Enhancement of Environments for Analysis of Trace files of Parallel Programs

Veena Avula*
Department of Computer Science
Cornell University
Ithaca NY 14850
veena@cs.cornell.edu

December 23, 1994

Abstract

One of the important phase of parallel programming is performance analysis. Trace data provides information about where time is spent in programs. Since this data is huge, a tool for analyzing and visualizing the trace data is convenient and necessary for performance analysis of parallel programs. Environments which provide such a facility are many and varied. In this report, we discuss our work on the enhancement of one such environment for accessibility over more platforms and better visualization capabilities. The environment is Pablo.

CTC95TR204

01/01/95

*This work was part of a Masters of Engineering Project.

1 Introduction

Two important phases of parallel programming are debugging and performance analysis. The second phase - improving performance - requires detecting the bottlenecks and trying to eliminate these. Trace data gives us information about where the time is spent by the program. Since this data is huge, a tool for analyzing and visualizing the trace data is convenient and necessary for performance analysis of parallel programs. This report describes our work in enhancing such environments. This work is particularly in two specific contexts.

Pablo is a tool used to study the performance of a system in order to make algorithmic enhancements. First, we wrote a utility to convert tracefiles of the KSR-1 machine into the SDDF [1] format used by Pablo. The second activity was adding a module to the Pablo visualization software. In the process, we set up a software base that we believe, lends itself to easier software development in the future by providing a systematic setup. Also, we acquired a great deal of intimacy with the internals of Pablo in this process, which we believe will be helpful in guiding future development.

We describe the installation of Pablo in Section 2. Section 3 describes how to use Pablo. Section 4 is on trace files and Section 5 discusses the design and impelmentation of the *ksr2sddf* utility. Section 6 discusses the *CtrlInput* module and we conclude with Section 7.

2 Installation of Pablo

We installed Pablo on a SUN-SPARC snake.tc.cornell.edu, a SPARC 2 in the Cornell Theory Center. This procedure involved two main steps - installing GNU software and installing Pablo itself. In this section, the details of the above two steps are discussed.

2.1 GNU software - A good environment for installing Pablo

To install Pablo, we used GNU software. This is free software provided by Free Software Foundation and is known for its quality and portability. GNU has sofware for most of the commonly used Unix commands, compilers, linkers and editors. It can be installed on different unix platforms and operating systems fairly easily. Installing Pablo using GNU software gives us the advantage of utilizing GNU software's portability across different unix platforms and different operating systems to install Pablo on different unix platforms and different operating systems quite easily. We installed all this on a SUN-SPARC snake.tc.cornell.edu, a SPARC 2 in the Cornell Theory Center.

The organization for gnu software on snake is as follows. The source files are in the directory /usr/local/gnu/src. This directory has sub-directories for all the gnu utilities installed on the system. These sub-directories are unpacked and uncompressed versions of the utility available in GNU anonymous FTP sites.

The /usr/local/gnu directory is where the packages are installed. There is a directory for each utility. Typically these directories contain sub-directories containing binary executables, library files, info documentation and man pages.

The /usr/local/gnu/bin directory contains all the executable files for the gnu utilities. These are actually symbolic links to files in the bin subdirectories of the installed utility directories (which are sub-directories of /usr/local/gnu).

The rationale behind maintaining the source files and binary files in separate areas for each of the utilities is that different machines (of different architectures) having a common file system can share the source files and maintain separate copies of only the binary, library files etc.

Some of the GNU mirror sites (where the gnu software is available by anonymous ftp) are given in Appendix A.

We used shell-scripts for commands which are frequently used while installing gnu software. On snake, these shell-scripts are maintained in the bin directory of the user gnu. A shell-script used to get a gnu utility from one of the GNU mirror sites is ftpget. This and other shell scripts are in gnu/bin directory on snake.

All compressed files in the GNU mirror sites are in the .gz compressed form. Gzip is the GNU compression utility and all GNU utilities are compressed in this form, because the unix compress utility is patented. So,

the first gnu utility to be installed on the system was gzip. Gzip itself is available in uncompressed/shar/.Z format in all the GNU mirror ftp sites.

To install Pablo, we needed *gcc* and *libg++*. To install these, we had to install *gzip*, *tar*, *fileutils* and *make* utilities.

