Parallel Systems: Performance Analysis of Parallel Processing

Jan Lemeire

November 6, 2007

Chapter 1

Performance Analysis of Parallel Processing

PARALLEL processing is the only answer to the ever-increasing demand for more computational power. Nowadays, the big giants in hardware and software, like Intel and Microsoft, are increasingly aware of it and have pounced onto the market. But unlike sequential programs running on the Van Neumann computer, the parallelization of programs is not trivial. It depends quite heavily on the underlying parallel system architecture. Automatic parallelization of programs is a 50-year old dream in which a program is efficiently matched with the available computing resources. This has become possible, but only for a very limited number of applications, the class of trivially parallelizable programs. For those, the computational work can be divided into parts which can be processed completely independently. Other programs, on the other hand, need manual adaptation to the available resources. This cannot be achieved without a detailed understanding of the algorithm. Intelligent reasoning is necessary to engineer the matching of the patterns of the concurrently operating entities to the pattern of the processors and the network resources, in order to obtain an efficient interplay of computation and communication. The aim of a performance analysis is to provide support for the developer of parallel programs.

The goals of a performance analysis are multifold:

- An understanding of the computational process in terms of the underlying processes: instructions performed, processor cycles spent, cache misses, memory hierarchy utilization, communication, resource utilization per program section, number of iterations, and so on.
- An identification of inefficient patterns, the bottlenecks that unnecessarily slow down the execution process. In particular, performance values that are, given the context, 'abnormally' low and which can be considered for optimization and improvements. They indicate up to which points tuning efforts are most effective.
- A prediction of the performance for a new program or system configurations. A performance model should provide an expectation of the achievable performance with a reasonable fidelity, as a function of program and system parameters.
- The definition of program and system properties that fully characterize their performance, i.e. which allow the quantification of their performance for a wide range of systems and system configurations.

Various tools exist nowadays for automated diagnosis and control. Considerably more effort is needed to improve current work to present the user a simple, comprehensible and reasonably accurate performance evaluation [Pancake, 1999]. Current challenges are further automation, tackling complex situations (e.g. GRID environments [Zsolt Nemeth, 2004]) and providing the software developer with understandable results with a minimum of learning overhead. To sketch the difficulty of the task, consider the study of network performance. Communication delays should be attributed to the different steps of the communication process, such as machine latency, transfer time, network contention, flight time, etc [Badia, 2003]. A correct understanding of the origins of the delays is indispensable. The task of identifying them becomes even more difficult when implementation-specific low level issues come into play, such as specific protocol behavior, window delays or chatter [NetPredict, 2003]. These are not always fully understood and can often not be measured directly.

This chapter discusses the parallel performance metrics which where employed to build our performance analysis tool, EPPA. These metrics are based on the lost cycle approach. Overhead ratios are defined to quantify the

impact of each type of lost cycle, or overhead, on the overall performance, the speedup. The second section explains which information is recorded by EPPA and compares the tool with related work.

1.1 Parallel Performance Metrics

Parallel processing is the simultaneous execution of a program by multiple processors. **Parallelization** is the rewriting of a sequential program into a program that can be processed in parallel and that gives the same result as the sequential program. The advantage is that the combined computing and memory resources of the processor group can be utilized. Calculation or memory intensive programs can fruitfully exploit the aggregated resources to finish the job in less time.

Example 1.1 (Parallel processing I: Protein folding).

Proteins are long chains of thousands of amino acids. After creation, the sequence 'folds' into a unique 3-dimensional structure that determines the protein's properties. The shape into which a protein naturally folds is known as its native state. Fig. 1.1 shows an antibody against cholera, unfolded and in its native state. Understanding the structure of a protein is critical in understanding its biological function. The structure of (synthetic) proteins can be determined by running detailed simulations of the folding process. Because of the complexity and multitude of interactions, these computations require 'zillions' of processor cycles. It takes with today's computers about 10000 days to simulate a particular folding of an average protein. **Folding@Home** is a distributed computing project from Stanford university to tackle this performance problem (`http://folding.stanford.edu/`). People from throughout the world run the software and make one of the largest supercomputers in the world in the form of a computational grid. The participation in this project throughout the world is depicted in Fig. 1.2. Every computer runs a section of the simulation for one of the many protein foldings that need to be calculated in research on Alzheimer's Disease, Cancer, Parkinson's Disease, etc. To contribute, you simply install a small program on your computer which runs in the background only consuming processor time when there is no other work.

