Towards a Generalised Performance Analysis of Parallel Processing

Jan LEMEIRE
And
Erik DIRKX
PADX, VUB
Brussels, Belgium
Email: jan.lemeire@vub.ac.be

Unpublished, Submitted for IPDPS 2003, Nice, France

ABSTRACT

This paper proposes a generalised model of parallel performance. Our research investigates how far a generalised approach can obtain the same detailed results as current instantiated analysis of specific parallel algorithms. Performance is decomposed into the different overhead sources, that are expressed analytically and we showed how to separate the dependency of the algorithm and the system. We present a new method to overcome reported problems on measurability of overheads and this detailed analysis reveals 'hidden' sources of blocking overhead, due to communication or extra computation. The presented approach can lead to an automated experimental analysis, in order to support performance prediction, straight forward development of parallel programs, and can be used in partitioning and load balancing algorithms. Therefore, it is focussed on the programmers viewpoint of parallel processing.

Keywords: Parallel processing, performance analysis, speedup, performance overheads.

1. INTRODUCTION

To process a task in parallel, the use of specific, instantiated parallel algorithms is still necessary. However, for succes, the development of efficient parallel programs should be straight forward. We want to support this task by automating the performance analysis, as performance is the sole reason for parallel processing [Miller 2002]. Therefore we develop a generalised - but detailed – analysis. This will serve as the basis of our investigation of the possibility to fully automate the parallel performance analysis that covers all aspects and can interprete the results to obtain full insight in the performance. So we want to extract the general rules that guides a specific analysis, which is after all the goal of all science.
We intend to serve a program-oriented view [Bull '96] [Pancake '99] on a simple, understandable and reasonably accurate performance evaluation.

Note that we focus on message-passing architectures and ignore memory bottlenecks in our discussion.

The next section outlines a performance metrics for investigating its handicaps.

2. PARALLEL OVERHEAD

Our approach starts by looking for the reasons of non-ideal performance: speedup less than the number of processors. Figure 1 shows the timeline of a typical parallel program on a message passing architecture, with its different phases: the partitioning of the work, the communication of data, the useful work of the processors job, the synchronisation and the induced blocking.

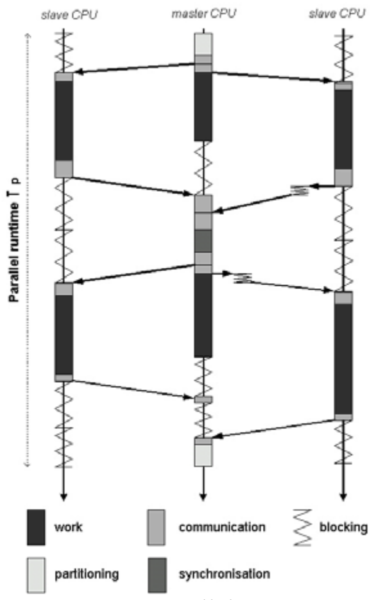

Figure 1: Parallel Processing

By writing the parallel execution time T_p as the perfectly parallelised sequential work plus the parallel overhead, speedup can be expressed by [Kumar '94, Lemeire 2001]:

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

with p the number of processors and T_s the runtime of the sequential algorithm. The total parallel overhead is the sum of all sources of overhead, the **overhead terms OT_j**:

$$overhead = \sum_j OT_j \quad (2)$$

$$OT_j = \sum_i^p OT_{j,i} \quad (3)$$

Each overhead term is summed over all processors (index *i*) to get an average value and an overall view of the overhead. The relative impact of the overhead on the speedup is then expressed by the **slowdown terms ST_j**:

$$ST_j = \frac{OverheadTerm_j}{SeqTime} = \frac{OT_j}{T_s} \quad (4)$$

Sequential Runtime

By ignoring memory bottlenecks, the sequential runtime can be written as:

$$T_s = \delta_{instr}.\#instr \quad (5)$$

Where δ_{instr} represents the system dependency and *#instr* the work of the program. From the programmers viewpoint, the number of instructions is the major abstract interpretation of the work, so we use this as a useful first order approximation of the sequential runtime.

