AN OVERVIEW OF HETEROGENEOUS HIGH PERFORMANCE AND GRID COMPUTING [*]

JACK DONGARRA[†] AND ALEXEY LASTOVETSKY[‡]

Abstract. This paper is an overview the ongoing academic research, development, and uses of heterogeneous parallel and distributed computing. This work is placed in the context of scientific computing.

The simulation of very large systems often requires computational capabilities which cannot be satisfied by a single processing system. A possible way to solve this problem is to couple different computational resources, perhaps distributed geographically.

Heterogeneous distributed computing is a means to overcome the limitations of single computing systems.

Key words. heterogeneous, parallel, grid, high performance

1. Introduction.

The recent availability of advanced-architecture computers has had a significant impact on all spheres of scientific computation. In last 50 years, the field of scientific computing has seen a rapid change of vendors, architectures, technologies and the usage of systems. Despite all these changes the evolution of performance on a large scale however seems to be a very steady and continuous process.

Two things remain consistent in the realm of computational science: i) there is always a need for more computational power than we have at any given point, and ii) we always want the simplest, yet most complete and easy to use interface to our resources. In recent years, much attention has been given to the area of Grid Computing. The analogy is to that of the electrical power grid. The ultimate goal is that one day we are able to plug any and all of our resources into this Computational Grid to access other resources without need for worry, as we do our appliances into electrical sockets today.

This paper will give an overview of the area of heterogeneous and grid computing.

2. Hardware platforms.

Heterogeneous hardware used for parallel and distributed computing always include:

- Multiple processors;
- Communication network interconnecting the processors.

Distributed memory multiprocessor systems can be heterogeneous in many different ways, but there is only one way for such a system to be homogeneous. Namely, all processors in the system have to be identical and interconnected via a homogeneous communication network, that is, a network providing communication links of the same latency and bandwidth between any pair of processors. But this definition is not complete. One more important restriction has to be satisfied: the system has to be dedicated, that is, at any time it can execute only one application providing all its resources to this application. We will see later how violation of this restriction makes the system heterogeneous.

Homogeneous distributed memory multiprocessor systems are designed for high performance parallel computing and typically used to run a relatively small number

[*] This work was supported in part by the Science Foundation Ireland and also in part by the Applied Mathematical Sciences Research Program of the Office of Mathematical, Information, and Computational Sciences, U.S. Department of Energy under contract DE-AC05-00OR22725 with UT-Battelle, LLC.

[†]University of Tennessee and Oak Ridge National Laboratory (dongarra@cs.utk.edu).

[‡]University College Dublin (Alexey.Lastovetsky@ucd.ie).

of similar parallel applications.

The property of homogeneity is easy to break and may be quite expensive to keep. Any distributed memory multiprocessor system will become heterogeneous if it allows several independent users to simultaneously run their applications on the same set of processors. The point is that in this case different identical processors may have different workload and hence demonstrate different performance for different runs of the same application depending on external computations and communications.

Clusters of commodity processors are seen a cheap alternative to very expensive vendor homogeneous distributed memory multiprocessor systems. But they may become quite expensive if you try and keep their homogeneity for relatively long time. First of all, they cannot be used as multiprocessing computer systems as such a usage immediately makes them heterogeneous because of the dynamic change of the performance of each particular processor. Secondly, the cluster cannot be upgraded in part. If processors in the cluster become obsolete and slow, you can only replace all them simultaneously by more powerful ones, otherwise you lose homogeneity by getting non-identical processors in your cluster. Therefore, any upgrade of the existing cluster, which keeps its homogeneity, is a costly operation.

Thus, distributed memory multiprocessor systems are naturally heterogeneous, and the property of heterogeneity is an intrinsic property of the overwhelming majority of such systems. Next we would like to classify the systems in the increasing order of heterogeneity and complexity and briefly characterize each heterogeneous system. The classes are:

- Heterogeneous clusters;
- Local networks of computers;
- Organizational networks of computers;
- Global general purpose networks of computers.

Heterogeneous cluster is still a dedicated computer system designed mainly for high performance parallel computing, which is obtained from the classical homogeneous cluster architecture by relaxing one of its three key properties and leading to the situation when:

- Processors in the cluster may not be identical.
- The communication network may have a regular but heterogeneous structure (for example, it can consist of a number of faster communication segments interconnected by relatively slow links).
- The cluster may be a multi-user computer system (but still dedicated to high performance parallel computing). As we have discussed, this, in particular, makes the performance characteristics of the processors dynamic and non-identical.

The heterogeneity of the processors can take different forms. The processors can be of different architectures. They may be of the same architecture but of different models. They may be of the same architecture and model but running different operating systems. They may be of the same architecture and model and running the same operating system but configured differently or using different basic software (compilers, run-time libraries, etc). All the differences can have an impact on performance and some other characteristics of the processors.

In terms of parallel programming, the most demanding is a multi-user heterogeneous cluster made up of processors of different architectures interconnected via heterogeneous communication network.

In the general case, **local network of computers** (LNC) consists of diverse computers interconnected via mixed network equipment. By its nature, LNCs are multi-user computer systems. Therefore, just like highly heterogeneous clusters, LNCs consist of processors of different architectures, which can dynamically change their performance characteristics, interconnected via heterogeneous communication network.

Unlike heterogeneous clusters, which are a parallel architecture designed mainly for high performance computing, LNCs are general-purpose computer systems typically associated with individual organizations. This impacts the heterogeneity of this architecture in several ways. First of all, the communication network of a typical LNC is not that regular and balanced as in heterogeneous clusters. The topology and structure of the communication network in such a LNC is determined by many different factors, among which high performance computing is far away from being a primary one if considered at all. The primary factors include the structure of the organization, the tasks that are solved on computers of the LNC, the security requirements, the construction restrictions, the budget limitations, the qualification of technical personnel, etc. An additional important factor is that the communication network is constantly developing rather than fixed once and forever. The development is normally occasional and incremental. Therefore, the structure of the communication network reflects the evolution of the organization rather than its current snapshot. All the factors make the communication network of the LNC extremely heterogeneous and irregular. Some communication links in this network may be of very low latency and/or low bandwidth.

Secondly, different computers may have different functions in the LNC. Some computers can be relatively isolated. Some computers may provide services to other computers of the LNC. Some computers provide services to both local and external computers. This makes different computers have different levels of integration into the network. The heavier the integration is, the more dynamic and stochastic is the workload of the computer, and the less predictable become its performance characteristics. Another aspect of this functional heterogeneity is that a heavy server is normally configured differently compared to ordinary computers. In particular, it is typically configured to avoid paging and hence to avoid any dramatic drop in performance with the growth of requests to be served. At the same time, this results in abnormal termination of any application that tries to allocate more memory than it fits into the main memory of the computer leading to the loss of continuity of its characteristics.

Thirdly, in general-purpose LNCs different components are not as strongly integrated and controlled as in heterogeneous clusters. LNCs are much less centralized computer system than heterogeneous clusters. They consist of relatively autonomous computers, each of which may be used and administered independently by its users. As a result, their configuration is much more dynamic than that of heterogeneous clusters. Computers can come and go just because their users switch them on and off, or re-boot them.

Unlike a local networks of computers, all components of which are situated locally a **global network of computers** (GNC) includes computers that are geographically distributed. There are three main types of GNCs, which are briefly presented in the increasing order of their heterogeneity. The first type of GNCs is a dedicated system for high performance computing that consists of several interconnected homogeneous distributed memory multiprocessor systems or/and heterogeneous clusters. Apart from geographical distribution of its components, such a computer system is similar to heterogeneous clusters.

GNC of the second type is a organizational network. Such a network comprises geographically distributed computer resources of some individual organization, such as a company or university department. The organizational network can be seen as a geographically extended LNC. It is typically very well managed by a strong team of hardware and software experts. Its level of integration, centralization and uniformity is often even higher than that of LNCs. Therefore, apart from geographical distribution of their components, organizational networks of computers are quite similar to local networks of computers.

Finally, GNC of the third type is a general-purpose global network of computers. Such a network consists of individual computers interconnected via Internet. Each of the computers is managed independently. This is the most heterogeneous, irregular, loosely integrated and dynamic type of heterogeneous network.

Of course our list of heterogeneous distributed memory multiprocessor systems is not comprehensive. We only outlined systems that are most relevant for scientific computing. Some other examples of heterogeneous sets of interconnected processing devices are given by:

- Mobile telecommunication system with different types of processors, from ones embedded into mobile phones to central computers processing calls;
- Embedded control multiprocessor systems (cars, airplanes, spaceships, household, etc.).

3. Typical Uses of Heterogeneous Networks. In this section, we outline how heterogeneous networks of computers are typically used by their end-users. In general, heterogeneous networks are used traditionally, for parallel computing or for distributed computing.

The traditional use means that the network of computers is used just as an extension of the user's computer. This computer can be serial or parallel. The application to be run is a traditional application, that is, the one that can be executed on the user's computer. The code of the application and input data are provided by the user. The only difference from the fully traditional execution of the application is that it can be executed not only on the user's computer but on any other relevant computer of the network. The decision where to execute one or other application is made by the operating environment and mainly aimed at better utilization of available computing resources (e.g., at higher throughput of the network of computers as a whole multi-user computer system). Faster execution of each individual application is not the main goal of the operating environment but can be achieved for some applications as a side-effect of its scheduling policy. This use of the heterogeneous network assumes the application and hence the software is portable and can be run on another computing resource. This assumption may not be true for some applications.

