The following changes were made to meet the referee's comments:

1. The paper was re-organized to have key material in the main paper and illustrative code in four appendices.

2. More discussion was given to existing work on scheduling and load balancing as well as the Network Weather Service and related ideas from San Diego and Tennessee were referenced (see Related Work in section 5).

A Parallel Language and Its Programming System for Heterogeneous Networks

Alexey Lastovetsky (contact author), Dmitry Arapov, Alexey Kalinov, Ilya Ledovskih

Institute for System Programming, Russian Academy of Sciences

25, Bolshaya Kommunisticheskaya str., Moscow 109004, Russia

E-mail: lastov@ispras.ru, phone: +7(095)9120754, fax: +7(095)9121524

Abstract

The paper presents a new parallel language, mpC, designed specially for programming high-performance computations on heterogeneous networks of computers, as well as its supportive programming environment. The main idea underlying mpC is that an mpC application explicitly defines an abstract network and distributes data, computations and communications over the network. The mpC programming environment uses at run time this information as well as information about any real executing network in order to map the application to the real network in such a way that ensures efficient execution of the application on this real network. Experience of using mpC for solving both regular and irregular real-life problems on networks of heterogeneous computers is also presented.

Keywords: parallel architectures, parallel programming languages, software tools, heterogeneous computing, efficient portable modular parallel programming.

1. Introduction

As far as ten years ago, parallel computer systems were restricted mostly by so-called supercomputers - distributed memory multiprocessors (MPPs) and shared memory multiprocessors (SMPs). Parallel computing on common networks of workstations and PCs did not make sense, since it could not speed up solving most of problems because of low performance of commodity network equipment. But in the 1990s, network capacity increases surpassed processor speed increases [1, pp.6-7]. Up-to-date commodity network technologies, such as Fast Ethernet, ATM, Myrinet, etc., enable data transfer between computers at the rate of hundreds Mbits per seconds and even Gigabits per second. This has led to the situation when not only specialized parallel computers, but also local networks of computers and even global ones could be used as parallel computer systems for high performance parallel computing. Announcement by President Clinton of a strategic initiative, aimed at 1000-fold speedup in the near future of communication rates in the Internet, supported by leading telecommunication, computer and software companies, clearly points to the started shift of high performance computing to network computing.

So, networks of computers become the most common parallel architecture available, and very often more performance can be achieved by using the same set of computers utilized via up-to-date network equipment as a single distributed memory machine rather than with a new more powerful computer.

The use of networks for parallel high performance computing is kept back only by the absence of appropriate system software. The point is that, unlike supercomputers, networks are inherently heterogeneous and consist of diverse computers of different performances interconnected via mixed network equipment providing communication links of different speeds and bandwidths. Therefore, the use of traditional parallel software, ported from (homogeneous) supercomputers, on a heterogeneous network makes the network behave as if it was a

homogeneous network consisting of weakest participating computers, just because the traditional software distributes data, computations and communications not taking into account the differences in performances of processors and communication links of the network. This sharply decreases the efficiency of utilization of the performance potential of networks and results in their poor usage for high performance parallel computing.

Currently, main parallel programming tools for networks are MPI [2], PVM [3] and HPF [4].

PVM (Parallel Virtual Machine) and MPI (Message Passing Interface) are message-passing packages providing, in fact, the assembler level of parallel programming for networks of computers. The low level of their parallel primitives makes the writing of really complex and useful parallel applications in PVM/MPI tedious and error-prone. In addition, the tools are not designed to support development of adaptable parallel applications, that is, such applications that distribute computations and communications in accordance with input data and peculiarities of the executing heterogeneous network. Of course, due to their low level, one may write a special run-time system to provide that property for his/her application, but such a system is usually so complicated that the necessity of its development can frighten off most of normal users.

HPF (High Performance Fortran) is a high-level parallel language originally designed for (homogeneous) supercomputers as the target architecture. Therefore, the only parallel machine visible when programming in HPF is a homogeneous multiprocessor providing very fast communications among its processors. HPF does not support irregular or uneven data distribution as well as coarse-grained parallelism. A typical HPF compiler translates an HPF program into a message-passing program in PVM or MPI, and the programmer cannot exert influence on the level of balance among processes of the target message-passing program. In addition, HPF is a very difficult language to compile. Even the most advanced HPF compilers produce target code running on homogeneous clusters of workstations in average 2-3 times

slower than the corresponding MPI counterparts (see [5] for a report on the 1998 HPF Users Group's annual meeting in Porto, Portugal). So, HPF is also not quite suitable for programming high-performance parallel computations on networks.

So, to utilize a heterogeneous network of computers (NoCs) as a single distributed memory machine, dedicated tools are needed.

The paper presents the first parallel language specially designed for programming NoCs as well as its supportive programming environment. The language, named mpC, is an ANSI C superset. Like HPF, it includes a vector subset, named the C[] language [6]. Traditionally, the target parallel architecture is implicitly embedded in a parallel language. In the most advanced cases, the target architecture is parameterized with a small number of parameters available to the programmer. This traditional approach to designing parallel languages is not applicable to a parallel language aimed at NoCs, since unlike all preceding parallel architectures, the NoC architecture is not of regular structure. The main idea, underlying the mpC language, is to provide language constructs allowing the user to define in details an abstract heterogeneous parallel machine, the most appropriate to his/her parallel algorithm. The mpC programming environment uses this information together with information about an executing physical parallel system in order to ensure efficient running the mpC application on the physical parallel system.

The paper is built as follows. Section 2 introduces the mpC language. Section 3 outlines principles of its implementation. Section 4 describes experiments with real-life mpC applications. Section 5 is about related work. Section 6 concludes the paper.

2. Outline of the mpC language

In mpC, the notion of *computing space* is defined as a set of virtual processors of different performances connected with links of different communication speeds accessible to the user for management.

The basic notion of the mpC language is *network object* or simply *network*. A network comprises virtual processors of different performances connected with links of different communication speeds. A network is a region of the computing space, which can be used to compute expressions and execute statements.

Allocating network objects in the computing space and discarding them is performed in similar fashion to allocating data objects in storage and discarding them. Conceptually, a virtual processor of some network already created initiates the creation of a new network. This virtual processor is called a *parent* of the created network. The parent belongs to the created network. The only virtual processor defined from the beginning of program execution till program termination is the pre-defined virtual *host-processor*.

Every network declared in an mpC program has a type. The type specifies the number and performances of virtual processors, links between these processors and their communication speeds, as well as separates the parent. For example, the network-type declaration

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

introduces network type `Rectangle` that corresponds to networks consisting of 4 virtual processors of different performances interconnected with undirected links of the normal speed in a rectangular structure.

In this example, line 1 is a *header* of the network-type declaration. It introduces the name of the network type.

Line 2 is a *coordinate declaration* declaring the coordinate system to which virtual processors are related. It introduces integer coordinate variable `I` ranging from 0 to 3.

Line 3 is a *node declaration*. It relates virtual processors to the coordinate system declared and declares performances. It stands for the predicate ***for all `I<4` if `I<2` then*** *a virtual processor, whose relative performance is specified by the value of* ***`I+1`****, is related to the point with coordinate* ***`[I]`***. Expression `I+1` is called a *power specifier*. It is meant that the greater value of the power specifier, the more performance is specified. Namely, in this example the 0-th virtual processor is twice slower than the 1-st one, triple slower than the 2-nd one, and four times slower than the 3-rd one. For any network of this type, this information allows the compiler to associate an absolute weight with each virtual processor of the network normalizing it in respect to the weight of the parent.

