Algorithms and Tools for (Distributed) Heterogeneous Computing: A Prospective Report

Jean-Francois Mehaut, Yves Robert

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 5668

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

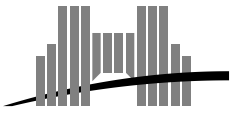
SPI

Algorithms and Tools for (Distributed) Heterogeneous Computing: A Prospective Report

J.F. Méhaut and Y. Robert

August 1999

Research Report N° 1999-36



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr

Algorithms and Tools for (Distributed) Heterogeneous Computing: A Prospective Report

J.F. Méhaut and Y. Robert

August 1999

Abstract

We discuss algorithms and tools to help program and use metacomputing resources in the forthcoming years. Metacomputing with highly distributed heterogeneous environments stands to become a major, if not dominant, method to implement all kinds of parallel applications. In this report, we survey some general aspects of metacomputing (hardware, system and administration issues, as well as the application field). Next we identify some algorithmic issues and software challenges that must be solved to efficiently program and/or transparently use such platforms:

- Data decomposition techniques for cluster computing,
- Granularity issues for metacomputing,
- Scheduling and load-balancing methods,
- Programming models.

We illustrate each of these issues and challenges by the analysis of several case studies: Cluster ScaLAPACK, AppLeS, Globus, Legion, Albatross and Netsolve. We conclude this report by stating some final remarks and recommendations.
mbox

Acknowledgments: This research report is directly issued from the prospective report written by Yves Robert for the "ERCIM Prospective Reports on ICST Research in Europe", an initiative of ERCIM which we gratefully acknowledge.

Keywords: meta-computing, heterogeneous networks, computational grid, distributed environments.

Résumé

Le calcul distribué à hautes performances, encore appellé "metacomputing", constitue aujourd'hui une des approches les plus prometteuses pour implémenter des applications parallèles. Ce rapport présente une synthèse des outils facilitant la programmation et l'utilisation des ressources du métacomputing, et discute les nouvelles techniques algorithmiques à mettre en oeuvre pour utiliser efficacement de telles plateformes.

Dans la première partie, nous introduisons les principes généraux du métacomputing (infrastructures matérielles, systèmes et environnements, administration ainsi que le champ applicatif). Nous détaillons ensuite quelques uns des nouveaux problèmes algorithmiques et les challenges logiciels pour pouvoir programmer efficacement et/ou utiliser de manière transparente ces nouvelles infrastructures:

- Techniques d'allocation de données pour le calcul sur les grappes,
- Problèmes de granularité pour le métacomputing,
- Méthodes pour ordonnancer et réguler la charge de calcul,
- Modèles de programmation.

Nous illustrons cette étude par l'analyse de plusieurs projets: Cluster ScaLAPACK, AppLeS, Globus, Legion, Albatross et Netsolve. Nous concluons ce rapport avec un certain nombre de recommandations sur les directions à suivre dans ce nouveau domaine de recherche.

Mots-clés: "meta-computing", "computational grid", plateforme hétérogène.

Contents

1 Framework

The future of parallel computing is best described by the key-words *distributed* and *heterogeneous*. Making use of distributed collections of heterogeneous platforms is the activity of *metacomputing*. In this section, we briefly survey the field of distributed and heterogeneous (i.e. metacomputing) platforms. Next we sketch some algorithmic issues and software challenges that must be solved to efficiently program and/or transparently use such platforms. Then we state the objectives of this report, whose contents are outlined at end of this section. We stress that we are interested in tightly-coupled high-performance distributed applications rather than loosely-coupled cooperative applications.

1.1 Metacomputing Platforms

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. Computing with a heterogeneous networks of workstations is known as *cluster computing*[1]. Several aspects of cluster computing are covered in the survey books edited by Buyya [19, 20].

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*, which is nicely described in Foster and Kesselman's book [38]. Providing desktop access to this "grid" will make computing routinely parallel, distributed, collaborative and immersive[2].

In the middle of the field, we can think of connecting medium-size parallel servers through fast but non-dedicated links. For instance, each ERCIM institution could build its own specialized parallel machine equipped with application-specific databases and application-oriented software, thus creating a "meta-system". The user is then able to access all the machines of this meta-system remotely and transparently, without each institution duplicating the resources and the exploitation costs.

1.2 Algorithmic and Software Issues

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. 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

[1] See the very interesting discussion on the definition of cluster computing at `www.eg.bucknell.edu/~hyde/tfcc/`. This discussion has been initiated by the IEEE Task Force on Cluster Computing.

[2] See the two special issues of *Communications of the ACM*: November 1997 ("Blueprint for the future of high-performance computing") and November 1998 ("The high-performance computing continuum")

(see Section 4). 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 an 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 are needed to handle distributed collections of clusters: see Section 3.4 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).

A major issue for runtime systems is to address the problem of configuration and performance optimization in a metacomputing environment. Managing resources within this framework is much more complicated than in a local and homogeneous environment. These are challenging issues because of (i) the inherent complexity of networked and heterogeneous systems, (ii) the fact that resources are often identified at runtime and (iii) the dynamic nature of resource characteristics. High-performance computing applications must be able to configure themselves to fit the execution environment, and then adapt their behavior to subsequent changes in resource characteristics.

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.

1.3 Objectives

The objectives of this report are:

- to survey existing algorithm design methods for heterogeneous platforms, both at the low end (collection of workstations) and at the high end (multi-component applications to be run on meta-systems).
- to survey existing technology in the fields of schedulers, compilers, languages and libraries (including numerical libraries, communication libraries and meta-computing libraries) for such platforms.
- and to identify new directions of research.

1.4 Outline

The rest of the report is organized as follows. In Section 2 we briefly discuss some key issues in cluster- and meta-computing: hardware, system and administration issues, as well as the application field. These issues are not covered in further depth in the report, but Section 2 includes several pointers to bibliographical sources, mostly from the Web. In the following sections we deal with all the issues related to the algorithmic of metacomputing:

- Data decomposition techniques for cluster computing (Section 3.1),

- Granularity issue for metacomputing (Section 3.2),
- Scheduling and load-balancing methods (Section 3.3),
- Programming models (Section 3.4).

We illustrate each of these issues and challenges by the in-depth analysis of case studies. Because this report is oriented to algorithmic issues, we devote a long section to the design of Cluster ScaLAPACK (Section 4). Next in Section 5, we cover several metacomputing environments: AppLeS (Section 5.1), Globus (Section 5.2), Legion (Section 5.3), and Albatross (Section 5.4). We give a detailed presentation of NetSolve in Section 6). We conclude this report by stating some final remarks and recommendations in Section 7.

2 General Remarks on Metacomputing

In this section we briefly deal with some aspects of metacomputing that will not be covered in details in this report: hardware, system and administration issues, as well as the application field. We have no objective of comprehensiveness (refer to Chapter 7 of [19] for a detailed survey). Instead, we give our personal view on some important aspects of metacomputing, together with several pointers to external references.

2.1 Tomorrow's Virtual Super-Computer

Metacomputing is likely to represent a major evolution in the future of computing. Today the development of Internet is mainly due to communication tools (e-mail, news) and information services (Web). Functionally, the Web is a huge data-base, which is distributed on the whole set of computers and servers over the earth. In terms of computing resources, the Web (and the associated data-base) is built using (i) a set of disks to store the data; (ii) a network infrastructure enabling a large number of users to access this data. The idea of metacomputing is to use the computing power of the computers linked by the Internet network to execute various applications (numerically-intensive applications first, but many other applications to follow): these applications that used to be implemented a few years ago on dedicated parallel machines. In other words, Internet will slowly evolve into a virtual super-computer, owing to the fast interconnection of the computing resources of all centers equipped with parallel machines or even simply with workstation clusters. Therefore, metacomputing applications will execute on a hierarchical grid, made up with the interconnection of clusters scattered all around the world. In this context, a cluster can be defined as a group of processors communicating either through a shared memory (SMP architecture) or through a communication network whose performances range from 10 Mbits (Ethernet) up to a couple of Gigabits (high-speed networks such as Myrinet, SCI or Giga-ethernet). See the book by Culler and Singh [28] for further details on cluster architectures. A fundamental characteristic of the virtual super-computer is to be composed of a set of strongly heterogeneous and geographically scattered resources.

2.2 Hardware Platforms

Several experiments in metacomputing have been launched in the US in the last few years. These experiments are supported by NSF, NCSA, the Department of Energy, etc; they are coordinated by the National Partnership for Advanced Computational Infrastructure (NPACI)[3]. Several supercomputer centers, national laboratories and universities are linking their resources through high-speed dedicated links, with the main objective to build computational grids. One famous example is the GUSTO (Globus Ubiquitous Supercomputing Testbed) grid which interconnects 17 sites et 330 supercomputers (over 3600 processors) to deliver an aggregated global power in excess of 2 TeraFlops per second! The main objective here is to provide scientific researchers with a simple access to the grids, enabling them to run parallel, distributed, or even cooperative applications.

Right now in Europe, such ambitious projects are not yet launched, due to the lack of supercomputer infrastructures. Intensive parallel computing has been democratized through the use of heterogeneous networks of workstations (HNOWs), which are widely available in research laboratories and enterprises. The computing power of HNOWs is very inefficiently exploited so far: for instance, a simple local network of 50 workstations or PCs has an aggregated computing power

[3] NPACI publishes several newsletters and technical reports , see `http://www.npaci.edu`.

exceeding that of expensive parallel machines. Clearly, the workstations of most local networks are used interactively during business hours, but remain completely idle otherwise. This gives the opportunity for overnight runs of numerically intensive parallel applications over the whole local network. HNOWs represent a cheap and efficient solution for parallel and distributed computing.

A medium way for metacomputing is to rely on a large-scale interconnection of HNOWs which are ubiquitous in universities and research laboratories. Such experiments have been initiated in various locations. For instance in the Netherlands, the DAS platform (Distributed ASCI Supercomputer, `http://www.cs.vu.nl/~bal/das.html`) uses an ATM network to interconnect 200 stations (or PCs) located within four different universities (Amsterdam, Delft, Leiden et Utrecht). The DAS user can start computations over the whole set of available processors, without knowing the exact location of these processors. To avoid the duplication of some resources on each processor (files or data-bases), the runtime support must also provide features to access remote resources. See Section 5.4 for a description of the DAS programming environment. The programming environment plays a critical role in cluster computing: while network technologies are certainly mature enough to build very large HNOWs, substantial efforts are still to be spent on the software side (system, administration, runtime, ...).

2.3 Software

Internet and the Web would not have been so successful without the many simple and easy-to-use tools (mailers, browsers) that have been rapidly made available for the users. The success of metacomputing also depends upon the delivery of a large collection of tools and programming/development environments. In a word, the objective is to provide the users with all layers of software needed to design applications in a simple way, while preserving a reasonable level of performance (far below the goal of squeezing the most out of the infrastructures).

However, during the last 10 years or so, research efforts in the domain of parallel run-time environments have focused on architectures that are strongly homogeneous (processors, memory, networks). Homogeneity has motivated the splendid research on array and loop distribution, parallelizing compilers, HPF constructs, gang scheduling, MPI, ... , which has contributed a lot to the success of high performance computing. As already pointed out, metacomputing platforms are strongly heterogeneous. Run-time environments will have to take this heterogeneity into account. Today we perfectly know how to communicate data between heterogeneous processors, via standard data representation formats (XDR). This is not true for communication protocols: so far, the dominant approach was to rely on a unique communication protocol (TCP/IP) in order to interconnect all computers over the earth. However, this protocol is severely limited for high-performance computing. In a metacomputing framework, a challenge will be the following: within different runtime environments such as MPI and Corba, special-purpose but fast protocols such as BIP for Myrinet (`http://lhpca.univ-lyon1.fr`) or VIA (`http://www.viarch.org`) are to co-exist with the large-scale communication protocol TCP/IP.

2.4 Administrative Issues

The execution of intensive computations on a set of processors distributed across several countries and institutions requires strict rules to define the (good) usage of shared resources. These rules must be accepted and obeyed by all the different users. For instance a simple rule is that a workstation processor cannot be used by other users if the workstation owner is interactively logged on the machine. Similarly, the processors of a cluster could be reserved for those applications which are local to a given research center, and therefore would not be available for external demands. It is

mandatory to establish precise rules for the exploitation and sharing of the resources, as it used to be the case in the first computing centers ... long time ago! The respect of these rules must be guaranteed by the runtime system, together with methods to migrate computations to other sites whenever some local request is raised.

Metacomputing provokes new problems at the system administration level. Currently, system administration is heavily centralized, and mainly consists in managing the accounts and the environments of the users. In a metacomputing context, a major difficulty is to avoid a large increase in the administrative overhead. It is not reasonable to think of a solution where each user would have an account on each machine on the network (that would dramatically reduce the number of both users and machines). Neither is it reasonable to propose a solution where a single user (meta-user) would be the one and only authorized user on the whole set of machines: this would give no protection at all between the data and programs of the different users (in fact, this would be like coming back to a mono-user system). The challenge is to find a tradeoff that does not increase the administrative load while preserving the security of the users.

