Institute for System Programming

mpC

Tutorial

by Alexey Lastovetsky

May 2002

Version 1.0

Table of Contents

What is mpC?

The mpC programming language is an extension of the ANSI C language designed specially for programming parallel computations on common networks of heterogeneous computers. The main goal of parallel computing is to speed up solving problems. Just this differs parallel computing from distributed computing the main goal of which is to make software, which is inherently distributed over different computers, work together. In the case of parallel computing, partitioning the entire program into components running on different computers is the only way to speed up execution of the program and not its inherent property. Therefore, mpC provides means that facilitate writing efficient and portable parallel applications for solving problems on common networks of computers. Therefore, in mpC the basic attention is paid to the means that facilitate writing efficient and portable applications solving problems on common networks of computers.

This tutorial explains how to write simple parallel programs using the mpC programming language. It includes the following chapters:

- First programs
- Networks
- Network type
- Network parent
- Synchronization of processes
- Network functions
- Subnetworks
- Vector computations
- Heterogeneous parallel computing

First programs

First programs

The parallel program is a set of processes running in parallel. Processes of parallel program interact (that is, synchronize their work and interchange data) passing to each other messages. In implementation time mpC programmer doesn't know how many processes will constitute the program and which computers will execute which processes. It's environment, in which the program is launched, that decides on which processor which process is executed. mpC programmer can only determine what computations are performed by every process constituting the parallel program.

This chapter covers the following topics:

- Single-processor "Hello, world!"
- Multiple-processor "Hello, world!"
- "Hello, world!" with MPC_Printf
- Printing hostname of the machine executing host-process
- Printing hostnames of all machines
- Number of processes
- Getting number of processes with the MPC_Total_nodes function
- Using simple assignment

Single-processor "Hello, world!"

To begin with let us consider the simplest program – p1.mpc – which does the same as the most popular C program known under the name "Hello, world!"

```
#include <stdio.h>
int [*]main() {
  [host]printf("Hello, world.\n");
  return 0;
}
```

Figure 1. Text of the program p1.mpc

The code of this mpC program differs very little from the code of the C program. The first difference is the **[*]** specifier before the name **main** in the definition of the main function. The **[*]** specifier before function name in function definition says that all processes of the parallel program shall execute the code of the function. The function, code of which all processes of the parallel program execute, is called ***basic function*** in mpC. A basic function can be called to from other basic function only.

The second difference is the construct **[host]** before **printf** function call. Unlike the function **main**, the function **printf** does not need to be called in parallel by all processes of the parallel program. Moreover, the function **printf** can be called to from any single process of parallel program. In mpC such functions are called ***nodal.*** Also a nodal function can be called to in parallel from any process of any group of processes of parallel program. In the program p1.mpc only the process associated

with the terminal, from which the program was launched, executes the function **printf.** The keyword **host** is associated with this process. So all the processes of the program pr1 except host-process do nothing. The host-process outputs "Hello world!" to terminal.

Multiple-processor "Hello, world!"

```
#include <stdio.h>
int [*]main() {
  printf("Hello, world.\n");
  return 0;
}
```

Figure 2. Text of the program p2.mpc

Syntax of the program p2.mpc differs even less than syntax of p1.mpc from syntax of the C "Hello, world!" program. Nevertheless, this program is more parallel in a sense than p1.mpc. In p2.mpc all the processes of the parallel program call to **printf** function. The result of execution of the program depends on the operating environment. In some environments, standard output of all the parallel processes will go to the user's terminal from which the program has been started. In that case, the user will see as many greetings "Hello, world!" on the terminal as many parallel processes the program will consist of - one greeting from each process. In other environments, you will see greetings only from the processes that run on the same computer as the host-process or even a single greeting from the host-process.

"Hello, world!" with MPC_Printf

```
#include <mpc.h>
int [*]main() {
  MPC_Printf("Hello, world.\n");
  return 0;
}
```

Figure 3. Text of the program p3.mpc

The main disadvantage of the program p1.mpc is that the result of its execution may differ in different environments. This means that the program is not portable. This is a serious disadvantage of the program. The program p3.mpc, which outputs "Hello, world!" to the user's terminal from all processes of the parallel program, is free of this disadvantage. The mpC library function **MPC_Printf** guarantees that each process calling this function will output "Hello, world!" to the user's terminal only.

Printing hostname of the machine executing host-process

It’s often useful to know the hostname of the computer, which is executing some process of a parallel program. The program p4.mpc demonstrates how to print to terminal the name of the computer, on which host-process is executed. In addition to "Hello, world!" the host-process outputs the name of the computer, on which this process is being executed.

```
#include <stdio.h>
#include <mpc.h>
int [*]main() {
  [host]printf("Hello world!
    Host-process runs on \"%s\".\n",
    MPC_Get_processor_name());
  return 0;
}
```

Figure 4. Text of the program p4.mpc

MPC_Get_processor_name is mpC library function. It's declared as follows:

```
char *MPC_Get_processor_name (void);
```

Its return value is the name of the computer on which the process of the parallel program calling to this function is being executed.

Printing hostnames of all machines

```
#include <stdio.h>
#include <mpc.h>
int [*]main() {
  char *host_name;
  host_name = MPC_Get_processor_name();
  MPC_Printf ("Hello world!
    Host-process runs on \"%s\".\n",
    host_name);
  return 0;
}
```

Figure 5. Text of the program p5.mpc

The next program, p5.mpc, sends to the user's terminal richer in content messages from all processes of the parallel program. In addition to the greeting "Hello, world!" each process outputs the name of the computer, on which the process is running. To do this, so-called ***distributed*** variable **host_name** is defined in the program.

The variable is called distributed because each process of the parallel program holds in its memory a copy of the variable, and, hence, the region of storage, represented by this variable, is distributed over the processes. Thus, the distributed variable **host_name** is no more than a set of normal (undistributed) variables and each of these variables is in turn a projection of the distributed variable onto the corresponding process.

After each process of the parallel program p5.mpc calls to the function **MPC_Get_processor_name**, the corresponding projection of the distributed variable **host_name** will point to the string of characters containing the name of the computer running this process.

Values of distributed variables and distributed expressions, such as **host_name**, are distributed over processes of the parallel program in the natural way and called ***distributed values***.

