Modular Parallel Programming in mpC for Distributed Memory Machines

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

Ted Lewis
Naval Postgraduate School, Code CS, Monterey, CA 93943-5118
lewis@cs.nps.navy.mil

Abstract

The mpC language is an ANSI C superset supporting modular parallel programming for distributed memory machines. It allows the user to specify dynamically an application topology, and the mpC programming environment uses this information in run time to provide the most efficient execution of the program on any particular distributed memory machine. The paper describes the features of mpC and its programming environment which allow to use them for developing libraries of parallel programs.

1 Introduction

Programming for distributed memory machines (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. Therefore, a number of high-level programming languages have been developed each of which supports either task or data parallelism. Task parallel [3-4] and data parallel [5-11] languages allow the user to implement different classes of parallel algorithms. 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. The user can describe network topology, create and discard networks, and distribute data and computations over the networks. It is important that mpC allows dynamic creation of networks of dynamic structure.

All programming environments (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 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 (in particular, dynamically), 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.

When developing the mpC programming environment, 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 networks of workstations (especially, for heterogenous ones). Modularity of mpC allows to get such programs up in the form of libraries.

The paper describes the features of mpC and its PE which allow to use them for developing libraries of parallel programs. Details of the language and its implementation are presented elsewhere [13-15].

0-8186-7870-4/96 $10.00

2 mpC in brief

We will introduce mpC briefly, using a sample program which multiplies 2 dense square matrices. Our parallel program will use a number of virtual processors, each of which will compute a number of rows of the resulting matrix. The dimension of matrices and the number of virtual processors are defined in run time. Our mpC program is:

```
/*.....includes and defines....*/

/* 1*/ void [*]main()
/* 2*/ {
/* 3*/  float *[host]x, *[host]y, *[host]z;
/* 4*/  int [host]N;
/* 5*/  void [host]Input(), [host]Output(),
/* 6*/       [*]MxM();
/* 7*/  Input(&x, &y, &N);
/* 8*/  z=([host]calloc)
/* 9*/     (N*N, [host](sizeof(float)));
/*10*/  MxM(x, y, z, N);
/*11*/  Output(z);
/*12*/ }

/*13*/ void [*]MxM(float *[host]x,
/*14*/              float *[host]y,
/*15*/              float *[host]z,
/*16*/              int[host]n)
/*17*/ {
/*18*/  repl double *powers;
/*19*/  repl nprocs, nrows[MAXNPROCS], dn;
/*20*/  void Partition();
/*21*/  MPC_Processors_static_info(&nprocs,
/*22*/                             &powers);
/*23*/  dn=n;
/*24*/  Partition(nprocs,powers,nrows,dn);
/*25*/  {
/*26*/   nettype Star(m, n[m]) {
/*27*/    coord I=m;
/*28*/    node {I>=0: fast*n[I] scalar;};
/*29*/    link {I>0:  [I]->[0], [0]->[I];};
/*30*/    parent [0];
/*31*/   };
/*32*/   net Star(nprocs, nrows) w;
/*33*/   float *[w]dx, *[w]dy, *[w]dz;
/*34*/   int [w]myn, [w]sof;
/*35*/   repl [w]n;
/*36*/   void [net SimpleNet(p)] ParMult(
/*37*/     float*, float*, float*, repl*,
/*38*/     repl);
/*39*/
/*40*/   myn=([w]nrows)[I coordof dx];
/*41*/   sof=[w](sizeof(float));
/*42*/   n=[w]dn;
/*43*/   dx=([w]calloc)(n*myn, sof);
/*44*/   ([host]free)([host]dx);
/*45*/   [host]dx=(void *)x;
/*46*/   dy=([w]calloc)(n*n, sof);
/*47*/   ([host]free)([host]dy);
/*48*/   [host]dy=(void *)y;
/*49*/   dz=([w]calloc)(n*myn, sof);
/*50*/   ([host]free)([host]dz);
/*51*/   [host]dz=(void *)z;
/*52*/   ([[w]nprocs)w])ParMult(dx,dy,dz,
/*53*/                          [w]nrows,n);
/*54*/  }
/*55*/ }

/*56*/ void [net SimpleNet(p)v] ParMult(
/*57*/     float *dx, float *dy, float *dz,
/*58*/      repl *r, repl n)
/*59*/ {
/*60*/  repl s=0;
/*61*/  int myn, i;
/*62*/  int *d, *l, c;
/*63*/  void SeqMult(float*, float*,
/*64*/               float*,  int, int);
/*65*/
/*66*/  myn=r[I coordof r];
/*67*/  ([(p)v])MPC_Bcast(&s, dy, 1,
/*68*/                    n*n, dy, 1);
/*69*/  d=calloc(p, sizeof(int));
/*70*/  l=calloc(p, sizeof(int));
/*71*/  for(i=0, d[0]=0; i<p; i++) {
/*72*/   l[i]=r[i]*n;
/*73*/   if(i+1<p) d[i+1]=l[i]+d[i];
/*74*/  }
/*75*/  c=l[I coordof c];
/*76*/  ([(p)v])MPC_Scatter(&s, dx ,d,
/*77*/                      l, c, dx);
/*78*/  SeqMult(dx, dy, dz, myn, n);
/*79*/  ([(p)v])MPC_Gather(&s,dz,d,l,c,dz);
/*80*/ }

/*81*/ void SeqMult(float *a, float *b,
/*82*/              float *c, int m, int n)
/*83*/ {
/*84*/  int i, j, k, ixn;
/*85*/  double s;
/*86*/
/*87*/  for(i=0; i<m; i++)
/*88*/   for(j=0, ixn=i*n; j<n; j++) {
/*89*/    for(k=0, s=0.0; k<n; k++)
/*90*/     s+=a[ixn+k]*(double)(b[k*n+j]);
/*91*/    c[ixn+j]=s;
/*92*/   }
/*93*/ }

/*94*/  void Partition(int p, double *v,
/*95*/                 int *r, int n)
/*96*/  {
/*97*/   int sr, i;
/*98*/   double sv;
/*99*/
/*100*/  for(i=0, sv=0.0; i<p; i++)
```

```
/*102*/    sv+=v[i];
/*102*/  for(i=0, sr=0; i<p; i++) {
/*103*/    r[i]=(int)(v[i]/sv*n);
/*104*/    sr+=r[i];
/*105*/  }
/*106*/  if(sr!=n) r[0]+=n-sr;
/*107*/ }
```