Lines 4-7 are a *link declaration*. It specifies links between virtual processors. Line 5 stands for the predicate ***for all `I<4` if `I>0` then*** *there exists a link of normal speed connecting virtual processors with coordinates* ***`[I]`*** *and* ***`[I-1]`***, and line 6 stands for the predicate ***for all `I<4` if `I==0` then*** *there exists a link of normal speed connecting virtual processors with coordinates* ***`[I]`*** *and* ***`[3]`***. Note, that if no link between two virtual processors is specified explicitly, it means not absence of a link but existence of a very slow link.

Line 8 is a *parent declaration*. It specifies that the parent has coordinate [0].

With the network type declaration, the user can declare a network identifier of this type. For example, the declaration

```
net Rectangle r1;
```

introduces identifier `r1` of network.

The notion of *distributed data object* is introduced in the spirit of C* [7] and Dataparallel C [8]. Namely, a data object distributed over a region of the computing space comprises a set of components of any one type so that each virtual processor of the region holds one component. For example, the declarations

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

declare:

- integer variable `Derror` distributed over the entire computing space;
- integer 10-member array `Da` distributed over the network `r2`;
- undistributed floating variable `f` belonging to the virtual host-processor;
- floating variable Df distributed over a subnetwork of `r2`;
- integer variable `Di` replicated over the entire computing space.

By definition, a distributed object is *replicated* if all its components are equal to each other.

The notion of *distributed value* is introduced similarly.

In addition to a network type, the user can declare a parameterized family of network types called *topology* or *generic network type*. For example, the declaration

```
/* 1 */  nettype Ring(n, p[n]) {
/* 2 */    coord I=n;
/* 3 */    node {
```

```
/* 4 */      I>=0: p[I];
/* 5 */    };
/* 6 */    link {
/* 7 */      I>0:  [I]<->[I-1];
/* 8 */      I==0: [I]<->[n-1];
/* 9 */    };
/* 10 */   parent [0];
/* 11 */ };
```

introduces topology `Ring` that corresponds to networks consisting of `n` virtual processors interconnected with undirected links of normal speed in a ring structure.

The header (line 1) introduces parameters of topology `Ring`, namely, integer parameter `n` and vector parameter `p` consisting of `n` integers.

Correspondingly, coordinate variable `I` ranges from `0` to `n-1`, line 4 stands for the predicate ***for all `I<n` if `I>=0` then*** *a virtual processor, whose relative performance is specified by the value of `p[I]`, is related to the point with coordinate `[I]`,* and so on.

With the topology declaration, the user can declare a network identifier of a proper type. For example, the fragment

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

introduces identifier `rr` of the network, the type of which is defined completely only at run time.

Network `rr` consists of m virtual processors, the relative performance of `i`-th virtual processor being characterized by the value of `n[i]`.

A network has a computing space duration that determines its lifetime. There are 2 computing space durations: static, and automatic. A network declared with *static* computing space duration is created only once and exists till termination of the entire program. A new instance of a network declared with *automatic* computing space duration is created on each entry into the block in which it is declared. The network is discarded when execution of the block ends.

Now, let us consider a simple mpC program (see Appendix A) that multiplies two dense square matrices `X` and `Y` using a number of virtual processors, each of which computes a number of rows of the resulting matrix `Z`.

The program includes 5 functions - `main` defined here, `Input` and `Output` defined in other source files, and the library functions `MPC_Processors` and `MPC_Partition_1b`. Line 5 declares functions `Input` and `Output`, functions `MPC_Processors` and `MPC_Partition_1b` are declared in the header file `mpc.h`. In general, mpC allows 3 kinds of functions. Here, functions of all these kinds are used: `main` is a *basic* function, `MPC_Processors` and `MPC_Partition_1b` are *nodal* functions, and `Input` and `Output` are *network* functions.

A call to a *basic function* is an *overall* expression (that is, an expression evaluated by the entire computing space; no other computations can be performed in parallel with evaluation of an overall expression). Its arguments (if any) shall either belong to the host-processor or be distributed over the entire computing space, and the returning value (if any) shall be distributed over the entire computing space. In contrast to other kinds of functions, a basic function can define network objects. In line 11, the construct `[*]`, placed just before the function identifier

`main`, specifies, that an identifier of basic function be declared.

A *nodal function* can be completely executed by any one processor. Only local data objects of the executing processor can be created in such a function. In addition, the corresponding component of an externally-defined distributed data object can be used in the function. A declaration of nodal function does not need any additional specifiers. From the point of view of mpC, any pure C function is nodal.

In general, a *network function* is called and executed on a region of the computing space, and its arguments and value (if any) are also distributed over this region. Two network functions can be executed in parallel, if the corresponding regions are not intersected. `Input` and `Output` are representatives of the simplest form of network function, when a function can be called only on a statically defined region of the computing space. Here, functions `Input` and `Output` are declared in line 5 as network functions that can be called and executed only on the host-processor (it is specified with the construct `[host]` placed just before the function identifiers). So, calls to the functions in line 16 and 36 are executed on the host-processor.

Lines 11-38 contain the definition of `main`. Line 13 contains the definition of the arrays `x`, `y` and `z` all belonging to the host-processor.

Line 14 defines the integer variable `nprocs` replicated over the entire computing space. The distribution is specified implicitly without the use of the construct `[*]`. In general, such an implicit distribution is the distribution of the smallest block enclosing the corresponding declaration and having an explicitly specified distribution. Here, the declaration in line 14 is enclosed by the body of the function `main`, explicitly distributed over the entire computing space.

Line 15 defines the variable `powers` of the type *pointer to* `double` distributed over the entire computing space. The declaration specifies that any distributed data object, pointed by

powers, be replicated.

Line 17 calls to the library nodal function MPC_Processors on the entire computing space returning the number of actual processors and their relative performances. So, after this call, nprocs holds the number of actual processors, and powers points to the initial element of an nprocs-element replicated array holding the relative performances of the actual processors.

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

Line 20 calls to the library nodal function MPC_Partition_1b on the entire computing space. Based on the number and relative performances of the actual processors, this function calculates how many rows of the resulting matrix are computed by each of the actual processors. So, after this call, ns[i] holds the number of rows computed by i-th actual processor. In general, MPC_Partition_1b divides the given whole (specified in the above call with N) into a number of parts in accordance with the given proportions.

Line 22 defines the automatic network w consisting of nprocs virtual processors, the relative performance of the i-th virtual processor being characterized by the value of ns[i]. So, its type is defined completely only at run time. Network w, which executes the rest of computations and communications, is defined in such a way, that the more powerful is the virtual processor, the greater number of rows it computes. Note, that the mpC environment will ensure the optimal mapping of the virtual processors constituting w into a set of processes constituting the entire computing space. So, just one process from the processes running on each of actual processors will be involved in the multiplication of matrices, and the more powerful is the actual processor, the greater number of rows its process will compute.

Line 23 defines the integer variable myn distributed over w.

The result of the binary operator `coordof` (in line 24) is an integer value distributed over `w`, each component of which is equal to the value of coordinate `I` of the virtual processor to which the component belongs. The right operand of the operator `coordof` is not evaluated and used only to specify a region of the computing space. Note that coordinate variable `I` is treated as an integer variable distributed over the region. So, after execution of the asynchronous statement in line 24, each component of `myn` will contain the number of rows of the resulting matrix computed by the corresponding virtual processor.