For each installed utility, we wrote a README.CORNELL file which contained all the steps to be done for intstalling the utility after getting the sources for that utility. For example, such a Readme file for gcc would look like:

```
gcc 2.5.8 Build and installation procedure:
o setenv GCC $GNU/gcc-2.5.8; \
  setenv LIBGPP $GNU/libg++-2.5.3; \
  setenv GCT $GNU/gct-1.3; \
  cd $SRC/gcc-2.5.8
o For SunOS...configure --target=sparc-sun-sunos4.1 --prefix=$GCC
o logmake stage1 LANGUAGES=c &
o make stage1
o logmake stage2 CC="stage1/xgcc -Bstage1/" CFLAGS="-g -O" &
o mkdir -p $GCC/info; \
  logmake install CC="stage1/xgcc -Bstage1/" CFLAGS="-g -O" install &
o if libg++-2.5.3 is installed...
  + rm -f $GCC/lib/g++-include; ln -s $LIBGPP/lib/g++-include $GCC/lib; \
    ln -sf $LIBGPP/lib/libg++.a $GCC/lib/gcc-lib/*/*; \
    foreach x ($LIBGPP/*/include/*)
       ln -sf $x $GCC/*/include
    end
o if gct is installed...
  rm $GCT/lib/gct-include/gcc; \
  ln -sf $GCC/lib/gcc-lib/sparc-sun-sunos4.1/2.5.8/include $GCT/lib/gct-include/gcc
o rm -f $GCC/bin/*-gcc; \
  ln -sf gcc $GCC/bin/gcc-2; \
  ln -sf gcc.1 $GCC/man/man1/gcc-2.1
o finish-up gcc-2.5.8
o rehash; \
  make distclean
```

Note that several of the commands in the README.CORNELL file are calls to the gnu shell scripts which are in gnu/bin directory.

2.2 Installing Pablo

Pablo can be obtained by anonymous ftp from the machine bugle.cs.uiuc.edu in the directory /pub/Release/Pablo. After the file has been obtained, it has to be uncompressed and untarred. (Even though Pablo can be obtained at no cost, it is NOT in the public domain).

To build and install Pablo we did the following.

Bulid procedure:

- To have gcc as the defualt C compiler:
  Change the CC flag in INSTR_FLAGS to gcc for the selected INSTR_TARGET.
- To have g++ as the default C++ compiler:
  Select the COMPILER to be GNU. Change the pathnames of g++ and cpp to point to g++ and cpp in /usr/local/gnu/gcc-2.5.8 directory in GNU g++/gcc version 2.5.8 on Sparcstation.
- Use g++ and gcc with -I/usr/include option.
  The reason was the older versions of include files in /usr/local/X11R5/include directory.
- Comment out the line
  XPM_LIB := /usr/local/X11R5/lib/libXpm.a in XPM_LIBRARY, since libXpm.a was not available on snake.tc.cornell.edu.

- make all

Installation procedure:

- Change PABLOHOME to /nfs/PABLO/pablo3.0 to install Pablo in /nfs/PABLO/pablo3.0.
- make install

The Makefile.defines file with all these changes is listed in /nfs/PABLO/src/pablo3.0 directory. Building and installing is quite straight forward after this. This process takes approximately two hours.

3 Using Pablo

This section discusses some of the better ways of getting to know Pablo. Details of configuring the Pablo Visualization environment are also given here.

3.1 Manuals, Tutorials, Examples

The best way of getting familiar with Pablo is by using it. The current release of the software (Release 3.0) does not include a comprehensive user's manual. However, this release includes a comprehensive set of tutorials which introduce the features of Pablo to the user. There are six tutorials installed on /nfs/PABLO/pablo3.0/config/TUTORIALS on snake. *An Informal Guide to Using Pablo* [3] contains a detailed discussion of each of these tutorials. Also, the *Pablo Performance Analysis Environment* [4] provides an overview of the Pablo system.

The first four tutorials are aimed at providing an introduction to the basic operations which must be performed whenever Pablo is used. These turorials work with small files which do not compare to the performance data trace files which usually contain many types of records and potentially thousands of records of each type. The simplicity of the files chosen for these tutorials is justified, because they make it easier to see how input record types affect the choices when configuring the Pablo visualization environment. The final two tutorials do not cover the details of using Pablo; instead they show how the basic building blocks provided by the Pablo visualization system can be manipulated and present information from actual performance trace files to gain insight into program behaviour. These tutorials take as input a performance trace data file that was captured using Pablo instrumentation system. This trace data was gathered on an eight node iPSC/860 hypercube executing a parallel PDE solver.

