Programming and Computer Software, Vol. 26, No. 4, 2000, pp. 216–236.

A Language and Programming Environment for High-Performance Parallel Computing on Heterogeneous Networks

A. L. Lastovetsky, A. Ya. Kalinov, I. N. Ledovskikh, D. M. Arapov, and M. A. Posypkin

Institute of System Programming, ul. Bol'shaya Kommunisticheskaya 25, Moscow, 109004 Russia

Abstract—An mpC language designed specifically for programming high-performance computations on heterogeneous networks is described. An mpC program explicitly defines an abstract computing network and distributes data, computations, and communications over it. At runtime, the mpC programming environment uses this information and that about the actual network to distribute the processes over the actual network so as to execute the program in the most efficient way. Experience in using mpC for solving problems on local networks consisting of heterogeneous workstations is discussed.

1. INTRODUCTION

Several years ago, only the so-called supercomputers were classified as high-performance parallel computer systems (they were divided into symmetrical multiprocessors (SMP) and massively parallel processors (MPP)). Parallel computing on networks consisting of workstations and personal computers were useless, since they could not speed up the process of solving most problems that were due to the low performance of standard network hardware. However, beginning with the 1990s, the rate of performance growth of the network hardware became greater than that of processors (see 1, pp. 6–7]. Modern network technologies, such as Fast Ethernet, ATM, Myrinet, and others allow data exchange between computers at the rate of hundreds of megabits, or even several gigabits per second. In this situation, not only parallel computers, but also conventional local and even wide area networks can be used as systems for high-performance parallel computing. The strategic initiative announced by the US president and supported by leading telecommunications and computer companies aimed to increase the rate of data exchange on the Internet a thousand-fold marks a tendency to networked high-performance parallel computing.

Thus, computer networks are presently the most available and widespread parallel architecture; often, they make it possible to speed up the process of solving certain problems without having to purchase a more powerful computer, but through the use of available computers connected in a network by modern hardware. Networked parallel computing is retarded only by the absence of the proper software. The point is that, in contrast to supercomputers, networks are inherently heterogeneous: they include various computers with different performance levels, and the network hardware is often diversified as well. Thus, the rate of data exchange between different processors is not the same. As a rule, a program written for a (homogeneous) supercomputer is executed on a heterogeneous network with the same speed as it would be on a homogeneous network consisting of processors that are equivalent (in terms of performance) to the slowest processor of the heterogeneous network and the number of which is the same as the number of processors in the heterogeneous network. This is due to the fact that parallel programs distribute data, computations, and communications over the network without regard for differences in the performance of processors and communication links. As a result, networks are rarely used for high-performance parallel computing.

At the present time, the most widespread tools for parallel programming for networks are MPI (Message Passing Interface) [2], PVM (Parallel Virtual Machine) [3], and HPF (High Performance Fortran) [4].

PVM and MPI are libraries designed for passing messages; in essence, they provide tools for low-level (Assembler) parallel programming. For this reason, developing real (useful and complex rather than model) programs is very difficult and requires highly qualified programmers. In addition, these libraries were not designed for developing adaptable parallel programs (i.e., programs that can distribute computations and communications depending on input data and specific features of a particular heterogeneous network). Certainly, due to the low level of these libraries, it is possible to develop a specialized runtime system that can make a program adaptable; however, such a system is often too complex and its development is beyond the capabilities of most users.

HPF (High Performance Fortran) is a parallel high-level language designed for programming (homoge-

0361-7688/00/2604-0216$25.00 © 2000 MAIK "Nauka/Interperiodica"

neous) supercomputers. The only parallel architecture available in HPF is a homogeneous multiprocessor with very fast communication links between its processors. HPF supports neither irregular and inhomogeneous data distribution nor middle-sized block parallelism. A typical compiler translates an HPF program into PVM or MPI, and the programmer cannot balance processes in the target message-passing program. Besides, HPF is very difficult to compile. Even the best HPF compilers produce a code that is 2 or 3 times slower than the manually written MPI program when executed on homogeneous clusters of workstations (see proceedings of the conference of HPF users [5], held in June 1998 in Porto, Portugal). For these reasons, HPF is not very well suited for programming networked parallel computing.

Thus, new specially designed tools are required to efficiently use available heterogeneous networks as parallel distributed memory computer systems.

In this paper, we describe the first language (and the corresponding programming environment) specifically designed for programming heterogeneous networks, which we called mpC. It is an extension of ANSI C. Similarly to HPF, it includes a vector subset [6]. The available languages for parallel programming are designed for programming regular parallel architectures. These architectures can be described by a small number of parameters. Due to regularity, the target architecture can be implicitly built in these languages. This conventional approach is not suitable for designing a parallel language for developing programs to be executed on heterogeneous networks, since this architecture has no regular structure. The basic idea of mpC is to provide language constructs that make it possible for the user to define an abstract heterogeneous parallel machine that is best suited for executing a specific algorithm. This information, together with information about the actual parallel system, is used by the mpC programming environment to efficiently execute the corresponding program on the particular parallel system.

The paper is organized as follows. Section 2 contains an introduction to mpC. Section 3 gives a brief description of implementation principles. In Section 4, problems of evaluating characteristics of the parallel computer system on which the program is to be executed are discussed. Section 5 presents the experience in developing real applications in mpC. Section 6 describes related works, and Section 7 contains conclusions.

2. INTRODUCTION TO MPC

The mpC language includes a notion of computing space that is defined as a set of available virtual processors of various performance characteristics; the processors are connected by communication links of various transmission rates.

The concept of a network object or just a network is fundamental for mpC. The network comprises virtual processors of various performances; the processors are connected by communication links of various transmission rates. The network is a domain in the computing space that can be used for evaluating expressions and executing various statements and instructions.

Allocating and de-allocating network objects in the computing space is done similarly to allocating and de-allocating data objects in memory in the C language. From the conceptual point of view, the creation of a new network is initiated by a processor of the existing network. This processor is called the parent of the network to be created. The parent always belongs to the created network. The only processor that is defined from the beginning to the end of the program execution is a predefined virtual host processor.

Every network object declared in the program belongs to a certain type. The type specifies the number, types, and performance characteristics of the processors, communication links between them, transmission rates over those links, and the network parent. For example, the following declaration declares the network type *Rectangle*, describing networks consisting of four virtual processors of different performances connected in a rectangle by undirected links of the standard transmission rate.

```
/* Line 1 */ nettype Rectangle {
/* Line 2*/ coord I=4;
/* Line 3*/ node { I>=0 : I+1; };
/* Line 4*/ link {
/* Line 5*/ I>0: [I]<->[I-1];
/* Line 6*/ I==0: [I]<->[3];
/* Line 7*/ };
/* Line 8*/ parent [0];
/* Line 9*/ };
```

Here line 1 contains the header (name) of the network type declaration.

Line 2 contains the declaration of the reference frame for processors. It introduces the integer coordinate variable I that can take values in the range from 0 to 3.

Line 3 declares the processor nodes. It defines the location of the processors in the reference frame defined and declares their types and performances. Line 3 corresponds to the predicate "for all $I < 4$, if $I \geq 0$, then the virtual processor located at the node $[I]$ has the performance $I + 1$." The expression $I + 1$ is called the performance specifier. Greater numbers correspond to greater performances. In our example, the virtual processor 0 is two times as slow as processor 1, three times as slow as processor 2, and four times as slow as processor 3. For any network of this type, the information given in the declaration makes it possible to assign to each virtual processor a weight normalized relative to the parent node.

Lines 4–7 contain the declaration of links between the processors. Line 5 corresponds to the predicate "for all $I < 4$, if $I \geq 0$, then there exists an undirected link of the normal transmission rate between the processors located at the points [I] and [$I - 1$]." Line 6 corresponds to the predicate "for all $I < 4$, if $I == 0$, then there exists an undirected link of the normal transmission rate between the processors located at the points [I] and [3]." Note that if no link between two processors is explicitly specified, then it is assumed that there exists a link between them with a minimal (for this network) transmission rate.

Line 8 contains the declaration of the parent; it specifies that the parent processor be located at the point [0] of the network created.

Having declared a network type, we may declare an identifier of the corresponding type. For example, the declaration

```
net Rectangle r1;
```

declares *r*1 as the network object of type *Rectangle*.

The concept of a distributed object is defined in the style of the languages C* [7] and Dataparallel C [8]. By definition, a data object distributed over a domain of the computing space consists of conventional (not distributed) objects of the same type (called the components of the distributed data object) located at the processor nodes of this domain so that every processor node stores exactly one component. For example, the declarations

```
net Rectangle r2;
int [*]Derror, [r2]Da[10];
float [host]f, [r2:I<2]Df;
repl [*]di;
```

declare the following variables:

- the integer variable *Derror* distributed over the entire computing space;
- the array *Da* of ten integers distributed over the network *r*2;
- the nondistributed floating point variable *f* stored on the node of the virtual host processor;
- the floating point variable *Df* distributed over a subnetwork of the network *r*2;
- the integer variable *di* replicated over the entire computing space. A distributed object is called replicated if its components are equal to each other.

The concept of the distributed value is defined similarly to that of the distributed data object.

