Fourteenth International Conference on Domain Decomposition Methods
Editors: Ismael Herrera , David E. Keyes, Olof B. Widlund, Robert Yates ©2003 DDM.org

37. Generic parallel multithreaded programming of domain decomposition methods on PC clusters

A.S. Charão[1], I. Charpentier[2], B. Plateau[3], B. Stein[1]

1. Introduction. Since the early implementations of domain decomposition methods on parallel computers, programming techniques and computer architectures have significantly evolved. Due to the increasing availability of powerful microprocessors and high-speed networks, clusters of PCs become an attractive, low cost option for high-performance computing. While this trend makes parallel computing much more accessible, developing efficient programs for these architectures needs in general some expertise in parallel programming. In this paper we focus our attention on generic and object-oriented programming techniques for parallel implementation of domain decomposition methods. These techniques are central to Ahpik, our multithreaded programming tool targeted to such families of numerical methods.

Parallel efficiency is not the only goal when developing applications based on domain decomposition: flexibility and portability of parallel codes are also essential to preserve investments made in their development. Some existing parallel libraries, like PSPARSLIB[12] and PETSc[4], offer a compromise among all these goals. They provide a large set of linear equation system solvers which use domain decomposition methods as parallel preconditioners. Ahpik differs from these tools as it is rather an experimental library for doing research and experimentation involving parallel computing and numerical methods. It does not aim to provide a fairly complete set of algorithms and data structures for numerical computations, however it has some "plug-in" points allowing for easy integration of such components.

Ahpik offers a highly modular support for developing parallel domain decomposition solvers where numerical aspects are completely decoupled from parallel implementation details. This tool includes patterns of parallel coordination (task identification, communication and synchronization) that can be reused to implement different domain decomposition methods for the resolution of a PDE problem. These patterns can be viewed as drivers for parallel iterative computations, provided as C++ templates that must be "filled-in" with computations characterizing each numerical method. Performance results obtained with Ahpik were published in [2, 3, 1]. In this paper we concentrate on qualitative aspects of our approach. To do so, our evaluation criteria are based on the visualization of the parallel, multithreaded execution of some domain decomposition methods within different scenarios (good/bad workload distribution, synchronous/asynchronous iterations).

The outline of the paper is as follows. Our experience with a generic programming approach for parallel implementation of a large spectrum of domain decomposition methods is reported in section 2. Execution traces of a multithreaded driver are presented in section 3.

2. Genericity. A key idea within Ahpik is the representation of a parallel program as a graph of interacting tasks, namely *internal* and *interface* tasks. As far as domain decomposition methods are concerned, internal tasks correspond to local computations, *i.e.* computations that require only data local to a sub-domain (solving the linear system associated to a sub-domain for instance). Besides, interface tasks carry out operations requiring data from neighboring sub-domains.

[1]Departmento de Eletrônica e Computação (DELC-UFSM), Universidade Federal de Santa Maria, Brazil, andrea@inf.ufsm.br, benhur@inf.ufsm.br

[2]Projet Idopt (CNRS-INRIA-UJF-INPG), Laboratoire de Modélisation et Calcul (LMC-IMAG), Grenoble, France, Isabelle.Charpentier@imag.fr

[3]Projet Apache (CNRS-INRIA-UJF-INPG), Laboratoire Informatique et Distribution (ID-IMAG), Brigitte.Plateau@imag.fr

In previous works[2, 3, 1] Ahpik was presented as a three level library: the domain decomposition level containing specificities of some mainstream methods (Schwarz[13], FETI[8], Mortar[5]), the parallel drivers level (fixed-point, conjugate gradient and generalized minimum residual schemes), and the kernel of Ahpik which is based on a communicating threads library named Athapascan[6]. In this paper, Ahpik is described in an enhanced manner in order to point out the genericity of its parallel patterns and its abilities in the context of experimental studies.