In mpC, the notion of *computing space* is defined as a set of typed virtual processors connected by links. Most common virtual processors are of the `scalar` type. In addition, a virtual 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 explicitly defined from the beginning of program execution till program termination is the pre-defined *host-processor* of the `scalar` type.

Execution of the program begins from a call to function `main` on the entire computing space. Lines 3-4 define variables `x`, `y`, `z`, and `N` all belonging to the host-processor. Variable `N` will hold the dimension of matrices. Variables `x`, `y` will point to arrays holding input matrices, and `z` will point to an array holding the output matrix.

Lines 5-6 declare functions `Input`, `Output` and `MxM`. In addition, the library function `calloc` is used in the `main` function. In general, there are 3 kinds of functions in mpC: basic, nodal, and network ones. In lines 1, 6, the construct `[*]`, placed just before the function identifier, specifies that `main` and `MxM` are identifiers of *basic* functions. A declaration of nodal function does not need any additional specifiers, so `calloc` is an identifier of *nodal* function. In line 5, the construct `[host]`, placed just before the function identifier, specifies that `Input` and `Output` are identifiers of *network* functions associated with the host-processor.

In line 7, the call to function `Input` is executed on the host-processor and initiates arrays `x`, `y` and variable `N`.

A nodal function may be executed completely by any one processor of the computing space. In lines 8-9, the function `calloc` is called on the host-processor to allocate storage for array `z`. Note, that the operator `sizeof` is not an operator of mpC compile time.

In line 10, the call to the basic function `MxM` is executed on the entire computing space. Its arguments belong to the host-processor. It multiplies dense `NxN` matrices `x` and `y` and puts the result in `z`.

In line 11, the network function `Output` is called (on the host-processor) to output `z`.

Lines 13-55 contain the definition of the function `MxM`.

Line 18 declares variable `power` *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.

Line 19 declares variables `nprocs` and `dn` and array `nrows`, all *replicated* over the entire computing space. By definition, a distributed data object is *replicated* if all its components are equal to each other. Note, that line 18 specifies the data object `*power` to be replicated.

Lines 24-25 call to the standard nodal function `MPC_Processors_static_info` on every virtual processor of the entire computing space returning the number of actual processors and their relative performances. The point is that the entire computing space is reputed to be constituted by a number of processes (playing the role of virtual processors) running on a number of actual processors. So, after this call the variable `nprocs` will hold the number of actual processors, and the array `powers` will hold their relative performances.

Line 23 broadcasts the value of `n` to all components of distributed variable `dn`.

Lines 24-25 call to the nodal function `Partition` on every virtual processor of the entire computing space. Based on relative performances of actual processors, this function computes how many rows of the resulting matrix will be computed by every actual processor. So, after this call `nrows[i]` will hold the number of rows computed by `i`-th actual processor.