A heterogeneous network of computers can be used for **parallel computing**. The network is used as a parallel computer system in order to accelerate the solution of a single problem. In this case, the user provides a dedicated parallel application written to efficiently solve the problem on the heterogeneous network of computers. High performance is the main goal of that type of use. As in the case of traditional use, the user provides both the (source) code of the application and input data. In the general case, when all computers of the network are of the different architecture, the source code is sent to the computers where it is locally compiled. All the computers are supposed to provide all libraries necessary to produce local executables.

A heterogeneous network of computers can be also used for **distributed computing**. In the case of parallel computing, the application can be executed on the user's computer or on any other single computer of the network. The only reason to involve more than one computer is to accelerate the execution of the application. Unlike parallel computing, distributed computing deals with situations when the application cannot be executed on the user's computer just because not all components of the application are available on this computer. One such a situation is that some components of the code of the application cannot be provided by the user and are only available on remote computers. The reasons behind this can be various: the user's computer may not have the resources to execute such a code component, or the efforts and amount of resources needed to install the code component on the user's computer are too significant compared with the frequency of its execution, or this code may be just not available for installation, or it may make sense to execute this code only on the remote processor (say, associated with an ATM machine), etc.

Another situation is that some components of input data for this application cannot be provided by the user and reside on remote storage devices. For example, the size of the data may be too big for the disk storage of the user's computer, or the data for the application are provided by some external party (remote scientific device, remote data base, remote application, and so on), or the executable file may not be compatible with the machine architecture, etc.

The most complex is the situation when both some components of the code of the application and some components of its input data are not available on the user's computer.

4. Programming Issues: Results and Open Problems. Programming for heterogeneous networks of computers is a difficult task. Among others, performance and fault tolerance are probably the most important and challenging issues of heterogeneous parallel and distributed programming. Performance is the primary issue of parallel programming for any parallel architecture but becomes particularly challenging for parallel programming for heterogeneous networks. The issue of performance is also primary for high performance distributed computing.

Fault tolerance has always been one of the primary issues of distributed computing. At the same time, it never used to be a primary issue for parallel applications running on traditional homogeneous parallel architectures. The probability of unexpected resource failures in a centralized dedicated parallel computer system was quite small because the system had a relatively small number of processors. This only becomes an issue for modern large-scale parallel systems counting tens thousand processors. At the same time, this probability reaches quite high figures for common networks of computers of even a relatively small size. Firstly, any individual computer in such a network may be switched off or rebooted unexpectedly for other users in the network. The same may happen with any other resource in the network. Secondly, not all building elements of the common network of computers are equally reliable. These factors make fault tolerance a desirable feature for parallel applications intended to run on common networks of computers; and the longer the execution time of the application is, the more important the feature becomes.

In this section we analyze these two issues with respect to parallel and distributed programming for heterogeneous networks of computers, then outline recent research results in these areas, and finally formulate some open problems needed further research.

4.1. Performance. An immediate implication from the heterogeneity of processors in a network of computers is that the processors run at different speeds. A good parallel application for a homogeneous distributed memory multiprocessor system tries to evenly distribute computations over available processors. This very distribution ensures the maximal speedup on the system consisting of identical processors. If the processors run at different speeds, faster processors will quickly perform their part of computations and begin waiting for slower ones at points of synchronization and data transfer. Therefore, the total time of computations will be determined by the time elapsed on the slowest processor. In other words, when executing parallel applications, which evenly distribute computations among available processors, the heterogeneous network will demonstrate the same performance as a network of interconnected identical processors equivalent the slowest processor of the heterogeneous network of computers.

Therefore, a good parallel application for the heterogeneous network must distribute computations unevenly taking into account the difference in processor speed. The faster the processor is, the more computations it must perform. Ideally, the volume of computation performed by a processor should be proportional to its speed.

The problem of distribution of computations in proportion to the speed of processors is typically reduced to the problem of partitioning of some mathematical objects such as sets, matrices, graphs, etc. Optimal data partitioning over a set of heterogeneous processors has attracted constantly growing attention of researchers in past 10 years. A number of interesting mathematical problems have been formulated and investigated [13, 22, 24, 25, 7, 6, 5, 16, 33]. In a generic form, a typical partitioning problem is formulated as follows:

- Given a set of processors, the speed of each of which is characterized by a positive constant,
- Partition a mathematical object into sub-objects of the same type (a set into sub-sets, a matrix into sub-matrices, a graph into sub-graphs, etc) so that
  - There is one-to-one mapping between the partitions and the processors,
  - The size of each partition (the number of elements in the sub-set or sub-matrix, the number of nodes in the sub-graph) is proportional to the speed of the processor owing the partition (that is, it is assumed that the volume of computation is proportional to the size of the processed mathematical object),
  - Some additional restrictions on the relationship between the partitions are satisfied (for example, the sub-matrices of the matrix may be required to form a two-dimensional $p \times q$ arrangement, where p and q may be either given constants or the parameters of the problem, the optimal value of which should be also found),
  - The partitioning minimizes some functional, which is used to measure each partitioning (for example, minimizes the sum of the perimeters of the rectangles representing the sub-matrices; intuitively, this functional measures the volume of communications for some parallel algorithms).

The investigated problems mainly deal with partitioning matrices because matrices are probably the most widely used mathematical objects in scientific computing. A general problem of optimal partitioning a square into rectangles with no restrictions on the shape and arrangement of the rectangles was studied in [7, 6] and proved to be NP-complete. It has been also proved that the optimal column-based partitioning that minimizes the sum of the perimeters of the rectangles could be achieved in polynomial time. A version of this optimal column-based partitioning obtained under the additional restriction that the rectangles must be arranged into a given two-dimensional $p \times q$ grid was originally suggested and analyzed in [24]. The latter partitioning can be easily achieved in linear time.

Another issue in optimal matrix partitioning is that even given a shape of 2D arrangement of the rectangles and a set of processors, there is still freedom in mapping of the processors onto the shape. For example, different mappings of the processors onto the $p \times q$ grid may influence the performance of the underlying dense matrix multiplication algorithm. This issue was addressed in [5] in relation to the 2D dense matrix multiplication problem. The optimization problem was proved to be NP-hard. At the same time, it was found that the optimal mapping should arrange processors in a non-increasing order of their speed along each row and each column of the 2D arrangement.

Another interesting question is how to select the shape of processors arrangement for the heterogeneous parallel algorithm if we have, say, 12 processors. Should we use 2×6, 3×4, 4×3, or 6×2? This problem was addressed in [28] in relation to the 2D block-cyclic dense matrix multiplication algorithm. In the experiments, the size of computation unit, $\mathbf{r}$, the size of generalized block, $\mathbf{l}$, and the shape of arrangement, $p \times q$ were varied. It was found that the shape of arrangement has a visible (sometimes quite significant) impact on the efficiency of the algorithm if values of $\mathbf{r}$ and $\mathbf{l}$ are far away from the optimal ones. At the same time, it practically does not influence the execution time of the algorithm if the parameters be optimal.

Summarizing the results, we conclude that only the very first steps have been taken in theoretical and experimental research on optimal data partitioning across a set heterogeneous processors, the speed of each of which is represented a constant positive number. More algorithms should be investigated together with the influence of different parameters of the algorithms on their performance.

Performance analysis of the heterogeneous parallel algorithms is also an open issue. Design of heterogeneous parallel algorithms is typically reduced to the problem of optimal data partitioning of one or other mathematical object such as a set, a matrix, etc. As soon as the corresponding mathematical optimization problem is formulated, the quality of its solution is assessed rather than the quality of solution of the original problem. As the optimization problem is typically NP-hard, some sub-optimal solutions are proposed and analyzed. The analysis is typically statistical: the sub-optimal solutions for a big number of generated inputs are compared to each other and the optimal one. This approach is used in many papers, in particular, in [13, 22, 7, 6]. This approach estimates the heterogeneous parallel algorithm indirectly and additional experiments are still needed to assess its efficiency in real heterogeneous environments.

Another approach is just to experimentally compare the execution time of the heterogeneous algorithm with that of its homogeneous prototype or a heterogeneous competitor. Some particular heterogeneous network is used for such experiments. In particularly, this approach is used in [24, 25, 16, 33]. This approach directly estimates the efficiency of heterogeneous parallel algorithms in some real heterogeneous environment but still leaves an open question about their efficiency in other particular heterogeneous environments.

The approach proposed in [28] is to carefully design a relatively small number of experiments in a real heterogeneous environment in order to experimentally compare the efficiency of the heterogeneous parallel algorithm with some experimentally obtained ideal efficiency (namely, the efficiency of its homogeneous prototype in an equally powerful homogeneous environment). This approach directly estimates the efficiency of heterogeneous parallel algorithms providing relatively high confidence in the results of such an experimental estimation.

An ideal approach would be to use a comprehensive performance model of heterogeneous networks of computers for analysis of the performance of heterogeneous parallel algorithms in order to predict their efficiency without real execution of the algorithms in heterogeneous environments. We will come back to this issue later in the overview.

Several authors consider scalability more important property of heterogeneous parallel algorithms than efficiency. They propose some approaches to analysis of scalability of heterogeneous parallel algorithms [38, 23].

Yet another approach to parallel programming for heterogeneous networks can be summarized as follows:

- The whole computation is partitioned into a large number of equal chunks;
- Each chunk is performed by a separate process;
- The number of processes run by each processor is proportional to the relative speed of the processor.