The runtime of a sequential program is defined as T_{seq}. The parallel version, whose runtime is denoted as T_{par}, will hopefully finish faster. The profit of switching from one to multiple processors is characterized by the **speedup**:

$$Speedup = \frac{T_{seq}}{T_{par}} \tag{1.1}$$

It expresses how much faster the parallel version runs relative to the sequential one. Note that in the context of parallel processing I denote the computation time of the sequential program as T_{seq}, whereas in context of sequential computing I denote the computation time as T_{comp}.

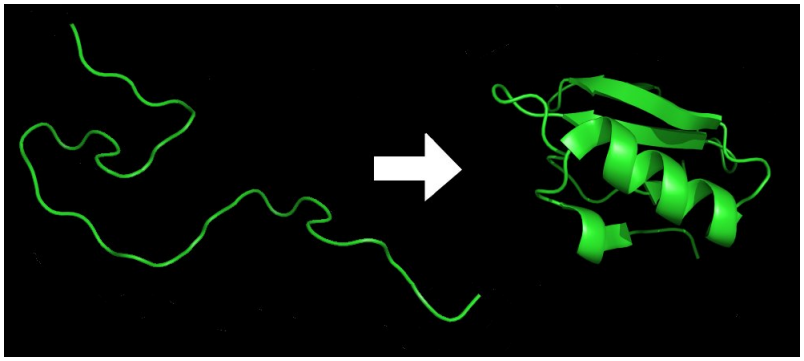

Figure 1.1: Protein folding, from amino acid sequence to a 3-dimensional structure.

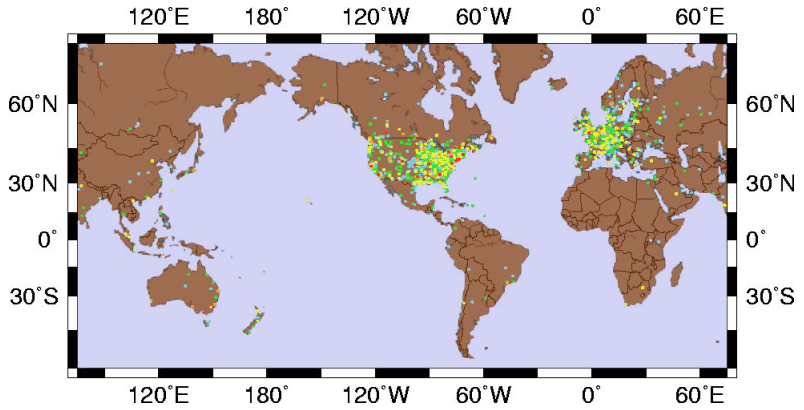

Figure 1.2: Folding@Home's supercomputer. Distribution accross the world of the computers participating in the project.

When the parallel program is run on p processors, the **efficiency** is defined as:

$$Efficiency = \frac{Speedup}{p} = \frac{T_{seq}}{p.T_{par}} \tag{1.2}$$

Efficiency measures how well a processor is used during the parallel computation. It represents the effectivity of the set of cooperating parallel processes. Efficiency quantifies the portion of the parallel runtime during which the processors where doing *useful work*, i.e. when the parallel execution is performing parts of the sequential execution. Ideally, the efficiency is 100%, which is equivalent to a speedup of p. Each processor optimally executes an equal part of the sequential program. In practice, the effectivity of the parallel program is limited due to the inevitable *parallel overhead*, such as communication of data between the processors. Hence, speedup will be smaller than p. It can even become lower than 1, which indicates a slow down instead of a speed up. On the other hand is it also possible to attain a speedup higher than p, called *superlinear speedup*. It typically occurs when the parallel program succeeds in a more efficient utilization of the memory hierarchies of the processors. This results in lower access times of the memory hierarchies.

It must be noted that the here developed performance metrics focus on the *computation time* of the process. Other performance metrics, such as energy utilization, are, despite their increasing importance, not considered here. On the other hand, a generic approach is pursued, one that applies for a multivariate analysis in general.