2.5 Applications

All applications involving parallel computing are good candidates to be ported on a metacomputing platform. On one hand, the severe constraints due to (i) using a network of heterogeneous machines and (ii) relying on current (limited) programming environments, may lead to unacceptable performances, if speed if the main criterion. On the other hand, the other traditional motivation to go parallel, namely increasing the available memory size and storage capacity, remains valid on all metacomputing platforms.

"Classical applications" such as the grand challenges can be ported on metacomputing platforms: see the NPACI reports for an impressive list ranging from molecular dynamics to climate modeling, including all kinds of simulations. For such applications, the granularity of the computations and of the communications is the main factor to performance. Parallelizing these applications on traditional parallel machines is generally conducted with a very fine grain, and even at the innermost loop level. Other strategies must be employed on heterogeneous platforms, taking into account the deep hierarchy between all memory and communication layers. Code coupling is one of the nicest applications for metacomputing, because of (i) its large granularity and (ii) the loose exchanges between the different component applications. See [4] or `http://www.acl.lanl.gov/PAWS` for a widely known example of a code coupling approach. Note that fault tolerance is an issue here, because such codes run for hours or days.

Another class of applications that can easily benefit from heterogeneous platforms is the following: when the application is divided into phases, such as (i) accessing the data; (ii) computing; and (iii) visualizing the results, then a disk server for phase (i), a big mainframe for (ii) and a graphics workstation for (iii) will do the job ideally. A typical example is volume rendering in medical imaging. Other applications can be considered out of the old world of numerical (or scientific) computing: data-bases, decision-support systems, and all kinds of multimedia servers (e.g. see the PPI project at `http://www.infospheres.caltech.edu`).

To summarize in a sentence, the best candidates for metacomputing are all loosely-coupled applications, those which exchange little data between processors. All kinds of decomposition (functional, pipeline, data-parallel) can be brought into play for such applications. The actual challenge is the implementation of tightly-coupled applications.

3 Algorithmic and Programming Aspects

For current parallel applications, most of the work done at the algorithmic level has been targeting platforms which are more or less homogeneous in terms of computing power, memory access, and network latency and bandwidth. Cluster computing or metacomputing implies a major effort in several directions. In this section, we have in mind the implementation of a tightly-coupled application, such as a matrix-matrix product or a dense linear system solver, on a metacomputing platform. How do the man two characteristics of metacomputing, namely heterogenity and distribution, impact the design of such an application? We successively analyze data decomposition techniques, granularity issues, scheduling and load-balancing heuristics, communication protocols, and programming environments.

3.1 Data Decomposition Techniques for Cluster Computing

The Scalable Linear Algebra Package (ScaLAPACK) [9] is a library of high-performance linear algebra routines for distributed-memory message-passing MIMD computers as well as networks or clusters of workstations supporting PVM [40] or MPI [61]. It is a continuation of the LAPACK project, and contains routines for solving systems of linear equations, least squares problems, and eigenvalue problems. ScaLAPACK views the underlying multi-processor system as a rectangular process *grid*. Global data is mapped to the local memories of the processes in that grid assuming specific data-distributions. For performance and load balance reasons, ScaLAPACK uses the two-dimensional block cyclic distribution scheme for dense matrix computations. Inter-process communication within ScaLAPACK is done via the Basic Linear Algebra Communication Subprograms.

ScaLAPACK obeys the block-cyclic distribution paradigm, which is very natural for inexperienced users. Blocked algorithms are needed to squeeze the most out of state-of-the-art processors with pipelined arithmetic units and a multilevel memory hierarchy [31, 23]). Then cyclically distributing blocks to processors nicely regulates the load. This explains why block-cyclic distribution is the preferred layout for data-parallel programs (maybe written in High Performance Fortran [47]). However, such a distribution evenly balances the total workload among all processors *only if* their speeds are all the same. This shows that ScaLAPACK is restricted to collections of homogeneous processors.

Extending ScaLAPACK to heterogeneous clusters turns out to be surprisingly difficult. The naive idea would be to assign to each processor an amount of computations that is inversely proportional to its cycle-time, so that the load remains balanced despite the differences in the processing power of the cluster workstations. Such a good load balancing could be achieved either through dynamic or static strategies. Unfortunately, dynamic strategies may fail because of the many redistributions that they imply throughout the execution of the algorithm. Even worse, data dependences may well slow everything down to the pace to the slowest processor. Furthermore, in a library oriented approach, dynamic strategies are difficult to introduce, because they imply a complicated memory management. What about static strategies then ? Static strategies are less general but may prove useful if the speed of the different workstations is known rather accurately. In that case, sophisticated periodic data distribution strategies are absolutely needed: as already pointed out, block-cyclic distribution is no longer enough. Determining efficient static strategies is possible but involves quite a lot of efforts: see Section 4 for further details.

These are bad news: if designing a matrix-matrix product or a dense linear solver proves a hard task on a heterogeneous cluster, it is likely than more ambitious programs will require even more work ! The next problems are:

1. implementing simple linear algebra kernels on a collection of clusters (extending the platform)
2. implementing more ambitious routines, composed of a variety of more elementary kernels, on a heterogeneous cluster (extending the application)
3. implementing more ambitious routines on more ambitious platforms (extending both).

However, the research community has to tackle these problems for the computational grid to be fully successful!

3.2 Granularity Issues for Metacomputing

In this section we target the first problem stated above: given a distributed collection of heterogeneous NOWs rather than a simple heterogeneous cluster, can we find good data allocation strategies for numerical kernels such as dense linear solvers on top of such platforms?

A first task is to model the underlying architecture. A simple model for a heterogeneous grid is the following: we hierarchically define a $(d+1)$-deep grid as a homogeneous network of heterogeneous 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. A 3-deep grid is a collection of 2-deep grids linked by a slower network, and so on.

We address the problem of finding good data allocation strategies for typical numerical kernels on 2-deep grids. To this purpose, we assume that in each NOW, a processor is dedicated to handle the communications between NOWs, as shown in Figure 1.

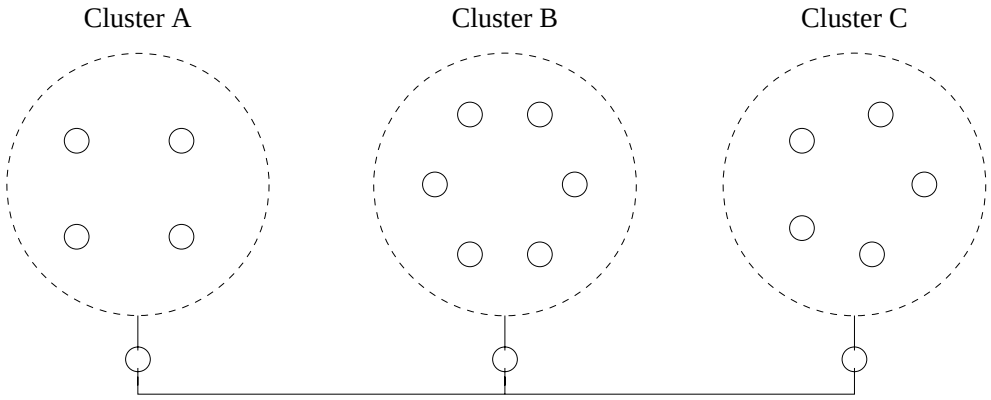


Figure 1: Modeling a 2-deep grid.

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 blocks of r columns, where r is chosen to ensure Level 3 BLAS performance [9] 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 τ_a is the elemental computation time. Communicating a panel between NOWs requires $nBr\tau_c$ units of time, where τ_c is the inter-NOW communication rate. Of course τ_c is several orders of magnitude greater than τ_a, but letting 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.

We report in Section 4.5 several strategies to implement LU and QR factorizations on a 2-deep grid. These strategies are preliminary: they are intended to balance computations while overlapping communications. Actual experiments are needed to validate these 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 described in Section 3.4. What seems unavoidable is a change in the design philosophy of parallel numerical libraries: 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.

These results indicate that a major algorithmic and software effort is needed to come up with efficient numerical libraries on the computational grid.

3.3 Scheduling and Load-Balancing Applications

On classical parallel machines, it is already difficult to trade-off parallelism and communication, even in the presence of unlimited resources. The most powerful scheduling methods (such as Gerasoulis and Yang's dominant sequence clustering [66]) operate in two-steps: in the first step, heuristics are used to grouping tasks into clusters. This clustering operation is made assuming unlimited resources, to simplify things. In the second step, clusters will be allocated to available processors, and the final ordering of the tasks will be computed. The basic rule of the game is that all the tasks of a given cluster will be allocated to the same processor. We can think of a virtual processor per cluster in the first phase, while several clusters will be allocated to physical processors in the second phase. Why is clustering a useful heuristic? Sarkar [58] gives the following justification: "If tasks are scheduled on the same processor on the best possible architecture with unbounded number of processors, then they should be scheduled in the same processor in any other architecture". Sarkar's argument, although not true in every case, is very intuitive, and clustering techniques have been widely explored.

Heterogeneity poses new challenges to scheduling techniques. Of course clustering has no more meaning, because assuming unlimited resources in a heterogeneous environment would lead to using only the fastest processors. Still, building upon old ideas such as critical paths and bottom levels, several sophisticated scheduling heuristics have been developed to cope with heterogeneous resources: for instance Maheswaran and Siegel [52] propose a dynamic remapping of tasks after having computed a first allocation based on critical paths. See also [60, 41, 44, 62, 65, 43] for various scheduling heuristics. A comparative analysis of many scheduling techniques is available in [63, 16, 51].

Another 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 [27, 26, 5]. 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.

As an example of a high-level scheduling and load-balancing tool, we describe the AppLeS project (see `http://www-cse.ucsd.edu/groups/hpcl/apples/apples.html`) in Section 5.1.

3.4 Programming Models

An overview of software support for heterogeneous computing is given by Siegel et al. [59]. There are several programming models that can be exploited.

The first task is to provide an extension of MPI to enable clusters scattered around the world to communicate efficiently. Several projects are under development: MPI_Connect [35], Nexus [36], PACX-MPI [32], MPI-Plus [56], Data-Exchange [33], VCM [57] and MagPIe [46].

Several metacomputing projects are currently building the infrastructure on top of which such extensions of MPI may utilize distributed computing capacity. The most prominent systems are Globus [37] (see `http://www.globus.org`) and Legion [42] (see `http://www.cs.virginia.edu/~legion`). Globus follows a layered approach to building grid infrastructure. The most fundamental layer consists of a set of core services, including resource management, security, and communications that enable the linking and interoperation of distributed computer systems. Toolkits, such as the Message Passing Interface (MPI) for parallel computing and CavernSoft (for collaborative virtual reality) provide higher-level, application-friendly access to Grid services. See Section 5.2 for more information on Globus. Legion inherits features of earlier work on parallel processing systems and heterogeneous distributed computing systems. In particular, Legion is building on Mentat, an object-oriented parallel processing system developed at the University of Virginia. See Section 5.3 for more information on Legion.

It is no surprise that Legion makes use of object-oriented approaches. On classical parallel machines, data-parallelism has been one of the only alternatives (not always successful) to explicit message-passing, where the user is requested to describe in extension all the communications that will take place during the execution. On the computational grid, many programming paradigms are yet to be found! In software engineering, object-oriented language and methodologies have received a lot of attention. Standards such as Java and Corba are very promising. See [53] for a metacomputing approach based on Corba. In a near future, efficient implementations of Java may well represent a viable alternative to Fortran for scientific computations (see `http://www.javagrande.org`). In the framework of metacomputing, with strongly heterogeneous environments, object-oriented technologies provide an elegant solution to the problems raised by the heterogeneity of the resources. The idea is to encapsulate technical "details" such as protocols, data representations, migration policies, into the implementation of the objects. Object-oriented specifications of resources and services are described in [18].

In Section 5.4, we describe in more details the Dutch Albatross project [3] (see `http://www.cs.vu.nl/albatross`). The Albatross method relies on a high-performance Java system, with a highly efficient implementation of Java Remote Method Invocation. A quite different approach is to extend data-parallelism so as to cope with heterogenity: this is the objective of the OPUS project [49] (see `http://www.par.univie.ac.at/~erwin/opus`. OPUS is an object-based extension of HPF that supports the integration of coarse-grain task parallelism with HPF-style data-parallelism.

Despite all their advanced characteristics and features, the previous programming environments are far from achieving the holy goal: using the computing resources remotely and transparently, just as we do with electricity, without knowing where it comes from. The closest approximation to electricity consumption may come from remote computing systems such as the NetSolve project [21,

22] (see `http://www.cs.utk.edu/netsolve`). We describe NetSolve in Section 6.

4 Case Study: Cluster ScaLAPACK

In this section we deal with our first case study. We give a particular emphasis on the algorithmic aspects relevant to the design of a Cluster ScaLAPACK library in this section. Our discussion is based on [14, 15, 12, 13, 11, 10].

4.1 Introduction

Consider a heterogeneous NOW: whereas programming a large application made up of several loosely-coupled tasks can be performed rather easily (because these tasks can be dispatched dynamically on the available processors), implementing a tightly-coupled algorithm (such as a linear system solver) requires carefully tuned scheduling and mapping strategies. 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 processors of different speed. In this section, we illustrate the design of static strategies that achieve a good load balance on a heterogeneous NOW. Our target applications are simple numerical kernels: matrix-matrix product and dense linear solvers from the ScaLAPACK library.

