Using binary code to build execution graph models for performance evaluation of parallel programs

Aleardo Manacero[1,2]

Computer Science and Statistics Dept.
São Paulo State University - UNESP
S.J. do Rio Preto, Brasil

Abstract

This paper describes a novel approach to create program models, based on the generation of an execution flow graph from the binary code. This model can be used as a program model in the Herzog's Three-Step Methodology for performance evaluation. When modeling a program through this approach one overcomes instrumentation problems and can provide very good performance data if the simulation of the flow graph is performed correctly. Results achieved with a prototype of a simulator implemented using the approaches for program modeling described here are also presented.

Keywords: Performance evaluation, parallel computing, program modeling, program simulation, graph models

September 5, 2011

1 Introduction

Designing parallel programs is a rather complex task since it follows a distinct programming paradigm and the hardware involved has considerably higher costs. The hardware cost implies in the need for the implementation of highly efficient programs, demanding even better development techniques. This problem can be solved by performance analysis until optimal algorithm and problem decomposition are achieved. Unfortunately this demands the use of a

[1] Thanks to Prof. André Morelato (FEEC/Unicamp), who took part in an initial phase of this work.
[2] Email: aleardo@sjrp.unesp.br

parallel system in tasks that are not strictly related to data production. Although the hardware cost cannot be eliminated, its use for evaluation purposes can be reduced by the use of simulation and analytical techniques, postponing actual benchmarks until the final phases of optimization.

The use of simulation or analytical techniques imply in less accurate results despite a lower cost of the performance evaluation task. Several approaches for analytical techniques and few simulators have been proposed. Some provide interesting results but the performance evaluation field is still oriented to benchmarking tools and techniques. The drawback of benchmarking tools is that they have a higher cost and do not avoid accuracy problems since they must modify the original code by the intrusion of instrumenting code.

There are several works aiming to improve benchmarking techniques, including less intrusive instrumentations and the reduction of the amount of time that the actual hardware is used. Other works are dedicated to simulation and analytical techniques, trying to reduce problems originated from possible modeling inaccuracies. Indeed the model used on simulation or analytical techniques is the major constraint to achieve accurate performance results. Inexact models are somewhat unavoidable since the techniques are usually based of traces achieved from small benchmarks, which are reasonably compromised by intrusive instrumentation and sampling problems.

A non-intrusive approach for modeling parallel programs is proposed in this paper. This method is composed of two phases: first, the binary code of the parallel program under analysis is converted into a directed graph, which maps all the execution paths of the program (these paths may be optimized in order to reduce the number of nodes and edges in the graph). Finally the resulting graph can be simulated later, to provide the performance data needed for the program analysis.

The advantages of this approach are the improved accuracy, provided by the rewriting of the executable code, and the lower cost of simulation, which can be carried out on conventional hardware. The proposed approach follows the Herzog's three-step methodology [7], in which the program model, the machine model and the program-machine interaction model are handled separately. Here, the parallel program model is represented by its execution graph and the machine model as well as the interaction model should be provided by the user as simulation input parameters, as illustrated in Figure 1.

In the following sections one finds initially a brief description of the research on graph-based performance analysis tools. After this description, a complete review of the execution graph approach is made, including the problems involved with the modeling of specific programming structures, such as loops and subroutines. The fourth section contains information about the implementation of this approach into a prototype, and results achieved with it. In the final section, some relevant conclusions are drawn from this work.