Line 26 defines the integer variables `i` and `j` both replicated over `w`.

Line 27 defines three arrays distributed over `w`, namely, `dy` of the static type *array of `N` arrays of `N doubles`*, as well as `dx` and `dz`, both of the dynamic type *array of `myn` arrays of `N doubles`*. Note, that the number `myn` of elements of the arrays `dx` and `dz` is not the same for different components of the distributed arrays.

Line 28 contains an unusual unary postfix operator, `[]`. The point is that, strictly speaking, mpC is a superset of the vector extension of ANSI C, named the C[] language [14], where the notion of *vector*, defined as an ordered sequence of values of any one type, is introduced. In contradiction to an array, a vector is not a data object but just a new kind of value. In particular, the value of an array is a vector. The operator `[]` was introduced to support access to an array as a whole. It has an operand of the type *array of type* and blocks conversion of the operand to pointer. So, `y[]` designates array `y` as a whole, and `dy[]` designates distributed array `dy` as a whole.

The statement in line 28 broadcasts matrix `Y` from the virtual host-processor to all virtual processors of `w`. As a result, each component of the distributed array `dy` will contain this matrix. In general, if the left operand of the = operator is distributed over some region R of the computing space, the value of the right operand belongs to a virtual processor of some network

or hard subnetwork enclosing R and may be assigned without a type conversion to a component of the left operand, then the execution of the operator consists in sending the value of the right operand to each virtual processor of R, where the value is assigned to the corresponding component of the left operand.

The statement in line 29 scatters matrix `X` from the virtual host-processor to all virtual processors of `w`. As a result, each component of `dx` will contain the corresponding portion of matrix `X`.

In general, the first operand of the quaternary scatter operator `=::` should be an array of *type* distributed over some region R (say, consisting of K virtual processors). The second (if any), third (if any) and fourth operands should be undistributed and belong to a virtual processor of some network or hard subnetwork enclosing R. The second operand is optional and should either point to a pointer to the initial element of an integer K-element array or point to the integer K-element array. The third operand is optional and should either point to a pointer to the initial element of a K-element integer array, the value of the i-th element of which having to be not greater than the number of elements of the i-th component of the first operand, or point to the integer K-element array. The fourth operand should be an array of type.

The execution of `e1=e2:e3:e4` consists in cutting from the array `e4` K subarrays (possibly, overlapped) and sending the value of the i-th subarray to the i-th virtual processor of R, where it is assigned to the corresponding component of the distributed array `e1`. The displacement of the i-th subarray (relative to the initial element of the array `e4`) is defined by the value of the i-th element of the array, pointed to by `e2`, and the length of the subarray is defined by the value of the i-th element of the array pointed to by `e3`.

If `e3` points to a pointer having the `NULL` value, the scatter operator is executed, as if `*e3` points to the initial element of a K-element integer array, the value of the i-th element of which is

equal to the length of the i-th component of `e1`. Moreover, in this case such an array is really formed as a result of the operator execution, and the pointer to its initial element is assigned to `*e3`.

If `e2` points to a pointer having the `NULL` value, the scatter operator is executed, as if `*e2` points to the initial element of a K-element integer array, the value of the 0-th element of which is equal to 0 and the value of the i-th element of which is equal to the sum of the value of its (i-1)-th element and the value of the i-th element of the array pointed to by `e3`. Moreover, in this case such an array is really formed as a result of the operator execution, and the pointer to its initial element is assigned to `*e2`.

The second operand may be omitted. In this case the scatter operator is executed as if `e2` points to a pointer having the `NULL` value. The only difference is that the formed K-element integer array is discarded.

The third operand may be omitted. In this case the scatter operator is executed as if `e3` points to a pointer having the `NULL` value. The only difference is that the formed K-element integer array is discarded.

So, `dx[]=::x[]` in line 29 results in splitting `N`-element array `x` of arrays of `N doubles` into `nprocs` subarrays, the length of the i-th subarray being equal to the value of the component of `myn` belonging to the i-th virtual processor (that is, to the value of `[w:I==i]myn`). The same operation could be done faster if to use the form `dx=:&ns:x`, avoiding additional (and redundant in this case) communications and computations needed to form the omitted operands.

The asynchronous statement in lines 30-32 computes the corresponding portions of the resulting matrix `Z` on each of virtual processors of `w` in parallel. So, after the statement the distributed array `dz` will hold the portions.

Finally, the statement in line 33 gathers these portions to the virtual host-processor forming the resulting array `z`, using the gather operator `:::=`. This quaternary operator matches the scatter operator. They execute in similar ways inverse communication operations.

3. Implementation of mpC

Currently, the mpC programming environment includes a compiler, a run-time support system (RTSS), a library, and a command-line user interface.

The compiler translates a source mpC program into the ANSI C program with calls to functions of the RTSS. It uses optionally either the SPMD model of target code, when all processes constituting a target message-passing program run identical code, or a quasi-SPMD model, when it translates a source mpC file into 2 separate target files - the first for the virtual host-processor and the second for the rest of virtual processors.

The RTSS manages the computing space that consists of a number of processes running over a target DMM as well as provides communications. It has a precisely specified interface and encapsulates a particular communication package (currently, a small subset of MPI). It ensures platform-independence of the rest of system components.

The library consists of a number of functions that support debugging mpC programs as well as provide some low-level efficient facilities.

The command-line user interface consists of a number of shell commands supporting the creation of a virtual parallel machine and the execution of mpC programs on the machine. While creating the machine, its topology is detected by a topology detector that runs a special parallel test code and saves the topology information in a file used by the RTSS.

3.1. Model of target program

All processes constituting the target program are divided into 2 groups - a special process, so-called *dispatcher*, playing the role of computing space manager, and common processes, so-called *nodes*, playing the role of virtual processors of the computing space. The dispatcher works as a server. It receives requests from nodes and sends them commands.

In the target program, every network of the source mpC program is represented by a set of nodes called a *region*. At any time of the target program running, any node is either free or hired in one or several regions. Hiring nodes in created regions and dismissing them are responsibility of the dispatcher. The only exception is the pre-hired *host-node* representing the mpC pre-defined virtual host-processor. Thus just after initialization the computing space is represented by the host and a set of temporarily free (unemployed) nodes.

The main problem in managing processes is hiring them to network regions and dismissing them. Solution of this problem establishes the whole structure of the target code and forms requirements for functions of the run-time support system.

To create a network region, its parent node computes, if necessary, parameters of the corresponding network topology and sends a creation request to the dispatcher. The request contains full topological information about the created region including the number of nodes and their relative performances. On the other hand, the dispatcher keeps information about the topology of the target network of computers including the number of actual processors, their relative performances and the mapping of nodes onto the actual processors. Based on the topological information, the dispatcher selects a set of free nodes, which are the most appropriate to be hired in the created network region. After that, it sends to every free node a message saying whether the node is hired in the created region or not.

To deal locate a network region, its parent node sends a message to the dispatcher. Note, that

the parent node remains hired in the parent-network region of the deal located region. The rest of members of the deal located network region become free and begin waiting for commands from the dispatcher.

Any node can detect its hired/free status. It is hired if a call to function `MPC_Is_busy` returns 1. If such a call returns 0, the node is free.