Thus, while distributed evenly across parallel processes, data and computations are distributed unevenly over processors of the heterogeneous network so that each processor performs the volume of computations proportional to its speed. More details on this approach can be found in [25].

Whatever partitioning algorithm is selected for distribution of computations in proportion to the speed of processors, it should be provided with a set of positive constants representing the relative speed of the processors. The efficiency of the corresponding application will strongly depend on the accuracy of estimation of the relative speed. If this estimation is not accurate enough, the load of processors will be unbalanced, resulting in poorer execution performance.

Traditional approach to this problem is to run some test code once to measure the relative speed of the processors of the network and then use this estimation when distributing computation across the processors. This approach is used in [2].

Unfortunately, the problem of accurate estimation of the relative speed of processors is not as easy as it may look. Of course, if you consider two processors, which only differ in clock rate, it is not a problem to accurately estimate their relative speed. You can use a single test code to measure their relative speed, and the relative speed will be the same for any application. This approach may also work if the processors that you use in computations have very similar architectural characteristics.

But if you consider processors of really different architectures, the situation changes drastically. Everything in the processors may be different: set of instructions, number of instruction execution units, number of registers, structure of memory hierarchy, size of each memory level, and so on, and so on. Therefore, the processors may demonstrate different relative speeds for different applications. Moreover, processors of the same architecture but different models or configurations may also demonstrate different relative speeds on different applications. Even different applications of the same narrow class may be executed by two different processors at significantly different relative speeds.

Another complication comes up if the network of computers allows for multiprocessing. In this case, the processors executing your parallel application may be also used for other computations and communications. Therefore, the real performance of the processors can dynamically change depending on the external computations and communications. One approach to this problem is to measure the level of the current workload of each processor, $w \in [0, 1]$, and correct its relative speed by multiplying by this coefficient [9]. At the same time, this simple and efficient approach still needs solid theoretical and experimental research on the accuracy of estimation of the relative speed obtained this way.

A combined approach to solution of these two related problems is proposed in the mpC programming language [26]. In brief, mpC allows the application programmer to explicitly specify the test code that is representative for computations in different parts of the application. It is supposed that the relative speed demonstrated by the processor during the execution of this test code is approximately the same as the relative speed of the processors demonstrated during the execution of the corresponding part of the application. It is also supposed that the time of execution of this code is much less then the time of the execution of the corresponding part of the application. The application programmer uses the **recon** statement to specify both the test code and when it should be run. During the execution of this statement, all processors execute this test code in parallel, and the execution time elapsed on each processor is used to obtain their relative speed. Thus, the accuracy of estimation of the relative speed of the processors is fully controlled by the application programmer. The code provided to measure the relative speed and the points and frequency of the **recon** statements in the program are based on the knowledge of both the implemented parallel algorithm and the executing network of computers.

The accuracy of estimation of the relative speed of the processors depends not only on how representative is the test code used to obtain the relative speed or how frequently this estimation is performed during the execution of the application. There are some objective factors that do not allow us to estimate the speed of some processors accurately enough. One of these factors is the level of integration of the processor in the network. As we have discussed, in general-purpose local and global networks integrated into the Internet most computers constantly run some processes interacting with the internet, and some computers act as servers for other computers. This results in constant unpredictable fluctuations in the workload of processors in such networks. Some very preliminary results presented in [29] show that this fluctuations can range from 5% for relatively autonomous computers to over 40% for heavy servers. Much more extensive experimental research is still needed to classify computers based on their level of integration into the network and incorporate this classification into models, algorithms and software for high performance computing on general-purpose heterogeneous networks of computers. We would like to stress that this problem is specific for general-purpose local and global heterogeneous networks. Heterogeneous clusters are much more regular and predictable in this respect.

We have seen that the relative speed of heterogeneous processors can depend on the code and the external workload of the processors. At the same time, one can see that the design of the outlined heterogeneous parallel algorithms implicitly assumes that the relative speed does not depend on the size of the problem solved by the algorithm. Indeed, the formulation of a typical data partitioning problem clearly states that the speed of each processor is characterized by a positive constant, and the data partitioning algorithms strongly rely on this assumption. The assumption is quite satisfactory if the code executed by the processors fully fits into the main memory. But as soon as this restriction is relaxed it will not be true.

First of all, two processors may have different sizes of the main memory. Therefore, beginning from some critical problem size, the application will still fit into the main memory of one processor and not fit in the main memory of the other, causing the paging and visible degradation of the speed of the processor. This means that their relative speed will not be a constant function of the problem size for problem sizes greater than this critical value. Secondly, even if the two processors have the main memory of the same size but are of different architectures, they may differ in how the paging affects their performance. Again, beginning from some critical problem size that causes paging, their relative speed may change compared to that without paging. Moreover their relative speed may differ for different sizes of the problem causing the paging. Some experimental results on this can be found in [29, 27].

This significantly complicates the design of algorithms distributing computations in proportion with the relative speed of heterogeneous processors. One approach to this problem is just to avoid the paging as it is normally done in the case of parallel computing on homogeneous multi-processors. It is quite easy to calculate the maximal size of the problem (in the total size of stored data) that can be executed on a homogeneous parallel system without paging: the size of the main memory is multiplied by the number of processors. But it is not that easy even for heterogeneous clusters. At a first glance, in the case of heterogeneous cluster that maximal size might be calculated just by summing the sizes of the main memory of the processors. But the point is that the relative speed of heterogeneous processors may differ from the relative size of their main memory. Therefore distribution of data in proportion to the relative speed of the processors may result in the data not fitting into the main memory of some processors. Thus, to avoid paging when distributing computation across processors of the heterogeneous cluster, a more advanced performance model of the cluster is needed that would characterize each processor not only by its relative speed but also by the maximal allowed problem size. Correspondingly, data partitioning with such a model will be a more difficult task.

Avoiding paging in local and global networks may not make sense because in such networks it is likely to have one processor running in the presence of paging even faster than other processors without paging. It is even more difficult to avoid paging in the case of distributed computing on global networks. There just may not be a server available to solve the task of the size you need without paging.

Therefore, to achieve acceptable accuracy of distribution of computations across heterogeneous processors in the possible presence of paging, a more realistic performance model of a set of heterogeneous processors is needed. This model should characterize the speed of each processor not by a constant but by a function of the problem size. How to build and efficiently maintain such a model is an open problem. Data partitioning with this model will be also a much more difficult problem. At the same time, some preliminary research results show that efficient algorithms of data partitioning with this model can be designed. For example, a problem of partitioning of an **n**-element set over **p** heterogeneous processors using this advanced performance model has an efficient solution of the complexity $\mathrm{O}(\mathbf{p}^2 \times \log_2 \mathbf{n})$ (see [27]).

There is one more factor, which has a significant impact on the optimal distribution of computations over heterogeneous processors but not been taken into account so far. This factor is the communication network interconnecting the processors. This factor can be ignored if the contribution of communication operations in the total execution time of the application is negligibly small compared to that of computations. Otherwise, any algorithm of distribution of computations aimed at the minimization of the total execution time should take into account not only **p** heterogeneous processors but also up to $\mathbf{p}^2$ communication links. This increases the complexity of the optimization problem even under assumption that the optimal solution distributes computations over all available processors. If some of the communication links are of low latency or/and low bandwidth, then it may be profitable to involve in computations not all available processors just because the cost of communication via these links may not compensate gains due to extra parallelization. In this case, the problem of optimal distribution of computations becomes much more complex as the space of possible solutions is significantly increasing including distributions not only over all available processors but also distributions over every their subset.

As soon as communication operations are taken into consideration, the characterization of heterogeneous processors by their relative speed will make no sense any more. The point is that the relative speed cannot help to estimate the contribution of computations in the total execution time of the application. The time of execution of computations on a processor can only be calculated if the processor is characterized by its absolute speed, that is, the volume of computation that it performs in a unit time. Then, given a volume of computations to be performed on the processor, the time of execution of the computations can be easily calculated. Similarly, the execution time of communication operations can only be calculated if the performance of communication links is characterized by absolute units of performance, not by relative ones.

It is not a problem to specify the performance of a communication link in absolute units. For example, *byte-per-second* can be used to do it. Given a size in bytes of the data block transferred via the communication link, the time of execution of the communication operation can be calculated.

Specification of the absolute speed of the processors is not such an easy problem. What is the natural unit of computation to measure the volume of computations performed by the processor per second? One approach is to use the same units that are used in mathematics to estimate the complexity of the algorithm, that is, the number of arithmetical operations, say, Mflop (a million floating point operations). Unfortunately such an average characteristic of the speed of the processor is not useful in accurate prediction of the execution time of particular applications. We have discussed that two processors of different architectures may demonstrate significantly different relative speeds for different applications. Equally two processors of different architectures, which are characterized by the same average speed (say, the theoretical peak expressed in Mflop/sec), may execute the same application in significantly different times.

One solution is to increase the granularity of computation units used in the measurement of the processor speed and make the units sensitive to the computations performed by the particular application. In other words, the processor speed has to be measured in the context of the application. This approach is proposed and implemented in the mpC programming language [26]. The speed of each processor is measured in computation units per second where the computation unit is a piece of code provided by the application programmer. It is supposed that the code provided by the application programmer is representative for the computations performed during the execution of the application. Therefore, given an absolute volume of computations to be performed by the processor, which is expressed in these units, the execution time of the application can be accurately calculated.