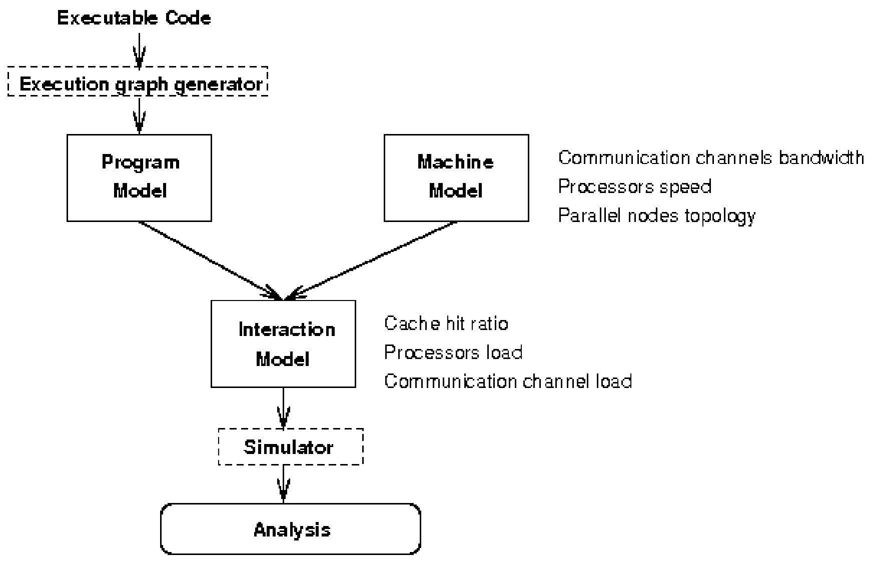

Figure 1. Herzog's methodology mapped onto the proposed technique.

2 Related work

Performance evaluation of parallel programs is a specific field inside the whole area of performance analysis and measurement. Most of the works in performance analysis are concerned with sequential systems (software and hardware). The research in this field can be classified through several taxonomies, such as by their data collection strategy or by their data analysis technique. Classical works on taxonomies are found on Jain [8], Pierce and Mudge [12] and Reed [13], who provide thorough descriptions of such taxonomies. In this work the classifications are restricted to how the different approaches for performance evaluation collect their data ([12]) and what techniques are used to analyze such data ([8]).

There are two basic forms for data collection, which are monitoring and code modification. While monitoring could provide better results, it is rarely used because it is more complex and expensive, since it demands special hardware and knowledge. Monitoring is usually applied to hardware performance evaluation, where the costs are comparably negligible and the hardware knowledge comes without extra effort. Besides that, there are tools, such as Etnus TotalView [4], which uses software monitoring to collect data.

Code modification is a more common approach. It can be performed at various levels, starting from source code modification up to executable code modification. Their accuracy depends on how much code is inserted by the instrumentation tool (code intrusion) and how the time spent on this code is taken into account at analysis time. In this category one finds all profilers and event tracers, such as SvPablo [14], Paradyn [10], Vampir [17], TAU [11], and P3T+ [3], among others.

As already stated in this paper, benchmarking is more accurate but demands larger amount of investment since it depends on the use of the actual hardware. Benchmarking techniques, such as NAS [2] will not be described here since they do not employ program models. On the other hand, techniques based on analytical and simulation approaches strongly depend of accurate models to provide reasonable results. Despite their dependance, they have a large application on performance evaluation due to their lower cost and the capability of its application at very early stages of the system development.

The formulation of good program models, even for sequential programs, is a rather complex task. There are several issues that must be addressed during the modeling procedure, such as decision points, loop iterations, memory and CPU availability among others. These inherent hazards on modeling become even more noticeable when one is working with parallel systems, mainly due to synchronization and cooperation between processes/processors. Besides these issues, a distinct problem is posed by the selection of a good metric for evaluation ([1],[8],[15]), since different users may be interested in different parameters such as communication overheads, system speedup or throughput.

Despite the hazards in modeling, there are several proposals of modeling techniques applied to software systems. Although distinct in their details, most of them have similar characteristics. One common feature among most of the models used in analytical or simulation techniques is their probabilistic behavior. Although probabilistic models are harder to manipulate, they mimic the system more realistically. Deterministic models are usually avoided because the program execution on computers present a large degree of unexpected reactions (differences in data, system speeds, and system load).

Another characteristic is the use of graph-based approaches to model the program and, sometimes, the entire system. Since this is also true for the approach presented here, the remaining of this section will concentrate on previous graph-based techniques, such as Petri nets, DAGs and their derivations.

Tools based on Petri nets include stochastic models such as the GSPN from Gandra et all [5], which is used also by Marsan et all [9] to evaluate multiprocessed systems. Petri net offers simple models but have a poor time treatment, which prevents an easy achievement of more accurate results.

Methods based on queueing theory are applied to large scale models, where each server and queue represent large portions of the system. Gelenbe and Liu [6] or Sotz [18] are classical examples of the application of queues into performance evaluation.