Any node can detect if it is hired in some particular region or not. A region is accessed via its descriptor. If the descriptor `rd` corresponds to the region, then a node belongs to the region if and only if the function call `MPC_Is_member(&rd)` returns 1. In this case, descriptor `rd` allows the node to obtain comprehensive information about the region as well as identify itself in the region.

When a free node is hired in a network region, the dispatcher must let the node know, in which region it is hired, that is, must specify the descriptor of that region. The simplest way - to pass the pointer to the region descriptor from the parent node through the dispatcher to the free node, is senseless for distributed memory systems not having common address space. Therefore, in addition to the region descriptor, something else is needed to identify the created region in a unique fashion. The additional identifier must have the same value on both the parent and the free node and be transferable from the parent node through the dispatcher to the free node.

In a source mpC program, a network is denoted by its name, being an ordinary identifier and not having to have file scope. Therefore, the network name cannot serve as a unique network identifier even within a file. One could enumerate all networks declared in the file and use the number of a network as an identifier unique within the file. However, such an identifier being unique within a file may not be unique within the whole program that may consist of several files. Nevertheless, one can use it without collisions when creating network regions, if during network-region creation all participating nodes execute the target code located in the same file.

Our compiler just enumerates networks defined in the same file and uses their numbers as network identifiers in target code when creating the corresponding network regions. It does ensure that during the creation of a network region all involved nodes are executing the target code located in the same file.

Creating a network region involves its parent node, all free nodes and the dispatcher. The parent node calls to function

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

where `name` contains the unique number of the created network in the file, and `net` points to the corresponding region descriptor. The function computes all topological information and sends a creation request to the dispatcher.

Meantime, free nodes are waiting for commands from the dispatcher at a so-called *waiting point* calling the function

```
MPC_Offer(MPC_Names* names, MPC_Net** nets, int count);
```

where `names` is an array of numbers of all networks the creation of which are expected at the waiting point, `nets` points to an array of pointers to descriptors of the regions the creation of which are expected at the waiting point, and `count` contains the number of elements in array `names`.

The correspondence between the network numbers and region descriptors is established in the following way. If a free node receives from the dispatcher a message saying that it is hired in a network the number of which is equal to `names[i]`, then the node is hired in the network region the descriptor of which is pointed by `nets[i]`.

A free node leaves the waiting function `MPC_Offer` either after it becomes hired in a network region or after the dispatcher sends to all free nodes the command to leave the current

waiting point.

3.2. Structure of target code for mpC block

In general, target code for an mpC block having network definitions has two waiting points. At the first, called *creating waiting point*, free nodes are waiting for commands on region creations. At the second, called *deallocating waiting point*, they are waiting for commands on region deallocation. In general, free nodes not only participate in creation/deallocation of regions for networks defined in the mpC block, but also take part in overall computations (that is, in computations distributed over the entire computing space) and/or in creation/deallocation of regions for networks defined in nested blocks. We call the first mpC statement in the block involving all free nodes in its execution a *waiting-point break statement*.

In the most general case, the compiler generates target code depicted in Appendix B. If the source mpC block does not contain a waiting-point break statement (that is, overall statements and nested blocks with network definitions or overall statements), then creating and deallocating waiting points can be merged. We call such a waiting point *shared waiting point*. Target code for the mpC block with a shared waiting point looks is given in Appendix C.

To ensure that during the creation of a network region all involved nodes execute target code located in the same file, the compiler puts a global barrier into the epilogue of waiting point.

The following scenario ensures the coordinated arrival of the nodes at the epilogue of waiting point:

- the host makes sure that all other hired nodes, which might send a creation/deallocation request expected in the waiting point, have already reached the epilogue;

- after that, the host sends a message to the dispatcher, saying any creation/deallocation request expected at this waiting point will not come yet;

- after receiving the message, the dispatcher sends all free nodes a command ordering to leave the waiting point;

- after receiving the command, each free node leaves the waiting function and reaches the epilogue.

3.3. Mapping algorithm

We have mentioned, that creation of a network region includes selecting appropriate free nodes for the region by the dispatcher. The section explains how the dispatcher performs such selection.

For every network-type definition in the source mpC program, the compiler generates 6 topological functions that are used at run time to compute diverse topological information about a network object of a relevant type. The first function returns the total number of nodes in the network object. Since the RTSS uses a linear numeration of nodes from `0` to `n-1`, where `n` is the total number of nodes, the second and third functions convert the coordinates of a node into its linear number and vice versa. This linear numeration is determined by lexicographic ordering on the set of coordinates of processor nodes. The fourth function returns the linear number of the parent node. The fifth function returns the type and the relative performance of the specified node. Finally, the sixth returns the length of the directed link connecting the specified nodes.

While selecting free nodes for a created network region, the dispatcher first calls to the corresponding topological functions to compute relative performances of virtual processors of the corresponding network as well as lengths of links between them. The relative performance of a virtual processor is represented by a positive floating number, the parent virtual processor having the relative performance equal to 1. After, the dispatcher computes the absolute performance of each virtual processor by multiplying its relative performance by the parent virtual processor absolute performance.

On the other hand, the dispatcher holds a map of the computing space reflecting its topological properties. The initial state of the map is formed during the dispatcher initialization and contains the following information:

- the number of actual computing nodes and their performances;
- a set of nodes associated with each of the actual computing nodes;
- for every pair of actual computing nodes, the time of message-passing initialization and the time of transferring data blocks of different sizes;
- for every actual computing node, the time of message-passing initialization and the time of transferring data blocks of different sizes between a pair of processes both running on the actual computing node.

In general, performance of an actual computing node is characterized by two attributes. The first says how quickly a single process runs on the actual computing node, and the second, named *scalability,* says how many non-interacting processes may run on the actual computing node in parallel without loss of speed. The latter attribute is useful, for example, if the actual computing node is a multiprocessor workstation.

Currently, the selection procedure performed by the dispatcher is based on the following simplest scheme.

The dispatcher assigns a weight, being a positive floating number, to every virtual processor of the allocated network in such a way that:

- the more powerful is the virtual processor, the greater weight it obtains;
- the shorter are the links outgoing from the virtual processor, the greater weight it obtains.

These weights are normalized in such a way, that the host-processor is of the weight equal to 1. Similarly, the dispatcher transforms the map of the computing space in such a way, that every actual computing node obtains a weight, being a positive floating number characterizing its computational and communicational power when running a single process.

The dispatcher appoints free nodes of the computing space to virtual processors successively, starting from the virtual processor having the greatest weight v. It tries to appoint the most powerful free node to this virtual processor. To do it, the dispatcher estimates the power of a free node for each actual computing node as follows. Let w be a weight of the actual computing node P, and N be a scalability of P. Let the set H of hired nodes associated with P be divided into N subsets $h_1,...,h_N$. Let v_{ij} be the weight of the virtual process currently occupying j-th node in h_j. Then the estimation of the power of a free node associated with P is equal to

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

Thus, if the estimation is maximum for the actual computing node P, the dispatcher appoints some free node, associated with P, to the virtual processor and adds this virtual processor to h_k such that

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

The above procedure appeared good enough for networks of workstations and ensures an optimal allocation, if just one node of the computing space is associated with each of actual processors.