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

Lines 26-31 declares the parametrized family of network types, named `Star`, which has 2 formal *topological parameters*: scalar parameter `m` and vector parameter `n`, the latter consisting of `m` elements. Line 27 declares the coordinate system to which (virtual) processors are related. It introduces coordinate variable `I` ranging from 0 to `m-1`.

Line 28 declares processor nodes saying that ***for all*** *`I<m`* ***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 29 declares links saying that ***for all*** *`I<m`* ***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 30 says that the parent processor has coordinate [0].

Line 32 defines automatic network w. Its type is defined completely only in run time. The network w, which executes the rest of computations and communications, is defined in such a way, that the more powerful the virtual processor, the greater number of rows it computes. The mpC programming environment will ensure the optimal mapping of the virtual processors of w into a set of processes constituting the entire computing space. So, just one process from processes running on each of actual processors will be involved in multiplying the matrices, and the more powerful the actual processor, the greater number of rows its process will compute.

Lines 33-34 define variables dx, dy, dz, myn and sof all distributed over w. Line 35 defines variable n replicated over w.

The statement in line 40 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 (operand dx is used only to specify the network over which the result will be distributed). The unary operator [w] cuts from pointer nrows, distributed over the entire computing space, a pointer distributed over w. So, after execution of this statement, each component of myn will hold the number of rows of the resulting matrix that the corresponding processor will compute.

Statements in lines 41-43 are also asynchronous and distributed over w. After execution statements in lines 43-45, each component of dx will point to an array which will hold the corresponding portion of the matrix x. Lines 46-51 do the same for matrices y and z. Note, that the components of dx, dy and dz belonging to the host-processor will point to arrays x, y and z correspondingly.

Lines 52-53 call to the network function ParMult declared in lines 36-38 and defined in lines 56-80. Unlike the network functions Input and Output, this network function is not hardly associated with some particular network. The header of this function definition declares the identifier v of a network being a special *network formal parameter* of the function. The function can be called on any network of an appropriate type.

Unlike basic functions, no network other than the network formal parameter can be created or used in the body of the network function. Only data objects belonging to the network formal parameter can be defined in the body. In addition, the corresponding components of an externally-defined distributed data object may be used.

Since the network formal parameter v has a parametrized type, the corresponding topological parameter p is also declared in the header of the function definition being also a special formal parameter. In the function body, this parameter is treated as an unmodifiable variable of the type int replicated over the network formal parameter v. The rest of formal parameters (regular formal parameters) of the function are also distributed over v.

When calling to this function, the topological argument [w]nprocs specifies a network type as an instance of the parametrized network type SimpleNet, and the network argument w specifies a region of the computing space treated by the function as a network of this type.

A declaration of the parametrized network type SimpleNet is contained in a standard mpC header and is:

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

It specifies networks consisting of n processors of scalar type such that each pair of processors is connected with a link of normal length, the parent having the 0-th coordinate.

So, the network function ParMult is called and executed on the network w, and its arguments is also distributed over this region of the computing space.

Lines 56-58 contains the header of the ParMult definition. The special formal parameter p holds the number of virtual processors in the network v. The regular formal parameter n holds the dimension of matrices, r points to p-element array i-th element of which holds the number of rows of the resulting matrix that i-th virtual processor of the network v computes. Each component of dy points to an array to contain nxn matrix y. Each component of dz points to an array to contain the rows of z computed on the corresponding virtual processor of v. Each component of dx points to an array to contain the rows of x used in computations on the corresponding virtual processor. Note, that on the parent dx, dy, dz are reputed to point to matrices x, y and z correspondingly.

Line 60 defines the integer variable s replicated over v. Lines 61-62 define variables myn, i, d, l and c all distributed over v.

After execution of the asynchronous statement in line 66,

each component of myn will contain the number of rows of the resulting matrix that computes the corresponding virtual processor.

Lines 67-68 call to the embedded network function MPC_Bcast which is declared in a standard mpC header as follows:

```
int [net SimpleNet(n)] MPC_Bcast(
    repl const *coordinates_of_source,
    void *source_buffer,
    const source_step,
    repl const count,
    void *destination_buffer,
    const destination_step);
```

This call broadcasts the matrix y from the parent of v to all virtual processors of v. As a result, each component of the distributed array dy will contain the matrix y.

Statements in lines 69-74 are asynchronous. They initiate 2 distributed p-element arrays d and l. After their execution, l[i] will hold the number of elements in the portion of the resulting matrix which is computed by the i-th virtual processor of v, and d[i] will hold the displacement which corresponds to this portion in the resulting matrix. Equivalently, l[i] will hold the number of elements in the portion of the matrix x which is used by the i-th virtual processor of v, and d[i] will hold the displacement which corresponds to this portion in matrix x.

