Multi-Level Monitoring of Parallel Programs

Robert G. Babb, II, David C. DiNucci, and Lise Store

Technical Report No. CS/E 87-013

November, 1987

MULTI-LEVEL MONITORING OF PARALLEL PROGRAMS

Robert G. Babb, II, David C. DiNucci, and Lise Storc

Oregon Graduate Center
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 87-013
November, 1987

MULTI-LEVEL MONITORING OF PARALLEL PROGRAMS [1]

Robert G. Babb II
David C. DiNucci
Lise Storc

Department of Computer Science and Engineering
Oregon Graduate Center
Beaverton, Oregon

Abstract— Monitoring parallel program execution can be simplified if the programs being monitored have very structured inter-process interactions. This paper describes a prototype software tool for monitoring and analyzing execution of parallel programs which were written using the Large-Grain Data Flow model. The ultimate goal of the project is to build software tools which would aid in real-time debugging and optimization by allowing relatively non-intrusive multi-level monitoring of parallel system execution and performance.

1. Introduction

Following the execution of a program can be very helpful in debugging and in analyzing performance. Tracing at the subroutine call level has proven to be a suitable level of granularity for debugging, since subroutines are typically created to perform "large-grain" tasks which are meaningful to the software designer.

Parallel programs add another dimension of complexity to the debugging and performance monitoring tasks. While sequential programs typically invoke subroutines only in a controlled hierarchical manner, the processes of a parallel program can execute independently, yielding *many* more possible combinations of executing procedures. Some of these combinations will represent program errors (e.g., when a process incorrectly expects a message or event). While the combination of currently active procedures may not tell the whole story of the correctness or incorrectness of an execution history, it can serve either as a first step of debugging or as a high level view of an execution, from which the user can descend to more detailed views of interesting processes.

[1] This work was supported under Los Alamos National Laboratory contract 9-Z34-P3915-1 Modification 2.

In debugging parallel programs, it can be important to see not only *when* a process performs a certain task, but *how* that task is related to events in other processes. In unrestricted forms of parallel processing, this causality can be very difficult to track. Debugging is hindered by the lack of structured interactions within parallel event histories. We found it useful to develop our ideas about parallel program monitoring within the restrictions of a more disciplined model of parallel processing, Large-Grain Data Flow (LGDF), which is described briefly in the next section. The primary benefit of this approach lies in greatly reducing the number of "interesting" parallel interactions to be monitored. Although our work was specifically aimed at monitoring parallel programs written in LGDF form, similar issues would have to be dealt with in any (less restricted) model of parallel processing.

2. Overview of LGDF

Large-Grain Data Flow is a model of parallel computation that combines sequential programming with dataflow-like program activation. LGDF applications are constructed using networks of program modules connected by datapaths. Parallel execution is controlled indirectly via the production and consumption of data. With the aid of the LGDF Toolset, LGDF programs are automatically transformed for efficient implementation on a particular parallel machine.

2.1. LGDF Networks and Datapaths

A Large-Grain Data Flow *network* consists of a set of *nodes* (represented graphically by circles or "bubbles") connected by *datapaths* (represented graphically by directed arcs). Nodes are tagged for referencing purposes with "p#'s", datapaths are tagged with "d#'s". Datapaths can also be *dot-shared* so that one datapath can be an input to or an output from more than one node. See figure 1.

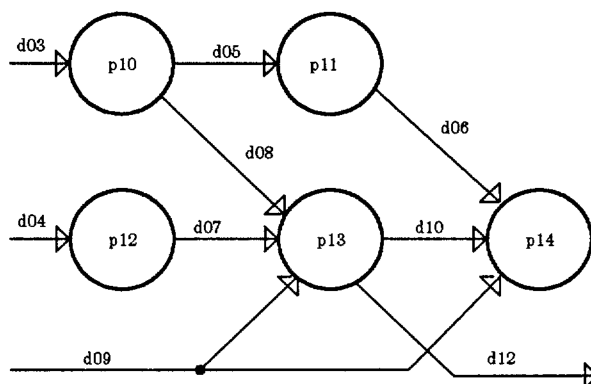

Figure 1. Example of LGDF graph

A node can represent either an LGDF program or an LGDF network, in which case it is called a subnetwork node. The meaning of a network containing a subnetwork node is

the same as if the lower level network were substituted graphically for the higher level node.