4. Experiments with mpC

The first implementation of the mpC programming environment appeared by the end of

1996. For more than 3 years it has been freely available via Internet (at http://www.ispras.ru/~mpc), and version 2.0.4 is currently preparing. There have been already more than 700 installations of the mpC programming environment around the world, and a number of organizations are experimenting with mpC, mainly, for parallel solving scientific problems on local networks of workstations and PCs. These problems include matrix computations, N-body problem, linear algebra problems (LU decomposition, Cholesky factorization, etc.), numerical integration, simulation of oil production, calculation of building strength and many others. These experiments have demonstrated that mpC allows developing portable modular parallel applications, much faster solving both regular and irregular problems on heterogeneous networks of diverse workstations and PCs as well as irregular problems on homogeneous multiprocessors than their counterparts developed with traditional tools.

The section presents a few typical mpC applications.

4.1. Irregular mpC application

Let us consider an irregular application simulating the evolution of a system of stars in a galaxy (or a set of galaxies) under the influence of Newtonian gravitational attraction.

Let our system consist of a few large groups of bodies. It is known, that since the magnitude of interaction between bodies falls off rapidly with distance, the effect of a large group of bodies may be approximated by a single equivalent body, if the group of bodies is far enough away from the point at which the effect is being evaluated. Let it be true in our case. So, we can parallelize the problem, and our application will use a few virtual processors, each of which updates data characterizing a single group of bodies. Each virtual processor holds attributes of all the bodies constituting the corresponding group as well as masses and centers of gravity of other groups. The attributes characterizing a body include its position, velocity and mass.

Finally, let our application allow both the number of groups and the number of bodies in

each group to be defined in run time.

The application implements the following scheme:

```
Initialization of a galaxy on the virtual host-processor
Creation of a network
Scattering groups over virtual processors of the network
Parallel computing masses of groups
Interchanging the masses among virtual processors
while(1) {
  Visualization of the galaxy on the virtual host-processor
  Parallel computation of centers of gravity of groups
  Interchanging the centers among virtual processors
  Parallel updating groups
  Gathering groups on the virtual host-processor
}
```

The corresponding mpC program is given in Appendix D. This mpC source file contains the following external definitions:

- the definitions of variables `M`, `t` and arrays `N`, `Galaxy` all belonging to the virtual host-processor;
- the definition of variable `dM` and array `dN` both replicated over the entire computing space;
- the definition of the network type `Galaxynet`;
- the definition of the basic function `Nbody` with one formal parameter `infile` belonging to the virtual host-processor;
- the definitions of the network functions `Mint` and `Cint`.

In general, a *network function* is called and executed on some network or subnetwork, and the value it returns is also distributed over this region of the computing space. The header of the definition of the network function either specifies an identifier of a global static network or subnetwork, or declares an identifier of the network being a special formal parameter of the function. In the first case, the function can be called only on the specified region of the computing space. In the second case, it can be called on any network or subnetwork of a suitable type. In any case, only the network specified in the header of the function definition may be used in the function body. No network can be declared in the body. Only data objects belonging to the network specified in the header may be defined in the body. In addition, corresponding components of an externally-defined distributed data object may be used. Unlike basic functions, network functions (as well as nodal functions) can be called in parallel.

Network functions `Input` and `VisualizeGalaxy`, both associated with the virtual host-processor, as well as the nodal function `UpdateGroup` are declared and called in the program.

Automatic network `g`, executing most of computations and communications, is defined in such a way, that it consists of `M` virtual processors, and the number of bodies in the group, which it computes, characterizes the relative performance of each processor.

So, the more powerful is the virtual processor, the larger group of bodies it computes. The mpC programming environment bases on this information to map the virtual processors constituting network `g` into the processes constituting the entire computing space in the most appropriate way. Since it does it at run time, the user does not need to recompile this mpC program, to port it to another NoC.

Call expression `([g]UpdateGroup)(...)` causes parallel execution of nodal function `UpdateGroup` on all virtual processors of network `g`. It is meant, that function name `UpdateGroup` is converted to a pointer-to-function distributed over the entire computing

space, and operator `[g]` cuts from this pointer a pointer distributed over `g`. So, the value of expression `[g]UpdateGroup` is a pointer-to-function distributed over `g`. Therefore, expression `([g]UpdateGroup)(...)` denotes a distributed call to a set of undistributed functions.

Network functions `Mint` and `Cint` have 3 special formal parameters. Network parameter `p` denotes the network executing the function. Parameter `m` is treated as a replicated over `p` integer variable, and parameter `n` is treated as a pointer to the initial member of an integer unmodifiable `m`-member array replicated over `p`. The syntactic construct `([(dM,dN)g])`, placed on the left of the name of the function called in the call expressions in the function `Nbody`, just specifies the actual arguments corresponding to the special formal parameters.

The running time of the mpC program was compared to its carefully written MPI counterpart. Three workstations - SPARCstation 5 (hostname `gamma`), SPARCclassic (`omega`), and SPARCstation 20 (`alpha`), connected via 10Mbits Ethernet were used as a NoC. There were 23 other computers in the same segment of the local network. LAM MPI version 5.2 [12] was used as a particular communication platform.

The computing space of the mpC programming environment consisted of 15 processes, 5 processes running on each workstation. The dispatcher run on `gamma` and used the following relative performances of the workstations obtained automatically upon the creation of the virtual parallel machine: 1150 (`gamma`), 331 (`omega`), 1662 (`alpha`).

The MPI program was written in such a way to minimize communication overheads. All our experiments dealt with 9 groups of bodies. Three MPI processes was mapped to `gamma`, 1 process to `omega`, and 5 processes to `alpha`, providing the optimal mapping if the numbers of bodies in these groups were equal to each other.

The first experiment compared the mpC and MPI programs for homogeneous input data

when all groups consisted of the same number of bodies. In fact, it showed how much we pay for the usage of mpC instead of pure MPI. It turned out that the running time of the MPI program consists about 95-97% of the running time of the mpC program. That is, in this case we loose 3-5% of performance.

The second experiment compared these programs for heterogeneous input data. The groups consisted of 10, 10, 10, 100, 100, 100, 600, 600, and 600 bodies correspondingly.

The running time of the mpC program did not depend on the order of the numbers. In any case, the dispatcher selected:

- 4 processes on `gamma` for virtual processors of network `g` computing two 10-body groups, one 100-body group, and one 600-body group;
- 3 processes on `omega` for virtual processors computing one 10-body group and two 100-body groups;
- 2 processes on `alpha` for virtual processors computing two 600-body groups.

The mpC program took 94 seconds to simulate 15 hours of the galaxy evolution.

The running time of the MPI program essentially depended on the order of these numbers. It took from 88 to 391 seconds to simulate 15 hours of the galaxy evolution dependant on the particular order. Figure 1 shows the relative running time of the MPI and mpC programs for different permutations of these numbers. All possible permutations can be broken down into 24 disjoint subsets of the same power in such a way that if two permutations belong to the same subset, the corresponding running time is equal to each other. Let these subsets be numerated so that the greater number the subset has, the longer time the MPI program takes. In figure 1, each such a subset is represented by a bar, the height of which is equal to the corresponding value of t_{MPI}/t_{mpC}.