3.2 Configuration

Pablo data analysis environment supports the graphical interconnection of performance data transformation *modules*, to form a directed acyclic data analysis graph (Figure 1). The semantics of data analysis are embedded in this graphical *configuration*.

One of the key concepts of Pablo's trace visualization system is that one can configure one's own view of the data. The user specifies the desired data transformations and presentations by building an appropriate graph of functional units (the data transformation and visualization modules) and interactively specifying which kinds of data records should be processed by each of the functional units. Pablo uses the record and field names from the record descriptors in the input SDDF trace file to provide an interactive session by which the user can configure the functional units.

3.3 Internals

The Pablo data analysis environment is written in C++ and uses the X window system and Motif toolkit as its user interface.

A Pablo data analysis graph consists of two primary components: a collection of nodes and their connections. These nodes are referred to as *functional units* and the connections as *pipes*. There are four primary module types:

- File input.
- File output.
- Data analysis.
- Data presentation.

File input and *File output* modules lie at the root(s) and leaves of the graph respectively. These modules read and write data in the Pablo self-defining format. *Data analysis* modules lie in the interior of the graph and transform the performance data flowing through the graph. *Data performance* modules usually form the leaves of the data analysis graph and are used to display data produced by internal graph nodes. Note that there are two different types of data presentation modules - graphical and sonic.

4 Trace files

Trace files are used to record the execution path taken by a program. Typically, trace files are used to find out why a program spends its time in a certain region. This information is required to analyze bottlenecks in the program for better parallelism. One of the disadvantages associated with a trace file is its huge size, especially in parallel programs where several executions are merged together. Typical trace files easily run into megabytes and sometimes they run into tens or even hundreds of megabytes. Another disadvantage of trace files is that different systems trace different information and store this information in different formats. Thus, to compare the performance on two different systems requires a great deal of intimate knowledge of the trace file representation format of the different systems. Pablo overcomes this problem to attain its goal of portability. It uses a trace file format that is self explanatory. It is called SDDF and stands for Self Defining Data Format. We describe this in the next section with an example.

4.1 Pablo Trace files

The principal idea behind the SDDF format is that the trace file uses a few standard types and larger types built up from these. Also, the trace file contains header information that describes the various "types" of records in the trace file. Thus the header information is sufficient to "parse" the trace records that occur in the trace file. SDDF - trace description language specifies both the structure of data records and data record instances. This approach also lends itself to portability, since the only default types assumed are standard types such as strings, integers etc, which are universal. By storing these types in some universal code (such as ascii), it is easy to port a tracefile from one system to another. For reasons of compressed storage it is possible to store the SDDF trace files in binary format also (this sacrifices portability for storage requirements). Here is an example of a file in SDDF format (this is from one of the tutorials of Pablo):

```
SDDFA
#1:
"Concessions" {
        int     "Inning";
        int     "Hot Dogs Sold";
};;

#2:
"Player Record" {
        int     "Team";
        int     "Player Number";
        int     "Hits";
        int     "At-Bats";
};;

"Concessions" { 1, 100 };;
"Concessions" { 2, 300 };;
"Concessions" { 3, 200 };;
"Concessions" { 4, 500 };;
"Concessions" { 5, 909 };;
```

```
"Concessions" { 6, 1200 };;
"Concessions" { 7, 2300 };;
"Concessions" { 9, 300 };;
"Concessions" { 8, 100 };;

"Player Record" { 1, 4, 4, 5 };;
"Player Record" { 1, 3, 2, 5 };;
"Player Record" { 1, 7, 0, 4 };;
"Player Record" { 1, 5, 4, 5 };;
```

The ascii and binary versions of the SDDF trace files describe four classes of records.

- Stream attributes - information about entire trace file. In the above example, this is SDDFA, which indicates that the trace file is an ascii trace file in SDDF format.In general, this could include execution date, machine on which the trace was generated etc.
- Record descriptors - These are templates for data records. A useful analogy is with the typedef statement in a C file. Again with reference to the above example, the concession and the player record descriptions which occur right at the beginning of the file are record descriptors.
- Data - These are the actual trace records in the format of the templates. These are the remaining entries in the above example.
- Command - These indicate action to be taken. None are present in the above example. These are used by PABLO data analysis to control the environment's execution, and rarely appear in performance trace files.