Other techniques based on graphs usually build them from function call lists (large-sized grains) or path profiles from program execution (small-sized grains). In all cases a common characteristic is the use of post-mortem analysis, that is, the tool builds a graph, or synthetic program, that is analyzed at a later moment, when the program is no longer running. Research into this category includes IDTrace [12], Reverse Tracer [16], or Partial Execution [19].

The approach presented here is distinct from previous works in the sense that it performs all of its actions outside the real system. Although the model generation is done using the real machine code, providing a higher level of accuracy, it can be executed on a completely different machine. The data acquisition and analysis phase needs only an execution graph simulator, which also may be executed elsewhere. The use of conventional machines to perform the modeling and simulation provide a very attractive strategy to perform a parallel program tuning, and should avoid the need for large investments during software development (even if one considers the availability of low-end low-cost clusters).

3 The execution graph modeling methodology

As previously stated, this method models parallel programs as directed graphs built from the executable code. Figure 1 shows how a performance analysis tool should map an execution graph into the Herzog's 3-step methodology. From this figure, one sees that the tool may be implemented through two separate subsystems, an *execution graph generator* and a *graph simulator*. With this approach the performance data may be collected if one has the executable code for the program and general knowledge from the physical and logical environment, avoiding the need for the actual hardware.

This framework also avoids intrusive instrumentation, providing a closer correlation between the actual code and its model. In the next paragraphs we provide a few basic model definitions, followed by the description of techniques to achieve such results.

3.1 The execution graph

In the execution graph, vertices represent sets of assembly instructions that must be executed strictly in sequential order, without any intermediary break/branch instruction. An illustrative graph is seen in Figure 2. There, the edges 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.

The vertices are classified accordingly to the purpose of the assembly instructions clustered into them and by the number and kind of edges on them. The number and kind of edges in a vertex describe the existence of loops, decisions and exit points in the program. There are six basic types of vertices

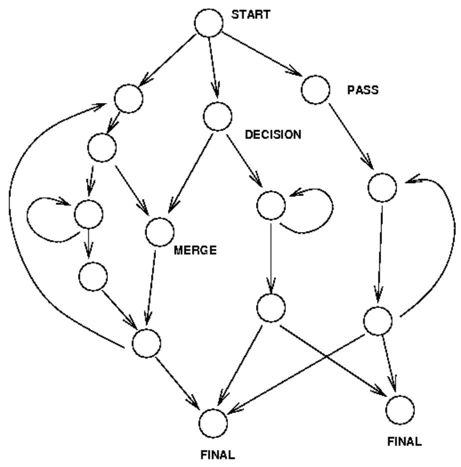

Figure 2. Execution graph of a hypothetical program.

following the number/kind scheme:

- **START**: characterized by a vertex with no edges incident to it.
- **PASS**: a vertex that has only one edge incident to and only one edge incident from it.
- **DECISION**: a vertex that has more than one edge incident from it.
- **MERGE**: a vertex with more than one edge incident to it.
- **FINAL**: a vertex that has no edges incident from it.
- **CALL**: a vertex that is a PASS vertex with the incident from edge pointing out to a vertex outside the current graph scope.

There is only one restriction in vertices compositions, which is that any execution graph have one and only one START vertex since it maps the starting point of the program. Besides that, the basic types can be merged to create a more complex vertex, such as a MERGE-DECISION vertex. The constraints that guide the merge procedure are related to the action performed by each original vertex in the merge, that is, the purpose of the clustered assembly instructions. In this classification scheme there are three basic actions (synchronization, execution and communication) that should be considered as categories, as described below:

- **Synchronization**: denotes points where some parallel tasks should synchronize, as in synchronism barriers.
- **Communication**: denotes points where some parallel tasks communicate to each other. The communication can be synchronous or asynchronous.
- **Execution**: denotes any point that is neither a synchronization nor a communication point.

With vertices categorized following these schemes it is feasible to build a graph that maps all interactions between parallel programs in an accurate form. All the interprocess interactions are handled by control points defined by *Communication* and *Synchronization* vertices. The meaning of each vertex is defined during the graph extraction process from the binary code, 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 reads the executable code for a parallel program and creates 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 phases:

(i) code reading, which reads the executable code and disassembles it;

(ii) instruction interpretation, that identifies the functional meaning of each machine instruction and maps it to actions that should be simulated afterwards;

(iii) instruction clustering, which merges consecutive instructions with similar functional meaning into a single block (a future vertex).