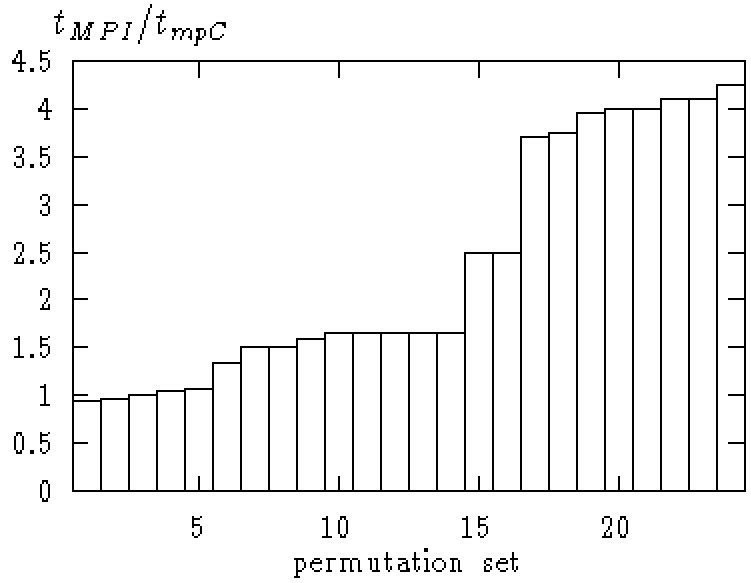

Figure 1: Speedups for different permutations of the numbers of bodies in groups.

One can see that almost for all input data the running time of the MPI program exceeds (and often, essentially) the running time of the mpC program.

4.2. Regular mpC application

An irregular problem is characterized by some inherent coarse-grained or large-grained structure implying quite deterministic decomposition of the whole program into a set of processes running in parallel and interacting via message passing. As rule, there are essential differences in volumes of computations and communications to perform by different processes. The N-body problem is just an example of an irregular problem.

Unlike an irregular problem, for a regular problem decomposition of the whole program into a large set of small equivalent programs, running in parallel and interacting via message passing, is the most natural one. Multiplication of dense matrices, discussed in section 2, is an example of a regular problem. The main idea of efficient solving a regular problem is to reduce it to an irregular problem the structure of which is determined by the topology of underlying hardware rather than the topology of the problem. So, the whole program is decomposed into a set of programs, each made of a number of the small equivalent programs stuck together and running

on a separate processor of the underlying hardware.

The section presents the experience of solving a real-life regular problem - simulation of oil extraction - in heterogeneous parallel environments. More exactly, the corresponding story is about the use of mpC for porting an application, written in Fortran 77 with calls to PVM to simulate oil extraction (about 3000 lines of source code), from a Parsytec supercomputer to a local network of heterogeneous workstations.

The oil-extraction process by means of non-piston water displacement was described by the following system of equations [12]:

$$m \times (dS_? / dt) + div(F_?(S_?) K(S_?) grad(P)) = q \times F_?(\underline{S}) \quad (1)$$

$$m \times (dS_? / dt) + div(F_?(S_?) K(S_?) grad(P)) = q \times F_?(S_?) \quad (2)$$

$$div(K(S_?) grad(P)) = q \quad (3)$$

where

$$K(S_?) = -k \times (k_?(S_?)/\mu_? + k_0(S_?)/\mu_0) \quad (4)$$

$$F_?(S_?) = (k_?(S_?)/\mu_?)/(k_?(S_?)/\mu_? + k_0(S_?)/\mu_0) \quad (5)$$

Initial and boundary conditions were

$$S_?|_{t=0} = \underline{S}, \qquad P|_{t=0} = P_0 \quad (6)$$

$$(dS_?/dn)|_G = 0, \qquad (dP/dn)|_G = 0 \quad (7)$$

Equation (1) was the water fraction transport equation at sources, and equation (2) was the water fraction transport equation in domain. Equation (3) was the pressure elliptic equation. Solutions of this system were water saturation $S_?$ (fraction of water in the fluid flow) and pressure in the oil field P. These equations included coefficients for medium characteristics: the coefficient of porosity (m), the absolute permeability (k) and nonlinear relative phase

permeabilities of oil ($k_0(S_в)$) and water ($k_в(S_в)$), the viscosities of oil (μ_0) and water ($\mu_в$), the function of sources/sinks (q), critical and connected values of water saturation (S and $\underline{S}$) and the strongly nonlinear Bucley-Leverett function ($F_в(S_в)$).

Numerical solutions were sought in a domain with conditions of impermeability at the boundary (7). This domain was a subdomain of symmetry singled out from the unbounded oil field, which is simulated. The numerical algorithm was based on completely implicit methods of solving equations (1)-(3), namely, equations (1)-(2) were solved by the iterative secant method, meanwhile the (α-β)-iterative algorithm [12] was employed to solve equation (3). The (α-β)-elliptic solver was an extension of the sweep method for the many-dimensional case. It did not need any a priori information about problem operators and was enough general-purpose. According to this algorithm the solution sought for was obtained via eight auxiliary functions calculated iteratively. It was profitable to include a relaxation parameter into equations for some of those coefficients in order to reduce the number of (α-β)-iterations.

The standard seven-point ("honeycomb") scheme of oil/water well disposition was simulated. The computational domain was approximated by uniform rectangular grid of 117x143 points. Parallel implementation of the algorithm for running on MPPs was based on computational domain partitioning (data parallelization): the domain was divided into equal subdomains in one direction according to the Y-coordinate, with each subdomain being computed by a separate processor of an executing MPP. That domain distribution was more profitable for reducing the number of message-passing operations in the data parallel (α-β)-algorithm than domain distributions according to the X-coordinate and according to the both coordinates. In each subdomain, system (1)-(3) was solved as follows. At every time level, the water saturation was obtained by solving equations (1)-(2) using pressure values from the previous time level. Then, employing the just found water saturation, new pressure was calculated at the present time level by solving equation (3). After that, this procedure was

repeated at the next time level.

The main difficulty of this parallel algorithm was estimation of the optimal relaxation parameter ω for the (α-β)-solver because this parameter varies while dividing the computational domain into different quantities of equal subdomains. Employing a wrong parameter led to slow convergence or in some cases to non-convergence of (α-β)-iterations. Numerous experiments allowed to find the optimal relaxation parameter for each number of subdomains.

The above parallel algorithm was implemented in Fortran 77 with PVM as a communication platform and demonstrated good scalability, speedups and parallelization efficiency while running on the Parsytec PowerXplorer System - an MPP consisting of PowerPC 601 processors as computational nodes and T800 transputers as communicational nodes (one T800 provides four bi-directional 20 Mbits/sec communication links).

Table 1: Performance of the Fortran/PVM parallel oil extraction simulator on the Parsytec PowerXplorer System

Number of processors	ω	Number of iterations	Time (sec)	Real 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%

Table 1 presents some results of the experiments for one time level. Parallelization efficiency is defined as (S_{real}/S_{ideal})x100%, where S_{real} is the real speedup achieved by the

parallel oil extraction simulator on the parallel system, and S_{ideal} is the ideal speedup that could be achieved while parallelizing the problem. The latter is calculated as the sum of performances of processors, constituting the executing parallel system, divided by the performance of a base processor. All speedups are calculated relative to the original sequential oil extraction simulator running on the base processor. Note, that the more powerful are communication links and the less powerful are processors, the higher efficiency is achieved.

Generally speaking, the parallel oil extraction simulator was intended to be a part of a portable software system able to run on local networks of heterogeneous computers and providing a computerized working place of an oil extraction expert. Therefore, a portable application efficiently solving the oil extraction problem on networks of computers was required.

