Modeling parallel programs through their execution graphs

Aleardo Manacero Jr.[1] and André L. Morelato França[2]

[1] Computer Science Department, Paulista State University
R. Cristovao Colombo, 2265, S.J do Rio Preto, Brazil
[2] College of Electrical and Computer Engineering, State University of Campinas
Campinas, Brazil

Abstract. This paper describes a novel approach to create a program model, based in the construction of an execution graph from its binary code. This model can be used as a program model in the Herzog's Three-Step Methodology for performance prediction. When applying the execution graph to model the program one can overcome instrumentation problems and achieve a very accurate benchmarking if a good simulator is used. Some results achieved with this technique are presented.

1 Introduction

Designing parallel programs is a rather complex task, mainly due to its different programming paradigm and to the hardware cost. The former problem can be overcome through performance analysis until convenient algorithm and problem decomposition are achieved. The latter is, at last, unavoidable, but can be postponed until the final phases of software design if benchmarks are not necessary. Unfortunately the best solution for these problems are conflicting, since real machine benchmarks usually produce better data for performance analysis than simulation or analytical models.

The approaches found in the literature for performance evaluation can be classified either by their nature, comprising analytical [6],[20],[21], benchmarking [2],[4],[16],[18], and simulation approaches [10],[14],[22], or by their data collecting form, comprising monitoring and code modification (Pierce and Mudge [17] and Reed [18] give a thorough description of such classification schemes). Although monitoring might be much more accurate, it is also much more expensive. On the other hand, code modification techniques, such as profiling and event tracing, require that extra code be inserted into the original program (code intrusion), introducing inaccuracies in the performance analysis. Even though benchmarking can be very accurate, it is an expensive technique. Yet analytical and simulation based techniques may be much less expensive, their accuracy depends strongly on the quality standard of modeling that describe both the program and machine environment. In short, to find out the right balance among data acquisition form, measurement techniques and performance analysis is a complex task and remains an open issue.

Besides the inherent difficulties on modeling, this task gets even more hazardous on the model of parallel systems. In such systems the model is more complex due to synchronization and cooperation between processors, which imply on new metrics for evaluation [1],[11],[19].

Here, a non-intrusive new approach for modeling of parallel programs is proposed. This method is composed by two stages: first, the executable code of the parallel program under analysis is converted into a directed graph which maps all the computation paths of the program; second, the flows described by this execution graph are optimized in order to reduce the number of nodes and arcs in the graph. The resulting graph can be simulated at a later moment, providing the performance data needed for the program analysis.

The advantages of this approach are: high accuracy, provided by the rewriting of the executable code, and, later, low cost of simulation, which need not be carried out on a real parallel machine. The proposed approach, for the whole performance evaluation process, follows the Herzog's three-step methodology [9], in which the program modeling, the machine modeling and the modeling of the program-machine interaction are handled separately. Here, the parallel program model is represented

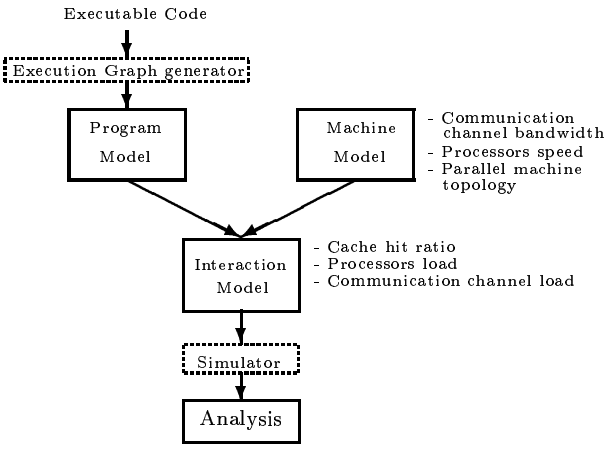


Fig. 1. Herzog's methodology mapped onto the proposed technique.

by its execution graph and the machine model as well as the interaction model should be provided by the simulator, as illustrated in Figure 1.

In the remaining text one finds a brief description of the research on graph-based performance analysis tools in the very next section. After that, a complete review of the execution graph approach is made, including the problems involved with the modeling of certain programming structures. The fourth section contains information about the implementation of this approach into a prototype, and results achieved with it. Finally, in the last section, we point out some relevant conclusions from this work.

2 Related work

As can be drawn from the Introduction, there are many different approaches to analyze the program's performance when it runs on a specific system. Such efforts can be classified by the technique used for data acquisition, separating them roughly in three large groups: analytical, benchmarking and simulation methods. Each class has some advantages and disadvantages, accordingly to the parameter used as the chosen metric. If the cost of analysis is considered, the less expensive ones are analytical and simulation methods. However, in both cases the accuracy of the analysis is strongly tied with the models for software and hardware.

Among the analytical and simulation methods it is possible to notice that most of them have, as their background, some kind of graph to model the software or the whole system. The importance of graph models in such systems can not be neglected since a graph is a very powerful mathematical tool to represent any dynamical system.

