EDPEPPS: A Graphical Environment for the Parallel Software Development Life-Cycle*

M.J. Zemerly, T. Delaitre, G.R. Justo, S.C. Winter
University of Westminster, Centre for Parallel Computing
115 New Cavendish Street, London W1M 8JS
Email: `edpepps-all@cpc.wmin.ac.uk`
Web Site: `http://www.cpc.wmin.ac.uk/~edpepps`

Abstract *This paper describes an environment for performance-oriented design of portable parallel software. The environment consists of a graphical design tool for building parallel algorithms, a state-of-the-art simulation engine, a CPU characterisation tool, a distributed debugging tool and a visualisation/replay tool. The environment is used to model a virtual machine composed of a cluster of heterogeneous high-performance workstations interconnected by a local area network. The simulation model used is modular and its components are interchangeable which allows easy re-configuration of the platform. The model is validated using experiments on two parallel Givens linear solver algorithms with average errors of about 8%.*

Keywords: Performance Prediction. Graphical Design. Simulation. Visualisation. Parallel Programming Environments

1 Introduction

The emergence of high-speed networks and high-performance processors is making networks of computers an appealing vehicle for cost-effective parallel computing. However, a major obstacle to the widespread adoption of the use of network in parallel computing is the difficulty in program development due mainly to lack of parallel programming design tools. In particular, there is a need for performance-oriented tools, especially for clusters of heterogeneous workstations, to enable software designers to choose between design alternatives such as different parallelisation strategies or paradigms. A portable message-passing environment such as Parallel Virtual Machine (PVM) [5] permits a heterogeneous collection of networked computers to be viewed by an application as a single distributed-memory parallel machine. Rapid prototyping is a useful approach to the design of (*high-performance*) parallel software in that complete algorithms, outline designs, or even rough schemes can be evaluated at a relatively early stage in the program development life-cycle, with respect to possible platform configurations, and mapping strategies. Modifying the platform configurations and mappings will permit the prototype design to be refined, and this process may continue in an evolutionary fashion throughout the life-cycle before any parallel coding takes place.

The EDPEPPS environment is based on a rapid prototyping philosophy and comprises five main tools:

- A graphical design tool (PVMGraph) [6] for designing of parallel applications, based on PVM.
- A simulation tool (SES/Workbench [15]) based on discrete-event simulation.
- A visualisation tool (PVMVis) for animation of program execution using traces generated by the simulator and visualisation of platform and network performance measures and statistics.
- A CPU characteriser for computational code based on basic operations in C to al-

*This project is funded by an EPSRC PSTPA programme, Grant No.: GR/K40468 and also by EC Contract Numbers: CIPA-C193-0251 and CP-93-5383.

low the simulator to predict the execution time of each block depending on which machine the code is executed.

- A debugging tool (PVMDebug) [1] based on the distributed debugging tool DDBG [8] which provides debugging functionalities for distributed C/PVM programs.

Other tools used in the environment for integration purposes are:

- A trace instrumentation utility (Tape/PVM) [10].
- A translator (SimPVM) [3] from C/PVM code to queueing network graphical representation.
- A translator from existing C/PVM parallel applications into PVMGraph graphical representation (C2Graph) [14] based on the SAGE++ toolkit [2].

This paper describes the various tools within the EDPEPPS environment and presents a case study for validation of the models used. In the next section we describe several modelling tools with similar aims to EDPEPPS and we highlight the differences between them. In section 3 we describe the tools in the EDPEPPS environment. In section 4 we present results obtained from the case study. Finally, in section 5 we present conclusions and future work.

2 Parallel Performance Modelling Tools

Several parallel software modelling environments supporting performance engineering activities have been developed recently [11] but few of them follow the prototyping approach of EDPEPPS. In this section, we describe the most significant of these environments and compare them with EDPEPPS.

The GRADE (GRAphical Development Environment) [8] supports the main activities in the parallel software development cycle in a similar way to EDPEPPS. The key difference to EDPEPPS is its graphical language, GRAPNEL [7], which is based on its own message passing interface implemented on top of PVM and therefore does not model exactly the semantics of C/PVM (unlike EDPEPPS). In addition, GRAPNEL is a "pure" graphical language in the sense that even the sequential part of each process must be described graphically. In EDPEPPS, combined graphical and textual representations are permitted.