In addition to a simple network type, it is possible to declare a parameterized family of network types, called a topology or parameterized network type. For example, the following declaration declares the topology *Ring* describing networks comprising *n* virtual processors connected in a ring by undirected links of the normal transmission rate.

```
/* Line 1 */ nettype Ring (n, p[n]) {
/* Line 2*/ coord I=n;
/* Line 3*/ node {
/* Line 4*/ I>=0: p[I];
/* Line 5*/ };
/* Line 6*/ link {
/* Line 7*/ I>0: [I]<->[I-1];
/* Line 8*/ I==0: [I]<->[n-1];
/* Line 9*/ };
/* Line 10*/ parent [0];
/* Line 11*/ };
```

Line 1 contains the header of the declaration: it declares the integer parameter *n* and the vector parameter *p* consisting of *n* integers. The coordinate variable *I* runs through values from 0 to $n - 1$. Line 4 corresponds to the predicate "for all $I < n$, if $I >= 0$, then the virtual processor with the relative performance $p[I]$ is located at the point [I]" and so on.

Having declared a topology, we can declare an identifier of the network object of this type. For example, the fragment

```
repl [*]m, [*]n[100];
/* Computation m, n[0], ..., n[m-1] */
net Ring(m,n) rr;
```

declares the identifier *rr* of the network object; the type of this object is completely defined only at runtime. The network *rr* consists of *m* virtual processors; the relative performance of the *i*th processor is defined by the value of $n[i]$.

Every network object is characterized by the class of the computing space that is allocated to it; this class determines the lifetime of the object. The computing space can be allocated statically or dynamically. The computing space for a static network is allocated only once. Being created, the network exists up to the termination of the program. If a network is declared as automatic, a new instance is created every time the program execution reaches the block in which the network is declared and destroyed when this block is exited.

Consider a simple mpC program that calculates the product of two dense square matrices *X* and *Y*; the program uses several virtual processors, each of which calculates a part of the rows of the resulting matrix *Z*.

```
/* 1 */ #include <stdio.h>
/* 2*/ #include <stdlib.h>
/* 3*/ #include <mpc.h>
/* 4*/ #define N 1000
/* 5*/ void [host]Input(), [host]Output();
/* 6*/ nettype Star(m, n[m]) {
```

```
/* 7*/ coord I=m;
/* 8*/ node { I>=0: n[I]; };
/* 9*/ link { I>0: [0]<->[I]; };
/* 10*/ };
/* 11*/ void [*]main()
/* 12*/ {
/* 13*/ double [host]x[N][N], [host]y[N][N], [host]z[N][N];
/* 14*/ repl int nprocs;
/* 15*/ repl double *powers;
/* 16*/ Input(x, y);
/* 17*/ MPC_Processors(&nprocs, &powers);
/* 18*/ {
/* 19*/ repl int ns [nprocs];
/* 20*/ MPC_Partition_1b(nprocs, powers, ns, N);
/* 21*/ {
/* 22*/ net Star(nprocs, ns) w;
/* 23*/ int [w]myn;
/* 24*/ myn=([w]ns)[I coordof myn];
/* 25*/ {
/* 26*/ repl int [w]i, [w]j;
/* 27*/ double [w]dx[myn][N], [w]dy[N][N], [w]dz[myn][N];
/* 28*/ dy[]=y[];
/* 29*/ dx[]=::x[];
/* 30*/ for(i=0; i<myn: i++)
/* 31*/ for(j=0; j<N; j++)
/* 32*/ dz[i][j]=[+](dx[i][]*(double[*][N:N])(dy[0]+j)[]);
/* 33*/ z[]::=dz[];
/* 34*/ }
/* 35*/ }
/* 36*/ Output(z);
/* 37*/ }
/* 38*/ }
```

The program includes five functions: *main*, specified in the above fragment; *Input* and *Output*, specified in a different source file; and library functions *MPC_Processors* and *MPC_Partition_1b*. The functions *Input* and *Output* are declared in line 5, and *MPC_Processors* and *MPC_Partition_1b* are declared in the file *mpc.h*.

In general, mpC admits three classes of functions. In our example, functions of all three classes are used: *main* falls into the class basic functions, *Input* and *Output* to the class of network functions, and *MPC_Processors* and *MPC_Partition_1b* to the class of nodal functions.

The call to a basic function is always a total expression (i.e., it is calculated on the entire computing space; no other computations can be performed concurrently with the calculation of a total expression). Its arguments (if any) either belong to the host processor or are distributed over the entire computing space, and the return value (if any) is distributed over the entire computing space. In contrast to functions of other types, basic functions can include declarations of networks. The construct [*] in line 11 placed before the identifier *main* indicates that this is the identifier of a basic function.

A nodal function can be completely executed on a single processor of the computing space. In a nodal function, only local data objects of the virtual processor where this function belongs can be created; in addition, components of external data objects belonging to this processor can be used. The declaration of a nodal function identifier does not require any additional specifications. From the mpC point of view, all normal C functions belong to this class.

Generally, a network function is called and executed on a domain of the computing space, and the arguments and the return value of this function (if any) are also distributed over the same domain. Two network functions can be executed concurrently if the domains in which they are called do not overlap. The functions *Input* and *Output* are examples of the simplest form of network functions that can only be called on a statically defined domain of the computing space. They are declared in line 5 as network functions that can only be called on the virtual host processor (this is indicated by the construct [*host*] that appears before the function

identifiers). Thus, the calls to these functions in lines 16 and 36 are performed on the virtual host processor.

Lines 11 trough 38 contain the definition of the function *main*. In line 13, the arrays *x*, *y*, and *z* located at the virtual host processor are declared.

Line 14 declares the integer variable *nprocs* replicated over the entire computing space. Its distribution is set by default, without using the construct [*]. In general, the default distribution is defined by the distribution of the smallest block that contains the declaration in question and has an explicit distribution specification. The declaration in line 14 appears in the body of the function *main*, for which the distribution over the entire computing space is explicitly specified.

Line 15 declares the variable *powers* as a pointer to a floating point number distributed over the entire computing space. This declaration specifies that all distributed data objects referenced by *powers* are replicated.

In line 17, the library function *MPC_Processors* is called on the entire computing space. This function returns the number of physical processors and their performances (the mechanism used to evaluate the performance of processors is described in Section 4). Thus, after this function is executed, the replicated variable *nprocs* contains the actual number of physical processors, and the replicated array *powers* contains their performances.

Line 19 declares the dynamic integer array *ns* replicated over the entire computing space. All components of this array consist of the same number of elements *nprocs*.

The library nodal function *MPC_Partition_1b* is called on the entire computing space. This function uses the performance characteristics of the physical processors to evaluate the number of rows of the resultant matrix that will be calculated by each of the physical processors. Thus, after the execution of this function, *ns*[*i*] contains the number of rows to be calculated by the *i*th physical processor. *MPC_Partition_1b* divides the given integer (*N* in our example) into parts in the given proportion.

In line 22, an automatic network *w* consisting of *nprocs* virtual processors is declared. The relative performance of the *i*th virtual processor is determined by *ns*[*i*]. Thus, the type of this network is completely defined only at runtime. This network, which performs the remaining computations and data exchange, is defined so as to assign a greater number of rows to be calculated to more powerful virtual processors. The mpC programming environment makes an optimal mapping of the virtual processors constituting the net *w* onto the set of processes that constitute the computing space. Thus, exactly one process of all processes executed by each physical processor will participate in multiplying matrices, and the more powerful the processor, the greater the number of rows that will be calculated by this processor.

Line 23 declares the variable *myn* distributed over *w*.

The result of the binary operation *coordof* in line 24 is an integer value distributed over *w*; every component of this value is the coordinate *I* of the virtual processor where this component resides. The right-hand operand of this operation is not calculated, but is used to specify the domain of the computing space. Note that the coordinate variable *I* is interpreted as an integer variable distributed over the computation domain. Thus, after the execution of the statement in line 24, every component of *myn* contains the number of rows of the resulting matrix that are calculated by the virtual processor where this component resides.

Line 26 declares the integer variables *i* and *j* replicated over the network *w*.

Line 27 declares three arrays distributed over the network *w*. The type *dy* is declared statically as an array of *N* arrays consisting of *N* floating point numbers each. The types *dx* and *dz* are declared dynamically as arrays of *myn* arrays consisting of *N* floating point numbers each. We note that the dimension *myn* of *dx* and *dz* is different for different components of these arrays.

Line 28 includes an unusual unary postfix operator []. The point is that, strictly speaking, mpC is an extension of the vector extension of ANSI C called C[] [14], where the concept of a vector considered as an ordered sequence of a certain type of values was introduced. In contrast to an array, a vector is a new type of value rather than a data object. In particular, the value of an array is a vector. The operator [] was introduced to access the array as a whole. The operand of this operation is of the type "array," and it blocks the transformation of the operand to the pointer type. Thus, *y*[] designates the array *y* as a single object, and *dy*[] designates the distributed array *dy* as a single object.

