# Algorithmic Issues for (Distributed) Heterogeneous Computing Platforms

V. Boudet, F. Rastello and Y. Robert
LIP, UMR CNRS–ENS Lyon–INRIA 5668
Ecole Normale Supérieure de Lyon
F - 69364 Lyon Cedex 07

**Abstract** *Future computing platforms will be* distributed *and* heterogeneous. *Such platforms range from heterogeneous networks of workstations (NOWs) to collections of NOWs and parallel servers scattered throughout the world and linked through high-speed networks. Implementing tightly-coupled algorithms on such platforms raises several challenging issues. New data distribution and load balancing strategies are required to squeeze the most out of heterogeneous platforms. In this paper, we first summarize previous results obtained for heterogeneous NOWs, dealing with the implementation of standard numerical kernels such as finite-difference stencils or dense linear solvers. Next we target distributed collections of heterogeneous NOWs, and we discuss data allocation strategies for dense linear solvers on top of such platforms. These results indicate that a major algorithmic and software effort is needed to come up with efficient numerical libraries on the computational grid.*

*Keywords:* meta-computing, heterogeneous networks, scheduling, mapping, finite-difference stencils, numerical libraries.

## 1 Introduction

The future of computing is best described by the key-words *distributed* and *heterogeneous*. At the low end of the field of distributed and heterogeneous computing, heterogeneous networks of workstations or PCs are ubiquitous in university departments and companies, and they represent the typical poor man's parallel computer: running a large PVM or MPI experiment (possibly all night long) is a cheap alternative to buying supercomputer hours. The idea is to make use of all available resources, namely slower machines *in addition to* more recent ones. At the high end of the field, linking the most powerful supercomputers of the largest supercomputing centers through dedicated high-speed networks will give rise to the most powerful computational science and engineering problem-solving environment ever assembled: the so-called *computational grid* [10]. Providing desktop access to this "grid" will make computing routinely parallel, distributed, collaborative and immersive.

Whereas the architectural vision is clear, the software developments are not so well understood. Even at the low end of the field, the programmer is faced with several challenges. The major limitation to programming heterogeneous platforms arises from the additional difficulty of balancing the load when using processors running at different speeds. Distributing the computations (together with the associated data) can be performed either dynamically or statically, or a mixture of both. Some simple schedulers are available, but they use naive mapping strategies such as master-slave techniques or paradigms based upon the idea *"use the past predict the future"*, i.e. use the currently observed speed of computation of each machine to decide for the next distribution of work [7, 1]. Furthermore, data dependences may well lead to slowing the whole computing process down to the pace of the slowest processor, as examples taken from standard linear algebra kernels demonstrate (see below). In fact, extensions of parallel libraries such as ScaLAPACK are not yet available. A major algorithmic effort must be undertaken to tackle heterogeneous computing resources. Block-cyclic distribution is no longer enough: there is a challenge in determining a trade-off between the data distribution parameters and the process spawning and possible migration policies. Redundant computations might also be necessary to use a heterogeneous cluster at its best capabilities. At the high end of the field, the first task is to logically assemble the distributed computer: given the network infrastructure, configure the distributed collection of machines to which access is given. Software of this category includes low-level communication protocols that enable distributed resources to efficiently communicate. Extensions of PVM and MPI such as Plus [13], PACX-MPI[8] or PVMPI [9] are needed. Once this software layer is built, the user must be provided with meta-computing tools and libraries, i.e software that is able to split the computation into tasks that will be dynamically allocated to the different resources available. Current strategies to allocate tasks to resources are very simple (similar to those previously discussed). We discuss more elaborate strategies in this paper.

The ultimate goal would be to use the computing resources remotely and transparently, just as we do with electricity: without knowing where it comes from. Before reaching this ambitious goal, there are several layers of software to be provided. Lots of efforts in the area of building and operating meta-systems are targeted to infrastructure, services and applications. Not so many efforts are devoted to algorithm design and programming tools, while (we believe) they represent the major conceptual challenge to be tackled.

In this paper, we first target heterogeneous NOWs, and we report both theoretical results and PVM experiments on the implementation of standard numerical kernels such as finite-difference stencils or dense linear solvers. It turns out that refined static allocation strategies lead to good results for such kernels. Owing to these strategies, quite satisfactory results can be obtained with little software efforts on a single heterogeneous NOW. Next we target distributed collections of heterogeneous NOWs, and we discuss both static and dynamic data allocation strategies for dense linear solvers on top of such platforms. These results indicate that a major algorithmic and software effort is needed to come up with efficient numerical libraries on the computational grid.