2.1. Parallel patterns. While mathematical arguments differ significantly from one domain decomposition method to another, the parallel behavior of such methods is generally the same: it is dictated by the iterative, intrinsically synchronous, resolution procedure. In a traditional parallel programming method, the internal task (local computation) and three interface tasks (send, receive, interface computations) corresponding to a subdomain are gathered and executed sequentially on a unique UNIX process. Using multithreading, these four independent tasks may be assigned to different threads that are gathered on a UNIX process. There is no real order between them until one applies a parallel pattern for scheduling these tasks. This property eases the balancing of the computations and the implementation of asynchronous algorithms that allow for masking communication overhead. Such a programming model is particularly interesting when using clusters of shared-memory multiprocessors[10].

The assignment of the tasks onto threads depends on the iterative method one chooses to drive the parallel resolution of the domain decomposition problem. For the sake of simplicity, we present the *fixed point* parallel driver (coded in Ahpik) within a domain decomposition into two subdomains. Since we work with threads, we distinguish *read* and *write* tasks that are carried out through the memory shared by the threads associated to the same subdomain, from *send* and *receive* tasks that require communication through message passing. Threads are denoted using letters and numbers, the latter is equal to 1 (resp. 2) for threads of subdomain 1 (resp. 2), and equal to 0 for the thread devoted to the verification of the stopping criterion related to the convergence of the scheme. According to these definitions, threads perform the following tasks:

1. *Internal Thread* `IT1`: computes local PDE solutions, writes data for `ST1`, reads data from `ST1`, computes a local error and sends it to `CT0`, and finally reads data from `CT1`,
2. *Send Thread* `ST1`: reads data from `IT1`, sends them to `RT2`, reads data from `RT1`, computes and updates interface contributions and writes them for `IT1`,
3. *Receive Thread* `RT1`: receives data from `ST2` and writes them for `ST1`,
4. *Convergence Thread* `CT1`: receives data from `CT0` and writes them for `IT1`,
5. *Convergence Thread* `CT0`: receives data from `IT1` and `IT2`, computes a global error as specified by the user and sends it to `CT1` and `CT2`.

Send and receive tasks are assigned to different threads: this allows for overlapping communication with computations, because no sequential order is imposed between send and receive operations. For example, thread RT1 may receive data from ST2 before IT1 has finished its computations. There exist other manners of achieving that (for example using non-blocking send and receive primitives), but multithreading is a more elegant alternative.

When the *fixed point* driver is a synchronous one, these tasks are ordered uniquely. This may be observed on figure 2.1. In that picture, we draw the activity of threads, during one iteration, with respect to the execution time: a colored box signifies that the thread is active, a white box indicates the thread is blocked, waiting for data. Threads assigned to the same processor communicate through shared memory: a red arrow represents a synchronization point where one thread must wait for data made available by another. Message passing (blue arrows) is used to exchange data between threads running on different processing nodes. For the sake of clarity, the size of colored boxes corresponding to interface tasks has been enlarged

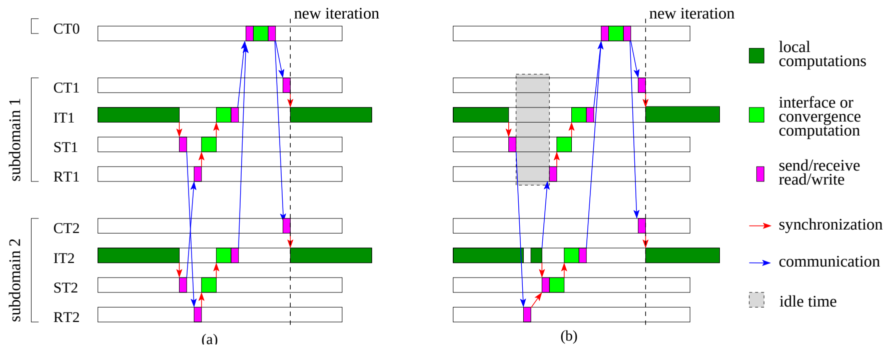

Figure 2.1: Theoretical traces for a synchronous "fixed point" parallel driver: (a) well-balanced workload (b) non-balanced workload

in order to avoid the superposition of blue and red arrows as it happens in actual execution traces presented in section 3.

