ELSEVIER

Parallel Computing 28 (2002) 1369–1407

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Adaptive parallel computing on heterogeneous networks with mpC

Alexey Lastovetsky [*]

Department of Computer Science, University College Dublin (UCD), Belfield, Dublin 4, Ireland

Received 20 January 2001; received in revised form 20 November 2001; accepted 27 June 2002

Abstract

The paper presents a new advanced version of the mpC parallel language. The language was designed specially for programming high-performance parallel computations on heterogeneous networks of computers.

The advanced version allows the programmer to define at runtime all the main features of the underlying parallel algorithm, which have an impact on the application execution performance, namely, the total number of participating parallel processes, the total volume of computations to be performed by each of the processes, the total volume of data to be transferred between each pair of the processes, and how exactly the processes interact during the execution of the algorithm. Such an abstraction of parallel algorithm is called a *network type* in mpC.

Given a network type, the programmer can define a network object of this type and describe in details all the computations and communications to be performed on the network object. The mpC programming system uses the information extracted from the network-type definition together with information about the actual performance of the executing network to map the processes of the parallel program to this network in such a way that leads to its better execution time.

In addition, the programmer can use a special operator, `timeof`, which predicts the total time of the algorithm execution on the underlying hardware without its real execution. That feature allows the programmer to write such a parallel program that can follow different parallel algorithms to solve the same problem, making choice at runtime depending on the particular executing network and its actual performance.

The paper describes both the language model of parallel algorithm and the model of executing parallel environment used by the mpC programming system. It also discusses principles of the implementation of mapping mpC network objects to the computing network.
© 2002 Elsevier Science B.V. All rights reserved.

[*] Fax: +353-1716-7777.
E-mail address: alexey.lastovetsky@ucd.ie (A. Lastovetsky).

Keywords: Parallel programming language; Parallel programming system; Heterogeneous network of computers; Heterogeneous parallel computing

1. Introduction

Heterogeneous networks of computers are the most general and common parallel architecture. In the most general case, a heterogeneous network includes PCs, workstations, multiprocessor servers, clusters of workstations, and even supercomputers. Unlike traditional homogeneous parallel platforms, the heterogeneous parallel architecture uses processors running at different speeds. What is even more important, the processors demonstrate different relative speeds on different code mixtures. Speeds of data transfer between different processors in heterogeneous networks can also differ significantly. Communications between processors of the same shared-memory multiprocessor server will be much faster than communications between processors of different workstations. It makes programming heterogeneous platforms a challenging task. Data, computations, and communications should be distributed unevenly to provide the best execution performance.

To the best of the author's knowledge, there are a relatively small number of papers dealing with design and implementation of parallel algorithms on heterogeneous platforms. Actually, this area of parallel computing is only taking its first steps. The research already conducted reveals the intrinsic difficulty of designing heterogeneous algorithms. Even such a simple linear algebra kernel as matrix multiplication turns out to be surprisingly difficult on heterogeneous platforms [1].

The heterogeneous algorithms are normally implemented using the MPI library [2]. Code responsible for uneven distribution of data, computations and communications is a substantial part of the corresponding MPI applications, and, probably, their most important and complex part. MPI is a well-designed and powerful tool for programming distributed-memory architectures. The only disadvantage of MPI is the low level of its parallel primitives. It is tedious and error-prone to write really complex and reliable MPI applications, like those implementing heterogeneous algorithms. In fact, MPI is a tool of the assembler level for programming distributed-memory architectures.

The paper deals with a high-level programming language, mpC, designed to facilitate implementation of heterogeneous algorithms. Once a heterogeneous algorithm has been designed, and the corresponding performance analysis has been carried out, it can be explicitly specified in a high-level form as a part of the mpC application. That specification is used to generate some algorithm-specific code, which, in concert with the mpC run-time library, provides support for the uneven distribution of data, computations, and communications dictated by the heterogeneous algorithm.

The mpC language [3–5] was designed in 1994. Its first programming system was implemented in 1996 and made free available via Internet in October 1996 at the address http://www.ispras.ru/~mpc. Now version 2.2.0 of the system is available providing improved functionality and more reliable and portable implementation. Nonetheless, the underlying programming and architectural models remain quite simplified through all the versions. Namely, the primary attention is paid to balancing processor loads; meanwhile data transfer operations were practically neglected. At the same time, the slower are communication links, the more critical are both the good schedule of data transfer operations and the good balance between computations and communications for the total execution time.

The paper presents a new advanced version of the mpC parallel language allowing the programmer to define at runtime all the main features of parallel algorithm, which have an impact on the execution performance of the application on heterogeneous platforms, including:

- the total number of participating parallel processes,
- the total volume of computations to be performed by each of the processes,
- the total volume of data to be transferred between each pair of the processes, and
- how exactly the processes interact during the execution of the algorithm.

Such an abstraction of parallel algorithm is called a *network type*. Given a network type, the programmer can define a network object of this type and describe in details all the computations and communications to be performed on the network object.

The mpC programming system uses the information extracted from the definition of network type together with information about actual performances of processors and communication links of the executing network to map the processes of the parallel program to this network in such a way that achieves its better execution time.

The programmer can also use a special operator, `timeof`, which predicts the total time of the algorithm execution on the underlying hardware without its real execution. That feature allows the programmer to write such parallel programs that follow different parallel algorithms to solve the same problem, making choice at runtime depending on the particular executing network and its actual performance.

The paper describes both the language model of parallel algorithm and the model of the executing parallel environment used by the mpC programming system. It also discusses principles of implementation of the mapping of mpC network objects to the computing network.

The rest of the paper is organized as follows. Section 2 introduces the language model of parallel algorithm. Section 3 introduces the model of the executing parallel environment. Section 4 outlines principles of implementation of the mapping of mpC network objects to the computing network. Section 5 presents some experimental results. Section 6 surveys related work. Section 7 concludes the paper and outlines future research work.

2. Abstraction of parallel algorithm in the mpC language

2.1. Basic model of heterogeneous parallel algorithm

The mpC language is an ANSI C superset designed specially for programming parallel computations on common networks of diverse computers. The main goal of parallel computing is to speed up solving problems on available computer resources. Just this differs parallel computing from distributed computing, the main goal of which is to make different software components, inherently located on different computers, work together. In the case of parallel computing, partition of the whole program into a number of distributed components located on different computers is just a way to speed up execution of the program not its inherent property. Therefore, when designing the mpC language, the primary attention was paid to the means that facilitate development of high efficient and portable programs solving single problems on common networks of computers.

A parallel program running on the network of computers is a set of processes interacting (that is, synchronizing their work and transferring data) by means of message passing. Source mpC code does not specify how many processes constitute the parallel program as well as which computer runs one or another process. This is done by some means external to the language when the program is started up. Source mpC code only describes which computations are performed by each of the processes constituting the program.

A group of processes executing together some parallel computations to solve a logical unit of the entire problem reflects in the mpC language in the notion of *network*. In mpC, network is an abstract mechanism to facilitate managing actual physical processes of the parallel program (just like the notions of *data object* and *variable* facilitate memory management in programming languages).

In the simplest case, a network is just a set of *virtual processors*. To code computations on a given number of parallel processes, the mpC programmer first defines a network consisting of this number of virtual processors and then describes parallel computations on this network.

The network definition causes creation of a group of processes representing the network, so that each virtual processor will be represented by a separate process. Description of computations on the network will cause execution of the corresponding computations by the processes that represent virtual processors of the network.

An important difference of real processes from virtual processors is that at different moments of program execution the same process can represent different virtual processors of different networks. In other words, the definition of the network causes mapping of its virtual processors to actual processes of the parallel program, and such a mapping does not change during lifetime of the network.

The following simple mpC program

```
#include <mpc.h>
#define N 3
int [*]main() {
  net SimpleNet(N) mynet;
  [mynet]MPC_Printf("Hello, world!\n");
}
```

defines the network **mynet** of **N** virtual processors and then calls the library function **MPC_Printf** on the network.

Execution of the program consists in parallel call of the function **MPC_Printf** by those **N** processes of the program onto which virtual processors of the network **mynet** are mapped. This mapping is performed by the mpC programming system at runtime. Execution of the function **MPC_Printf** by a process consists in sending the message "Hello, world!" to the user's terminal from which the entire parallel program has been started up. So, the user will see **N** messages "Hello, world!" on this terminal—just one from each involved process.

The **[*]** specifier before the name **main** in the definition of the main function says that the code of the function shall be executed by all processes of the parallel program. Functions similar to the function **main** are called *basic* functions in mpC. Correct work of a basic function is possible only if all processes of the parallel program call it. The mpC compiler controls correctness of basic function calls.

Unlike the function **main**, the function **MPC_Printf** does not need to be called in parallel by all processes of the parallel program in order to work correctly. Moreover, a call to the function in any single process of the parallel program makes sense and is correct. Such functions are called *nodal* in mpC. The mpC language allows any single process of the parallel program to call a nodal function as well as any group of processes to call the function in parallel.

The following program

```
#include <mpc.h>
#include <sys/utsname.h>
#define N 3
int [*]main() {
  net SimpleNet(N) mynet;
  struct utsname [mynet]un;
  [mynet]uname(&un);
  [mynet]MPC_Printf("Hello world! I'm on\"%s\".\n",
                    un.nodename);
}
```

also outputs messages from those processes of the parallel program to which the virtual processors of the network **mynet** are mapped. But in addition to "Hello, world!", each involved process outputs the name of the computer, which executes the process.

To achieve it, the program defines the variable **un** *distributed over network* **mynet**. Only a process implementing one of the virtual processors of **mynet** holds in its memory a copy of **un**. Only those processes call the function **uname** (what is specified with the construct [**mynet**] placed before the function name). After this call the member **nodename** of each projection of the distributed structure **un** will contain a pointer to the name of the computer running the corresponding process.

Lifetime of both the network **mynet** and the variable **un** is limited by the block in which they are defined. When execution of the block ends, all processes of the program that have been taken for virtual processors of the network **mynet** are freed and can be used for other networks. Such mpC networks are called *automatic*. Lifetime of *static* networks is only limited by the time of program execution.

The following two programs demonstrate the difference between static and automatic networks. The programs look almost identical. Both consist in cyclic execution of a block defining a network and executing already familiar computations on the network. The only but essential difference is that the first program defines an automatic network meanwhile the second one defines a static network.

