A Refinement Strategy for a User-Oriented Performance Analysis.

Jan Lemeire, Erik Dirkx

Parallel Systems lab, Vrije Universiteit Brussel,
Pleinlaan 2, 1000 Brussels, Belgium
{jlemeire, erik}@info.vub.ac.be

Submitted for Euro-Pvm 2004, Budapest, Hungary

Abstract. We introduce a refinement strategy to bring the parallel performance analysis closer to the user. The analysis starts with a simple high-level performance model. It is based on first-order approximations, in terms of the logical constituents of the parallel program and characteristics of the system. This model is then gradually refined with more detailed low-level performance aspects, to explain divergences from a 'normal', linear regime. We use a causal model to structure the relations between all variables involved. The approach intends to serve as a link between detailed performance data and the developer. It is demonstrated with a parallel matrix multiplication algorithm.

1 Introduction

This paper investigates how the non-expert developer should be given clear insight in the performance of its parallel program. For efficient parallel processing, the developer must master the various performance aspects, ranging from high-level software issues to low-level hardware characteristics. The performance analysis is nowadays supported by various tools that automatically analyze parallel programs. The current challenge however is to give the software developer understandable results with a minimum of the learning overhead [APART: http://www.fz-juelich.de/apart], as the tools seem hard to sell to the user community [9].
Most profiling tools like SCALEA [14], AIMS [15], Pablo [11], KOJAK [7] or VAMPIR [8] (semi-) automatically instrument the parallel program and use hardware-profiling to measure very detailed performance data. In the post-mortem analysis, they automatically filter out relevant parts, like bottlenecks, situations of inefficient behavior, performance losses, from the huge amount of low-level information and try to map them onto the developer's program abstraction, like code regions. Our approach works in the opposite direction; it is based on a simple performance model, using the terminology of a non-expert developer. This high-level model is used as a first-order approximation for explaining the performance. Experimental data will then support the model, or indicate the need for a more in-depth analysis when divergences appear. In this manner, the model is extended with low-level characteristics.

We present all variables or fluents in a causal model to indicate their dependencies. The refinements gradually add extra low-level fluents to the model.

The next section explains the parallel algorithm for matrix multiplication, section 3 defines the causal performance model and section 4 our tool EPPA. Section 5 shows the experimental results and the refinement strategy.

2 Parallel Matrix Multiplication

We illustrate our approach with the analysis of parallel multiplication of matrices, $C = A \times B$. The sequential runtime to multiply two dense $n \times n$ matrices is of $O(n^3)$, what makes it worth for being computed in parallel for high values of n. The parallel algorithm uses with a checkerboard partitioning, the matrices are divided in r strips of contiguous rows and c strips of contiguous columns, where $r \times c = p$ and r close to $\sqrt{p}$ [6]. Then, blocks of size $n/r \times n/c$ of matrices A and B are attributed to each processor. The p processes are labeled from $p_{0,0}$ to $p_{r-1,\ c-1}$. This is the starting point of our parallel algorithm. Each process $p_{i,j}$ will compute submatrix $C_{i,j}$ of the result matrix. Therefore, it requires all submatrices $A_{i,k}$ and $B_{l,j}$ for $0 \leq k < r$ and $0 \leq l < c$ (Fig. 1). The simplest way of acquiring these blocks, is an all-to-all broadcast of matrix A's blocks could be performed in each row of processes $p_{i,j}$, and an all-to-all broadcast of matrix B's blocks in each column [6, 13]. A more memory-efficient version is Cannon's algorithm. Each process first multiplies its local submatrices and then sends its submatrix $A_{i,j}$ to the 'left' and $B_{i,j}$ to its 'upper' process. The 2 submatrices it receives from its direct neighbors are then multiplied, added to the result and sent again to the next process in a rowwise (for A) and columnwise (for B) circular shift operation [6].

