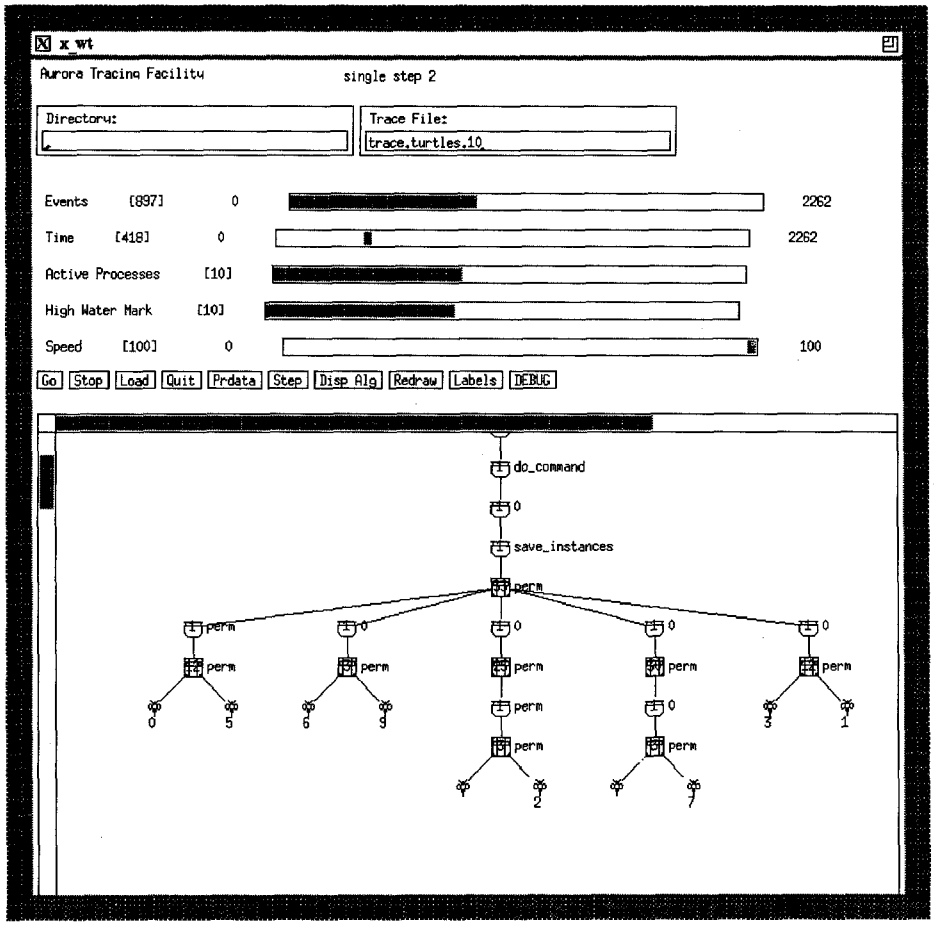

Fig. 4. Wamtrace

of the dispatching strategy. **Wamtrace** has been used both for tuning the Aurora system and for understanding the behavior of application programs.

Wamtrace was chronologically the first of the three systems described in this paper. It was developed in conjunction with the research into various dispatching strategies for Aurora.

5. Conclusions and future work

The systems described here have been useful in our study of parallel algorithms expressed in both high- and low-level languages. We believe that high-level parallel programming languages like Strand and PCN represent significant strides in the progress toward a usable parallel programming environment. At the same time, it is clear that the more abstract the programming model (and therefore the easier it is to compose a correct parallel program specification), the more difficult it will be to understand the precise sequence of events it causes on

a particular invocation and therefore the reasons for its performance characteristics. Therefore tools for understanding the behavior of programs specified in such high-level languages will be critical if these languages and the parallel computers they direct are to enter the mainstream of high-performance computing.

Currently these systems are good for examining the execution traces from runs with up to about fifty processes. We are particularly interested in extensions of these systems that will be suitable for displaying the activities of programs running on parallel machines with hundreds of nodes. We are also planning more general animation systems, in particular for data structure animation.

Upshot is available by anonymous **ftp** from **info.mcs.anl.gov**, where the directory is **pub/upshot** and the file is **upshot.tar.Z**. The companion file **alog.tar.Z** contains the logging routines that produce the logfiles to be displayed with **upshot**.

References

1. Boyle J, Butler R, Disz T, Glickfeld B, Lusk E, Overbeek R, Patterson J, Stevens R (1987) Portable programs for parallel processors. Holt, Rinehart, Winston, New York
2. Chandy M, Taylor S (1991) An introduction to parallel programming. Jones and Bartlett, New York
3. Disz T, Lusk E (1987) A graphical tool for observing the behavior of parallel logic programs. In: Proc 1987 Symp Logic Programming, p 46
4. Foster I, Taylor S (1990) Strand: New concepts in parallel programming. Prentice-Hall, Englewood Cliffs, NJ
5. Gorlick M, Kesselman C (1987) Timing Prolog programs without clocks. In: Proc 1987 Symp Logic Programming, p 426
6. Heath MT, Etheridge JA (1991) Visualizing the performance of parallel programs. Techn Rep ORNL TM-11813, Oak Ridge Natl Lab
7. Lusk E, Butler R, Disz T, Olson R, Overbeek R, Stevens R, Warren DHD, Calderwood A, Szeredi P, Haridi S, Brand P, Carlsson M, Ciepielewski A, Hausman B (1990) The aurora or-parallel prolog system. New Generation Computing 7(3):243
8. Lusk E, McCune W, Slaney J (1991) Roo – a parallel theorem prover. Techn Rep MCS-TM-149, Argonne Natl Lab