4.2 KSR Trace files

The KSR is a distributed shared memory multiprocessor that provides shared memory at the user level. It provides user level parallelism through the pthreads package. These threads also provide the basis on which the KSR generates trace data.

On the KSR, trace data is captured in 'events' and 'states'. An event is defined by a name and a code. The events are logged in the Event log file by the time and the pthread in which they occured. A 'state' is defined by two events which mark the beginning and end of that state. Elog (event log) library routines are used in the parallel program to create these event log files. These event log files include KSR's pre-defined events and states. Event log files are generated when programs instrumented with elog routines are run. Some of the typical elog library routines are:

- elog_init - This must be called before any other event logging functions. It initializes the event log file and returns a non-zero value on success.
- elog_log - This logs an event of a particular type with the data
- elog_setup - This routine performs per thread initialization which is necessary for event logging.
- elog_output - This writes the events logged by the process to the event log file.

[1] The gist program can be used to display the event log files. Gist supports the following two formats for the event log files:

- Gisttext format - Each event is output in the following format : E event trace_id time delta-time event data where delta-time is the time since the previous event.
- Brief format - Each event is output in the following format: trace-id time event data

[1] Programs compiled and linked with the Presto runtime library automatically generate event log files when run with the environment variable PL_ELOG set to 1

While the Gisttext format is more readable, the brief format has the advantage of being smaller in size and more easily processed by other programs. For example, the same event log file in the gisttext format and in the brief format compares as follows.

Gisttext format:

```
E 0x0                     0        0  11  Begin serial -1
E 0x0                  9762     9762   1  BEGIN program 0
E 0x0                360319   350557  12  End serial 1
E 0x0                360610      291  41  Begin Parallel Region 2
E 0x0                361016      406  42  End Parallel Region 2
E 0x3                416640    55624  41  Begin Parallel Region 1
E 0x3                417360      720  42  End Parallel Region 1
E 0x1                462212    44852  41  Begin Parallel Region 1
E 0x1                462562      351  42  End Parallel Region 1
E 0x2                472599    10036  41  Begin Parallel Region 1
E 0x2                473902     1303  42  End Parallel Region 1
E 0x0                474174      272  11  Begin serial 1
E 0x0                665469   191295  12  End serial 1
E 0x0                665586      117  31  Begin Parallel Sections 1
E 0x4                666441      855  31  Begin Parallel Sections 1
E 0x4               2273876  1607435  32  End Parallel Section 1
E 0x0               2724154   450278  32  End Parallel Section 1
E 0x2               2752055        8  21  Begin Parallel Tiling 1
E 0x2               2752214      159  23  new tile 1
E 0x3               3110626   358412  21  Begin Parallel Tiling 1
E 0x3               3110766      139  23  new tile 1
E 0x2             170008404 166897638  22  End Parallel Tiling 1
E 0x1             181531311 11522907  22  End Parallel Tiling 1
E 0x0             181853265   321954  22  End Parallel Tiling 1
E 0x3             184039879  2186614  22  End Parallel Tiling 1
E 0x0             186393498       11  99  END program 1
```

Brief format:

```
0 0 11 -1
0 9762 1 0
0 360319 12 1
0 360610 41 2
0 361016 42 2
3 416640 41 1
3 417360 42 1
1 462212 41 1
1 462562 42 1
2 472599 41 1
2 473902 42 1
0 474174 11 1
0 665469 12 1
0 665586 31 1
4 666441 31 1
4 2273876 32 1
0 2724154 32 1
2 2752055 21 1
2 2752214 23 1
3 3110626 21 1
3 3110766 23 1
2 170008404 22 1
1 181531311 22 1
0 181853265 22 1
3 184039879 22 1
0 186393498 99 1
```

5 Converting KSR Tracefiles to SDDF format

As has been pointed earlier, Pablo's SDDF format allows us make Pablo an universal environment for analyzing parallel programs of different machines. The **ksr2sddf** utility adds to this feature of Pablo. This section disscusses the design and implementation details of this utility.

5.1 ksr2sddf

ksr2sddf is a utility to convert KSR tracefiles to SDDF format. Facility for transformation to both ascii and binary versions of SDDF format is provided via command line options to this utility.

As has already been pointed out before, SDDF format is portable and enables the use of the *Pablo* Visualization package. Converting *ksr* tracefiles to *sddf* format enables us to use Pablo for performance analysis of the KSR programs.