Figure 2.1.a corresponds to a well-balanced domain decomposition in which the internal tasks of the two subdomains have the same computational cost. One observes that the two sets of threads have the same activity: the main steps described before are drawn on the scheme. In figure 2.1.b the two internal tasks have different computational costs. Since `IT1` performs less local computations than `IT2`, data are sent by `ST1` before the termination of local computations perfomed by `IT2`. In that case these data may be received by `RT2` quite immediately, this is why a break in the activity of `IT2` can occur[4]. Other steps are similar to those of figure 2.1.a. One observes that the four threads of subdomain 1 are inactive in the shadowed time interval. They wait for data sent by `ST2`, available only at the end of local computations performed by `IT2`. This induces idle times that may be reduced by placing multiple subdomains on each processing node.

An alternate solution relies on asynchronous iterations. As described in [9], an asynchronous scheme may be designed for the Schwarz alternating method. A theoretical trace is proposed in figure 2.2. There are no more synchronization points (no red arrows) between RT and ST threads (resp. CT and IT threads) because ST threads (resp. IT) do not wait for data made available by their RT (resp. CT) counterparts. In practice, ST and IT threads simply read data from shared memory without concern on the moment these data have been updated. As a consequence, there is always an active thread at any time.

On our trial trace, one observes that `IT1` is performing twice the same computation (first two iterations) because it uses the same interface data. The latter are updated when `IT2` has finished its first iteration. In more general situations (large number of subdomains), this problem is not so glaring because some interface data are usually updated before a new iteration begins.

2.2. An experimental library for domain decomposition methods. Ahpik is a generic parallel multithreaded environment that allows for the implementation of domain decomposition methods. We have been using generic programming facilities of the C++ programming language to allow users of Ahpik for a rather easy modification of the library with respect to the PDE problem of interest. This is why we decide to build Ahpik with regards to usual mathematical components. Moreover many "plug-in" points exist for

[4]Such breaks actually depend on the threads package and the operating system, they are not visible in the traces presented on section 3.

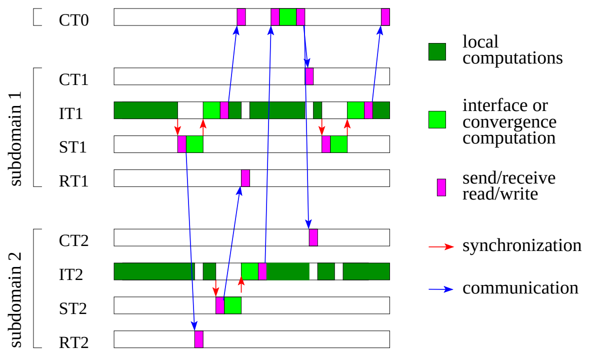

Figure 2.2: Theoretical trace for a non-balanced asynchronous "fixed point" parallel driver

coupling Ahpik with other existing libraries. The following classes and C++ templates are part of Ahpik[5]:

1. mesh capabilities: mesh data structures, mesh partitioning algorithms, subdomain and interface;
2. some discretization methods;
3. domain decomposition methods: additive Schwarz method as well as FETI and mortar methods;
4. local solvers: these are provided by SuperLU[7] for a LU direct solver and IML++[11] for iterative solvers. When appropriate, one may also define a matrix-vector product to perform local computation tasks;
5. generic parallel drivers: fixed point and conjugate gradient.

On one hand, Ahpik classes may be viewed as model classes for experimental solution of a PDE problem by a parallel domain decomposition method. Any user may plug his own C++ library at the level of interest (local solver, domain decomposition, discretization, etc.) as far as the mathematical aspects have been verified. For instance, changing the data structures representing matrices and vectors does not affect the code corresponding to subdomain or interface computations because these functions occur as C++ templates.

It is also possible to develop a new multithreaded parallel pattern that takes into account a preconditioner for example. Whatever the parallel pattern is, the management of the multithreading implementation remains hidden in the Ahpik kernel. Such an implementation allows to use Ahpik in a more general framework: grid-nesting and multigrid schemes are potential targets for our future works.

One the other hand, Ahpik is an experimental library devoted to parallel implementation of domain decomposition methods. It can be viewed as a set of trial problems (Laplace equation, various DDM, ...) for the evaluation of parallel programming alternatives. Developing new strategies to deal with parallelism only affects the kernel of Ahpik: the trial applications included in the object-oriented library are reusable.