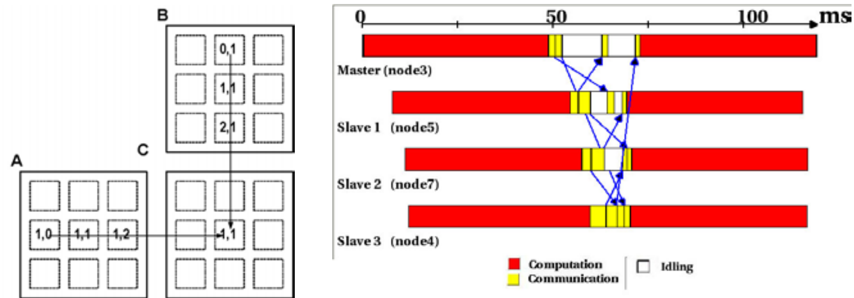

Fig. 1. Parallel Matrix Multiplication: partitioning (left, $p=9$) and execution profile (right, n=150, p=4, Cannon's algorithm)

The experiments are performed on a cluster of 9 dedicated 333MHz Pentium II processors with 256MB RAM, connected by a 100Mb/s non-blocking switch. Figure 1 shows the time view for Cannon's algorithm, showing the three types of phases: computation, communication and idling.

3 The High-Level Performance Model

The performance analysis should consider the impact of each phase, whether it is useful work or overhead, of the parallel program on the speedup. This is reflected by the ratio of the time T^i_{phase} of a phase on processor *i* with the sequential runtime T_{seq} divided by the number of processors *p*, what we call the *overhead ratio*:

$$Ovh^i_{phase} = \frac{T^i_{phase}}{T_{seq}/p}. \quad \textbf{(1)}$$

and totalized over all processors, it gives the global impact:

$$Ovh_{phase} = \frac{\sum_i^p T^i_{phase}}{T_{seq}} = \frac{T_{phase}}{T_{seq}}. \quad \textbf{(2)}$$

These definitions differ slightly from the *normalized performance indices* used by AIMS, defined as $Index_{phase} = T_{phase}/T_{par}$ [12], which are always less than one. The overhead ratios become more than one if an overhead surpasses the run time of the useful work. Our choice for this definition is motivated by the direct relation with the efficiency *E*:

$$E = \frac{1}{\sum_{phases} Ovh_{phase}}. \quad \textbf{(3)}$$

This can easily been derived from its definition $E=S/p=T_{seq}/T_{par}/p$ and writing the parallel runtime as:

$$T_{par} = T^1_{par} = T^2_{par} = \ldots = T^p_{par} = \frac{\sum_i T^i_{par}}{p}. \quad \textbf{(4)}$$

Where each processor *i* takes the parallel run time T_{par} to perform its part of the job, and is the summation over all phases: $T^i_{par} = \sum_j T^{i,j}_{phase}$. This leads to the following equation, which is equivalent to Eq. 3:

$$Speedup = \frac{T_{seq}}{T_{par}} = \frac{p}{\frac{\sum_i \sum_j T^{i,j}_{phase}}{T_{seq}}}. \quad \textbf{(3)}$$

Fig. 2 shows the overhead ratios of all phases for the parallel matrix multiplication.

Fig. 2. Overhead ratios of parallel matrix multiplication with n=150, p=4.

We identify three top-level phase types: the computation, the communication and the idle times (Fig. 2, right). This classification can easily be refined by the developer to identify logical phases in its program and to subdivide different overheads, as done by several authors, for example by Bull [2]. The communication is defined as the overhead time not overlapping with computation: the computational overhead due to the exchange of data between processes, in the sense of loss of processor cycles [3].
To structure all variables and to show their dependencies, we use a causal model. See Pearl [10] for an overview of current theory about causality and statistics. The initial performance model is shown in Fig. 3, where direct cause-effect relations are indicated by directed links.