This utility makes use of the SDDF library[2]. The SDDF library provides us with classes and methods which can be used to perform input and output to files in SDDF format (both binary and ascii). A brief description of some of the routines we used in writing ksr2sddf is given below. [2]

- OutputFileStreamPipe: This class is used to open files for output. It provides an interface to standard UNIX files that is compatible with the SDDF PipeWriter class.
- AsciiPipeWriter: This class is used to write SDDF data packets in ascii form to the OutputFileStreamPipe.
- BinaryPipeWriter: This class is used to write SDDF data packets in binary form to the Output-FileStreamPipe.
- StructureDescriptor: This class is used to build up data record structures, providing an internal representation for the information conveyed by an SDDF Record Descriptor packet.
- FieldDescriptor: This class is used to define a field by specifying its name, attributes, type and dimensonality (in the case of structured objects such as arrays).
- putDescriptor: This is a method of the AsciiPipeWriter and BinaryPipeWriter classes. This writes a Record Descriptor packet to the OutputFileStreamPipe.
- putData: This is a method of the AsciiPipeWriter and BinaryPipeWriter classes. This writes a Record Data packet to the OutputFileStreamPipe.
- putAttributes: This is a method of the AsciiPipeWriter and BinaryPipeWriter classes. This writes a Stream Attribute packet to the OutputFileStreamPipe.
- putCommand: This is a method of the AsciiPipeWriter and BinaryPipeWriter classes. This writes a Command packet to the OutputFileStreamPipe.
- RecordDossier: This class provides a mechanism for caching the structure information and managing data values for the fields in record.

ksr2sddf has been installed on snake in directory /nfs/PABLO/bin/ksr2sddf1.0. The source for this utility is available in /nfs/PABLO/src/ksr2sddf1.0.

5.2 Calling conventions for ksr2sddf

ksr2sddf takes four command line arguments. The first argument specifies whether the input ksr trace file is in the gist format(*g*) or the binary format (*b*). The second argument is the input file. This file is the ksr trace file which needs to be converted to the sddf format. The third argument is the output file - file in sddf format created corresponding to the input ksr tracefile. The fourth argument allows the user to specify the option of creating the sddf file either in binary format (specified by a *b*) or in ascii format (specified by an *a*).

[2] To link with SDDF library link with -lPablo

5.3 Examples

An example of using ksr2sddf:

Let ksr.file (ksr tracefile) be the file used earlier in section 3.2 while discussing ksr tracefiles. (Note that this is in the gist format).