The statement in line 28 sends the matrix *Y* from the parent of the network *w* to all its virtual processors. As a result, every component of the distributed array referenced by *dy* will contain this matrix. Generally, if the left-hand side operand of the statement = is distributed over a certain domain of the computing space *R*, the value of the right-hand side operator belongs to a domain of the computing space that includes *R* and the assignment can be performed without type casting, then the execution of the statement consists in sending the value of the right-hand side operand to every virtual processor of the domain *R* where it is assigned to the corresponding component of the left-hand side operand.

The statement in line 29 sends the matrix *X* from the virtual host processor to all virtual processors of the network *w*. As a result, every component of *dx* contains the corresponding portion of the matrix *X*.

In the general case, the first operand of the four-place operation =:: must be an array distributed over a certain domain *R* consisting of *NP* virtual processors. The remaining operands (the second and the third operands are optional) are not distributed and belong to the same processor. The second operand is optional; if it is present, then it must either point to the initial element

of an integer array consisting of *NP* elements or to an integer array consisting of *NP* elements. The third operand is optional; if it is present, then it must either point to the pointer to the initial element of an integer array consisting of *NP* elements such that the value of its *i*th element is not greater than the number of elements of the *i*th component of the first operand, or point to an array of this type. The fourth operand must be an array consisting of elements of a type that can be assigned to any elements of any component of the first operand without type casting.

Execution of the operation $e1 = e2{:}e3{:}e4$ consists in cutting from the array $e4$ subarrays consisting of *NP* elements (maybe overlapping) and sending the value of the *i*th subarray to the *i*th virtual processor of the domain *R*, where it is assigned to the corresponding component of the distributed array $e1$. The offset of the *i*th subarray relative to the initial element of $e4$ is specified by the value of the *i*th element of $e2$ and its length is specified by the value of the *i*th element of $e3$.

If $e3$ points to the *NULL* pointer, then the operation is performed as if $*e3$ pointed to the initial element of the *N*-element integer array with the *i*th element equal to the length of the *i*th component of the distributed array $e1$. Moreover, in this case, such an array is actually created as a result of the execution of the operation, and the pointer to its initial element as assigned to $*e3$.

If $e2$ points to the *NULL* pointer, then the operation is performed as if $*e2$ pointed to the initial element of the *N*-element integer array with the *0*th element equal to 0 and the *i*th element equal to the sum of the value of its $(i-1)$th element and the value of the *i*th element of $e3$. Moreover, in this case, such an array is actually created as a result of the operation being executed, and the pointer to its initial element as assigned to $*e2$.

The second operand can be omitted. In this case, the operation is performed as if $e2$ pointed to the *NULL* pointer. The only difference is in that the *NP*-element array created in the process of execution is freed.

The third operand can be omitted. In this case, the operation is performed as if $e3$ pointed to the *NULL* pointer. The only difference is that the *N*-element array created in the process of execution is freed.

Thus, the statement $dx[] =:: x[]$ in line 29 results in dividing the *N*-element array x consisting of arrays of *N* floating point numbers into *nprocs* subarrays such that the length of the *i*th subarray equals the value of the component of *myn* belonging to the *i*th virtual processor (i.e., it is equal to $[w: I == i]myn$). The same operation will be performed more quickly if it is written in the form $dx =: \&ns : x$, which makes it possible to avoid additional (and redundant in this case) data transmission and computations necessary for the formation of the omitted operands.

The asynchronous fragment in lines 30–32 (i.e. the set of statements that do not require data exchange between the virtual processors) concurrently calculates the corresponding portion of the resulting matrix Z on every virtual processor of the net w.

Finally, the statement in line 33 collects these portions on the virtual host processor forming the resultant array z with the help of the assembly statement ::=. This 4-place operation corresponds to the sending operation and performs the inverse communication operations in the similar fashion.

3. IMPLEMENTATION OF MPC

At the present time, the mpC programming environment includes a compiler, a runtime support system, a library, and a command-line user interface.

The compiler translates the source code in mpC into an ANSI C program that includes calls to functions of the runtime support system. Either the SPMD model of the target code is used, in which all processes constituting the parallel program execute the same code, or the quasi-SPMD model in which the source mpC code is translated into two different programs—one for the virtual host processor and the other for all other virtual processors.

The runtime support system manages the computing space, which consists of several processes executing on the target distributed memory computer system (for example, on a network consisting of PCs and workstations), and communications. It encapsulates a particular communication package (currently, MPI 1.1) and ensures that the compiler is independent of the particular target platform.

The library consists of functions that support program debugging, provide access to the characteristics of the computing space and to efficient low-level functions.

The command-line user interface includes commands for creating a virtual machine with distributed memory and executing mpC programs on it. When the virtual parallel machine is created, its topology (in particular, the number and performance of processors and characteristics of the communication links between the processors) is determined automatically by executing a special benchmark program; the information obtained is saved in a file that is used by the runtime support system.

3.1. Model of the Target Program

All processes that constitute the executing target program are divided into two groups: a special process, called dispatcher, that controls the computing space, and normal processes, called nodes, that act as virtual processors of the computing space. The dispatcher functions as the server receiving requests from the nodes. It does not belong to the computing space.

In the target program, every network or subnetwork of the initial mpC program is represented by a set of nodes called the domain. At any moment, when the tar-

get program is executed, every node is either free or included in one or more domains. The node included in a domain is called busy. The dispatcher is responsible for including nodes in domains and freeing them. The only exception is the so-called host node that acts as the virtual host processor. Immediately after the initialization, the computing space is represented by the host and the set of free nodes.

The main difficulty in managing processes is in including them in domains and freeing them. The solution to this problem determines the structure of the target code and forms requirements for the functions of the runtime support system.

To create a domain that represents a network (a network domain), the parent node evaluates (if necessary) topological characteristics of the network and sends to the dispatcher the request for creating the domain. The request includes the number of nodes and their relative performances. In turn, the dispatcher possesses information about the target network of computers that includes the number of physical processors, their relative performances, and the number of nodes on the physical processors. Using this topological information, the dispatcher selects a set of free nodes that are most suitable for the network domain being created (the algorithm used by the dispatcher is described in Subsection 3.3 in more detail). Then, the dispatcher sends a message to all free nodes informing them whether or not they are in the new domain.

To free a network domain, its parent node sends a corresponding request to the dispatcher. Note that the parent node remains in the parent network domain of the domain being freed. All other nodes of the freed domain become free and wait for other commands from the dispatcher.

Every node can determine if it is busy. The node is busy if the function *MPC_Is_busy*() called when it returns 1; it is free if this function returns 0.

Every node can determine whether or not it is busy in a certain domain. A domain can be accessed through its handler. If *rd* is the handler of a domain, then a node of the computing space belongs to this domain if and only if the function *MPC_Is_member*(&*rd*) called at this node returns 1. In this case, the node can obtain information about the domain through its handler *rd* and identify itself within this domain.

When a free node is included in a network domain, the dispatcher must inform it of the handler of the domain it was included in. The simplest method—sending the pointer to the handler from the parent node—is inapplicable for computer systems with distributed memory, which have no common address space. For this reason, we need an additional identifier of the network being created that has the same value on the parent node and free nodes and has a form that makes it possible to send it from the parent node to the free ones via the dispatcher.

In the source mpC program, every network has a name that is a normal identifier and obeys the conventional scope rules of C. Thus, the name of the network cannot be used as its unambiguous identifier even within the same file. We can enumerate all networks within a file and use this number as an unambiguous identifier of the network within this file. However, this identifier will not be unique within the entire program, since the program can include several files. However, this identifier can be used to create the network if all nodes that participate in the creation execute the code corresponding to one and the same source file of the mpC program. This is this scheme that is used by our compiler. All networks in the file are enumerated, and the structure of the program guarantees that when a network is created, all the nodes participating in the process execute the code from the same file.

Thus, when a domain representing a network (a network domain) is created, the parent node, the dispatcher, and all free nodes participate in this process. The parent node calls the function

```
MPC_Net_create(MPC_Name name, MPC_Net* net);
```

where *name* contains the unique identifier (within the file) of the network being created and *net* points to the corresponding handler of the domain. The function calculates all the necessary topological characteristics and sends to the dispatcher a request for creating the network.

Meanwhile, the free nodes wait for the command from the dispatcher at the so-called wait point by calling the function

```
MPC_Offer(MPC_Names* names, MPC_Net** nets_voted, int voted_count);
```

where *names* is the array of the numbers of networks that can be created at this wait point, *nets_voted* is the array of pointers to the handlers of the domains that can be created at this wait point, and *voted_count* contains the number of such domains.

The correspondence between the network number and the handler of the domain is set as follows. If a free node receives a message informing it that it was included in the network with the number *names*[*i*], then it is included in the network domain with the handler *nets_voted*[*i*].

The free nodes leave the wait function *MPC_Offer* either after they have been included in the network

domain or when they receive from the dispatcher the command to leave the wait point.

