Managing the Computing Space in the mpC Compiler

Dmitry Arapov Alexey Kalinov Alexey Lastovetsky
Institute for System Programming, Russian Academy of Sciences
25, Bolshaya Kommunisticheskaya str., 109004, Moscow, Russia
lastov@ivann.delta.msk.su

Abstract

The mpC parallel programming language is an ANSI C superset based on the notion of network comprising processor nodes of different types connected with links of different lengths. It allows the user to describe a network topology, create and discard networks, distribute data and computations over networks. The paper describes the implementation of managing networks in the mpC programming environment.

1 Introduction

The mpC language and its programming environment was initially developed to support programming for massively parallel computers, mainly for high-performance distributed memory machines (DMMs). In brief, our motivation for the creation of mpC was as follows.

Programming for DMMs is based mostly on message-passing function extensions of C or Fortran, such as PVM [1] and MPI [2]. However, it is tedious and error-prone to program in a message-passing language, because of its low level. High-level programming languages, that have been developed for DMMs, can be divided into two classes depending on the parallel programming paradigm, task or data parallelism, underlying them. Task parallel [3-4] and data parallel [5-11] programming languages allow the user to implement different classes of parallel algorithms. Therefore, we have developed mpC (as an ANSI C superset) supporting both task and data parallelism. It is based on the notion of *network* comprising processor nodes of different types and performances connected with links of different bandwidths. It allows the user to describe network topology, create and discard networks, and distribute data and computations over the networks.

When developing the mpC programming environment (PE), we used a network of workstations running MPI as a target parallel DMM and found, that the principles, on which mpC based, make it and its PE convenient tools to develop efficient and portable parallel programs for heterogenous networks of workstations.

The point is that all PEs for DMMs which we know of have one common weakness. Namely, when developing a parallel program, either the user has no facilities to describe the virtual parallel system executing the program, or such facilities are too poor to specify an efficient distribution of computations and communications over the target heterogenous DMM. Even MPI's topological facilities have turned out insufficient to solve the problem. So, to ensure the efficient execution of the program on a particular DMM, the user must use facilities which are external to the program, such as boot schemes and application schemes [12]. If the user is familiar with both the topology of target DMM and the topology of the application, then, by using such configurational files, he can map the processes which constitute the program onto processors which make up DMM, to provide the most efficient execution of the program. But if the application topology is defined in run time (that is, if it depends on input data), it won't be successful.

mpC allows the user to specify an application topology, then its programming environment uses the information in run time to map processes onto processors of target DMM resulting in the efficient execution of the application. DMM's topology is detected automatically dependent on execution of a special program.

This paper outlines mpC and presents the implementation of network management. Details of the language are presented elsewhere [13-14].

2 mpC in brief

We will introduce mpC briefly, using a sample program which computes the center of gravity for a system of bodies. Let our system consist of large groups of bodies. Each body is characterized by its position and mass. Our parallel program will use a number of virtual processors, each of which will compute the center of gravity of a single group. The number of groups and the number of bodies in