1.1.1 Lost Cycle Approach

For the analysis of the parallel runtime and overhead, I adopt the *lost cycle approach*, as conceived by Crovella and LeBlanc [1994]. It provides a measure of the impact of the overhead on the speedup. Ideally, each processor computes its part of the total work. Thus without additional work, we would have $T_{par} = T_{seq}/p$. The work is divided among the processors of the parallel system. The total useful computational work is characterized by the sequential runtime. The *Speedup* then equals to p. In practice, however, additional processor cycles are needed to manage the parallel execution, $T_{par} > T_{seq}/p$. **Overhead** is therefore defined as

$$overhead = p.T_{par} - T_{seq}. \tag{1.3}$$

Each process has T_{par} time allocated to perform its part of the job. The cycles during this period that are not employed for useful computation are

therefore considered *lost processor cycles*. Take the following example.

Example 1.2 (Parallel processing II: Parallel Matrix Multiplication).

Consider the multiplication of 2 square matrices with size $n \times n$: $C = A \times B$. The elements of the product matrix C are calculated according to the formula

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k}.B_{k,j}, \tag{1.4}$$

with i and j indicating respectively the row and column of the element. The computation involves n^3 multiplications and $n^2 \times (n-1)$ additions. The runtime rapidly increases for higher values of n, what makes it worth for being computed in parallel for high values of n. There exist many ways to calculate the product in parallel. A simple version is illustrated by Fig. 1.3. The A matrix is striped into p blocks n/p of contiguous rows, the B matrix into p blocks of n/p columns. They are distributed among the p processors. Each processor stores a submatrix of A and one of B, labeled in Fig. 1.3, in which p is 3. A master processor does the partitioning and sends the submatrices to the slave processors. The algorithm then alternates p computation and communication steps. In each computation step, each processor multiplies its A submatrix with its B submatrix, resulting in a submatrix of C. The black circles in Fig. 1.3 indicate the step in which each submatrix is computed. After the multiplication, each processor sends it B submatrix to the next processor and receives one from the preceding processor, in such way that the communication forms a circular shift operation. When finished, the slaves send their part of C to the master computer. The timeline of the execution on our cluster of Pentium II processors connected by a 100MBs switch is shown in Fig. 1.4. Two types of overhead can be identified: communication and idling. The speedup for the computation of a 100×100 matrix is 2.55 and the efficiency is 85%.

The parallel runtime on processor i consists of its part of the useful work, T^i_{work}, and the cycles spent on the overheads. The impact of the different

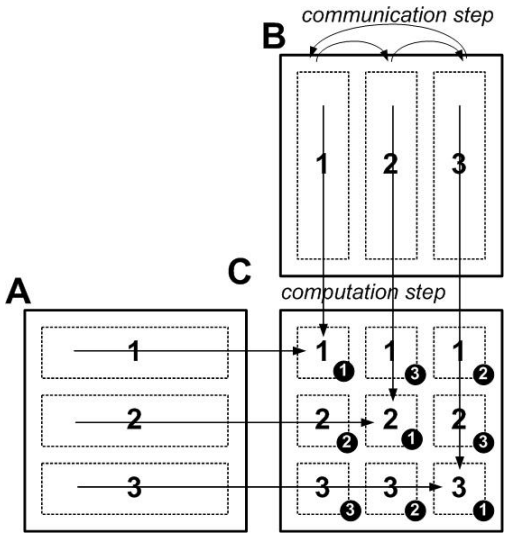

Figure 1.3: Parallel Matrix Multiplication on 3 processors: partitioning, computation and communication in 3 steps. At each step, 3 submatrices are calculated, indicated with black circles.

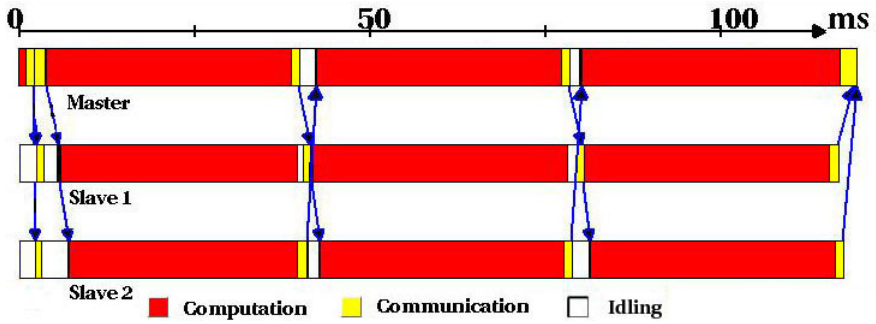

Figure 1.4: Execution profile of a Parallel Matrix Multiplication of two 100x100 matrices.