Since our proposal is also graph-based, all the work presented in this section is also graph-based. In the scenario for analytical techniques and tools, the most typical methods to provide software models include stochastic petri nets, queueing theory, Markov chains and other forms of graphs. As one sees, the preferred model is probabilistic in nature. Deterministic models are usually avoided since the program execution on computers suffer from unexpected conditions in the computing environment.

In the Petri nets field one may find the work of Gandra et all [6], that uses generalized stochastic Petri nets (GSPN) to model the whole system. The same technique (GSPNs) is used by Marsan et all [15] to evaluate multiprocessed systems. The great advantage of using Petri net models is that their models are reasonably easy to understand and analyze. Their problem is related to the inaccurate nature of time treatment by the Petri net models, which would lead to inaccurate results.

Markov chains are present in several techniques, such as the works from Kapelnikov et all [12], or Trivedi et all [21], whose differences are in the methods used to reduce the number of states in the chains. The accuracy of models created with Markov chains are mostly constrained by the capability of dealing with infinite chains of states and the probabilistic computations underneath the model.

A third technique, used mostly for large-sized grains, is the queueing theory. Here the software is treated as a set of clients to be served by the processors, with clients executing during some kind of probabilistic distributed intervals, as in Gelenbe and Liu [8] or Sotz [20]. The main problem with queues is the exact match between the distribution functions and the real world timings, which can be circumvented by simulation.

Other techniques based on graphs usually build graphs from function calls lists (large-sized grains) or path profiles from program execution (small-sized grains). In all cases a characteristic in common is the use of post-mortem analysis, that is, the tool builds a graph that is analyzed in a second moment, when the program no longer runs. Work on this category includes Alpes [13] or IDTrace [17]. Most of these graphs are built from benchmarking efforts over the program, as done by Paradyn [3] and P3T+ [5].

Our approach is distinct from all of the previous work in the sense that it performs all of its actions outside of the real system. Although the graph generation is done using the real machine code, and therefore is highly accurate, it can be executed on a completely different machine. The data acquisition and analysis need only a simulator capable of reading and understanding the graph structure, which also may be executed elsewhere. This makes the process of providing performance data inexpensive and indeed accurate. When someone thinks about the large amount of investment needed to purchase parallel machines (even for Beowulf clusters), being capable of tuning the program without using the real machine is a very useful goal.

3 The execution graph methodology

As stated before, this method models the parallel program as a directed graph built from the executable code. Figure 1 shows how a typical performance analysis tool should map execution graph into the Herzog's 3-step methodology. From that figure, one sees that it can be implemented through two separate subsystems, an *execution graph generator* and a *graph simulator*. The performance data may be collected using only the executable code for the program and general knowledge from the physical and logical environment.

These subsystems, and the methodology underneath them, differs from other similar approaches in the sense that the program's model maps its real counterpart in a higher correlation than other techniques, and that the simulator could extract the performance data without any intrusive instrumentation. These distinctions are natural consequences of the execution graph model and can be better understood after few basic definitions of what such graph stands for.

3.1 The execution graph

In the execution graph, vertices represent sets of assembly instructions that must be executed strictly in a sequential order, without any break/branch point. An illustrative graph is seen in Figure 2. There, the arcs map precedence constraints between any two vertices, that is, there is an arc starting from vertex *v1* and incident to vertex *v2* if the computer must execute all instructions mapped by vertex *v1* before it may execute those on vertex *v2*. The execution of all assembly instructions clustered into a vertex is called the execution of such vertex.

We classify the vertices into categories accordingly to the kind and purpose of assembly instructions clustered into them. The defined categories are used to coordinate the simulation process. A vertex can be classified twofold: by the number-kind of edges and by the action performed when it gets executed.

The number and kind of edges in a vertex describes the existence of loops, decisions and exit points in the program. There are five basic types of vertices following this approach:

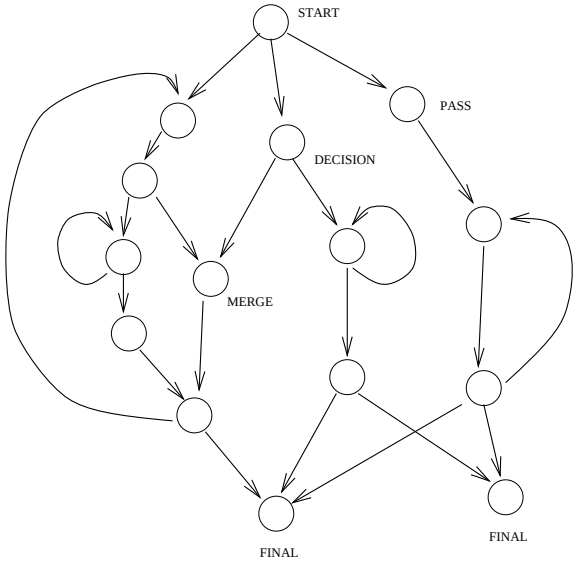


Fig. 2. Execution graph of a hypothetical program.

- **START**: no edges incident to it.
- **PASS**: only one edge incident to and only one edge incident from it.
- **DECISION**: one edge incident to and more than one incident from it.
- **MERGE**: one edge incident from and more than one incident to it.
- **FINAL**: no edges incident from it.