Let us summarize our brief analysis of the performance-related programming issues in parallel and distributed computing for heterogeneous networks of computers.

There are two basic related issues that should be addressed:

- Performance model of heterogeneous network of computers quantifying the ability of the network to perform computations and communications,
- Performance model of application quantifying the computations and communications to be performed by the application.

In the presence of such models, given parameters of the application model, parameters of the model of the executing network of computers and a mapping of the application to the processors of the network, it would be possible to calculate the execution time of the application on the network. Application and system programmers could incorporate the models in their applications and programming systems to achieve better distribution of computations and communications leading to higher performance.

The performance model of heterogeneous network of computers includes two submodels:

- Performance model of a set of heterogeneous processors which is used to estimate the execution time of computations,
- Performance model of communication network which is necessary to predict the execution time of communication operations.

Now we briefly outline performance models currently used in programming systems and operating environments for heterogeneous networks of computers.

Network Weather Service (NWS) [40] is a popular software tools for monitoring networks of heterogeneous computers and providing information about the performance of their components. Many distributed programming systems rely on NWS in obtaining the model of the underlying heterogeneous network [2, 1]. NWS supports the following performance model:

- In the absence of external load and paging, all processors are supposed to demonstrate the same speed;
- Each processor is characterized by the two parameters:
  - Current CPU utilization, $u \in [0, 1]$,
  - The size of available main memory;
- Communication link between two processors is characterized by the two parameters: the *latency* and the *bandwidth*.

Condor [31] is a batch job system that schedules jobs, submitted by users, to non-dedicated computers of the underlying heterogeneous network. Its main goal is to utilize desktop computers when they are not intensively used by their owners. The core of the system is a matchmaker that matches each submitted job with a computer suitable to execute the job. Each submitted job is scheduled separately and independently on other submitted jobs. No computer can be matched with more than one submitted task. Thus, the matchmaker deals with a sequence of independent jobs, each of which should be scheduled to one of currently under-used computers. The scheduling algorithm is based on the description of the job, which is provided by the user submitted the job, and the description of the computers, provided by their owners. The job is described by a number of parameters specifying, in particular,

- the required computer hardware and software (e.g., the processor architecture and operating system),
- the resource requirements (e.g., the required memory and disk space), and
- the restrictions on the use of the resources (i.e., no paging during the execution of the job, which can be specified by an inequality stating that the required memory should be less than the available memory of the computer).

Only computers satisfying the restrictions and requirements are considered for the job.

On the other hand, each computer is described by a number of parameters specifying, in particular,

- its hardware and basic software (e.g., processor architecture, operating system),
- performance characteristics (e.g., Mips and KFlops),
- available resources (memory and disk space),
- the current state of the computer (e.g., the current time, CPU utilization, if the computer is claimed or not, how long the keyboard has been idle);
- conditions that must be satisfied to make the computer available for scheduling (e.g., it can be only used after 5pm if the keyboard has been idle for more than 15 minutes and the CPU utilization is less than 30%).

Only unclaimed computers satisfying the conditions are considered available for scheduling.

If there are several computers that can execute the job, one more additional performance–related characteristic of the job will be used to make a final decision. This characteristic is a formula for calculation of the rank of each pair (job, computer). It is supposed that the formula calculates some combined performance characteristic of the execution of the job on the computer by using both the resource requirements of the job and the performance characteristics and available resources of the computer. The formula is provided by the user and is a part of the job description. The computer making the highest rank is selected for the job.

NetSolve [2, 1] is a programming system for high performance distributed computing on global networks using a remote procedure call mechanism. It deals with the situation when some components of the application cannot be provided by the user and are only available on remote computers. To program a NetSolve application, the user writes a NetSolve client program, which is any program (say, in C or Fortran) with calls to the NetSolve client interface. Each call specifies the name of the remote task to be performed, pointers to the data on the user's computer required by the task, and pointers to locations on the user's computer where the results will be stored. When the program runs, a NetSolve call results in a task to be executed on a remote computer. The NetSolve programming system is responsible for selection of the remote computer to perform the task, transferring input data from the user's computer to the remote computer, and delivering output data from the remote computer to the user's one.

The mapping of the remote tasks of the NetSolve application to computers of the global networks is the core operation having a major impact on the performance of the application. Each task is mapped separately and independently of other tasks of the application. Thus, the NetSolve agent responsible for the mapping deals with a sequence of independent tasks, each of which should be mapped to one of the remote computers that are able to perform the task. The mapping algorithm is based on two performance models:

- the performance model of the heterogeneous networks of computers,
- the performance model of the task.

The heterogeneous network of computers is seen as a set of interconnected heterogeneous processors. The performance of each processor is characterized by the execution time of the same serial code multiplying two dense square matrices of some fixed size. This characteristic of the processor is obtained once on the installation of the NetSolve software and does not change. Communication links are characterized the same way as in NWS. Unlike the performance model of processors, the communication model is dynamic. The NetSolve system periodically updates the performance characteristics of communication links.

The performance model of each task is provided by person who installs the task on a remote computers. A formula is associated with each task that can be used to calculate the execution time of the task by the solver. The formula uses parameters of the task and the execution time of the standard computation unit on the computer (which is currently the multiplication of two dense square matrices of the fixed size). Apart from the formula, the sizes of input and output data are also specified. Thus, the performance model of each task comprises a distributed set of performance models associated with each task.

The mapping algorithm tries to minimize the total execution time of the task, which is calculated as the sum of the time of computation on the remote computer and the time of communications between the user's computer and the remote one to deliver input data and receive output data. To estimate the contribution of computation in the total execution time for each possible mapping, the agent evaluates the formula associated with corresponding remote task. To estimate the contribution of communication, the agent uses the current characteristics of the communication link between the user's computer and the remote computer, and the sizes of input and output data that should be transferred between the computers.

mpC is a programming language for parallel computing on heterogeneous networks. It allows the application programmer to implement their heterogeneous parallel algorithms by using high level language abstractions rather than going into details of the message passing programming model of the MPI level. In addition, it takes care of the optimal mapping of the algorithm to the computers of the executing heterogeneous networks. This mapping is performed at runtime by the mpC programming system and used on two performance models:

- the performance model of the implemented algorithm,
- the performance model of the executing heterogeneous network.

The performance model of the heterogeneous network of computers is summarized as follows:

- The performance of each processor is characterized by the execution time of the same serial code
  - The serial code is provided by the application programmer.
  - It is supposed that the code is representative for the computations performed during the execution of the application.
  - The code is performed at runtime in the points of the application specified by the application programmer. Thus, the performance model of the processors provides current estimation of their speed demonstrated on the code representative for the particular application.
- The communication model is seen as a hierarchy of homogeneous communication layers. Each is characterized by the latency and bandwidth. Unlike the performance model of processors, the communication model is static. Its parameters are obtained once on the initialisation of the environment and do not change since then.

The performance model of the implemented algorithm is provided by the application programmer and is a part of the mpC application. The model is specified in a generic form and includes:

- The number of processes executing the algorithm (which is normally a parameter of the model).

- The total volume of computation to be performed by each process during the execution of the algorithm.
  - The volume is specified in the form of formula including the parameters of the model.
  - The volume of computation is measured in computation units provided by the application programmer (the very code which has been used to characterize the performance of processors of the executing heterogeneous network).
- The total volume of data transferred between each pair of the processes during the execution of the algorithm.
- How exactly the processes interact during the execution of the algorithm, that is, how the processes perform the computations and communications (which computations are performed in parallel, which are serialized, which computations and communication overlap, etc.).
  - To specify the interaction of the processes, traditional serial and parallel are used such as **for**, **while**, **parallel for**, etc.
  - Expressions in the statements specify the amount of computations or communications rather than the communications and computations themselves. Parameters of the algorithm and locally declared variables are widely used in that description.

The mpC compiler will translate this model specification into the code calculating the total execution time of the algorithm for every mapping of the processes of the application to the computers of the heterogeneous network. In the mpC program, the programmer can specify all parameters of the algorithm. In this case, the mpC programming system will try to find the mapping of the fully specified algorithm minimizing its estimated execution. At the same time, the programmer can leave some parameters of the algorithm unspecified (for example, the total number of processes executing the algorithm can be unspecified). In that case, the mpC programming system tries to find both the optimal value of unspecified parameters and the optimal mapping of the fully-specified algorithm.

4.2. Fault Tolerance. Today, end-users and application developers of high performance computing systems have access to larger machines and more processors than ever before. Systems such as the Earth Simulator, the ASCI-Q machines and the IBM Blue Gene consist of thousands or even tens of thousand of processors. Machines comprising 100,000 processors are expected within the next few years. A critical issue of systems consisting of such large numbers of processors is the ability of the machine to deal with process failures. Based on the current experience with high-end machines, it can be concluded, that a 100,000-processor machine will experience a processor failure every few minutes. While on earlier massively parallel processing systems (MPPs) crashing nodes often lead to a crash of the whole system, current architectures are more robust. Typically, the applications utilizing the failed processor will have to abort, the machine, as an entity is however not affected by the failure.

Although MPI [37] is currently the de-facto standard system used to build high performance applications for both clusters and dedicated MPP systems, it is not without it problems. Initially MPI was designed to allow for very high efficiency and thus performance on a number of early 1990s MPPs, which at the time had limited OS runtime support. This led to the current MPI design of a static process model. While this model was possible to implement for MPP vendors, easy to program for, and more importantly something that could be agreed upon by a standards committee. The second version of MPI standard known as MPI-2 [21] did include some support for dynamic process control, although this was limited to the creation of new MPI process groups with separate communicators. These new processes could not be merged with previously existing communicators to form intracommunicators needed for a seamless single application model and were limited to a special set of extended collectives (group) communications.