3. Visualizing the execution of parallel drivers. Ahpik generates execution traces compatible with the post-processing tool Pajé[14]. This allows to make clear the role of each thread and the interactions between threads. Execution traces we present in this section are relative to a fixed point parallel driver again. The choice of this method against a

[5]More information on these components along with examples of their utilization will be included in the Ahpik distribution: `http://www.inf.ufsm.br/ahpik`

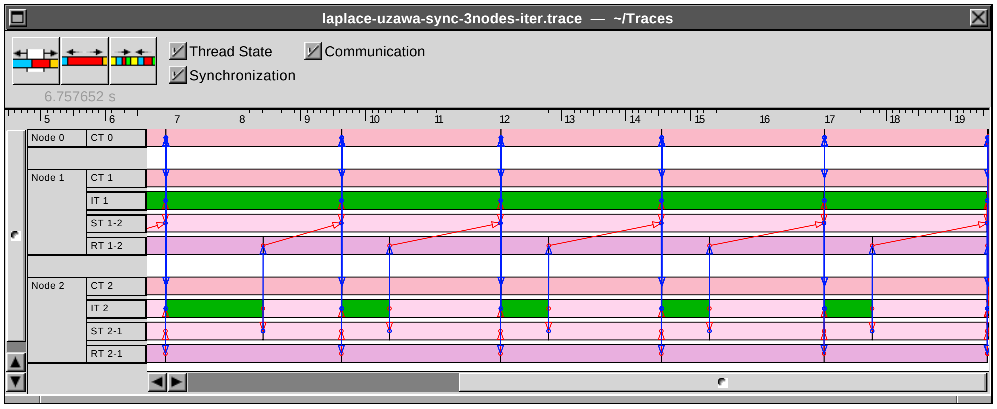

Figure 3.1: Synchronous fixed point driver, iterative local solver

conjugate gradient method is done for the sake of clarity of the drawings: the former induces a unique global synchronization point per iteration (global error computation), while the latter requires two global synchronizations (descent step and global error computation).

The main window of Pajé provides a space-time diagram which shows the activity of threads running on each processing node. As seen in section 2.1, Ahpik uses a set of threads for each subdomain. A single node can deal with multiple sets of threads *i.e* multiple subdomains. In the following, all executions are realized on 3 processing nodes, the first one checking the global convergence criterion of the domain decomposition method only.

Therefore node 0 runs a single thread (CT0) while nodes 1 and 2 run the sets of threads devoted to subdomain computations (CT, IT, ST, RT). The thread activity along the iterations is represented by a horizontal bar which is either green when the thread is working or pink when the thread is waiting for data. Two kinds of arrows are used to represent synchronization points. Red ones are synchronization between threads associated to the same subdomain (synchronized access to shared data) whereas blue ones show communication phases (message passing). The problem solved is the Laplace equation applied in rectangular domains, the geometric decompositions are described gradually.

3.1. General behavior of synchronous drivers. Figure 3.1 points out idle times that may appear when dealing with synchronous parallel drivers. For this experiment we used the domain decomposition of a rectangular domain into two well-balanced subdomains. Local computations are performed using a conjugate gradient solver which converges faster for subdomain 2 than for subdomain 1. The same behavior would have been observed using either a LU solver on a non-balanced domain decomposition or different discretization methods on each side of the interface. There are several ways of reducing idle times. Two of them, assignment of several process to a processor and implementation of asynchronous schemes, are discussed below. Solutions depending on dynamic load-balancing techniques will be discussed in a future work.

One of the key points of this work lies on the genericity of parallel drivers, which are completely independent of computations characterizing each domain decomposition method. As said before, the execution trace shows the behavior of a parallel driver. As a matter of fact, a Schwarz method or a FETI method applied to a domain decomposition in vertical stripes lead to the same kind of execution trace, the difference being in the computational cost and the number of iterations.