The unique restriction is that any execution graph has one and only one START vertex since it maps the starting point of the program. These basic types can be merged to create a more complex vertex. The constraints to follow in the merge procedure are related to the action performed by each original vertex in the merge, what leads to the second form for vertex classification. At this time it is reasonable to define three basic actions that should be considered as categories for vertices classification. They are the synchronization, execution and communication actions, defined as follows:

- **Synchronization**: denotes points where some parallel tasks should synchronize.
- **Communication**: denotes points where some parallel tasks communicate to each other. The communication can be synchronous or asynchronous.
- **Execution**: denotes any point that is not a synchronization or a communication point.

With vertices categorized by these parameters it is feasible to build a graph that maps all interactions between parallel programs in a accurate form. The problem that rises is how to achieve such graph from the executable code of a parallel program, which is described next.

3.2 Modeling programs through an execution graph

The first subsystem needed to implement this approach is the **Execution Graph Generator**, which will read the executable code for a parallel program and create an execution graph that will be simulated to produce performance data. This subsystem can be understood as a decompiler whose target language is a graph instead of the program's source language. The decompilation process involves three tasks:

1. code reading, which reads the executable code byte by byte and disassembles it;

2. instruction interpretation, that identifies the functional meaning of each machine instruction and maps it to actions that should be simulated afterwards;
3. instruction clustering, which merges consecutive instructions with similar functional meaning into a single block (a graph vertex to be).

There is not much challenge in the first two tasks. They are quite simple and should be performed instruction by instruction, in an ordered way. As a natural consequence of our method, they strongly depend on the processor that is used in the machine that will host the program under analysis. This puts a problem with the tool's portability, which can be solved with careful software engineering and implementation of processor-specific libraries.

After each instruction interpretation, the third task has to decide if the amount of time that the specific instruction takes at runtime should be accounted for in the current block (vertex) or it goes to a different, possibly new, block. This task is complex and time consuming. At this point, the graph should start to get shape, with finished blocks becoming vertices and the links between them, created by the sequence of updates in the program counter, becoming the edges in the directed graph. Although it is easy to decide when to start a new block, it is very hard to decide when two blocks should be linked. While the former decision is solved by the functional meaning of each instruction, with *jumps*, *calls* and *exits* marking the end of a block, the latter has to take in account addresses of branches, subroutine calls, and so on, which may not be easily available at the clustering time.

Every time that an edge is created, the addresses of the current instruction and the new target are used to uniquely identify the edge, including its flow direction. This is simple to manage for forward branches (current address is lower than target) inside the current program scope. However, for backward branches (target is lower than current) or subroutine calls/returns this creates several troubles to model the program in the right way. One major major trouble comes while managing program loops, subroutine calls and recursion. The techniques to solve them use some tricky features of the graph structure, which are described in the next paragraphs.

Loop modeling Before we start solving the loop problem we should show how loops are modeled from scratch. The first task is to identify the loop existence, which can be done simply looking for a backward branch inside a single scope. This condition holds true for every single loop in a program since to execute a loop one has to return to a previously executed instruction at the loop entry. The loop modeling difficulty comes from the variety of loop models, that is, from the different points where a loop condition can be tested while repeating its body and how they are implemented with machine instructions.

Structured loops exist in three different flavors: *while with conditional branch*, *while with unconditional branch* and *repeat-until* forms, as shown in Figure 3. The last two are easy to model, that is, it is quite simple to identify the vertex where the loop condition is tested. However, in the first form it is hard to detect where the loop condition should be tested since there are DECISION vertices on both ends of the loop body. Although it seems awkward, this construction is broadly used to implement "while" loops since the last DECISION vertex comes from a branch instruction that tests always a constant value (usually a Branch-If-Zero instruction testing a zero value). Note also that a similar problem is given by non-structured loops implemented by GOTOs instructions in low-level and some high-level languages.

In order to model such loops one has to observe that repeat-until loops never come with a DECISION vertex at its entry point. Therefore, if a loop is detected one has to check how many vertices inside the loop's body points to an address located outside of it. If there are more than one of such vertices, two possible conditions maybe present: the loop is non-structured or it is an while with unconditional branch. If the former occurs there is not much to worry about because non-structured loops can be easily modeled as repeat-until loops. In the latter case the loop model can be achieved replacing the final vertex by a PASS vertex that points to the first vertex of the loop body.

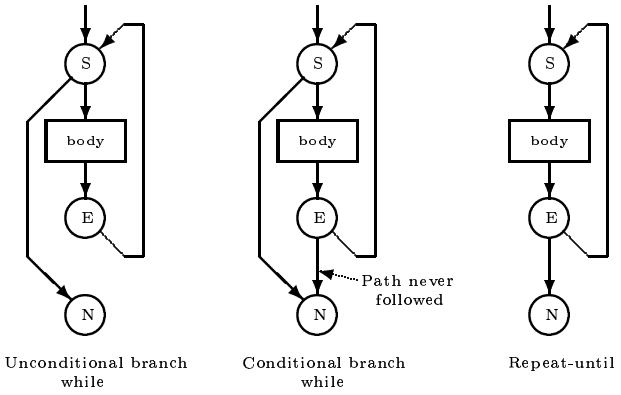


Fig. 3. Loop implementation flavors.