1089-795X/96 $5.00 © 1996 IEEE
Proceedings of PACT '96

each group are defined in run time. Our mpC program is:

```
/*1 */ .../*Includes and defines*/
/*2 */ typedef double Coords[3];
/*3 */ typedef struct{Coords pos; double m;}
/*4 */         Body;
/*5 */ int [host]l, [host]n[MAXL];
/*6 */ Body (*[host]u[MAXL])[MAXN];
/*7 */ nettype Star(l, n[l]) {
/*8 */   coord I=l;
/*9 */   node  { I>=0: fast*n[I] scalar;};
/*10*/   link  { I>0: [I]<->[0]; };
/*11*/   parent [0];
/*12*/ };
/*13*/ void [*]main()
/*14*/ { Body [host]gs[MAXL];
/*15*/   repl dl, dn[MAXL], di;
/*16*/   .../*Initializing l, n and u */
/*17*/   dl=l;
/*18*/   dn[]=n[];
/*19*/   { net Star(dl, dn) w;
/*20*/     int [w]myN;
/*21*/     Body [w]g[MAXN], [w]gA;
/*22*/     repl [w]i, [w]j;
/*23*/     myN = dn[I coordof j];
/*24*/     for(i=0; i<dl; i++)
/*25*/       [w:I==i]g[] = (*u[i])[];
/*26*/     for(j=0; j<myN; j++)
/*27*/       gA.m+=g[j].m;
/*18*/     (gA.pos)[]=0.0;
/*29*/     for(j=0; j<myN; j++)
/*30*/       (gA.pos)[]+=
/*31*/          (g[j].m/gA.m)*(g[j].pos)[];
/*32*/     for(i=0; i<dl; i++)
/*33*/       gs[i]=[w:I==i]gA;
/*34*/   }
/*35*/   .../*Output of results*/
/*36*/ }
```

In mpC, the notion of *computing space* is defined as a set of typed virtual processors connected by links. Most common processors are of the `scalar` type. In addition, a processor is characterized by its relative performance. A directed *link* connecting two virtual processors is a one-way channel for transferring data from source processor to the processor of destination.

The basic notion of mpC is *network object* or simply *network*. Network consists of processors connected by links. Network is a region of the computing space which can be used to compute expressions and execute statements. Allocating network objects in the computing space and discarding them is performed in similar fashion as allocating data objects in storage and discarding them. Conceptually, the creation of new network is initiated by a processor of an existing network. This processor is called a *parent* of the created network. The parent belongs to the created network. The only processor defined from the beginning of program execution till program termination is the predefined *host-processor* of the `scalar` type.

Lines 5-6 define variable `l` and arrays `n` and `u` all belonging to the host-processor. Variable `l` will hold the number of groups, `n[i]` will hold the number of bodies in the i-th group, and `u[i]` will point to an array holding positions and masses of bodies in the i-th group. Line 14 defines array `gs` belonging to the host-processor, where `gs[i]` will hold the center of gravity and the total mass of the i-th group.

Every network object declared in an mpC program has a type. The type specifies the number, types and relative performances of processors, links and their lengths, as well as separates the parent.

Lines 7-12 declares the parametrized family of network types named `Star` having scalar formal parameter `l` and vector formal parameter `n`, the latter consisting of `l` elements. Line 8 declares the coordinate system to which processors are related. It introduces coordinate variable `I` ranging from 0 to `l-1`.

Line 9 declares processor nodes saying that ***for all*** *`I<l`* ***if*** *`I>=0`* ***then*** *fast scalar processor with relative performance `n[I]` is related to coordinate `[I]`*. The value of `n[I]` shall be positive integer. It is meant that in the framework of this network-type declaration the greater value of `n[I]` the more performance it specifies.

Line 10 declares links saying that ***for all*** *`I<l`* ***if*** *`I>0`* ***then*** *there exists an undirected link of the normal length between processors with coordinates `[I]` and `[0]`*. In general, the shorter the link, the wider bandwidth it specifies. If no link is specified from one processor to another, they are reputed to be connected with a very long link.

Line 11 says that the parent processor has coordinate [0].

Execution of the program begins from a call to function `main` on the entire computing space. Line 15 declares variables `dl` and `di` and array `dn`, all *distributed* over the entire computing space. By definition, a data object *distributed* over a region of the computing space comprises a set of components of the same type so that every processor of the region holds just one component. In addition, line 15 declares `dl`, `di` and `dn` to be *replicated* data objects. By definition, a distributed object is *replicated* if all its components are equal to each other.