3.2. The Structure of the Target Code of an mpC Block

In the general case, there are two wait points in the code of the target block containing network declarations. At the first point, called the creating wait point, free nodes wait for dispatcher commands related to the creation of network domains; at the second point, called the freeing point, they wait for commands related to freeing. Between these points, free nodes can execute the code unrelated to creating and freeing networks declared in the source mpC block; namely, they can take part in total computations and/or in creating and freeing the networks declared in nested mpC blocks. The first statement in the source mpC block that requires free nodes for its execution is called the break statement of the wait point. Thus, in the general case, the target block has the following structure.

```
{
  declarations corresponding to the declarations of variables
  in the source mpC block
  {
    if(!MPC_Is_busy()) {
      the target code executed by free nodes
      for creating network domains for the networks
      declared in the source mpC block
    }
    if(MPC_Is_busy()) {
      the target code executed by busy nodes
      for creating network domains for the networks and subnetworks
      declared in the source mpC block;
      the target code for the statements
      that precede the break statement of the wait point
    }
    epilogue of the creating wait point
  }
  the target code for the statements of the source
  mpC block that begin with the break statement of the wait point
  {
    the target code executed by busy nodes for
    freeing the networks and subnetworks declared in the source mpC block

    label of the freeing wait point:

    if(!MPC_Is_busy()) {
      the target code executed by free nodes for
      freeing network domains for the networks
      declared in the source mpC block
    }
    epilogue of the freeing wait point
  }
}
```

If the source mpC block does not include the break statement of the wait point (i.e., if it (and the nested blocks) does not contain total operations or nested blocks with network declarations), then it is possible to merge the creating and freeing wait points into a single wait point in the target block. In this case, the target block has the following structure.

```
{
  declarations corresponding to the declarations of variables
  in the source mpC block
  {
    prologue of the common wait point
```

```
        label of the common wait point:

        if(!MPC_Is_busy()) {
          the target code executed by free nodes for
          creating and freeing network domains for the networks
          declared in the source mpC block
        }
        if(MPC_Is_busy()) {
          the target code executed by busy nodes for
          creating and freeing network domains for the networks and subnetworks
          declared in the source mpC block;
          the target code for the statements of the source mpC block
        }
        epilogue of the common wait point
    }
}
```

To ensure that all participating nodes execute the code from the same file while creating the network domain, the compiler includes a global barrier in the epilogue of the wait point.

The coordinated arrival of the nodes is ensured by the following scenario in the epilogue of the wait point:

- the host makes sure that all other busy nodes that could send a creating or freeing request that is awaited at the wait point have already reached the epilogue;
- the host sends a message to the dispatcher informing it that there are no remaining creating or freeing requests that are awaited at the wait point; then, it reaches the epilogue;
- having received this message, the dispatcher sends the command to all free nodes to leave the wait point;
- having received this command, all free nodes complete the wait function and reach the epilogue.

3.3. The Mapping Algorithm

We have already mentioned that to create a network domain, the dispatcher selects free nodes that are best suited for the domain created. In this section, we describe how the dispatcher performs this selection.

For every network type that occurs in the source mpC program, the compiler generates six topological functions that are used at runtime to evaluate various topological characteristics of the network object of this type. The first function returns the total number of nodes in the network object. Since the runtime support system uses the linear numeration of nodes from 0 to $n - 1$, where n is the total number of nodes, there are two functions (the second and the third ones) that transform the coordinates of the nodes into the linear number and vice versa. The fourth function returns the linear number of the parent node. The fifth function returns the relative performance of the given node. At last, the sixth function returns the length of the directed link connecting the given pair of nodes.

When the dispatcher maps the virtual processors of the network created into the set of free nodes, it first calls the corresponding topological functions to evaluate the relative performance of the virtual processors and the length of the links between them. The relative performance of a virtual processor in the network created is described by a positive real number normalized to the performance of the parent virtual processor (which is assumed to be 1). Then, the dispatcher evaluates the absolute performance of each virtual processor in the network created by multiplying it by the absolute performance of the parent virtual processor.

On the other hand, the dispatcher supports the map of the computing space that represents its topological characteristics. The initial state of this map is formed when the dispatcher is initialized and contains the following information:

- the number of physical processors constituting the computer system with distributed memory and the performance of those processors;
- the number of nodes (processes) of the computing space residing on every physical computation node;
- for every pair of physical nodes, the initialization time and the transmission time of one byte of information;
- for every physical node, the initialization time of message exchange and the transmission time of one byte of information between two processes that are executed on this physical node.

On the whole, the performance of a physical computer node is characterized by two attributes. The first attribute specifies the speed of a process execution on the given physical computer node; the second one specifies the maximal number of noninteracting processes that can be simultaneously executed on the given computer node without loss of speed (i.e., this attribute specifies the scalability of the physical computer node). For example, if a multiprocessor is used as a physical computer node, then the value of the second attribute is the number of processors.

Presently, the procedure of selecting nodes of the computing space to allocate virtual processors of the network created is based on the following simplest scheme.

The dispatcher assigns a weight (a positive real number) to every virtual processor of the network being allocated in such a way that

- the more powerful the virtual processor, the more is its weight;
- the shorter the communication links outgoing the virtual processor, the greater its weight.

The weights are normalized so as to make the weight of the virtual host processor equal to unity.

Similarly, the dispatcher modifies the map of the computing space so as to assign to every physical computer node a weight (a positive real number) that characterizes its computation and communication power when executing a single process.

The dispatcher selects free nodes of the computing space to allocate virtual processors successively, beginning with the virtual processor with the maximum weight. For this virtual processor, the dispatcher assigns the most powerful free node. For this purpose, the dispatcher evaluates the performance of the free node for every physical computer node using the following algorithm. Let w be the weight of the physical computer node P and N be its scalability. Let the set H of busy nodes connected with P be decomposed into N subsets $h_1, \ldots, h_N$, and let v_{ij} be the weight of the virtual processor that is currently located at the jth node of the set h_i. Then, the estimate of the performance of the free node connected with P is evaluated by the formula

$$\max_i \left\{ \frac{w}{v + \sum_j v_{ij}} \right\}.$$

Thus, the dispatcher selects one of the free nodes of the physical computer node that has the maximal performance estimate, places the next virtual processor on it (the most powerful among unallocated ones), and adds this node to the set h_k of the set of busy nodes of the corresponding computer node; the value of k is determined by the relation

$$\frac{w}{v + \sum_j v_{kj}} = \max_i \left\{ \frac{w}{v + \sum_j v_{ij}} \right\}.$$

The procedure described yields sufficiently good results for networks of workstations and personal computers and ensures the optimal allocation if exactly one node of the computing space is connected with every physical processor.

4. EVALUATING PERFORMANCE

As was shown above, mpC gives the programmer an opportunity to define an abstract heterogeneous network that is best suited for executing a particular parallel algorithm; the mpC programming environment maps the abstract network into a physical computer network. The mapping is performed at runtime and is based on the information about the relative performance of the processors and communication links in the physical network. The efficiency of using the potential performance of the available network by a parallel program directly depends on the quality of this mapping, which, in turn, depends on the accuracy of the performance estimates for the processors and network hardware.

Initially, only one method for evaluating the performance of processors and network hardware was supported. The evaluation was performed by executing a special benchmark program and was a part of the execution of a command of the user interface external with respect to mpC.

The principal disadvantage of this method is that it is based on a static integral estimate of the hardware performance. This estimate is independent of the program code and remains the same during its execution. However, the parallel code executed on every network is often considerably different from the mixture of statements in the benchmark program. This difference is not very important when the performance of microprocessors of the same architecture is evaluated; however, for microprocessors with a different architecture, performance estimates obtained with the help of a mixture of statements can differ considerably. As a result, this estimate may become rather inaccurate, which leads to an unbalanced workload of the processors and to a decreased efficiency of the program execution.

We examined four approaches to improving the accuracy of the processor performance evaluation. The first approach is based on a classification of mpC applications and the use of a special benchmark program for every class. This approach was rejected, since experiments showed that a processor performance could be quite different even for applications of the same class. For example, consider two fragments of C programs:

```
for(k=0; k<500; k++) {
  for(i=k, lkk=sqrt(a[k][k]); i<500; i++)
```

```
      a[i][k]/=lkk;
    for(j=k+1; j<500; j++)
      for(i=j; i<500; i++)
        a[i][j]-=a[i][k]*a[j][k];
}
```

and

```
for(k=0; k<500; k++) {
    for(j=k, lkk=sqrt(a[k][k]); j<500; j++)
      a[k][j]/=lkk;
    for(i=k+1; i<500; i++)
      for(j=i; j<500; j++)
        a[i][j]-=a[k][j]*a[k][i]
}
```

Both fragments implement the Cholesky factorization for the matrix of the dimension 500 × 500. If these fragments are used as benchmark programs, then the first one evaluates the relative performance of SPARCstation-5 and SPARCstation-20 as 10 : 9, whereas the second fragments yields the estimate 10 : 14. Note that the function *dpotf2* included in the LAPACK package and solving the same problem estimates the relative performance of these processors as 10 : 10.

The second approach consists in using a special benchmark code for every mpC application. In particular, we considered the problem of the automatic generation of a benchmark code by the source code of the application. This approach considerably complicates the mpC programming environment, but it does not work if the problem solved by the program splits into several subproblems, each of which is solved on a separate network. In this case, the estimate of processor performance is obtained by averaging the estimates obtained for different parallel parts of the program and can be inaccurate as with the first approach.