The MPI static process model suffices for small numbers of distributed nodes within the currently emerging masses of clusters and several hundred nodes of dedicated MPPs. Beyond these sizes the mean time between failures (MTBF) of CPU nodes starts becoming a factor. As attempts to build the next generation Peta-flop systems advance, this situation will only become more adverse as individual node reliability becomes out weighted by orders of magnitude increase in node numbers and hence node failures.

Summarizing the findings of this section, there is a discrepancy between the capabilities of current high performance computing systems and the most widely used parallel programming paradigm. When considering systems with tens of thousand of processors, the only currently available technique to handle fault tolerance, checkpoint / restart, has its performance and conceptual limitations. However, if today's and tomorrow's high performance computing resources were to be used as a means to perform single, large scale simulations and not solely as a platform for high throughput computing, extending the core communication library of HPC systems to deal with aspects of fault-tolerance is inevitable.

For parallel and distributed systems, the two main sources of failure/disconnection are the nodes and the network. Human factors (machine/application shutdown, network disconnection), hardware or software faults may also be at the origin of failures/disconnections. For the sake of simplicity, we consider the failures/disconnections as node volatility: the node is no more reachable and the eventual results computed by this node after the disconnection will not be considered in the case of a later reconnection.

Fault Tolerant MPI (FT-MPI) [17] is an independent implementation of the MPI 1.2 message passing standard that has been built from the ground up offering both user and system level fault tolerance. The FT-MPI library gives application developers the ability to build fault tolerant or survivable applications that do not immediately exit due to the failure of a processor, node, or MPI task. To reach this goal, FT-MPI extends the current MPI specification, playing a leading role in current world-wide efforts to improve the error-handling of MPI applications.

FT-MPI is an efficient MPI implementation of the MPI standard and its performance is comparable to other public MPI implementations. This has been achieved though the use of optimized data type handling, an efficient point to point communications progress engine and highly tuned and configurable collective communications.

The aim of FT-MPI is to build a fault tolerant MPI implementation that can survive failures, while offering the application developer a range of recovery options other than just returning to some previous check-pointed state. FT-MPI is built on the HARNESS [8] meta-computing system, and is meant to be used as its default application level message passing interface.

5. Software.

The world of parallel programming today is diverse and complex. New parallel programming systems are designed and developed all over the world, yet few are in practical use. Many of the new systems show great promise, but do not scale well to *real world applications*. Others never leave the testing state. The few systems that seem to be used for real applications are mostly more than 5 years old (i.e. *MPI*, *OpenMP*).

5.1. Parallel Programming Systems. Parallel programming systems can be categorized according to whether they support an explicitly or implicitly parallel programming model. An explicitly parallel system requires that the programmer specify directly the activities of the multiple concurrent "threads of control" that form a parallel computation. In contrast, an implicitly parallel system allows the programmer to provide a higher-level specification of program behavior in which parallelism is not represented directly. It is then the responsibility of the compiler or library to implement this parallelism efficiently and correctly.

Implicitly parallel systems can simplify programming by eliminating the need for the programmer to coordinate the execution of multiple processes. For example, in the implicitly parallel, primarily data-parallel language High Performance Fortran, the programmer writes what is essentially sequential Fortran 90 code, augmented with some directives. Race conditions cannot occur, and the HPF program need not be rewritten to take advantage of different parallel architectures.

Explicitly parallel systems provide the programmer with more control over program behavior and hence can often be used to achieve higher performance. For example, an MPI implementation of an adaptive mesh-refinement algorithm may incorporate sophisticated techniques for computing mesh distributions, for structuring communications among subdomains, and for redistributing data when load imbalances occur. These strategies are beyond the capabilities of today's HPF compilers.

A parallel programming style that is becoming increasingly popular is to encapsulate the complexities of parallel algorithm design within libraries (e.g., an adaptive-mesh-refinement library, as just discussed). An application program can then consist of just a sequence of calls to such library functions. In this way, many of the advantages of an implicitly parallel approach can be obtained within an explicitly parallel framework.

Explicitly parallel programming systems can be categorized according to whether they support a shared- or distributed- memory programming model. In a shared-memory model, the programmer's task is to specify the activities of a set of processes that communicate by reading and writing shared memory. In a distributed-memory model, processes have only local memory and must use some other mechanism (e.g., message passing or remote procedure call) to exchange information. Shared-memory models have the significant advantage that the programmer need not be concerned with data-distribution issues. On the other hand, high performance implementations may be difficult on computers that lack hardware support for shared memory, and race conditions tend to arise more easily.

Distributed-memory models have the advantage that programmers have explicit control over data distribution and communication; this control facilitates high-performance programming on large distributed-memory parallel computers.

5.1.1. MPI. The Message-Passing Interface (MPI) is a specification for a set of functions for managing the movement of data among sets of communicating processes. Official MPI bindings are defined for C, Fortran, and C++; bindings for various other languages have been produced as well. MPI defines functions for point-to-point communication between two processes, for collective operations among processes, for parallel I/O, and for process management. In addition, MPI's support for communicators facilitates the creation of modular programs and reusable libraries. Communication in MPI specifies the types and layout of data being communicated, allowing MPI implementations to both optimize for noncontiguous data in memory and support clusters of heterogeneous systems.

MPI programs are commonly implemented in terms of a Single Program Multiple Data (SPMD) model, in which all processes execute essentially the same logic. MPI is today the technology of choice for constructing scalable parallel programs, and its ubiquity means that no other technology can beat it for portability. In addition, a significant body of MPI-based libraries has emerged that provide high-performance implementations of commonly used algorithms. Nevertheless, given that programming in an explicit message-passing style can place an additional burden on the developer, other technologies can be useful if our goal is a modestly parallel version of an existing program in which case OpenMP or HPF may be useful.

5.1.2. PVM. Parallel Virtual Machine (PVM) represents another popular instantiation of the message-passing model that was one of the principal forerunners of MPI and the first de facto standard for implementation of portable message-passing programs. Although PVM has been superseded by MPI for tightly coupled multiprocessors, it is still widely used on networks of commodity processors. PVM's principal design goal was portability, even to nonhomogeneous collections of nodes, which was gained by sacrificing optimal performance. MPI, on the other hand, provides high-performance communication. MPI-1 provided only a nonflexible static process model, while MPI-2 adds a scalable dynamic process model.

Central to the design of PVM is the notion of a "virtual machine" — a set of heterogeneous hosts connected by a network that appears logically to the user as a single large parallel computer. PVM API functions provide the ability to (1) join or leave the virtual machine; (2) start new processes by using a number of different selection criteria, including external schedulers and resource managers; (3) kill a process; (4) send a signal to a process; (5) test to check that a process is responding; and (6) notify an arbitrary process if another disconnects from the PVM system.

If an application is going to be developed and executed on a single MPP, then MPI has the advantage of expected higher communication performance. In addition, MPI has a much richer set of communication functions, so it is favored when an application is structured to exploit special communication modes, such as nonblocking send, not available in PVM. PVM has the advantage when the application is going to run over a networked collection of hosts, particularly if the hosts are heterogeneous. PVM includes resource management and process control functions that are important for creating portable applications that run on clusters of workstations and MPPs. PVM is also to be favored when fault tolerance is required. MPI implementations are improving in all of these areas, but PVM still provides better functionality in some settings.

5.1.3. Comparison between PVM and MPI. PVM is one of a number of parallel distributed computing environments (DCEs) [34] that were introduced to assist users wishing to create portable parallel applications [39]. The system has been in use since 1992 [20] and has grown in popularity over the years, leading to a large body of knowledge and a substantial quantity of legacy code accounting for many man-years of development.

Standardization efforts have attempted to address many of the deficiencies of the different DCEs and introduce a single stable system for message passing. These efforts culminated in the first Message Passing Interface (MPI) standard, introduced in June

1994 [4]. Within a year, several different implementations of MPI were available, including both commercial and public systems.

One of MPI's prime goals was to produce a system that would allow manufacturers of high-performance massively parallel processing (MPPs) computers to provide highly optimized and efficient implementations. In contrast, PVM was designed primarily for networks of workstations, with the goal of portability, gained at the sacrifice of optimal performance. PVM has been ported successfully to many MPPs by its developers and by vendors, and several enhancements have been implemented with much success. Nevertheless, PVM's inherent message structure has limited overall performance when compared with that of native communications systems. Thus, PVM has many features required for operation on a distributed system consisting of many (possibly nonhomogeneous) nodes with reliable, but not necessarily optimal, performance. MPI, on the other hand, provides high-performance communication and a nonflexible static process model.

Central to the design of PVM was the notion of a "virtual machine" – a set of heterogeneous hosts connected by a network that appears logically to the user as a single large parallel computer. One aspect of the virtual machine was how parallel tasks exchanged data. In PVM this was accomplished using simple message-passing constructs. There was a strong desire to keep the PVM interface simple to use and understand. Portability was considered much more important than performance for two reasons: communication across the internet was slow; and, the research was focused on problems with scaling, fault tolerance, and heterogeneity of the virtual machine.

The PVM virtual machine is defined by the number and location of the running daemons. Although the number of hosts can be indicated by a fixed list at start-up time, there exists only a single point of failure, the first master daemon to start. All other hosts can join, leave, or fail without affecting the rest of the virtual machine.