types of overhead will be analyzed separately. Each overhead type is labeled with an index j. The number of overhead types is denoted with O. $T_{ovh}^{i,j}$ then denotes the time of overhead j on processor i. The runtime on every processor can then be written as:

$$T_{par}^{i} = T_{work}^{i} + \sum_{j}^{O} T_{ovh}^{i,j} \quad \text{with} \quad i = 1 \dots p \tag{1.5}$$

$$T_{seq} + T_{anomaly} = \sum_{i}^{p} T_{work}^{i} \tag{1.6}$$

where $T_{anomaly}$ is the difference between the sum of all cycles spent on useful work by the different processors and the sequential runtime. In most cases it is very close to zero. If positive, the execution of the useful work takes more time in parallel. If negative, the parallel execution is faster, for example by a more efficient use of the memory hierarchy. The parallel runtime is the same on all processors:

$$T_{par} = T_{par}^{1} = \ldots = T_{par}^{p}. \tag{1.7}$$

Hence, we may write:

$$T_{par} = \frac{\sum_{i}^{p} T_{par}^{i}}{p} \tag{1.8}$$

Together with 1.5 it follows that

$$T_{par} = \frac{\sum_{i}^{p} T_{work}^{i} + \sum_{i}^{p} \sum_{j}^{O} T_{ovh}^{i,j}}{p} \tag{1.9}$$

$$= \frac{T_{seq} + T_{anomaly} + \sum_{j}^{O} T_{ovh}^{j}}{p} \tag{1.10}$$

with $T_{ovh}^{j} = \sum_{i}^{p} T_{ovh}^{i,j}$, the total time of overhead j. Parallel anomaly is also regarded as overhead, although that it might be negative. It is therefore

added to the overheads, T_{ovh}^{O+1} The speedup can then be rewritten as

$$Speedup = \frac{T_{seq}}{\frac{T_{seq} + \sum_{j}^{O+1} T_{ovh}^{j}}{p}} = \frac{p}{\frac{\sum_{j}^{O+1} T_{ovh}^{j}}{T_{seq}} + \frac{T_{seq}}{T_{seq}}} \tag{1.11}$$

Hence [Kumar and Gupta, 1994]:

$$Speedup = \frac{p}{1 + \sum_{j} \frac{T_{ovh}^{j}}{T_{seq}}}. \tag{1.12}$$

The equation expresses how the overheads influence the speedup. The lost processor cycles must be considered relative to the sequential runtime. Without any overhead, the speedup equals to p.

1.1.2 Overhead Ratios

From Eq. 1.12 it follows that the impact of overhead on the speedup is reflected by its ratio with the sequential runtime. I call these terms the **overhead ratios**. They express the relative weight of the overhead term:

$$Ovh_j = \frac{T_{ovh}^{j}}{T_{seq}}. \tag{1.13}$$

The speedup is then:

$$Speedup = \frac{p}{1 + \sum_{j} Ovh_j}, \tag{1.14}$$

and the efficiency gives

$$Efficiency = \frac{1}{1 + \sum_{j} Ovh_j}. \tag{1.15}$$

These definitions differ slightly from the **normalized performance indices** used by the performance tool AIMS, defined as $index_j = T_{ovh}^{j}/T_{par}$ [Sarukkai et al., 1994]. They are always less than one, while the overhead ratios become more than one if the runtime of the overhead surpasses the

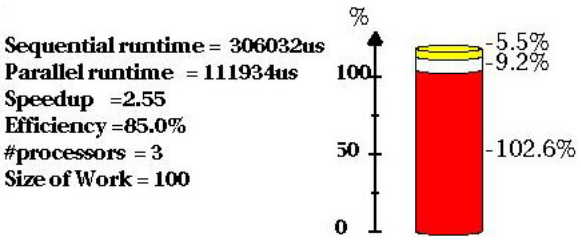

Figure 1.5: Parallel Matrix Multiplication on 3 processors: overall performance.

sequential runtime. The advantage of the overhead ratios is that they are independent of the other overheads. This is not the case for the indices, since T_{par} incorporates all overheads. If one overhead increases, its index increases and the indices of the others decrease, since their relative weight decreases.

Example 1.3 (Overheads of Parallel Matrix Multiplication).