Line 17 broadcasts the value of `l` to all components of distributed variable `dl`. Line 18 contains unusual unary postfix operator `[]`. The point is that mpC is a superset of a vector extension of ANSI C named the C[] language [15], where the notion of *vector* is defined as an ordered sequence of values of any one type. Unlike an array, a vector is not a data object but just a new kind of value. In particular, the value of an array is a vector. Operator `[]` supports access to arrays as a whole. It has operand of the type "array of *type*" and blocks (forbids) its conversion to pointer. So, expression `n[]` designates array `n` as a whole,

and expression `dn[]=n[]` broadcasts the value of array `n` to all components of distributed array `dn`.

Line 19 defines automatic network w. Its type is defined completely only in run time. Lines 20-21 defines variables `myN` and `gA` as well as array `g`, all distributed over w. Line 22 defines variables `i` and `j`, both replicated over w.

The statement in line 23 is executed on network w. It is an example of a so-called *asynchronous* statement, that is, a distributed statement, the execution of which is divided into a set of independent undistributed statements each of which is executed on the corresponding processor using the corresponding data components. Most operators of mpC are asynchronous in the sense that either both operands and the result belong to the same processor, or they both are distributed over the same region of the computing space, and the distributed operator is divided into a set of independent undistributed operators each of which is performed on corresponding components of the operands. The result of binary operator `coordof` is an integer value distributed over w each component of which is equal to the value of coordinate `I` of the processor to which the component belongs.

The statement in lines 24-25 is executed on w and scatters the arrays which contains information about groups of bodies from the host-processor to all processors of w. As a result, each component of the distributed array `g` will contain the information about the corresponding group.

The statements in lines 26-31 are asynchronous and runs on network w. For each group of bodies they compute the total mass and the center of gravity in parallel.

The statement in line 32-33 gathers this information from all processors of w to the host-processor.

Please, note, that the network w, which executes the computations and communications, is defined in such a way, that the more powerful the virtual processor, the larger the group of bodies it computes.

3 The mpC compiler architecture

The main function of the compiler is to translate an mpC program into a set of programs, each of which runs on its own (virtual) processor, and which in total implement the computations specified by the initial mpC program which interacts by means of message passing. The target language is optional: either C[] or C (depending on the compiler mode) used with the library of the run-time support system. The compilation unit is an mpC file.

Currently, the compiler consists of a run-time support system (RTSS), a front-end, and a back-end.

RTSS provides the operations on networks and subnetworks as well as message passing between virtual processors. It has a precisely specified interface and covers a particular communicating package (currently, MPI). It ensures platform-independence of the rest of the compiler components.

Front-end translates source code into internal representation (a kind of attributed tree) and performs static semantic checking. In addition, for every network-type declaration it generates an internal representation of the functions, used for computation of topological information.

Back-end uses the internal representation as input and builds the relevant C[] or C code with necessary calls to RTSS functions.

4 Target program model

Our compiler uses the SPMD model of the target code, in which all the processes which constitute a target message-passing program, run the identical code.

All the processes which constitute the target program are divided into 2 groups - the special process, named *dispatcher,* playing the role of the computing space manager, and general processes, named *nodes,* playing the role of processor nodes of the computing space. The special node, named *host,* is separated. The dispatcher works as a server accepting requests from nodes. The dispatcher does not belong to the computing space.

The running of the target program begins with a call to the initialization function `MPC_Init` executed by all processes constituting the program (including the dispatcher). The initialization includes:

- initializing the message-passing platform on which RTSS is based;
- separating the dispatcher and determining the host;
- local initialization of nodes;
- initialization of the dispatcher.

The dispatcher is the only process that does not leave the `MPC_Init` function. It loops inside this function accepting requests. A particular example of such request is a program termination.

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

A region is accessed via its descriptor. If the descriptor `rd` corresponds to the region, then a node belongs to the region iff the function call `MPC_Is_member(&rd)` returns 1. In this case, the descriptor `rd` allows the node to

obtain comprehensive information about the region as well as identify itself in the region.