Number of processes

```
#include <stdio.h>
#include <mpc.h>
int [*]main() {
  char *host_name;
  repl int one;
  repl int number_of_processes;
  host_name = MPC_Get_processor_name();
  one = 1;
  number_of_processes = one[+];
  MPC_Printf("Hello world! I'm one of %d processes"
             "and run on \"%s\".\n",
             number_of_processes, host_name);
  return 0;
}
```

Figure 6. Text of the program p6.mpc

The program p6.mpc extends the output of the program p5.mpc with information about the total number of processes of the parallel program. To do it, the program defines two integer distributed variables **one** and **number_of_processes**. First, all projections of the variable **one** are assigned **1**. The result of applying the postfix operator **[+]** to the variable **one** will be a *distributed value* the projection of which to any process will be equal to the sum of values of all projections of the variable **one**. In other words, the projection of the value of the expression **one[+]** to any process of the parallel program will be equal to the total number of processes. After assigning the distributed value to the distributed variable **number_of_processes**, all projections of the latter will hold the same value, namely, the total number of processes of the parallel program.

The definition of the distributed variable **one** contains the keyword **repl** (an abbreviation stands for *replicated*). It informs the compiler that all projections of the value of the variable shall be equal to each other in any expression of the program. Such distributed variables are called ***replicated*** in mpC (correspondingly, the value of a replicated variable is called replicated value). Replicated variables and expressions play an important role in mpC. The mpC compiler assumes that, at any point in the program, all projections of a value of a **repl** variable equal to each other. The compiler warns about all possible violations of this rule.

Getting number of processes with the MPC_Total_nodes function

```
#include <stdio.h>
#include <mpc.h>
int [*]main() {
  char *host_name;
  host_name = MPC_Get_processor_name();
  MPC_Printf("Hello world! I'm one of %d processes"
             "and run on \"%s\".\n",
              MPC_Total_nodes (), host_name);
  return 0;
}
```

Figure 7. Text of the program p7.mpc

Note that the more simple than p6.mpc program, p7.mpc, produces the same result as the program p6.mpc by means of use of the mpC library function

MPC_Total_nodes. The function **MPC_Total_nodes** returns the total number of processes of the parallel program. Besides, the program p7.mpc is more efficient than the program p6.mpc, because unlike execution of the expression **one[+]** a parallel call to the function **MPC_Total_nodes** does not need data transfer between processes of the program.

Using simple assignment

```
#include <mpc.h>
int [*]main() {
  int [host]local_one;
  repl int one;
  repl int number_of_processes;
  char *host_name;
  host_name = MPC_Get_processor_name();
  local_one = 1;
  one = local_one;
  number_of_processes = one[+];
  MPC_Printf("Hello world! I'm one of %d processes"
             "and run on \"%s\".\n",
              number_of_processes, host_name);
}
```

Figure 8. Text of the program p8.mpc

The program p8.mpc is slightly modified program p6.mpc. It's less efficient than p6.mpc, but it demonstrates how assignment can be used for transferring data between processes in mpC. The variable **local_one** belongs to the host-process and is initialized by **1**. The variable **one** is replicated over all the processes of the parallel program. Execution of the assignment **one=local_one** consists in broadcasting the value of the variable **local_one** to all processes of the program followed by its assigning to the projections of the variable **one**.

Networks

Networks

It isn’t rare that the number of processes involved in parallel solution of the problem depends on the problem itself or/and the parallel algorithm of its solution and is determined by input data. For example, let a single process be used for a single group of bodies when simulating the evolution of N groups of bodies under the influence of Newtonian gravitational attraction. It means that exactly N processes will have to be involved in the corresponding parallel computations independent of the total number of processes constituting the parallel program. Remember that the parallel program is started up from operational environment. It means that mpC programmer can’t define number of processes of parallel program in implementation time – number of processes of parallel program can be retrieved at runtime only.

This chapter covers the following topics:

- "Hello, world!" on a network
- Using distribution labels
- Changing number of network nodes at runtime
- Using automatic networks
- Using static networks

"Hello, world!" on a network

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

Figure 9. Text of the program p9.mpc

The program p9.mpc gives the first introduction to the language means that allow the programmer to describe parallel computations on the needed number of processes. The computations themselves remain quite simple - each of participating processes just outputs "Hello, world!" to the user's terminal. But the number of participating processes, N=3, is defined by the programmer and does not depend on the total number of processes constituting the parallel program.

In mpC, the notion of ***network*** corresponds to a group of processes jointly performing some parallel computations. In mpC, network is an abstraction facilitating the work with actual processes of the parallel program (just like the notion of data object and variable in programming languages facilitate the work with memory).

In the simplest case, the network is simply a set of ***virtual processors***. To code computations executed by a given number of parallel processes, first of all the programmer should define a network consisting of the corresponding number of

virtual processors, and only after the network is defined, the programmer can start describing parallel computations on the network.

Definition of the network causes creation of a group of processes representing the network, so each virtual processor is represented by a single process of the parallel program. Description of parallel computations on the network causes execution of the corresponding computations just on those processes that represent virtual processors of the network. The important difference of actual processes from virtual processors is that at different moments of program execution the same process can represent different virtual processors of different mpC networks. In other words, definition of the network causes mapping of virtual processors of this network to actual processes of the parallel program, and this mapping is constant during lifetime of the network.

So, the program p9.mpc first defines the network **mynet** of **N** virtual processors and then calls the nodal 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. If the programming system cannot perform such mapping (for example, if **N** is greater than the total number of processes of the program), the program stops abnormally with the corresponding diagnostics.

Note the similarity of language constructs **[mynet]** and **[host]**. Indeed, the keyword **host** can be considered as the name of the pre-defined network consisting exactly of one virtual processor mapped to the host-process associated with the user's terminal.

Using distribution labels

```
#include <mpc.h>
#define N 3
int [*]main() {
  net SimpleNet(N) mynet;
  [mynet]:{
    char *host_name;
    host_name = MPC_Get_processor_name();
    MPC_Printf("Hello world!
      I'm on \"%s\".\n", hostname);
  }
}
```

Figure 10. Text of the program p10.mpc

The program p10.mpc outputs messages from those processes of the parallel program to which the virtual processes of the network **mynet** are mapped. In addition to "Hello, world!", each involved process outputs the name of the computer executing the process. To do so, in the program there defined the variable **host_name** *distributed over network* **mynet**. Only these processes, onto which the virtual processors of **mynet** are mapped, hold in memory a copy of **host_name**. Only these processes call to the function **MPC_Get_processor_name**, and after this call each projection of the distributed variable **host_name** will contain a pointer to the name of the computer running the corresponding process. To specify the region of the computing space, on which the set of expressions will be executed, a special

distribution label, **[mynet]**, was used. Any statement labeled with such a label (in our case, this is a compound statement) will be completely executed by virtual processors of the corresponding network.