## 2 Three case studies on a heterogeneous NOW

In this section we demonstrate that static allocation strategies are the key to efficiently implementing tightly-coupled algorithms on a dedicated heterogeneous NOW. Dynamic strategies are likely to prove useful in the following two situations: (i)when targeting non-dedicated workstations with unpredictable workload and possible failures; and (ii)when programming a large application made up of several loosely-coupled tasks. However, when implementing a tightly-coupled algorithm (such as a linear system solver), carefully tuned scheduling and mapping strategies are required. At first sight, we may think that dynamic strategies are likely to perform better, because the machine loads will be self-regulated, hence self-balanced, if processors pick up new tasks just as they terminate their current computation. However, data dependences, communication costs and control overhead may well lead to slow the whole process down to the pace of the slowest processors. On the other hand, static strategies will suppress (or at least minimize) data redistributions and control overhead during execution.

To be successful, static strategies must obey a more refined model than standard block-cyclic distributions: such distributions are well-suited to processors of equal speed but would lead to a great load imbalance with pro-

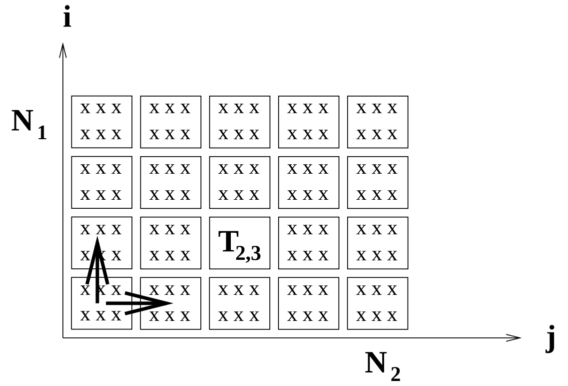


Figure 1: A tiled iteration space with horizontal and vertical dependences.

cessors of different speed. To show that the design of static strategies that achieve a good load balance on a heterogeneous NOW can be achieved for a variety of problems, we briefly sketch three case studies hereafter.

### 2.1 Tiling

We start with a simple example where dependences prevent dynamic strategies to reach a good efficiency: consider a tiled computation over a rectangular iteration space as represented in Figure 1 (see [11] for further information on tiling).

There are $p$ available processors, numbered from 1 to $p$, which are assigned columns of tiles. When targeting a homogeneous NOW, a natural way to allocate tile columns to physical processors is a pure cyclic allocation [11]. For heterogeneous NOWs, we use a refined periodic allocation (see [5]) which proves efficient both theoretically and experimentally: we have run several MPI experiments. We point out that purely cyclic allocations do not perform well, while they would be the outcome of a greedy master-slave strategy. Indeed, processors will be allocated the first $p$ columns in any order. Re-number processors according to this initial assignment. Then throughout the computation, $P_j$ will return after $P_{j-1}$ and just before $P_{j+1}$ (take indices modulo $p$) , because of the dependences. Hence computations would only progress at the speed of the slowest processor.

In a word, our solution consists in allocating panels of $B$ tile columns to the $p$ processors in a periodic fashion. Inside each panel, processors receive an amount of columns inversely proportional to their speed. Within each panel, the work is weell-balanced, and dependences do not slow down the execution. See [5] for details.

### 2.2 Finite-difference stencil computations

In this example we study a tiled version of the Fermi-Pasta-Ulam one-dimensional relaxation problem [14]. Let $x_i^t$ denote the coordinate of the $i^{th}$ molecule at time $t$. According to the model, we have

$$\begin{aligned} x_i^t &= 2x_i^{t-1} + x_i^{t-2} + \frac{(dt)^2}{m} f(x_{i-1}^{t-1}, x_i^{t-1}, x_{i+1}^{t-1}) \\ &= g(x_{i-1}^{t-1}, x_i^{t-1}, x_{i+1}^{t-1}, x_i^{t-2}) \end{aligned}$$

After some compiler transformations, we end up with a a tiled iteration space whose shape is a parallelogram and whose dependence vectors are the pair $(0,1)^t$ and $(1,0)^t$. This turns out to be a very general situation when solving finite-difference problems.