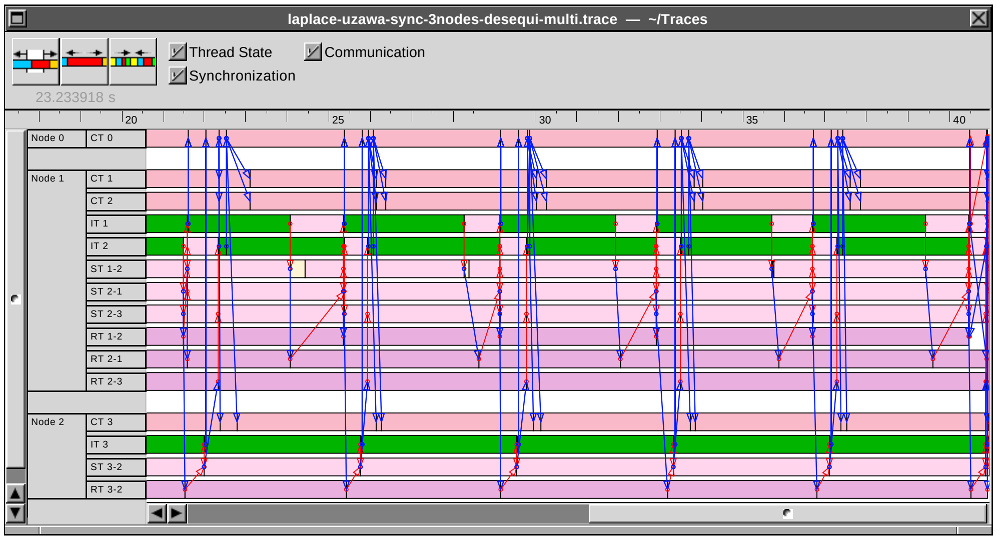

Figure 3.2: Synchronous fixed point driver, 2 subdomains assigned to node 1.

3.2. Reducing idle times. In MPI parallel programming, diminution of idle times can be achieved by placing multiple Unix processes (each one corresponding to a subdomain) to a processing node. A similar solution may be adopted for threaded programming. In that case a processor deals with a unique Unix process. The latter manages multiple sets of threads, each one being associated to a subdomain. Such a balancing method is presented in figure 3.2 for a decomposition of a rectangle into 3 subdomains (vertical bands). This execution trace, as well as others presented before, was generated on uniprocessor nodes. The first two (smaller) contiguous subdomains are assigned to node 1, while node 2 works on a single band. Even though subdomains have different computational costs (the first two subdomains are smaller than the third), one notices that the workload is well distributed over the processing nodes. Indeed, each node always has at least one active thread at any time interval. The interleaving of active threads on node 1 is due to the concurrent computation of two neighboring subdomains. When running the same experiment on 3 multiprocessor (SMP) nodes, the aspect of this execution trace changes for node 1 because the two subdomains can be treated not concurrently but in parallel.

Figure 3.3 presents an execution trace corresponding to an asynchronous parallel driver (fixed point) applied with the additive Schwarz method. The rectangular domain is decomposed in two overlapping (non-balanced) bands. One clearly observes that no idle times occur for this execution. Besides, no more red arrows representing synchronizations occur between RT and ST threads (resp. CT and IT threads). Indeed, in such asynchronous methods, threads devoted to local computations do not block, they do not wait for data. As a consequence, the overall iterative procedure is less structured and some processors may perform more iterations than others. In this execution trace, we can assume that each arrow arriving at a receive thread (the fourth thread on each node) indicates the beginning of a new iteration for each subdomain. Therefore node 1 performs 5 iterations while node 2 performs only 3. As predicted in theoretical traces, a same local computation can be performed twice. In practice, the use of asynchronous drivers could be interesting when the decomposition involves a larger number of subdomains distributed over non-homogeneous processing nodes.