4.2 Load Balancing on Unidimensional (Heterogeneous) Grids

In this section, a purely static strategy to allocate data and computations to heterogeneous processors is presented. The basis for such strategies is to distribute computations to processors so that the workload is evenly balanced, and so that no processor is kept idle by data dependences. We start with the simple problem of distributing independent chunks of computations to processors. We use this result to tackle the implementation of linear solvers.

Distributing independent chunks To illustrate the static approach, consider the following simple problem: given M independent chunks of computations, each of equal size (i.e. each requiring the same amount of work), how can we assign these chunks to p physical processors P_1, P_2, ..., P_p of respective execution times t_1, t_2, ..., t_p, so that the workload is best balanced? Here the execution time is understood as the number of time units needed to perform one chunk of computation. A difficulty arises when stating the problem: how accurate are the estimated processor speeds? won't they change during program execution? We come back on estimating processor speeds later, and we assume for a while that each processor P_i will indeed execute each computation chunk within t_i time units. Then how to distribute chunks to processors? The intuition is that the load of P_i should be inversely proportional to t_i. Since the loads must be integers, we use the following algorithm to solve the problem:

Algorithm 4.2: Optimal distribution for M independent chunks, over p processors of speed $t_1, \ldots, t_p$

Initialization: Approximate the c_i so that $c_i \times t_i \approx Constant$,
and $c_1 + c_2 + \ldots + c_p \leq M$.

$forall\, i \in \{1, \ldots, p\}$, $c_i = \left\lfloor \frac{\frac{1}{t_i}}{\sum_{i=1}^{p} \frac{1}{t_i}} \times M \right\rfloor$.

Iteratively increment some c_i until $c_1 + c_2 + \ldots + c_p = M$

for $m = c_1 + c_2 + \ldots + c_p$ **to** M

find $k \in \{1, \ldots, p\}$ such that $t_k \times (c_k + 1) = \min\{t_i \times (c_i + 1))\}$

$c_k = c_k + 1$

Algorithm 4.2 gives the optimal allocation [14, 15]. This algorithm can only be applied to simple load balancing problems such as matrix-matrix product on a processor ring. Indeed, such an algorithm can be decomposed into successive communication-free steps. The communication between steps are reduced to a simple shift across the ring of processes. Each step consists of a bunch of independent chunks that can be distributed using Algorithm 4.2. Consider a toy example with 3 processors of respective cycle-times $t_1 = 3$, $t_2 = 5$ and $t_3 = 8$. We aim to compute the product $C = A \times B$, where A and B are of size 2496×2496. The matrices can be decomposed into 78×78 square blocks of size 32×32 (32 is a typical blocksize for cache-based workstations [9]). Hence, $M = 78$ blocks of columns have to be distributed among the processors, and $M = 78$ independent chunks will be computed at each step. Table 1 applies Algorithm 4.2 to this load balancing problem. A few different steps for matrix multiplication are represented in Figure 2. Our simple allocation is quite sufficient for matrix multiplication, because each step is optimally load-balanced.

Steps	c_1	c_2	c_3	$\max_i(c_i t_i)$
Init, m=76	39	23	14	117
m=77	40	23	14	120
m=M=78	40	24	14	120

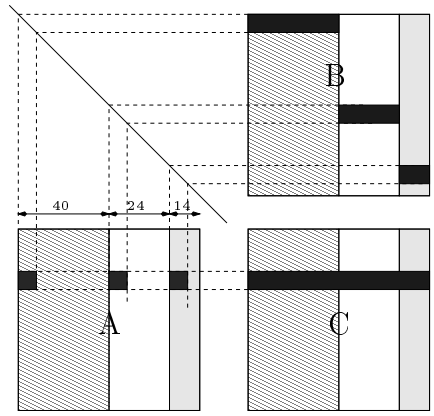
Table 1: Steps of algorithm 4.2 for 3 processors with $t_1 = 3$, $t_2 = 5$ and $t_3 = 8$, and $M = 78$

When processor speeds are accurately known and guaranteed not to change during program execution, the previous approach provides the best possible load balancing of the processors. Let us discuss the relevance of both hypotheses:

Estimating processor speed. There are too many parameters to accurately predict the actual speed of a machine for a given program, even assuming that the machine load will remain the same throughout the computation. Cycle-times must be understood as *normalized cycle-times* [27], i.e application-dependent elemental computation times, which are to be computed via small-scale experiments (repeated several times, with an averaging of the results).

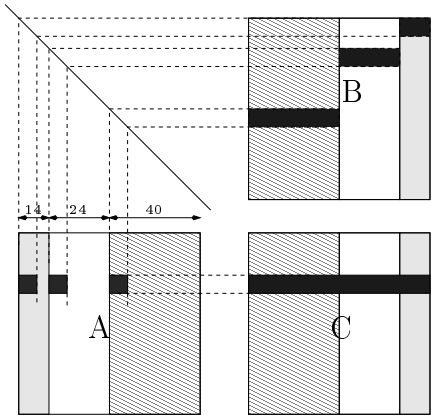
Changes in the machine load. Even during the night, the load of a machine may suddenly and dramatically change because a new job has just been started. The only possible strategy is to "use past to predict future": we can compute performance histograms during the current computation, these lead to new estimates of the t_i, which we use for the next allocation. See the survey paper of Berman [5] for further details.

In a word, a possible approach is to slice the total work into phases. We use small-scale experiments to compute a first estimation of the t_i, and we allocate chunks according to these



1^{st} **step**:

- p_1: $C[*, 1:40] += A[*, 1] \times B[1, 1:40]$;
- p_2: $C[*, 41:64] += A[*, 41] \times B[41, 41:64]$
- p_3: $C[*, 65:78] += A[*, 65] \times B[65, 65:78]$.



39^{th} **step**:

- p_1: $C[*, 1:40] += A[*, 39] \times B[39, 1:40]$

...

Figure 2: Different steps of matrix multiplication on a platform made of 3 heterogeneous processors of respective cycle-times $t_1 = 3$, $t_2 = 5$ and $t_3 = 8$. All indices in the figure are block numbers.

values for the first phase. During the first phase we measure the actual performance of each machine. At the end of the phase we collect the new values of the t_i, and we use these values to allocate chunks during the second phase, and so on. Of course a phase must be long enough, say a couple of seconds, so that the overhead due to the communication at the end of each phase is negligible. Each phase corresponds to B chunks, where B is chosen by the user as a trade-off: the larger B, the more even the predicted load, but the larger the inaccuracy of the speed estimation.

Linear solvers Whereas the previous solution is well-suited to matrix multiplication, it does not perform efficiently for LU decomposition. Roughly speaking, the LU decomposition algorithm works as follows for a heterogeneous NOW: blocks of r columns are distributed to processors in a cyclic fashion. This is a $CYCLIC(r)$ distribution of columns, where r is typically chosen as $r = 32$ or $r = 64$ [9]. At each step, the processor that owns the pivot block factors it and broadcasts it to all the processors, which update their remaining column blocks. At the next step, the next block of r columns become the pivot panel, and the computation progresses. The preferred distribution for a homogeneous NOW is a $CYCLIC(r)$ distribution of columns, where r is typically chosen as $r = 32$ or $r = 64$.

Because the largest fraction of the work takes place in the update, we would like to load-balance the work so that the update is best balanced. Consider the first step. After the factorization of the first block, all updates are independent chunks: here a chunk consists of the update of a single block of r columns. If the matrix size is $n = M \times r$, there are $M - 1$ chunks. We can use Algorithm 4.2 to distribute these independent chunks.

But the size of the matrix shrinks as the computation goes on. At the second step, the number of blocks to update is only $M - 2$. If we want to distribute these chunks independently of the first step, redistribution of data will have to take place between the two steps, and this will incur a lot of communications. Rather, we search for a static allocation of columns blocks to processors that will remain the same throughout the computations, as the elimination progresses. We aim at balancing the updates of all steps with the same allocation. As illustrated in Figure 3, we need a distribution that is kind of repetitive (because the matrix shrinks) but not fully cyclic (because processors have different speeds).

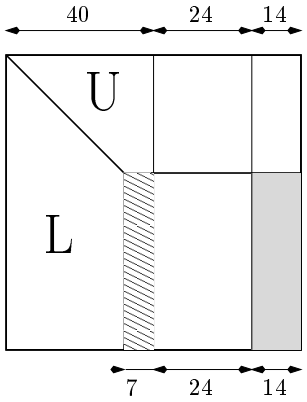


Figure 3: 33^{rd} step of LU decomposition (indices are block numbers): with the former distribution, the computation becomes less balanced. Here, after factoring block 33, processor 1 has 7 updates and works for $7 \times 3 = 21$ units of time, while processor 2 works $24 \times 5 = 120$ units of time.

Looking closer at the successive updates, we see that only column blocks of index $i+1$ to M are

updated at step i. Hence our objective is to find a distribution such that for each $i \in \{2, \ldots, M\}$, the amount of blocks in $\{i, \ldots, M\}$ owned by a given processor is approximately inversely proportional to its speed. To derive such a distribution, we use a dynamic programming algorithm which is best explained using the former toy example again:

Number of chunks	c_1	c_2	c_3	Cost	Selected processor
0	0	0	0		1
1	1	0	0	3	2
2	1	1	0	2.5	1
3	2	1	0	2	3
4	2	1	1	2	1
5	3	1	1	1.8	2
6	3	2	1	1.67	1
7	4	2	1	1.71	1
8	5	2	1	1.87	2
9	5	3	1	1.67	3
10	5	3	2	1.6	

Table 2: Running the dynamic programming algorithm with 3 processors: $t_1 = 3$, $t_2 = 5$, and $t_3 = 8$.

In Table 2, we report the allocations found by the algorithm up to $B = 10$. The entry "Selected processor" denotes the rank of the processor chosen to build the next allocation. At each step, "Selected processor" is computed so that the cost of the allocation is minimized. The cost of the allocation is computed as follows: the execution time, for an allocation $\mathcal{C} = (c_1, c_2, \ldots, c_p)$ is $\max_{1 \leq i \leq p} c_i t_i$ (the maximum is taken over all processor execution times), so that the average cost to execute one chunk is

$$cost(\mathcal{C}) = \frac{\max_{1 \leq i \leq p} c_i t_i}{\sum_{i=1}^{p} c_i}$$

For instance at step 4, i.e. to allocate a fourth chunk, we start from the solution for three chunks, i.e. $(c_1, c_2, c_3) = (2, 1, 0)$. Which processor P_i should receive the fourth chunk, i.e. which c_i should be incremented? There are three possibilities $(c_1 + 1, c_2, c_3) = (3, 1, 0)$, $(c_1, c_2 + 1, c_3) = (2, 2, 0)$ and $(c_1, c_2, c_3 + 1) = (2, 1, 1)$ of respective costs $\frac{9}{4}$ (P_1 is the slowest), $\frac{10}{4}$ (P_2 is the slowest), and $\frac{8}{4}$ (P_3 is the slowest). Hence we select $i = 3$ and we retain the solution $(c_1, c_2, c_3) = (2, 1, 1)$.

Of course, if we are to allocate 10 chunks, we can use Algorithm 4.2 and find that 5 chunks should be given to processor 3 to P_2 and 2 to P_3. But the dynamic programming algorithm returns the optimal solution for allocating any number of chunks, from 1 chunk up to B chunks [15].

How can the dynamic programming algorithm be applied to LU decomposition? We allocate slices of B blocks to processors, as illustrated in Figure 4. B is a parameter that will be discussed below. For a matrix of size $n = m \times r$, we can simply let $B = m$, i.e. define a single slice. Within each slice, we use the dynamic programming algorithm for $s = 0$ to $s = B$ in a "reverse" order. Consider the toy example in Table 1 with 3 processors of relative speed $t_1 = 3$, $t_2 = 5$ and $t_3 = 8$. The dynamic programming algorithm allocates chunks to processors as shown in Table 3. The allocation of chunks to processors is obtained by reading the second line of Table 3 from right to left: $(3, 2, 1, 1, 2, 1, 3, 1, 2, 1)$ (see Figure 5 for the detailed allocation within a slice). As illustrated in Figure 4, at a given step there are several slices of at most B chunks, and the number of chunks in the first slice decreases as the computation progresses (the leftmost chunk in a slice is computed

Chunk number	1	2	3	4	5	6	7	8	9	10
Processor number	1	2	1	3	1	2	1	1	2	3

Table 3: Static allocation for $B = 10$ chunks.

first and then there only remains $B-1$ chunks in the slice, and so on). In the example, the reversed allocation best balances the update in the first slice at each step: at the first step when there are the initial 10 chunks (1 factor and 9 updates), but also at the second step when only 8 updates remain, and so on. The updating of the other slices remains well-balanced by construction, since their size does not change, and we keep the best allocation for $B = 10$. See Figure 5 for the detailed allocation within a slice, together with the cost of the updates.

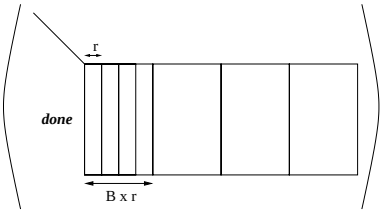


Figure 4: Allocating slices of B chunks.