During execution of the program

```
#include <mpc.h>
#define Nmin 3
#define Nmax 5
int [*]main() {
  repl n;
  for(n = Nmin; n <= Nmax; n++) {
    auto net SimpleNet(n) anet;
    [anet]MPC_Printf("I'm from an automatic network of %d.\n",
                      [anet]n);
  }
}
```

at the first loop iteration (**n** = **Nmin** = 3) a network of three virtual processors is created on the entry into the block, and this network is destructed when execution of the block ends. At the second loop iteration (**n** = 4) a new network of four virtual processors is created on the entry into the block, and that network is also destructed when execution of the block ends. So at the moment of repeated initialisation of the loop (execution of the expression **n**++), the 4-processor network no longer exists. Finally, at the last iteration an automatic network of five virtual processors (**n** = **Nmax** = 5) is created on the entry into the block.

Note, that the integer variable **n** is distributed over all processes constituting the parallel program. Its definition contains keyword **repl** (a shortcut of *replicated*), which informs the compiler that all projections of the value of the variable shall be equal to each other in any expression in the program. Such distributed variables are called *replicated* in mpC (correspondingly, the value of a replicated variable is called a replicated value). Replicated variables and expressions play an important role in mpC. The mpC compiler checks the property *to be replicated* declared by the programmer and warns about all possible its violations.

During execution of the program

```
#include <mpc.h>
#define Nmin 3
#define Nmax 5
int [*]main() {
  repl n;
  for(n=Nmin; n<=Nmax; n++) {
    static net SimpleNet(n) snet;
    [snet]MPC_Printf("I'm from a static network of %d.\n",
                     [snet]n);
  }
}
```

at the first loop iteration a network of three virtual processors is also created on the entry into the block, but this network is not destructed when execution of the block ends. It simply becomes invisible. Thus in this case the block is not a region where the network exists but a region of its visibility. Therefore, at the time of repeated initialisation of the loop and evaluation of the loop condition the static 3-processor network is existing but not available (because these points of the program are out of scope of the network name **snet**). On next entries into the block at subsequent loop iterations no new network is created, but the static network, which has been created on the first entry into the block, becomes visible.

Thus, meanwhile the name **anet** denotes absolutely different networks at different loop iterations; the name **snet** denotes a unique network existing from the first entry in the block, in which it is defined, until the end of program execution.

Generally speaking, in mpC one cannot simply define a network but only a network of some type. *Type* is the most important attribute of network. In particular, it determines how to access separate virtual processors of the network.

The type specification is a mandatory part of any network definition. Therefore, any network definition should be preceded by the definition of the corresponding network type.

In above programs the definition of the network type **SimpleNet** can be found among other standard definitions of the mpC language in the header file **mpc.h** and is included in these programs with the **#include** directive. The definition looks as follows:

```
nettype SimpleNet(int n) {
  coord I=n;
};
```

It introduces the name **SimpleNet** of the network type parameterised with the integer parameter **n**. The body of the definition declares the *coordinate variable* **I** ranging from **0** to **n − 1**. The type **SimpleNet** is the simplest parameterised network type that describes networks consisting of **n** virtual processors well ordered by their positions on the coordinate line.

The following program

```
#include <mpc.h>
#define N 5
```

```
int [*]main() {
  net SimpleNet(N) mynet;
  int [mynet]my_coordinate;
  my_coordinate = I coordof mynet;
  if(my_coordinate%2==0)
    [mynet]MPC_Printf("Hello, even world!\n");
  else
    [mynet]MPC_Printf("Hello, odd world!\n");
}
```

demonstrates how execution of differing computations by different virtual processors can be coded.

The program uses the binary operator **coordof** with the coordinate variable **I** and the network **mynet** as its left and right operands correspondingly. The result is an integer value distributed over the network **mynet**, whose projection to a virtual processor will be equal the value of the coordinate **I** of this virtual processor in that network. After execution of the assignment **my_coordinate = I coordof mynet**, each projection of the variable **my_coordinate** will hold the coordinate of the corresponding virtual processor of the network **mynet**. As a result, virtual processors with even coordinates will output "Hello, even world!", meanwhile ones with odd coordinates will output "Hello, odd world!".

We have discussed that lifetime of an automatic network is limited by the block in which the network is defined. When execution of the block ends, the network ceases to exist, and all processes taken for virtual processors of the network are freed and can be used for other networks. The question is how results of computations on automatic networks can be saved and used in further computations. Our previous programs did not raise the problem, because the only result of parallel computations on networks was output of some messages to the user's terminal.

Actually, in mpC, networks are not absolutely independent on each other. Every newly created network has exactly one virtual processor shared with already existing networks. That virtual processor is called a *parent* of this newly created network and is the connecting link, through which results of computations are passed if the network ceases to exist. The parent of a network is always specified by the definition of the network, explicitly or implicitly.

So far, no network was defined with explicit specification of its parent. The parent was specified implicitly, and the parent was the so-called *virtual host-processor*. At any moment of program execution the existence of only one network can be guaranteed, namely, the pre-defined network **host** consisting of exactly one virtual processor, which always maps onto the host-process associated with the user's terminal.

The program

```
#include <mpc.h>
nettype AnotherSimpleNet(int n) {
  coord I = n;
  parent [0];
};
#define N 3
int [*]main() {
  net AnotherSimpleNet(N) [host] mynet;
  [mynet]MPC_Printf("Hello, world!\n");
}
```

is completely equivalent to the very first program except that in the definition of the network the explicit specification of the parent substitutes the implicit one.

One more difference can be found in the definition of the network type. A line explicitly specifying the coordinate of the parent in networks of the type (the coordinate defaults to zero) is added. Should for some reason we needed that the parent of the network **mynet** had not the least but the greatest coordinates, then in the definition of the network type **AnotherSimpleNet** the specification **parent** [**n** − **1**] had to be used instead of **parent** [0].

The library function **MPC_Barrier** allows synchronising the work of the virtual processors of any network. For example, the following program

```
#include <mpc.h>
#define N 5
int [*]main() {
  net SimpleNet(N) mynet;
  [mynet]:{
    int my_coordinate;
    my_coordinate = I coordof mynet;
    if(my_coordinate%2 = = 0)
     MPC_Printf("Hello, even world!\n");
    ([(N)mynet])MPC_Barrier();
    if(my_coordinate%2 = = 1)
     MPC_Printf("Hello, odd world!\n");
  }
}
```

makes all messages from virtual processors with odd coordinates come to the user's terminal only after messages from virtual processors with even coordinates. Note, that in this program only the processes implementing the network **mynet** participate in the barrier synchronization.

The call of the function **MPC_Barrier** looks a little bit unusual. Indeed, this function principally differs from all functions we have met and represents so-called *network* functions. Unlike basic functions, which are always executed by all processes of the parallel program, network functions are executed on networks and hence can be executed in parallel with other network or nodal functions. Unlike nodal functions, which can also be executed in parallel by all processes of one or another network, virtual processors of the network executing a network function can transfer data, and this makes them a bit similar to basic functions.

The declaration of the function **MPC_Barrier** is in the header file **mpc.h** and looks as follows:

```
int [net SimpleNet(n) w] MPC_Barrier(void);
```

Any network function has a special *network formal parameter*, which represents the network executing the function. In the declaration of the network function, a specification of that parameter is in brackets just before the name of the function and looks like normal network definition. In the case of the function **MPC_Barrier**, the specification of the network parameter looks as follows:

```
net SimpleNet(n) w
```

In addition to the formal network **w** executing the function **MPC_Barrier**, this declaration introduces the parameter **n** of this network. Like regular formal parameters, this parameter is available in the body of the function as if it was declared with specifiers **repl** and **const**. Since in accordance with the definition of the network type **SimpleNet** the parameter **n** is of the type **int**, one can say that the parameter **n** is treated in the body of the function **MPC_Barrier** as if it were a regular formal parameter declared as follows:

```
repl const int n
```

All regular formal parameters are considered distributed over the formal network parameter. Thus the replicated over the network **w** integer constant parameter **n** determines the number of virtual processors of the network. If the function **MPC_Barrier** were not a library one, it could be defined as follows:

```
int [net SimpleNet(n) w] MPC_Barrier(void) {
  int [w:parent] bs[n], [w]b=1;
  bs[]=b;
  b=bs[];
}
```

In the body of this function the automatic array **bs** of **n** elements (the mpC language supports *dynamic* arrays) is defined. This array is located on the parent of the network **w** that is specified with the construct **[w:parent]** before the name of the array in its definition. In addition, the variable **b** distributed over the network **w** is also defined there. A couple of statements following the definition implement a barrier for virtual processors of the network **w**.

In the above program, the call to the network function **MPC_Barrier** passes the actual network parameter **mynet** as well as the actual value of the only parameter of the network type **SimpleNet**. At the first glance, the latter looks redundant. But it should be taken into account that networks of any type, not only the **SimpleNet** type, can be passed to this function as an actual network parameter. Actually, the function **MPC_Barrier** only *treats* the group of processes, on which it is called, as a network of the type **SimpleNet**. In general, the actual network parameter may be of any type that allows its correct interpretation for various values of the parameters of the network type used in the definition of the called network function. Therefore, the values of the parameters should be explicitly determined in the function call.

So far either all processes of the parallel program or all virtual processors of some network took part in data transfer, and the data transfer itself mainly consisted in either broadcasting some value to all participating processes or gathering values from all participating processes on one of them.

The basic means of the mpC language for describing complicated data transfer are *subnetworks*. Any subset of the virtual processors of the network can make up a subnetwork of this network. In the following program

```
#include <string.h>
#include <mpc.h>
#include <sys/utsname.h>

nettype Mesh(int m, int n) {
  coord I=m, J=n;
  parent [0,0];
};
#define MAXLEN 256
int [*]main() {
  net Mesh(2,3) [host]mynet;
  [mynet]:{
    struct utsname un;
    char me[MAXLEN], neighbour[MAXLEN];
    subnet [mynet: I==0]row0, [mynet: I==1]row1;
    uname(&un);
    strcpy(me, un.nodename);
    [row0]neighbour[] = [row1]me[];
    [row1]neighbour[] = [row0]me[];
    MPC_Printf("I'm (%d, %d) from \"%s\"\n"
             "My neighbour (%d, %d) is on \"%s\".\n\n",
             I coordof mynet, J coordof mynet, me
             (I coordof mynet+1 )%2, J coordof mynet,
             neighbour);
  }
}
```

each virtual processor of the network **mynet** of type **Mesh(2,3)** outputs to the user's terminal not only the name of the computer hosting this virtual processor but also the name of the computer hosting the closest virtual processor from the neighbouring row. To do it, the program defines two subnetworks **row0** and **row1** of the network **mynet**. The subnetwork **row0** consists of all virtual processors of the network **mynet** whose coordinate **I** is equal to zero, that is, corresponds to the zero row of the network **mynet**. This fact is specified with the construct [**mynet:I** = = **0**] before the name of the subnetwork in its definition.