PVM API functions allow the user to:

- add or delete hosts,
- check that a host is responding,
- be notified by a user-level message that a host has been deleted (intentionally or not) or has been added, and
- shut down the entire virtual machine, killing attached processes and daemons.

PVM API functions provide the ability to:

- join or leave the virtual machine;
- start new processes by using a number of different selection criteria, including external schedulers and resource managers;
- kill a process;
- send a signal to a process;
- test to check that it is responding; and
- notify an arbitrary process if another disconnects from the PVM system.

If an application is going to be developed and executed on a single MPP, then MPI has the advantage of expected higher communication performance. The application would be portable to other vendor's MPP so it would not need to be tied to a particular vendor. MPI has a much richer set of communication functions so MPI is favored when an application is structured to exploit special communication modes not available in PVM. The most often cited example is the non-blocking send.

Some sacrifices have been made in the MPI specification in order to be able to produce high communication performance. Two of the most notable are the lack of interoperability between any of the MPI implementations, that is, one vendor's MPI cannot send a message to another vendor's MPI. The second is the lack of ability to write fault tolerant applications in MPI. The MPI specification states that the only thing that is guaranteed after an MPI error is the ability to exit the program.

Because PVM is built around the concept of a virtual machine, PVM has the advantage when the application is going to run over a networked collection of hosts, particularly if the hosts are heterogeneous. PVM contains resource management and process control functions that are important for creating portable applications that run on clusters of workstations and MPP.

The larger the cluster of hosts, the more important PVM's fault tolerant features becomes. The ability to write long running PVM applications that can continue even when hosts or tasks fail, or loads change dynamically due to outside influence, is quite important to heterogeneous distributed computing.

Programmers should evaluate the functional requirements and running environment of their application and choose the API that has the features they need.

5.1.4. Linda. Linda[11] is based on an associative shared virtual memory system, or Tuple Space. Rather than sending messages from one process to another, processes created, consumed, duplicated or evaluated data objects known as tuples. Creating a tuple was performed by calling out(), which was then passed into a common shared space which all other processes had access to. Reading a tuple was performed by either rd() or in(), where rd() just read (duplicated) the tuple and in() consumed the tuple and removed it from the shared space. Matching of tuples was performed by specifying a template of possible fields in the tuple when calling either rd() or in(). eval() created tuples asynchronously, with each field making the tuple evaluated in parallel (or at least in a non-deterministic order much like in the Occam ALT construct).

Linda allowed for very simple but powerful parallel computer programs by simplifying addressing (no explicit addressing of processes, only the data they handled) and by removing the coupling between processes. In Linda, one could not specify the consumer of a tuple, and the consumer might not even had existed when the tuple was created.

For example in a conventional message passing system the following steps are required to pass a message from one process to another:

```
proc A                                  proc B find address of 'B'
(addrB)             find address of 'A' (addrA) Send (outdata,
addrB, messagetag)      Recv (indata, addrA, messagetag)
```

Where in Linda one could have:

```
proc A out (messagetag, outdata) exit
```

Some time later:

```
proc B in (messagetag, ?indata) {process data} exit
```

In this case the tuple that contained 'messagetag' as the first field would be consumed by process B (the '?' specifies a wild card) upon which its data would be placed in the indata memory buffer. Neither process ever overlaps temporarily or knows of each others 'address'.

Although initial implementations were slower than native libraries, later versions that utilized compile time analysis of data fields used by application programs allowed the run-time system to select tuned low-level services that implemented the tuple space management and matching operations. On a number of tests[15], some versions of Linda performed comparably to both networked message passing systems such as

PVM as well as native vendor message passing libraries on a number of MPPs for medium to large message payloads.

5.1.5. HPF. High Performance Fortran (HPF) is a de facto standard language for writing data parallel programs for shared and distributed memory parallel architectures. HPF programs can be easier to write than conventional message-passing programs.

HPF is an extension to Fortran 90. It makes use of the fact that the array handling features of Fortran 90 are ideal for expressing parallel array operations. It also provides ways of describing parallel loops. The extensions to Fortran 90 include:

- Compiler directives: to guide the compiler to distribute the data in a given array across the available processors, or to label parallel DO loops
- A FORALL statement and construct to complement the Fortran 90 WHERE - both are parallel constructs in HPF
- New intrinsic and library procedures: to extend the already rich set in Fortran 90
- Subset HPF is a version of High Performance Fortran which omits from Full HPF some Fortran 90 features plus those compiler directives related to dynamic re-distribution of data at runtime. It remains much more powerful than FORTRAN 77.

An HPF program should run effectively on a wide range of parallel machines, including distributed memory machines. On these, programmers have become used to writing "message passing" programs; and resigned to the difficulties of doing this. HPF programs contain no message-passing (though the compiler will inevitably insert some into the final code). An HPF program is as easy to write as a Fortran 90 program; and as easy to read too. So HPF provides a much friendlier programming paradigm for users, who want their results fast, without needing to worry about details of the parallelism.

The HPF approach is based on two fundamental observations. First, the overall performance of a program can be increased if operations are performed concurrently by multiple processors. Second, the efficiency of a single processor is highest if the processor performs computations on data elements stored locally. Based on these observations, the HPF extensions to Fortran 90 provide a means for the explicit expression of parallelism and data mapping. An HPF programmer can express parallelism explicitly, and using this information the compiler may be able to tune data distribution accordingly to control load balancing and minimize communication.

An HPF program has essentially the same structure as a Fortran 90 program, but is enhanced with data distribution and alignment directives. When writing a program in HPF, the programmer specifies computations in a global data space. Array objects are aligned to abstract arrays with no data, called "templates". Templates are distributed according to distribution directives in "block" and "cyclic" fashions. All arrays aligned to such templates are implicitly distributed.

Parallelism can be explicitly expressed in HPF using the following language features:

- Fortran 90 array assignments
- masked array assignments
- WHERE statements and constructs
- HPF FORALL statements and constructs
- HPF INDEPENDENT directives
- intrinsic functions

- the HPF library
- EXTRINSIC functions

The compiler uses these parallel constructs and distribution information to produce code suitable for parallel execution on each node of a parallel computer. The efficiency of an HPF application strongly depends on the quality of the compiler as the programmer has no language means to advise the compiler of many features of the implemented parallel algorithm having significant impact on its performance such as the optimal communication pattern.

5.1.6. mpC. mpC is a language for writing parallel programs for heterogeneous networks of computers. mpC is an extension to the C[] language, which is a Fortran 90 like extension to ANSI C supporting array-based computations. The mpC programmer provides the programming system with comprehensive information about the features of the implemented parallel algorithm that have a major impact on the execution time of this algorithm. In other words, the programmer provides a detailed description of the performance model of this algorithm (see more details in section 4.1). The programming system uses the provided information to optimally map at runtime this algorithm to the computers of the executing network. The quality of this mapping strongly depends on the accuracy of the estimation of the actual performance of the processors and communication links demonstrated at runtime on the execution of this application. Therefore, the mpC programming system employs an advanced performance model of a heterogeneous network of computers, and the mpC language provides constructs that allow the programmer to update the parameters of this model at runtime by tuning them to the code of this particular application. As a result, mpC allows the programmer to write an efficient program for heterogeneous networks in a portable form. The program will be automatically tuned at runtime to each executing network of computers trying to run on the network with maximal possible speed. Such programs might be written in MPI but it is much easier to write them in mpC.

5.2. Distributed Programming Systems.

5.2.1. NetSolve. NetSolve is a project that aims to bring together disparate computational resources connected by computer networks. It is a RPC based client / agent / server system that allows one to remotely access both hardware and software components.

The purpose of NetSolve is to create the middleware necessary to provide a seamless bridge between the simple, standard programming interfaces and desktop Scientific Computing Environments (SCEs) that dominate the work of computational scientists and the rich supply of services supported by the emerging Grid architecture, so that the users of the former can easily access and reap the benefits (shared processing, storage, software, data resources, etc.) of using the latter.

This vision of the broad community of scientists, engineers, research professionals and students, working with the powerful and flexible tool set provided by their familiar desktop SCEs, and yet able to easily draw on the vast, shared resources of the Grid for unique or exceptional resource needs, or to collaborate intensively with colleagues in other organizations and locations, is the vision that NetSolve will be designed to realize.

NetSolve uses a client-server system that enables users to solve complex scientific problems remotely. The system allows users to access both hardware and software computational resources distributed across a network. NetSolve searches for computational resources on a network, chooses the best one available, and using retry for fault-tolerance solves a problem, and returns the answers to the user. A load-balancing policy is used by the NetSolve system to ensure good performance by enabling the system to use the computational resources available as efficiently as possible. Our framework is based on the premise that distributed computations involve resources, processes, data, and users, and that secure yet flexible mechanisms for cooperation and communication between these entities is the key to metacomputing infrastructures.

Some goals of the NetSolve project are: ease-of-use for the user efficient use of the resources, and the ability to integrate any arbitrary software component as a resource into the NetSolve system.

Interfaces in Fortran, C, Matlab, Mathematica, and Octave have been designed and implemented which enable users to access and use NetSolve more easily. An agent based design has been implemented to ensure efficient use of system resources.

One of the key characteristics of any software system is versatility. In order to ensure the success of NetSolve, the system has been designed to incorporate any piece of software with relative ease. There are no restrictions on the type of software that can be integrated into the system.