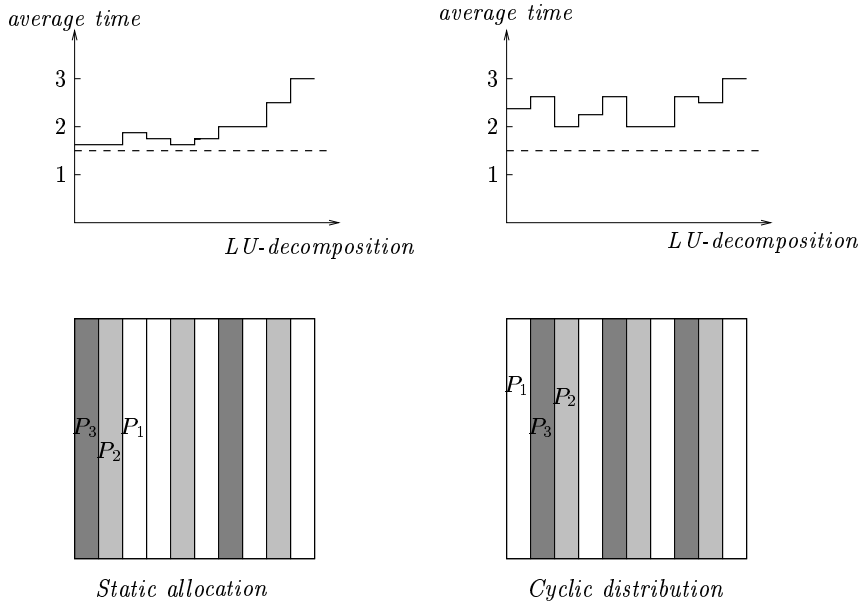


Figure 5: Comparison of two different distributions for the LU-decomposition algorithm on a heterogeneous platform made of 3 processors of relative speed 3, 5 and 8. The first distribution is the one given by our algorithm, the second one is the cyclic distribution. The total number of chunks is $B = 10$.

We are ready to propose a first solution for a heterogeneous cluster ScaLAPACK library devoted to dense linear solvers such as LU or QR factorizations. It turns out that all these solvers share the same computation unit, namely the processing of a block of r columns at a given step. They all

exhibit the same control graph: the computation processes by steps; at each step the pivot block is processed, and then it is broadcast to update the remaining blocks. The proposed solution is fully static: at the beginning of the computation, we distribute slices of the matrix to processors in a cyclic fashion. Each slice is composed of B chunks (blocks of r columns) and is allocated according to the previous discussion. The value of B is defined by the user and can be chosen as M if $n = m \times r$, i.e. we define a single slice for the whole matrix. But we can also choose a value independent of the matrix size: we may look for a fixed value, chosen from the relative processor speeds, to ensure a good load-balancing.

A major advantage of a fully static distribution with a fixed parameter B is that we can use the current ScaLAPACK release with little programming effort. In the homogeneous case with p processors, we use a $CYCLIC(r)$ distribution for the matrix data, and we define p PVM processes. In the heterogeneous case, we still use a $CYCLIC(r)$ distribution for the data, but we define B PVM processes which we allocate to the p physical processors according to our load-balancing strategy. The experiments reported in [13] demonstrate that this approach is quite satisfactory in practice.

4.3 Load Balancing on Two-Dimensional (Heterogeneous) Grids

In this Section we deal with matrix-matrix multiplication and dense linear solvers on 2D heterogeneous) grids. We configure the HNOW as a (virtual) 2D grid for scalability reasons [23]. First we briefly recall the algorithms implemented in the ScaLAPACK library [9] on 2D homogeneous grids. Then we discuss how to modify the two-dimensional block-cyclic distribution which is used in ScaLAPACK to cope with 2D heterogeneous grids.

Matrix-Matrix Product on Homogeneous Grids For the sake of simplicity we restrict to the multiplication $C = AB$ of two square $n \times n$ matrices A and B. In that case, ScaLAPACK uses the outer product algorithm described in [1, 39, 48]. Consider a 2D processor grid of size $p \times q$.

Assume first that $n = p = q$. In that case, the three matrices share the same layout over the 2D grid: processor $P_{i,j}$ stores $a_{i,j}$, $b_{i,j}$ and $c_{i,j}$. Then at each step k,

- each processor $P_{i,k}$ (for all $i \in \{1, .., p\}$) horizontally broadcasts $a_{i,k}$ to processors $P_{i,*}$.
- each processor $P_{k,j}$ (for all $j \in \{1, .., q\}$) vertically broadcasts $b_{k,j}$ to processors $P_{*,j}$.

so that each processor $P_{i,j}$ can independently compute $c_{i,j} += a_{i,k} \times b_{k,j}$.

This algorithm is used in the current version of the ScaLAPACK library because it is scalable, efficient and it does not need any initial permutation (unlike Cannon's algoritm [48]). Moreover, on a homogeneous grid, broadcasts are performed as independent ring broadcasts (along the rows and the columns), hence they can be pipelined.

Of course, ScaLAPACK uses a blocked version of this algorithm to squeeze the most out state-of-the-art processors with pipelined arithmetic units and multilevel memory hierarchy [31, 23]. Each matrix coefficient in the description above is replaced by a $r \times r$ square block, where optimal values of r depend on the communication-to-computation ratio of the target computer.

Finally, a level of virtualization is added: usually, the number of blocks $\lceil \frac{n}{r} \times \rceil \lceil \frac{n}{r} \rceil$ is much greater than the number of processors $p \times q$. Thus blocks are scattered in a cyclic fashion along both grid dimensions, so that each processor is responsible for updating several blocks at each step of the algorithm. An example is given in Figure 6 with $p = q = 4$ and $\lceil \frac{n}{r} \rceil = 10$.

1	2	3	4	1	2	3	4	1	2
5	6	7	8	5	6	7	8	5	6
9	10	11	12	9	10	11	12	9	10
13	14	15	16	13	14	15	16	13	14
1	2	3	4	1	2	3	4	1	2
5	6	7	8	5	6	7	8	5	6
9	10	11	12	9	10	11	12	9	10
13	14	15	16	13	14	15	16	13	14
1	2	3	4	1	2	3	4	1	2
5	6	7	8	5	6	7	8	5	6

Figure 6: Processors are numbered from 1 to 16. This figure represents the distribution of 10×10 matrix blocks onto 4×4 processors.

Matrix-Matrix Product on Heterogeneous Grids Suppose now we have a $p \times q$ grid of heterogeneous processors. Instead of distributing the $r \times r$ matrix blocks cyclically along each grid dimension, we distribute *block panels* cyclically along each grid dimension. A block panel is a rectangle of consecutive $B_p \times B_q$ $r \times r$ blocks. See Figure 7 for an example with $B_p = 4$ and $B_q = 3$: this panel of 12 $r \times r$ blocks will be distributed cyclically along both dimensions of the 2D grid. The previous cyclic dimension for homogeneous grids obviously corresponds to the case $B_p = p$ and $B_q = q$. Now, the distribution of individual blocks is no longer purely cyclic but remains periodic. We illustrate in Figure 8 how block panels are distributed on the 2D-grid.

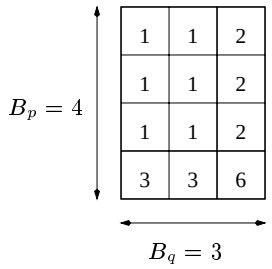


Figure 7: A block panel with $B_p = 4$ and $B_q = 3$. Each processor is labeled by its cycle-time, i.e the (normalized) time it needs to compute one $r \times r$ block: the processor labeled 1 is twice faster than the one labeled 2, hence it is assigned twice more blocks within each panel.

How many $r \times r$ blocks should be assigned to each processor within a panel ? Intuitively, as in the case of uni-dimensional grids, the workload of each processor (i.e. the number of block per panel it is assigned to) should be inversely proportional to its cycle-time. In the example of Figure 7, we have a 2×2 grid of processors of respective cycle-time $t_{1,1} = 1$, $t_{1,2} = 2$, $t_{2,1} = 3$ and $t_{2,2} = 6$. The allocation of the $B_p \times B_q = 4 \times 3 = 12$ blocks of the panel perfectly balances the load amongst the four processors.

There is an important condition to enforce when assigning blocks to processors within a block panel. We want each processor in the grid to communicate only with its four direct neighbors. This implies that each processor in a grid row is assigned the same number of matrix rows. Similarly, each processor in a grid column must be assigned the same number of matrix columns. If these conditions do not hold, additional communications will be needed, as illustrated in Figure 9.

Translated in terms of $r \times r$ matrix blocks, the above conditions mean that each processor P_{ij}, $1 \leq j \leq q$ in the i-th grid row must receive the same number r_i of blocks. Similarly, P_{ij}, $1 \leq i \leq p$ must receive c_j blocks. This condition does hold in the example of Figure 8, hence each processor

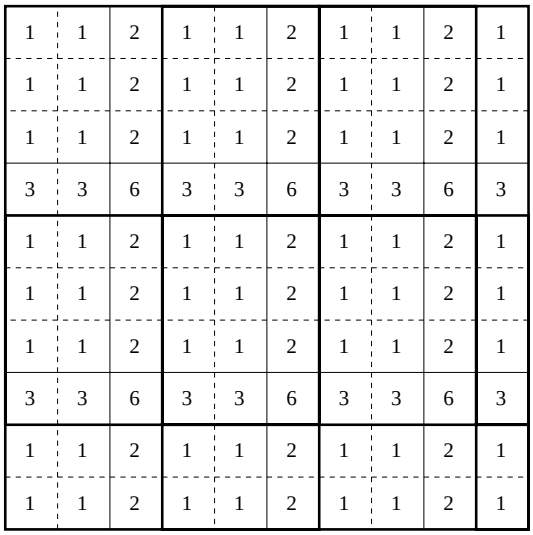


Figure 8: Allocating 4×3 panels on a 2×2 grid (processors are labeled by their cycle-time). There is a total of 10×10 matrix blocks.

only communicates with its direct neighbors.

Unfortunately, and in contrast with the uni-dimensional case, the additional constraints induced by the communication pattern may well prevent to achieve a perfect load balance amongst processors. Coming back to Figure 7, we did achieve a perfect load balance, owing to the fact that the processor cycle-times could be arranged in the rank-1 matrix

$$\begin{pmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 6 \end{pmatrix}.$$

For instance, change the cycle-time of $P_{2,2}$ into $t_{22} = 5$. If we keep the same allocation as in Figure 7, P_{22} remains idle every sixth time-step. Note that there is no solution to perfectly balance the work. Indeed, let r_1, r_2, c_1 and c_2 be the number of blocks assigned to each row and column grid. Processor P_{ij} computes $r_i \times c_j$ blocks in time $r_i \times c_j \times t_{ij}$. To have a perfect load balance, we have to fulfill the following equations:

$$r_1 \times t_{11} \times c_1 = r_1 \times t_{12} \times c_2 = r_2 \times t_{21} \times c_1 = r_2 \times t_{22} \times c_2$$

$$\text{that is} \quad r_1 c_1 = 2 r_1 c_2 = 3 r_2 c_1 = 6 r_2 c_2.$$

We derive $c_1 = 2c_2$, then $r_1 = 3r_2 = \frac{5}{2} r_2$, hence a contradiction. Note that we have not taken into account the additional condition $(r_1 + r_2) \times (c_1 + c_2) = 12$, stating that there are 12 blocks within a block panel: it is impossible to perfectly load-balance the work, whatever the size of the panel.

If we relax the constraints on the communication pattern, we can achieve a perfect load-balance as follows: first we balance the load in each processor column independently (using the uni-dimensional scheme); next we balance the load between columns (using the uni-dimensional scheme again, weighting each column by the inverse of the harmonic mean of the cycle-times of the processors within the column, see below). This is the "heterogeneous block cyclic distribution" of Kalinov and Lastovetky [45], which leads to the solution of Figure 9. Because processor $P_{2,2}$ has two west neighbors instead of one, at each step of the algorithm it is involved in two horizontal broadcasts instead of one.

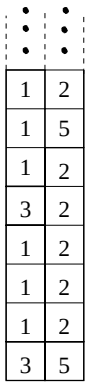


Figure 9: The distribution of Kalinov and Lastovetky. Two consecutive columns are represented here. Processors have two west neighbors instead of one.

We use the example to explain with further details how the heterogeneous block cyclic distribution of Kalinov and Lastovetky [45] works. First they balance the load in each processor column independently, using the uni-dimensional scheme. In the example there are two processors in the first grid column with cycle-times $t_{11} = 1$ and $t_{21} = 3$, so P_{11} should receive three times more matrix rows than P_{21}. Similarly for the second grid column, P_{12} (cycle-time $t_{12} = 2$) should receive 5 out of every 7 matrix rows, while P_{22} (cycle-time $t_{22} = 5$) should receive the remaining 2 rows. Next how to distribute matrix columns? The first grid column operates as a single processor of cycle-time $2\frac{1}{1+\frac{1}{3}} = \frac{3}{2}$. The second grid column operates as a single processor of cycle-time $2\frac{1}{\frac{1}{2}+\frac{1}{5}} = \frac{20}{7}$. So out of every 61 matrix columns we assign 40 to the first processor column and 21 to the second processor column.