Fig. 1.5 shows the overall performance of the run of the previous example. Two overheads are identified: the communication and the idle time. Their ratio with the sequential time, Ovh_j, is given. The sum of the processor's computation times, $\sum_i^p T_{work}^i$, divided by the sequential runtime is also given, but is not equal to 100%. A value of 100% means that the computation time of the useful work is equal for a sequential as for a parallel execution. It is 102.6% instead, which means that the overhead ratio of the parallel anomaly is 2.6%. In parallel, 2.6% more cycles are needed to do the same work. Additionally, Fig. 1.5 shows the overhead ratios per processor individually.

Figure 1.6: Parallel Matrix Multiplication on 3 processors: overhead ratios per process.

1.1.3 Overhead Classification

The different overheads of a parallel execution can be classified into the following classes:

1. *Control of parallelism* ($T_{ctrlPar}$) identifies the extra functionality necessary for parallelization. This additional work can be further subdivided into the different logical parts of the parallel algorithm, like partitioning or synchronization, as done by several authors [Bull, 1996] [Truong and Fahringer, 2002].

2. *Communication* (T_{comm}) is the overhead due to the exchange of data between processors. It 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 due to a communication operation.

3. *Idling* (T_{idle}) is the processors idle time. It happens when a processor has to wait for further information before it can continue. Reasons for idling are for example load imbalances, when the work is unequally distributed among the processes, or a bottleneck at the master, when it has to serve all slaves.

4. *Parallel anomaly* ($T_{anomaly}$) is the difference between the sum of all cycles spent on useful work by the different processors and the sequential runtime (Eq. 1.6). By the alternative speedup formula (Eq. 1.12), $T_{anomaly}$ was regarded as overhead. It influences the speedup, given by its ratio with the sequential runtime.

1.1.4 Granularity

To illustrate how a performance analysis is performed, this section introduces one of its most influential concepts: granularity. The key to the execution of parallel algorithms is the communication pattern between concurrently operating entities. By choosing speedup as the main goal of parallelization, Eq. 1.12 shows that overheads should be considered relatively. Communication overhead T_{comm} must be considered with respect to the computation time. The inverse of the communication overhead ratio is called the **granularity** [Stone, 1990]:

$$Granularity = \frac{T_{comp}}{T_{comm}} = \frac{1}{Ovh_{comm}} \tag{1.16}$$

Granularity is a relative measure of the ratio of the amount of computation to the amount of communication of a parallel algorithm implementation. The bigger the granularity, the more the application spends time in computation relative to communication. Another interpretation is that it expresses the size of the tasks. Since the communication is often the main overhead, the granularity gives a good indication of the feasibility of parallelization. With a granularity of one, the efficiency is 50%.

The communication time can be modeled as a simple linear function of the transmitted data size and a constant additive factor representing link startup overheads (*latency*). This is a conventional approach in analyzing communications for most message-passing, distributed systems [Steed and Clement, 1996]. The communication time can thus be split into a component proportional to the communicated data size and a part proportional to the latency of the communication links. For computing-intensive tasks and large data chunks, the data-proportional part overwhelms the constant part so that the latter can be neglected. Since parallelization is used for computation-intensive tasks, the approximation is valid. The communication overhead time can then be written as $\beta.q_{data}$, with q_{data} the size in bytes of the communicated data. Assume that we can approximate the computation time by $\tau.q_{operations}$, with $q_{operations}$ the number of basic operations of the algorithm and τ the cycles per operation. The granularity can then be rewritten as:

$$Granularity = \frac{T_{comp}}{T_{comm}} = \frac{\tau}{\beta}.\frac{q_{operations}}{q_{data}} \tag{1.17}$$

This ratio depends on hardware and software, so τ/β is called the **hardware granularity** and $q_{operations}/q_{data}$ the **software granularity**. The

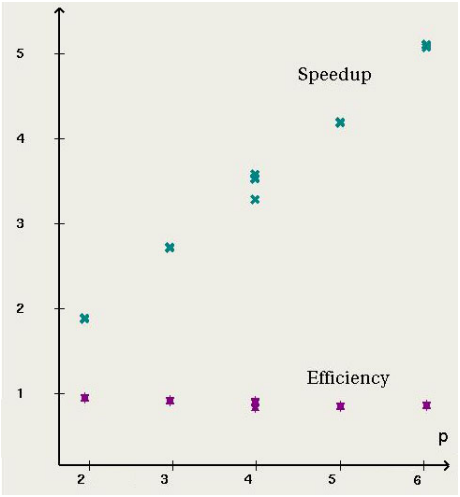

Figure 1.7: Performance of Parallel Matrix Multiplication as a function of number of processors (with $n = 100$).

performance is affected by the overall granularity, independent of how it is spread over software and hardware.