Similarly, the subnetwork **row1** corresponds to the first row of the network **mynet**. In general, logical expressions describing virtual processors of subnetworks can be quite complex and allow specifying very sophisticated subnetworks. For example, the expression **I** < **J** && **J**%**2** == **0** specifies the virtual processors of the network over the main diagonal in even columns.

Execution of assignment [**row0**]**neighbour**[] = [**row1**]**me**[] consists in parallel transferring the corresponding projection of the distributed array **me** from each **j**-th virtual processor of the row **row1** to the each **j**-th virtual processor of the row **row0** followed by its assignment to the corresponding projection of the array **neighbour**.

Similarly, execution of the assignment [**row1**]**neighbour**[] = [**row0**]**me**[] consists in parallel transferring the content of the corresponding projection of the distributed array **me** from each **j**-th virtual processor of the row **row0** to the each **j**-th virtual processor of the row **row1** followed by its assignment to the corresponding projection of the array **neighbour**.

As a result, a projection of the distributed array **neighbour** on the virtual processor **(0,j)** contains the name of the computer hosting the virtual processor **(1,j)**, and a projection of this array on the virtual processor **(1,j)** contains the name of the computer hosting the virtual processor **(0,j)**.

The row subnetworks might be defined implicitly, i.e., the **subnet** definition might be omitted, and the assignments might look as follows:

```
[mynet: I= =0]neighbour[] = [mynet: I= =1]me[];
[mynet: I= =1]neighbour[] = [mynet: I= =0]me[];
```

In this particular case, the usage of implicitly defined subnetworks is justified because it simplifies the program code without loss of program efficiency or functionality. But there exist situations when you cannot avoid explicit definition of subnetworks. For example, network functions cannot be called on implicitly defined subnetworks.

We have discussed that definition of a network causes mapping virtual processors of the network to actual processes of the parallel program, and this mapping is constant during the lifetime of this network. But we have not discussed how the programming system performs that mapping and how the programmer can manage it.

We have emphasized that the main goal of parallel computing is to speed up solving problems. Therefore it is natural that minimization of the running time of the parallel program is the main target while mapping virtual processors of the network to actual processes. While performing the mapping, the programming system bases, on the one hand, on information about configuration and performance of components of the parallel computer system executing the program, and on the other hand, on information about relative volumes of computations that will be performed by different virtual processors of the defined network.

We have not specified volumes of computations in our programs yet. Therefore, the programming system considered all virtual processors of the network to perform the same volumes of computations. Proceeding from this assumption, it tried to map virtual processors to keep the total number of virtual processors mapped to an actual processor approximately proportional to its performance (naturally taking into account the maximum number of virtual processors that could be hosted by one or another real processor). Such mapping ensures all processes representing virtual processors of the network to execute computations approximately at the same speed. Therefore, if volumes of computations performed by different virtual processors between points of synchronisation or data transfer are approximately the same, the parallel program as a whole will be balanced in the sense, that the processes will not wait for each other at those points of the program.

Such mapping appeared acceptable in all our programs, because, indeed, computations performed by different processors of the network were approximately the same and, in addition, of a very small volume. But in case of essential differences in volumes of computations performed by different virtual processors it can lead to very low speed of program execution. The reason is that in that case execution of computations by different processes at the same speed leads to the situation when processes performing smaller volumes of computation will wait at synchronisation points and points of data transfer for processes performing computations of the bigger volume. In this case, the mapping that ensures speeds of processes to be proportional to volumes of performed computations would lead to a more balanced and faster parallel program.

The mpC language provides means for specification of relative volumes of computations performed by different virtual processors of one or another network. The mpC programming system uses this information to map virtual processors of the network to processes of the parallel program in such a way that ensures each virtual processor to perform computations at the speed proportional to the volume of the computations.

The following program

```
...
typedef struct {double length;
               double width;
               double height;
               double mass;} rail;

nettype HeteroNet(int n, double v[n]) {
  coord I=n;
  node {I>= 0: v[I];};
```

```
    parent[0];
  };

  double MassOfRail(double l, double w, double h, double delta)
  {
    double m, x, y, z;
    for(m=0., x=0.; x<l; x+=delta)
      for(y=0.; y<w; y+=delta)
       for(z=0.; z<h; z+=delta)
        m+=Density(x,y,z);
    return m*delta*delta*delta;
  }

  int [*] main(int [host]argc, char **[host]argv) {
    repl N=[host]atoi(argv[1]);
    static rail [host]s[[host]N];
    repl double volumes[N];
    int [host] i;
    repl j;

    [host]InitializeSteelHedgehog(s, [host]N);
    for(j=0; j<N; j++)
      volumes[j]=s[j].length*s[j].width*s[j].height;

    recon MassOfRail(0.2, 0.04, 0.05, 0.005);
    {
      net HeteroNet(N, volumes) mynet;
      [mynet]:
      {
       rail r;
       r = s[];
       r.mass = MassOfRail(r.length, r.width, r.height, DELTA);
       [host]printf("The total weight is %g kg\n",
                        [host]((r.mass)[+]));
      }
    }
  }
```

introduces those means.

The program defines the network type **HeteroNet** parameterised with two parameters. The integer scalar parameter **n** determines the number of virtual processors of the corresponding network. The vector parameter **v** consists of **n** elements of the type **double** and is used just for specification of relative volumes of computations performed by different virtual processors.

The definition of the network type **HeteroNet** contains the declaration **node**$\{$**I** $>=$ **0** : **v**[**I**]$\}$ saying that for any **I** $>=$ **0** the relative volume of computations performed by the virtual processor with coordinate **I** is equal to **v**[**I**].

This program calculates the mass of a metallic construction welded from **N** heterogeneous rails. For parallel computation of the total mass of the metallic "hedgehog", it defines the network **mynet** consisting of **N** virtual processors each calculating the mass of one of those rails. The calculation is performed by numerical 3-dimensional integration of the density function **Density** with a constant integration step.

Obviously, the volume of computations to calculate the mass of a rail is proportional to the volume of the rail. Therefore, the replicated array **volumes**, the **i**-th element of which just contains the volume of the **i**-th rail, is used as the second actual parameter of the network type **HeteroNet** in the definition of the network **mynet**. Thus, the program specifies that the volume of computations performed by the **i**-th virtual processor of the network **mynet** is proportional to the volume of the rail, the mass of which the virtual processor computes.

The mapping of virtual processors to computers is based on information about the performance of the computers. The relative performance of computers, that is, the relative speed of executing computations, very substantially depends on which exactly computations are executed. Often, a computer showing the best performance when executing one program appears the slowest when executing another program. This is clearly seen when one analyses the published results of measurement of the performance of different computers using a pretty wide range of special testing program packages.

By default, the mpC programming system uses the estimation of performances once obtained as a result of execution of a special parallel program during initialization of the system on the particular network of computers. That estimation is very rough and can differ significantly from the real performance demonstrated by the computers while executing code substantially differing from the code of this special test program. Therefore, the mpC language provides a special statement, **recon**, which allows the programmer to change the default performance estimation by tuning it to the computations, which will be really executed.

In the above program this statement is executed right before definition of the network **mynet**. Execution of the statement is that all physical processors running the program execute in parallel the code provided by the statement (in our case it is a call of the function **MassOfRail** with actual parameters **20.0, 4.0, 5.0** and **0.5**), and the time elapsed by each of the real processors to execute the code is used to refresh the estimation of its performance.

The main part of the total volume of computations performed by each virtual processor of the network **mynet** just falls into execution of calls to the function **MassOfRail**. Therefore, while creating the network **mynet**, the programming system bases on the estimation of performances of real processors that is very close to their actual performance shown while executing the program.

It is very important that the **recon** statement allows updating the estimation of processor performances dynamically, at runtime, just before using the estimation by the programming system. It is especially important if computers, executing the mpC program, are used for other computations as well. In that case, the real performance of processors can dynamically change dependent on the external computations. The use of the **recon** statement allows writing parallel programs sensitive to such dynamic variation of the workload of the underlying computer system. In those programs, computations are distributed over real processors in accordance to their actual performances at the moment of execution of the computations.

An interesting issue is the choice of the total number of processes constituting a running mpC application. How many processes should be allocated to each participating computer when the user starts up the application? Obviously, the more processes you have, the better load balancing can be achieved. On the other hand, more processes consume more resources and cause more inter-process communications, which can significantly increase the total overhead. Some basic rules to make choice are the following. First of all, the number of processes running on each individual computer should not be less than the number of processors of this computer just to be able to exploit all available processor resources. As to the upper bound, the number is limited by the underlying operating system and/or the underlying MPI implementation. For example, LAM MPI version 5.2 installed under Solaris 2.3 does not allow more than 15 MPI processes running on an individual workstation. If an mpC application does not define a significant amount of static data, then all processes, which are not selected for virtual processors of some abstract network defined in the application, are very light-weighted and do not consume too much resources such as processor cycles or memory. In this case, the only overhead is additional communications with such processes, which include initialisation of the underlying MPI platform and the mpC specific communications during execution of the application. The latter mainly fall into the creation of network. The time elapsed by this operation does not grow rapidly with the growth of the total number of processes. For example, the use of six processes instead of one process per workstation on a network of nine uniprocessor workstations only caused 30% increase of this time. This is because the operation includes some relatively significant calculations, and the amount of the calculations is more sensitive to the number of computers than to the number of processes running on each of the computers. At the same time, due to their design some applications just do not need more than one process per processor. An example of such an application is the matrix multiplication from the next section.