Because we have a library designer's approach, we do not want the number of horizontal and vertical communications to depend upon the data distribution. For large grids, the number of horizontal neighbors of a given processor cannot be bounded a priori if we use Kalinov and Lastovetky's approach. We enforce the grid communication pattern (each processor only communicates with its four direct neighbors) to minimize communication overhead. The price to pay is that we have to solve a difficult optimization problem to load-balance the work as efficiently as possible. Solving this optimization problem is the objective of Section 4.4.

LU and QR Decomposition on Homogeneous Grids In this section we briefly review the direct parallelization of the right-looking variant of the LU decomposition. We assume that the matrix A is distributed onto a two-dimensional grid of (virtual) homogeneous processors. We use a `CYCLIC(b)` decomposition in both dimensions. The right-looking variant is naturally suited to parallelization and can be briefly described as follows: Consider a matrix A of order N and assume that the LU factorization of the $k \times b$ first columns has proceeded with $k \in \left\{0, 1, \ldots \frac{N-1}{b}\right\}$. During the next step, the algorithm factors the next panel of r columns, pivoting if necessary. Next the pivots are applied to the remainder of the matrix. The lower trapezoid factor just computed is broadcast to the other process columns of the grid using an increasing-ring topology, so that the the upper trapezoid factor can be updated via a triangular solve. This factor is then broadcast to the other process rows using a minimum spanning tree topology, so that the remainder of the matrix can be updated by a rank-r update. This process continues recursively with the updated matrix. In other words, at each step, the current panel of columns is factored into L and the trailing submatrix $\bar{A}$ is updated. The key computation is this latter rank-b update $\bar{A} \leftarrow \bar{A} - LU$ that can

be implemented as follows:

1. The column processor that owns L broadcasts it horizontally (so there is a broadcast in each processor row)

2. The row processor that owns U broadcasts it vertically (so there is a broadcast in each processor column)

3. Each processor locally computes its portion of the update

The communication volume is thus reduced to the broadcast of the two row and column panels, and matrix $\bar{A}$ is updated in place (this is known as an outer -product parallelization). Load balance is very good. The simplicity of this parallelization, as well as its expected good performance, explains why the right-looking variants have been chosen in ScaLAPACK [23]. See [30, 23, 8] for a detailed performance analysis of the right-looking variants, that demonstrates their good scalability property. The parallelization of the QR decomposition is analogous [25, 24]

LU and QR Decomposition on Heterogeneous Grids For the implementation of the LU and QR decomposition algorithms on a heterogeneous 2D grid, we modify the ScaLAPACK $CYCLIC(r)$ distribution very similarly as for the matrix-matrix multiplication problem. The intuitive reason is the following: as pointed out before, the core of the LU and QR decompositions is a rank-r update, hence the techniques for the outer-product matrix algorithm naturally apply.

We still use block panels made up with several $r \times r$ matrix blocks. The block panels are distributed cyclically along both dimensions of the grid. The only modification if that the order of the blocks within a block panel becomes important.

Consider the previous example with four processors laid along a 2×2 grid as follows:

$$\begin{pmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 5 \end{pmatrix}.$$

Say we use a panel with $B_p = 8$ ad $B_q = 6$, i.e. a panel composed of 48 blocks. Using the methods described below (see Section 4.4), we assign he blocks as follows:

- Within each panel column, the first processor row receives 6 blocks and the second processor rows receives 2 blocks

- Out of the 6 panel columns, the first grid column receives 4 and the second grid column receive 2

This allocation is represented in Figure 10. We need to explain how we have allocated the six panel columns. For the matrix multiplication problem, the ordering of the blocks within the panel was not important, because all processors execute the same amount of (independent) computations at each step of the algorithm. For the LU and QR decomposition algorithms, the ordering of the columns is quite important: the size of the matrix shrinks at each step of the computation. We use the uni-dimensional algorithm to compute the column ordering for the 2D panel. In the example, the first processor column operates like 6 processors of cycle-time 1 and 2 processors of cycle-time 3, which is equivalent to a single processor A of cycle-time $\frac{3}{20}$; the second processor column operates like 6 processors of cycle-time 2 and 2 processors of cycle-time 5, which is equivalent to a single processor B of cycle-time $\frac{5}{17}$. The uni-dimensional algorithm allocates the six panel columns as $ABAABA$, and we retrieve the allocation of Figure 10.

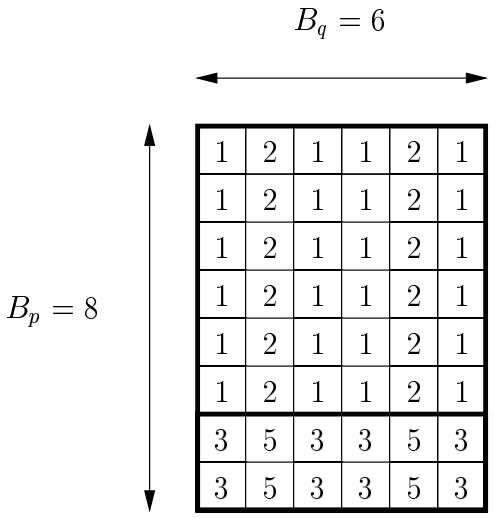


Figure 10: Allocation of the blocks within a block panel with $B_p = 8$ and $B_q = 6$. Each processor of the 2×2 grid is labeled by its cycle-time.

To conclude this section, we have a difficult load-balancing problem to solve. First we do not know which is the best layout of the processors, i.e. how to arrange them to build an efficient 2D grid. In some cases (rank-1 matrices) we are able to load-balance the work perfectly, but in most cases it is not the case. Next, once the grid is built, we have to determine the number of blocks that are assigned to each processor within a block panel. Again, this must be done so as to load-balance the work, because processors have different speeds. Finally, the panels are cyclically distributed along both grid dimensions.

4.4 Solving the 2D Heterogeneous Grid Allocation Problem

Problem Statement and Formulation Consider n processors P_1, P_2, ..., P_n of respective cycle-times $t_1, t_2, \ldots, t_n$. The problem is to arrange these processors along a two-dimensional grid of size $p \times q \leq n$, in order to compute the product $Z = XY$ of two $N \times N$ matrices as fast as possible. We need some notations to formally state this objective.

Consider a given arrangement of $p \times q \leq n$ processors along a two-dimensional grid of size $p \times q$. Let us re-number the processors as P_{ij}, with cycle-time t_{ij}, $1 \leq i \leq p, 1 \leq j \leq q$. Assume that processor P_{ij} is assigned a block of r_i rows and c_j columns of data elements, meaning that it is responsible for computing $r_i \times c_j$ elements of the Z matrix: see Figure 11 for an example.

There are two (equivalent) ways to compute the efficiency of the grid:

- Processor P_{ij} is assigned a rectangular data block of size $r_i \times c_j$, which it will process within $r_i \times c_j \times t_{ij}$ units of time. The total execution time T_{exe} is taken over all processors:

$$T_{exe} = \max_{i,j} \{r_i \times t_{ij} \times c_j\}.$$

  T_{exe} must be normalized to the average time T_{ave} needed to process a single data element: since there is a total of N^2 elements to compute, we enforce that $\sum_{i=1}^{p} r_i = N$ and that $\sum_{j=1}^{q} c_j = N$. We get

$$T_{ave} = \frac{\max_{i,j}\{r_i \times t_{ij} \times c_j\}}{(\sum_{i=1}^{p} r_i) \times \left(\sum_{j=1}^{q} c_j\right)}.$$

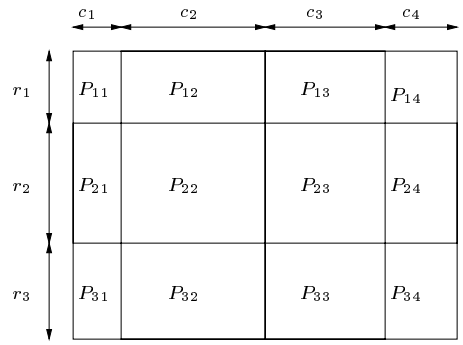


Figure 11: Allocating computations to processors on a 3×4 grid

We are looking for the minimum of this quantity over all possible integer values r_i and c_j. We can simplify the expression for T_{ave} by searching for (nonnegative) rational values r_i and c_j which sum up to 1 (instead of N):

$$\textbf{Objective } Obj_1\text{: } \min_{(\sum_{i=1}^{p} r_i=1;\sum_{j=1}^{q} c_j=1)}\{r_i \times t_{ij} \times c_j\}$$

Given the rational values r_i and c_j returned by the solution of the optimization problem Obj_1, we scale them by the factor N to get the final solution. We may have to round up some values, but we do so while preserving the relation $\sum_{i=1}^{p} r_i = \sum_{j=1}^{q} c_j = N$. Stating the problem as Opt_1 renders its solution generic, i.e. independent of the parameter N.

- Another way to tackle the problem is the following: what is the largest number of data elements that can be computed within one time unit? Assume again that each processor P_{ij} of the $p \times q$ grid is assigned a block of r_i rows and c_j columns of data elements. We need to have $r_i \times t_{ij} \times c_j \leq 1$ to ensure that P_{ij} can process its block within one cycle. Since the total number of data elements being processed is $(\sum_{i=1}^{p} r_i) \times \left(\sum_{j=1}^{q} c_j\right)$, we get the (equivalent) optimization problem:

  $$\textbf{Objective } Obj_2\text{: } \max_{r_i \times t_{ij} \times c_j \leq 1}\{(\sum_{i=1}^{p} r_i) \times \left(\sum_{j=1}^{q} c_j\right)\}$$

  Again, the rational values r_i and c_j returned by the solution of the optimization problem Obj_2 can be scaled and rounded to get the final solution.

  Although there are $p + q$ variables r_i and c_j, there are only $p + q - 1$ degrees of freedom: if we multiply all r_i's by the same factor λ and divide all c_j by λ, nothing changes in Obj_2. In other words, we can impose $r_1 = 1$, for instance, without loss of generality.

The 2D load-balancing problem In the next sections we give a solution to the 2D load-balancing problem which can be stated as follows: given $n = p \times q$ processors, how to arrange them along a 2D grid of size $p \times q$ so as to optimally load-balance the work of the processors for the matrix-matrix multiplication problem. Note that solving this problem will in fact lead to the solution of many linear algebra problems, including dense linear system solvers.

The problem is even more difficult to tackle than the optimization problem stated above, because we do not assume the processors arrangement as given. We have to search among all possible arrangements (layouts) of the $p \times q$ processors as a $p \times q$ grid, and for each arrangement we must solve the optimization problem Obj_1 or Obj_2.

The approcach of [10] is the following: first the number of arrangements to be searched is reduced. . Next an algorithm is derived to solve the optimization problem Obj_1 or Obj_2 for a fixed

(given) arrangement. Despite the reduction, there remains an exponential number of arrangements to search for. Even worse, for a fixed arrangement, the above algorithm exhibits an exponential cost. Therefore a heuristic is introduced to give a fast but sub-optimal solution to the 2D load-balancing problem.

Conclusion Solving the 2D load balancing problem turns out to be surprisingly difficult. However, we have dealt with simple numerical kernels (matrix-matrix product and dense linear solvers) on simple target platforms (heterogeneous networks of workstations). We believe that many efforts are to be spent if tightly-coupled applications are to be implemented on the grid.

4.5 Load Balancing on Collections of Clusters

In this section, 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. We use the model presented in Section 3.2. We restrict ourselves to the problem of finding an optimal allocation for the LU and QR factorizations on a 2-deep grid. To this purpose, we assume that in each NOW, a processor is dedicated to handle the communications between NOWs, as shown in Figure 1.

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 blocks of r columns, where r is chosen to ensure Level 3 BLAS performance [9] 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 τ_a is the elemental computation time. Communicating a panel between NOWs requires $nBr\tau_c$ units of time, where τ_c is the inter-NOW communication rate. Of course τ_c is several orders of magnitude greater than τ_a, but letting 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.

Static strategy We decompose our matrix into panels of size B: a panel is a slice of B 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 [11]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.

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 [54]. It is illustrated in Figure 12. Each task in Figure 12 represents a panel (and not a single column block). 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.

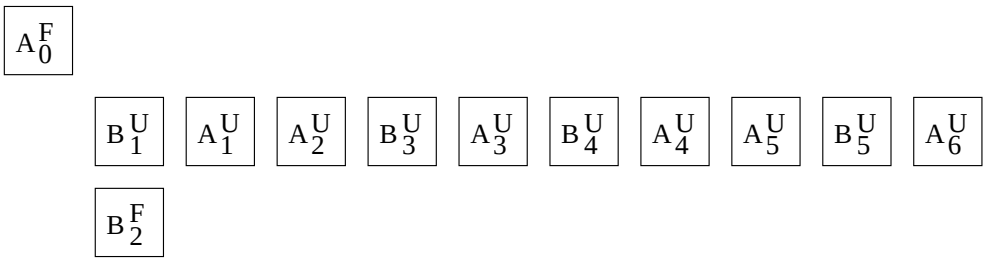


Figure 12: Scheduling with 2 clusters A and B. Exponents represent the nature of the tasks (factoring or updating a panel). Indices represent the steps at which the tasks are processed.