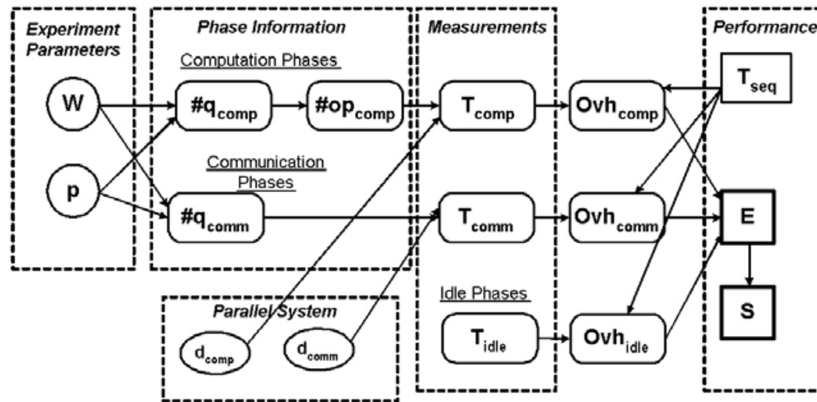

Fig. 3. Causal Performance Model of First-Order Approximation.

The goal of the model is to bring it to simple, mostly linear, relations by introducing useful intermediate variables. Instead of trying to directly estimate the functional relation between for example the speedup S or the computation time T_{comp} and the number of processors p, we pass through relevant characteristics of each phase. Each computational phase can be characterized by the number of *operations* and processed *quantums*. The number of operations $\#op$ relate directly to the processing time as the number of identical operations that are performed during that phase (Eg. the number of compare and swap operations for a sort algorithm). The number of quantums $\#q$ relate directly to the problem size W (Eg. the number of elements sorted or communicated). In this manner, complex relations can be unraveled and symbolically interpreted more easily, as will be shown in section 5.

Experimental runs are identified by the experiment parameters, like the problem size *W*, number of processors *p* and additional algorithm-specific parameters. The parallel system is initially characterized by δ_{comp} and δ_{comm}.

4 EPPA Tool Overview

EPPA (Experimental Parallel Performance Analysis) is implemented in C++ and is independent of the parallel communication layer [4]. A parallel program should be instrumented manually with one *EPPAProbe* object per process, which will collect all relevant program data. We envisage that the extra work needed for manual code instrumentation is compensated with the advantage of having all program information understood and controlled by the user. The programmer should identify the logical parts of its program, specify phase variables and the parameters of the experiment.
A method call should be inserted at the end of each *phase* of the program. These *phases* do not have to correspond with loops or function calls, as is the case for most automatic instrumentation tools, but with the functional parts of the program. Then, each phase is characterized by the number of operations *#op* and processed quantums *#q*. Different experimental runs are identified by the program parameters, which are passed with the *EPPAProbe* constructor.
At the end of program execution, the data is written to a mySQL database (www.mysql.com). The analysis of the data is written in java and performed automatically. Results are shown graphically in different *views*, each representing a different aspect of the analysis: the time (Fig. 1), overhead (Fig. 2), causal (Fig. 3 & 5) and functional views (Fig. 4 & 6).

5 Model Refinement

The relations between the variables are analyzed by finding the best predicting equations, using standard regression analysis [4, 5]. We use the LOOCV (Leave-One-Out Cross Validation) method to choose among polynomial equations of different degree. This method overcomes overfitting by testing how each individual observation can be predicted by the other observations.

5.1 Computation

As a first order approximation, we take the computational runtime as proportional to the number of identical operations:

$$T_{computation} = \partial_{comp}.\#operations\,. \tag{6}$$

Clearly, this equation is not realistic, as for example, in the presence of superscalar architectures with multiple arithmetic units and hardware pipelines. It is even not

applicable for a simple PC, when the memory usage exceeds the RAM capacity, as is the case in our experiments. However, we argue that it still is a useful relation. It can serve as a first approximation, but it should be extended when going out of the 'normal' linear regime. Figure 4 shows the experimental results for the matrix multiplication. Up to ± 54.10^9 operations (n=3800) Eq. 6 holds firmly and reveals a δ_{comp} of 0.25µs/operation (the straight line). After that point, the processors start using the swap memory and the runtime increases super linear. Eq. 6 then fails to explain the performance. These divergences should then be supported by a more refined performance model, namely measurement of the memory usage (Fig. 5). The extra runtime, caused by memory swapping T_{swap}, gives a roughly linear trend with the swap memory with δ_{swap} = 0.27ms/#q_{swap}.