The ALPSTONE project [9] comprises performance-oriented tools to guide a parallel programmer. The process starts with an abstract description. From this description, it is possible to generate a time model of the algorithm which allows performance estimation and prediction of the algorithm runtime on a particular system with different data and system sizes. This project differs from EDPEPPS in two ways. Firstly, the language used to describe the program skeletons in not graphical and too abstract, which means that the derivation of an implementation involves several refinement steps. Secondly, the approach to performance prediction is based on analytical modelling and not discrete-event simulation models, as EDPEPPS.

3 The EDPEPPS Environment

Figure 1 shows the components of the EDPEPPS environment. The design process within the EDPEPPS environment starts with the graphical design tool (PVMGraph) by building a graph representing a parallel program design based on the PVM programming model. The graph is composed of computational tasks and communications. The tool provides graphical representation for PVM calls which the user can select to build the required design.

The software designer can then generate (by the click of a button) C/PVM code (.c files) for both simulation and real execution. The environment also provides a tool to translate existing parallel C/PVM code into graphical representation suitable for PVMGraph.

In the simulation path each C/PVM source code file obtained from the PVMGraph is instrumented using the Tape/PVM trace pre-

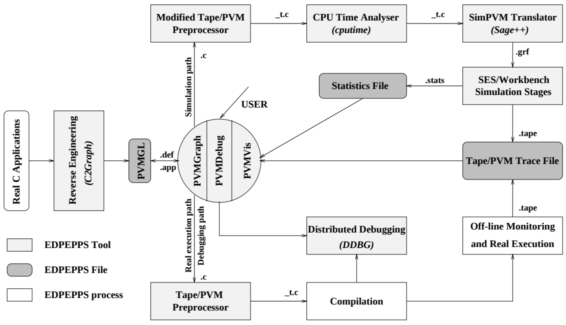

Figure 1: The EDPEPPS Environment.

processor [10]. The output is then parsed using the CPU Time Analyser (CTA) to characterise the code and insert *cputime* calls at the end of each computational block. The instrumented C source files are translated using the SimPVM Translator [3] into a queueing network representation suitable for Workbench graph (.grf file). SES/Workbench translates the graph file into the Workbench object oriented simulation language called SES/*sim* [15] using an SES utility (*sestran*). The *sim* file is then used to generate an executable model using some SES/Workbench utilities, libraries, declarations and the PVM platform model. The simulation executable code is run using three input files containing parameters concerning the target virtual environment (e.g. number of hosts, host names, architecture, the UDP communication characteristics and the timing costs for the set of instructions used by the CTA [4]). The UDP model and the instruction costs are obtained by benchmarking (benchmarks are provided off-line) the host machines in the network.

The simulation outputs a Tape/PVM trace file and a statistics file about the virtual machine. These files are then used by the visualisation tool (PVMVis) to animate the program execution and visualise the performance of the system. The design can be modified and the same cycle can be repeated until a satisfactory performance is achieved.

In the real execution path the Tape/PVM pre-processor is used to instrument the C source files and these are then compiled and executed to produce the Tape/PVM trace file required for the visualisation/animation process. This step can be used for validation of simulation results but only when the target machine is accessible.

In the debugging mode, the user will be able to debug the loaded C/PVM application. The distributed debugging of the application is handled by controlling the execution of all the tasks using a textual window representing the code for each task. The textual window here is slightly different from that of the design and offers the user basic features of a debugging tool. PVMGraph, PVMVis and PVMDebug are incorporated within the same Graphical User Interface where the designer can switch between these possible modes. The following sections describe in more detail the main tools within EDPEPPS.

3.1 PVMGraph

PVMGraph is a graphical programming environment to support the design and implementation of parallel applications. PVMGraph offers a simple yet expressive graphical representation and manipulation for the components of a parallel application. The main function of PVMGraph is to allow the parallel software designer or programmer to develop PVM applications using a combination of graphical objects and text. Graphical objects are composed of boxes which represent tasks (which may include computation) and arrows which represent communications. The communication actions are divided into two groups: input and output. The PVM actions (calls) are numbered to represent the link between the graph and text in the parallel program. Also different types and shapes of arrows are used to represent different types of PVM communication calls. Parallel programs (PVM/C) can be automatically generated after the completion of the design. Additionally, the designer may enter PVM/C code directly into the objects. The graphical objects and textual files are stored separately to enable the designer to re-use parts of existing applications [6].