Creation of the region involves its parent node, the dispatcher and all free nodes. The parent node calls the function `MPC_Net_create` to send the creation request containing the necessary information about the network topology to the dispatcher. Also, the dispatcher holds a map of the computing space which reflects its topological properties. The initial state of the map is formed during the dispatcher initialization. Based on this information, the dispatcher selects the most appropriate set of free nodes. After that, it sends to every free node a message saying whether the node is hired in the created region or not. In addition, for the nodes which are hired in the region, this message contains the information necessary to complete initialization of the region descriptor. Meanwhile, all free nodes call to the function `MPC_Offer` which wait for the message from the dispatcher. After returning from these functions, nodes constituting the region, have the corresponding region descriptor initialized completely. Deallocation of the region involves all its members, calling to the function MPC_Net_free, as well as the dispatcher.

The dispatcher keeps a queue of creation requests that cannot be satisfied immediately, but can be served in the future. It implements a strategy for serving the requests aimed at minimizing the probability of a deadlock. The dispatcher detects such a situation when the sum of the number of free nodes and the number of such hired nodes that could be released is less than the minimum number of free nodes required by a request in the queue. In this case, it calls to the function `MPC_Abort` that terminates the program.

5 Translation of network declarations

When compiling a source mpC file, for every mpC function definition the compiler generates a target function definition. If the source function contains or may contain network declarations, the target function body will contain an additional code for every external network declaration for which the function is in scope. This code performs the creation of the corresponding region if it has not been created yet.

Free nodes should not only wait for a message from the dispatcher but also participate in overall computations (that is, computations involving the entire computing space, the statements in lines 17-18 in our sample program have just specified overall computations). Therefore, in general the code performed by a free node may contain several points (called *waiting points*) where the node waits for a message from the dispatcher. Also, portions of code producing creation requests can be performed by network parents in parallel. Finally, sometimes one cannot predict even in run-time how many times a portion of code producing creation requests will be performed by the network parent. The latter situation occurs, for example, when translating the network declaration in a loop body or in a selection statement. All these require very accurate synchronization between portions of code that are performed by hired nodes and produce creation requests, and portions of code performed by free nodes waiting for a message from the dispatcher.

To describe this synchronization, all mpC statements are divided into 2 groups: overall statements and parallel statements. An overall statement is performed on the entire computing space. Execution of a parallel statement involves only a part of the computing space.

All network declarations in the source function and all external network declarations for which the function is in scope are divided into disjoint sets (called *waiting sets*) each corresponding to its own waiting point. In addition, for every waiting point a set of parallel statements, executed in parallel with code associated with the waiting point, is created. The set of parallel statements is called *parallel section* associated with the corresponding waiting point.

For each waiting point, a target code of the form

```
MPC_waiting_point:
    if(MPC_Is_free())
      waiting_statement
      if(!MPC_Is_free())
        creating_statement
      MPC_Global_barrier();
```

is generated. Here, the waiting statement is just the code performed by free nodes at the waiting point, and the creating statement consists of a code producing requests for network creation and deallocation and the code implementing the parallel section. The call to the function `MPC_Global_barrier` is a barrier synchronization across all nodes of the entire computing space.

Parallel execution of the waiting and creating statements are synchronized to enable the following scenario. All free nodes call to the function `MPC_Offer`. The call returns at any free node in two ways. Firstly, after some hired node, executing the creating statement, calls to the function `MPC_Net_create` which sends the creation request. In this case, if a free node is hired in the created region, it leaves the waiting statement and enters the creating statement. If the free node is not hired, it calls to `MPC_Offer` again. Secondly, the call to `MPC_Offer` returns at any free node after it becomes known that further execution of the creating statement will not produce new creation requests. In this case, all free nodes leave the waiting statement and enter the call to `MPC_Global_barrier`.

Besides, some hired nodes executing the creating statement and constituting a region may call to the function MPC_Net_free to discard the network. In this case, every node, that becomes free, jumps to the label MPC_waiting_point.

Note that a single network declaration may declare several network identifiers. Moreover, more than one creation request, associated with a network identifier, may be produced. Such a situation occurs, for example, if a declaration of automatic network appears in a loop body.