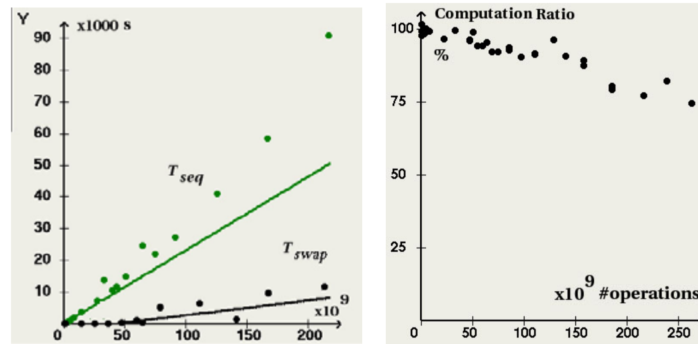

Fig. 4. Sequential run time (left) and computation ratio (right) versus number of operations (Cannon's algorithm, p=4).

The right curve of Fig. 4 shows the advantages of Cannon's algorithm. When the sequential runtime starts to saturate, the sum of the parallel computation times on all processors is less. The computation ratio drops to 75% for n > 6000 and a better than ideal speedup of 5.4 is reached.

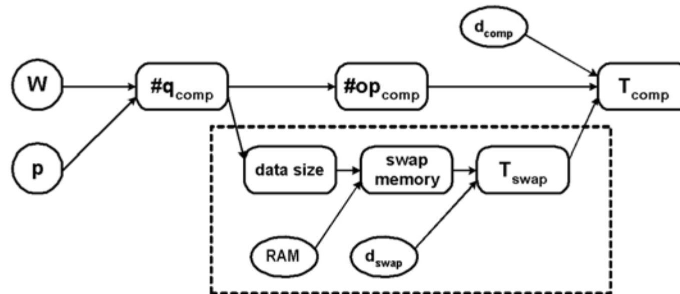

Fig. 5. Refined performance model for computation phases.

5.2 The Communication Overhead

Cannon's algorithm requires *max*(*r,c*)*-1* communication steps, where each process sends 2 submatrices of $n/r\times n/c$, containing n^2/p elements or quantums. The communication time is then

$$\sum_{j}^{p} T_{comm}^{j} \sim p.2.\frac{n^2}{p}.\max(r,c) \sim n^2.\max(r,c) \tag{7}$$

For quadratic values of p, this results in $O(n^2\sqrt{p})$ relation. Non-quadratic values of p however require partitions with different number of rows and columns, hence a higher communication. Prime numbers result in a 1 row, p column matrix partitioning with communication of $O(p)$.

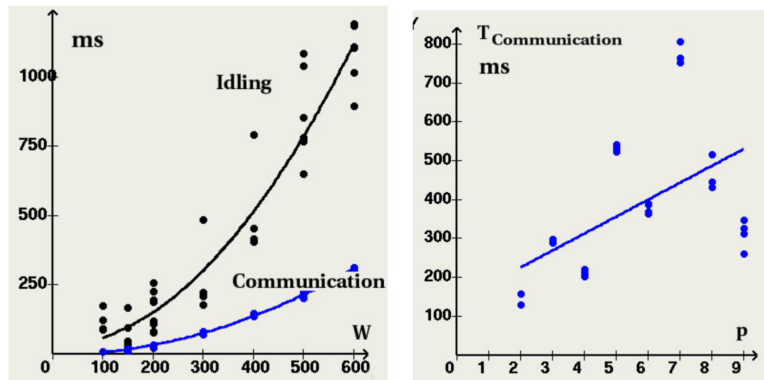

Fig. 6. Communication and idle time versus *W* (*p*=4) and *p* (*W*=500).