2.2. Advanced model of parallel algorithm

The abstraction of heterogeneous parallel algorithm, used in the presented above basic version of the mpC language, is simple enough. In fact, the implemented parallel algorithm is characterised by two main attributes—the number of processes to perform the algorithm and the relative volume of computations to be executed by each of the processes. This model does not take into account another two features having essential impact on execution performance of parallel algorithms on heterogeneous networks.

The first one is interprocess communication. In fact, the basic model implicitly assumes that the time of communication is neglectably small compared to the time of computation. At the same time, the lower is the performance of the communication links and the bigger is the volume of transferred data, the further from the truth is that assumption and the more critical are both the good schedule of data transfer operations and the good balance between computations and communications for the total execution time.

The second neglected feature is the order of execution of the computations (and communications) by the involved parallel processes. The basic model implicitly assumes that all computations performed by different processes are executed strictly in parallel. That assumption is satisfactory only for a restricted class of parallel algorithms. In most parallel algorithms, there are data dependencies between computations performed by different processes. On the other hand, many parallel algorithms try to overlap computations and communications. Therefore, often the use of the default simplified structure of the executed parallel algorithm leads to the mapping far away from the optimal one. The most obvious example is an algorithm with completely serialised computations being performed by different processes. The optimal mapping should always assign all the participating processes to the most powerful processor. At the same time, the use of the basic simplified model often leads to the mapping that involves all available processors.

To introduce the new advanced model of parallel algorithm, let us consider two typical mpC applications. The first one simulates the evolution of groups of bodies under the influence of Newtonian gravitational attraction. Since the magnitude of interaction between bodies falls off rapidly with distance, a single equivalent body may approximate the effect of a large group of bodies. This allows paralleling the problem, and the parallel application will use a few parallel processes, each of which will update data characterizing a single group of bodies. Each process 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. The application will implement the following parallel algorithm:

```
Initialisation of galaxy on host-process
Scattering groups of bodies over processes
Parallel computing masses of groups
Sharing the masses among processes
while (1){
  Visualization of galaxy by host-process
  Parallel computing centers of gravity
  Sharing the centers among processes
  Parallel updating groups
  Gathering groups on host-process
}
```

It is assumed that each iteration of the main loop calculates new coordinates of all bodies in some fixed interval of time.

The core of the mpC application, implementing the above algorithm, is the following description of this algorithm describing those features that influence its running time:

```
nettype Galaxy(m, k, n[m]) {
  coord I=m;
  node {I>=0: bench*((n[I]/k)*(n[I]/k)); };
  link {I>0: length*(n[I]* sizeof(Body)) [I]->[0]; };
  parent [0];
  scheme{
    int i;
    par (i=0; i<m; i++) 100%%[i];
    par (i=1; i<m; i++) 100%%[i]->[0];
  };
};
```

Informally, it looks like a description of an abstract network of computers, which executes the algorithm, complemented by a description of the workload of each involved element of this abstract network and a description of the scenario of interaction between these elements during execution of the algorithm. From the mpC language's point of view, that description defines a parameterised type of abstract networks or a network type definition.

The first line of the above network type definition introduces the name **Galaxy** of the network type and a list of parameters—integer scalar parameters **m** and **k** and vector parameter **n** of **m** integers. Next line declares the coordinate system to which abstract processors will be related. It introduces coordinate variable **I** ranging from **0** to $\mathbf{m}-\mathbf{1}$.

Next line associates virtual processors with this coordinate system and describes the (absolute) volume of computations to be performed by each of the processors. As a unit of measurement, the volume of computations performed by some benchmark code is used. In this particular case, it is assumed that the benchmark code computes a single group of **k** bodies. It is also assumed that **i**-th element of vector parameter **n** is just equal to the number of bodies in the group computed by the **i**-th virtual processor. The number of operations to compute one group is proportional to the number of bodies in the group squared. Therefore, the volume of computations to be performed by the **I**-th virtual processor is $(\mathbf{n}[\mathbf{I}]/\mathbf{k})^2$ times bigger than the volume of computations performed by the benchmark code. This line just says it.

Next line specifies volumes of data in bytes to be transferred between the virtual processors during execution of the algorithm. It simply says that **i**-th virtual processor ($\mathbf{i} = 1,\ldots$) will send attributes of all its bodies to the host-processor where they should be visualized. Note that this definition describes just one iteration of the main loop of the algorithm what is quite good approximation because practically all computations and communications concentrate in this loop. Therefore, the total time of the execution of this algorithm is approximately equal to the running time of one iteration multiplied by the total number of iterations.

Finally, the **scheme** block describes how exactly virtual processors interact during execution of the algorithm. It says that first all the virtual processors perform in parallel 100 per cent of computations that should be performed, and then all the processors, except the host processor, send in parallel 100 per cent of data that should be sent to the host-processor.

The most principal fragments of the rest code of this mpC application are:

```
void [*] main(int [host]argc, char **[host]argv)
{
  ...
  TestGroup[] = (*AllGroups[0])[];
  recon Update_group(TestGroup, TestGroupSize);
  {
    net Galaxy(NofG, TestGroupSize, NofB) g;
    ...
  }
}
```

The **recon** statement uses a call of the function **Update_Group** with actual parameters **TestGroup** and **TestGroupSize** to update the estimation of the performance of the physical processors executing the application. The main part of the total volume of computations performed by each virtual processor just falls into execution of calls to the function **Update_Group**. Therefore, the obtained estimation of performances of the real processors will be very close to their actual performances shown while executing this program.

Next line defines the abstract network **g** of the type **Galaxy** with the actual parameters **NofG**—the actual number of groups of bodies, **TestGroupSize**—the size of the test group of bodies used in the benchmark code, and **NofB**—an array of **NofG** elements containing actual numbers of bodies in the groups. The rest computations and communication will be performed on this network.

The mpC programming system maps virtual processors of the abstract network **g** to real parallel processes constituting the running parallel program. While performing the mapping, the programming system uses, on the one hand, the information about configuration and performance of physical processors and communication links of the network of computers executing the program, and on the other hand, the above information about the parallel algorithm to be performed by the defined abstract network. The programming system does the mapping at runtime and tries to minimise the total running time of the parallel program.

Next example is an application multiplying matrix **A** and the transposition of matrix **B**, i.e., implementing matrix operation $\mathbf{C} = \mathbf{A} \times \mathbf{B}^T$, where **A**, **B** are dense square $\mathbf{n} \times \mathbf{n}$ matrices. This application implements a heterogeneous 1D clone of the parallel algorithm used in ScaLAPACK for matrix multiplication, which can be summarized as follows:

- Each element in **C** is a square $\mathbf{r} \times \mathbf{r}$ block and the unit of computation is the computation of one block, i.e., a multiplication of $\mathbf{r} \times \mathbf{n}$ and $\mathbf{n} \times \mathbf{r}$ matrices. For the sake of simplicity, we assume that **n** is a multiple of **r**.
- The **A**, **B**, and **C** matrices are identically partitioned into **p** horizontal slices, where **p** is the number of processors. There is one-to-one mapping between these slices and the processors. Each processor is responsible for computing its **C** slice.
- At each step, a row of blocks (the pivot row) of matrix **B**, representing a column of blocks of matrix $\mathbf{B}^T$, is communicated (broadcast) vertically; and all processors compute the corresponding column of blocks of matrix **C** in parallel.
- Because all **C** blocks require the same amount of arithmetic operations, each processor executes an amount of work proportional to the number of blocks that are allocated to it, hence, proportional to the area of its slice. Therefore, to balance the load of the processors, the area of the slice mapped to each processor is proportional to its speed.
- Communication overheads may exceed gains due to parallel execution of computations. Therefore, there exists some optimal number of available processors to perform the matrix multiplication. The algorithm involves in computations this optimal number of processors.

The following definition of the network type **ParallelAxBT**

```
nettype ParallelAxB(int p, int n, int r, int t, int d[p]) {
  coord I=p;
  node {I>=0: bench*(d[I]*n/r/t); };
  link (J=p) {
    I!=J: length*(d[J]*n*sizeof(double)) [J]->[I];
  };
  parent [0];
  scheme {
    int i, j, PivotProcessor=0, PivotRow=0;
    for(i=0; i<n/r; i++, PivotRow+=r)
    {
     if(PivotRow>=d[PivotProcessor])
     {
      PivotProcessor++;
      PivotRow=0;
     }
     for(j=0; j<p; j++)
      if(j!=PivotProcessor)
       (100.*r/d[PivotProcessor])%%[PivotProcessor]->[j];
     par (j=0; j<p; j++)
      (100.*r/n)%%[j];
      }
    };
  };
```

just describes the algorithm. It is assumed that the test code, used for estimation of the speed of actual processors, multiplies **r** × **n** and **n** × **t** matrices, where **t** is small enough compared to **n** and supposed to be a multiple of **r**. It is also assumed that **i**-th element of vector parameter **d** is just the number of rows in the **C** slice mapped to the **i**-th virtual processor of the abstract network performing the algorithm. Correspondingly, the **node** declaration specifies that the volume of computations to be performed by the **i**-th virtual processor is **d**[**i**] * **n**/**r**/**t** times bigger than the volume of computations performed by the test code.

The **link** declaration specifies that each virtual processor will send its **B** slice to all other virtual processors. The communication pattern described by this declaration, and known as *star*, is static. It is the same for all network objects of the **ParallelAxBT** type. In general, the mpC language allows dynamic communication patterns. There are many ways to describe dynamic patterns in mpC. For example, the following network type definition

```
  nettype DynPattern(int p, int pattern) {
    coord I=p;
    node { I>=0: bench*I; };
    link {
      pattern= =STAR: length*(I*sizeof(double)) [0]->[I];
      pattern= =RING: length*(I*sizeof(double)) [I]->[(I+1)/
      p];
    };
  };
```

describes the star or ring communication topology dependent on parameter **pattern**.

The **scheme** declaration specifies **n**/**r** successive steps of the algorithm. At each step, the virtual processor **PivotProcessor**, which holds the pivot row, sends it to each of the rest virtual processors thus executing **r**/**d**[**PivotProcessor**] * **100** per cent of total data transfer through the corresponding communication link. Then, all virtual processors compute the corresponding column of blocks of matrix **C** in parallel, each thus executing **r**/**n** * **100** per cent of the total volume of computation to be performed by the processor.