Changing number of network nodes at runtime

```
#include <stdio.h>
#include <stdlib.h>
#include <mpc.h>
int [*]main(int [host]argc, char **[host]argv) {
  repl n;
  if(argc<2)
    n = 1;
  else
    n = [host]atoi(argv[1]);
  if(n<1)
    [host]printf("Wrong input
      (%d processes required).\n",
      [host]n);
  else if(n>MPC_Total_nodes())
    [host]printf("Required too many processes
      (%d against %d available).\n",
      [host]n, [host]MPC_Total_nodes());
  else {
    net SimpleNet(n) mynet;
    char *[mynet]host_name;
    [mynet]:{host_name = MPC_Get_processor_name();}
    [mynet]MPC_Printf("Hello world!
      I'm on \"%s\".\n", host_name);
  }
  MPC_Printf("* ");
}
```

Figure 11. Text of the program p11.mpc

The next program, p11.mpc, demonstrates that the number of virtual processors of the network can be specified dynamically, that is, at runtime. This program treats its only external argument as such a number. The argument is specified by the user when starting up the program and is accessible at least to the host-process. The expression **[host]atoi(argv[1])** is calculated by the host-process and then assigned to the integer variable **n** replicated over all processes of the parallel program. Execution of this assignment consists in broadcasting the calculated value to all processes of the program followed by its assignment to projections of the variable **n**. Before defining (creating) a network of the user-specified number of virtual processors and executing the described computations on the network, the program checks correctness of input data. If the specified number of virtual processors is incorrect (less than 1 or greater than the total number of processes of the parallel program), the program outputs to the corresponding diagnostics. Otherwise, the program defines the network **mynet** consisting of **n** virtual processors as well as the variable **host_name** distributed over the network. After the **MPC_Get_processor_name** function is called each projection of the distributed variable **host_name** points to the string of characters containing the name of the computer running the corresponding process. Finally, the call of the nodal function **MPC_Printf** on the network **mynet** outputs "Hello, world!" from each virtual processor together with the name of the computer hosting the virtual processor.

Duration of both the network **mynet** and the variable **host_name** 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***.

Using automatic networks

```
#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]: {
      char *host_name;
      host_name = MPC_Get_processor_name();
      MPC_Printf("I'm from an automatic network on
        \"%s\" (n=%d).\n", host_name, n);
    }
  }
}
```

Figure 12. Text of the program p12.mpc

During execution of the program p12.mpc, at the first loop iteration (**n=Nmin=3**) a network of 3 virtual processors is created at 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 4 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 initialization of the loop (execution of the expression **n++**), the 4-processor network no longer exists. Finally, at the last iteration an automatic network of 5 virtual processors (**n=Nmax=5**) is created on the entry into the block.

Using static networks

```
#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]: {
      char *host_name;
      host_name = MPC_Get_processor_name();
      MPC_Printf("I'm from an automatic network on
        \"%s\" (n=%d).\n", host_name, n);
    }
  }
}
```

Figure 13. Text of the program p13.mpc

During execution of the program p13.mpc, at the first loop iteration a network of 3 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, in which it's visible. Therefore, at the time of repeated initialization 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 networks are created but the static network, which has been created on the first entry into the block, becomes visible and the number of virtual processors remains unchanged, i.e equals 3.

Thus, while in the program p12.mpc the same name **anet** denotes absolutely different networks at different loop iterations, in the program p13.mpc 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.

Network type

Network type

By now in our mpC programs all virtual processors of the same network have performed the same computations. Therefore, we did not need to separate different virtual processors inside the network. But if a parallel algorithm that should be coded implies different processes to execute differing computations, some means to separate some virtual processors inside the network are needed. The mpC language provides the programmer with the relevant means. In particular, it allows the programmer to associate the virtual processors of any network with a *coordinate system* and to separate a single virtual processor specifying its coordinates.

Generally speaking, in mpC one cannot just 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 all examples that have been considered the definition of the used 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 *parameterized* 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 parameterized network type that describes networks consisting of **n** virtual processors well ordered by their positions on the coordinate line.

This chapter covers the following topics:

- One-dimensional network
- Two-dimensional network

One-dimensional network

```
#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");
    else
      MPC_Printf("Hello, odd world!\n");
  }
}
```

Figure 14. Text of the program p14.mpc

The program p14.mpc is slightly modified p9.mpc. It gives an example of code of a program that executes different computations on different virtual processors. The program uses the binary operator **coordof** with the coordinate variable **I** and the network **mynet** as its left and right operands respectively. The result is the 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 the network **mynet**. 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!" and virtual processors with odd coordinates will output "Hello, odd world!".

Two-dimensional network

```
#include <mpc.h>
nettype Mesh(int m, int n) {
  coord I=m, J=n;
};
#define M 2
#define N 3
int [*]main() {
  net Mesh(M,N) mynet;
  [mynet]: {
    char *host_name;
    host_name = MPC_Get_processor_name();
    MPC_Printf("I'm on \"%s\" and have
      coordinates (%d, %d).\n", host_name,
      I coordof host_name, J coordof host_name);
  }
}
```

Figure 15. Text of the program p15.mpc

The program p15.mpc demonstrates a network whose virtual processors are associated with a 2-coordinate system. Each virtual processor of the network outputs its coordinates and the name of the computer hosting it. Note that although the variable **host_name** is not a network it is used as the second operand of the operator **coordof**. In general, if the second operand of the operator **coordof** is an expression not a network, the expression is not evaluated but only used to determine the network that the expression is distributed over, and the operator is executed as if that network was its second operand.