1.1.5 Parameter Dependence

As can be expected, parallel performance heavily depends on program and system configuration. Most numerical, computation-intensive algorithms have a parameter that determines the size of the computational work. I call it the **work size** parameter, which I denote with W. For parallel processing, p, the number of processors participating, is the most important system parameter. The following example gives a typical parameter dependency analysis.

Example 1.4 (Parameter Dependence of Parallel Matrix Multiplication).

Let's go back to the example of the multiplication of 2 $n \times n$ matrices. Fig. 1.7 shows speedup and efficiency in function of p. Although the speedup increases when more processors are employed, the efficiency decreases.

Performance as a function of matrix size n gives a different picture. Experimental results are shown in Fig. 1.8. Communication increases with increasing n, but computation increases a lot

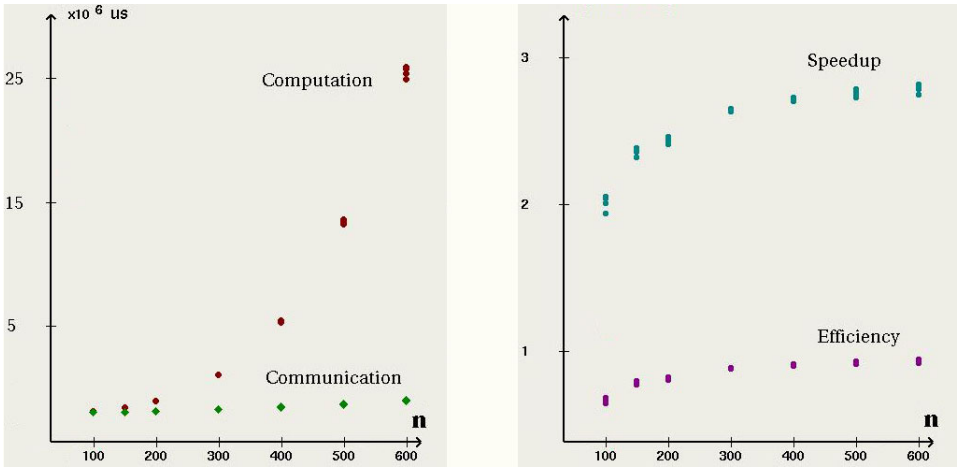

Figure 1.8: Performance of Parallel Matrix Multiplication in function of worksize (with $p = 3$).

faster, as it is proportional to n^3. The net result is that the impact of the overhead decreases and, consequently, the efficiency increases. With $p = 3$, the ideal speedup is 3. The results show that with increasing p, the speedup asymptotically approaches the ideal speedup. For large matrices, the communication overhead can be neglected and an ideal speedup can be achieved. Software granularity is proportional to n.

The performance results for matrix multiplication are typical for a lot of parallel programs: overhead increases with p so that speedup decreases and overhead relatively decreases with increasing work size W. Applications with a computational part that increases faster than the communication as a function of W are appropriate for parallelization.

The influence of other system parameters, such as clock frequency and memory size, or application parameters, such as the datatype used for the datastructure, can be studied similarly.

1.2 Tool

Between 2000 and 2004 a tool for the performance analysis of parallel application was developed at the parallel lab of the VUB, by Jan Lemeire, John Crijns and Andy Crijns [Lemeire et al., 2004][Lemeire, 2004]. The

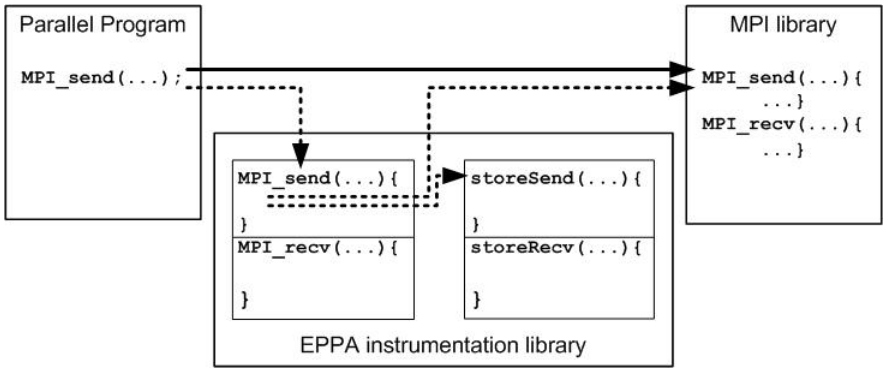

Figure 1.9: The MPI profiling interface to intercept MPI calls (dashed lines).

tool is called **EPPA**, which stands for **E**xperimental **P**arallel **P**erformance **A**nalysis. Experimental data is gathered through the profiling of parallel runs. The post-mortem analysis is based on the performance metrics developed in the previous section. The goal of EPPA is to support the developer of parallel applications with an easy and clear analysis.