The most interesting fragments of the rest code of this mpC application are:

```
  ...
  recon SerialAxBT(a, b, c, r, n, t);
  ...
  [host]:
  {
    int j;
    struct {int p; double time;} min;
    double time;
    for(j=1; j<=p; j++) {
      Partition(j, powers, d, n, r);
      time=timeof(net ParallelAxBT(j, n, r, t, d) w);
      if(time<min.time) {
       min.p=j;
       min.time=time;
      }
    }
    p=min.p;
  }
  ...
  Partition (p, powers, d, n, r);
  {
    net ParallelAxBT(p, n, r, t, d) w;
    repl [w]N, [w]i;
    int [w]myN;
    N=[w]n;
    myN=([w]d)[I coordof w];
    [w]:
    {
      double A[myN/r][r][N], BT[myN/r][r][N],
             C[myN/r][r][N], Brow[r][N];
      repl PivotProcessor, RelPivotRow, AbsPivotRow;
      ...
      for(AbsPivotRow=0, RelPivotRow=0, PivotProcessor=0;
         AbsPivotRow < N;
         RelPivotRow+ = r, AbsPivotRow+ = r)
      {
       if(RelPivotRow >= d[PivotProcessor]) {
          PivotProcessor++;
          RelPivotRow=0;
       }
       Brow[]=[w:I==PivotProcessor]BT[RelPivotRow/r][];
       for(i=0; i<myN/r; i++)
         SerialAxBT(A[i][0],Brow[0],C[j][0]+AbsPivotRow,r,
         N,r);
      }
    }
  }
```

The **recon** statement updates the estimation of performances of actual processors using serial multiplication of test **r** × **n** and **n** × **t** matrices with function **SerialAxBT**.

The computations performed by each virtual processor mainly fall into the execution of calls to **SerialAxBT**.

Next block, executed by the virtual host-processor, calculates the optimal number of actual processors to be involved in the parallel matrix multiplication. In case of relatively slow communications, the speedup due to parallelization of computations may not compensate the time elapsed by data transfer. Therefore, there exists some optimal number of available processors to perform the parallel algorithm. The **timeof** operator just estimates the running time of the algorithm described by its operand—some particular network type, without its real execution. After execution of the block, the value of variable **p** will be equal to this optimal number.

The definition of the network **w** causes mapping the virtual processors of this abstract network to the actual processors of the underlying network of computers in such a way that minimizes the execution time.

The block, following the definition of the network **w** and executed by this network, just implements the algorithm of parallel matrix multiplication, the main features of which have been specified in the definition of the network type **ParallelAxBT**.

The scheme declarations in the above two examples are relatively simple. They just reflect the relative simplicity of the underlying parallel algorithms. In general, the mpC language allows the programmer to describe quite sophisticated heterogeneous parallel algorithms by means of wide use of parameters, locally declared variables, expressions and statements.

2.3. Brief discussion of the mpC programming language

This section briefly discusses some important topics regarding the mpC language, which are not presented in details in this paper but should be addressed. The first topic is the kind of applications the mpC language is suited for. It is most suitable for parallel solution of irregular problems, such as galaxy simulation, on both homogeneous and heterogeneous distributed-memory computer systems. It is also suitable for solution of regular problems, like dense liner algebra problems, in heterogeneous environments. Two approaches to solving regular problems on heterogeneous clusters with mpC are presented in [18]. The first approach is the irregularization of problem in accordance with irregularity of the executing hardware. The second one is the distribution of a relatively large number of homogeneous parallel processes over the heterogeneous cluster in accordance with the performance of its elements. The mpC language provided natural implementation in the portable form of both the approaches.

The mpC language is designed with common networks of computers as a target parallel architecture in mind. In such networks, the performance of communication links and processors are rarely balanced for high performance computing. Normally, the ratio of the speed of data transfer and the speed of processors is significantly lower than that of specialised parallel computer systems. In addition, the performance characteristics of common networks are normally not stable and can vary significantly due to many independent users having access to their resources. The reasons make very difficult (if possible at all) a portable and efficient implementation of parallel algorithms with fine and sophisticated structure of communications or communications prevailing over computations on common networks of computers. Therefore, the mpC language is mainly suitable for programming parallel algorithms with relatively simple and coarse-grained structure of communications as well as computations prevailing over communications.

The mechanism of nodal and especially network functions supports task parallelism. Different nodal and network functions can be called in parallel each having different control flow. Network functions enable modular parallel programming in mpC. One programmer can implement some parallel algorithm in the form of network function, and other programmers can safely use such a program unit in parallel with other computations in their applications without any knowledge of its code.

Another interesting topic is applications where different parallel algorithms are coupled. There are many possible ways of programming such applications in mpC. If the two algorithms are loosely coupled, two different network objects of different type executing the algorithms in parallel can be defined (the mpC language allows many different network objects to exist in parallel). The mpC programming system will try to map the algorithms in such a way to ensure the best execution time of the whole application. Alternatively, two different network objects of different type executing the algorithms serially can be defined (especially, in case of strong data dependency). In the latter case, the first network object should be destructed before the second one is created to make all resources available when mapping each of the algorithms on the underlying hardware. If the two algorithms are tightly coupled, they can be described in the framework of the same network type and executed on the same network object.

The next important topic is data redistribution when utilizing different automatic network objects. Currently it is mainly possible by gathering and scattering via the parent what is not efficient. Other possible ways such as the use of static data objects and their explicit redistribution are not very natural and easy to use. The problem of efficient and natural data redistribution when utilizing different automatic network objects will be addressed in the future work on the mpC language.

3. Model of heterogeneous network of computers

The basic model of the executing heterogeneous network of computers did not take into account communication links between computers and considered the network as a set of heterogeneous multiprocessors. Each computer was characterised by two attributes—the time of execution of a (serial) benchmark code on the computer and the number of physical processors. The first attribute was a function of time, **b(t)**, and could vary even during the execution of the same mpC application (if the application used the **recon** statement). The second was a constant, **n**, and determined how many non-interacting parallel processes could run on the computer without loss of performance.

The new advanced model is more sophisticated and takes into account material nature of communication links and their heterogeneity. The model considers the executing heterogeneous network as a multilevel hierarchy of interconnected sets of heterogeneous multiprocessors. The hierarchy reflects the heterogeneity of communication links and can be represented in the form of attributed tree.

Each node of the tree represents a homogeneous communication space of the heterogeneous network. The first attribute associated with an internal node is the set of computers, which is just a union of sets of computers associated with its children.

The second is the speed of data transfer between two computers from different sets associated with its children. This attribute characterizes point-to-point communication at this communication layer and is a function of size of the transferred data block, $\mathbf{s}(\mathbf{d})$. Note, that $\mathbf{s}(\mathbf{0})$ is not zero and equal to startup time of point-to-point communication at this layer.

The third attribute specifies if the communication layer allows parallel point-to-point communications between different pairs of computers without loss of data transfer speed, or the layer serializes all communications. This attribute can have two values—**Serial** and **Parallel**. A pure Ethernet network is serial. At the same time, the use of switches can make it parallel.

The next group of attributes is only applicable to a parallel communication layer. It characterizes collective communication operations such as broadcast, scatter, gather, and reduction. The point is that a collective communication operation cannot be considered as a set of independent point-to-point communications. It normally has some special process, called *root*, which is involved in more communications than other participating processes. The level of parallelism of each of the collective communication operations depends on its implementation and is reflected in the model by means of the corresponding attribute. For example, the attribute $\mathbf{f_b}$ characterizes the level of parallelism of the broadcast operation. It is supposed that the execution time $\mathbf{t}$ of this operation can be calculated as follows

$$t = f_b * t_p + (1 - f_b) * t_s$$

where $\mathbf{t_s}$ is the time of purely serial execution of the operation, and $\mathbf{t_p}$ is the time of ideally parallel execution of this operation ($0 <= \mathbf{f_b} <= 1$).

Each terminal node of this tree represents a single (homogeneous) multiprocessor computer. In addition to the attributes inherited from the basic model, such a node is also characterised by the attributes of the communication layer provided by the computer.

Fig. 1 depicts the model for a local network of five computers, named A, B, C, D and E. Computer A is a distributed-memory 8-processor computer, D is a shared-memory 2-processor server. Computers B, C and E are uniprocessor workstations. The local network consists of two segments with A, B and C belonging to the first segment. Computers D and E belong to the second segment.

The speed of transfer of a data block of $\mathbf{k}$ bytes from a process running on computer C to a process running on computer D is estimated by $\mathbf{s_0}(\mathbf{k})$, meanwhile the speed of transfer of the same data block from a process running on computer C to a process running on computer A is estimated by $\mathbf{s_1}(\mathbf{k})$.

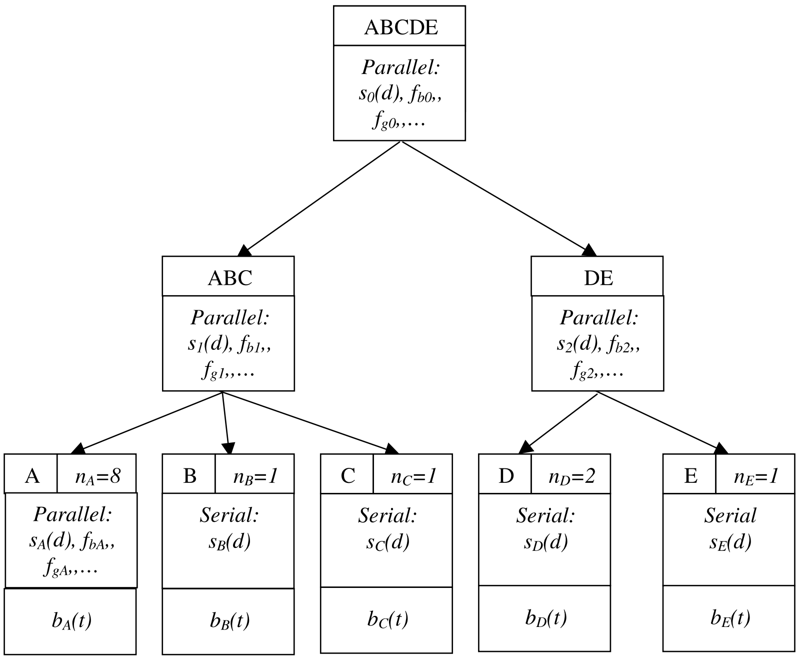
ABCDE
Parallel: $s_0(d)$, f_{b0},, f_{g0},,...
ABC
Parallel: $s_1(d)$, f_{b1},, f_{g1},,...
DE
Parallel: $s_2(d)$, f_{b2},, f_{g2},,...
A $n_A=8$
Parallel: $s_A(d)$, f_{bA},, f_{gA},,...
$b_A(t)$
B $n_B=1$
Serial: $s_B(d)$
$b_B(t)$
C $n_C=1$
Serial: $s_C(d)$
$b_C(t)$
D $n_D=2$
Serial: $s_D(d)$
$b_D(t)$
E $n_E=1$
Serial $s_E(d)$
$b_E(t)$

Fig. 1. The hierarchical model of the heterogeneous network of five computers.