Network parent

Network parent

We have discussed that duration 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 mpC networks are not absolutely independent of 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 nothing but the virtual host-processor. The solution is obvious because at any moment of program execution the existence of only one network can be guaranteed, namely, the pre-defined network **host** consisting of the only virtual processor always mapping onto the host-process associated with the user's terminal. Let us consider several examples of use of network parent. This chapter gives two examples of use of network parent:

- Specifying network parent
- Multiple networks with different parents

Specifying network parent

```
#include <mpc.h>
nettype Mesh(int m, int n) {
  coord I=m, J=n;
  parent [0,0];
};
#define M 2
#define N 3
int [*]main() {
  net Mesh(M,N) [host]mynet;
  [mynet]: {
    char *host_name;
    host_name = MPC_Get_processor_name();
    MPC_Printf("I'm on \"%s\" and
      have coordinates (%d, %d).\n",
      host_name, I coordof host_name, J coordof host_name);
  }
}
```

Figure 16. Text of the program p16.mpc

The program p16.mpc is completely equivalent to the program p15.mpc except that in the definition of the network its parent is specified explicitly.

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

Multiple networks with different parents

By now at any moment of mpC program execution there existed not more than one network. This is not a restriction of the mpC language. The mpC language allows the programmer to write programs with arbitrary number of simultaneously existing (and visible) networks. The only limitation is the total number of processes constituting the parallel program.

```
#include <mpc.h>
nettype TrivialNet(int n, int m) {
  coord I=n;
  parent [m];
};
int [*]main() {
  [host]MPC_Printf("I'm host. I'll be a parent of net1.\n\n");
  {
    net TrivialNet(3,0) net1;
    int [net1]mycoordinnet1;
    mycoordinnet1 = I coordof net1;
  [net1]:
    if(mycoordinnet1)
      MPC_Printf("I'm a regular member of net1. "
                 "My coordinate in net1 is %d.\n"
                 " I'll be a parent of net%d.\n\n",
                  mycoordinnet1, mycoordinnet1+1);
    else
      MPC_Printf("I'm a parent of net1. "
        "My coordinate in net1 is %d.\n\n",
        mycoordinnet1);
    {
        net TrivialNet(3,1) [net1:I==1]net2;
        net TrivialNet(3,2) [net1:I==2]net3;
        [net2]: {
          int mycoordinnet2;
          mycoordinnet2 = I coordof net2;
          if(mycoordinnet2!=1)
            MPC_Printf("I'm a regular member of net2. "
              "My coordinate in net2 is %d.\n\n",
              mycoordinnet2);
          else
            MPC_Printf("I'm a parent of net2. "
              "My coordinate in net2 is %d.\n\n",
              mycoordinnet2);
        }
        [net3]: {
          int mycoordinnet3;
          mycoordinnet3 = I coordof net3;
          if(mycoordinnet3!=2)
```

```
            MPC_Printf("I'm a regular member of net3. "
              "My coordinate in net3 is %d.\n\n",
              mycoordinnet3);
          else
            MPC_Printf("I'm a parent of net3. "
              "My coordinate in net3 is %d.\n\n",
              mycoordinnet3);
        }
    }
  [net1]:
    if(mycoordinnet1)
      MPC_Printf("I'm a regular member of net1. "
        "My coordinate in net1 is %d.\n"
        " I was a parent of net%d.\n\n",
        mycoordinnet1, mycoordinnet1+1);
  }
  [host]MPC_Printf("I'm host. I was a parent of net1.\n\n");
}
```

Figure 17. Text of the program p17.mpc

In program p17.mpc, there simultaneously exist three networks - **net1**, **net2** and **net3**. The parent of the network **net1** is the virtual host-processor. The parent of the network **net2** is the virtual processor of the network **net1** with coordinate 1. The parent of the network **net3** is the virtual processor of the network **net1** with coordinate 2.

Synchronization of processes

Synchronization of processes

We have already mentioned that the parallel program is a set of parallel processes synchronizing their work and interchanging data by means of message passing. The means of the mpC language that have been introduced allow the programmer to specify the number of processes needed for parallel solution of the problem as well as to distribute computations among the processes. In principle, the same means is enough for synchronization of the processes during execution of the parallel program.

The basic synchronization mechanism for parallel processes interacting via message passing is a *barrier*. The barrier is a point of the parallel program where a process waits for all the other processes with which it synchronizes its work. Only after all the processes synchronizing their work reach the barrier they can continue further computations. If by some reason even one of the processes does not reach the barrier, all the other processes will "hang" at this point of the program and the program itself will never stop normally. There can be a number of ways to "hang" at some point of program execution. In the following topics there described three of them:

- Using assignment as barrier
- Using reduction as barrier
- Using the MPC_Global_barrier function

Using assignment as barrier

```
#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");
    Barrier:
      {
        int [host]bs[N], b=1;
        bs[] = b;
        b = bs[];
      }
    if(my_coordinate%2==1)
      MPC_Printf("Hello, odd world!\n");
  }
}
```

Figure 18. Text of the program p18.mpc

Let we want to change the program p18.mpc in such a way that messages from virtual processors with odd coordinates come to the user's terminal only after messages from virtual processors with even coordinates. The program p18.mpc solves the problem as follows. In the block labeled by the label **Barrier**, the array **bs**

located on the virtual host-process and the variable **b** distributed over the network **mynet** are defined. The number of elements in the array **bs** is equal to the number of virtual processors in the network **mynet**. When the assignment **bs[]=b** is executed the following tasks are performed. Each virtual processor of the network **mynet** sends the value of its projection of the variable **b** to the virtual host-processor where this value is assigned to the element of the array **bs** which index is equal to the coordinate **I** of the virtual processor. When the assignment **b=bs[]** is executed there performed sending of the value of the **i-** th element of **bs** to the virtual processor of **mynet** with coordinate **I=i** where this value is assigned to the corresponding projection of **b**. One can see that none of the virtual processors of **mynet** leaves the block marked "**Barrier**" until all the processes enter the block and end the execution of the first assignment. Only after each processor has sent its projection of the variable **b,** the virtual host-processor will be able to end execution of the first assignment and start execution of the second one freeing one by one all the other virtual processors that were suspended at the point. Thus, this block is nothing but a barrier serializing statements that output messages from even and odd virtual processors of the network **mynet** correspondingly.