Our solution is similar to that for the rectangular-shaped problem: columns of tiles are distributed to the processors. More precisely, the panel size $B$ is chosen such that the load is best balanced and that some idle time is not created by too large blocks of columns. However, because the domain has been skewed, dependences further constrain the problem, and there is a technical condition to enforce so that no processor is kept idle. See [4] for more details.

### 2.3 Dense linear system solvers

The last kernels that we deal with are the (blocked) LU and QR decompositions taken from the ScaLapack library [6]. We report both theoretical results and PVM experiments in a companion paper [3]. Static strategies are much more efficient than dynamic ones because

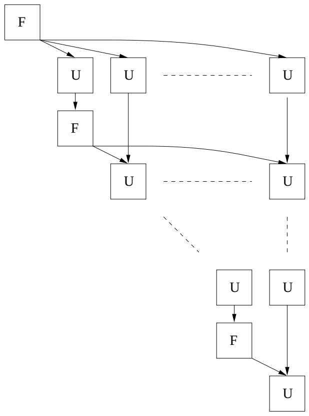


Figure 2: The task graph of LU and QR decompositions.

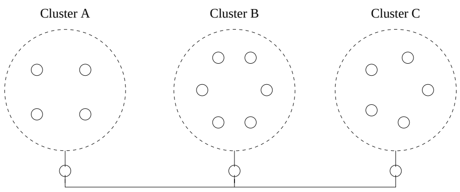


Figure 3: Modeling a 2-deep grid.

they dramatically reduce the amount of data redistributions during the execution.

First we find an optimal allocation of the column blocks, i.e. such that each phase of updates (a row of task labeled U in Figure 2) is best balanced. In this solution the only communications are the broadcasts of the pivot block at each phase (after each task labeled F in Figure 2). Then we address the more complicated problem where the pivot block factorization is taken into account for load balancing the block: we propose an algorithm such that each pivot factorization is executed by the fastest processor. This induces a few more communications, but the algorithm is perfectly load-balanced.

# 3 Semi-static strategies for collections of heterogeneous NOWs

Here we target distributed collections of heterogeneous NOWs, and we discuss data allocation strategies for dense linear solvers on top of such platforms. These results indicate that a major algorithmic and software effort is needed to come up with efficient numerical libraries on the computational grid. The machine and network models are the following: we hierarchically define a $(d+1)$-deep grid as a network of possibly different $d$-deep grids. Of course a 1-deep grid simply is a heterogeneous NOW. Then a 2-deep grid is a collection of heterogeneous NOWs, where the inter-NOW communication links are assumed to have the same speed, typically one order of magnitude slower than the intra-NOW communication links. For instance two local networks in Europe and in the US may be connected by a slower (non-dedicated) link.

We address the problem of finding an optimal allocation for the LU and QR factorizations on a 2-deep grid (note that Section 4 of [3] deals with the same problem on a single heterogeneous NOW). To this purpose, we assume that in each NOW, a processor is dedicated to handle the communications between NOWs, as shown in Figure 3: there is a (slow) serial link between any pair of NOWs. To communicate to a processor of the cluster $B$ (in Figure 3), a processor of cluster $A$ has to send its message to the dedicated processor of its own cluster. Then the message will be forwarded to the dedicated processor of cluster $B$; this communication can take place in parallel with independent computations within the clusters $A$ and $B$ Finally, the receiver in cluster $B$ receives its data from the dedicated processor.

Because of the characteristics of the 2-deep grid, we have to increase the granularity of the computations. The basic chunk of data that is allocated to a given NOW is *a panel* of $B$ (consecutive) blocks of $r$ columns, where $r$ is chosen to ensure Level 3 BLAS performance [2] and $B$

is a machine-dependent parameter. The basic idea is to overlap inter-NOW communications (typically the broadcast of a panel) with independent computations. Updating a panel requires $nB^2r^2\tau_a$ units of time, where $\tau_a$ is the elemental computation time. Communicating a panel between NOWs requires $nBr\tau_c$ units of time, where $\tau_c$ is the inter-NOW communication rate. Of course $\tau_c$ is several orders of magnitude greater than $\tau_a$, but setting $B$ large enough (in fact $B \geq \frac{\tau_c}{r\tau_a}$) will indeed permit the desired communication-computation overlap. Note that such an overlap cannot usually be achieved within a single NOW.