A datapath is the only mechanism for communicating data values from one LGDF process to another. It consists of a data area and a state which can be either *empty* or *full.* Read/write access to datapaths is controlled by the empty/full state. The empty/full state is changed explicitly by the programs that the datapath is connected to, and signifies (to other processes connected to the datapath) whether there is valid data to be read on an input datapath (full) and whether an output datapath is ready to accept new data (empty).

2.2. LGDF Process Activation

An LGDF program's eligibility for activation depends only upon the state of its associated input and output datapaths, according to following rule:

> **Execution Rule**—A program may start an execution cycle if and only if all of its input datapaths are in the *full* state and all of its output datapaths are in the *empty* state.

Once a process wakes up, it may read the data associated with any of its input datapaths and modify the data associated with any of its output datapaths. When a program has finished accessing an input datapath, it can change the state of the path to empty by using the LGDF *clear_* command. (In LGDF programs, the command refers to a specific datapath by its unique tag, e.g. *clear_(d06)*). The usual reason an LGDF program clears an input datapath is to allow an upstream process to wake up and write the next set of data values onto the datapath. Likewise, when a program has finished writing values onto an output datapath, it can change the state of the path to full by using the LGDF *set_* command. An LGDF process, then, is limited in the amount of work it can do in a single execution cycle. When it is finished with that work, it puts itself to sleep using the *suspend_* command, and it will remain suspended until after its datapaths again satisfy the LGDF Execution Rule. At the time a process suspends, it checked for conformance with the following rule:

> **Data Flow Progress Rule**—Upon suspension of an execution cycle, a program must have cleared at least one input or set at least one output datapath. Otherwise, it is terminated.

A process that is in the *terminated* state can never re-awaken, regardless of the empty/full state of its datapaths.

2.3. Implementation of LGDF

LGDF programs are implemented with aid of set of macro-based tools. The connectivity of a network is defined in a *wirelist* file. The clear, set, and suspend commands are implemented as macro calls in the program source text associated with LGDF programs (nodes).

For more information on the Large-Grain Data Flow model, see [1].

3. Design for the Prototype Monitoring Tool

The purpose of this research project was to design a graphics monitor to aid in the debugging and analysis of Large-Grain Data Flow (LGDF) programs. Our goal was to prototype a parallel program animation system that would give LGDF programmers "intuition" about how an LGDF computation was progressing. As a secondary goal, we wanted the tool to aid in controlled, very high-level debugging of large parallel application codes.

Our approach was to enhance the LGDF macros so that they would optionally generate sufficient trace data to monitor the interactions and initiations of the processes running in a parallel environment. This trace data was designed to be used by a LGDF network monitor program (running on an IBM PC) to display parallel process initiations and interesting data flow events in a graphic form.

An LGDF application is represented graphically on the monitor screen by its associated Large Grain Data Flow graph. Process bubbles change color with execution state (running, sleeping, and terminated) while datapaths change color to reflect the state of the associated empty/full flag. In this way, all process interactions and initiations are clearly illustrated, and all possible future interactions are apparent from the structure of the graph.

The relative time between events is also important information which would normally be lost unless the monitor could keep up with the real-time production of the trace stream. To allow the monitor to run at any speed without losing this information, we included a timestamp on each trace record and a "speedometer" on the monitor screen to show the speed of the trace display relative to the speed the program would have been running without monitoring. A user can either set the speedometer to some fixed value so that the relative speed of display is made to accurately reflect that of an actual execution, or can let the speedometer float to allow screen updates to occur as quickly as possible.

The "wirelist" file contains all the semantic information about the interconnection structure of an LGDF network for a given application, but does not contain how such a network should be presented in graphic form. While the monitor could deduce this from the information already in the wirelist, we decided to add the network presentation format to the wirelist, expecting that the user would someday deal with the wirelist only in its graphic form by using a special wirelist editor. In this arrangement, it seemed most proper to keep all information on the placement of nodes and datapaths explicitly to give the user maximum control. This does not preclude the possibility that the user could request the wirelist editor to automatically place graph elements.

Our design for the monitoring system was intended to be as interactive as possible. For example, the user can dynamically expand any subnetwork bubble into its components or can collapse a subnetwork back into a single subnetwork bubble while the trace is

progressing, thereby enabling hierarchical views of the display.