Another problem here is concerned with fixed-cycles loops (as "DO" in fortran or "for" in C), besides they always have the form of "while" loops and are modeled such as. The problem is how to register the fixed number of cycles that this loop will repeat during simulation time. The fixed number of iterations is set twofold: by a constant or by the runtime value of a loop limitant. In the second case we let the determination of its value for the simulator, since all runtime values are determined at that time. In the first case, however, the constant value can be easily determined by a quick trace over the direct parameters given in the hardware instruction.

Finally, the last problem related with loop modeling deals with the determination of backward vertices, that is, the vertex that defines the entry point of every loop. The backward address given by the last vertex will not be, exactly, the same initial address of the vertex in the entry point. This means that the vertex containing such address must be split into two new vertex, one that gets executed before the loop and another that becomes the actual entry point. The solution is the separation of these vertices, making the first one points only to the second one and the second one points to every address that the initial vertex pointed to. The execution time of both vertices are complements to the initial vertex time.

Subroutine modeling Subroutines pose a different modeling problem, which is related to the program's capability of calling them from distinct points. The procedure of building an execution graph composed by two or more subroutines follows exactly the same procedure used for a single block, although in the new scenario a subgraph is created for each separate subroutine. The subgraphs are connected with edges linking both caller and called subroutines, which leads, therefore, to problems on uniquely identifying the return point for multiple called subroutines, while avoiding unnecessary replication of subgraphs. This problem has an even greater impact, due to obvious reasons, when we are dealing with recursive subroutines.

A solution for the replication problem is to leave the resolution of the return address to be done at simulation time. This is feasible by the use of FINAL vertices for every returning instruction, making the simulator able to distinguish between subroutine returns and the program termination. With this strategy, the simulator works extra checking for unfinished subroutines every time it arrives in a FINAL vertex, but the graph generation becomes an easier task since now there is no difference between multiple- and single-called subroutines.

This solution brought an unexpected result with the solution of the troubles with recursive subroutines. In this case the problem was how to identify such subroutines and the number of times that a recursion would occur. The latter is postponed for simulation time, as we did for the determination of the return address, while the former is let unsolved, that is, leaving the number of recursive calls to be determined during simulation there is no need to identify what subroutines are recursive prior to that moment. Actually, they do not have to be identified even at that time.

Figure 4 sketches this solution scheme for a single call. There, the dashed edges in the graph represent the subroutine call and return. The call edge actually exists in the graph, while the return edge is virtual, becoming "physical" only during the simulation process. Therefore, if a subroutine is called from two different vertices, there are two call edges but the return edges will be created virtually only if they become necessary to proceed the computation.

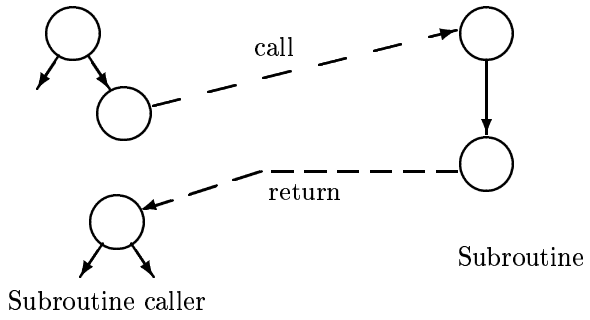


Fig. 4. Subroutine calls and returns

3.3 Graph optimization

During the generation of the execution graph the goal is to cluster as many instructions into a single vertex as possible. Besides this, the effective number of vertices and edges is usually very high (tens of thousands of vertices). In order to bring this number to a more manageable limit one has to perform several optimizations over a graph. Such optimizations are feasible from the composition of vertices eliminations and associations. The degree of optimization depends on the level of accuracy that the graph must keep after the operation, and then level of speed that is expected from the simulations (remember that the simulation is usually faster if there are less vertex and paths to be traversed).

The minimization on the number of vertices can be done in a variety of forms. In the standard form, the minimizations would be performed only if the resulting graph would provide exactly the same simulated times as the original one. In this case we say that the optimization is nondegenerative, in contrast with degenerative optimizations that occur in all other cases, where some of the minimizations may introduce small differences for the time evaluation. Here we will describe only those minimizations that do not modify de simulated time, keeping in mind, however, that sharper minimizations could be made simply ignoring the rules imposed for the non degenerative strategies.

Elimination of PASS vertices The first kind of minimization is the elimination of PASS vertices, as seen in Figure 5. The first restriction for this elimination to occur is that the PASS vertex cannot be either a synchronization or a communication point. A second constraint is that at least one of the edges must be the unique edge of that type in the neighbor vertex, that is, if we call the neighbors as *previous* and *next*, then either the edge incident to the PASS vertex is the unique edge incident from the *previous* vertex or the edge incident from PASS is the unique edge incident to the *next* vertex.