5.2.2. Nimrod. Parametric computational experiments are becoming increasingly important in science and engineering as a means of exploring the behavior of complex systems. For example, an engineer may explore the behavior of a wing by running a computational model of the airfoil multiple times while varying key parameters such as angle of attack, air speed, etc. The results of these multiple experiments yield a picture of how the wing behaves in different parts of parametric space.

Nimrod is a tool that manages the execution of parametric studies across distributed computers. It takes responsibility for the overall management of an experiment, as well as the low-level issues of distributing files to remote systems, performing the remote computation and gathering the results. EnFuzion is a commercial version of the research system Nimrod. When a user describes an experiment to Nimrod, they develop a declarative plan file which describes the parameters, their default values, and the commands necessary for performing the work. The system then uses this information to transport the necessary files and schedule the work on the first available machine.

Nimrod/G is a Grid aware version of Nimrod. It takes advantage of features supported in the Globus toolkit such as automatic discovery of allowed resources. Furthermore, we introduce the concept of computational economy as part of the Nimrod/G scheduler. The architecture is extensible enough to use any other grid-middleware services such as Legion, Condor and NetSolve.

5.2.3. Java. Java is a high-level object-oriented general purpose programming language with a number of features that make the language quite popular for distributed computing on global heterogeneous networks. Java source code is compiled into bytecode, which can then be executed by a Java interpreter. Compiled Java code can run on most computers because Java interpreters and runtime environments, known as Java Virtual Machines, exist for most operating systems. The definition of Java makes the compiled Java code be interpreted identically on any computer and unable to penetrate the operating environment. Therefore Java code are highly portable and safe, and hence well suited for code sharing. It is easy and safe to use in your distributed application remote Java code. Strong support for object-oriented, safe and portable programming makes Java a unique software tool.

5.2.4. GridRPC. Although Grid computing is regarded as a viable next-generation computing infrastructure, its widespread adoption is still hindered by several factors, one of which is the question "how do we program on the Grid (in an easy manner)". Currently, the most popular middleware infrastructure, the Globus toolkit, by and large provides the basic, low-level services, such as security/authentication, job launching, directory service, etc. Although such services are an absolute necessity especially provided as a common platform and abstractions across different machines in the Grid for interoperability purposes (as such it could be said that Globus is a GridOS), there still tends to exist a large gap between the Globus services and the programming-level abstractions we are commonly used to. This is synonymous to the early days of parallel programming, where the programming tools and abstractions available to the programmers were low-level libraries such as (low-level) message passing and/or thread libraries. In a metaphoric sense, programming directly on top of only Globus I/O can be regarded as performing parallel programming using only the Linux API on a Beowulf cluster.

By all means there have been various attempts to provide a programming model and a corresponding system or a language appropriate for the Grid. Many such efforts have been collected and catalogued by the Advanced Programming Models Research Group of the Global Grid Forum [30]. One particular programming model that has proven to be viable is an RPC mechanism tailored for the Grid, or "GridRPC". Although at a very high level view the programming model provided by GridRPC is that of standard RPC plus asynchronous coarse-grained parallel tasking, in practice there are a variety of features that will largely hide the dynamicity, insecurity, and instability of the Grid from the programmers. These are namely:

- Ability to cope with medium to coarse-grained calls, with call durations ranging from > 1 second to < 1 week.
- Various styles of asynchronous, task-parallel programming on the Grid, with thousands of scalable concurrent calls.
- "Dynamic" RPC, e.g., dynamic resource discovery and scheduling.
- Scientific datatypes and IDL, e.g., large matrices and files as arguments, call-by-reference and shared-memory matrix arguments with sections/strides as part of a "Scientific IDL".
- Grid-level dependability and security, e.g., grid security with GSI, automated fault tolerance with checkpoint/rollback and/or retries.
- Simple client-side programming and management, i.e., no client-side IDL management and very little state left on the client.
- Server-side-only management of IDLs, RPC stubs, "gridified" executables, job monitoring, control, etc.
- Very (bandwidth) efficient—does not send entire matrix when strides and array-sections are specified.

As such, GridRPC allows not only enabling individual applications to be distributed, but also can serve as the basis for even higher-level software substrates such as distributed, scientific components on the Grid. Moreover, recent work [35] has shown that GridRPC could be effectively built upon future Grid software based on Web Services such as OGSA [19]. Some representative GridRPC systems are NetSolve [12], and Ninf [32]. Historically, both projects started about the same time, and in fact both systems facilitate similar sets of features as described above. On the other hand, because of differences in the protocols and the APIs as well as their functionalities, interoperability between the two systems has been poor at best. There had been crude attempts at achieving interoperability between the two systems using protocol translation via proxy-like adapters [32], but for various technical reasons full support of mutual features proved to be difficult.

This experience motivated the need for a more unified effort by both parties to understand the requirements of the GridRPC API, protocols, and features, and come to a common ground for potential standardization. In fact, as the Grid became widespread, the need for a unified standard GridRPC became quite apparent, in the same manner as MPI standardization, based on past experiences with different message passing systems, catapulted the adoption of portable parallel programming on large-scale MPPs and clusters.

6. Applications. Heterogeneous parallel and distributed system have transformed a number of science and engineering disciplines, including cosmology, environmental modelling, condensed matter physics, protein folding, quantum chromodynamics, device and semiconductor simulation, seismology, and turbulence. As an example, consider cosmology — the study of the universe, its evolution and structure — where one of the most striking paradigm shifts has occurred. A number of new, tremendously detailed observations deep into the universe are available from such instruments as the Hubble Space Telescope and the Digital Sky Survey [41]. However, until recently, it has been difficult, except in relatively simple circumstances, to tease from mathematical theories of the early universe enough information to allow comparison with observations.

However, parallel and distributed systems have changed all of that. Now, cosmologists can simulate the principal physical processes at work in the early universe over space-time volumes sufficiently large to determine the large-scale structures predicted by the models. With such tools, some theories can be discarded as being incompatible with the observations. Parallel and distributed systems have allowed comparison of theory with observation and thus have transformed the practice of cosmology.

Another example is the DOE's Accelerated Strategic Computing Initiative (ASCI) which applies advanced capabilities in scientific and engineering computing to one of the most complex challenges in the nuclear era-maintaining the performance, safety, and reliability of the Nation's nuclear weapons without physical testing. A critical component of the agency's Stockpile Stewardship Program (SSP), ASCI research develops computational and simulation technologies to help scientists understand aging weapons, predict when components will have to be replaced, and evaluate the implications of changes in materials and fabrication processes for the design life of aging weapons systems. The ASCI program was established in 1996 in response to the Administration's commitment to pursue a comprehensive ban on nuclear weapons testing. ASCI researchers are developing high end computing capabilities far above the current level of performance, and advanced simulation applications that can reduce the current reliance on empirical judgments by achieving higher resolution, higher fidelity, 3-D physics and full-system modelling capabilities for assessing the state of nuclear weapons.

Parallelism is a primary method for accelerating the total power of a supercomputer. That is, in addition to continuing to develop the performance of a technology, multiple copies are deployed which provide some of the advantages of an improvement in raw performance, but not all.

Employing parallelism to solve large-scale problems is not without its price. The complexity of building parallel supercomputers with thousands of processors to solve real-world problems requires a hierarchical approach - associating memory closely with Central Processing Units (CPUs). Consequently, the central problem faced by parallel applications is managing a complex memory hierarchy, ranging from local registers to far-distant processor memories. It is the communication of data and the coordination of processes within this hierarchy that represent the principal hurdles to effective, correct, and wide-spread-acceptance-of parallel computing. Thus today's parallel computing environment has architectural complexity layered upon a multiplicity of processors. Scalability, the ability for hardware and software to maintain reasonable efficiency as the number of processors is increased, is the key metric.

The future will be more complex yet. Distinct computer systems will be networked together into the most powerful systems on the planet. The pieces of this composite whole will be distinct in hardware (e.g. CPUs), software (e.g. Operating System), and operational policy (e.g. security). This future is most apparent when we consider geographically distributed computing on the Computational Grid [18]. There is great emerging interest in using the global information infrastructure as a computing platform. By drawing on the drawing on the power of high performance computing resources geographically distributed it will be possible to solve problems that cannot currently be attacked by any single computing system, parallel or otherwise.

Computational physics applications have been the primary drivers in the development of parallel computing over the last twenty years. This set of problems has a number of features in common, despite the substantial specific differences in problem domain.

1. Applications were often defined by a set of partial differential equations (PDEs) on some domain in space and time.
2. Multiphysics often took the form of distinct physical domains with different processes dominant in each.
3. The life cycle of many applications was essentially contained within the computer room, building or campus.

These characteristics focused attention on discretizations of PDEs, the corresponding notion of resolution = accuracy, and solution of the linear and nonlinear equations generated by these discretizations. Data parallelism and domain decomposition provided an effective programming model and a ready source of parallelism. Multiphysics, for the most part, was also amenable to domain decomposition and could be accomplished by understanding and trading information about the fluxes between the physical domains. Finally, attention was focused on the parallel computer, its speed and accuracy, and relatively little attention was paid to I/O beyond the confines of the computer room.