The first two phases are quite simple and should be performed instruction by instruction, in an orderly way. As a natural consequence of this method, these phases strongly depend of the processor that is used in the machine that will host the program under analysis. This constraint reduces the tool's portability, although it can be solved with careful software engineering and implementation of processor-specific libraries.

After each instruction interpretation, it must be decided if the amount of time (clock cycles) that that specific instruction takes at runtime should be accounted for in the current block (a graph vertex) or should go to a different, possibly new, block. At this point, the graph would start to take shape, with finished blocks becoming vertices and the links between them becoming the edges in the directed graph.

The graph edges map the control flows that the computer may follows during the program execution. The edges are created as block entry- and exit-points at the moment that the instruction interpretation finds an instruction that causes any kind of program deviation. The deviation identification is easily performed when a exit-point is found. This creates one or two outgoing edges from the current vertex, which can be directed to a new vertex or an existing one, depending on what are the destination addresses. On the other side of the directed edge, the problem is to determine if the destination address is already inside of any previously built block. This determination is solved by the functional meaning of each instruction, such as *jumps*, *calls* and *exits*. The possible cases are:

(i) **Forward jump -** meaning that an instruction has, possibly, a choice between two addresses that are inside the current function scope and ahead of the current address. These addresses usually are the next instruction address and a branch address. When such instruction is reached the current block is marked as a DECISION vertex and the system creates (if necessary) two new blocks, whose starting addresses are the addresses given by the current instruction.

(ii) **Backward jump -** when the instruction has the choice between addresses that are inside the current function scope and one of them is previous to the current address. This means that the instruction is performing a loop (except in rare, manually crafted situations) and its addresses include the backward one and the address of the next instruction. This also marks the block as a DECISION vertex and creates (if necessary) a single forward vertex and may cause a split of a previously created vertex.

(iii) **Subroutine call -** when the destination address is outside the current scope and is the consequence of any kind of call instruction. This marks the vertex as a CALL vertex, and adds a "return from subroutine" manipulation that will be explained in the next subsection.

(iv) **Exit point -** when the destination address is outside the current scope and points out to a scope that is still under analysis, characterizing a return instruction. The current vertex is marked as a FINAL vertex and receives some subroutine call manipulation as the previous case.

An extra issue appears in the management of edges that are incident to previously created vertices, that may occur in any of the situations just described. For exit and call points the treatment is performed by subroutine call modeling. For jumps the treatment involves, possibly, the breakdown of a constructed vertex into two new vertices if the destination address is internal to the vertex. The techniques to solve these problems use features of the graph structure, that are described in the next paragraphs.

3.2.1 Loop modeling

Once a loop is identified by the existence of a backward branch, it is necessary to identify where the conditional instruction that actually controls the loop is. Structured loops exist in three different flavors after compilation: *while with conditional branch*, *while with unconditional branch* and *repeat-until* forms. They have different strategies for assembling the loop control, implying different sets of conditions to model the loop.

In order to model such loops one has to observe that repeat-until loops never come with a branch instruction at its entry point. Therefore, if a loop is detected one has to inspect all DECISION vertices inside loop's body. If all DECISION vertices point to addresses that are internal to the loop, one has identified a repeat-until structure. On the other hand, if there is at least one DECISION vertex whose destination address is outside the loop's body, two possible conditions maybe present: the loop is non-structured, or it is an *while* branch.

While non-structured loops can be modeled as repeat-until loops, that is, the final DECISION vertex is the vertex where the system will perform the loops's control, *while* loops are more complex to manage. The first action is to define what is the type of branch present at the end of the loop. If that instruction is an unconditional branch, the first branch at the beginning of the loop is tagged as the control vertex and no other changes have to be made to the graph. Otherwise, with conditional branch at the end, the first branch is also tagged as the control vertex, but the final vertex is modified to eliminate the branch to the next instruction.