The statement in line 75 is also asynchronous. After its execution, each component of c will hold the number of elements in the portion of the resulting matrix which is computed by the corresponding virtual processor (equivalently, the number of elements in the portion of the matrix x which is used by this virtual processor).

Lines 76-77 call to the embedded network function MPC_Scatter which is declared in a standard mpC header as follows:

```
int [net SimpleNet(n) w] MPC_Scatter(
    repl const *coordinates_of_source,
    void *source_buffer,
    const *displacements,
    const *sendcounts,
    const receivecount,
    void *destination_buffer);
```

This call scatters the matrix x from the parent of v to all virtual processors of v. As a result, each component of dx will point to the array containing the corresponding portion of the matrix x.

The statement in line 78 is executed asynchronously on v. It calls to the nodal function SeqMult on all virtual processors of v, which computes the corresponding portion of the resulting matrix on each of the virtual processors in parallel. Lines 81-93 contains the definition of the function SeqMult which implements traditional sequential algorithm of matrix multiplication.

Finally, line 79 calls to the embedded network function MPC_Gather which is declared in a standard mpC header as follows:

```
int [net SimpleNet(n) w] MPC_Gather(
  repl const *coordinates_of_destination,
  void *destination_buffer,
  const *displacements,
  const *receivecounts,
  const sendcount,
  void *source_buffer);
```

This call gathers the resulting matrix z each virtual processor of v sending its portion of the result to the parent of v.

3 The mpC programming environment

Currently, the programming environment includes a compiler, a run-time support system (RTSS), a small library, a detector of DMM's topology, and a non-graphical user interface.

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 and communications specified by the initial mpC program interacting by means of message passing. The target language is currently C with calls to functions of RTSS. The compilation unit is an mpC file.

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

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

The topology detector executes a special test program to detect performances of workstations constituting the target DMM, the number of processors in each of these workstations, as well as bandwidths of links connecting the workstations (optionally).

The user interface consists of a number of programs supporting the creation of a virtual parallel machine and the execution of mpC programs on the machine. While creating the machine with the command mpccreate, its topology is detected by the topology detector and saved in the file used by RTSS. Currently, it works on networks of UNIX workstations running MPI.

Currently, our compiler uses optionally either the SPMD model of target code, where all processes constituting a target message-passing program run the identical code, or a quasi-SPMD model, where it translates the source mpC file into 2 separate target files - the first for the host-processor and the second for the rest of virtual processors.

All processes constituting 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 virtual processors 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.

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

Creation of the region representing a network involves the parent node, the dispatcher and all free nodes. The parent node sends a creation request containing the necessary information about the network topology to the dispatcher. Based on this information and the information about the topology of the virtual parallel machine, 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. Deallocation of network region involves all its members 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 some strategy of serving the requests aimed at minimization of the probability of occurring 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. Note, that although currently the dispatcher is implemented as a single process, the RTSS interface allows one to implement it as a distributed component also.

4 Experimental results

We measured the running time of our mpC program multiplying two dense square matrices. We used three Sun SPARCstations 5 (hostnames `gamma`, `beta`, and `delta`), and Sun SPARCstation 20 (hostname `alpha`) connected via 10Mbits Ethernet. There were more than 20 other computers in the same segment of the local network. We used LAM MPI Version 5.2 as a particular communication platform.

Three virtual parallel machines were created:

- `gabd` consisting of `gamma` (its relative performance detected during the creation of this virtual parallel machine was equal to 1055), `alpha` (1641), `beta` (1161), and `delta` (1171));
- `gad` consisting of `gamma` (1151), `alpha` (1640), and `delta` (1141));
- `ga` consisting of `gamma` (1137), and `alpha` (1887)).

The computing space of `gabd` was constituted by 20 processes (5 processes running on each of workstations). Computing spaces of `gad` and `ga` were constituted by 15 and 10 processes correspondingly. As a base of the comparison we used the running time of the sequential C program implementing the traditional algorithm of matrix multiplication (we used `gamma` to execute this program). Table 1 shows the experimental results for matrices of different dimensions.

Table 1:Time of multiplying nxn matrices(sec)

n	gabd	gad	ga	C
200	15	7	9	11
300	40	25	31	41
400	66	56	80	105
500	141	125	146	221
600	192	229	265	381
700	353	340	430	613
800	456	494	628	931

Note, that the running time of mpC program substantially depends on the network load. We monitored the network activity during our experiments. We have observed up to 32 collisions per second. The collisions occurred more often during broadcasting large data portions. The collisions resulted in visible degradation of the network bandwidth. One of our computers (`alpha`) also operates as a file server, and its workload was unstable during tests. Sometimes it spent some more time to complete its part of computations than it was predicted by `mpccreate` utility.