3.2 PVMVis

The main objective of this tool is to offer the designer graphical views and animation representing the execution and performance of the designed parallel application.

The animation is an event-based process and is used to locate an undesirable behaviour such as deadlocks or bottlenecks. The animation view in PVMVis is similar to the design view in PVMGraph except that the pallet is not shown and two extra components for performance analysis are added: barchart view and platform view. The barchart view shows historical states for the simulation and the platform view shows some statistics for selected performance measures at the message passing layer, the operating system layer and the hardware layer.

3.3 PVMDebug

PVMDdebug is based on a Distributed Debugging tool (DDBG[8]) which provides a set of debugging functionalities for distributed programs written in C/PVM.

In order to debug a distributed application and to control the execution of the distributed tasks, the debugger needs to know all PVM task identifiers (tids). During the execution, the graphical task objects in PVMGraph do not have their tids because their values change at each run. Therefore, a debugging interface in the form of a wrapper program and a mapping table was developed to manage the link between the graphical objects and the PVM execution tasks. The wrapper program uses the on-line monitoring facility in PVM and collects the PVM events from the application and routes them to PVMDebug which manages the mapping list [1].

3.4 CPU Time Analyser

The CPU Time Analyser (CTA) estimates the time taken by computational blocks within a parallel algorithm. The CTA characterises a workload by a number of high-level language instructions (e.g. float addition) [4] taking into account the instruction and data caches effect. Assumptions have been made to reduce the number of possible machine instructions to 43 (see [4] for more details). The costs associated with the various instructions are kept in a file in the hardware layer accessible by the SES utilities. These costs are obtained by benchmarking the instructions on different machines.

The CTA first parses an instrumented C/PVM program using the SAGE++ toolkit. In the second stage, the CTA traverses the parse tree using the SAGE++ library and inserts *cputime* calls with the number of machine instructions within each sequential C code fragment. A *cputime* function has a fixed number of parameters (a total of 31). This is different from the number of machine instructions because the instruction cache duplicates some of the instructions (hit or miss). Each parameter of the *cputime* function represents the number of times each instruction is executed within the code fragment. The last parameter determines whether the instruction cache is hit or miss for the fragment in question.

3.5 C2Graph

This tool is provided to allow existing C/PVM parallel code to be directly used within the EDPEPPS environment rather than building the design from scratch. The C/PVM application files are first parsed using SAGE++ to get the .dep files. These files are then traversed by the C2Graph translator which also uses the SAGE++ library routines. The translator also takes into account the PVM calls in the original code and generates their corresponding graphical representation in the PVMGraph files. The translator then determines the master process, positions it with the other tasks by calculating appropriate coordinates for them in the PVMGraph screen, and writes the PVMGraph definition files (.def) for each task. The translator finally writes the application file (.app) required for PVMGraph.

3.6 SimPVM Translator

From PVMGraph graphical and textual objects, executable and “simulatable” PVM programs can be generated. The “simulatable” code generated by PVMGraph is written in a special intermediary language called SimPVM, which defines an interface between PVMGraph and SES/Workbench [3].

To simulate the application, a model of the intended platform must also be available. Thus, the simulation model is partitioned into two sub-models: a dynamic model described in SimPVM, which consists of the application software description and some aspects of the platform (e.g. number of hardware nodes) and a static model which represents the underlying parallel platform.

The SimPVM language contains C instructions, PVM and PVM group functions, and constructs such as computation delay and probabilistic functions.

3.7 EDPEPPS Simulation Model

The EDPEPPS simulation model consists of the PVM platform model library and the PVM programs for simulation. The PVM platform model is partitioned into four layers: the *message passing layer*, the *PVM group layer* which sits on top of the message passing layer, the *operating system layer* and the *hardware layer*. Modularity and extensibility are two key criteria in simulation modelling, therefore layers are decomposed into modules which permit a re-configuration of the entire PVM platform model.

A program generated by the PVMGraph tool is translated into the SES/Workbench simulation language and passed to its simulation engine, where it is integrated with the platform model for simulation. The message-passing layer models a single (parallel) virtual machine dedicated to a user. It is composed of a daemon which resides on each host making up the virtual machine, a group server and the libraries (PVM and PVMG) which provide an interface to PVM services.

The main components in the operating system layer are the System Call Interface, the Process Scheduler and the Communication Module.