3.1. Display Actions

The display of the network on the graphics screen can be affected by trace records read from the trace stream or by user input from the keyboard or mouse.

3.1.1. Trace Actions

Each record within the trace stream contains a record of one LGDF command execution (clear, set, or suspend) or the beginning or end of a process execution cycle. In each case, the tag of the entity (node or datapath) being described is included, as well as the execution time and wall clock time from its associated processor.

When the monitor reads a trace record, the display will be affected only if the entity described by the trace record is present in the currently displayed view of the network. Even then, if the speedometer has been set to a fixed value, screen updates may be delayed, or collected and applied in batches, to ensure that the display speed matches the speedometer. Whether or not the displayed view of the network is updated, the color and state of all entities will be updated internally with each trace record.

3.1.2. User Actions

User interactions from the keyboard or mouse can be used to change the view of the network currently on the display or to control the speed at which the display is updated. Whenever the network view is changed, the monitor's internal representation of the network is used to accurately represent the state of previously unseen parts of the network.

The user can perform the following trace actions:

- Expand or Collapse a Subnetwork
  Provides a more detailed view of some specific part of network.
- Restrict or Unrestrict the Display
  To narrow view to some subset of network. In addition to focussing attention, this also speeds up monitor by alleviating uninteresting screen updates.
- Stop trace temporarily or Continue
  Allows time to analyze some particular state of the display.
- Perform single visible trace step
- Control trace speed
- Abort trace
- Set/Clear Breakpoint on a process or datapath
  Stops trace whenever some interesting dataflow event occurs.
- Clear all Breakpoints

3.2. Performance metrics and meters

We included some experimental performance metrics in the form of meters to expand the functionality of the monitor to include performance analysis. These would aid in detecting bottlenecks caused by fast processes waiting for slow ones, and could also help the user decide how processors should be allocated to processes (load balancing).

We have already mentioned the speedometer, which is used to measure the speed of display relative to speed of original program execution. Rather than attempting to determine an instantaneous speed, we felt that it would be more useful if it had more stability, so we measured the speed over the last δ trace records, giving an 'average relative speed' over a short or long period depending on the value of δ. The formula then is

$$\frac{\Delta t_{timestamp}}{\Delta t_{display}}$$

where $\Delta t_{timestamp}$ is the difference in the timestamps of the last record and the record δ records ago, while $\Delta t_{display}$ is the difference in the wall clock time when the affects of those records were displayed on the monitor. The choice of δ is left to be specified at run-time by the user or decided by experimentation. Note that this measure loses validity when the trace is stopped temporarily, so it is recalculated each time the trace is restarted.

The amount of parallelism achieved within the a "process set" could be represented by the equation

$$\frac{\sum_{p \epsilon P} E_p}{\Delta T - I}$$

where P represents the process set, E_p represents the amount of time process p was executing, I is the amount of time that all processes in P were idle, and ΔT is the length of the interval all of the other factors were measured over. Although our initial design defines the process set to be all of the processes currently on the monitor screen, it would be desirable to let the user pick the processes to be included.

If all processes in the process set are on separate processors, this is a measure of speedup, since it is ratio of the time that the process would have taken on one processor to the time it took on the many processors. If the processes in the process set are all on a single processor, this is a measure of contention among processes for CPU time. To get a speedup measure when $\frac{processes}{processors} > 1$, perhaps a similar metric could be devised using CPU time measures.

3.3. Host/Monitor Interaction

In our initial design, the monitor was intended to run while the host program was running, requiring bidirectional data transfer between the monitor and host systems. Parallel host to monitor data would include the trace stream and any normal program output, while monitor to parallel host data would consist of flow control and interactive program input.

This design was abandoned for the prototype because of complications arising from the need for flow control. In order for the speedometer to remain accurate, the actual speed of execution of the host program could not be affected by the speed of the display, meaning that a large output buffer had to be maintained for the trace between the host program and the display system. Although there are probably ways to handle this on some operating systems (e.g. in Unix, by having the trace go to a file on the host system and then having a separate process such as `"tail"` spool the file to the monitor), this is very system dependent and offers little advantage over simply creating a trace file which is moved over to the monitor after the trace is complete.

