Automated Experimental Parallel Performance Analysis

Jan LEMEIRE
And
Erik Dirkx
PADX, VUB
Brussels, Belgium
Email: jlemeire@info.vub.ac.be, erik@info.vub.ac.be

Extended Abstract for 2nd PACT Workshop, Edegem, Belgium, September, 2002

ABSTRACT

Performance is the key issue in parallel processing. We want to investigate how far we can automate experimental performance analysis in order to achieve all necessary performance results for performance prediction, load balancing and algorithm optimisation. This paper describes the approach of generalising the performance analysis and obtaining the specific results by experiments.

Keywords: Parallel processing, performance analysis, speedup, load balancing, performance prediction.

1. INTRODUCTION

The current research of the Parallel Systems lab[1] of the Vrije Universiteit Brussel (VUB) focusses on generalised performance analysis of parallel applications.
Parallel processing is running a (sequential) program on multiple processors to get the job done in less time.

The only goal is to have a performance gain, what we call the speedup. The main problem is that parallelisation and its performance is problem and system dependent, in a complex way. Furthermore, it cannot yet be automated and speedup is not guaranteed. However, for succes, the design of efficient parallel programs should be straight forward.
Our current research investigates the possibility to fully automate experimental parallel performance analysis, which will offer insight of performance in function of all parameters and can be used for load balancing. So we want to extract the general rules that guides the specific, instantiated analysis and this is after all, the goal of all science.
Note that our lab focusses on the message passing paradigm, used on a cluster of workstations.

In the next section we describe briefly the (scientific) problems involved with parallel processing.

2. PARALLEL PROCESSING

Figure 1 views the process of parallelisation. A sequential algorithm is parallelised by partitioning it and adding synchronisation. It is run on the parallel system, resulting in a certain speedup, which is analysed in order to predict the performance, to detect speedup bottlenecks for effective optimisation and for efficient load balancing.

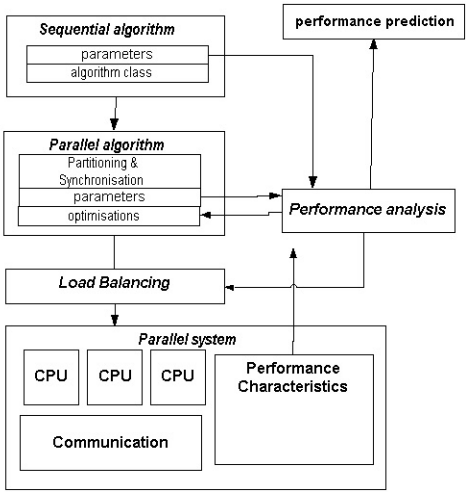

Figure 1: parallel processing

The major tasks are the parallelisation, the load balancing [Gupta 99, Zaki 95] and the performance analysis (figure 1). There is however a difference between *embarassingly parallel* problems and non-trivial parallel algorithms. In the first category load balancing is the main difficulty. The performance analysis is reduced to the communication – computation ratio and therefore easy to compute. The only benefit of our approach would be the automatic analysis of the system dependency of the performance (as we will explain later).
For the second category, particular parallel solutions are necessary, resulting in specific performance bottlenecks [Kumar 94]. We will have to proof that our general approach can reveal these 'hot spots'. Here, the performance analysis will also serve the load balancing.

The next section outlines how a performance analysis, that covers *all* aspects, can be performed.

3. STANDARD PERFORMANCE ANALYSIS

The time diagram of figure 2 shows the timeline of a typical parallel program. The sequential work is divided among the processors of the parallel system. Besides this usefull computation, parallel processing requires partitioning synchronisation,

[1]check http://parallel.vub.ac.be for all information

communication, and it generates blocking on the processors.