The third approach consists in the automatic generation, for every mpC program, of a benchmark code that could produce a vector estimate of processor performances. With this approach, every parallel part of the program that is executed on a separate network is characterized by a specific estimate used when this network is created. This approach is very difficult to implement and does not work in the case when the actual network is also used for other computations. In this case, from the standpoint of the mpC program, the actual processor performance is a function of time. Thus, the use of any static estimate that is not modified in the course of the program execution often leads to the inaccurate estimation of processor performance and, consequently, to decreased speed of the program execution.

In our implementation of mpC, we used the fourth approach. We introduced a new language construct that allows the programmer to update the estimate of processor performances during the program execution using the most suitable (from his point of view) benchmark. The new construct is as follows:

```
recon benchmark
```

Here *benchmark* is either an empty statement consisting of a semicolon or a generic type statement distributed over the entire computing space that executes the same piece of code on all virtual processors without specifying the communication between them. This construct makes it possible to update information about the relative performance of the processors in the network. If the *benchmark* is an empty statement, then a standard benchmark is used; otherwise, the routine specified by the *benchmark* is used.

In addition, the library of standard functions includes several functions that make it possible to obtain the current estimate of processor performances and set this estimate explicitly (without executing the benchmark program).

The *recon* statement and the corresponding library functions are not difficult to use and implement. To demonstrate the use of this statement, consider the following subroutine.

```
/*1*/ #define N 100
/*2*/ nettype Grid(P, Q) {
/*3*/ coord I=P, J=Q;
/*4*/ };
/*5*/ int [net Grid(p, q) v] mpC2blacs_gridinit(int*, char*);
/*6*/ void [*]Cholesky(repl int P, repl int Q) {
/*7*/ {
/*8*/ int n = N, info;
/*9*/ double a[N][N];
```

```
/*10*/ init(a);
/*11*/ recon dpotf2_("U", &n, a, &n, &info);
/*12*/ }
/*13*/ {
/*14*/ net Grid(P, Q) w;
/*15*/ [w]: {
/*16*/ int ConTxt;
/*17*/ ([(P, Q)w])mpC2Cblacs_gridinit(&ConTxt, "R");
/*18*/ pdlltdriver1_(&ConTxt);
/*19*/ mpC2Cblacs_gridexit(ConTxt);
/*20*/ }
/*21*/ }
/*22*/ }
```

This subroutine performs the Cholesky factorization of a square matrix. Here, the heterogeneous breakdown of the computation processes between processors is done by mpC and the parallel Cholesky factorization is performed by the means provided by the ScaLAPACK package [10].

The *recon* statement that occurs in line 11 determines the performance of actual processors by executing the program *dpotrf* included in the package LAPACK. The code of this program is a good approximation of the code that will be executed by every node. Thus, the information about the performance of processors is updated at runtime by executing the program *dpotrf2* as a benchmark.

The network w that executes the parallel computations is declared in line 14 (its type is declared in lines 2–4). This network consists of $P \times Q$ virtual processors (by default, they have an identical performance). The parent of this network (the virtual host processor) is located at the point $I = 0$, $J = 0$. At runtime, this declaration of w leads to such a mapping of its virtual processors onto the processes of the parallel program that the number of the processes that participate in the computation at each actual processor is proportional to its performance that was estimated earlier in the program.

A slightly modified test driver of the ScaLAPACK package performing the Cholesky factorization is called on the network w (lines 15–20). This driver reads the parameters of the problem from a file (the size of the matrix and blocks), constructs the test matrix, and factors it.

The mpC language includes three types of functions: basic, network, and nodal functions.

Basic functions are called and executed on the entire computing space. Networks can only be created within those functions. The *Cholesky* function provides an example of a basic function. This is indicated by the construct [*] in line 6 placed directly before the function name.

Network functions are executed on the network (an example is provided by *mpC2Cblacs_gridinit* defined in lin 5). They have three specific parameters: v, p, and q. The so-called network formal parameter v indicates the network on which the function is executed. The parameters p and q are considered as integer variables replicated over the network v. They are parameters of the network type $Grid(p, q)$ of the network v. In line 17, this function is called for the network w and the corresponding actual parameters P and Q.

A nodal function may be executed on any single virtual processor. Conventional C functions are considered as node functions in mpC.

If a variable is declared without a special distribution attribute, then it is considered distributed over the entire computing space in basic functions; in network functions, it is considered distributed over the corresponding network. The distribution attribute [w] in line 15 specifies the network that executes the compound operator in lines 15–20.

5. EXPERIENCE IN USING MPC

The first implementation of mpC was released at the end of 1996. It has been available on the Internet for over two years (http://www.ispras.ru). During this time, over 400 installations have been made all over the world. mpC is mainly used for scientific computations on networks consisting of workstations and PCs. Typical applications include multiplying matrices, solving the multibody problem, linear algebra (LU decomposition, Cholesky factorization, and so on), numerical integration, simulating oil extraction, analyzing constructions for stresses and strains, and many others. The experience shows that mpC allows for developing portable modular parallel programs that considerably speed up the solution of both regular and irregular problems on heterogeneous networks. In addition, mpC makes it possible to solve irregular problems on homogeneous networks much more quickly than by using traditional methods.

In this section, we present several typical applications developed in mpC.

5.1. Irregular Applications Developed in mpC

As an example of an irregular problem, we consider simulating the evolution of a stellar system in a galaxy (or a set of galaxies) under gravitational attraction force.

We assume that the system to be simulated consists of a certain number of large groups of bodies. It is well known that, since the gravitational interaction quickly decreases with distance, the effect of a large group of bodies may be approximated by the effect of a single equivalent body if this group of bodies is located sufficiently far from the point where its effect is analyzed. We suppose that this assumption is true in our case; i.e., let the groups of bodies under consideration be located sufficiently far from each other.

This problem allows for a natural parallelization. Our simulating mpC program will use several virtual processors each of which will update the data concerning one group of bodies. Every virtual processor must store the attributes of all bodies included in the corresponding group and the masses and centers of gravity of all other groups. Every body is described by its coordinates, velocity, and mass.

Finally, let the number of groups and the number of bodies in each group become available only when the program is executed.

The mpC program designed for solving this problem is structured as follows.

```
Initialize the galaxy on the virtual host processor
Create the network for future computations and data exchange
Send the groups of bodies to the virtual processors in the network
Compute (concurrently) the masses of groups
Exchange information about the masses of groups between the virtual processors
while (1) {
  Visualize the galaxy on the virtual host processor
  Compute (concurrently) the masses of gravity for the groups
  Exchange information about the centers of gravity of groups between the virtual
processors
  Update group attributes (concurrently)
  Collect the groups on the virtual host processor
}
```

The corresponding mpC programs is as follows.

```
#define MaxGs 30 /* the maximal number of groups */
#define MaxBs 600 /* the maximal number of bodies in a group */
typedef double Triplet[3];
typedef struct {Triplet pos; Triplet v; double m;} Body;
int [host]M; /* the number of groups */
int [host]N[MaxGs]; / *array of the number of bodies in groups */
repl dM, dn[MaxGs];
double [host]t; /* the galaxy clock */
  /* the array of the body attributes in the galaxy */
Body (*[host]Galaxy[MaxGs][MaxBs];
nettype GalaxyNet(m, n[m]) {
  coord I=m;
  node { I>=0: n[I]*n[I];};
};
void [host]Input(),UpdateGroup(), [host]VisualizeGalaxy();
void [*]Nbody(char *[host]infile)
{
  Input(infile); /* initializing Galaxy, M, and N */
  dM=M; /* sending the number of groups */
  /* sending the array containing the number of bodies in the groups */
  dN[]= N[]; {
    /* creating the network g */
    net GalaxyNet(dM, dN) g;
```

```
    int [g]myN, [g]mycoord;
    Body [g]Group[MaxBs];
    Triplet [g]Centers[MaxGs];
    double [g]Masses[MaxGs];
    repl [g]i;
    void [net GalaxyNet(m, n[m])]Mint(double (*)[MaxGs]);
    void [net GalaxyNet(m, n[m])]Cint(Triplet (*)[MaxGs]);
    mycoord = I coordof body_count;
    myN = dN[mycoord];
    for (i=0; i<[g]dM; i++) /* sending groups */
      [g:I==i]Group[] = (*Galaxy[i])[];
    for (i=0; i<myN; i++)
        Masses[mycoord] += Group[i].m;
    ([([g]dM, [g]dN)g])Mint(Masses);
    while(1) {
      VisualizeGalaxy();
      Centers[mycoord][] = 0.0;
      for(i=0; i<myN; i++)
        Centers[mycoord][] +=
          (Group[i].m/Masses[mycoord])*(Group[i].pos)[];
    ([([g]dM, [g]dN)g])Cint(Centers);
    ([g]UpdateGroup)(Centers, Masses, Group, [g]dM);
    for (i=0; i<[g]dM; i++) /* collecting groups */
      (*Galaxy[i])[] = [g:I==i]Group[];
  }
}
}
void [net GalaxyNet(m, n[m]) p]Mint(double (*Masses)[MaxGs])
{
  double MassOfMyGroup;
  repl i, j;
  MassOfMyGroup = (*Masses)[I coordof i];
  for(i=0; i<m; i++)
    for(j=0; j<m; j++)
      [p:I==i](*Masses)[j] =
        [p:I==j]MassOfMyGroup;
}
void [net GalaxyNet(m, n[m]) p]Cint(Triplet (*Centers)[MaxGs])
{
  Triplet MyCenter;
  repl i, j;
  MyCenter = (*Centers)[I coordof i][];
  for(i=0; i<m; i++)
    for(j=0; j<m; j++)
      [p:I==i](*Centers)[j][] =
        [p:I==j]MyCenter[];
}
```

This program contains the following external declarations:

- declarations of the variables *M*, and *t*, and arrays *N* and *Galaxy* located at the virtual host processor;
- declarations of the variable *dM* and array *dN* replicated over the entire computing space;
- declaration of the network type *GalaxyNet*;
- declaration of the basic function *Nbody* with a single formal parameter *infile* located at the virtual host processor;
- declaration of the network functions *Mint* and *Cint*.