The level of parallelism of a broadcast involving processes running on computers B, C and E is $\mathbf{f_{b0}}$, meanwhile that of a broadcast involving processes running on computer A is $\mathbf{f_{bA}}$.

The communication model presented is simple and rough enough. It is used at runtime by the mpC programming system to predict the execution time of the implemented parallel algorithm. It uses a small number of integral attributes presenting some average characteristics rather then detailed and fine-structured description.

The main reason of this simplicity is that the target architecture for the mpC language is common networks of computers, which normally are multi-user environment of irregular structure with not very stable characteristics. Therefore, fine-grained communication effects can hardly be reliably predicted for that architecture.

Secondly, the mpC language is aimed at programming applications, in which computations prevail over communications, i.e., the contribution of computations in the total execution time is much higher than that of communications. If it is not the case, it normally means that the main goal of the application is not to speed up the solution of some individual problem, and the distribution of its components over different computers is its intrinsic feature, i.e., the application is actually distributed not parallel one.

Thus, the mpC language needs an efficient communication model of common heterogeneous networks of computers suitable for prediction of the execution time of data transfer operations involving the transfer of relatively big volumes of data.

The accuracy of the prediction does not need to be too high because the contribution of communications in the total execution time is supposed to be relatively small. Actually, the accuracy cannot be high because of the nature of the modelled hardware.

This communication model is designed to satisfy the primary necessities of the mpC language. Its main disadvantage that should be addressed in the future work is that it is static. An efficient way to update its parameters at runtime to reflect the current situation could improve its accuracy. Another possible direction of improvement is the model of parallel communication layer and collective communication operations. More experiments with different network configurations are needed to make the model more accurate for a wide range of common networks.

4. Mapping of abstract mpC networks to real executing network

Any mpC program, running on the network of computers, is nothing more then a number of processes interacting via message passing. The total number of the processes and the number of processes running on each computer of the network are determined by the user when the latter starts up the program. This information is available to the component of the mpC runtime system responsible for mapping of abstract mpC networks to these processes. Each definition of the abstract network in the mpC application causes creating the group of processes that will play the role of virtual processors of the abstract network. The main criteria of selection of processes for this group is to minimize the time of execution of the parallel algorithm, described by the corresponding abstract mpC network, on the particular network of computers.

Thus, at runtime the mpC programming system solves the problem of optimal mapping of virtual processors of the abstract mpC network into the set of processes running on different computers of the heterogeneous network. When solving the problem, the mpC system is based on the following:

- The mpC model of the parallel algorithm, which should be executed.
- The model of the executing network of computers, which reflects the state of this network just before the execution of the algorithm.
- A map of processes of the parallel program, for each computer displaying both the total number of running processes and the number of free processes, that is, those processes available to play the role of virtual processor of the abstract mpC network.

Each particular mapping, μ: $I -> C$, where I is a set of coordinates of the virtual processors of the abstract mpC network, and C is a set of computers of the executing network, is characterized by the estimation of the time of execution of the algorithm on the network of computers. The estimation is calculated based on the models of the parallel algorithm and the executed network.

Each computation unit in the **scheme** declaration of the form $e\%\%[i]$ is estimated as follows:

$$\mathbf{timeof}(e\%\%[i]) = (e/100) * v_i * b_{\mu(i)}(t_0),$$

where v_i is the total volume of computations to be performed by the virtual processor with the coordinates i, and $b_{\mu(i)}(t_0)$ is the time of execution of the benchmark code on the computer $\mu(i)$ provided by the execution of the corresponding **recon** statement (t_0 denotes that time when this execution took place).

Each communication unit of the form $e\%\%[i]\text{->}[j]$ is estimated as follows:

$$\mathbf{timeof}(e\%\%[i]- > [j]) = (e/100) * w_{i->j} * s_{\mu(i)} - > \mu(j)(w_{i->j}),$$

where $w_{i->j}$ is the total volume of data to be transferred from the virtual processor with the coordinates i to the virtual processor with the coordinates j, and $s_{\mu(i)->\mu(j)}(w_{i->j})$ is the speed of transfer of data block of $w_{i->j}$ bytes between computers $\mu(i)$ and $\mu(j)$.

A simple calculation rule is associated with each sequential algorithmic pattern in the **scheme** declaration. For example, the estimation **T** of the pattern

```
for(e1;e2;e3)a
```

is calculated as follows:

```
for(T = 0, e1; e2; e3)
  T += timeof(a);
```

The estimation **T** of the pattern

```
if(e) a1 else a2
```

is calculated as follows:

```
if(e)
  T = timeof(a1);
else
  T = timeof (a2);
```

The above rules just reflect semantics of the corresponding serial algorithmic patterns.

The rule to calculate the estimation **T** of the parallel algorithmic pattern

```
par(e1;e2;e3)a
```

is more complicated. Informally, the above pattern describes parallel execution of some actions (mixtures of computations and communications) on the corresponding abstract mpC network. Let $A = \{a_0, a_1, \dots, a_{N-1}\}$ be a set of the actions ordered in accordance with the estimation of the time of their execution, namely, **timeof**(a_0) >= **timeof**(a_1) >= ⋯ >= **timeof**(a_{N-1}). Let B be a subset of A consisting of all actions that only perform communications, $B = \{b_0, b_1, \dots, b_{Q-1}\}$. Let $C = \{c_0, c_1, \dots, c_{M-1}\}$. Finally, let v_i be the number of virtual processors of the abstract mpC network mapped on the computer c_i, and f_i be the total number of physical processors of the computer. Then the rule to calculate the estimation **T** of the pattern looks as follows:

```
for(j = 0, T = 0; j<M; j++) {
  for(i = 0, $T_0$ = 0, k = 0; k<Upper($v_j$,$f_j$) && i<N; i++) {
    if($a_i$ performs some computations on $c_j$) {
     $T_0$ += timeof($a_i$);
     k++;
    }
  }
  T = max(T, $T_0$);
}
T = max(T, timeof(B));
```

Here, the function **Upper** is defined as follows:

```
Upper (x, y) = if(x/y <= 1)
               then 1
               else if((x/y) * y == x)
                    then x/y
                    else x/y + 1
```

Informally, the above system of loops first computes for each computer the estimation T_0 of the time of parallel execution of those actions, which use that computer for some computations. The estimation is calculated, proceeding from the assumption, that if the number of parallel actions on one computer exceeds the number of its physical processors, then

- The actions are distributed evenly over the physical processors, that is, the number of actions executed by different physical processors differs by at most one.
- The most computationally intensive actions are executed on the same physical processor.

Then those parallel actions, which are not related to computations, that is, perform pure communications, are taken into account. These communication actions make up the set B. Let $I(B)$ be the least communication layer covering all communication links involved in B, and let f_b, f_g be the level of parallelism of broadcast and gather correspondingly for this layer. Then the rule to calculate the estimation **T** of parallel execution of communication operations from set B looks as follows:

```
if (l(B) is serial)
  for(i = 0, T = 0; i < $Q$; i++)
    T += timeof($b_i$);
else if(B matches broadcast/scatter) {
  for(i = 0, $T_{serial}$ = 0, $T_{parallel}$ = 0; i < $Q$; i++) {
    $T_{serial}$ += timeof($b_i$);
    $T_{parallel}$ = max($T_2$, timeof($b_i$));
  }
  T = $f_b$*$T_{parallel}$ + (1 - $f_b$) * $T_{serial}$
}
else if(B matches gather) {
  for(i = 0, $T_{serial}$ = 0, $T_{parallel}$ = 0; i < $Q$; i++) {
    $T_{serial}$+ = timeof($b_i$);
    $T_{parallel}$ = max($T_2$, timeof($b_i$));
  }
  T = $f_g$*$T_{parallel}$+(1 - $f_g$)*$T_{serial}$
}
else
  for (i = 0, T = 0; i < $Q$; i++)
    T+ = max(T, timeof($b_i$));
```

The rule just sums the execution time of parallel communication operations if the underlying communication layer serializes all data packages. Otherwise we have a parallel communication layer, and if the set B of communication operations looks like broadcasting or scattering, i.e., one virtual processor sends data to other involved virtual processors, then the time of parallel execution of the communication operations is calculated as if they performed broadcast. Similarly, if B looks like gathering, i.e., one virtual processor receives data from other involved virtual processors, then the time of parallel execution of the communication operations is calculated as if they performed gather. In all other cases, it is assumed that B is a set of independent point-to-point communications. It is responsibility of the programmer not to specify different communication operations sharing the same communication link as parallel ones.

The rule for estimation of the execution time of the parallel algorithmic pattern is the core of the entire mapping algorithm determining its accuracy and efficiency. It takes into account material nature and heterogeneity of both processors and network equipment. It relies on fair allocating processes to processors in a shared-memory multiprocessor normally implemented by operating systems for processes of the same priority (mpC processes are just the case). At the same time, it proceeds from the pessimistic point of view when estimating workload of different processors of that multiprocessor. Estimation of communication cost by the rule is sensitive to scalability of the underlying network technology. It treats differently communication layers serializing data packages and supporting their parallel transfer. The most typical and widely used collective communication operations are treated specifically to provide better accuracy of the estimation of their execution time. An important advantage of the rule is its relative simplicity and effectiveness. The effectiveness is critical because the algorithm is supposed to be multiply executed at runtime.