1.2.1 EPPA

The EPPA analysis is based on traces of the execution of a parallel program: every phase is identified together with the important characteristics of each phase. The tracing is performed automatically when the program uses MPI [Snir et al., 1996]. MPI, the Message Passing Interface, defines the standard for writing parallel programs based on *message passing*. Parallel processes communicate by exchanging messages. The other approach is *shared memory*, according to which memory may be simultaneously accessed by the different parallel processes.

The MPI profiling interface makes it easy to intercept the MPI calls made by a parallel program. How this works is shown in Fig. 1.9. After linking an MPI program with the EPPA library, each MPI call, before going the MPI library, is intercepted by the library. The information about the MPI operations and their durations are stored in the EPPA database. By this, EPPA collects information about the communication operations: when messages are send, at what times they arrive, when and how long a process is waiting for an incoming message, etcetera. The time between successive MPI calls is stored as computation phases. Four phases are identified automatically: computation, sending, receiving and idling. The user program

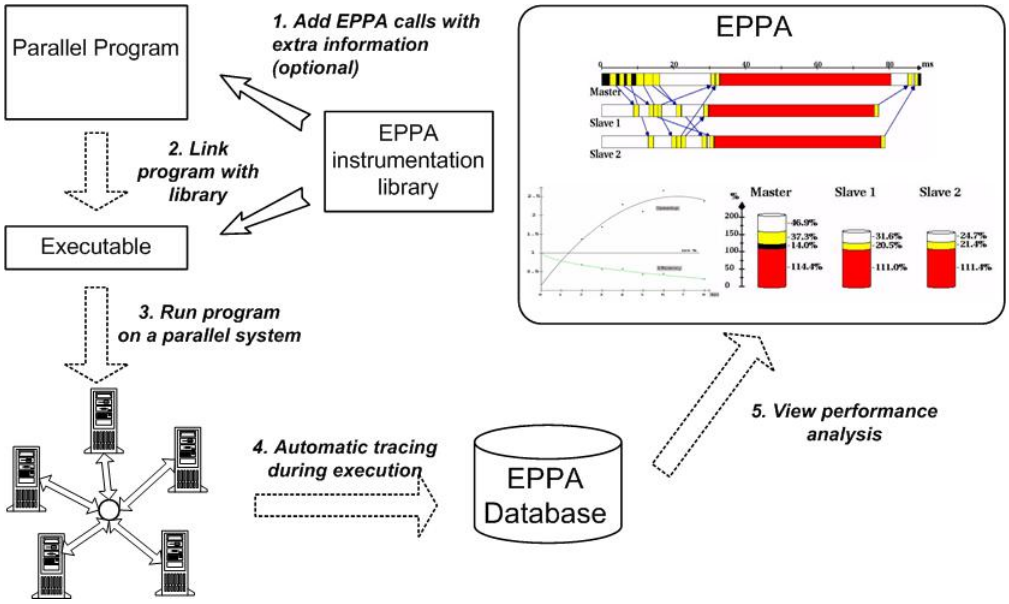

Figure 1.10: Scheme of the EPPA tool.

has only to be linked with the EPPA instrumentation library to activate the tracing of all communication activity. This is shown in Fig. 1.10. Programs using the older PVM library for message-passing should be instrumented manually by adding an EPPA function call after each call to PVM.

The EPPA Tool presents the performance analysis in different views:

- The *timeline* shows the program execution of each process (Fig. 1.4).
- The overall performance gives speedup, efficiency and global overhead ratios (Fig. 1.5).
- The overhead ratios per process (Fig. 1.6).
- The performance variables in function of number of processors p or work size W (Fig. 1.7 and 1.8). Besides the visualization, a regression analysis can be applied on the displayed functions, returning the curve that best fits the data.