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 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 13 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.

Conclusion Implementing linear algebra kernels on several collections of workstations or parallel servers, scattered all around the world and connected through fast but non-dedicated links, would give rise to a "Computational Grid ScaLAPACK". The above results constitute a very preliminary step towards achieving this ambitious goal (see [12] for further details). Again, a major algorithmic effort is needed to efficiently implement tightly-coupled applications on meta-systems (collections

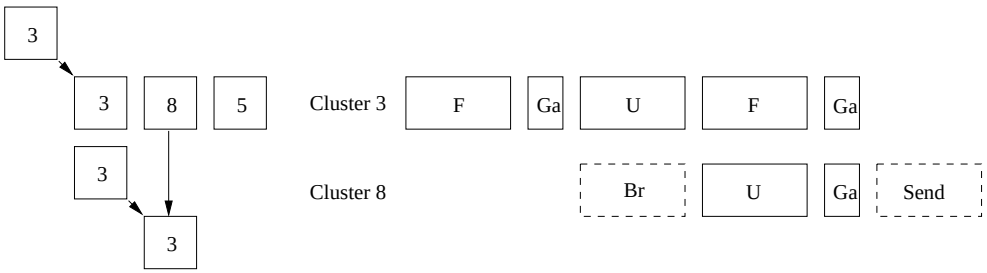


Figure 13: Scheduling the tasks and the communications with the dynamic strategy.

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.

5 Case Study: Metacomputing Environments

In this section we summarize four major programming environments: AppLeS (Section 5.1), Globus (Section 5.2), Legion (Section 5.3) and Albatross (Section 5.4).

5.1 AppLeS

The AppLeS project is dedicated to the achievement of performance for metacomputing applications, a difficult goal due to the heterogeneity of metacomputing platforms. This is especially true for parallel applications whose performance is highly dependent upon the efficient coordination of their constituent components.

In metacomputing systems, applications cannot be efficiently scheduled by a global scheduling mechanism. Currently, to achieve a performance-efficient implementation on a distributed heterogeneous system, the application developer must select a potentially efficient configuration of resources based on load and availability, evaluate the potential performance on such configurations based on their own performance criteria, and interact with the relevant resource management systems in order to implement the application. This approach is termed *application-centric* by Berman and Wolski [6, 7].

The AppLeS (Application Level Scheduler) project is developing application-level scheduling agents to provide a mechanism for scheduling individual applications at machine speeds on production heterogeneous systems. The following statements are used as guidelines to implement the agents [6]:

- Both application-specific and system-specific information are required for good schedules
- Performance depends upon the application's own performance criteria
- The "distance" between resources is dependent upon how the application uses them
- Dynamic information is necessary to accurately assess system state
- Predictions are accurate only within a particular time frame
- A schedule is only as good as its underlying prediction

The AppLeS application-level scheduling paradigm addresses heterogeneity by explicitly assuming that all resources (even homogeneous resources) exhibit individual performance characteristics, and that these performance characteristics may vary over time. Contention is addressed by assessing the fraction of available resources dynamically, and using this information to predict the fraction available at the time the application will be scheduled.

AppLeS agents utilize a Network Weather Service (NWS) to monitor the varying performance of resources potentially usable by their applications. Recently, adaptive regression modeling has been introduced as a technique to determine data transfer times for network-bound data-intensive applications [34]. Each AppLeS uses static and dynamic application and system information to select viable resource configurations and evaluate their potential performance. AppLeS then interacts with the relevant resource management system to implement application tasks.

The end-user or application developer provides its AppLeS agent with application-specific information about current implementation(s) (via the Heterogeneous Application Template or HAT) as well as user preferences. This information is combined with dynamic system information (provided by the Network Weather Service) by the AppLeS Coordinator to determine a potentially

performance-efficient application schedule. The Coordinator then works with the appropriate resource management systems to implement the schedule on the relevant resources. The AppLeS architecture is shown in Figure 14, borrowed from [6]. It is important to note that AppLeS agents are not resource management systems. The AppLeS system is built on top on tools such as Globus and Legion.

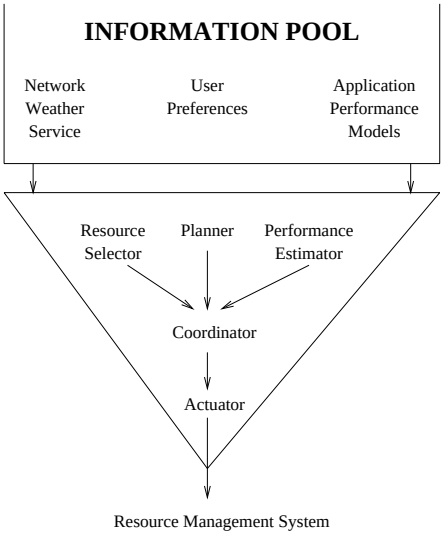


Figure 14: Organization of an AppLeS agent.

The AppLeS team has reported the design of AppLeS agents for a number of applications including:

- Two-dimensional and three-dimensional Jacobi iterative applications,
- Genetic Algorithm application,
- Mandelbrot application,
- Protein docking application based on DOT,
- Synthetic Apperture Radar Atlas (SARA) application.

We refer the reader to the AppLeS publications (listed at `http://www.cs.ucsd.edu/groups/hpcl/apples/hetpubs.html` for more information on these applications. See also the survey paper by Berman [5] for a comparison with several other scheduling systems.

5.2 Globus

The Globus toolkit [37] (see `http://www.globus.org`) is a collection of sotware components designed to support the development of applications for computational grids. The Globus architecture may be defined as a "bag of services" provided to application and tool developers (as opposed to a monolithic system). Each Globus component provides a basic service, such as authentication, resource allocation, information and remote data-access. Different applications and tools can combine these services in different ways to construct "grid-enabled" systems.

Service	Name	Description
Communication	Nexus	Unicast, multicast multiprotocol communications
Resource management	GRAM	Resource allocationt and process management
Executable management	GEM	Construction, cachingt and location of executables
Health and status	HBM	Monitoring of healtht and status of components
Information	MDS	Distributed access to structuret and state information
Remote access	GASS	Remote access to datat via sequential and parallel interfaces
Security	GSI	Authenticationt and related security services

Table 4: The core services of the Globus toolkit.

Briefly, the Globus toolkit comprises the core services listed in Table 4, plus a selection of higher-level services defiend in terms of these core services. Each core service defines an application program interface (API) that provides a uniform interface to a local service. As a first example, the Globus Resource Allocation Manager (GRAM) provides an API for requesting that some computation be started on a computational resource, and for managing these computations once they are started [29]. A second example is the following: resource brokers and co-allocators use services provided by the GRAM and by the Globus information service (named MDS for Metacomputing Directory Service) to locate available resources and to start computations across distributed computing resources. Both static information (amount of memory, CPU speed) and dynamic information (network latency, CPU load) are handled by MDS. MDS has been built on the data representation and API defined by the Lightweight Directory Access Protocol (LDAP), which in turn is based on the X-500 standard.

The Globus testbed has been used by several US research groups to conduct a variety of application experiments, in such areas as multi-user collaborative environments (tele-immersion), distributed supercomputing and high-throughput computing.

5.3 Legion

The Legion project [42] (see `http://www.cs.virginia.edu/~legion`) investigates issues relating to software architecture and base technologies for grids environments. Legion is mainly organized around an object-oriented model in which all components of the system are represented by independent, active objects that communicate using a uniform method invocation service. In many ways, Legion's object model is similar to Corba's: object interfaces are described using an interface description language (IDL), and are compiled and linked to implementations in a given language (C++, Java, Fortran). This object-based approach enables component interoperability between multiple programming languages and heterogeneous platforms. Objects provides a clean, natural approach to the well-known problems of encapsulation and interoperability. Because all the elements in the system are objects, they can communicate with one another regardless of location,

heterogeneity or implementation details.

Both processors and storage resources are represented as objects, called host objects and vault objects. There are two primary benefits resulting from this object-based approach. First, each object defines a uniform interface to host and vault resources. Host objects provides a uniform interface to task creation, and vault objects provides a uniform storage allocation interface. Second, these objects naturally act as resource guardians and policy makers. For example, the host objects used to manage the processor resources at a given site are the points of access control for task creation at that site. If an organization participating in Legion wishes to restrict job creation on local resources exclusively to local users, the host objects can enforce this policy.

In Legion, access control and resources protection are specified entirely at the object level. Legion allows messages to be fully encrypted for privacy, and signed for integrity checking, or sent in the clear if low performance overhead is an application priority. Cryptographic services in Legion are based on the RSA public key system. To protect the system components, object encode their RSA public key directly in their object identifier (LOID: Legion Object IDentifier), Simply by knowing the LOID for an object, a client is assured of being able to communicate securely with that object.

Globus and Legion share a common base of target environments, technical objectives, and target end users. However, the design philosophies are fundamentally different and driven by different high level objectives. Globus strives to provide a basic set of services that makes it possible to write applications that operate in a wide-area environment. The Globus implementation is based on the composition of working components into a composite metacomputing toolkit. Legion strives to reduce complexity and provide the programmer with a single view of the underlying resources. Legion builds higher-level system functionality on top of a single unified object model.

5.4 Albatross

The aim of the Albatross project (see `http://www.cs.vu.nl/albatross`) is to study applications and programming environments for wide-area cluster computers. As mentioned in Section 2.2, the main experimentation platform for the Albatross project is the DAS platform (Distributed ASCI Supercomputer, `http://www.cs.vu.nl/~bal/das.html`), which consists of four Myrinet-based cluster computers located at four Dutch universities and linked through a 6Mbit/s ATM network.

The focus of Albatross is on programmability and performance, and concentrates on the following two issues [3]:

- Efficient communication-protocols and runtime systems for local cluster computers
- Efficient algorithms and programming environments for wide-area cluster computers

The communication software used in Albatross is based on the Panda library [2], which provides multithreading and communication primitives (point-to-point message passing, RPC and broadcast). Several wide-area programming environments have been implemented on the top of Panda:

- Orca, a parallel language that provides an object-based distributed shared memory model [2]
- MagPIe, an MPI library based on MPICH whose collective communications primitives have been optimized for wide-area hierarchical systems [46]
- Manta, a high-performance wide-area Java system [64]

We say a few words about Manta, which looks a very promising environment for a wide range of metacomputing applications. Manta uses a native compiler that generates executable code rather than byte code. A key feature of Manta is its highly efficient implementation of Remote method Invocation (RMI). The Albatross team reports that in doing experiments with Java RMI and JavaParty, they found the programming model of RMI and JavaParty to be convenient, but they also found that performance of RMI for programming parallel clusters of workstations was too slow by far. Manta does a null-RMI in 35 microsecond, as opposed to 1200 microsecond or a two-way latency of Sun's RMI on Myrinet.

The Albatross project has optimized four applications for wide-area systems: successive over-relaxation (SOR), all-pairs shortest paths, traveling salesperson problem (TSP) and iterative deepening A^*. These four applications have been implemented on the DAS platform. Good speedups were obtained when using the four distributed Myrinet clusters, at the price of a significant programming effort to optimize the applications. The Albatross results, although preliminary, demonstrate that it is feasible to efficiently run parallel applications on multiple clusters connected by wide-area networks. The next step in the project is to develop programming support that eases wide-area parallel programming.

6 Case Study: NetSolve

In we section we describe NetSolve, a high-level environment for metacomputing. Our description is based on [55].

6.1 Introduction

The main idea of the NetSolve project is to provide non-specialists users with an easy-to-use programming environment that enables them to benefit from a variety of distributed computing resources such as massively parallel processors, networks and clusters of workstations and "piles" of PCs. In order to use efficiently such a diverse and lively computational environment, many challenging research aspects of network-based computing such as fault-tolerance, load balancing, user-interface design, computational servers or virtual libraries, must be addressed. User-friendly, network-enabled, application-specific toolkits have been specifically designed and conceived to tackle the problems posed by such a complex and innovative approach to scientific problem solving [38].

In the networked computing paradigm, vital pieces of software and information used by a computing process are spread across the network, and are identified and linked together only at run time. This is in contrast to the current software usage model where one acquires a copy (or copies) of task-specific software package for use on local hosts. In this section, as a case study, we focus on the ongoing NetSolve project developed at the University of Tennessee and at the Oak Ridge National Laboratory (see `http://www.cs.utk.edu/netsolve`).

The NetSolve software system [21, 22] provides users with a pool of computational resources. These resources are computational servers that provide run-time access to arbitrary optimized numerical software libraries. The NetSolve software system transforms disparate, loosely-connected computers and software libraries into a unified, easy-to-access computational service. This service can make enormous amounts of computing power transparently available to users on ordinary platforms.

The NetSolve system allows users to access computational resources, such as hardware and software, distributed across the network. These resources are embodied in computational servers and allow users to easily perform scientific computing tasks without having any computing facility installed on their computer. Users' access to the servers is facilitated by a variety of interfaces: Application Programming Interfaces (APIs), Textual Interactive Interfaces and Graphical User Interfaces (GUIs). As the NetSolve project matures, several promising extensions and applications will emerge. In this section, we provide an overview of the project and examine two extensions developed for NetSolve: an interface to the Condor system [50], and an interface to the ScaLAPACK parallel library [9].