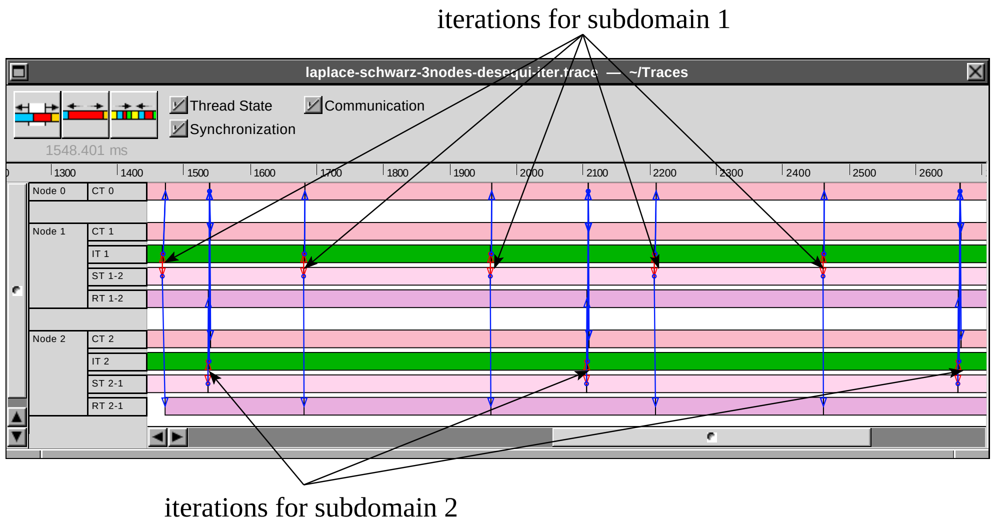

Figure 3.3: Asynchronous fixed point driver.

REFERENCES

[1] A. B. Abdallah, A. S. C. ao, I. Charpentier, and B. Plateau. Ahpik: A parallel multithreaded framework using adaptivity and domain decomposition methods for solving PDE problems. In R. H. J. P. D. K. Y. K. N. Debit, M. Garbey, editor, *13th International Conference on Domain Decomposition Methods*, pages 295–301, 2000.

[2] A. C. ao, I. Charpentier, and B. Plateau. A framework for parallel multithreaded implementation of domain decomposition methods. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and H. J. Sips, editors, *Parallel Computing: Fundamentals and Applications*, pages 95–102. Imperial College Press, 2000.

[3] A. S. C. ao, I. Charpentier, and B. Plateau. Programmation par objet et utilisation de processus légers pour les méthodes de décomposition de domaine. *Technique et Science Informatiques*, 5(19), 2000.

[4] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. *PETSc 2.0 User Manual*. Argonne National Laboratory, http://www.mcs.anl.gov/petsc/, 1997.

[5] C. Bernardi, Y. Maday, and A. T. Patera. A new non conforming approach to domain decomposition: The mortar element method. In H. Brezis and J.-L. Lions, editors, *Collège de France Seminar*. Pitman, 1994. This paper appeared as a technical report about five years earlier.

[6] J. Briat, I. Ginzburg, M. Pasin, and B. Plateau. Athapascan runtime : Efficiency for irregular problems. In *Proceedings of the Europar'97 Conference*, pages 590–599. Springer Verlag, august 1997.

[7] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915–952, 1999.

[8] C. Farhat and F.-X. Roux. A Method of Finite Element Tearing and Interconnecting and its Parallel Solution Algorithm. *Int. J. Numer. Meth. Engrg.*, 32:1205–1227, 1991.

[9] R. Guivarch and P. Spiteri. Implantation de méthodes de sous-domaines asynchrones avec PVM et MPI sur l'IBM-SP2. *Calculateurs Parallèles*, 10(4):431–438, 1998.

[10] E. L. Lusk and W. W. Gropp. A taxonomy of programming models for symmetric multiprocessors and SMP clusters. In *Proceedings of Programming Models for Massively Parallel Computers*, pages 2–7, 1995.

[11] R. Pozo et al. IML++ WWW home page, 1997.

[12] Y. Saad and A. V. Malevsky. PSPARSLIB: A portable library of distributed memory sparse iterative solvers. In *Proceedings of Parallel Computing Technologies (PaCT-95)*, 1995.

[13] H. A. Schwarz. *Gesammelte Mathematische Abhandlungen*, volume 2, pages 133–143. Springer, Berlin, 1890. First published in Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich, volume 15, 1870, pp. 272–286.

[14] B. O. Stein, J. C. de Kergommeaux, and P.-E. Bernard. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing*, 26:1253–1274, 2000.