Elimination of MERGE vertices For MERGE vertices the restriction about communication and synchronization vertices also apply. The usual process is a forward elimination, where the vertex is incorporated into the *next* vertex. Forward elimination is performed always when the *next* vertex following the MERGE vertex has only one incident to edge, as shown in Figure 6(a). The constraint

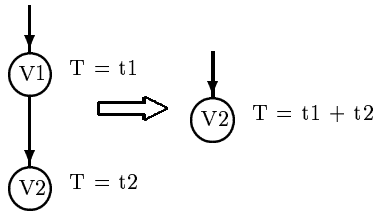


Fig. 5. PASS vertex elimination.

about the number of edges in the *next* vertex may be ignored under certain circumstances, when all the paths that are closed by the MERGE vertex were started after the other paths that arrive to the *next* vertex, as seen in Figure 6(b). In this case we say that a backward elimination takes place.

Elimination of DECISION vertices The last minimization technique deals with edges leaving a DECISION vertex, aiming the reduction on the number of available paths. This minimization is possible when none of the paths leaving the DECISION vertex contains either a synchronization or a communication or a MERGE vertex. The idea behind the technique is to keep the simulated time constant after the removal of the vertex with the minimal execution time (hoisting). The removal is performed at the expense of time increment in the DECISION vertex and decrement in the vertices following it. The procedure is depicted in Figure 7, where one vertex is removed. The number of removed vertex may be bigger if there are two or more vertices whose time is equal to the minimum.

4 Implementation of a graph generator

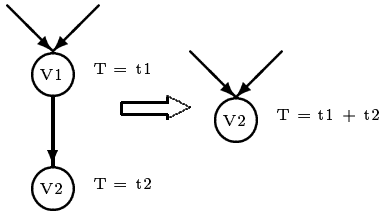
The methodology just described has been applied to the building of a generator capable to understanding codes from MIPS and Sparc processors. A set of libraries were written in order to uniquely identify and evaluate every instruction in the instruction set of these processors.

The choice for these processors was guided primarily by the behavior of their instructions, that is strongly limited by their RISC design. In other words, the instructions of these processors have uniformity on size and execution time. This uniformity leads to a simple treatment in the clustering process, since most (if not all) instructions have the same structure.

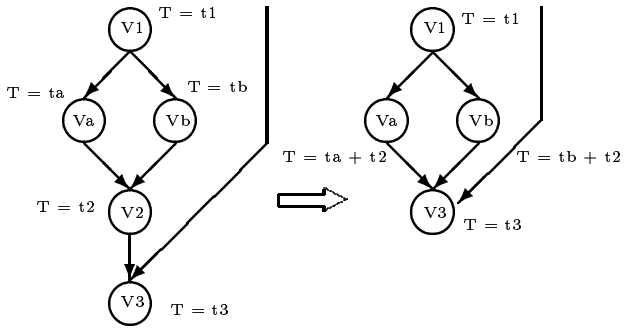
With this behavior, the generation of an execution graph becomes just a task of reading an instruction, decoding and inserting it in a given block. The decoding procedure for the MIPS and Sparc instruction sets is quite simple. In both cases the instructions can be separated as branch, arithmetic and processing instructions. The timing of each one can be easily evaluated from processor's manuals as well as their binary codification.

After the whole clustering process, there will be a forest of subgraphs as the resulting execution graph. The forest-like form of this graph is a direct consequence of the technique used to model subroutines, since each subroutine is mapped to a partially connected subgraph, whose connections are links to subgraphs mapping subroutines called from the first subroutine (its execution graph).

The output for this graph is a file containing data about each subgraph in consecutive lines. The format for this graph is seen on Figure 8, which depicts some of the first lines of a graph file. Every vertex in the graph is denoted by a single line where the first integer gives the type of that vertex (among execution, call, return, branches, communication, etc.). It is followed by a string that gives the logical address of its first instruction, an integer that gives the accounted number of clock cycles in its execution, addresses of candidates vertices as the next one to be executed and other information when necessary. The beginning of the graph file is composed by the list of all subgraphs, given as a pair of initial address and function name. Every entry on this list has a subgraph given, in the order of list's appearance, by a *"Tree Number N"* line.



(a) Forward



(b) Backward

Fig. 6. MERGE vertices eliminations.

4.1 Results

The results presented at this section were collected from a simple prototype of a tool built using this approach. This tool provided only coarse grained information about the evaluated programs. Besides not very useful as a performance prediction tool it provided meaningful insights about the technique, since we could verify both the graph generation and the measuring simulation phases.

In order to verify the correctness of this approach we built the graph of several small programs, including the master-slave programs coded in the PVM's manual [7]. Here we understand correctnes as the true execution path construction from the program executable, that is, the graph generation process is correct if the potential paths in the program are exactly match by the potential paths in the graph. From this definition of correctness the task was simply to verify, by hand, the generated graphs. As expected from the methodology described in the previous sections, all potential paths, and only them, were present in the graphs built. In all cases the graphs were verified by hand and provided exact matches for the potential paths in the execution of the programs. We do not show these tests here because they do not provide a significant insight about the methodology.