Finally, the last problem related with loop modeling is the determination of the backward vertex, that is, the vertex that defines the entry-point of every loop. The backward address given by the last instruction in the loop may not be, exactly, the initial address of the vertex in the entry-point. This means that the vertex containing such address must be split into two new vertices, one that gets executed before the loop and another that becomes the actual entry-point.

3.2.2 Subroutine modeling

Subroutines pose a different modeling problem, which is related to the program's capability of calling them from distinct points. Since the subgraph modeling the subroutine must appear only once, independently of how many times it is called from distinct points in the program, it becomes difficult to manage the return point (or points) from the subroutine, which has to point to different addresses, one for each call site. This problem has an even greater impact, due to obvious reasons, when modeling recursive subroutines.

The solution for this problem is to postpone the resolution of the return address to the simulation time. This is feasible if the return instructions are modeled as FINAL vertices and a stack of calling addresses. The distinction between subroutine returns and program termination is performed by the status of the calling address stack, which will be empty only when the simulated program must terminate.

Figure 3 sketches this scheme for a single call. There, the dashed edges in the graph represent the subroutine call and return. The call edge actually

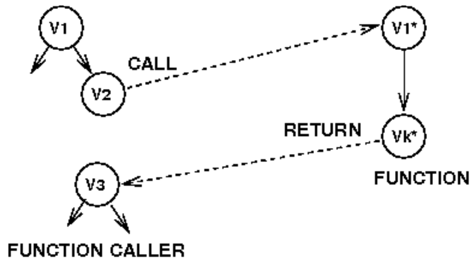

Figure 3. Subroutine calls and returns

exists in the graph, while the return edge is virtual (return address stack), becoming "physical" only during the simulation process. Therefore, if a subroutine is called from two different vertices, there will be two call edges but the return edges are created only if they become necessary to proceed the computation.

This strategy solves also the management of recursive subroutines, since the recursion will be defined by three vertices: one FINAL vertex, indicating that the recursion terminated, one CALL vertex, indicating that recursion must proceed one more level, and a DECISION vertex, which decides which of these two vertices should be executed. Therefore, the definition of how many times a recursion occurs is postponed to simulation time (the DECISION vertex), as well as the addresses mapping for a subroutine that must return to itself (the calling address stack). Since the construction of recursion (the three vertices) is inherent to the assembly instructions in the subroutine, there is no need to actually identify a recursive function as such. This, at last, means that both recursive and non-recursive subroutines can be treated the same way.

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. However, the effective number of vertices and edges is usually very high (tens of thousands of vertices for a complex program). In order to bring this number to a more manageable limit one has to perform optimizations over the graph. Such optimizations consist of vertices eliminations and associations. The optimization degree depends of the accuracy that the graph must keep after this operation, and the speed that is expected during the simulations.

The vertex elimination or association can be performed in a variety of forms. In the standard form, optimizations are executed only if the resulting graph would provide exactly the same simulated times as the original one. In this case the optimization is named non-degenerative, in contrast with degenerative optimizations that occur when small differences in the time evaluation are introduced.

The main non-degenerative optimizations are:

(i) Elimination of PASS vertices, where a PASS vertex is incorporated into its precedent neighbor vertex. Considering $\mathcal{V}1$ a PASS vertex with *t1* cycles that is preceded by $\mathcal{V}0$ (with *t0* cycles) and succeeded by $\mathcal{V}2$, this elimination makes $\mathcal{V}2$ the new sucessor of $\mathcal{V}0$, which now will be (*t0* + *t1*) cycles long.

(ii) Elimination of MERGE vertices, where this vertex is incorporated by its successor, except when the successor is also a MERGE vertex;

(iii) Branch elimination in DECISION vertices, aimed to reduce the number of paths that leave a DECISION vertex, through the hoisting of the paths with minimal execution times.

Degenerative optimizations can be performed through the elimination of certain conservative constraints. Which constraints are eliminated depends upon the definition of how accurate must the resulting model be. This, usually, means that a given reduction is performed if the resulting vertex has a number of cycles that is within the original ranges of every eliminated path.