6.2 Overview of the NetSolve System

The NetSolve system uses the *remote computing* paradigm: the program resides on the server; the user's data is sent to the server, where the appropriate programs or numerical libraries operate on it; the result is then sent back to the user's machine.

Figure 15 depicts the typical layout of the system. NetSolve provides users with a pool of computational resources. These resources are computational servers that have access to ready-to-use numerical software. As shown in the figure, the computational servers can be running on single workstations, networks of workstations that can collaborate for solving a problem, or Massively Parallel Processor (MPP) systems. The user is using one of the NetSolve client interfaces.

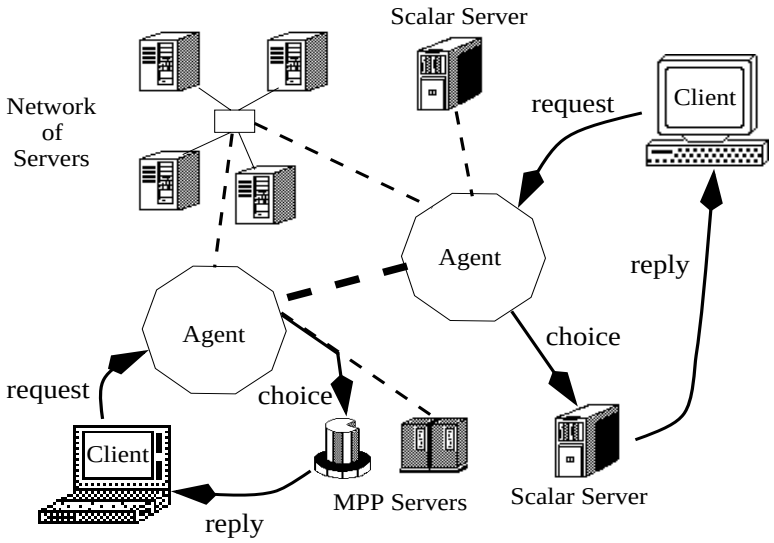


Figure 15: NetSolve's organization

Through these interfaces, the user can send requests to the NetSolve system asking for a numerical computation to be carried out by one of the servers. The main role of the NetSolve agent is to process this request and to choose the most suitable server for this particular computation. Once a server has been chosen, it is assigned the computation, uses its available numerical software, and eventually returns the results to the user. One of the major advantages of this approach is that the agent performs load-balancing among the different resources.

As shown in Figure 15, there can be multiple instances of the NetSolve agent on the network, and different clients can contact different agents depending on their locations. The agents can exchange information about their different servers and allow access from any client to any server if desirable. NetSolve can be used either via the Internet or on an intranet, such as inside a research department or a university, without participating in any Internet based computation. Another important aspect of NetSolve is that the configuration of the system is entirely flexible: any server/agent can be stopped and (re-)started at any time without jeopardizing the integrity of the system.

The Computational Resources When building the NetSolve system, one of the challenges was to design a suitable model for the computational servers. The NetSolve servers are configurable so that they can be easily upgraded to encompass ever-increasing sets of numerical functionalities. The NetSolve servers are also pre-installed, meaning that the end-user does not have to install any numerical software. Finally, the NetSolve servers provide uniform access to the numerical software, in the sense that the end-user has the illusion that he or she is accessing numerical subroutines from a single, coherent numerical library.

To make the implementation of such a computational server model possible, a general, machine-independent way of describing a numerical computation as well as a set of tools to generate new computational modules as easily as possible have been designed. The main component of this framework is a *descriptive language* which is used to describe each separate numerical functionality of a computational server. The description files written in this language can be compiled by NetSolve into actual computational modules executable on any UNIX or NT platform. These files

can then be exchanged by any institution wanting to set up servers: each time a new description file is created, the capabilities of the entire NetSolve system are increased.

A number of description files have been generated for a variety of numerical libraries: ARPACK, FitPack, ItPack, MinPack, FFTPACK, LAPACK, BLAS, QMR, Minpack and ScaLAPACK. These numerical libraries cover several fields of computational science; Linear Algebra, Optimization, Fast Fourier Transforms, etc.

The Client Interfaces A major concern in designing NetSolve was to provide several interfaces for a wide range of users. NetSolve can be invoked through C, Fortran, Java, Matlab and Mathematica. In addition, there is a Web-enabled Java GUI. Another concern was keeping the interfaces as simple as possible. For example, there are only two calls in the MATLAB interface, and they are sufficient to allow users to submit problems to the NetSolve system. Each interface provides asynchronous calls to NetSolve in addition to traditional synchronous or blocking calls. When several asynchronous requests are sent to a NetSolve agent, they are dispatched among the available computational resources according to the load-balancing schemes implemented by the agent. Hence, the user—with virtually no effort—can achieve coarse-grained parallelism from either a C or Fortran program, or from interaction with a high-level interface. All the interfaces are described in detail in the "NetSolve's Client User's Guide" [21].

The NetSolve Agent Keeping track of what software resources are available and on which servers they are located is perhaps the most fundamental task of the NetSolve agent. Since the computational servers use the same framework to contribute software to the system, it is possible for the agent to maintain a database of different numerical functionalities available to the users.

Each time a new server is started, it sends an application request to an instance of the NetSolve agent. This request contains general information about the server and the list of numerical functions it intends to contribute to the system. The agent examines this list and detects possible discrepancies with the other existing servers in the system. Based on the agent's verdict, the server can be integrated into the system and available for clients.

The goal of the NetSolve agent is to choose the best-suited computational server for each incoming request to the system. For each user request, the agent determines the set of servers that can handle the computation and makes a choice between all the possible resources. To do so, the agent uses computation-specific and resource-specific information. Computation-specific information is mostly included in the user request whereas resource-specific information is partly static (server's host processor speed, memory available, etc.) and partly dynamic (processor workload). Rationale and further detail on these issues can be found in [17], as well as a description of how NetSolve ensures fault-tolerance among the servers.

Agent-based computing seems to be a promising strategy. NetSolve is evolving into a more elaborate system and a major part of this evolution is to take place within the agent. Such issues as user accounting, security, data encryption for instance are only partially addressed in the current implementation of NetSolve and already is the object of much work. As the types of hardware resources and the types of numerical software available on the computational servers become more and more diverse, the resource broker embedded in the agent need to become increasingly sophisticated. There are many difficulties in providing a uniform performance metric that encompasses any type of algorithmic and hardware considerations in a metacomputing setting, especially when different numerical resources, or even entire frameworks are integrated into NetSolve. Such integrations are described in the following sections.

6.3 Interface to the Condor System

Condor [50], developed at the University of Wisconsin, Madison, is an environment that can manage very large collections of distributively owned workstations. Its development has been motivated by the ever increasing need for scientists and engineers to exploit the capacity of such collections, mainly by taking advantage of otherwise unused CPU cycles. Interfacing NetSolve and Condor is a very natural idea. NetSolve provides remote easy access to computational resources through multiple, attractive user interfaces. Condor allows users to harness the power of a pool of machines while using otherwise wasted CPU cycles. The users at the consoles of those machines are not penalized by the scheduling of Condor jobs. If the pool of machines is reasonably large, it is usually the case that Condor jobs can be scheduled almost immediately. This could prove to be very interesting for a project like NetSolve. Indeed, NetSolve servers may be started so that they grant local resource access to outside users. Interfacing NetSolve and Condor could then give priority to local users and provide underutilized only CPU cycles to outside users.

A Condor pool consists of any number of machines, that are connected by a network. Condor daemons constantly monitor the status of the individual computers in the cluster. Two daemons run on each machine, the *startd* and the *schedd*. The *startd* monitors information about the machine itself (load, mouse/keyboard activity, etc.) and decides if it is available to run a Condor job. The *schedd* keeps track of all the Condor jobs that have been submitted to the machine. One of the machine, the *Central Manager*, keeps track of all the resources and jobs in the pool. When a job is submitted to Condor, the scheduler on the central manager matches a machine in the Condor pool to that job. Once the job has been started, it is periodically checkpointed, can be interrupted and migrated to a machine whose architecture is the same as the one of the machine on which the execution was initiated. This organization is partly depicted in Figure 16. More details on the Condor system and the software layers can be found in [50].

Figure 16 shows how an entire Condor pool can be seen as a single NetSolve computational resource. The Central Manager runs two daemons in addition to the usual *startd* and *schedd*: the *negotiator* and the *collector*. A machine also runs a customized version of the NetSolve server. When this server receives a request from a client, instead of creating a local child process running a computational module, it uses the Condor tools to submit that module to the Condor pool. The negotiator on the Central Manager then chooses a target machine for the computational module. Due to fluctuations in the state of the pool, the computational module can then be migrated among the machines in the pool. When the results of the numerical computation are obtained, the NetSolve server transmits that result back to the client.

The actual implementation of the NetSolve/Condor interface was made easy by the Condor tools provided to the Condor user. However, the restrictions that apply to a Condor job concerning system calls were difficult to satisfy and required quite a few changes to obtain a Condor-enabled NetSolve server. A major issue however still needs to be addressed; how does the NetSolve agent perceive a Condor pool as a resource? Finding the appropriate performance prediction technique is at the focus of the current NetSolve/Condor collaboration.

6.4 Integrating Parallel Numerical Libraries

Integrating software libraries designed for distributed-memory concurrent computers into NetSolve allows a workstation's user to access massively parallel processors to perform large computations. This access can be made extremely simple via NetSolve and the user may not even be aware that he or she is using a parallel library on such a computer. As an example, we describe in this section, how the ScaLAPACK library [9] has been integrated into the NetSolve system.

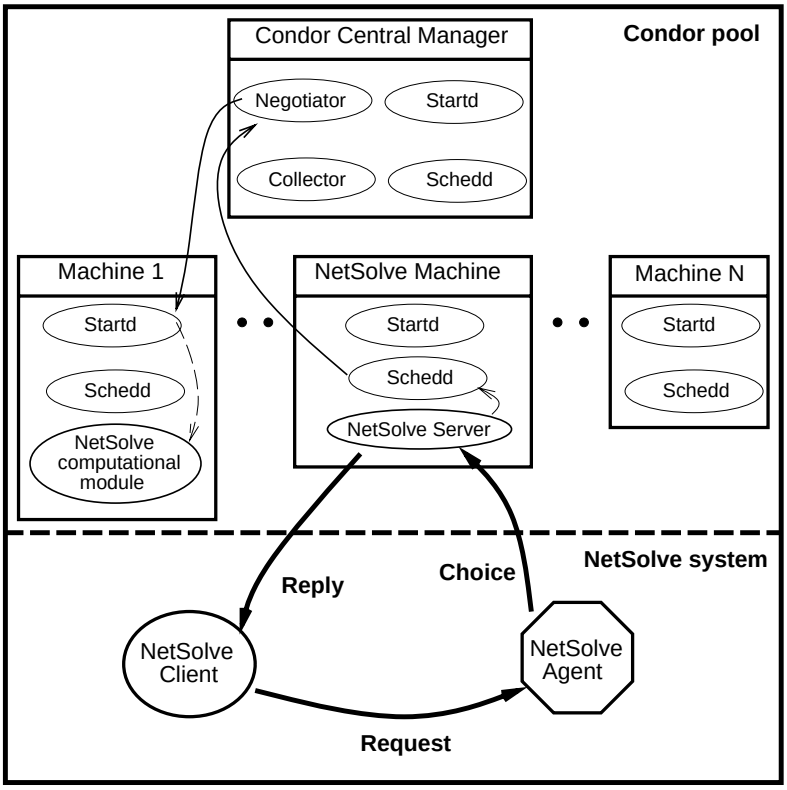


Figure 16: NetSolve and Condor

Figure 17 is a very simple description of how the NetSolve server has been customized to use the ScaLAPACK library. The customized server receives data input from the client in the traditional way. The NetSolve server uses BLACS calls to set up the ScaLAPACK process grid. ScaLAPACK requires that the data already be distributed among the processors prior to any library call. This is the reason why each user input is first distributed on the process grid according to the block cyclic decomposition when necessary. The server can then initiate the call to ScaLAPACK and wait until completion of the computation. When the ScaLAPACK call returns, the result of the computation is distributed on the two-dimensional process grid. The server then gathers that result and sends it back to the client in the expected format. This process is completely transparent to the user who does not even realize that a parallel execution has been taking place.

This approach is very promising. A client can use MATLAB on a PC and issue a simple call like `[x] = netsolve('eig',a)` and have an MPP system use a high-performance library to perform a large eigenvalue computation. A prototype of the customized server running on top of PVM [40] or MPI [61] has been designed. There are many research issues arising with integrating parallel libraries in NetSolve, including performance prediction, choice of processor-grid size, choice of numerical algorithm, processor availability, accounting, etc.