Different Processing Powers

In systems with heterogenous processors, eq (1) doesn't hold, we therefore have to introduce p_i, representing the relative processing power with respect to a reference processor [Zaki '96]:

$$p_i = \frac{CPU_i}{CPU_{reference}} \quad (6)$$

We redefine *p* as the sum of the processing powers:

$$p = \sum p_i \quad (7)$$

If the sequential runtime is measured on the reference processor, equation (1) can be rewritten:

$$T_s = \sum_i T_{s,i}.p_i \quad (8)$$

$$T_p = \frac{\sum T_{p,i}.p_i}{p} \quad (9)$$

$$S = \frac{p}{1 + \frac{\sum_i \sum_j OT_{j,i}.p_i}{T_s}} \quad (10)$$

This results in the (expected) conclusion that we have to scale all time measurement on a processor with p_i. This corresponds to the cycle counting of [Crovella '94].

For using the above performance equations, the next section investigates the overhead terms in detail.

3. OVERHEAD TERMS

In our quest for a generalised model, we also plead for a *complete*, *orthogonal* and *meaningful* overhead inventarisation, as [Crovella '94 and Bull '96]. However, our classification differs because we want to reflect our experience and we doesn't use *measurability* [Bull '96] as a criterion. We use *deducibility* as a basic requirement and will show in the discussion that all our overhead terms are deducible from the measured ones.

We first identify 3 major overhead classes:

1. **Control of parallelism** [Bull '96]: the extra functionality necessary for parallelisation. This can be subdivided into the different logical parts of the parallel algorithm, like the partitioning, the recombination and the synchronisation. The synchronisation overhead is defined as the extra calculations needed to ensure the correct course of the parallel processing, like the calculation of the cycle time for the barrier synchronisation in parallel discrete event simulation [Lemeire 2000]. In a similar way, each phase of the algorithm is added and the code of these parts can easily be instrumated.
2. **Communication:** computational overhead (in the sense of the loss of cycles [Crovella '94]) due to the exchange of data between processors.
3. **Blocking**: the processors idle time.

Table 1 subdivides these overhead classes, where the thick border represent the measurable overheads.

	Computation			**Blocking**	
Control of Parallelism					
	OT1	partitioning	=>	OT7	
	OT2	recombination	=>	OT8	
	OT3	synchronisation	=>	OT9	
Communicatic					
	OT4	bandwidth	=>	OT10	
	OT5	link	=>	OT11	
		Tdelay	=>	OT12	network delay
		Tcongestion	=>	OT13	network blocking
Work	Ts,j				
		useful parallel work	=>	OT14	Amdahl
	OT6	parallel work anomaly		OT15	global imbalance
				OT16	temporal imbalance

Table 1: Overhead Inventarisation

The following discussion investigates the influences of the algorithm and system on each overhead term.

Computational Overhead

The computational overhead is the extra computation of the parallel algorithm (first column of table 1). The partitioning overhead leads to a system-independent ratio partitioning versus sequential work:

$$ST_1 = \frac{OT_1}{T_s} = \frac{\#part}{\#seq} \quad (11)$$

Identically, recombination and synchronisation results in parallel performance factors:

$$ST_2 = \frac{OT_2}{T_s} = \frac{\#recomb}{\#seq} \quad (12)$$

$$ST_3 = \frac{OT_3}{T_s} = \frac{\#sync}{\#seq} \quad (13)$$

The communication time is adopted as a simple linear function of the transmitted data size and a constant additive factor representing "link startup" overheads. This is a conventional approach in analyzing communications for most message-passing, distributed systems [Bomans '89, Steed '96]. The communication overhead can thus be split into the computation proportional to the communicated data size (OT_4) and the part proportional to the setting up of the communication links (OT_5):

$$ST_4 = \frac{OT_4}{T_s} = \frac{\delta_{comm}}{\delta_{cpu}} \cdot \frac{\#bytes}{\#instr} \quad (14)$$

$$ST_5 = \frac{OT_5}{T_s} = \frac{\delta_{link}}{\delta_{cpu}} \cdot \frac{\#links}{\#instr} \quad (15)$$

We get 2 system performance characteristics (δ_{comm} and δ_{link}) and 2 algorithm performance characteristics, that again represent the programmers view on the impact of communication. Both terms can easily be deduced from the measured communication overhead by linear curve fitting on the experimental data.

The equations (11) – (15) of the slowdown terms result in 2 system and 5 algorithm **parallel performance factors**.