4 Implementation of a graph generator

The methodology described in this paper was applied to build a generator that is capable of understanding codes from MIPS and Sparc processors. The initial choice of these processors was made due to their simple architecture (RISC) and instruction sets. A set of libraries was written in order to uniquely identify and evaluate each instruction in the instruction set of these processors. Currently, a third library aiming the Intel family of processors is under work. This new library could be used for the analysis of a large amount of commercial cluster systems currently under production. This prototype is named **Grasptool** (**Gra**ph **S**imulator **P**erformance prediction **Tool**).

The graph generator clusters individual instructions into several subgraphs that will make up a large forest. The forest-like structure of this graph is a direct consequence of the technique used to model subroutines, since each subroutine is converted to a graph as if it were an atomic piece, that is linked to a partially connected sub-graph (caller and called functions). The connections between sub-graphs are formed by the call edge and the respective return instructions.

The output of the generation phase is a file containing data about each sub-graph, appearing in the order of their logical addresses. The format of this data is shown on Figure 4, which depicts selected lines of a given graph file. The beginning of the graph file is composed by the list of all sub-graphs, given as pairs of their initial addresses and function names. Every entry on this list has a corresponding sub-graph, in the order of their list's appearance,

```
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 5 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 ...
```

Figure 4. Section of execution graph's output file.

where each sub-graph is preceded by a *"Tree Number N"* line. Every vertex in the graph is denoted by a single line on that file. Each vertex line starts with an integer defining the type of that vertex (execution, call, return, branches, communication, etc.). This integer is followed by a string (the logical address of its first instruction), an integer (number of clock cycles in its execution), addresses of candidate vertices to proceed the execution and other data when necessary.

5 Results

Results presented here were collected with a simple prototype which was built following this approach. Although this prototype provided only coarse grained data about the evaluated programs, it is possible to draw interesting conclusions about the technique.

5.1 Timing tests

Timing tests were performed in order to verify what level of accuracy could be achieved at the simulation phase. The tests were performed through the comparison between the execution times of a benchmark program and the predicted execution times. The benchmark program consisted of a set of three

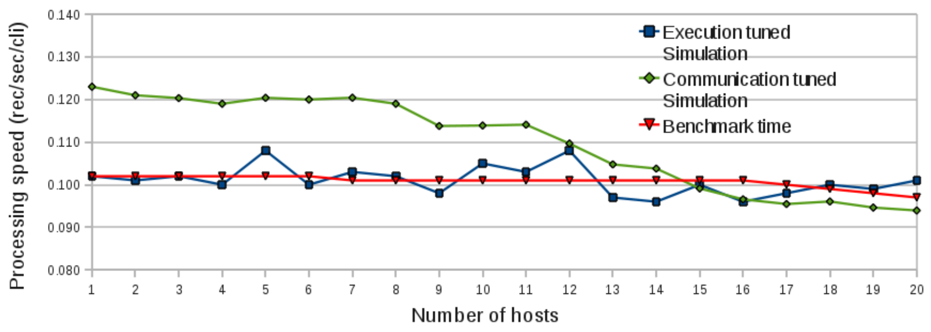

Figure 5. Measured and simulated execution speeds.

large interacting programs used in physics of particles. This set of programs is coded in Fortran and run over small clusters of workstations. Each cluster has 12 to 20 hosts and the program follows the bag-of-tasks model, with several *client* processes and a pair of *reader-writer* processes acting as masters. Measurement of smaller programs was not conducted since even major errors in the number of cycles would represent only few microseconds in time and these programs are not usual components of high performance systems.

Although the benchmarks were limited by the cluster's size and by the instrumentation that was already provided inside the code, a scalability test was executed by simulation. As described earlier, the size limitations are not present in the simulator, enabling the execution of other tests, such as the system's scalability and even measurements of metrics that were not part of the original production system. Another feature is that the environment and machine models for the simulator are provided by input parameters, enabling the simulator's tuning by two distinct approaches: communication oriented or execution oriented. Actually, the tuning can be performed simply by changing the expected delay value for communication vertexes, in order to introduce higher or lower communication costs into the system.

The plots in Figures 5, 6, and 7, show the results achieved with both the simulator's prototype and the benchmarks. Each plot is the average of at least 50 simulations for each point, with each simulation taking about 30 seconds in a personal computer. Each benchmark run took about 20 minutes, and the plots give the average of 10 runs for each point.