Besides the graph correctness test, we also executed timing tests, aiming the verification of accuracy at the simulation phase. The timing tests were performed through the comparison between the execution times of a benchmarked program and the simulated times. Here we show the results for a large program that is used for the reconstruction of trajectories of subatomic particles from data collected during a fixed-target accellerator experiment. This program is coded in Fortran and runs over small clusters of workstations. Each cluster has around 20 hosts and the program follows the bag-of-tasks model, with several *client* processes and a pair of *reader-writer* processes acting as the controller.

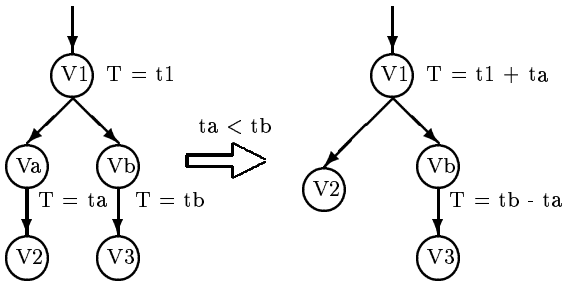


Fig. 7. Time composition reduction.

```
00407960 __start
00407a80 input_
00407ae0 async_reader_
                    ⋮
Tree
Tree Number 1
vtx 3 00407960 2 00407968 00407968 null
vtx 7 00407968 282 00407a44 00407a80 null
vtx 0 00407a44 112 return
Tree Number 2
vtx 7 00407a80 126 00407ad0 00407ae0 null
vtx 0 00407ad0 6 return
Tree Number 3
vtx 7 00407ae0 14 00407b08 00409400 null
vtx 7 00407b08 9 00407b20 004095e0 null
vtx 7 00407b20 1218 00407c80 00409ac0 null
vtx 7 00407c80 8 00407c98 0040a200 null
vtx 1 00407c98 235 1 id_sync 00407cf4 rdr cli null
vtx 0 00407cf4 5 00407d08
vtx 7 00407d08 12 00407d24 004133a8 null
vtx 3 00407d24 11 00407d40 00407d6c null
vtx 0 00407d40 18 00407d6c
vtx 4 00407d6c 79 00407e24 00407d08 null
vtx 2 00407e24 352 broadcast ···
                    ⋮
```

Fig. 8. Section of execution graph's output file.

The benchmarks were limited by the cluster's size and by the instrumentation that already provided. Therefore, the prototype should provide exactly the same. However, it had no limitations over the number of working nodes in the cluster, what enabled a test about the program's scalability.

The plots in Figures 9, 10, 11, and 12 shows the measured times achieved with both the simulator's prototype and the benchmarks. Figure 9 shows the data concerned with the execution timing, measured by the number of data records processed per second (each record contained about 64kbytes of data, concerned with about 10 to 12 particles). Since the environment and machine models for the simulator are provided by input parameters we tuned the simulator in two different views: communication oriented and execution oriented. The tuning was made simply by changing the delay value for communication vertexes, in order to provide higher or lower communication costs into the system.

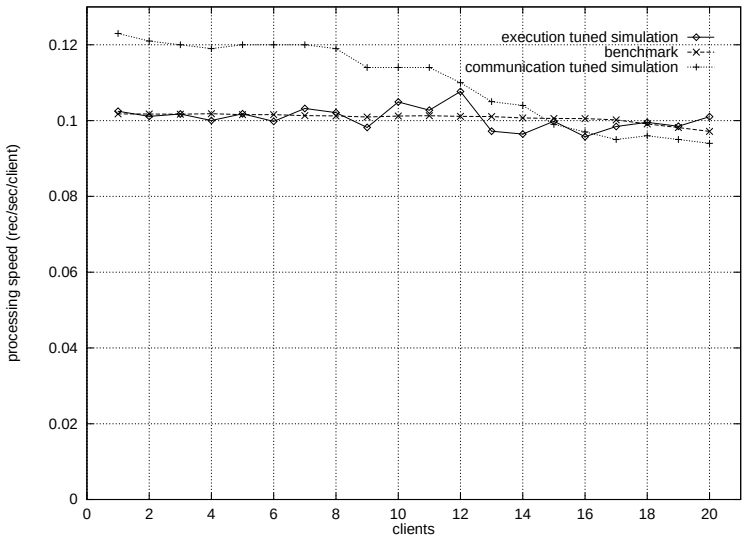


Fig. 9. Measured and simulated execution times.

As one can see from plot 9, when the simulator is oriented for execution the measured speeds have a higher correlation to the benchmarked values. A good correlation is also achieved with the communication oriented tuning, altough here the gradient of the speed reduction is higher, and the error goes up to 20%. This is, indeed, a reasonable result if one remembers that it is just a prediction.

On the other hand, the simulation could be performed for a large number of nodes, since it is not physically restricted by the actual cluster. Figure 10 shows the simulated processing speeds for a larger cluster. From this plot, or an equivalent one, the analyst can estimate speedup curves and, therefore, scalability limitations over the program and the parallel machine where it should run.