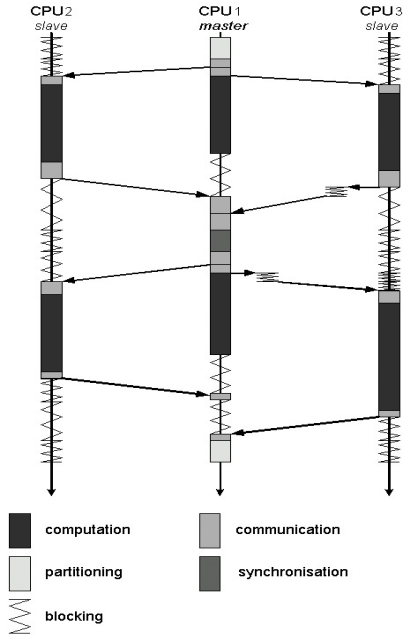

Figure 2: parallel algorithm

These 4 types of *overhead* slow down the performance, as expressed in Eq. 1 [Kumar 94b, Lemeire 2001]:

$$Speedup = \frac{p}{1 + \frac{overhead}{seqTime}} \quad (1)$$

with p the number of processors and *SeqTime* the runtime of the sequential algorithm. The total overhead is the sum of all **overhead terms**, which can be measured experimentally:

$$overhead = \sum_i OverheadTerm_i \quad (2)$$

The impact of the overhead on the speedup can be expressed by the **slowdown term**:

$$SlowdownTerm_i = \frac{OverheadTerm_i}{SeqTime} \quad (3)$$

Once all of the overhead terms are calculated, it can be analysed which terms are the major bottlenecks and which can be neglected.
Blocking overhead is mainly an effect of bad partitioning resulting in by load imbalances, but this is not the only reason. In most cases, partioning happens on the master processor, causing blocking on the slave processors, resulting in a O(p) dependency of the partioning overhead. This can also be the case for synchronisation and communication, when it blocks other processors (see figure 2). In our opinion, these effects are easily overlooked. We will thus have to develop an algorithm which determines the reasons of blocking.
Next, performance should be expressed in funtion of all algorithm and system parameters. The number of processors (p) and the problem size (W) are the most general, but each algorithm or system adds specific parameters. For automatic analysis, dependencies can be investigated experimentally by varying the parameters, and the resulting experimental data could be transformed into analytic equations between the data.
As stated before, the system-dependency is crucial. Here we want to introduce *first-order performance factors* that approximate these dependencies. For example, communication blocking can be caused by network congestion. For this overhead, there is an influence of system and algorithm which can be expressed by 2 sensitivity factors. One that represents the chance of network congestion of the system and one that represents the chance that this causes blocking during the parallel processing.

The detailed analysis will be explained in the next paper. In the next section we explain the expected benefits of our approach.

4. BENEFITS

The performance analysis results in S=f(par), the *speedup* in function of all parameters. This knowledge is usefull in several ways. First, for speedup prediction of a certain parameter configuration, for scalability analysis (S versus increasing p and W) [Kumar 94b], for cost-S tradeoff [Kumar 94], for calculation of the optimal S configuration, etc.
Secondly, the function S(par) is necessary for the load balancing algorithm. Here, the influence of the system on the speedup is essential.
Next, reducing the speedup analysis to the detailed analysis of the overhead terms, makes *understanding* of the performance results possible and optimisation of the parallel algorithm. Certainly in the cases when the overhead is non-trivial, for example when the blocking is caused by communication. This is more difficult to predict, making an experimental analysis indispensable.
Finally, a standard parallel performance analysis, makes exchange of results easier and more usefull.

Let's now look how this approach results in an application.

5. APPLICATION

The application for parallel performance analysis consists of 5 parts, as shown in figure 3.