In specific cases, the sum of the useful parallel work $T_{s,j}$ differs from the sequential work T_s, as for example in discrete optimisation problems [Kumar '94]:

$$\sum T_{s,j} = T_s + OT_6 \quad (16)$$

We call this the **parallel work anomaly** OT_6, which can be positive or negative.

Blocking overhead

In literature, blocking overhead is mainly traced to load imbalances, but this is not the only cause. [Bull '96] also refers to problems in classifying blocking. The time diagram of Fig. 2 (a detail of Fig. 1) shows that besides the load imbalance (OT_{14}), blocking can also be generated by communication phases (OT_{10}) and network delays (OT_{12}).

In general, every computation and every network delay can cause blocking. Partitioning for example happens mostly sequentally 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 phases, blocking other processors. In our opinion, these effects are easily overlooked.

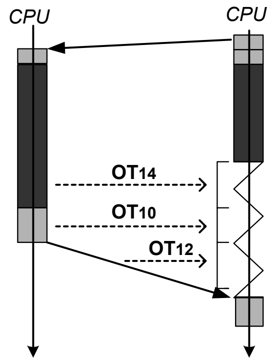

Figure 2: blocking source

It is necessary to determine the sources of blocking in order to add blocking overhead to its source overhead. First, imbalances in computation phases are represented by the ratios β_I:

$$\beta_i = \frac{\sum_{phases} diff(OT_i)}{OT_i} \quad (i=1..5) \quad (17)$$

Where the differences of computational phases are measured between synchronisation points.

But not every imbalance induces an equivalent blocking:

1. Imbalances on different processors can cancel each other out.
2. A processor can simultaneously generate blocking on several processors (master processor in Fig 1).
3. Blocking on one processor can be caused by several processors (slave processors in Fig 1).

We express this cause-effect relation by a global factor Γ, that can be measured. So, blocking due to computational imbalances can be calculated according to:

$$OT_{i+6} = \Gamma.\beta_i.OT_i \quad (i=1..5) \quad (18)$$

Network delays

Besides computational imbalances, network delays are also generating blocking. We separate the communication delay -depended on the speed of the connections - from the delay caused by network congestion. The totalisation of the communication delays is:

$$T_{delay} = \#links.\delta_{delay} \quad (19)$$

$$OT_{12} = \Gamma.T_{delay} \quad (20)$$

$$ST_{12} = \Gamma.\frac{\#links}{\#instr}.\frac{\delta_{delay}}{\delta_{cpu}} \quad (21)$$

where *#links* is determined by the algorithm and δ_{delay} the average delay of a message.

For calculating OT_{13} we need the total congestion time $T_{congestion}$, this delay is also proportional to the number of communication links:

$$T_{congestion} = \# links.\delta_{congestion} \quad (22)$$

However, for $\delta_{congestion}$ it is more difficult to seperate algorithm and system dependency. This delay depends on the congestion-sensitivity of the communication behaviour of system and algorithm: the overlap of communication paths, the amount of simulataneous communication and how the communication maps on the network topology. A first order approximation for $\delta_{congestion}$ is:

$$\delta_{congestion=} \textit{system-sensitivity*alg-sensitivity} \quad (23)$$

where *system-sensitivity* is the average congestion delay with random communication and the *algorithm-sensitivity* is measured on an average network. Equation (23) thus seperates system- and algorithm dependency, but is only a rough approximation. For better results, models of the communication behaviour is necessary.
Note that δ_{delay} in Eq (19) is also a first order approximation: the *average* communication delay between 2 processors, without any knowledge of the specific communication of the algorithm.
Seperating the influence of algorithm and system benefits in porting parallel programs between different parallel systems, as all parallel performance characteristics can be measured independently.

Workload imbalances
The most important blocking, due to work imbalances, can be subdivided into 3 important parts. First, Amdahls law expresses the limitation of parallelism, parallel execution time can be written as

$$T_p = \frac{(1-s).T_s}{p} + s.T_s \quad (24)$$

with s, the serial, unparallelisable fraction of the algorithm. Speedup gives then [Barton '89]:

$$S = \frac{p}{1+(p-1).s} \quad (25)$$

So, to obtain a uniform analysis, we will represent Amdahls law by OT_{14}, the blocking, due to the limitation of parallelisation:

$$OT_{14} = (p-1).s \quad (26)$$

This overhead is different than blocking due to bad partioning, which can be split into 2 terms: the global and the temporal load imbalances. The first is the difference of the *total* work of the processors, the temporal is caused by load fluctuations between synchronisation points. Whereas the global load imbalance can be reduced by good partitioning, temporal load imbalances are more difficult to master and can give high slowdowns, especially with increasing p, as reported in [Lemeire 2000].
This subdivision of imbalances is useful for partitioning algorithms. This and other benefits of our approach are discussed in the next section.