After running the ksr2sddf utility, by typing
ksr2sddf g ksr.file sddfa.file a,
file *sddfa.file* will contain the following:

```
SDDFA
#11:
"Begin serial" -
        int     "thread id";
        int     "time stamp";
        int     "delta time";
        int     "event id";
        int     "data";
~;;

#1:
"BEGIN program" -
        int     "thread id";
        int     "time stamp";
        int     "delta time";
        int     "event id";
        int     "data";
~;;

#12:
"End serial" -
        int     "thread id";
        int     "time stamp";
        int     "delta time";
        int     "event id";
        int     "data";
~;;

#41:
"Begin Parallel Region" -
        int     "thread id";
        int     "time stamp";
        int     "delta time";
        int     "event id";
        int     "data";
~;;

#42:
"End Parallel Region" -
        int     "thread id";
        int     "time stamp";
        int     "delta time";
        int     "event id";
        int     "data";
~;;

#31:
"Begin Parallel Sections" -
        int     "thread id";
        int     "time stamp";
        int     "delta time";
        int     "event id";
        int     "data";
~;;

#32:
"End Parallel Section" -
        int     "thread id";
        int     "time stamp";
        int     "delta time";
        int     "event id";
        int     "data";
~;;

#21:
"Begin Parallel Tiling" -
        int     "thread id";
        int     "time stamp";
        int     "delta time";
        int     "event id";
        int     "data";
~;;

#23:
"new tile" -
        int     "thread id";
        int     "time stamp";
        int     "delta time";
        int     "event id";
        int     "data";
~;;

#22:
"End Parallel Tiling" -
        int     "thread id";
        int     "time stamp";
        int     "delta time";
        int     "event id";
        int     "data";
~;;
```

```
#99:
"END program" -
      int     "thread id";
      int     "time stamp";
      int     "delta time";
      int     "event id";
      int     "data";
~;;

"Begin serial" - 0, 0, 0, 11, 0 ~;;

"BEGIN program" - 0, 9762, 9762, 1, 0 ~;;

"End serial" - 0, 360319, 350557, 12, 0 ~;;

"Begin Parallel Region" - 0, 360610, 291, 41, 0 ~;;

"End Parallel Region" - 0, 361016, 406, 42, 0 ~;;

"Begin Parallel Region" - 3, 416640, 55624, 41, 0 ~;;

"End Parallel Region" - 3, 417360, 720, 42, 0 ~;;

"Begin Parallel Region" - 1, 462212, 44852, 41, 0 ~;;

"End Parallel Region" - 1, 462562, 351, 42, 0 ~;;

"Begin Parallel Region" - 2, 472599, 10036, 41, 0 ~;;

"End Parallel Region" - 2, 473902, 1303, 42, 0 ~;;

"Begin serial" - 0, 474174, 272, 11, 0 ~;;

"End serial" - 0, 665469, 191295, 12, 0 ~;;

"Begin Parallel Sections" - 0, 665586, 117, 31, 0 ~;;

"Begin Parallel Sections" - 4, 666441, 855, 31, 0 ~;;

"End Parallel Section" - 4, 2273876, 1607435, 32, 0 ~;;

"End Parallel Section" - 0, 2724154, 450278, 32, 0 ~;;

"Begin Parallel Tiling" - 2, 2752055, 8, 21, 0 ~;;

"new tile" - 2, 2752214, 159, 23, 0 ~;;

"Begin Parallel Tiling" - 3, 3110626, 358412, 21, 0 ~;;

"new tile" - 3, 3110766, 139, 23, 0 ~;;

"End Parallel Tiling" - 2, 170008404, 166897638, 22, 0 ~;;

"End Parallel Tiling" - 1, 181531311, 11522907, 22, 0 ~;;

"End Parallel Tiling" - 0, 181853265, 321954, 22, 0 ~;;

"End Parallel Tiling" - 3, 184039879, 2186614, 22, 0 ~;;

"END program" - 0, 186393498, 11, 99, 0 ~;;
```

6 CtrlInput - A Widget to Examine only Part of a Trace

Very often the trace data of a parallel program is really huge and the user is interested in analyzing only a particular portion of this data. This is because detailed analysis of parallel programs is done step by step and hence only parts of the program of particular interest are vey often analyzed. Pablo's *File Input* does not provide such a facility. Adding such a feature for selecting particular parts of the trace file and analyzing only these selected parts (i.e) *controlling input*, provides the user with the ability to "browse" trough a trace file, selecting interesting portions at a time for special attention. This is helpful for instance when, in a large program, most of the interesting events happen over a small portion of the execution time, and it is necessary for the user to go over this portion in detail.

6.1 Functional Description

The functional unit that we added had to have the following requirements. It would enable a user to do ordinary types of fileinput if he wanted to. However, the module would also provide the user with "controls" that the user would use to select only a portion of the input file. The desired operation sequence would be - the user is told how many records exist in the input file. The user runs a simulation. Then the user decides that the interesting part of the trace file is from records 100 - 200. So, now the user goes back to the fileinput module and tells Pablo to redo the graph execution for only these input records in the specified range (figure 2). So, he reconfigures the fileinput module with this input range of input records and reruns the simulation. Thus, effective browsing of the input file is provided.

6.2 Structure of the module

There were some design decisions as to how to actually add this feature to Pablo. Initially we decided to add a functional unit that would work as a number filter. In other words, it would at run time, read in all the data records, and put out only a selected range of records on the output pipe. This range would be selected at the configuration stage by the user. In order to do this, we needed a dynamic count of the number of data records at the input pipe of this module. However, the Pablo model of execution is one-one in which a data records starts off at the start of the processing pipeline and trickles trough the various data processing modeules getting modified. In other words, by the time a dynamic count is available, the input records themselves are unavailable. The next option was to write a Functional Unit that could be configured to read directly from a file and thus have the dynamic count at configure time. However, Pablo requires all Functional Units to have an input pipe. It treats file I/O specially and these are the only modules that can function without input. So, the decision we took was to augment the FileInput module with a filter capability. We did this by adding two scale widgets that could be used to specify the range of interest of the input file.