Besides the information about the program's communication that is collected automatically, the user is given the possibility to specify additional information. The data collected in this way facilitates the refinement of the analysis. EPPA provides the following options:

- The user can differentiate computational phases. EPPA automatically traces computation phases, as the cycles between two successive MPI calls. But it can not know whether these computations are part of useful work or overhead (control of parallelism). To make this difference, the user can add EPPA calls to specify the role of each computational phase.

- System parameter p and the program's work size parameter W are added for each experiment. Besides these, the user can add other system or program parameters. The performance variables can then be studied as a function of these parameters.

- The size of each message in bytes is automatically recorded by EPPA. The communication performance can be studied as a function of message size. Additionally, the user can specify the number of *quantums* that are processed and communicated in each phase. The definition of a quantum depends on the specific program. For a matrix operation, a quantum is an element of the matrix. EPPA provides the functionality to visualize performance metrics in function of the number of quantums.

- Finally, the main part of an algorithm usually is the repetitive execution of a basic *operation*. For matrix multiplication, the main computations consist of a multiplication and addition. The number of basic operations can also be passed to EPPA and studied in detail.

1.3 Summary of Chapter

For optimizing performance, the *lost processor cycles* of a parallel execution should be minimized. The impact of an overhead (a source of lost cycles) on the speedup is quantified by its ratio with the sequential runtime. When considering the main phases of a parallel execution, overheads can be classified according to 4 types: control of parallelism, communication, idling and parallel anomaly. The tool EPPA can be used to automatically trace these phases during a run of a parallel program. The execution is visualized together with the different performance metrics. The user can augment the analysis by providing additional information about the parallel program.

Bibliography

Rosa M. et all. Badia. Dimemas: Predicting mpi applications behavior in grid environments. In *Workshop on Grid Applications and Programming Tools (GGF8)*, 2003.

J. Mark Bull. A hierarchical classification of overheads in parallel programs. In Innes Jelly, Ian Gorton, and Peter R. Croll, editors, *Software Engineering for Parallel and Distributed Systems*, volume 50 of *IFIP Conference Proceedings*, pages 208–219. Chapman & Hall, 1996. ISBN 0-412-75740-0.

Mark Crovella and Thomas J. LeBlanc. Parallel performance using lost cycles analysis. In *SC*, pages 600–609, 1994.

Vipin Kumar and Anshul Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing (special issue on scalability)*, 22(3):379–391, 1994.

Jan Lemeire. Documentation of eppa tool (experimental parallel performance analysis). `http://parallel.vub.ac.be/eppa`, 2004.

Jan Lemeire, Andy Crijns, John Crijns, and Erik F. Dirkx. A refinement strategy for a user-oriented performance analysis. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *PVM/MPI*, volume 3241 of *Lecture Notes in Computer Science*, pages 388–396. Springer, 2004. ISBN 3-540-23163-3.

Inc NetPredict. Common mistakes in performance analysis, white paper. NetPredict Inc, 2003.

Cherri M. Pancake. Applying human factors to the design of performance tools. In Patrick Amestoy, Philippe Berger, Michel J. Daydé, Iain S. Duff, Valérie Frayssé, Luc Giraud, and Daniel Ruiz, editors, *Euro-Par*, volume 1685 of *Lecture Notes in Computer Science*, pages 44–60. Springer, 1999. ISBN 3-540-66443-2.

Sekhar R. Sarukkai, Jerry Yan, and Jacob K. Gotwals. Normalized performance indices for message passing parallel programs. In *ICS '94: Proceedings of the 8th international conference on Supercomputing*, pages 323–332, New York, NY, USA, 1994. ACM Press. ISBN 0-89791-665-4. doi: http://doi.acm.org/10.1145/181181.181548.

Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.

Michael R. Steed and Mark J. Clement. Performance prediction of pvm programs. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pages 803–807, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7255-2.

Harold S. Stone. *High-performance computer architecture (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. ISBN 0-201-51377-3.

Hong Linh Truong and Thomas Fahringer. SCALEA: A performance analysis tool for distributed and parallel programs. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par*, volume 2400 of *Lecture Notes in Computer Science*, pages 75–85. Springer, 2002. ISBN 3-540-44049-6.

Zoltan Balaton Zsolt Nemeth, Gabor Gombas. Performance evaluation on grids: Directions issues and open problems. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'04) February 11 - 13, 2004, A Coruna, Spain*, 2004. URL `citeseer.ist.psu.edu/nemeth04performance.html`.