Most disadvantages of the rule are just the backside of its simplicity and the necessity to keep it effective. Except some common collective communication operations, it is not sensitive to different collective communication patterns such as ring data shifting, tree reduction, etc., treating all them as a set of independent point-to-point communications. The main problem is that recognition of such patterns is very expensive. A possible solution is introduction in the mpC language some explicit constructs for communication pattern specification as a part of the scheme description. Another disadvantage of the rule affecting the accuracy of estimation is that any set of parallel communications is treated as if all they take place at the same communication layer in the hierarchy, namely, at the lowest communication layer covering all involved processors. In reality, some of the communications may use different communication layers. Incorporation of multi-layer parallel communications in this algorithm without significant loss of its efficiency is a very difficult problem, which is supposed to be addressed in the future works.

Ideally, the mpC runtime system should find such a mapping that is estimated to ensure the fastest execution of the parallel algorithm. In general, for accurate solution of this problem as many as M^K possible mappings have to be probated to find the best one (here, K is the power of the set I of coordinates of virtual processors). Obviously, that computational complexity is not acceptable for a practical algorithm that should be performed at runtime. Therefore, the mpC runtime system searches for some approximate solution that can be found in some reasonable interval of time, namely, after probation of $M \times K$ possible mappings instead of M^K.

The underlying algorithm is the following. At the preliminary step, the set I is reordered in accordance with the volume of computations to be performed by the virtual processors, so that the most loaded virtual processor will come first. Let $P = \{p_k\}$ $(k = 0, \ldots, K-1)$ be this well-ordered set. Let Q_j be a subnetwork of the abstract mpC network formed by the set $P_j = \{p_i\}$ $(i = 0, \ldots, j)$ of virtual processors. By definition, a subnetwork is a result of projection of the abstract network onto some subset of its virtual processors. Semantically, the subnetwork is equivalent to its supernetwork modified in the following way:

- The zero volume of computations is set for each virtual processor not included in the subnetwork.
- The zero volume of communications is set for each pair of virtual processors with at least one of which not included in the subnetwork.

Finally, let c_j denote the j-th computer from the set C. Then the main loop of the algorithm can be described by the following pseudo-code:

```
for(k=0; k<K; k++) {
  for(j = 0, $t_{best}$ = MAXTIME, $c_{best}$ = $c_0$; j<M; j++) {
    if($p_k$ is not a parent of the mpC network) {
     Map $p_k$ to $c_j$
     Estimate execution time t for this mapping of $Q_k$ to C
     if(t < $t_{best}$) {
       $t_{best}$ = t;
       $c_{best}$ = $c_j$;
     }
     Unmap $p_k$
    }
  }
  Map $p_k$ to $c_{best}$
}
```

The presented algorithm reflects the focus of the mpC language on applications with computations prevailing over communications. Therefore, the algorithm is driven by virtual processors not communication links. Another argument for that approach is that the maximal number of virtual communication links is equal to the total number of virtual processors squared. Therefore, in general, an algorithm driven by virtual links would be more expensive.

Informally, the algorithm first maps the most loaded virtual processor not taking into account other virtual processors as well as communications. Then, given the first virtual processor mapped, it maps the second most loaded virtual processor only taking into account communications between these two processors and so on. At the **i**-th step, it maps the **i**-th most loaded virtual processor only taking into account data transfer between these **i** virtual processors. This algorithm exploits the obvious observation that the smaller are things, the easier they can be evenly distributed. Hence, bigger things should be distributed under weaker constraints than smaller ones. For example, if you want to distribute a number of balls of different size over a number of baskets of different size, you better start from the biggest ball and put it into the biggest basket; then put the second biggest ball into the basket having the biggest free space and so on. This algorithm keeps balance between ball sizes and free basket space and guarantees that if at some step you do not have enough space for the next ball, it simply means that there is no way to put all the balls in the baskets. Similarly, if the above algorithm cannot balance the load of actual processors in case of practically zero communication costs, it simply means that there is no way to balance them at all. This algorithm will also work well if data transfer between more loaded virtual processors is more significant than data transfer between less loaded ones. In this case, more loaded virtual communication links are taken into account at earlier stages of the algorithm.

An obvious case when this mapping algorithm may not work well is when the least loaded virtual processor is involved in transfer of much bigger volume of data than more loaded ones, and the contribution of communications in the total execution time is significant. But even quick analysis shows that it is not the case for most parallel algorithms.

5. Experimental results

This section presents some results of experiments with the matrix multiplication application presented in Section 2.2. All presented results are obtained for $\mathbf{r} = \mathbf{t} = 20$.

A small local network of 9 Solaris and Linux workstations (named `csultra01`, `csserver`, `csultra02`, `csultra03`, `csultra04`, `csultra05`, `csultra07`, `csultra08`, and `csultra10`) is used for the experiments. The network is based on 100 Mbit Ethernet with a switch enabling parallel communications between the computers. The initial static structure of the network automatically obtained by the mpC environment and saved in the form of ASCII file was the following:

```
parallel(0.49, 0.97) c154056 c4407404 c11188106
#csultra01
s1 p4288 n6 serial c251556 c6987375 c75412443
#csserver
s1 p6667 n6 serial c291589 c8708311 c32021610
#csultra02
s1 p4213 n6 serial c421605 c17034484 c73290148
#csultra03
s1 p4092 n6 serial c196882 c10150458 c69696275
#csultra04
s1 p4263 n6 serial c360042 c21758106 c74743908
#csultra05
s1 p4288 n6 serial c411954 c18204945 c75087604
#csultra07
s1 p4260 n6 serial c321599 c15395040 c78303652
#csultra08
s1 p4357 n6 serial c430860 c21868078 c72443345
#csultra10
s1 p4318 n6 serial c382766 c30094072 c70345882
```

Here, each computer is characterized by seven parameters. The first parameter, `s`, determines the number of processors. Thus, all the workstations are uniprocessor computers.

The second parameter, `p`, determines the performance of the computer demonstrated on execution of some serial test code. One can see that `csserver` is the most powerful computer, meanwhile the rest of computers are approximately of the same power. Note that at runtime the execution of the **recon** statement updates the value of the parameter for each participated computer.

The third parameter, n, determines the total number of processes of the parallel program to run on the computer. Each computer runs 6 processes, what means that the total number of processes is equal to 54.

The fourth parameter determines the scalability of the communication layer provided by the computer. In this case, all computers provide serial communication layers.

Finally, the last three parameters determine the speed of point-to-point data transfer between processes running on the same computer as a function of size of the transferred data block. The first of them specifies the speed of transfer of a data block of 64 bytes (measured in bytes per second), and the second and the third specify that of 64^2 and 64^3 bytes correspondingly. The speed of transfer of a data block of an arbitrary size is calculated by interpolation of the measured speeds. Although there could be used more points to approximate this function without visible loss of efficiency (remember that corresponding calculations were performed at runtime), so far the 3-point approximation appeared accurate enough.

The homogeneous communication space of higher level is also characterized by those three parameters. Besides, the layer is detected as a parallel communication layer with factors 0.49 and 0.97 characterizing the level of parallelism of broadcast and gather correspondingly.

The execution time of the matrix multiplication application is compared with its execution time predicted by the mpC programming system as well as with the execution time of a simplified version of this application not taking into account the cost of data transfers at all.

Fig. 2 depicts the speedup demonstrated by the advanced application against the simplified one. Meanwhile the simplified application always involves in matrix mul-

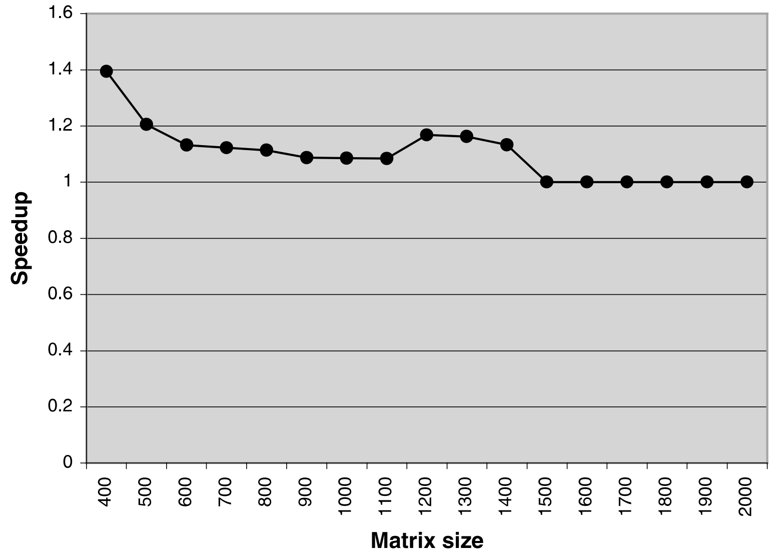

Fig. 2. The speedup against matrix multiplication involving all available workstations.

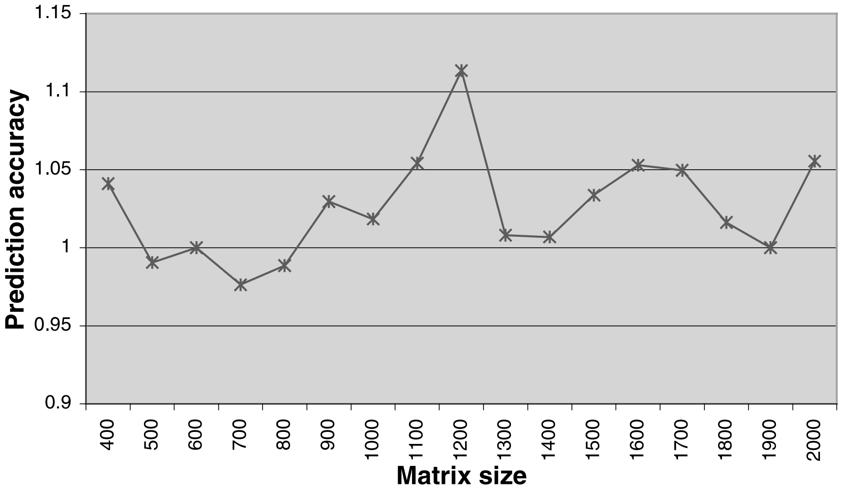

Fig. 3. Ratio of the predicted and real execution time.

tiplication all available workstations, the advanced application only involves some optimal number, which varies from 4 to 9 in our experiments and depends on the matrix size and the current performance demonstrated by different workstations.

Fig. 3 demonstrates how the actual execution time agrees with the execution time predicted at runtime by the mpC programming system. It gives the ratio of the predicted execution time and the real one as a function of matrix size. One can see that, as a rule, the real execution time differs from the predicted one not more than by 5–6%.