Figure 5 shows the execution speed, measured as the number of data records processed per second (each record contained about 64kbytes of data, and consisted of about 10 to 12 atomic particles). As one can see from the figure 5 (note that the y axis is scaled to improve visibilty), when the simulator is execution oriented, the measured speeds have a higher correlation to the benchmark values. A good correlation is also achieved with the communication oriented tuning, although in this case the deviation from the actual values is higher, and the error rate goes up to 20%. Even this result is quite

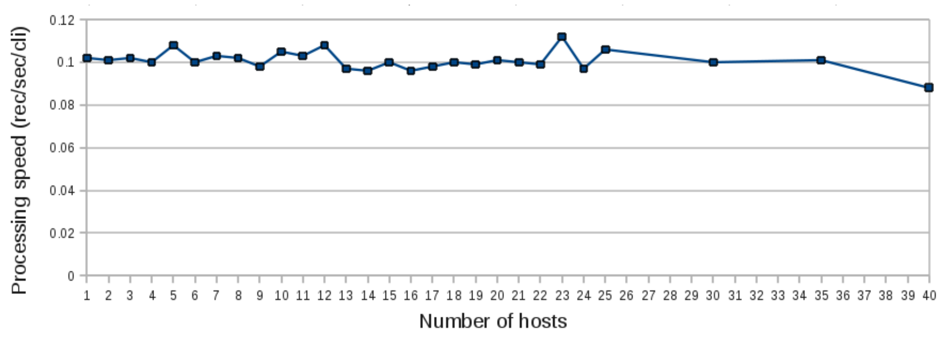

Figure 6. Simulated execution times for a larger cluster.

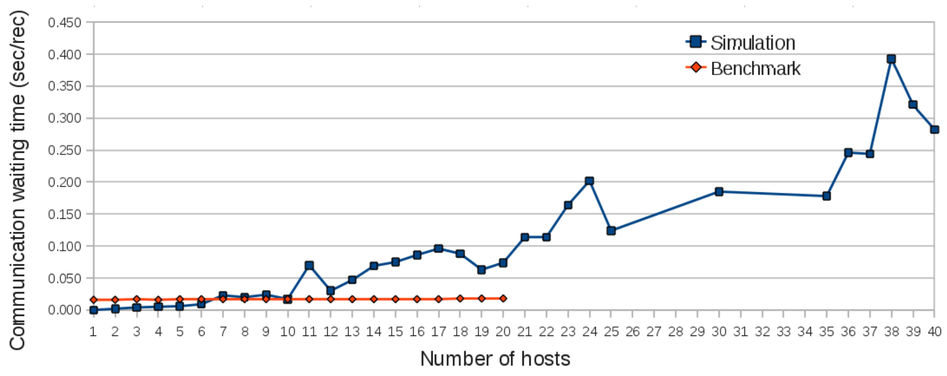

Figure 7. Measured and simulated communication times for execution oriented simulation.

reasonable since the proposed approach is aimed only at performance prediction.

As already stated, the simulation could be performed for a larger number of nodes, since it is not physically restricted by the actual cluster. Figure 6 shows the simulated processing speeds for a larger cluster. From this plot the analyst can estimate speedup curves and, therefore, scalability limitations over the program and the parallel machine where it should run. From this experiment one can realize that clusters larger than 30 machines are not efficient for this kind of system.

In order to understand the difference between the two opposite tunings, the influence of communication delays was evaluated. Figure 7 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 actual ones. This happens because the execution oriented tuning does not take into account the fact that although the system needs more messages between nodes, these messages occur in an interleaved fashion. If this condition is observed during the simulation the results improve remarkably.

The balance on communication costs implies in a compromise that should be easily achievable in a complete version of Grasptool through a set of tuning parameters. In spite of this the results provided by the simple prototype are very promising since the time spent on processing could be predicted with a margin of error under 10%. This is very accurate since the simulator was not optimally tuned and previous published works about prediction tools indicate that errors under the 40% range are admissible.

6 Conclusions

The preliminary tests provided interesting conclusions about the proposed methodology. First, its accuracy is very high, with an upper bound in the error rate for CPU time under 20%, even for the worst tuning attempted. This accuracy was achieved with very little manipulation over the environment model, which is a good indicator of its usability.