In order to understand the difference between the two opposite tunings used in our simulations we present here the results from communication delays with the simulator and the benchmarks. Figure 11 shows the communication delays achieved without tuning the simulator for communication costs. It is easy to see that the simulated costs grow in a much higher speed than the real ones. This happens because the execution oriented tuning does not take into account the fact that besides the system needs more messages between nodes, these messages occur in an interleaved fashion. If one observes this condition during the simulation we get the plot shown on Figure 12, which

Reporting to the Herzog's Three Steps methodology, this simulator should have inputs for both machine and software/hardware interaction models. These models were restricted to informations

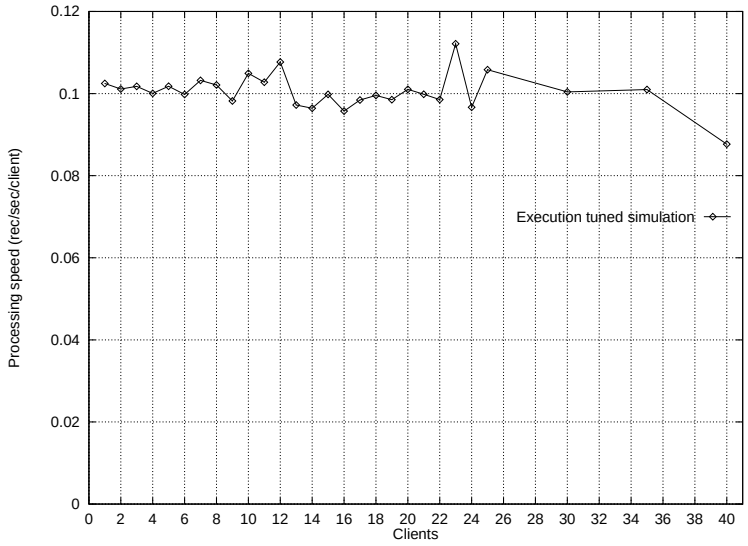


Fig. 10. Simulated execution times for a larger cluster.

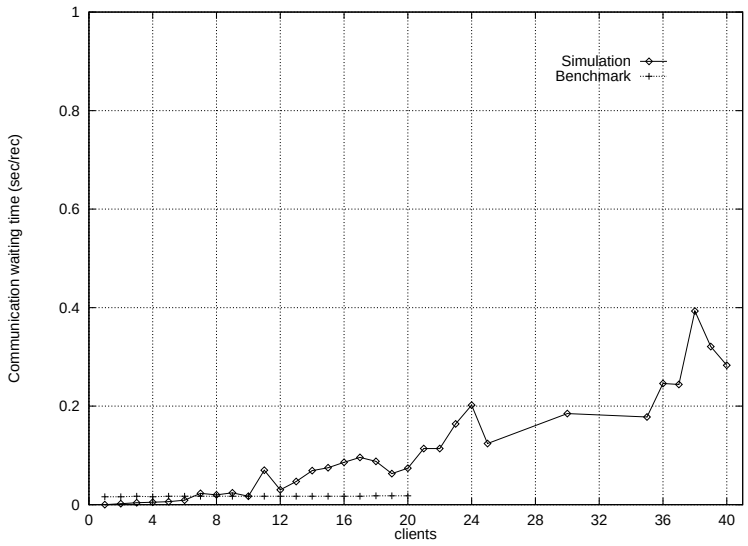


Fig. 11. Measured and simulated communication times for execution oriented simulation.

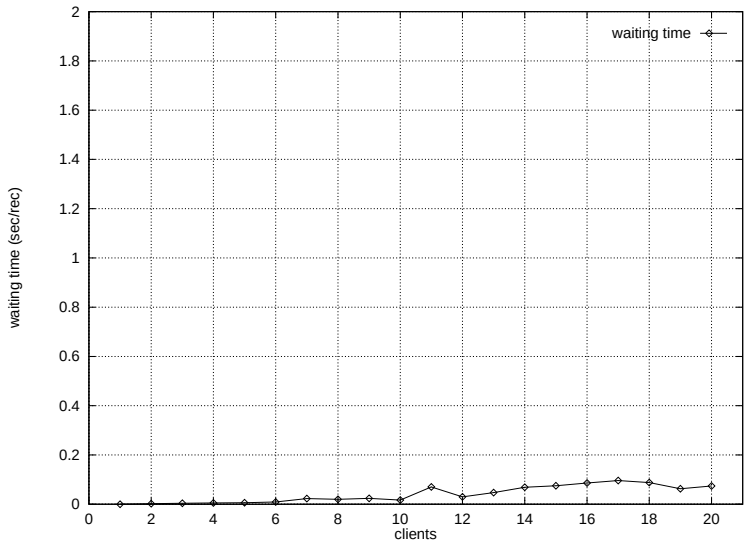


Fig. 12. Simulated communication times for communication oriented simulation.

such as clock speed, communication channels transfer rates and some informations about number of running processes and system's load. As one sees, all of these data is easily available.