6. Related work

The section surveys related papers from the literature. The papers range into four categories: papers focusing on general load-balancing strategies, papers dealing with design of heterogeneous parallel algorithms, papers devoted to high-level tools facilitating the implementation of parallel algorithms, and papers covering performance models of parallel architectures.

General load balancing strategies for heterogeneous platforms have been widely studied. Scheduling the tasks can be performed either dynamically or statically or a mixture of both. Most schedulers use simple mapping strategies such as master–slave techniques or the use of currently observed performance of each machine to decide for the next distribution of work (see survey papers [6–9]). Several scheduling and mapping heuristics have been proposed to map task graphs onto heterogeneous networks of workstations [10–13]. Scheduling tools such as Prophet [14] or AppLeS [15] are available (see also the survey paper [16]). Mapping algorithms underlying all the strategies are quite general. They make very little suggestions about the nature of scheduled tasks (if any) often considering them as a set of independent equal units. More attention is paid to the model of heterogeneous hardware.

A small number of recent papers are devoted to the design of concrete heterogeneous parallel algorithms [1,17–20]. They mainly deal with linear algebra. They analyse different modifications of traditional homogeneous algorithms and try to find their best mapping on heterogeneous networks of workstations. The mappings take into account all peculiarities of the corresponding parallel algorithms and are based on very careful performance analysis. The MPI library or the mpC language is used to implement the algorithms.

Several high-level languages have been developed to facilitate the implementation of parallel algorithms on distributed-memory architectures, among which HPF (High Performance Fortran) and Dataparallel C are the most popular. The languages were originally designed with homogeneous multiprocessor as target architecture in mind. HPF was standardized in 1994 as HPF 1.1 [21]. It only provided regular mapping patterns and did not support uneven distribution of the array elements over processors, necessary for balanced mappings of heterogeneous algorithms. This did not satisfy the HPF community; and in 1997 a new HPF version, 2.0, was standardized [22]. HPF 2.0 extends BLOCK distribution with the ability to explicitly specify the size of each individual block (GEN_BLOCK distribution). In addition, HPF 2.0 introduces a new distribution method, INDIRECT distribution, where each individual array element is explicitly assigned to a particular processor—this is achieved by the use of a vector-subscript type of array. Thus, HPF 2.0 provides some basic support for programming heterogeneous algorithms. At the same time, HPF 2.0 still provides no language constructs allowing the programmer to control better mapping of the heterogeneous algorithms to heterogeneous clusters. The HPF programmer should rely on some default mapping provided by the HPF compiler. The mapping cannot be sensitive to peculiarities of each individual algorithm just because the HPF compiler has no information about the peculiarities. Therefore, to control the mapping and take into account both the peculiarities of the implemented parallel algorithm and the peculiarities of the executing heterogeneous environment, the HPF programmer needs to additionally write a good peace of quite complex code.

Dataparallel C [23] is a C-based counterpart of HPF. It presents the model of computation characterized by a global name space, synchronous execution, and virtual processors as the unit of parallelism. The virtual topologies offered by Dataparallel C are ring or torus. Virtual processors are, therefore, organized as one- or two-dimensional arrays. To map the virtual processors, Dataparallel C uses a data file provided by the user and dictating the initial mapping. Dynamic load balancing is accomplished through periodic exchange of load information during calls to the routing library at runtime. Entire processes are not moved in this load-balancing scheme; merely some the virtual processors assigned to a given machine are shifted. Since a virtual processor is characterized by its data, migrating the data alone is sufficient.

A good deal of theoretical research has focused on models of parallel computers. The most widely used parallel models are the parallel random access machine (PRAM) [24], the bulk-synchronous parallel model (BSP) [25], and the LogP model

[26]. All the models assume a parallel computer to be a homogeneous multiprocessor. The PRAM is the most simplistic model. It assumes that all processors work synchronously and that interprocessor communication is free. The BSP allows processors to work asynchronously and models latency and limited bandwidth. Finally, the LogP is the most realistic model among them. It characterizes a parallel machine by the number of processors (P), the communication bandwidth (g), the communication delay (L), and the communication overhead (o). The LogP model has been successfully used for developing fast and portable parallel algorithms for (homogeneous) supercomputers. In this case, the portability of a parallel algorithm means that the algorithm adapts to the particular supercomputer configuration in terms of these parameters. Nonetheless, the Log P model or its extensions are not appropriate to model heterogeneous networks of computers, mainly due to their irregularity and non-deterministic nature. The irregularity multiply increases the number of parameters, meanwhile the non-deterministic behavior of common networks makes the parameters stochastic. The resulting model becomes extremely sophisticated and non-effective. At the same time, this over-sophistication is not justified by the provided accuracy. The model presented in Section 3 is much simpler and more effective and has demonstrated quite good accuracy.

7. Conclusion and future work

The paper has presented new advanced features of the mpC parallel language that allow the programmer to define all main features of the implemented parallel algorithm that can have an impact on the performance of execution of the algorithm on a heterogeneous network of computers. The features include the total number of participating parallel processes, the total volume of computations to be performed on each of the processes, the total volume of data to be transferred between each pair of the processes, and how exactly the processes interact during the execution of the algorithm. The mpC programming system uses that abstraction of the parallel algorithm together with the model of the executing heterogeneous network to map the processes of the parallel program to this network in such a way that ensure better execution time. The mapping is executed at runtime; therefore its efficiency is crucial for the total execution performance of mpC applications. The presented model of a heterogeneous network and the mapping algorithm were developed to keep balance between accuracy and efficiency.

Some directions of the future work on the mpC programming system include improvement of the underlying model of heterogeneous network and the mapping algorithm, as well as implementation of the mpC technology in the form of library with bindings to C++, Fortran, and Java.

Acknowledgements

The research work presented in the paper is purely initiative not supported by any organisation. It has been carried out by Alexey Lastovetsky (design and run-time support implementation), Ilya Ledovskih (compiler front-end implementation), and Alexey Kalinov (compiler back-end implementation). The author would like to thank the anonymous reviewers whose comments and suggestions have greatly improved the presentation of the paper.

References

[1] O. Beaumont, V. Boudet, F. Rastello, Y. Robert, Matrix multiplication on heterogeneous platforms, IEEE Transactions on Parallel and Distributed Systems 12 (10) (2001) 1033–1051.
[2] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarrra, MPI: The Complete Reference, MIT Press, Cambridge, MA, 1996.
[3] A. Lastovetsky, The mpC Programming Language Specification, Technical Report, ISPRAS, December 1994.
[4] D. Arapov, A. Kalinov, A. Lastovetsky, I. Ledovskih, A language approach to high performance computing on heterogeneous networks, Parallel and Distributed Computing Practices 2 (3) (1999) 87–96.
[5] A. Lastovetsky, D. Arapov, A. Kalinov, I. Ledovskih, A parallel language and its programming system for heterogeneous networks, Concurrency: Practice and Experience 12 (13) (2000) 1317–1343.
[6] F. Berman, High-Performance Schedulers, in: I. Foster, C. Kesselman (Eds.), The Grid: Blueprint for a New Computing Infrastructure, Morgan-Kaufmann, Los Altos, CA, 1999, pp. 279–309.
[7] M. Cierniak, M. Zaki, W. Li, Customized dynamic load balancing for a network of workstations, Journal of Parallel and Distributed Computing 43 (1997) 156–162.
[8] M. Cierniak, M. Zaki, W. Li, Scheduling algorithms for heterogeneous network of workstations, Computer Journal 40 (6) (1997) 356–372.
[9] S. Anastasiadis, K.C. Sevcik, Parallel application scheduling on networks of workstations, Journal of Parallel and Distributed Computing 43 (1997) 109–124.
[10] G. Sih, E. Lee, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, IEEE Transactions on Parallel and Distributed Systems 4 (2) (1993) 175–183.
[11] M. Tan, H.J. Siegel, J.K. Antonio, Y.A. Li, Minimizing the application execution time through scheduling of subtasks and communication traffic in a heterogeneous computing system, IEEE Transactions on Parallel and Distributed Systems 8 (8) (1997) 857–871.
[12] M. Maheswaran, H.J. Siegel, A dynamic matching and scheduling algorithm for heterogeneous computing systems, Proceedings of the Seventh Heterogeneous Computing Workshop, 1998.
[13] M. Iverson, F. Ozguner, Dynamic, competitive scheduling of multiple dags in a distributed heterogeneous environment, Proceedings of the Seventh Heterogeneous Computing Workshop, 1998.
[14] J. Weissman, X. Zhao, Scheduling parallel applications in distributed networks, Cluster Computing 1 (1) (1998) 109–118.
[15] F. Berman, R. Wolski, S. Figueira, J. Schopf, G. Shao, Application-level scheduling on distributed heterogeneous networks, Proceedings of the Supercomputing'96, 1996.
[16] H.J. Siegel, H.G. Dietz, J.K. Antonio, Software support for heterogeneous computing, ACM Computing Surveys 28 (1) (1996) 237–239.
[17] D. Arapov, A. Kalinov, A. Lastovetsky, 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, 1998, pp. 332–343.
[18] A. Kalinov, A. Lastovetsky, Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers, Journal of Parallel and Distributed Computing 64 (4) (2001) 520–535.
[19] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, Y. Robert, A proposal for a heterogeneous cluster scalapack (dense linear solvers), IEEE Transactions on Computers 50 (10) (2001) 1052–1070.
[20] O. Beaumont, A. Legrand, F. Rastello, Y. Robert, Static LU decomposition on heterogeneous platforms, International Journal of High Performance Computing 15 (3) (2001) 310–323.

[21] High Performance Fortran Forum, High Performance Fortran Language Specification, version 1.1, 1994.
[22] High Performance Fortran Forum, High Performance Fortran Language Specification, version 2.0, 1997.
[23] P.J. Hatcher, M.J. Quinn, Data-Parallel Programming on MIMD Computers, The MIT Press, Cambridge, MA, 1991.
[24] S. Fortune, J. Wyllie, Parallelism in random access machines, Proceedings of the 10th Annual Symposium on Theory of Computing, 1978, pp. 114–118.
[25] L.G. Valiant, A bridging model for parallel computation, Communications of the Association for Computing Machinery 33 (8) (1990) 103–111.
[26] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. von Eicken, LogP: Towards a realistic model of parallel computation, Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, May 1993.