Table 2: Relative performances of workstations

Workstation 's number	1	2	3 - 4	5 - 7	8 - 9
Performance	1150	575	460	325	170

As the first step toward such an application, the above Fortran/PVM program was ported to a local network of heterogeneous workstations based on 10 Mbits Ethernet. Our weakest workstation (SPARCclassic) executed the serial application a little bit slower than PowerPC 601, while the most powerful one (UltraSPARC-1) executed it more than six times faster. In general, 9 uniprocessor workstations were used, and Table 2 shows their relative performances.

Table 3: Performance of the Fortran/PVM parallel oil extraction simulator on subnetworks of workstations.

Subnetwork (workstation's numbers)	ω	Number of iterations	Time (sec)	Ideal speedup	Real 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
$\{2,3,5-$	1.21485	216	47	3.8	0.87	0.23
$\{1-8\}$	1.22175	226	46	3.3	0.41	0.12

Table 3 shows some results of execution of the Fortran/PVM program on different two, four, six and eight-workstation subnetworks of the network. In this table, for every subnetwork the speedup is calculated relative to the running time of the serial program on the fastest workstation of the subnetwork. (The total running time of the serial oil extraction simulator while executing on different workstations can be found in Table 4.) Visible degradation of parallelization efficiency is explained by the following three reasons - slower communication links, faster processors, and not balanced workload of the workstations.

Table 4: Performance of the serial oil extraction simulator.

Processor type	Ultra SPARC-1	SPARC 20	SPARC station 4		SPARC 5			SPARC classic	
Workstation's number	1	2	3	4	5	6	7	8	9
Number of iterations	205								
Time (sec)	18.7	40.7	51.2	51.2	71.4	71.4	71.4	133	133

The first two reasons are unavoidable, while the latter is avoided by means of slight modification of the parallel algorithm implemented by the Fortran/PVM program. Namely, to provide the optimal load balancing, the computational domain is decomposed into subdomains of non-equal sizes, proportional to relative performances of participating processors. To be exact the number of grid columns in each subdomain is the same, while the number of rows differs. Regarding the relaxation parameter, it is reasonable to assume its optimal value to be a function of the number of grid rows and to use for each subdomain its own relaxation parameter $? = ?(N_{row})$. While distributing the domain into different quantities of subdomains with equal numbers of grid rows a sequence of optimal relaxation parameters was found empirically. Now using the experimental data and piecewise linear interpolation, the optimal parameter $?$ can be calculated for any N_{row}. Note, that this approach gave high convergence rate and good efficiency of the parallel (a-$ß$)-solver with relaxation (see Numbers of iterations in Table 5).

Table 5: **Performance of the mpC parallel oil extraction simulator on subnetworks of workstations.**

Subnetwork (workstation's numbers)	Number of iterations	Time (sec)	Real speedup	Efficiency	Time of 205 iterations	Speedup on 205 iterations	Efficiency on 205 iterations
$\{2,5\}$	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 forcedly divided into equal subdomains							

Table 5 shows experimental results of execution of the mpC application on the network of workstations. In the experiments, the mpC programming environment used LAM MPI 6.0 as a communication platform. One can see that the mpC application demonstrates much higher efficiency in the heterogeneous environment than its PVM counterpart. To estimate pure contribution of the load balancing in the improvement of parallelization efficiency, we run the

mpC application on subnetwork {2, 3, 5, 6, 7, 8} with the forcedly even domain decomposition resulting in essential (more than 1.5 times) efficiency degradation (compare rows 5 and 6 in Table 5).

The mpC application only dealt with processor performances and distributed data taking into account this aspect of heterogeneity. It is easy to see that often the best data distribution essentially depends on speeds of communication links. Indeed, in case of slow links communication latencies can exceed parallelization speedup, slowing down solving the problem. Therefore, it can make sense to use not all available processors but only some part of them. In other words, the corresponding heterogeneous data distribution does not distribute data to some processors, and the application does not use those processors for computations. Currently, such an mpC application, simulating oil extraction, that takes into account not only processor performances, but also speeds of communication links is under testing.

5. Related work

All programming environments for parallel programming networks, which we know of, have one common property. Namely, when developing a parallel program, either the programmer has no facilities to describe the virtual parallel system executing the program, or such facilities are too poor to specify an efficient distribution of computations and communications over the target network. Even topological facilities of MPI (including MPI-2 [13]) have turned out insufficient to solve the problem. So, to ensure the efficient execution of the program on a particular network, the user must use facilities external to the program, such as boot schemes and application schemes [14]. If the user is familiar with both the topology (that is, the structure and processor/link performances) of the target network and the topology (that is, the parallel structure) of the application, then, by means of use of such configuration files, he or she can map the processes, which constitute the program, onto processors of the network, to provide

the most efficient execution of the program. Some tools that support and facilitate such a static mapping have appeared [15]. But if the application topology is defined in run time (that is, if it depends on input data), this approach will not work.

There are a few tools [16], [17] performing some functions of a distributed operating system and trying to take into consideration the heterogeneity of processor performances in commodity networks of computers when scheduling tasks in order to maximize the throughput of the corresponding network. Unlike such tools, mpC is aimed at minimization of the running time of a separate parallel application on the executing network. The feature is the most important for end-users, while the network throughput is important for network administrators.

A number of scheduling techniques [18]-[20] were developed to take into account the dynamic performance variability of resources on multi-user clusters and improve the execution performance of data-parallel applications. In particular, the stochastic scheduling approach [20] represents performance characteristics by stochastic values and modifies a time-balancing data allocation policy to use the stochastic information for determining the data allocation.

The mpC programming system is mainly aimed at parallel computing on local networks. Therefore, it pays primary attention to balancing processor loads. Scheduling data transfer operations and maintaining fault tolerance are considered by the system as problems of lower priority. This approach is absolutely unacceptable for software tools aimed at high performance computing on global networks. Much lower performance of communication links makes a good schedule of data operations often more critical for total execution time than a good balance between processor loads. Much higher probability of unexpected resource failures makes mandatory the feature of robustness to resource failure.

A few approaches to high performance computing on global networks have been proposed. NetSolve [21] is a system to support high-performance scientific computations on global networks. NetSolve offers the ability to look for computational resources on a network, choose

the best one available, solve the problem, and returns the solution to the user. A load-balancing policy is used to ensure good performance. Fault-tolerance is maintained by a retry mechanism in the event of a failure. The Computational Grid [22] is a platform for the implementation of high-performance applications using widely dispersed computational resources, and EveryWare [23] is a toolkit for the development of Grid applications.

The core part of any software system for high performance computing on global networks is a tool for monitoring network performance of a global computing network such as Network Weather Service (NWS) [24], Gloperf [25] or NIMI [26]. In particular, NWS takes periodic measurements of deliverable resource performance and dynamically generates forecasts of future performance levels. It is robust to resource failure and dynamically reconfigurable to accommodate changes in the underlying global computing network. Schedulers can use information from such monitoring tools. In particular, Application Level Scheduler (AppLeS) [27] uses information from the NWS to schedule distributed, resource-intensive applications. Each application has its own AppLeS that determines a performance-efficient schedule and implements that schedule with respect to the appropriate resource management systems. AppLeS is not a resource management system but an application-management system that manages the scheduling of the application for the benefit of the end-user.

6. Conclusion

The paper has presented the mpC language and its supportive programming environment aimed at efficiently-portable modular parallel programming networks of computers - the most general parallel architecture. The most important features of mpC are the following:

- once developed, an mpC application will run efficiently on any heterogeneous network of computers without any changes of its source code (we call the property *efficient portability*);

- it allows to write applications adapting not only to nominal performances of processors

but also to redistribute computations and communications dependent on dynamic changes of workload of separate computers of the executing network (the corresponding language constructs and their implementation are presented in [28]).