Using reduction as barrier

```
#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");
    Barrier:
      {
        int b=1;
        b[+];
      }
    if(my_coordinate%2==1)
      MPC_Printf("Hello, odd world!\n");
  }
}
```

Figure 19. Text of the program p19.mpc

In the program p19.mpc a similar barrier is implemented simpler and more concise. At the same time, its efficiency is hardly worse, because the most obvious implementation of the operator **[+]** only differs from the barrier in the program p18.mpc by additional summing on the host-processor. The time of this summing is neglible short as compared to the time of data transfer.

```
#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)
    MPC_Printf("Hello, even world!\n");
  MPC_Global_barrier();
  if(my_coordinate%2==1)
    MPC_Printf("Hello, odd world!\n");
}
```

Figure 20. Text of the program p20.mpc

Actually, the programmer does not need to invent different ways to implement barriers. The mpC language provides two library functions efficiently implementing barrier synchronization for the operating environment where the parallel program runs. Firstly, it is the basic function **MPC_Global_barrier** synchronizing the work of all processes of the parallel program. Its declaration is in the header **mpc.h** and looks as follows:

```
int [*]MPC_Global_barrier(void);
```

The program p20.mpc demonstrates the use of this function for serialization of the two already known parallel statements. Note that unlike two previous programs, in this program not only processes implementing the network **mynet** but also all free processes participate in the barrier synchronization. Naturally, this implies greater number of messages transferring during execution of the barrier and hence some slowing down of the program execution.

Network functions

Network functions

Network function is called and executed on some network or hard subnetwork, and its arguments and value (if any) is also distributed over this region of the computing space. In the following topics there are given examples of use of such functions:

- Using the MPC_Barrier function
- User defined network function
- Call to network functions on different networks

Using the MPC_Barrier function

```
#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");
  }
}
```

Figure 21. Text of the program p21.mpc

The library function **MPC_Barrier** allows synchronization of the work of the virtual processors of any network. The program p21.mpc demonstrates the use of this function. In this program, like in programs p18.mpc and p19.mpc, only the processes implementing the network **mynet** participate in the barrier synchronization.

In the program p21.mpc the call to the function **MPC_Barrier** looks a little bit unusual. Indeed, this function principally differs from all functions we have met before and represents ***network*** functions. Unlike basic functions that 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 cam 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 it 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 enclosed in square 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 normal 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 normal formal parameter declared as follows:

```
repl const int n
```

All normal 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.

User defined network function

```
#include <mpc.h>
#define N 5
int [net SimpleNet(n) w] MPC_Barrier( void ) {
       int [w:parent]bs[n], [w]b=1;
       bs[]=b;
       b=bs[];
}
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");
  }
}
```

Figure 22. Text of the program p22.mpc

If the function **MPC_Barrier** were not a library one, it could be defined as it is done in the program p22.mpc:

```
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 there is defined the automatic array **bs** of **n** elements (the mpC language supports *dynamic arrays*). 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**.

Call to network functions on different networks

```
#include <mpc.h>
nettype Mesh(int m, int n) {
  coord I=m, J=n;
  parent [0,0];
};
#define M 2
#define N 3
int [*]main() {
  net Mesh(M,N) [host]mynet;
  [mynet]: {
    char *host_name;
    host_name = MPC_Get_processor_name();
    if ( I coordof host_name == 0 )
      MPC_Printf("I'm on \"%s\" and have
        coordinates (%d, %d).\n", host_name,
        0, J coordof host_name);
    ([(M*N)mynet])MPC_Barrier();
    if ( I coordof host_name!= 0 )
      MPC_Printf("I'm on \"%s\" and have
        coordinates (%d, %d).\n", host_name,
        I coordof host_name, J coordof host_name);
  }
}
```

Figure 23. Text of the program p23.mpc

In the programs p21.mpc and p22.mpc, calls to the network function **MPC_Barrier** pass 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 and not only **SimpleNet** type can be passed to this function as an actual network parameter. In the program p23.mpc a network of the type **Mesh** is such an actual parameter. In fact, the function **MPC_Barrier** treats the group of processes on which it is called as a network of the type **SimpleNet**.

In general, the actual network parameter can be of the 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. The program p24.mpc is an example of the situation when the actual values of the network type passed to the network function are not the only possible ones.

```
#include <mpc.h>
#define N 6
nettype Mesh(int m, int n) {
  coord I=m, J=n;
  parent [0,0];
};
int [net Mesh (m, n) w] My_Barrier( void ) {
       int [w:parent]bs[n], [w]b=1;
       bs[]=b;
       b=bs[];
}
```

```
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");
  ([(1,N)mynet])My_Barrier();
  if(my_coordinate%2==1)
    [mynet]MPC_Printf("Hello, odd world!\n");
}
```

Figure 24. Text of the program p24.mpc

Subnetworks

Subnetworks

Let us recall again that the parallel program is a set of parallel processes synchronizing their work and interchanging data by means of message passing. The means of the mpC language that have been introduced allow the programmer to specify the number of processes needed for parallel solution of the problem, distribute computations among the processes as well as synchronize their work during execution of the parallel program. But the means are obviously not sufficient for specification of data transfer among processes.

Indeed, so far either all processes of the parallel program or all virtual processors of one or another 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. To describe more complicated data transfer, for example, data transfer between groups of virtual processors of the network or parallel data exchange between neighboring virtual processors of the network the introduced language means are not sufficient.

The basic means of the mpC language for describing complicated data transfers are ***subnetworks***. Any subset of the virtual processors of a network is a subnetwork of this network. The following topics give the examples of how to use subnetworks:

- Hard subnetworks
- Flexible subnetworks
- Distributed statements

Hard subnetworks

```
#include <string.h>
#include <mpc.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]: {
    char me[MAXLEN], neighbour[MAXLEN];
    subnet [mynet:I==0]row0, [mynet:I==1]row1;
    char *host_name;
    host_name = MPC_Get_processor_name();
    strcpy(me, host_name);
    [row0]neighbour[] = [row1]me[];
    [row1]neighbour[] = [row0]me[];
    MPC_Printf("I'm on \"%s\" and have
      coordinates (%d, %d),\n"
      "My neighbour with coordinates
      (%d, %d) is on \"%s\".\n\n",
      me, I coordof mynet, J coordof mynet,
      (I coordof mynet + 1)%2, J coordof mynet,
      neighbour);
```

```
  }
}
```

Figure 25. Text of the program p25.mpc

In the program p25.mpc each virtual processor of the network **mynet** having the 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 neighboring 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.

When the assignment **[row0]neighbour[]=[row1]me[]** is executed there performed parallel transferring of 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 networks **row0** and **row1** are *hard* subnetworks. There are two kinds of subnetworks in mpC – *hard* and *flexible*. Creation of a hard subnetwork is much more time-consuming than that of a flexible network. Network functions can be called on hard subnetworks only.

Flexible subnetworks

As it was mentioned in the previous topic there are two kinds of subnetworks in mpC – *hard* and *flexible*. If it isn’t necessary to call some network function on a subnetwork then for better performance it’s efficient to declare a flexible subnetwork. Flexible network can be declared either explicitly by using the specifier *flex* in subnetwork declaration or implicitly. The program p26.mpc demonstrates how flexible network can be declared implicitly.