The results achieved during the accuracy tests were very interesting. The time spent on processing could be predicted with error rates under 10%. This rate was higher for communication delays, where the errors were around 25%, leading to an overall rate of about 15%. These are reasonably accurate results since the simulator was not tuned to achieve the best predictions and the programs tested were rather small, introducing a large amount of measurement inaccuracies.

5 Conclusions

From our experiments we could draw some interesting conclusions. First, its accuracy is very high, with an upperbound in the error rate for CPU time under 10%. This precision was achieved with very little manipulation over the environment model, what is a good indicator of its usability.

We also observed that the execution graph model provides a great degree of flexibility, enabling to test a large set of variations during the simulation process (depending on the simulator capabilities, of course). This flexibility is provided by the graph itself, which does not prohibit any possible configuration prior to the simulation.

One drawback with the execution graph is that a binary code only can be understood, and therefore interpreted, if the specific library for its processor is available. At this moment there are libraries for the MIPS and Sparc processors. A third library is under construction for the Pentium family, while there are preliminary investigations for the IA-64 and Power architectures.

Looking at the future, the use of the execution graph approach seems to be promising. The tasks in this work should concentrate on the building of a larger set of processor specific libraries, as it has been done so far, and the implementation of a stronger simulator, capable of providing more analysis data, with a friendly interface. Some uncovered features include the use of the execution graph as a basis for a software verification tool, which would investigate all paths of concurrent or parallel programs in the search of breaches in the freedom of dependencies.

References

1. B. Alpern and L. Carter. The myth of scalable high performance. In *Proc. of 7th SIAM Parallel Computing Conference*, San Francisco, CA, 1995.
2. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.
3. H. Cain, B. Miller, and B. Wylie. A callgraph-based search strategy for automated performance diagnosis. In A. Bode, T. Ludwig, W. Karl, and R. Wismuller, editors, *Proc. of 6th Intl. Euro-Par Conference*. Springer Verlag, 2000.
4. F. Fagerstrom and C. Kuszmaul. The FTIO benchmark. Technical Report NAS-00-004, NASA Ames Research Center, Moffett Field, CA, 2000.
5. T. Fahringer and A. Pozgaj. P3T+: A performance estimator for distributed and parallel programs. *Journal of Scientif Programming*, 7(1), 2000.
6. M. Gandra, J. Drake, and J. Gregorio. Performance evaluation of parallel systems by using unbounded generalized stochastic petri nets. *IEEE Trans. on Software Engineering*, 18(1):55–71, 1992.
7. A. Geist et al. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, 1994.
8. E. Gelenbe and Z. Liu. Performance analysis approximations for parallel processing in multiprocessor systems. In M. B. M. Cosnard and M. Vanneschi, editors, *Parallel Processing*. Elsevier Science Pub., 1988.
9. U. Herzog. Formal description, time and performance analysis. In H. W. T. Harder and G. Zimmermann, editors, *Entwurf und Betrieb Verteilter Systeme*. Springer Verlag, 1990.
10. C. Hughes et al. Rsim: Simulating shared-memory multiprocessors with ilp processors. *IEEE Computer*, 35(2):40–49, 2002.
11. R. Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley and Sons, 1991.
12. A. Kapelnikov, R. Muntz, and M. Ercegovac. A methodology for the performance evaluation of distributed computations. In E. D. M.H. Barton and G. Reijins, editors, *Distributed Processing*, pages 465–479. Elsevier Science Pub., 1988.
13. J. Kitajima and B. Plateau. Modelling parallel program behaviour in alpes. *Information and Software Technology*, 36(7):457–464, 1994.
14. P. Magnusson et al. Simics: a full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.
15. M. Marsan, G. Balbo, and G. Conte. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. Computer Systems*, 2(2):93–122, 1984.
16. B. Miller et al. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.
17. J. Pierce and T. Mudge. Idtrace - a tracing tool for i486 simulation. Technical Report CSE-TR-203-94, University of Michigan, Ann Arbor, MI, 1994.
18. D. Reed. Experimental analysis of parallel systems: techniques and open problems. In *Proc. of the 7th Int'l Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, volume 794 of *Lecture Notes in Computer Science*, pages 25–51. Springer Verlag, 1994.
19. S. Sahni and V. Thanvantri. Parallel computing: metrics and models. *IEEE Parallel and Distributed Technology*, 4(1):43–56, 1996.
20. F. Sotz. A method for performance prediction of parallel programs. In *CONPAR 90-VAPP IV, Joint Intl. Conf. on Vector and Parallel Processing*, volume 457 of *Lecture Notes in Computer Science*, pages 98–107. Springer Verlag, 1990.
21. K. Trivedi, B. Haverkort, A. Rindos, and V. Mainkar. Techniques and tools for reliability and performance evaluation: problems and perspectives. In *Proc. of the 7th Int'l Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, volume 794 of *Lecture Notes in Computer Science*, pages 1–24. Springer Verlag, 1994.
22. R. Vemuri, R. Mandayam, and V. Meduri. Performance modeling using pdl. *IEEE Computer*, 29(4):44–53, 1996.