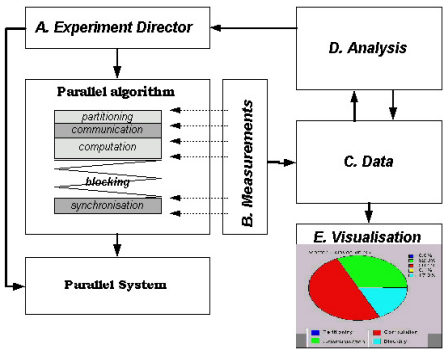

Figure 3: application

(A) The **Experiment Director** decides what experiments are necessary and configures the parameters of system and algorithm.
(B) The **Measurements** part adds chronometers to the code of the parallel algorithm that will time all phases (partitioning, communication, computation, synchronisation and blocking). Simultaneously, the values of algorithm variables are passed, like the communication datasize and the number of performed iterations. Here algorithm-specific parameters can be added. This part is integrated with pvm or mpi, the standard interface for message passing.
(C) All **Data** is stored in a relational database.
(D) Once an experiment finishes, the **Analysis** part first calculates the overhead terms, the speedup, etc. Then the relation with the different parameters is investigated and more experiments can be ordered.
(E) Finally, the **Visualisation** part shows the results in different layers, where each *layer* represents an aspect of the analysis:
    1) The *time layer* shows all variables in function of the algorithm runtime of one experiment (cf the xpvm graphical support of pvm).
    2) The *processor layer* shows all totalised values per processor.
    3) The *experiment layer* shows all total values of an experiment and the conclusions about speedup and bottlenecks.
    4) The *parameter layer* shows all values in function of the system and algorithm parameters.

Additional interesting features are the possiblity for the user to input equations between the parameters, that can then be compared with the experimental results. Furthermore, the possibility to perform partial measurements in order to extract equations of fundamental operations, eg. perform 1 sort iteration to measure its time constant. Moreover, performance estimatation before implementation by entering the estimated overhead and get the performance analysis.
The application will be implemented on our *LINUX* operating system, in *C++*, using *pvm/mpi* for parallel processing [Geist 94, MPI], the *QT* GUI library [Trolltech] and the *mSQL* database [Hughes].

6. OTHER RESEARCH TOPICS

Parallel performance analysis is the topic of the PhD research of Jan Lemeire. Besides this, the lab investigates within the scope of parallel processing:

- The construction of parallel algorithms for *algorithm classes*, that separate algorithm and parallelisation aspects as much as possible by using modern software engineering techniques.
- Visualisation of parallel processing, integrated in the performance application.
- Parallel discrete event simulation [Brissinck 99], for which the above performance analysis is also valid [Lemeire 2001].
- Detection of symmetry properties of problems for partitioning and visualisation.

7. CONCLUSIONS

The goal of our research is to facilitate parallel processing of algorithms by automating the performance analysis. We try to proof that a general, standard analysis can serve all desired results.
What are the difficulties we have to overcome? First, it is not yet clear if we can get the same detailed results of an instantiated algorithm-specific analysis. So, we will try to find again known results, like the detailed performance discussions described in [Kumar].
What are the expected limitations? Clearly, a lot of experiments will be necessary. This could be overcome by using partial experiments or perform experiments only when necessary. Second, in certain cases first-order approximation for the hardware dependencies will fail. Higher order analysis will become necessary.

This research track looks interesting, especially because we believe it involves more general scientific problems.

8. REFERENCES

Wouter Brissinck, “Tuneable Granularity Parallel Discrete Simulation”, PhD, Vrije Universiteit Brussel (VUB), Brussels, 1999.

A. Geist, A. Beguelin et al., “PVM: Parallel Virtual Machine”, the MIT press, 1994.
www.netlib.org/pvm3/book/pvm-book.html
www.epm.ornl.gov/pvm/ (pvm homepage)

D. Gupta and P. Bepari, "Load sharing in distributed systems", In Proceedings of the National Workshop on Distributed Computing, January 1999.

Hughes Technologies, mSQL database, http://www.Hughes.com.au/.

Kumar V., Grama A., Gupta A. and Karypsis G. Introduction to Parallel Computing. Design and Analysis of Algorithms. Benjamin CUmmings, California, 1994.

Vipin Kumar and Anshul Gupta, “Analyzing Scalability of Parallel Algorithms and Architectures”, Journal of Parallel and Distributed Computing (special issue on scalability), Volume 22, Number 3, September 1994, pp. 379-391. 1994b.

J. Lemeire and E. Dirkx, “Performance Factors in Parallel Discrete Event Simulation”, in Proceedings of the 15th European Simulation Multiconference (ESM 2001), SCS, Delft, 2001.

MPI homepage, http://www-unix.mcs.anl.gov/mpi/.

Trolltech AS, QT, the cross-platform C++ GUI toolkit, http://www.trolltech.com.

M.J. Zaki, Wei Li; S. Parthasarathy, “Customized dynamic load balancing for a network of workstations”, Proceedings of the High Performance Distributed Computing (HPDC'96), IEEE, 1996.