Generally, the network function is called and executed on a certain network or subnetwork; its argu-

ments and the return value (if any) are also distributed over the same network or subnetwork. The header of the declaration of the network function either specifies an identifier (within the scope of the file) of a static network or subnetwork or declares the identifier of the network that is a special formal parameter of the function. In the first case, the function may be called only on the network or subnetwork specified; in the second case, it may be called on any network or subnetwork of the proper type. In any case, no other networks, except for that specified in the header of the declaration, can be created and used in the body of the network function. Only data objects that are located on the network or subnetwork associated with a function can be created within its body. In addition, the function may use components of external data objects that are located within the corresponding domain of the computing space. In contrast to basic functions, network (as well as nodal) functions can be called concurrently.

In the program above, the nodal functions *Input*, *VisualizeGalaxy*, and *UpdateGroup* are specified; the first two of them are associated with the virtual host processor.

The automatic network g, which performs the greater part of the computations and communications, includes M virtual processors with the relative performances characterized by the square of the number of the bodies in the groups that will be analyzed by those processors.

Thus, a more powerful virtual processor will analyze a larger group of bodies. Using this information, the programming environment builds a suitable mapping of the virtual processor included in the network g onto the processors of the computing space. Since this is done at runtime, the program does not need to be recompiled before executing it on another network.

The call of ([g]*UpdateGroup*)(...) initiates concurrent execution of the nodal function *UpdateGroup* on every virtual processor of g. This means that the function name *UpdateGroup* is transformed into the pointer to the function distributed over the entire computing space, and the operator [g] cuts from this pointer a pointer distributed over g. Thus, the value of the expression [g]*UpdateGroup* is a pointer to the function distributed over g, and the expression ([g]*UpdateGroup*)(...) designates a distributed call of the set of undistributed functions.

The network functions *Mint* and *Cint* have three special formal parameters. The parameter p indicates the network on which the function is to be executed. The parameter m is treated as an integer variable replicated over p. The parameter n is treated as a pointer to the first element of the array of read-only integers replicated over p. The actual parameters that correspond to these formal parameters are specified by the construct ([([g]dM, [g]dN)g)] placed on the left of the name of the function when it is called from *Nbody*.

The execution time of the mpC program was compared with the execution time of a similar (thoroughly written) MPI program. The network consisted of three workstations SPARCstation-5 (*gamma*), SPARCclassic (*omega*), and SPARCstation-20 (*alpha*) connected by 10 Mbits Ethernet. In addition to these three, the segment of the local network included about 23 more workstations. LAM MPI version 5.2 [12] was used as the communication platform.

The computing space consisted of 15 processors: 5 on each workstation. The dispatcher was executed on the *gamma* station and used the following relative performances obtained automatically when the virtual machine was created: 1150 (*gamma*), 331 (*omega*), and 1662 (*alpha*).

The program in MPI was designed so as to minimize the communication burden. In all the experiments, nine groups of bodies were used. Three processes of the MPI program were run on *gamma*, one process on *omega*, and five on *alpha*. This mapping is optimal if the number of bodies in all groups is the same.

Two experiments were conducted. In the first one, the performance of the mpC and MPI programs were compared for homogeneous input data when the number of bodies in all groups was approximately the same. In essence, this experiment showed how much we pay for using mpC instead of MPI. It turned out that the execution time of the MPI program was about 95% that of the mpC program. That is, in this case, we only lose about 5% of the performance.

In the second experiment, we compared the same programs for heterogeneous input data. The groups consisted of 10, 10, 10, 100, 100, 100, 600, 600, and 600 bodies. The execution time of the mpC program is independent of the order of these numbers. In all cases, the dispatcher maps

- four processes for the virtual processors of the network g into *gamma*; these processes compute the data for two groups of 10 bodies, one group of 100 bodies, and one group of 600 bodies;
- three processes for the virtual processors of the network g into *omega*; these processes compute the data for one group of 10 bodies and two groups of 100 bodies;
- two processes for the virtual processors of the network g into *alpha*; these processes compute the data for two groups of 600 bodies.

The simulation time of 15 hours of the galaxy evolution by the mpC program was 94 s.

The execution time of the MPI program depends heavily on the order of the number of bodies in the groups: it varies from 88 s to 391 s when simulating 15 hours of the galaxy evolution. The figure shows the ratio of the execution time of the MPI and mpC programs for various permutations of these numbers. All permutations can be decomposed into 24 disjoined subsets of the same cardinality such that two permutations belong

to the same subset if the execution time for them is the same. We assume that these subsets are numbered in such a manner that the subset with a greater number corresponds to the permutations that require a longer execution time of the MPI program. In the figure, every subset is represented by a column with the height equivalent to the ratio of the execution times t_{MPI}/t_{mpC}.

It is seen that the execution time of the MPI program exceeds (often, substantially) that of the mpC program for almost all input data.

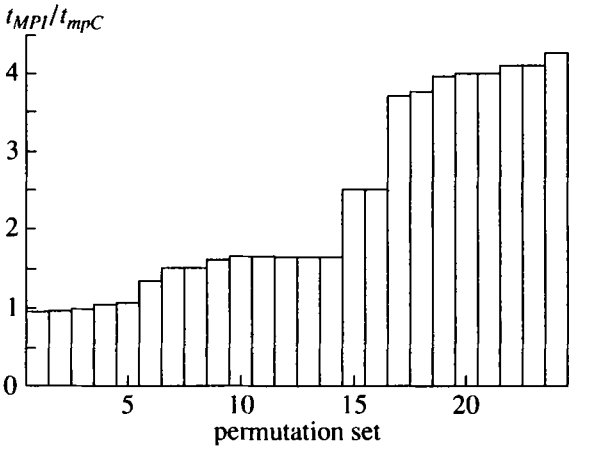
5.2 Regular Applications Developed in mpC

A salient feature of irregular problems is that they can be naturally decomposed into a small number of subproblems requiring a different number of computations. In turn, this natural decomposition leads to natural parallelism in solving the problem: the program can be executed as a small number of interacting concurrent processes, each of them solving a separate subproblem. An example of the irregular problem was considered in the preceding section (simulating the galaxy evolution).

A salient feature of regular problems is that they can be naturally decomposed into a relatively large number of small homogeneous subproblems that require the same amount of computations. Such a decomposition leads to natural parallelism in solving the problem: the program can be executed as a large number of identical small programs running concurrently and interacting by means of data exchange. An example of a regular problem is that of multiplying dense matrices considered in Section 2. The basic idea of solving a regular problem on a heterogeneous computer network is in reducing it to an irregular problem with the structure dependent on the topology of the computer network, rather than the natural topology of the problem. This is achieved by merging small homogeneous subproblems into larger ones; the number of new problems does not exceed the number of the available physical processes and the amount of computations is proportional to the power of those processors. Since mpC makes it possible to determine the topology of the computer system at runtime, the corresponding program can be written so as to execute efficiently on any heterogeneous computer network without modifying its source code or even recompilation.

In this section, we give an example of solving a complex regular applied problem using heterogeneous computer networks; more precisely, we present our experience in porting an application written in FORTRAN 77 using PVM (about 3000 lines of source code) for the Parsytec PowerXplorer supercomputer to a network of heterogeneous workstations. This the problem of simulating oil extraction.