### 3.1 Static strategy

We decompose our matrix into panels of size $B$: a panel is a slice of $B$ consecutive column blocks. The size of the panels is the same for all the clusters. The value for $B$ is discussed below. Roughly speaking, the number of panels allocated to each cluster is inversely proportional to its speed: we compute the time needed to update a panel of $B$ column blocks for each cluster. These times are the "cycle-time" of the clusters. For example, consider a cluster $A$ with 3 machines whose cycle-times are 2, 3 and 4, and a cluster $B$ with 3 machines whose cycle-times are 3, 5 and 8. Suppose that the size of the panels is $B = 5$. The optimal allocation for the cluster $A$ is 24232 (meaning that the processor of cycle-time equal to 2 receives the first, third and fifth blocks, starting the numbering from the right, and so on). The optimal allocation for $B$ is 35383. Hence the "cycle-time" for cluster $A$ is 6 and the "cycle-time" for the cluster $B$ is 9.

We distribute the panels as if we had machines with these "cycle-times". In the example, we would use the optimal allocation of [3]with two machines of cycle-times 6 and 9, leading to a periodic allocation of panels as $\ldots|BAABA|BAABA|BAABA$. In fact we can do slightly better and continue the optimized distribution inside each cluster from one panel to the next one. In the example, the allocation of the first five panels is $AAABA$ (as stated above, starting the numbering from the right), and inside the clusters we have the distribution

$B_8B_5B_3B_3B_5|A_3A_2A_4A_2A_3|A_2A_3A_2A_4A_3|$
$B_3B_8B_3B_5B_3|A_2A_4A_2A_3A_2$.

Finally, we can further improve this solution by re-evaluating the "cycle-times" of the clusters. Indeed, in our example, the time needed to compute the second panel of cluster $B$ is 10 and not 9. So to speak, to refine the allocation of the panels we may take the "cycle-times" of the clusters into account on the fly.

As explained above, the size of the panels is chosen so that updating a given panel takes less time than communicating a panel to another cluster. The scheduling corresponds to the look-ahead strategy for the pointwise algorithm [12]. It is illustrated in Figure 4. Each task in Figure 4 represents a panel (and not a single column block as in Figure 2). After the factor task at step 2 is completed, all processors of cluster $B$ gather the current panel on the dedicated processor. The broadcast of the panel can take place while cluster $A$ updates its third panel and cluster $B$ updates its second panel at step 3.

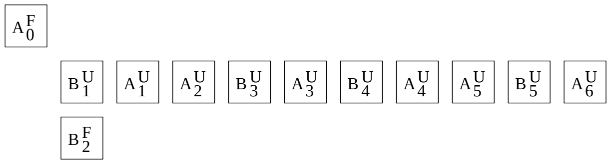


Figure 4: Scheduling with 2 clusters $A$ and $B$. Exponents represent the nature of the tasks (factoring or updating a panel). Indices represent the steps at which the tasks are processed.

### 3.2 Dynamic strategy

As above, we decompose the matrix into panels of size $B$ and we compute the "cycle-time" of the different clusters. We still distribute the panels to the different clusters as explained, using an optimal allocation inside the clusters. However, we decide that the factor task and the preceding update task is always executed

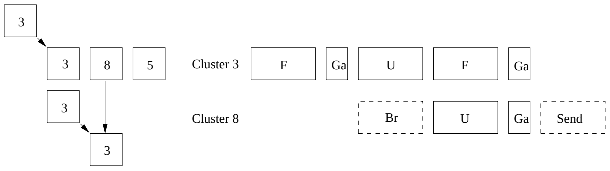


Figure 5: Scheduling the tasks and the communications with the dynamic strategy.

by the fastest cluster. In the initial distribution of panels, we suppress the first two occurrences of the fastest cluster to take into account the factor and update tasks.

For example, if we have 3 clusters of relative speeds 3, 5 and 8, the allocation will be 33|85335385. The two panels on the left correspond to the factor task and its corresponding update. As before, the scheduling strategy is "update, factor and broadcast ASAP". The fastest cluster will indeed compute the next pivot as soon as possible, and broadcast it to the other clusters, before computing its last updates. There are additional communications due to the fact that all the pivot panels must be processed by the fastest cluster, as illustrated in Figure 5 which corresponds to the example above with 3 clusters. The short communications labeled "Ga" are local gathers within a cluster (the dedicated communication processor gathers the panel) whereas the long communications stand for inter-cluster communications, either the receiving of the pivot panel before its factorization by the fastest cluster, or the broadcast of the pivot panel after factoring. The cost of an inter-cluster communication remains remains smaller than the time needed for an update because of the choice of $B$.