The hardware layer is comprised of hosts, each with a CPU layer, and the communications subnet (Ethernet). Each host is modelled as a single server queue with a time-sliced round-robin scheduling policy. The communications subnet is Ethernet, whose performance depends on the number of active hosts and the packet characteristics.

4 A Case Study

A case study is presented here to illustrate the use of the EDPEPPS environment. The case study selected is a communication intensive application based on two parallel Givens Linear Solver method developed in [12]. The EDPEPPS environment can help the designer to choose the best algorithm and to size the application by choosing the best number of processors. In this case study we also test the accuracy of the EDPEPPS CPU characterisation toolset using the sequential Givens code.

4.1 Givens Linear Solver

The Givens linear solver can be represented in the form: $Ax = b$, where A is a non-singular square matrix, b is the right hand side vector and x is a vector of unknowns. The Givens transformation is defined by a 2×2 rotation matrix:

$$G = \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \quad \text{where } c^2 + s^2 = 1 \qquad (1)$$

A Givens rotation is used to eliminate the elements of a vector or a matrix as follows:

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \times \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix} \qquad (2)$$

where $c = \frac{a}{\sqrt{a^2+b^2}}$ and $s = \frac{b}{\sqrt{a^2+b^2}}$

The Givens algorithm for solving a linear system with N equations can be decomposed into two computational stages. The first is the triangulation of the initial matrix, this stage is represented by the execution of the elimination block N times. The second stage is the substitution block which solves the triangular matrix.

4.2 Parallel Givens Algorithms

The triangulation stage, which is the time consuming part (with complexity of $O(N^3)$ as opposed to $O(N^2)$ for the back-substitution stage) is parallelised with two different techniques: collective and pipeline. The back-substitution stage is also programmed differently as will be seen later.

Initially, block-row data decomposition, which divides the matrix horizontally and assigns adjacent blocks of rows to neighbour processors is used in both methods. The first step (A) in the collective method, all processors eliminate the required columns for their rows (except the first row for each processor) in parallel. Then to eliminate the columns of the first rows (step B), collective communication is used to collect the first rows from processors (rows are collected by the processor holding the corresponding row of the current eliminated column) and distribute the rows back. The back-substitution stage is started by the last processor and its results are broadcasted to other processors and then in a similar way all the processors solve their triangulated matrices in a back-pipeline way.

In the second pipeline method the same first step A (as in method 1) is used but instead of using collective communications to eliminate the columns of the first rows (step B in method 1), the row in question is passed from the sender to its neighbour to eliminate its column and then passed in a pipeline fashion through other neighbours until all the columns are eliminated. The last processor keeps the lines in a full matrix to be used in the back-substitution stage later on its own.

4.3 Results and Discussion

The two methods were designed using the EDPEPPS environment and results for both simulation and real execution (on a real network) were obtained. The network used for this algorithm consists of 3 Ultra-Sparc 10 machines with 300MHz, A Pentium II with 233 MHz, A Pentium 150MHz, A Super-Sparc 20 with 75 MHz and A Micro-Sparc 5 with with 60 MHz. The mapping of tasks onto machines was done differently in the two algorithms based on a few simulation tests and the final optimal mapping only is used for the results shown here. The mapping for the machines was done in increasing power order (but giving priority to use the fastest machines if the number of processor is less than the total 7) for the pipeline method and in decreasing power order for the collective method.

The tests were done for problem sizes of 256 and 512 equations but for the 256 size we did not get any significant speedup relative to the fastest machine in the network as the algorithm is communication intensive and this will affect the performance of small problem sizes.

Figure 2 shows the results for the simulation and real execution measurements (averages of 10 times taken at night) for both the collective and the pipeline methods.

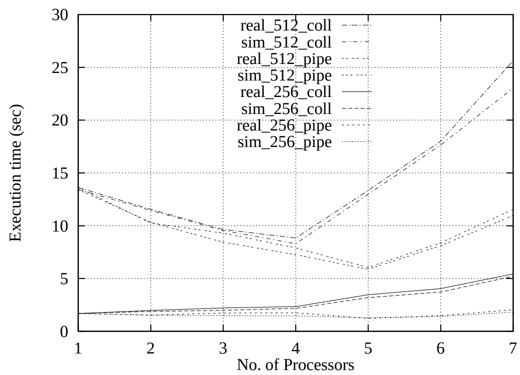

Figure 2: Comparison between predictions and measurements for Parallel Givens: collective and pipeline (the sequential results are for the UltraSparc 10 with error of 1.5% only.)