4. BENEFITS

First of all, a generalised analysis of parallel performance makes exchange of results easier and more relevant. Moreover, to support the programmer with a performance analysis tool, a standard overhead classification is necessary.
Next, the here developed detailed analysis of the overhead terms, ends in a clear and complete *understanding* of the parallel performance. This is certainly true in the cases of non-trivial and unpredictable overhead, for example when the blocking is mainly caused by simultaneous communication [Parent 2002]. This insight is indispensable for efficient *optimisation* of the parallel algorithm.

Parameter and system dependency
The performance should be known in function of all relevant algorithm and system parameters. The number of processors p and the problem size W are the most general parameters, but each algorithm and system adds specific ones. This is necessary for scalability analysis [Kumar '91], for cost-speedup tradeoff [Kumar '94], for calculation of the optimal speedup, etc.
A possible way of obtaining the analytic equations *Speedup=f(parameters)* is by experimentally measuring the perfomance for different values of the parameters. Then, analytic equations should be extracted from the experimental data. We expect this to be possible due to the detailed overhead measurement of our approach, so that each part will mostly depend on a simple equation derivable from experimental data.

The interpretation layer
Once all of the overhead terms are calculated, these results should be interpreted. Insignificant terms can be neglected in order to extract the major bottlenecks. The algorithm and system dependency of these bottlenecks will then reveal the nature of the parallel performance.

Utilisation of *S(parameters)*
The major problems at algorithmic level of parallel processing are the parallelisation, the load balancing [Zaki '96], the partioning and the performance analysis (fig. 3). We will investigate how far a generalised and automated performance analysis can serve the necessary performance information for parallelisation, load balancing and partitioning algorithms.

In this discussion, there should be made a difference between *embarassingly parallel* problems and non-trivial parallel algorithms. In the first category, the performance analysis can be reduced to the communication – computation ratio and is therefore easy to compute, the load balancing will be the main difficulty. The only benefit of our approach would be the automatic analysis of the system dependency of the performance. For the second category, particular parallel solutions are necessary, resulting

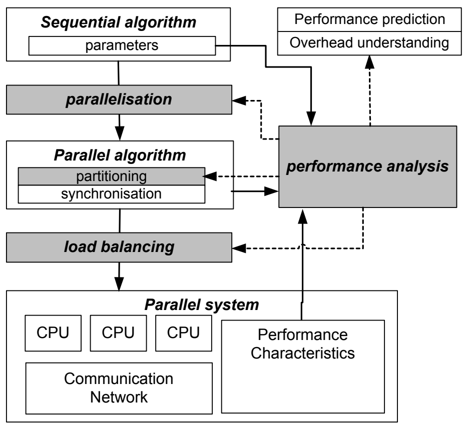

Figure 3: Parallel Processing

in specific performance bottlenecks [Kumar '94]. We will have to proof that our general approach can reveal these 'hot spots'.

An approach of generalisation should lead to a tool for an automated performance evaluation, discussed in the next section.

5. AUTOMATED ANALYSIS

The standard communication layer Pvm is supported by the visual tool XPvm [Kohl '95], which automatically analyses the computation, communication and blocking phases. These are measured within the pvm communication layer, so no extra code has to be added to the parallel program. However, we argued that a more detailed overhead analysis is necessary, as implemented in tools like VGV [Kim 2002], Ovaltine [Bane 2000], Paradyn [Miller '94] and SCALEA [Truong 2002]. Where this last one resembles best the tool we envisage.