It is also noticeable 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 disable any possible configuration prior to the simulation. Although it was not shown here, it is a very simple task to define, for example, a simulator capable of measuring time spent on program's functions and even inner loop blocks.

One drawback with the execution graph is that a binary code can be understood, and therefore interpreted, only if the specific library for its processor is available. Nowadays there are libraries for the MIPS and Sparc processors. A third library is under construction for the x86 family, and there are preliminary investigations for the IA-64 and Power architectures.

To conclude this discussion, the use of the execution graph reconstruction approach is promising. Future work should concentrate on the building of a larger set of processor specific libraries, as has been done so far, and the implementation of a powerful simulator, capable of providing more analysis data, with a friendly interface. On a different front, a probable extension would 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 for deadlocks, starvation and missed mutual exclusions.

References

[1] Alpern, B. and L. Carter, *The myth of scalable high performance*, in: *Proc. of 7th SIAM Parallel Computing Conference*, San Francisco, CA, 1995.

[2] Bailey, D. H., E. Barzcz, L. Dagum and H. D. Simon, *Nas parallel benchmark results*, IEEE Parallel Distrib. Technol. **1** (1993), pp. 43–51.

[3] et alii, T. F., *Evaluation of p3t+: A performance estimator for distributed and parallel applications*, in: *Proc. Of 14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, Cancun, 2000, pp. 229–234.

[4] Etnus(2004), "Totalview 6.6 multiprocess debugger," http://www.etnus.com, Natick, MA, EUA, 2004.

[5] Gandra, M., J. Drake and J. Gregorio, *Performance evaluation of parallel systems by using unbounded generalized stochastic petri nets*, IEEE Trans. on Software Engineering **18** (1992), pp. 55–71.

[6] Gelenbe, E. and Z. Liu, *Performance analysis approximations for parallel processing in multiprocessor systems*, in: M. B. M. Cosnard and M. Vanneschi, editors, *Parallel Processing* (1988).

[7] Herzog, U., *Formal description, time and performance analysis*, in: H. W. T. Harder and G. Zimmermann, editors, *Entwurf und Betrieb Verteilter Systeme* (1990).

[8] Jain, R., "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling," John Wiley & Sons, 1991, 2nd edition edition.

[9] Marsan, M., G. Balbo and G. Conte, *A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems*, ACM Trans. Computer Systems **2** (1984), pp. 93–122.

[10] Miller, B., *The paradyn parallel performance measurement tool*, IEEE Computer **28** (1995), pp. 37–46.

[11] Morris, A., A. D. Malony and S. Shende, *Supporting nested openmp parallelism in the tau performance system*, International Journal of Parallel Programming **35** (2007), pp. 417–436.

[12] Pierce, J. and T. N. Mudge, *Idtrace - a tracing tool for i486 simulation*, in: *MASCOTS '94: Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems* (1994), pp. 419–420.

[13] Reed, D. A., *Experimental analysis of parallel systems: techniques and open problems*, in: *Proceedings of the 7th international conference on Computer performance evaluation: modelling techniques and tools* (1994), pp. 25–51.

[14] Rose, L. D., Y. Zhang and D. Reed, *Svpablo: A multi-language performance analysis system*, in: *Computer Performance Evaluation (Tools)*, 1998, pp. 352–355.

[15] Sahni, S. and V. Thanvantri, *Performance metrics: keeping the focus on runtime*, IEEE Parallel and Distributed Technology **4** (1996), pp. 43–56.

[16] Sakamoto, M., L. Brisson, A. Katsuno, A. Inoue and Y. Kimura, *Reverse tracer: A software tool for generating realistic performance test programs*, in: *HPCA*, 2002, pp. 81–91.

[17] Solutions, C., "Vampir/Vampirtrace," http://www.cs-software.com/software/vampir/vampir.html, Ladson, SC, EUA, 2005.

[18] Sotz, F., *A method for performance prediction of parallel programs*, in: *CONPAR 90-VAPP IV, Joint Intl. Conf. on Vector and Parallel Processing*, Lecture Notes in Computer Science **457** (1990), pp. 98–107.

[19] Yang, L., X. Ma and F. Mueller, *Cross-platform performance prediction of parallel applications using partial execution*, in: *SC*, 2005, p. 40.