We continue working on the mpC system. The work is aimed both at providing more efficiency for a wide range of networks of computers (including clusters of supercomputers and global networks) and at facilitation of the parallel programming model.

References

[1] H.El-Rewini, and T.Lewis, Introduction To Distributed Computing, Manning Publications Co., 1997.

[2] Message Passing Interface Forum, MPI: A Message-passing Interface Standard, ver. 1.1, June 1995.

[3] A.Geist, A.Beguelin, J.Dongarra, W.Jlang, R.Manchek, V.Sunderam, PVM: Parallel Virtual Machine, Users' Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge, MA, 1994.

[4] High Performance Fortran Forum, High Performance Fortran Language Specification, version 1.1, Rice University, Houston TX, November 10, 1994.

[5] C.Koelbel, "Conferences for Scientific Applications", IEEE Computational Science & Engineering, 5(3), pp.91-95, 1998.

[6] S.S.Gaissaryan, and A.L.Lastovetsky, "An ANSI C Superset for Vector and Superscalar Computers and Its Retargetable Compiler", The Journal of C Language Translation, 5(3), March 1994, pp.183-198.

[7] Thinking Machines Corporation, "The C* Programming Language", CM-5 Technical

Summary, pp. 69-75, November 1992.

[8] P. J. Hatcher, and M. J. Quinn, Data-Parallel Programming on MIMD Computers, The MIT Press, Cambridge, MA, 1991.

[9] D.Arapov, A.Kalinov, A.Lastovetsky, and I.Ledovskih, "Experiments with mpC: Efficient Solving Regular Problems on Heterogeneous Networks of Computers via Irregularization", Proceedings of the Fifth International Symposium on Solving Irregularly Structured Problems in Parallel (IRREGULAR'98), LNCS 1457, Berkley, CA, USA, August 9-11, 1998, pp.332-343.

[10] B.Chetverushkin, N.Churbanova, A.Lastovetsky, and M.Trapeznikova, "Parallel Simulation of Oil Extraction on Heterogeneous Networks of Computers", Proceedings of the 1998 Conference on Simulation Methods and Applications (CSMA'98), the Society for Computer Simulation, November 1-3, 1998, Orlando, Florida, USA, pp. 53-59.

[11] A.Kalinov, and A.Lastovetsky, "Heterogeneous Distribution of Computations While Solving Linear Algebra Problems on Networks of Heterogeneous Computers", Proceedings of the 7'th International Conference on High Performance Computing and Networking Europe (HPCN Europe'99), LNCS 1593, Springer-Verlag, April 12-14, 1999, Amsterdam, the Netherlands, pp.191-200.

[12] B. N. Chetverushkin, N. G. Churbanova, and M. A. Trapeznikova, "Simulation of oil production on parallel computing systems", Proceedings of Simulation MultiConference HPC'97: Grand Challenges in Computer Simulation, Ed. A.Tentner, Atlanta, USA, April 6-10, 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 State University, 1997.

[15] F.Heinze, L.Schaefers, C.Scheidler, and W.Obeloeer, "Trapper: Eliminating performance

bottlenecks in a parallel embedded application", IEEE Concurrency, 5(3), pp.28-37, 1997.

[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/.

[18] M.Zaki, W.Li, and S.Parthasarathy, “Customized Dynamic Load Balancing for Network of Workstations”, Proceedings of HPDC’96, 1996.

[19] J.Weissman and X.Zhao, “Scheduling Parallel Applications in Distributed Networks”, Journal of Cluster Computing, 1998.

[20] J.Schopf, and F.Berman, “Stochastic Scheduling”, Proceedings of the Supercomputing’99, 1999.

[21] H.Casanova, and J.Dongarra, “NetSolve: A Network Server for Solving Computational Science Problems”, Proceedings of the Supercomputing’96, 1996.

[22] I.Foster and C.Kesselman, The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers, Inc., 1998.

[23] R.Wolski, J.Brevik, C.Krintz, G.Obertelli, N.Spring, and A.Su, “Running EveryWare on the Computational Grid”, Proceedings of the Supercomputing’99, 1999.

[24] R.Wolski, N.Spring, and C.Peterson, “Implementing a Performance Forecasting System for Metacomputing: The Network Weather Service”, Proceedings of the Supercomputing’97, 1997.

[25] C.Lee, J.Stepanek, R.Wolski, C.Kesselman, and I.Foster, “A Network Performance Tool for Grid Environments”, Proceedings of the Supercomputing’99, 1999.

[26] V.Paxson, J.Mahdavi, A.Adams, and M.Mathis, “An Architecture for Large-Scale Internet Measurement”, IEEE Communications, 36(8), pp.48-54, 1998.

[27] F.Berman, R.Wolski, S.Figueira, J.Schopf, and G.Shao, “Application-Level Scheduling on Distributed Heterogeneous Networks”, Proceedings of the Supercomputing’96, 1996.

[28] D.Arapov, A.Kalinov, A.Lastovetsky, and I.Ledovskih, "A Language Approach to High Performance Computing on Heterogeneous Networks", Parallel and Distributed Computing Practices, 2(3), 2000 (to appear).

Appendix A. Parallel matrix multiplication in mpC.

```
/* 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*/  }
```

Appendix B. Target code for an mpC block in the most general case.

```
{
  declarations
  {
   if(!MPC_Is_busy()) {
    target code executed by free nodes to create regions for
    networks defined in source mpC block
   }
   if(MPC_Is_busy()) {
    target code executed by hired nodes to create regions for
    networks defined in source mpC-block and target code
    for mpC statements before waiting-point break statement
   }
   epilogue of waiting point
  }
  target code for mpC statements starting from waiting-point
```

```
  break statement
  {
   target code executed by hired nodes to deallocate regions for
   networks defined in source mpC block

    label of deallocating waiting point:
    if(!MPC_Is_busy()) {
      target code executed by free nodes to deallocate regions
      for networks defined in source mpC block
    }
    epilogue of waiting point
  }
}
```

Appendix C. Target code for an mpC block with a shared waiting point.

```
{
  declarations
  {
    label of shared waiting point:
    if(!MPC_Is_busy()) {
      target code executed by free nodes to create and deallocate
      regions for networks defined in source mpC block
    }
    if(MPC_Is_busy()) {
      target code executed by hired nodes to create and deallocate
      regions for networks defined in block and target code
      for statements of source mpC block
```

```
    }
    epilogue of waiting point
  }
}
```

Appendix D. Parallel modeling the evolution of galaxies in mpC.

```
#define MaxGs 30 /* Maximum number of groups */
#define MaxBs 600 /* Maximum number of bodies in a group */
typedef double Triplet[3];
typedef struct {Triplet pos; Triplet v; double m;} Body;
int [host]M; /* Number of groups */
int [host]N[MaxGs]; /* Numbers of bodies in groups */
repl dM, dN[MaxGs];
double [host]t; /* Galaxy timer */
Body (*[host]Galaxy[MaxGs])[MaxBs]; /*Bodies of galaxy*/
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; /* Broadcasting the number of groups */
  dN[]=N[];/* Broadcasting numbers of bodies in groups */
```

```
{
    net GalaxyNet(dM,dN) g; /* Creation of network 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++) /* Scattering 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++) /* ???? ????? */
        (*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[];
}
```