The process of oil extraction under flooding was modeled by the following system of differential equations [12]:



Speedup obtained for various combinations of the number of bodies in groups.

$$m\frac{\partial S_w}{\partial t} + \operatorname{div}(\mathbf{u}_w) = q_w,$$

$$q_w = \begin{cases} q_+ \times F_w(\bar{S})\text{—for sources} \\ q_- \times F_w(S_w)\text{—for sinks} \\ 0\text{—outside of wells.} \end{cases} \quad (1)$$

$$\operatorname{div}(K(S)\operatorname{grad} P) = q, \quad (2)$$

where

$$K(S) = -k\left(\frac{k_1(S)}{\mu_1} + \frac{k_2(S)}{\mu_2}\right), \quad (3)$$

$$F(S) = \frac{k_1(S)/\mu_1}{k_1(S)/\mu_1 + k_2(S)/\mu_2}. \quad (4)$$

The initial values are specified (on the entire oil bed) as follows:

$$S_w|_{t=0} = \underline{S}, \quad P|_{t=0} = P_0. \quad (5)$$

$$\left.\frac{\partial S_w}{\partial n}\right|_\Gamma = 0, \quad \left.\frac{\partial P}{\partial n}\right|_\Gamma = 0. \quad (6)$$

Here S_w is the index of water saturation, S_o is the index of oil saturation, S is the index of bound water saturation, $\bar{S}$ is the index of critical water saturation, $\mathbf{u}_w$ and $\mathbf{u}_0$ are the filtering rates of water and oil, respectively, m is the porosity coefficient, k is the absolute permeability of the porous medium, $k_w(S_w)$ is the coefficient of the relative phase permeability of water, $k_o(S_w)$ is the coefficient of the relative phase permeability of oil (the relative phase permeabilities are experimentally obtained functions of the saturation of the displacing phase), $F_w(S_w)$ is the Buckley–Leverette function for the displacing phase, μ_w and μ_o are the coefficients of the dynamic viscosity of water and oil, respectively, q (q_- and q_+ are distinguished) are the vol-

Table 1. Performance of the parallel FORTRAN program implementing the implicit algorithm for solving the oil extraction problem executed on the multiprocessor system Parsytec PowerXplorer in the PARIX programming environment

Number of processors	ω	Number of iterations	Time (in s)	Speedup	Efficiency
1	1.197	205	120	1	100%
2	1.2009	211	64	1.875	94%
4	1.208	214	38	3.158	79%
8	1.22175	226	26	4.615	58%

ume sources and sinks of liquid (q_- is the volume of liquid extracted from the production well in the unit time and q_+ is the volume of liquid pumped in the injection well; outside the wells, $q = 0$), q_w is the source (sink) of water saturation, q_o is the source (sink) of oil saturation, t is the time, and P is the pressure in the bed (identical for both phases, since capillary forces are not taken into account).

Equations (1), (2) provide a model for the filtration of two-phase liquid (consisting of oil and water) through a porous medium in the water pressure mode. Equation (1) describes the transport of water saturation, and (2) is the diffusion (elliptic) equation for the pressure in the bed. This system of equations is solved for S_w (the water faction in the two-phase liquid) and the pressure P.

The numerical solution was sought in a symmetrical domain cut from the infinite homogeneous oil bed; on the boundary of this domain, the natural condition of zero flows is set. The numerical algorithm is based on explicit methods; namely, equation (1) was solved by the iterative secant method; equation (2) was solved by the iterative $(\alpha - \beta)$ algorithm [12]. To improve the convergence rate of the $(\alpha - \beta)$ algorithm, a relaxation parameter was included in the equations for certain sweep method coefficients.

The parallel implementation of the algorithm for executing on homogeneous multiprocessor systems was based on the decomposition of the computational domain (data parallelism): the domain was decomposed into subdomains of identical size along the Y-coordinate, and the computations for every subdomain were conducted concurrently by different processors of the supercomputer. This decomposition turned out to be more efficient than the decomposition along the X-coordinate and than the decomposition along both coordinates, since it requires a less intensive data exchange between the processors. In every subdomain, the system of equations (1), (2) was solved as follows.

Table 2. Relative performance of workstations

Number of the workstation	1	2	3–4	5–7	8–9
Performance	1150	575	460	325	170

For every time layer, water saturation was obtained by solving equation (1) using the values of the pressure obtained for the preceding time layer. This value of water saturation was then used to calculate the new value of the pressure for the current time layer by solving equation (2). Then, this procedure was repeated for the next time layer.

The main difficulty of this parallel algorithm lay in determining the optimal relaxation parameter for the $(\alpha - \beta)$ algorithm, since this parameter depends on the number of subdomains in the decomposition. The use of nonoptimal values of the relaxation parameter resulted in a considerable increase in the number of iterations, and even in the loss of convergence of the algorithm. The optimal value of the relaxation parameter for various numbers of subdomains in the decomposition was found by numerous experiments.

This algorithm was implemented in FORTRAN 77 with the use of the communication library PVM. It demonstrated remarkable scalability, speedup, and efficiency of parallelization when executed on the parallel computer Parsytec PowerXplorer—a multiprocessor system based on PowerPC-601 processors used as computation nodes and transputers T800 as communication nodes (a T800 transputer transfers data at the rate of 20 Mbit/s via 4 bi-directional links).

Table 1 presents the computation results for the first time layer. The efficiency of parallelization was determined as $(S_{real}/S_{ideal}) * 100\%$, where S_{real} is the actual speedup due to parallelization and S_{ideal} is the ideal speedup that might be achieved on the parallel computer system. The latter was determined as the ratio of the sum of the performances of the processors of the system to the performance of the base processor. The speedup was calculated relative to the execution time of the basic sequential program on the base processor. Note that the efficiency of parallelization is greater the faster the communication links and the slower the processors are.

The program under discussion was designed as a part of a portable software system able to work both on supercomputers and local networks of heterogeneous computers. Thus, a portable version of the program was needed to simulate oil extraction using computer networks.

Table 3. Performance of the parallel FORTRAN/PVM program modeling the oil extraction process run on subnetworks of workstations

Subnetwork (numbers of workstations)	ω	Number of iterations	Time (s)	Ideal speedup	Actual speedup	Efficiency
{2, 5}	1.2009	211	46	1.57	0.88	0.56
{5, 6}	1.2009	211	47	2.0	1.52	0.76
{2, 5–7}	1.208	214	36	2.7	1.13	0.42
{2–7}	1.21485	216	32	4.3	1.27	0.30
{2, 3, 5–8}	1.21485	216	47	3.8	0.87	0.23
{1–8}	1.22175	226	46	3.3	0.41	0.12

Table 4. Execution time of the sequential program simulating oil extraction when run on different workstations

Processor	Ultra SPARC-1	SPARC 20	SPARC station 4		SPARC 5			SPARC classic	
Workstation	1	2	3	4	5	6	7	8	9
Iterations	205								
Time (s)	18.5	40.7	51.2	51.2	71.4	71.4	71.4	133	133

As the first step of the development, the available FORTRAN/PVM program was ported (without any modifications) to the local network based on 10 Mbits Ethernet and consisting of 9 single-processor SUN workstations. To compare the performance of this network with the performance PowerPC-601 used in Parsytec PowerXplorer note that the least powerful workstation (SPARCclassic) executes the sequential program a bit more slowly than PowerPC-601, and the most powerful workstation (UltraSPARC-1) executes it more than six times as fast. The relative performances of the workstations (for the oil extraction problem) are presented in Table 2 (the workstations are assigned numbers used in the following tables).

Table 3 presents the results obtained when executing the FORTRAN/PVM program for a single time layer on different subnetworks of the network consisting of two, four, six, and eight workstations. The results include the value of the relaxation parameter and the corresponding number of β-iterations, the execution time, the ideal and actual speedup, and the efficiency of using the subnetwork. The speedup was calculated relative to the execution time of the sequential program on the most powerful workstation of the subnetwork (the execution time of the sequential program on different workstations is presented in Table 4). The noticeable decrease in the efficiency of parallelization as compared to Parsytec PowerXplorer is due to three reasons: slower communication lines, more powerful processors, and unbalanced workload of processors of different performance.

While the first two reasons are inevitable, the third one can be worked around by modifying the parallel algorithm underlying the FORTRAN/PVM program. Namely, to achieve the optimal workload of the processors, the computational domain is decomposed into subdomains of different size proportional to the performance of the processors that do the computation for them. More precisely, as a result of this decomposition, every subdomain contains the same number of columns of the grid, but a different number of rows. As to the relaxation parameter, it is reasonable to assume that its optimal value depends on the number of rows of the computational grid and use a specific value for each subdomain: $\omega = \omega(N_{row})$. We experimentally found a sequence of optimal values of the relaxation parameter for certain values of N_{row}. Then, for an arbitrary N_{row}, the value of ω was found by piecewise linear interpolation. Note that this approach yields a rather high convergence rate of the parallel (α – β) algorithm with relaxation (see the column "Number of iterations in Table 5).

The modified algorithm is very difficult to implement in portable form using PVM. The cause is that PVM, much like other message-passing libraries and HPF, does not support means for creating groups of processes depending on their relative performance. For this reason, the algorithm for simulating the oil extraction process was developed in mpC. This program determines (at runtime) the number and relative performance of the available processors, creates a group of processes such that every process is run by a separate

Table 5. Execution time of the parallel program simulating oil extraction when run on different subnetworks of workstations

Subnetwork (numbers of workstations)	Number of iterations	Time (s)	Actual speedup	Efficiency	Time for 205 iter. (s)	Speedup for 205 iter.	Efficiency for 205 iter.
{2, 5}	324	41.6	0.98	0.63	28.2	1.44	0.92
{5, 6}	225	38.8	1.84	0.92	36.4	1.96	0.98
{2, 5–7}	279	26	1.57	0.58	19.7	2.07	0.77
{2–7}	245	17.9	2.27	0.54	15	2.71	0.63
{2–8}	248	20.2	2.01	0.54	17	2.39	0.64
{2–8}*	260	32.8	1.24	0.33	26.8	1.52	0.40
{2–9}	268	21	1.94	0.40	16	2.54	0.53

* The computational domain was distributed into equal subdomains.

processor, and distributes data and computations proportionally to the relative performance of processors. We note that the mpC program not only suggests new functional capabilities, but is also three times shorter (in terms of the source code) compared to the original FORTRAN/PVM program.