6.3 Integrating the module into Pablo

As described above, we decided to modify the FileInput module for our purposes. We provided a CtrlInput class that has among its member elements two scale widgets and internal data structures that represent the range of input. Each member of the FileInput class has associated with it a CtrlInput class member. Thus every input file can be controlled independently. We had to indicate to Pablo to include this class also during the build process.

6.4 Installation of Enhanced Pablo

The following steps have to be followed to install the new Pablo with *CtrlInput*.

```
o setenv CTRL /nfs/PABLO/src/CtrlInput1.0

o setenv PABLO_CTRL_HOME /nfs/PABLO/src/pablo3.0

o setenv PABLO_CTRL_SRC $PABLO_CTRL_HOME/Visual/Src/System

o setenv PABLO_CTRL_BUILD $PABLO_CTRL_SRC/Build

o ln -sf $CTRL/CtrlInputWrapper.C $PABLO_CTRL/Wrapper/CtrlInputWrapper.C

o ln -sf $CTRL/CtrlInputWrapper.h $PABLO_CTRL/Includes/CtrlInputWrapper.h

o ln -sf $CTRL/FileInputWrapper.C $PABLO_CTRL_SRC/Wrapper/FileInputWrapper.C

o ln -sf $CTRL/FileInputWrapper.h $PABLO_CTRL_SRC/Includes/FileInputWrapper.h

o cd $PABLO_CTRL_BUILD

o Add the following line to ObjFiles:
        CtrlInputWrapper.o

o cd $PABLO_CTRL_HOME

o make all

o make install
```

The procedure to be followed to build original Pablo from the new Pablo is listed in Appendix B.

7 Conclusions

Pablo provides a good interface to tracing parallel programs. The formulation of the SDDF format lets us use Pablo on traces coming from different parallel platforms. This is particularly useful, because it becomes an universal environment to analyze parallel programs for an user of different machines. The *ksr2sddf* utility adds to the base set of parallel trace programs which Pablo can analyze. Often the trace data of these programs is really huge and one is interested in closely examining only a part of it. The original Pablo FileInput module does not provide us with this facility. However, the new CtrlInput module adds this functionality. The software organization of Pablo, though not free of hiccups, does make it easy for adding functionality. Finally, keeping in mind proprietary software, it may be useful to rewrite the Pablo graphical user interface using free software (possibly Tcl/Tk).

8 Acknowledgements

I would like to thank Donna Bergmark for her guidance and support throughout the course of this project.

References

[1] Ruth A.Aydt. The Pablo Self-Defining Data Format. University of Illinois at Urbana Champaign, Department of Computer Science, March, 1992

[2] The Picasso Research Group. A Description of Classes and Methods of the Pablo SDDF Interface Library.University of Illinois at Urbana Champaign, Department of Computer Science, Feb, 1992

[3] Ruth A.Aydt. An Informal Guide to Using Pablo. University of Illinois at Urbana Champaign, Department of Computer Science, May, 1992

[4] Pablo Performance Analysis Environment

A Ftp sites for GNU

prep.ai.mit.edu
wuarchive.wustl.edu
ftp.uu.net
col.hp.com
uxc.cso.uiuc.edu
gatekeeper.dec.com

B To Build Original Pablo from the new Pablo

```
In case you want to build  the Original Pablo do the following:
(These steps are to build orginal Pablo when you have the new Pablo
 installed and running)

o setenv CTRL /nfs/PABLO/src/CtrlInput1.0

o setenv PABLO_CTRL_HOME /nfs/PABLO/src/pablo3.0

o setenv PABLO_CTRL_SRC $PABLO_CTRL_HOME/Visual/Src/System

o setenv PABLO_CTRL_BUILD $PABLO_CTRL_SRC/Build

o rm $PABLO_CTRL/Wrapper/CtrlInputWrapper.C

o rm $PABLO_CTRL/Includes/CtrlInputWrapper.h

o ln -sf $CTRL/FileInputWrapper.C
                $PABLO_CTRL_SRC/Wrapper/FileInputWrapper_Orig.C

o ln -sf $CTRL/FileInputWrapper.h
                $PABLO_CTRL_SRC/Includes/FileInputWrapper_Orig.h

o cd $PABLO_CTRL_BUILD

o Delete the following line from ObjFiles:
        CtrlInputWrapper.o

o cd $PABLO_CTRL_HOME

o make all

o make install
```