In our approach the computational phases are differentiated by code instrumentation, indicating the role of each part. Simultaneously, the values of relevant algorithm variables are passed, like the communication datasize or the number of performed iterations. As with SCALEA [Truong 2002], a "multiple experiment performance analysis" is possible to investigate *S(parameters)*. Herefore, an "experiment director" decides what experiments are necessary and configures the parameters of system and algorithm. The visual part of the tool presents the interpreted results in different *layers*, where each layer represents an aspect of the analysis:

1) The *time layer* shows all variables of one experiment in function of the algorithm runtime (figure 1).
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. Also the possibility to perform partial measurements in order to extract equations of fundamental operations, eg. perform 1 sort iteration to measure its time constant.

6. CONCLUSIONS

This paper wants to contribute in the development of a generalised parallel performance analysis. The parallel overhead sources were studied in detail with the criterion of deducibility. This resulted in a better understanding of the reasons for blocking and we showed that it is a wrong assumption to completely dedicate blocking to load imbalances. Then the impact of the algorithm and the system were seperated by a first order approximation and we showed how this can result in an automated analysis. The goal of our research is to facilitate parallel processing, therefore we are investigating if a generalised, standard analysis can provide all necessary results for the parallel programmer. The here developed approach will serve as the basis of this research. First, we will have to proof that we can get the same detailed results as an instantiated algorithm-specific analysis. Herefore, we will try to find again known results, like the performance discussions described in [Kumar '94]. It is also not yet clear if experimental data suffices for obtaining analytic dependencies. Next, the desciption of the system-dependency is crucial, in certain cases, a first-order approximation will fail and higher-order analysis will become necessary. Finally, we will have to investigate whether these results can be used in partitioning and load balancing algorithms.

7. REFERENCES

Bane, M.K. and Riley, G.D., "Automatic Overheads Profiler for OpenMP Codes". In proceedings of EWOMP2000 conference, Edinburgh, Scotland, 2000.

Barton, M.L. and Whiters, G.R., "Computing performance as a function of the speed, quantity, and cost of the processors." Supercomputing, pp. 759-764, 1989.

Bomans, L. and Roose D., "Benchmarking the iPSC/2 Hypercube Multiprocessors", Concurrency: Practice and Experience, Vol. 1, No. 1, pp.3-18, September 1989.

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, pp. 208-219, March 1996.

Crovella, M. E. and Leblanc, T.J., "Parallel Performance Prediction using Lost Cycles Analysis", in *Proc. of Supercomputing '94*, IEEE Computer Society, 1994.

Kim, S.W., Voss, M. et al. "VGV: Supporting Performance Analysis of Object-Oriented Mixed MPI/OpenMP Parallel Applications", in *Proc. of the 16th IPDPS Conf.*, IEEE, California, April

2002.

Kohl, J.A. and Geist, G.A.. "XPVM 1.0 User's Guide". Tech. Rep. 12981, Computer Science and Mathematics Division, Oak Ridge National Laboratory, April 1995.

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

Kumar, V. and Gupta, A., "Analysis of scalability of parallel algorithms and architectures: a survey." in *Proc. of the 5th International Conference on Supercomputing*, pp. 396-405, ACM, Cologne, Germany, 1991

Lemeire, J. and Dirkx, E., "Performance Factors in Parallel Discrete Event Simulation", in *Proc. of the 15th European Simulation Multiconference (ESM 2001),* Prague, 2001.

Miller, Barton P., et all. "Performance Evaluation, Analysis and Optimization", *In Proc. of the 8th International Euro-Par Conf.*, p. 131, Paderborn, Germany, August 2002.

Miller, B.R, Hollingsworth, J.K. and Callaghan, M.D., "The Paradyn Parallel Performance Measurement Tools and PVM", in: J. Dongarra, B. Tourancheau (Eds.), "*Environments and Tools for Parallel Scientific Computing*", SIAM Press (1994).

Pancake, C.M. "Applying Human Factors to the Design of Performance Tools", in *Proc. of the 5th Euro-Par Conf.*, Springer, 1999.

Steed, M.R. and Clement, M.J. "Performance Prediction of PVM Programs", *Proc. of the 10th Int'l Parallel Processing Symposium*, april 1996.

Truong, H-L and Fahringer, T. "SCALEA: A Performance Analysis Tool for Distributed and Parallel Programs." In *Proc. of the 8th International Euro-Par Conf.*, Paderborn, Germany, August 2002.

Zaki, M.J., Wei Li and S. Parthasarathy, "Customized dynamic load balancing for a network of workstations", *Proc. of the High Performance Distributed Computing (HPDC'96),* IEEE, 1996.