On heterogeneous computer networks, the mpC program demonstrates moderate speedup and efficiency of parallelization (see Table 5), which are, however, much higher than those of the program written in FORTRAN/PVM (see Table 3). Despite the number of iterations being increased, the mpC program is executed more quickly thanks to the optimization of data exchange and, most importantly, thanks to the balanced workload of processors (cf. the column "Time" in Tables 5 and 3). To evaluate the net gain obtained by balancing the workload, the mpC program was executed on the same subnetwork consisting of the workstations number 2, 3, 5, 6, 7, and 8 twice. The first time, data were distributed according to the relative performance of the processors and the second time, the data were distributed uniformly. In the second case, the execution time was 1.5 times as long, and the speedup and efficiency of parallelization decreased correspondingly (marked by an asterisk in Table 5).

The moderate efficiency of parallelization of the mpC program can be largely explained by particularities of adaptation of the $(\alpha - \beta)$ algorithm with relaxation to heterogeneous networks. This algorithm is very sensitive to the accuracy of evaluating the values of ω, and the approximate procedure described above yields reasonable, but not the best possible, result. The number of iterations required for the convergence of the parallel algorithm is considerably different from that in the sequential algorithm. Thus, it would be interesting to compare the execution time required to perform 205 iterations (this number of iterations is required for the β process to converge when run by a single processor). The corresponding data are presented in Table 5, which shows that if we were able to avoid increasing the number of iterations (e.g., by a more accurate evaluation of the relaxation parameters), we could achieve remarkable speedup and parallelization efficiency for the mpC program simulating oil extraction.

6. RELATED WORKS

To our knowledge, all software systems designed for developing programs to be run on networks possess the following common property: either the programmer does not have the means for describing the virtual parallel system on which the program is to be run or these means are insufficient for determining an efficient distribution of computations and communications over the target network. Even topological capabilities of MPI (including MPI-2 [13]) are insufficient for solving this problem. For this reason, to guarantee that the program will be efficiently executed on a particular network, the user must employ means that are external relative to the language, such as the load scheme or application scheme [14]. If the user knows the topology of the target network (i.e., its structure and performance of the processors and communication lines) and the topology of the program (i.e., its parallel structure), then he can use configuration files to map the processes of the program onto the network processors so as to guarantee the most efficient execution. There exist systems that support this type of static distribution [15]. However, if the topology of the program becomes known only at runtime (e.g., depends on the initial data), this approach is inapplicable. There exist systems [16, 17] that realize some functions inherent in distributed operating systems: they try to take into account inhomogeneity of processor performances when managing tasks to maximize the performance of the computer network considered as a single computer. In contrast to these systems, mpC is designed to minimize the execution time of a

particular parallel program executed on a network, which is the most important thing for end users.

7. CONCLUSIONS

Computer network is the most general parallel architecture. The paper describes the mpC programming language and programming environment designed for developing efficiently portable modular programs to be run on computer networks.

The most important characteristics of mpC are as follows:

• mpC programs execute efficiently on any computer network without changes to the source code (we call this the efficient portability property);

• mpC makes it possible to develop programs that can not only can adapt to the normal performance of the processors, but also redistribute the computations and communications, depending on the dynamic variations of the processor workload in the network.

We have been experimenting with mpC for over two years and have developed a technology of using it for high-performance computations on heterogeneous networks. This technology was applied to solving the following problems.

• Efficient use of available parallel software designed for supercomputers on heterogeneous computer networks. The interface between mpC and ScaLAPACK that makes it possible to use the latter on heterogeneous networks provides an example (for details, see Section 4). The development of the interface took about a week, and porting a complex ScaLAPACK program to heterogeneous networks (using this interface) took several days.

• Rewriting parallel programs designed for supercomputers in mpC to be efficiently executed on heterogeneous networks. An example of such a problem is porting the program for simulating oil extraction from the supercomputer Parsytec to the network of workstations (for details, see Section 5.2). Originally, the program was written in FORTRAN 77 with calls of PVM. Developing the corresponding mpC program took about two weeks. This mpC program, which runs on a network consisting of eight workstations, is three times as fast as its FORTRAN/PVM analogue on this network and is twice as fast as the FORTRAN/PVM program that runs on the eight-processor segment of the supercomputer Parsytec.

• Parallelization of sequential programs for running on heterogeneous networks. For example, a parallel version of the classic adaptable FORTRAN program for numerical integration quanc8 [13]. This program uses the quadrature Newton–Cotes formula of the eighth order. In the case of complex (in terms of numerical computation) integrand functions, this mpC program considerably speeds up the computation of definite integrals using computers available in a local network. Note that this program automatically redistributes computations performed by computers depending on their current workload. The development of this program took two days.

• Developing original mpC programs. For example, we developed a parallel program for simulating the evolution of a system of bodies under Newtonian gravity attraction force (for details, see Section 5.1). This program demonstrated a considerable speedup (by many times) as compared to a thoroughly written MPI program that did not take heterogeneity into account.

We continue the work on mpC and its programming environment. The purpose of this work is to achieve the highest possible efficiency for a wide range of computer networks (including clusters of supercomputers and wide area networks) and to improve the program model.

ACKNOWLEDGMENTS

This work was supported by the Russian Foundation for Basic Research, project nos. 99-01-00205 and 99-07-90458.

REFERENCES

1. El-Rewini, H. and Lewis, T., *Introduction to Distributed Computing*, Manning, 1997.
2. *Message-Passing Interface Forum, MPI: A Message-Passing Interface Standard, ver. 1.1*, 1995.
3. Geist, A., Beguelin, A., Dongarra, J., Jlang, W., Manchek, R., and Sunderam, V., *PVM: Parallel Virtual Machine, Users' Guide and Tutorial for Networked Parallel Computing*, Cambridge: MIT Press, 1994.
4. *High Performance Fortran Forum, High Performance Fortran Language Specification, version 1.1*, Houston: Rice Univ., 1994.
5. Koelbel, C., Conferences for Scientific Applications, *IEEE Comput. Science Eng.*, 1998, vol. 5, no. 3, pp. 91–95.
6. Gaissaryan, S.S. and Lastovetsky, A.L., An ANSI C Superset for Vector and Superscalar Computers and Its Retargetable Compiler, *J. C Lang. Transl.*, 1994, vol. 5, no. 3, pp. 183–198.
7. *The C* Programming Language, CM-5 Technical Summary*, Thinking Machines Corporation, 1992, pp. 69–75.
8. Hatcher, P.J. and Quinn, M.J., *Data-Parallel Programming on MIMD Computers*, Cambridge: MIT Press, 1991.
9. Arapov, D., Kalinov, A., Lastovetsky, A., and Ledovskih, I., Experiments with mpC: Efficient Solving Regular Problems on Heterogeneous Networks of Computers via Irregularization, *Proc.Fifth Int. Symp. on Solving Irregularly Structured Problems in Parallel (IRREGULAR'98)*, Lect. Notes Comput. Sci., Berkley, 1998, no. 1457, pp. 332–343.
10. Chetverushkin, B., Churbanova, N., Lastovetsky, A., and Trapeznikova, M., Parallel Simulation of Oil Extraction on Heterogeneous Networks of Computers, *Proc. 1998 Conf. on Simulation Methods and Applications (CSMA'98)*, Society for Computer Simulation, Orlando, 1998, pp. 53–59.

11. Kalinov, A. and Lastovetsky, A., Heterogeneous Distribution of Computations While Solving Linear Algebra Problems on Networks of Heterogeneous Computers, *Proc. 7th Int. Conf. on High Performance Computing and Networking Europe (HPCN Europe'99), Amsterdam, 1999, Lect. Notes Comput. Sci.*, no. 1593, pp. 191–200.

12. Chetverushkin, B.N., Churbanova, N.G., and Trapeznikova, M.A., Simulation of Oil Production on Parallel Computing Systems, *Proc. Simulation MultiConference HPC'97: Grand Challenges in Computer Simulation*, Tentner, A., Ed., Atlanta, 1997, pp. 122–127.

13. *MPI-2: Extensions to the Message-Passing Interface*, http://www.mcs.anl.gov.

14. *Trollius LAM Implementation of MPI, Version 6.1*, Ohio: Ohio Univ., 1997.

15. Heinze, F., Schaefers, L., Scheidler, C., and Obeloeer, W., Trapper: Eliminating Performance Bottlenecks in a Parallel Embedded Application, *IEEE Concurrency*, 1997, vol. 5, no. 3, pp. 28–37.

16. *Dome: Distributed Object Migration Environment*, http://www.cs.cmu.edu/Dome/.

17. *Hector: A Heterogeneous Task Allocator*, http://www.erc.msstate.edu/russ/hpcc/.