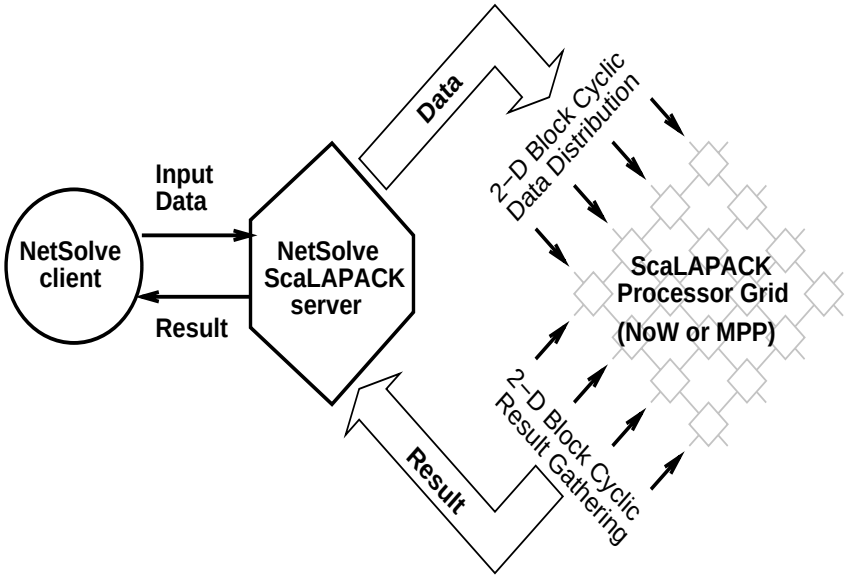


Figure 17: The ScaLAPACK NetSolve Server Paradigm

7 Conclusion

In this report, we have discussed some issued related to heterogeneous cluster computing and metacomputing. Here are a few recommendations:

- While there are several projects related to metacomputing in Europe, there is little coordination and exchange between these projects. Similarly, only few European institutions have joined the NPACI initiative: the NPACI Partnership Report of 1999 mentions only three international affiliates in Europe[4].
- The difficulty of algorithmic issues seem to be largely underestimated. Data decomposition, scheduling heuristics, load balancing, were known to be hard problems in the context of classical parallel architectures. They become extremely difficult in the context of metacomputing platforms. We believe that they should receive more attention from the research community, which seem to focus almost exclusively on low-level communication protocols and distributed system issues (light-weight process invocation, migration, etc).
- It is not clear which is the good level to program metacomputing platforms. Data-parallelism seems unrealistic, due to the strong heterogeneity. Explicit message passing is too low-level. Despite their many advantages, object-oriented approaches still request the user to have a deep knowledge and understanding of both its application behavior and the underlying hardware and network. Remote computing systems such as NetSolve face severe limitations to efficiently load-balance the work to processors. For the inexperienced user, relying on specialized but highly-tuned libraries of all kinds (communication, scheduling, application-dependent data decompositions) may prove a good trade-off until the programming environments evolve into "high-level-yet-general-purpose-and-efficient" solutions!

[4]The center for research on parallel computation and supercomputers in Naples, Italy; the computer engineering department of the university of Lecce, Italy; and the parallel computing center of the royal institute of technology in Stockholm, Sweden.

- Looking backward, we see that key applications (from scientific computing to data-bases) have dictated the way classical parallel machines are used, programmed, and even updated into more efficient platforms. Looking forward, we guess that key applications will strongly influence, or even guide, the development of metacomputing environments. Which applications will be worth the abundant but hard-to-access resources of the metacomputing grid: tightly-coupled grand challenges, mobile computing applications or micro-transactions on the Web ? All these applications require new programming paradigms to enable inexperienced users to access the magic grid!

References

[1] R. Agarwal, F. Gustavson, and M. Zubair. A high performance matrix multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM J. Research and Development*, 38(6):673–681, 1994.

[2] H.E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and F. Kaashoek. Performance evaluation of the orca shared object system. *ACM Trans. Computer Systems*, 16(1):1–40, 1998.

[3] H.E. Bal, A. Plaat, T. Kielmann, J. Maassen, R. van Nieuwpoort, and R. Veldema. Parallel computing on wide-area clusters: the albatross project. In *Extreme Linux Workshop*, pages 20–24, 1999.

[4] P.H Beckman, P.K. Fasel, and W.F. Humphrey. Efficient coupling of parallel applications using PAWS. In *High Performance Distributed Computing HPDC'98*, Chicago, IL, 1998. IEEE Computer Science Press.

[5] 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, 1999.

[6] F. Berman and R. Wolski. The apples project: A status report. In *Proceedings of the 8th NEC Research Symposium*, 1997. Available at .

[7] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing 1996*. IEEE Computer Society Press, 1996. Available as UCSD CS Tech Report CS96-482.

[8] L. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. Scalapack: A portable linear algebra library for distributed-memory computers - design issues and performance. In *Supercomputing '96*. IEEE Computer Society, 1996.

[9] 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.

[10] V. Boudet, F. Rastello, A. Petitet, and Y. Robert. Data allocation strategies for dense linear algebra kernels on heterogeneous two-dimensional grids. Technical Report RR-99-31, LIP, ENS Lyon, 1999. Available at `www.ens-lyon.fr/LIP/`. To appear in PDCS'99, Boston.

[11] 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`.

[12] Vincent Boudet, Fabrice Rastello, and Yves Robert. Algorithmic issues for (distributed) heterogeneous computing platforms. In Rajkumar Buyya and Toni Cortes, editors, *Cluster Computing Technologies, Environments, and Applications (CC-TEA'99)*. CSREA Press, 1999. Extended version available as LIP Technical Report RR-99-19.

[13] Vincent Boudet, Fabrice Rastello, and Yves Robert. A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers). In Hamid R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*. CSREA Press, 1999. Extended version available as LIP Technical Report RR-99-17.

[14] Pierre Boulet, Jack Dongarra, Fabrice Rastello, Yves Robert, and Frédéric Vivien. Algorithmic issues on heterogeneous computing platforms. In *Clusters and Computational Grids Workshop*. to appear as a journal special issue, 1998. Extended version available as LIP Technical Report RR-98-49.

[15] Pierre Boulet, Jack Dongarra, Yves Robert, and Frédéric Vivien. Static tiling for heterogeneous computing platforms. *Parallel Computing*, 25:547–568, 1999.

[16] T.D. Braun, H.J. Siegel, N. Beck, L.L Bölöni, M. Maheswaran, A.. Reuther, J.P. Robertson, M.D. Theys, B. Yao, D. Hensgen, and R.F. Freund. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In *Eight Heterogeneous Computing Workshop*, pages 15–29. IEEE Computer Society Press, 1999.

[17] S. Browne, H. Casanova, and J. Dongarra. Providing Access to High Performance Computing Technologies. In J. Wasniewski, J. Dongarra, K. Madse, and D. Olesen, editors, *Applied Parallel Computing*, volume Springer-Verlag LNCS 1184, pages 123–1345. Springer-Verlag, 1996.

[18] M. Brune, J. Gehring, A. Keller, B. Monien, F. Ramme, and A. Reinefeld. Specifying resources and services in metacomputing environments. *Parallel Computing*, 24:1751–1776, 1998.

[19] R. Buyya. *High Performance Cluster Computing. Volume* 1*: Architecture and Systems*. Prentice Hall PTR, Upper Saddle River, NJ, 1999.

[20] R. Buyya. *High Performance Cluster Computing. Volume* 2*: Programming and Applications*. Prentice Hall PTR, Upper Saddle River, NJ, 1999.

[21] H. Casanova and J. Dongarra. Netsolve: A network server for solving computational science problems. Technical Report UT-CS-95-313, University of Tennessee, Knoxville, 1995.

[22] H. Casanova and J. Dongarra. Netsolve: A network server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997.

[23] 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).

[24] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5:173–184, 1996.

[25] E. Chu and A. George. QR factorization of a dense matrix on a hypercube multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 11:990–1028, 1990.

[26] 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.

[27] Michal Cierniak, Mohammed J. Zaki, and Wei Li. Scheduling algorithms for heterogeneous network of workstations. *The Computer Journal*, 40(6):356–372, 1997.

[28] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, San Francisco, CA, 1999.

[29] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Proc. IPPS/SPDP'98 Workshop on Job Scheduling Strategies for Parallel Processing*. IEEE Computer Society Press, 1998.

[30] J. Dongarra, R. van de Geijn, and D. Walker. Scalability issues in the design of a library for dense linear algebra. *Journal of Parallel and Distributed Computing*, 22(3):523–537, 1994.

[31] J. J. Dongarra and D. W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, 1995.

[32] 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.

[33] G. Eisenhauer, B. Plale, and K. Schwan. Data-exchange: high-performance communications in distributed laboratories. *Parallel Computing*, 24:1713–1733, 1998.

[34] M. Faerman, A. Su, R. Wolski, and F. Berman. Adaptive performance prediction for distributed data-intensive applications. In *Proceedings of Supercomputing 1999*. IEEE Computer Society Press, 1999. Available at `http://www.cs.ucsd.edu/groups/hpcl/apples/hetpubs.html`.

[35] 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.

[36] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thituvathukal, and S. Tuecke. Wide-area implementation of the message passing interface. *Parallel Computing*, 24:1735–1749, 1998.

[37] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[38] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1999.

[39] G. Fox, S. Otto, and A. Hey. Matrix algorithms on a hypercube i: matrix multiplication. *Parallel Computing*, 3:17–31, 1987.

[40] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine: A Users'Guide and Tutorial for Networked Parallel Computing*. The MIT Press, 1996.

[41] Andrew S. Grimshaw, Jon B. Weissman, Emily A. West, and Ed. C. Loyot Jr. Metasystems: an approach combining parallel processing and heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 21:257–270, 1994.

[42] A.S. Grimshaw and W.A. Wulf. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, 1997.

[43] M. Iverson and F. Özgüner. Dynamic, competitive scheduling of multiple dags in a distributed heterogeneous environment. In *Seventh Heterogeneous Computing Workshop*. IEEE Computer Society Press, 1998.

[44] Maher Kaddoura and Sanjay Ranka. Run-time support fo parallelization of data-parallel applications on adaptive and nonuniform computational environments. *Journal of Parallel and Distributed Computing*, 43:163–168, 1997.

[45] A. Kalinov and A. Lastovetsky. Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers. In P. Sloot, M. Bubak, A. Hoekstra, and B. Hertzberger, editors, *HPCN Europe 1999*, LNCS 1593, pages 191–200. Springer Verlag, 1999.

[46] T. Kielmann, R.F.H. Hofman, H.E. Bal, A. Plaat, and R.A.F. Bhoedjan. Magpie: Mpi's collective communication operations for clustered wide area systems. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP'99*, pages 131–140, Atlanta, GA, 1999.

[47] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[48] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[49] E. Laure, P. Mehrotra, and H. Zima. Opus: Heterogeneous computing with data parallel tasks. *Parallel Processing Letters*, 1999. To appear.

[50] M. Litzkow, M. Livny, and M.W. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111. IEEE Computer Society Press, 1988.

[51] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, and R.F. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Eight Heterogeneous Computing Workshop*, pages 30–44. IEEE Computer Society Press, 1999.

[52] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Seventh Heterogeneous Computing Workshop*. IEEE Computer Society Press, 1998.

[53] K. Keahey nd D. Gannon. Developing and evaluating abstractions for distributed supercomputing. *Cluster Computing*, 1:69–79, 1998.

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

[55] A. Petitet, H. Casanova, J. Dongarra, Y. Robert, and R.C. Whaley. Parallel and distributed scientific computing: A numerical linear algebra problem solving environment designer's perspective. In J. Blazewicz, K. Ecker, B. Plateau, and D.Trystram, editors, *Handbook on Parallel and Distributed Processing*. Springer Verlag, 1999. Available as LAPACK Working Note 139.

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

[57] M.C Rosu, K. Schwan, and R. Fujimoto. Supporting parallel applications on clusters of workstations: the virtual communication machines-based architecture. *Cluster Computing*, 1:51–67, 1998.

[58] Vivek Sarkar. *Partitioning and scheduling parallel programs for multiprocessors*. Pitman, 1989.

[59] H.J. Siegel, H.G. Dietz, and J.K. Antonio. Software support for heterogeneous computing. *ACM Computing Surveys*, 28(1):237–239, 1996.

[60] G.C. Sih and E.A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, 1993.

[61] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI the complete reference*. The MIT Press, 1996.

[62] M. Tan, H.J. Siegel, J.K. Antonio, and Y.A. Li. Minimizing the aplication execution time through scheduling of subtasks and communication traffic in a heterogeneous computing system. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):857–1871, 1997.

[63] H. Topcuoglu, S. Hariri, and M.Y. Wu. Task scheduling algorithms for heterogeneous processors. In *Eight Heterogeneous Computing Workshop*, pages 3–14. IEEE Computer Society Press, 1999.

[64] R. van Nieuwpoort, J. Maassen, H.E. Bal, T. Kielmann, and R. Veldema. Wide-area parallel computing in Java. In *ACM 1999 Java Grande Conference*, pages 8–14, San Francisco, CA, June 1999.

[65] J.B. Weissman and X. Zhao. Scheduling parallel applications in distributed networks. *Cluster Computing*, 1(1):109–118, 1998.

[66] Tao Yang and Apostolos Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel and Distributed Systems*, 5(9):951–967, 1994.