The Holy Grail for software is *portable performance*. That is, software should be reusable across different platforms and provide significant performance, say, relative to peak speed, for the end user. Often, these two goals seem to be in opposition each to the other. Languages (e.g. Fortran, C) and libraries (e.g. Message Passing Interface (MPI) [36], Linear Algebra Libraries i.e., LAPACK [3]) allow the programmer to access or expose parallelism in a variety of standard ways. By employing standards-based, optimized libraries, the programmer can sometimes achieve both portability and high-performance. Tools (e.g. svPablo [14], Performance Application Programmers Interface (PAPI) [10]) allow the programmer to determine the correctness and performance of their code and, if falling short in some ways, suggest various remedies.

7. Trends in "Computing". The scientific community has long used the Internet for communication of email, software, and papers. Until recently there has been little use of the network for actual computations. This situation is changing rapidly and will have an enormous impact on the future.

Within the last few years many who work on the development of parallel methods have come to realize the need to get directly involved in the software development process. Issues such as robustness, ease of use, and portability are standard fare in any discussion of numerical algorithm design and implementation. The portability issue, in particular, can be very challenging. As new and exotic architectures evolve they will embrace the notions of concurrent processing, shared memory, pipelining, etc. in order to increase performance. The portability issue becomes formidable indeed as different architectural designs become reality. In fact, it is very tempting to assume that an unavoidable byproduct of portability must be an unacceptable degradation in the level of efficiency on a given variety of machine architecture. We contend that this assumption is erroneous and that its widespread adoption could seriously hamper the ability to effectively utilize machines of the future.

Architectures of future systems promise to offer a profusion of computing environments. The existing forerunners have already given many software developers cause to reexamine the underlying algorithms for efficiency sake. However, it seems to be an unnecessary waste of effort to recast these algorithms with only one computer in mind, regardless of how fast that one may be. The efficiency of an algorithm should not be discussed in terms of its realization as a computer program. Even within a single architecture class, the features of one system may improve the performance of a given program while features of another system may have just the opposite effect.

Software developers should begin to identify classes of problems suitable for parallel implementation and to develop efficient algorithms for each of these areas. With such a wide variety of computer systems and architectures in use or proposed, the challenge for people designing algorithms is to develop algorithms and ultimately software that is both efficient and portable.

REFERENCES

[1] S. Agrawal, D. Arnold, S. Blackford, J. Dongarra, M. Miller, K. Sagi, Z. Shi, K.Seymour, and S. Vahdiyar. Users' guide to netsolve v1.4.1. Technical Report ICL-UT-02-05, Innovative Computing Laboratory, University of Tennessee, June 2002.

[2] S. Agrawal, J. Dongarra, K. Seymour, and S. Vadhiyar. NetSolve: Past, present, and future — a look at a Grid enabled server. In F. Berman, G. Fox, and A. Hey, editors, Grid Computing: Making the Global Infrastructure a Reality, pages 613–622. Wiley, New York, NY, USA, 2003.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, S. Blackford, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorenson. LAPACK Users' Guide. SIAM Publication, Philadelphia, PA, USA, third edition, 1999.

[4] Anonymous. MPI: A message-passing interface standard. International Journal of Supercomputer Applications, 8(3/4):159–416, Fall/Winter 1994.

[5] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert. A proposal for a heterogeneous cluster scalapack (dense linear solvers). IEEE Transactions on Computers, 50(10):1052–1070, October 2001.

[6] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Heterogeneous matrix-matrix multiplication or partitioning a square into rectangles: Np-completeness and approximation algorithms. In Proceedings of the 9th Euromicro Workshop on Parallel and Distributed Processing, pages 298–302. IEEE Computer Society Press, 2001.

[7] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix multiplication on heterogeneous platforms. IEEE Transactions on Parallel and Distributed Systems, 12(10):1033–1051, October 2001.

[8] M. Beck, J. Dongarra, G. Fagg, A. Geist, Gray, Kohl, Migliardi, K. Moore, T. Moore, P. Papadopoulous, S. Scott, and V. Sunderam. Harness: a next generation distributed virtual machine. Journal of Future Generation Computer Systems, 15(5–6):571–582, 1999.

[9] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The grads project: Software support for high-level grid application development. International Journal of High Performance Computing Applications, 15(4):327–344, 2001.

[10] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. International Journal of High Performance Computing Applications, 14(3):189–204, 2000.

[11] N.J. Carriero, D. Gelernter, T.G. Mattson, and A.H. Sherman. The linda alternative to message-passing systems. Parallel Computing, 20(4):633–655, 1994.

[12] H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. In Proceedings of Super Computing '96, 1996.

[13] P. Crandall and M. Quinn. Problem decomposition for non-uniformity and processor heterogeneity. Journal of the Brazilian Computer Society, 2(1):13–23, July 1995.

[14] L. DeRose and D. Reed. Svpablo: A multi-language architecture-independent performance analysis system. In Proceedings of the International Conference on Parallel Processing (ICPP '99), Fukushima, Japan, September 1999.

[15] A. Deshpande and M.H. Schultz. Efficient parallel programming wiht linda. In Proceedings of the Supercomputing 1992, pages 238–244, Piscataway, NJ, 1992. IEEE Press.

[16] E. Dovolnov, A. Kalinov, and S. Klimov. Natural block data decomposition for heterogeneous clusters. In Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), Nice, France, 2003. IEEE Computer Society Press.

[17] G. Fagg and J. Dongarra. Building and using a fault tolerant mpi implementation. International Journal of High Performance Computing Applications, 18(3):353–362, 2004.

[18] I. Foster and C. Kesselman, editors. Computational Grids: Blueprint for a New Computing Infrastructure. Morgan Kaufman, 1996.

[19] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. http://www.globus.org/ogsa, January 2002.

[20] 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, Cambridge, Massachusetts, 1994.

[21] W. Gropp, E. Lusk, and R. Thakur. Using MPI-2: Advanced Features of the Message Passing Interface. MIT Press, Cambridge, MA, first edition, February 2000.

[22] M. Kaddoura, S. Ranka, and A. Wang. Array decompositions for nonuniform computational environments. Journal of Parallel and Distributed Computing, 36(2):91–105, 1 August

1996.

[23] A. Kalinov. Scalability analysis of matrix-matrix multiplication on heterogeneous clusters. In Proceedings of ISPDC '2004 / HeteroPar '04, Cork, Ireland, 2004. IEEE Computer Society Press.

[24] A. Kalinov and A. Lastovetsky. Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers. Lecture Notes in Computer Science, 1593:191–200, 1999.

[25] A. Kalinov and A. Lastovetsky. Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers. Journal of Parallel and Distributed Computing, 61(4):520–535, April 2001.

[26] A. Lastovetsky. Parallel Computing on Heterogeneous Networks. Wiley-Interscience, first edition, June 2003.

[27] A. Lastovetsky and R. Reddy. Data partitioning with a realistic performance model of networks of heterogeneous computers. In Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '2004), Santa Fe, New Mexico, April 2004. IEEE Computer Society Press.

[28] A. Lastovetsky and R. Reddy. On performance analysis of heterogeneous parallel algorithms. Parallel Computing, 30, 2004.

[29] A. Lastovetsky and J. Twamley. Towards a realistic performance model for networks of heterogeneous computers. In The 2004 IFIP International Symposium on High Performance Computational Science and Engineering (HPCSE -04), Toulouse, France, August 2004.

[30] C. Lee, S. Matsuoka, D. Talia, A. Sussman, M. Mueller, G. Allen, and J. Saltz. A Grid Programming Primer. http://www.gridforum.org/7_APM/APS.htm, submitted to the Global Grid Forum, August 2001.

[31] M. Litzkow, M. Livny, and M. Mutka. Condor: A hunter of idle workstations. In Proceedings of the 8th Int'l Conf. Distributed Computing Systems, pages 104–111, Piscataway, NJ, 1988. IEEE Press.

[32] H. Nakada, M. Sato, and S. Sekiguchi. Design and Implementations of Ninf: towards a Global Computing Infrastructure. In Future Generation Computing Systems, Metacomputing Issue, volume 15, Number 5-6, pages 649–658, 1999.

[33] Y. Ohtaki, D. Takahashi, T. Boku, and M. Sato. Parallel implementation of strassen's matrix multiplication algorithm for heterogeneous clusters. In Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '2004), Santa Fe, New Mexico, April 2004. IEEE Computer Society Press.

[34] W. Rosenberry, D. Kenney, and G. Fisher. Understanding DCE. O'Reilly, Sebastopol, CA, 1992.

[35] S. Shirasuna, H. Nakada, S. Matsuoka, and S. Sekiguchi. Evaluating Web Services Based Implementations of GridRPC. In Proc. of HPDC11 (to appear), 2002.

[36] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. MPI: The Complete Reference. MIT Press, Boston, MA, 1996.

[37] M. Snir, S. W. Otto, S. Huss-Lederman, D.W. Walker, and J. Dongarra. MPI: The Complete Reference, Volume 1: The MPI Core. MIT Press, Cambridge, MA, second edition, 1998.

[38] X.-H. Sun. Scalability versus execution time in scalable systems. Journal of Parallel and Distributed Computing, 62(2):173–192, February 2002.

[39] L. Turcotte. A survey of software environments for exploiting networked computing resources. Technical Report MSSU-EIRS-ERC-93-2, Enginerring Research Center, Mississippi State University, February 1993.

[40] R. Wolski. Forecasting network performance to support dynamic scheduling using the network weather service. In Proc. 6th IEEE Int'l Symp. High Performance Distributed Computing, pages 316–325, Los Alamitos, CA, 1997. IEEE Computer Soc. Press.

[41] D. York and et. al. The sloan digital sky survey: Technical summary. The Astronomical Journal, 120:1579–1587, September 2000.