4. Experience with Prototype

Our prototype monitor was implemented on an IBM PC/XT with an Enhanced Graphics Adaptor (EGA). This configuration was chosen because it was already available at our customer site, Los Alamos National Laboratory. Our graphics library was a version of the Graphics Kernel System (GKS), which we chose because it offered the hope of easy portability of our tools to other implementation environments.

The resulting demonstration software was relatively slow. This was primarily because we used a much higher-level graphics package than was required, and were thereby paying the cost of slower screen updates to buy functionality that we were not using. The monitor could be regarded as performing real-time animation, which GKS does not seem to be designed for. Some of the lack of speed could also be attributed to the hardware we were using, which had a relatively slow cycle time and no floating point hardware.

A more useful configuration for implementation of a system such as this would be to have a more powerful system (perhaps the parallel host) performing the computation with a graphics terminal performing the monitoring. A smart terminal would best preserve the bandwidth between the terminal and main system thereby maximizing display speed. If a personal computer/workstation configuration is desired, lower-level graphics should be considered, and perhaps a computer with more of a graphics orientation, such as a Commodore Amiga, Apple Macintosh, or Tektronix or Sun graphics workstation.

5. Future Directions - A Multi-level Debugger/Testbed

Although resources for the prototype project ran out before we were able to implement some portions of the design, this project allowed us to explore several in the realm of real time monitoring of parallel programs. Perhaps the most intriguing of these areas became

apparent while considering the kinds of interaction the user could have with the program while it was running.

In most parallel programming environments, the relative timing of the separate processes can affect the results of the program, yielding the nondeterministic behavior which these programs have become infamous for. Adding tracing to such programs can affect these relative timings, thereby affecting the results of the program. To minimize this problem, it is important that tracing be as unobtrusive as possible.

An interesting technique that might be used with such a program would be to trace only enough of the execution so that it could be executed with the same relative timings at a later time. This reconstructed execution could then be monitored in any more detailed way desired, as long as the important relative timings in the reconstruction occur in the same way they did during the actual execution. Although the trace itself would contain very little information, the trace together with the original program and original input data could yield a wealth of information.

It may not even be necessary for the reconstruction to take place on the same hardware that originally ran the program. A program that was originally run on a parallel processor could be recreated under a multiprocessing operating system such as Unix, as long as the compiler and machine arithmetic were functionally identical. The tool performing the reconstruction could, in fact, allow the user to call a standard interactive debugger (such as dbx on Unix) to allow the user to set breakpoints and/or monitor values during the execution.

The LGDF programming model provides support for such a system. Since the execution of a typical LGDF application will have very few opportunities to exhibit non-deterministic behavior, and only those instances would need to be recorded in the trace stream, the impact of tracing on an actual execution would be much smaller than in our prototype. During reconstruction of the execution, the user could watch using a high level monitor, such as the one we have outlined here, until the program enters a particularly important phase, at which time the user could query the data available on data paths directly from this high level or could opt to invoke a standard interactive debugger.

In fact, an LGDF program is so stable with respect to relative timing that it may be desirable in some cases to skip the tracing phase and simply run the monitored program in a testbed environment, where the user could dictate when processes should be held from executing. In addition, the user could manually deposit data on a datapath or modify a variable within a program. The tool controlling these interactions could warn the user when the program could act in a non-deterministic way (i.e., when there is a choice of processes that can start, all of which access a common data path).

6. Conclusion

The monitor described in this paper has capitalized on the fact that the programs monitored are restricted in the ways that their processes can interact. To design monitors for

programs written with less restrictive models of parallelism, it may be useful to start here and carefully analyze the complexities added as successive levels of restrictions are lifted.

For more information on this project, see [2].

7. References

[1] Robert G. Babb II, David C. DiNucci, "Design and Implementation of Parallel Programs with Large-Grain Data Flow," *Characteristics of Parallel Algorithms,* MIT Press, 1987.

[2] Robert G. Babb II, David C. DiNucci, Lise Storc, "Program Monitoring Tools for Parallel Processing with Large-Grain Data Flow Techniques," *Final Report for Los Alamos National Laboratory Computer Research and Applications Division under Contract 9-Z34-P3915 Modification No. 2,* Department of Computer Science and Engineering, Oregon Graduate Center, Beaverton, Oregon, 8 October 1986.