In the most general case, the compiler generates the following waiting statement:

```
{
  MPC_Net *MPC_nets[N]=
     {list_of_region_descriptor_pointers};
  MPC_Name MPC_names[N]=
      {list_of_network_numbers};
  while(1) {
    MPC_Offer(&MPC_command, MPC_names,
              MPC_nets, N);
    if(MPC_command == MPC_HIRED ||
       MPC_command == MPC_OUT)
      break;
  }
}
```

Here, N is the number of network identifiers whose declarations are associated with the waiting point. An element of the array MPC_names is initialized to contain a unique number of the corresponding network identifier in the source mpC file. The corresponding element of the array MPC_nets is initialized to point to the region descriptor associated with the network identifier. After the call to MPC_Offer returns at a free node, variable MPC_command will contain either value MPC_HIRED if the node is hired in a region (in this case, the corresponding region descriptor gets fully initialized), or value MPC_SKIP if the node is not hired in the region, or value MPC_OUT if the node should leave the waiting point.

The corresponding creating statement is of the following structure:

```
{
 code implementing the parallel section and
 producing creation/deallocation requests

 code initiating synchronous exit from the
 waiting point
}
```

The piece of code, producing a request for creation of an instance of the automatic network, looks as follows:

```
if(I_am_parent_of_the_network)
   MPC_Net_create(NET_ID,&net);
```

Here, NET_ID is a unique number of the network in the source mpC file, net is a region descriptor associated with the network. Since a static network is created only once, the piece of code producing the corresponding creation request tests if the network has already been created:

```
if(I_am_parent_of_the_network)
  if(!MPC_Is_created(&net))
    MPC_Net_create(NET_ID,&net);
```

For lack of space, we omit the complicated algorithm generating the code initiating synchronous exit from a waiting point.

A piece of code producing a network deallocation request is generated according to the following algorithm:

```
ISSUE_THE_CODE
  if(MPC_Is_member(&net)) {
    int MPC_free;
    MPC_free=!MPC_Is_parent(&net);
IF ( the deallocated network has a node being a parent
     of some network associated with the waiting point )
  ISSUE_THE_CODE
    MPC_Barrier(&net);
ENDIF
ISSUE_THE_CODE
  MPC_Net_free(&net);
  if(MPC_free) goto MPC_waiting_point;}
```

Although the target code described above will work correctly for any waiting point, there are a few particular cases in which the compiler generates a more efficient code, avoiding the expensive synchronization when leaving a waiting point. For example, if when entering a waiting point, a free node can compute the number of creation requests that will be produced, and creation and deallocation requests are not interleaved, then for the waiting point the following target code will be generated:

```
MPC_waiting_point:
  if(MPC_Is_free()) {
    MPC_Net *MPC_nets[N]=
      {list_of_region_descriptor_pointers};
    MPC_Name MPC_names[N]=
      {list_of_network_numbers};
    code computing
    the_number_of_creation_requests
    for(i=0;
        i<the_number_of_creation_requests;
        i++) {
      MPC_Offer(&MPC_command, MPC_names,
                MPC_nets, N);
      if(MPC_command == MPC_HIRED) break;
    }
  }
  if(!MPC_Is_free()) {
    code implementing the parallel section
    and producing creation and deallocation
    requests
  }
  MPC_Global_barrier();
```

Here, the piece of code producing a deallocation request is reduced to

```
if(MPC_Is_member(&net))
  MPC_Net_free(&net);
```

6 Experimental results

We compared the running time of our mpC program to its manually written MPI counterpart. To get concise result, we make the programs repeat computations corresponding to lines 26-31 in the mpC program 20 times. We use 3 workstations - SPARCstation 5 (hostname `gamma`), SPARCclassic (`omega`), and SPARCstation 20 (`alpha`), running MPI (LAM implementation [12]).