```
#include <string.h>
#include <mpc.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]: {
    char me[MAXLEN], neighbour[MAXLEN];
    char *host_name;
    host_name = MPC_Get_processor_name();
    strcpy(me, host_name);
    [mynet:I==0]neighbour[] = [mynet:I==1]me[];
    [mynet:I==1]neighbour[] = [mynet:I==0]me[];
    MPC_Printf("I'm on \"%s\" and have
      coordinates (%d, %d),\n"
      "My neighbour with coordinates
      (%d, %d) is on \"%s\".\n\n",
      me, I coordof mynet, J coordof mynet,
      (I coordof mynet + 1)%2, J coordof mynet,
      neighbour);
  }
}
```

Figure 26. Text of the program p26.mpc

The distribution label **[mynet:I==0]** before left or right operand of the simple assignment means that only these projections of the distributed variables **me** or **neighbour**, which belong to the virtual processors satisfying the predicate **I==0**, take part in communication. In fact, using the distribution label **[mynet:I==0]** we implicitly declare flexible subnet. The program p27.mpc is almost equivalent to the program p26.mpc. The only difference between them is that in the program p27.mpc flexible subnets are declared explicitly. Since creation of flexible subnetworks is in many respects cheaper than that of hard subnetworks, you should use flexible subnetworks if possible.

```
#include <string.h>
#include <mpc.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]: {
    char me[MAXLEN], neighbour[MAXLEN];
    char *host_name;
    host_name = MPC_Get_processor_name();
    strcpy(me, host_name);
    {
      flex subnet [mynet:I==0]row0, [mynet:I==1]row1;
      [row0]neighbour[] = [row1]me[];
      [row1]neighbour[] = [row0]me[];
    }
    MPC_Printf("I'm on \"%s\" and have
      coordinates (%d, %d),\n"
      "My neighbour with coordinates
      (%d, %d) is on \"%s\".\n\n",
      me, I coordof mynet, J coordof mynet,
      (I coordof mynet + 1)%2, J coordof mynet,
      neighbour);
  }
}
```

Figure 27. Text of the program p27.mpc

Statements of a C program control the flow of program execution. In C several kinds of statements are available to perform loops, to select other statements to be executed, and to transfer control. In mpC the notion of distributed statement is introduced. Almost all of C statements can be labeled with a distribution specifier. For example, if the *if* statement is labeled with a distribution specifier then the statement body is executed by every node belonging to the region of computing space specified by the distribution specifier provided the controlling expression evaluated on this node is nonzero. The program p28.mpc illustrates use of distributed *if* statement.

```
#include <mpc.h>
#define N 13
int [*]main() {
  net SimpleNet(N) mynet;
  subnet [mynet:I>0] mysubnet;
  [mysubnet]:if (I coordof mysubnet < 7)
    MPC_Printf("Hello, world! I'm \"%s\".
      My id is %d<7. \n",
      MPC_Get_processor_name(),
      I coordof mysubnet);
  else
    MPC_Printf("Hello, world! I'm \"%s\".
      My id is %d>=7. \n",
      MPC_Get_processor_name(),
      I coordof mysubnet);
  return 0;
}
```

Figure 28. Text of the program p28.mpc

In this program the control expression *if …else* is distributed over the subnet **mysubnet**. It means that coordinates of all the processes, of which the parallel program consists, are more than 0. If there reside communications in the body of control statement then the value of the controlling expression must be the same for all the processes belonging to the region of computing space, over which the control expression is distributed. This rule is applied to loop statements. The program p29.mpc illustrates the use of distributed loop statement.

```
#include <mpc.h>
#define N 13
int [*]main() {
  net SimpleNet(N) mynet;
  subnet [mynet:I>0] mysubnet;
  repl int [mysubnet]fl = 0, [mysubnet]i = 0;
  [mysubnet]:do
  {
    if (i > 10) fl = 1;
    i ++;
  } while (!fl);
  return 0;
}
```

Figure 30. Text of the program p29.mpc

Vector computations

Vector computations

Like other general-purpose parallel programming language, HPF (High Performance Fortran) for example, mpC extends most of scalar operators and allows their operands to be expressions designating sets of scalars, say, arrays or array segments as well. Thus the mpC language provides facilities to manipulate arrays and to perform vector computations. In the programs p25.mpc and p26.mpc expressions **neighbour[]** and **me[]** designate arrays **neighbour** and **me** not as pointers to their initial elements but as a whole. The assignment **neighbour[]=me[]** means assigning the value of the expression **me[]**, which is a vector of values of the type **char**, to the array **neighbour**. In the following topics there are given examples of using mpC vector expressions:

- Using prefix reduction operator [+]
- Using grids

Using prefix reduction operator [+]

```
#include <stdio.h>
#include <mpc.h>
void init_vector(int n, double (*vec)[n]) {
  static double base=1.0;
  int i;
  for(i=0; i<n; i++)
    (*vec)[i] = base;
  base++;
}
int [*]main(int [host]argc, char *[host]argv[]) {
  [host]: {
    int n;
    n = atoi(argv[1]);
    if(n<=0)
      printf("Wrong input: n = %d\n", n);
    else {
      double v1[n], v2[n];
      init_vector(n, &v1);
      init_vector(n, &v2);
      printf("Dot product of the vectors
        is equal to %g\n", [+](v1[]*v2[]));
    }
  }
}
```

Figure 30. Text of the program p30.mpc

In the program p30.mpc a single expression **[+](v1[]*v2[])** describes computation of dot product of two vectors contained in arrays **v1** and **v2**. The execution of the vector binary operator ***** consists in element-wise multiplication of its vector operands, and the result of the prefix unary operator **[+]** is the sum of elements of its vector operand. The following fragment of code written in pure C