These strategies are preliminary: they are intended to balance computations while overlapping communications. Actual experiments are needed to validate our implementation skeletons. However, we can already draw the conclusion that a major software effort is needed. Some basic tools to write the factorization routines, such as the BLAS3 operations, are still there. Some other tools such as the BLACS subroutines need to be extended to cope with several NOWs, using packages such as those in [13, 8, 9]. What seems unavoidable is a change in the philosophy: we would access pointers to local arrays rather than addressing a shared-memory global matrix as in the current ScaLAPACK distribution. Such a major change is a *sine-qua-non* to tackle the implementation of ScaLAPACK on $d$-deep grids, where $d \geq 2$: it does not seem reasonable to emulate a global addressing on a collection of heterogeneous NOWs or parallel servers that are scattered all around the world.

## 4 Conclusion

The main objectives of this paper have been: (i) to survey existing algorithm design methods for heterogeneous platforms, mainly at the low end of the field, i.e. targeting a single heterogeneous NOW; (ii) to demonstrate that refined static allocation strategies are very efficient for a variety of standard computational kernels; nd (iii) to point out that a major algorithmic effort is needed to efficiently implement tightly-coupled applications on meta-systems (collections of NOWs).

We believe that squeezing the most out of meta-computing systems will require to solve challenging algorithmic problems. We insist that the community should tackle these problems very rapidly to make full use of the many hardware resources that already are at its disposal.

## References

[1] F. Berman. High-performance schedulers. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 279–309. Morgan-Kaufmann, 1998.

[2] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and

R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[3] V. Boudet, F. Rastello, and Y. Robert. A proposal for an heterogeneous cluster ScaLAPACK (dense linear solvers). Technical Report RR-99-17, LIP, ENS Lyon, 1999. Available at `www.ens-lyon.fr/LIP`.

[4] P. Boulet, J. Dongarra, F. Rastello, Y. Robert, and F. Vivien. Algorithmic issues for heterogeneous computing platforms. Technical Report RR-98-49, LIP, ENS Lyon, 1998. Available at `www.ens-lyon.fr/LIP`.

[5] P. Boulet, J. Dongarra, Yves Robert, and Frédéric Vivien. Tiling for heterogeneous computing platforms. Technical Report UT-CS-97-373, University of Tennessee, Knoxville, 1997. To appear in the journal *Parallel Computing*.

[6] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. *Computer Physics Communications*, 97:1–15, 1996. (also LAPACK Working Note #95).

[7] Michal Cierniak, Mohammed J. Zaki, and Wei Li. Customized dynamic load balancing for a network of workstations. *Journal of Parallel and Distributed Computing*, 43:156–162, 1997.

[8] Th. Eickermann, J. Henrichs, M. Resch, R. Stoy, and R. Volpel. Metacomputing in gigabit environments: networks, tools and applications. *Parallel Computing*, 24:1847–1872, 1998.

[9] G. Fagg, J. Dongarra, and A. Geist. Heterogeneous MPI application interoperation and process management under PVMPI. In M. Bubak, J. Dongarra, and J. Wasniewski, editors, *Recent advances in PV and MPI*, volume 1332 of *Lectures Notes in Computer Science*, pages 91–98. Springer Verlag, 1997.

[10] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998.

[11] K. Högstedt, L. Carter, and J. Ferrante. Determining the idle time of a tiling. In *Principles of Programming Languages*, pages 160–173. ACM Press, 1997. Extended version available as Technical Report UCSD-CS96-489, and on the WEB at http://www.cse.ucsd.edu/~carter.

[12] J.M. Ortega and C.H. Romine. The ijk forms of factorization methods ii. parallel systems. *Parallel Computing*, 7:149–162, 1988.

[13] A. Reinefeld, J. Gehring, and M. Brune. Communicating across parallel message-passing environments. *Journal of Systems Architecture*, 44:261–272, 1998.

[14] M. Remoissenet. *Waves called solitons*. Springer Verlag, 1994.