The experimental results (Fig. 6) for the communication confirm this theoretical expression. The idle time follows the same trend as a function of *W*. As the load imbalances are very low, the idle time is mainly caused by the message delays, which are also proportional to the number of elements being sent.

The right curve of Fig. 6 right shows communication time as a function of the number of processors *p.* It can be seen that the points are more difficult to interpret symbolically, because of the non-polynomial relation between data size and *p* (Eq. 7). This illustrates the advantage of using an intermediate variable $\#q_{comm}$ to get 2 curves that can be interpreted more easily. The curve $\#q_{comm}$ versus *p* would reveal Eq. 7, while T_{comm} versus $\#q_{comm}$ size could reveal non-linear trends, which is much more difficult for the curve of Fig. 6.

6 Conclusions

Our approach aims at making the performance analysis of parallel processing comprehensible for users. A causal model structures the relations between all relevant performance characteristics. We introduced a refinement strategy to ensure that the user is only confronted with low-level characteristics if they affect the performance. Furthermore, causal diagrams can be exploited to *reason* about the performance [10].

7 References

1. Browne, S., Dongarra, J.J., Garner, N., Ho G., and Mucci, P. A Portable Programming Interface for Performance Evaluation on Modern Processors, International Journal of High Performance Computing Applications, 14:3 (Fall 2000), pp. 189-204.
2. Bull, J.M.: A Hierarchical Classification of Overheads in Parallel Programs. In: Proceedings of First IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems, Chapman Hall, (March 1996) 208-219.
3. Crovella, M. E. and Leblanc, T.J.: Parallel Performance Prediction using Lost Cycles Analysis. In: Proc. of Supercomputing '94, IEEE Computer Society (1994).
4. Crijns, J. and Crijns, A. Automatische Experimentele Analyse van Systeem en Algoritmeparameters op Parallelle Performanties. *Thesis*, Vrije Universiteit Brussel (VUB), Brussels, 2003.
5. KEEPING, E.S., Introduction to Statistical Inference, Dover Publications Inc, New York, 1995.
6. Kumar, V., Grama, A., Gupta, A. and Karypsis, G.: Introduction to Parallel Computing. Design and Analysis of Algorithms. Benjamin Cummings, California (1994).
7. Mohr, B., and Wolf, F. KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Programs. *Euro-Par Conf.* 2003: 1301-1304.
8. Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.-C. and Solchenbach, K. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69-80, January 1996.
9. Pancake, C.M.: Applying Human Factors to the Design of Performance Tools. In: Proc. of the 5th Euro-Par Conf., Springer (1999).
10. Pearl, J. Causality. Models, Reasoning and Inference. Cambridge University Press, Cambridge, 2000.
11. Reed, D.A., Aydt, R.A., Noe, R. J., Roth, P.C., Shields, K.A., Shwartz, B.W., and Tavera, L.F. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conf.*, IEEE Computer Society, 1993.
12. Sarukkai, S. R., Yan, J., Gotwals and J. K.. Normalized performance indices for message passing parallel programs. *In Proc. of the 8th international conference on Supercomputing, Manchester*, England, 1994.
13. Schmidt, B., and Sunderam, V.: Empirical Analysis of Overheads in Cluster Environments. Concurrency: Practice and Experience 6, 1 (February 1994), 1-32.
14. Truong, H-L and Fahringer, T.: Performance Analysis for MPI Applications with SCALEA. *In Proc. of the 9th European PVM/MPI Conf.*, Linz, Austria (September 2002).
15. Yan, J. C., Sarukkai, S. R., and Mehra, P.: Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit. *Software Practice & Experience*, April 1995.