We use the cyclic mapping of MPI processes on actual processors `gamma`, `omega`, and `alpha` constituting our DMM.

The computing space of the mpC programming environment consists of 9 processes, 3 processes running on each processor. The dispatcher runs on `gamma` and uses the following relative performances of the processors obtained automatically upon the creation of the virtual parallel machine: 1224 (`gamma`), 326 (`omega`), 1677 (`alpha`).

The mpC program runs no slower than the MPI program for all input data. Likewise, for most input data the mpC program runs much faster the its MPI counterpart.

For example, if we use 8 groups, each of which of 500 bodies, the mpC program takes 27 seconds, meanwhile the MPI program takes 53 seconds. In this instance, the dispatcher has selected 3 processes on `gamma`, 3 processes on `alpha` and 2 processes on `omega`.

If we use 8 groups consisting of 500, 9000, 500, 1000, 9000, 1000, 1000, and 9000 bodies respectively, the mpC program takes 27 seconds, meanwhile the MPI program takes 156 seconds. In this instance, the dispatcher has selected three processes on `gamma` for the virtual processors of network w with coordinates 0, 4, and 5, two processes on `omega` for virtual processors with coordinates 6 and 2, and three processes on `alpha` for virtual processors with coordinates 1, 7 and 3. Remember, that according to the mpC program, i-th virtual processor computes out i-th group of bodies.

7 Summary

The key peculiarity of the mpC language is its advanced facilities for managing such resources of DMMs as processors and links between them. The user can manage these resources in the manner similar to managing the storage in C. These facilities permit the development of parallel programs for DMMs, that once compiled, will run efficiently on any particular DMM, because the mpC programming environment ensures optimal distribution of computations and communications over DMM in run time. It makes mpC and its programming environment suitable tools for development of a library of parallel programs, especially for heterogenous DMMs.

This paper has presented some details of the implementation of these facilities in the framework of the mpC programming environment developed in the Institute for System Programming, Russian Academy of Sciences.

Acknowledgments

The work was supported by ONR and partially by Russian Basic Research Foundation.

References

[1] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience,* 2(4):315-339, 1990.

[2] Message Passing Interface Forum. MPI: A Message-passing Interface Standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[3] I. Foster, and K. M. Chandy. Fortran M: a language for modular parallel programming. Preprint MCS-P327-0992, Argonne National Lab, 1992.

[4] K. M. Chandy, and C. Kesselman. CC++: A Declarative Concurrent Object Oriented Programming Language. Technical Report CS-TR-92-01, California Institute of Technology, Pasadena, California, 1992.

[5] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. Fortran D Language Specification. Center for Research on Parallel Computation, Rice University, Houston, TX, October 1993.

[6] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31-50, 1992.

[7] High Performance Fortran Forum. *High Performance Fortran language specification, version 1.0.* Rice University, Houston, TX, May 1993.

[8] The CM Fortran Programming Language. *CM-5 Technical Summary*, pp. 61-67, Thinking Machines Corp., Nov. 1992.

[9] The C* Programming Language. *CM-5 Technical Summary*, pp. 69-75, Thinking Machines Corporation, November 1992.

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

[11] M. Philippsen, and W. Tichy. Modula-2* and its compilation. *First International Conference of the Austrian Center for Parallel Computation*, Salzburg, Austria, 1991.

[12] Trollius LAM Implementation of MPI. Version 5.2. Ohio State University, 1994.

[13] A. Lastovetsky. mpC - a Multi-Paradigm Programming Language for Massively Parallel Computers. *ACM SIGPLAN Notices, 31(2):13-20,* February 1996.

[14] A. Lastovetsky. The mpC Programming Language Specification. Technical Report, Institute for System Programming, Russian Academy of Sciences, Moscow, 1994.

[15] S. Gaissaryan, and A. Lastovetsky. ANSI C Superset for Vector and Superscalar Computers and Its Retargetable Compiler. *Journal of C Language Translation*, 5(3):183-198, 1994.