The figure shows clearly that the measurements and predictions for both methods are in good harmony with maximum error well below 10% (except for one case for the collective algorithm with 7 processors, 12%). All the measurements including the benchmarks have been performed at night at low ambient network load and the standard deviation for all the measurements did not exceed 5% on any of the runs. As expected, the figure also shows the superiority of the pipeline algorithm over the collective algorithm even for small number of processors. Note that the machines are heterogeneous and adding more processors sometimes results in increasing the execution time rather than improving it. For the 512 problem size we obtained a speedup of 2 and 2.25 for 4 and 5 processors respectively for the pipeline algorithm. As expected, increasing the number of processors above 5 (4 for the collective) in both cases increases the execution time as the other processors (other than the UltraSparc and Pentium II - 233 MHz) are considerably slower than the fastest four. Knowing that the pipeline algorithm is superior to the collective one, the user can then experiment with adding more powerful processors than available to see if better speedups can be obtained.

5 Conclusion

This paper has described the EDPEPPS environment which is based on a performance-oriented parallel program design method. The environment supports graphical design, performance prediction through modelling and simulation, and visualisation of predicted program behaviour. The designer is not required to leave the graphical design environment to view the program's behaviour, since the visualisation is an animation of the graphical program description. It is intended that this environment will encourage a philosophy of program design, based on a rapid synthesis-evaluation design cycle, in the emerging breed of parallel program designers.

Success of the environment depends critically on the accuracy of the underlying simulation system. Preliminary validation experiments showed average errors between the simulation and the real execution of less than 10%.

An important direction of our work is to generalise the simulation model and extend it to support other platforms, such as MPI.

References

[1] A. Audo, Integration of the DDBG Distributed Debugger within the EDPEPPS Toolset, Technical Report, Centre for Parallel Computing, University of Westminster, UK, June 1998.

[2] F. Bodin *et. al.*, Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools, *Proc. 2nd Annual Object-Oriented Numerics Conf.*, 1994.

[3] T. Delaitre *et. al.*, Final Syntax Specification of SimPVM, EDPEPPS/22, Centre for Parallel Computing, University of Westminster, UK, March 1997.

[4] T. Delaitre *et. al.*, Final Model Definition, EDPEPPS/23, Centre for Parallel Computing, University of Westminster, UK, April 1997.

[5] A. Geist *et. al.*, *PVM: Parallel Virtual Machine*, (MIT Press, 1994).

[6] G.R. Justo, PVMGraph: A Graphical Editor for the Design of PVM Programs, EDPEPPS/5, Centre for Parallel Computing, University of Westminster, UK, Feb 1996.

[7] P. Kacsuk, P. Dozsa and T. Fadgyas, Designing Parallel Programs by the Graphical Language GRAPNEL, *Microprocessing and Microprogramming* 41 (1996) 625-643.

[8] P. Kacsuk *et. al.*, A Graphical Development and Debugging Environment for Parallel Programs, *Parallel Computing*, 22:1747-1770, 1997.

[9] W. Kuhn and H. Burkhart, The ALPSTONE Project: An Overview of a Performance Modelling Environment, In *2nd Int. Conf. on HiPC'96*, (McGraw Hill 1996) 491–496.

[10] E. Maillet, Tape/PVM: An Efficient Performance Monitor for PVM Applications: User Guide, LMC-IMAG, ftp://ftp.imag.fr/ in pub/APACHE/TAPE, March 1995.

[11] C. Pancake, M. Simmons and J. Yan, Performance Evaluation Tools for Parallel and Distributed Systems, *Computer* 28 (1995) 16–19.

[12] J. Papay, M.J. Zemerly, G.R. Nudd, Pipelining the Givens Linear Solver on Distributed Memeory Machines, *Supercomputer Journal* 65, XII-3, 1996, pp 37–43.

[13] P. Pouzet, J. Paris and V. Jorrand, Parallel Application Design: The Simulation Approach with HASTE, in: W. Gentzsch and U. Harms, ed., *HPCN* 2 (1994) 379–393.

[14] F. Schinckmann, Reverse Engineering Tools, EDPEPPS/36, Centre for Parallel Computing, University of Westminster, UK, July 1997.

[15] Scientific and Engineering Software Inc., SES/Workbench Reference Manual, Release 3.1, 1996.