```
int i;
double sum = 0.;
for (i = 0; i < n; i ++) sum += v1 [i] * v2[i];
```

is equivalent to the mpC expression **sum = [+](v1[]*v2[]);**. You can see that mpC notation of dot product of two vectors is shorter than C notation of the same dot product.

Using grids

```
#include <stdio.h>
#include <stdlib.h>
#include <mpc.h>
void init_matrix(int n, double (*a)[n][n]) {
  int i,j;
  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
      (*a)[i][j]=10.;
  for(i=0;i<n;i++)
    (*a)[i][i]=10.*n;
}
void print_matrix(int n, double (*a)[n][n]) {
 int i,j;
  for(i=0;i<n;i++) {
    for(j=0;j<n;j++)
      printf("%lf\t",(*a)[i][j]);
  printf("\n");
  }
}
int [*]main(int [host]argc, char *[host]argv[]){
  [host]:
    {
      int n, i, j;
      n = atoi(argv[1]);
      if(n<=0)
        printf("Wrong input: n = %d\n", n);
      else {
        double a[n][n];
        init_matrix(n,&a);
        printf("\n Source matrix :\n");
        print_matrix(n,&a);
        for(i=0; i<n-1; i++) {
          for(j=i+1; j<n; j++) {
            double t;
            t = a[j][i] / a[i][i];
            if(a[j][i]!=0)
              a[j][i:n-1] -= t * a[i][i:n-1];
          }
        }
        printf("\n Matrix after LU
          transformation: \n");
        print_matrix(n,&a);
      }
    }
}
```

Figure 31. Text of the program p31.mpc

The program p31.mpc implementing LU decomposition of a square matrix (Gaussian elimination) shows the usage of array segments in vector computations. For example,

the expression **a[i][i:n-1]** designates the segment of the **i**-th row of the array **a** that includes all elements from **a[i][i]** to **a[i][n-1]**.

Heterogeneous parallel computing

Heterogeneous parallel computing

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. Just this differs parallel computing from distributed computing. Therefore it is natural that minimization of the time required for execution of the parallel program is the main goal when mapping virtual processors of the network to actual processes. When 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 of relative volumes of computations, which different virtual processors of the defined network will perform.

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 synchronization 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 the 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, because in this case execution of computations by different processes at the same speed leads to the situation when processes performing small volume computations will wait at synchronization points and points of data transfer for processes executing computations of bigger volumes. In that case, the mapping that ensures speeds of processes to be proportional to volumes of performed computations leads 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 a 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 examples illustrate how the means for specification of relative volumes of computations can be used:

- Calculation of the mass of a metallic construction
- Using the recon statement

Calculation of the mass of a metallic construction

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpc.h>
#define DELTA (0.5)
typedef struct {
          double len;
          double wid;
          double hei;
          double mass;
} rail;
nettype HeteroNet(int n, double v[n]) {
  coord I=n;
  node {
    I>=0: v[I];
  };
  parent [0];
};
double Density(double x, double y, double z) {
  return 6.0 * sqrt( exp( sin( sqrt( x*y*z ) ) ) );
}
int [*]main(int [host]argc, char **[host]argv) {
  repl N=3;
  if(argc>1)
    N = [host]atoi(argv[1]);
  if(N>0) {
    static rail [host]steel_hedgehog[[host]N];
    repl double volumes[N], [host]start;
    int [host]i;
    repl j;
    for(i=0; i<[host]N; i++) {
      steel_hedgehog[i].len = 200.0*(i+1);
      steel_hedgehog[i].wid = 5.0*(i+1);
      steel_hedgehog[i].hei = 10.0*(i+1);
    }
    start = [host]MPC_Wtime();
    for(j=0; j<N; j++)
      volumes[j] =
          steel_hedgehog[j].len *
          steel_hedgehog[j].wid *
          steel_hedgehog[j].hei;
    {
      net HeteroNet(N, volumes) mynet;
      [mynet]: {
        rail myrail;
        double x, y, z;
        myrail = steel_hedgehog[];
        for(myrail.mass=0., x=0.;
          x<myrail.len; x+=DELTA)
```

```
          for(y=0.; y<myrail.wid; y+=DELTA)
            for(z=0.; z<myrail.hei; z+=DELTA)
              myrail.mass += Density(x,y,z);
        myrail.mass *= DELTA*DELTA*DELTA;
        MPC_Printf("Rail #%d is %gcm x %gcm x%gcm and
          weights %g kg\n", I coordof mynet,
          myrail.len, myrail.wid, myrail.hei,
          myrail.mass/1000.0);
        [host]printf("The steel hedgehog
          weights %g kg\n",
          [host]( (myrail.mass)[+]) / 1000.0 );
      }
    }
    [host]printf("\nIt took %.1f seconds
      to run the program.\n",
      [host]MPC_Wtime() - start);
  }
  else
    [host]printf("Wrong input (N=%d)\n", [host]N);
}
```

Figure 32. Text of the program p32.mpc

The program p32.mpc defines the network type **HeteroNet** parameterized 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 an unusual declaration,

```
node { I>=0: v[I] },
```

saying the following: *for any **I>=0** the relative volume of computations performed by the virtual processor with coordinate **I** is equal to **v[I]**.*

The program p32.mpc 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 the 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.

Along with the results of calculations, the program p32.mpc outputs the wall time elapsed to execute the calculations. To do it, the program uses the library nodal function **MPC_Wtime** that returns the astronomical time in seconds elapsed from some moment in the past not specified but fixed for the process calling the function. Not going into philosophical speculations about relativity of time passing on different processes, note that in spite of its seeming simplicity just the wall time elapsed to solve the problem in parallel starting from data input and until output of results and measured on the host-process is the most objective and interesting for the

end-user temporal characteristics of the program. As a matter of fact, it is minimization of the characteristics that is the main goal of parallel computing.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpc.h>
#define DELTA (0.5)
typedef struct {
          double len;
          double wid;
          double hei;
          double mass;
} rail;
double Density(double x, double y, double z) {
  return 6.0 * sqrt( exp( sin( sqrt( x*y*z ) ) ) );
}
int [*]main(int [host]argc, char **[host]argv) {
  repl N=3;
  if(argc>1)
    N = [host]atoi(argv[1]);
  if(N>0) {
    static rail [host]steel_hedgehog[[host]N];
    double [host]start;
    int [host]i;
    for(i=0; i<[host]N; i++) {
      steel_hedgehog[i].len = 200.0*(i+1);
      steel_hedgehog[i].wid = 5.0*(i+1);
      steel_hedgehog[i].hei = 10.0*(i+1);
    }
    start = [host]MPC_Wtime();
    {
      net SimpleNet(N) mynet;
      [mynet]: {
        rail myrail;
        double x, y, z;
        myrail = steel_hedgehog[];
        for(myrail.mass=0., x=0.;
          x<myrail.len; x+=DELTA)
          for(y=0.; y<myrail.wid; y+=DELTA)
            for(z=0.; z<myrail.hei; z+=DELTA)
              myrail.mass += Density(x,y,z);
        myrail.mass *= DELTA*DELTA*DELTA;
        MPC_Printf("Rail #%d is %gcm x %gcm x%gcm and
          weights %g kg\n", I coordof mynet,
          myrail.len, myrail.wid, myrail.hei,
          myrail.mass/1000.0);
        [host]printf("The steel hedgehog
          weights %g kg\n",
          [host]( (myrail.mass)[+]) / 1000.0 );
      }
    }
    [host]printf("\nIt took %.1f seconds to run
      the program.\n",
      [host]MPC_Wtime() - start);
  }
  else
    [host]printf("Wrong input (N=%d)\n", [host]N);
}
```

Figure 33. Text of the program p33.mpc

The program p33.mpc is equivalent to the program p32.mpc except that it does not specify explicitly relative volumes of computations performed by different virtual

processors of the network **mynet**. Therefore, when mapping the virtual processors to actual processors, the programming system regards that they perform equal volumes of computations. In this case, it leads as a rule to non-optimal mapping and hence to a longer time of solving the problem as compared to the execution time of the program p32.mpc. The slowing down is especially visible in case of heterogeneous networks including processors significantly differing in performance. So, when executing the program p32.mpc it is quite possible that the mass of the biggest rail will be calculated on the weakest processor resulting in multi-fold slowing down as compared to execution of the same calculations by the program p32.mpc, which ensures that the most powerful processor will compute the mass of the biggest rail.

Stating that the mapping of virtual processors to real processors is based on information about performances of the latter, we said nothing about what processor performance was and how the mpC programming system got the information. The issue is not as simple as it may seem at the first glance. Indeed, what does one mean saying that computer **A** is twice as powerful as computer **B**? Strictly speaking, the claim makes little sense if one is not talking about computers of the same architecture and configuration only differing in processor clock rates. Otherwise, the relative performance of computers, that is, the relative speed of executing computations, very essentially depends on what 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 performance of different computers using a pretty wide range of special testing program packages.

Using the recon statement

By default, the mpC programming system uses the same estimation of performances of participating real processors once obtained as a result of execution of a special parallel program during initialization of the system in the certain parallel environment. It was noted in the previous topic that such estimation is very rough and can differ significantly from the real performance demonstrated by the same processors when executing code essentially differing from code of this special testing program. Therefore, the mpC language contains some means that allow the programmer to change the default performance estimation tuning it to the computations that will be really executed by the processors. The program p34.mpc illustrates the use of the means.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpc.h>
#define DELTA (0.5)
typedef struct {
          double len;
          double wid;
          double hei;
          double mass;
} rail;
nettype HeteroNet(int n, double v[n]) {
  coord I=n;
  node {
```

```
    I>=0: v[I];
  };
  parent [0];
};
double Density(double x, double y, double z) {
  return 6.0 * sqrt( exp( sin( sqrt( x*y*z ) ) ) );
}
double RailMass(double len, double wid, double hei, double delta)
{
  double mass, x, y, z;
  for(mass=0., x=0.; x<len; x+=delta)
    for(y=0.; y<wid; y+=delta)
      for(z=0.; z<hei; z+=delta)
        mass += Density(x,y,z);
  return mass*delta*delta*delta;
}
int [*]main(int [host]argc, char **[host]argv) {
  repl N=3;
  if(argc>1)
    N = [host]atoi(argv[1]);
  if(N>0) {
    static rail [host]steel_hedgehog[[host]N];
    repl double volumes[N], [host]start;
    int [host]i;
    repl j;
    for(i=0; i<[host]N; i++) {
      steel_hedgehog[i].len = 200.0*(i+1);
      steel_hedgehog[i].wid = 5.0*(i+1);
      steel_hedgehog[i].hei = 10.0*(i+1);
    }
    start = [host]MPC_Wtime();
    for(j=0; j<N; j++)
      volumes[j] =
          steel_hedgehog[j].len *
          steel_hedgehog[j].wid *
          steel_hedgehog[j].hei;
    recon RailMass(20., 4., 5., 0.5);
        {
      net HeteroNet(N, volumes) mynet;
      [mynet]: {
        rail myrail;
        myrail = steel_hedgehog[];
        myrail.mass =
          RailMass(myrail.len, myrail.wid,
          myrail.hei, DELTA);
        MPC_Printf("Rail #%d is %gcm x %gcm x%gcm and
          weights %g kg\n", I coordof mynet,
          myrail.len, myrail.wid, myrail.hei,
          myrail.mass/1000.0);
        [host]printf("The steel hedgehog
          weights %g kg\n",
          [host]( (myrail.mass)[+]) / 1000.0 );
      }
    }
    [host]printf("\nIt took %.1f seconds
      to run the program.\n",
      [host]MPC_Wtime() - start);
  }
  else
    [host]printf("Wrong input (N=%d)\n", [host]N);
}
```

Figure 34. Text of the program p34.mpc

The program differs from the program p32.mpc mainly by a new statement, **recon**, executed just 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 **RailMass** 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 **RailMass**. 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 refreshing the estimation of processor performances dynamically, at runtime, just before the estimation will be used by the programming system. It is especially important when the parallel computer system executing the mpC program is used for other compuatations as well. In this case, the real performance of processors can dynamically change dependent on their load by other external to the mpC program computations. The use of the **recon** statement allows writing the parallel programs that is sensitive to such dynamic variation of the load 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.

Index

L

M

N

O

P

R

S

T

U

V

W