Table 2 compares the contribution of communications and computations in the total running time of the mpC program (the results for `gabd` are presented). The first row shows matrix dimensions, second and third rows show percentage of communications and computations in the total running time, the fourth row shows communications in ratio of computations. The fifth row shows communications in ratio of computations, not measured but calculated in assumption that computation time $t_{comp} \sim n^3$ and communication time $t_{comm} \sim n^2$ (n is matrix dimension). Let $t_{comp}=an^3$, $t_{comm}=bn^2$, then $t_{comm}/t_{comp}=b/(an)$. Since $t_{comp}=t_{comm}$ for n=600, then b/a=600 and $t_{comm}/t_{comp}=600/n$.

Table 2:Contribution of communications and computations in the total running time

n	600	700	800
communications(%)	50	46	42
computations(%)	50	54	58
comm./comput.	1	0.85	0.72
assumed comm./comput.	1	0.86	0.75

Communications in our mpC program consist of three parts: scattering the first matrix, broadcasting the second matrix, and gathering the resulting matrix. Table 3 compares the contribution of each of these parts in the total communication time (for `gabd` virtual parallel machine).

Table 3:Contribution of broadcast, scatter, and gather in the total communication time

n	bcast	scatter	gather
600	0.56	0.32	0.14
700	0.65	0.26	0.09
800	0.74	0.2	0.1

While analyzing the presented results, it is necessary to take into account some peculiarities of both the implementation of MPI which we used and our local network.

Our local network does not support fast communications. It is based on 10Mbits Ethernet and uses old-fashioned network equipment. In addition, there are 26 computers in our segment of the network connected via cascade of 4 hubs. To characterize our network, it is enough to say that `ftp` transfers data from `gamma` to `alpha` at the rate of 300-400Kbytes/s. It means that real bandwidth of our network is not less than 25-30% of its maximum bandwidth.

On the other hand, LAM MPI Version 5.2 ensures broadcasting large floating arrays at the rate of 50-60Kbytes/s. In addition, it doesn't use multicasting facilities of our network.

Nevertheless, even under these conditions, our mpC program has demonstrated good speedup comparing with the sequential C program. If an implementation of MPI ensures the communication rate comparable with the real bandwidth of the local network and uses its multicasting facilities, the contribution of communications in the total running time of our mpC program will not exceed 10-15% (we are going to experiment with LAM MPI Version 6.0 and MPICH and hope to present the timing in the nearest future). If, in addition, we use 100Mbits Ethernet and up-to-date network technologies (for example, replacing hubs with switching devices), the contribution of communications in the total running time of the mpC program will not exceed 1-2%. That is, the mpC programming environment can ensure practically ideal speedup of the presented mpC program for up-to-date networks of workstations.

5 Discussion

The presented mpC program demonstrates how mpC and its programming environment can be used to write libraries of parallel programs for DMMs.

Basic library functions, similar to the function `MxM`, allow the user to write implicit parallel programs in a sequential style using calls to basic library functions. Once compiled, such functions will run efficiently on any particular DMM, because the mpC programming environment ensures optimal distribution of computations and communications over DMM in run time. Note, that although we didn't do it, we could call to some standard nodal library function to obtain information about real bandwidths of the current network in relation to send/receive, broadcast, scatter, and gather, measured in flops of the host-processor, and use this information to optimize more carefully the running time of our program. For example, in case of our slow network for some matrix dimensions it makes sense not to use more computers, because communication overheads start exceeding gains from parallelizing computations.

Network library functions, similar to the function `ParMult`, support explicit parallel programming. They are more difficult in usage, but allow avoid overheads from redundant creation of networks. In addition, unlike basic functions, network functions can be executed in parallel providing more efficient code.

Finally, mpC allows to utilize all C library functions treating them as mpC nodal library functions.

6 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.

The paper has described the features of mpC and its programming environment, developed in the Institute for System Programming, Russian Academy of Sciences, which allow to use them for developing libraries of parallel programs.

Acknowledgments

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

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] S.U. Hanssgen, E.A. Heinz, P. Lukowicz, M. Philippsen, and W.F. Tichy. The Modula-2* environment for parallel programming. *Proceedings of the Working Conference on Programming Models for Massively Parallel Computers,* Berlin, Germany, September 1993.

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

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

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

[15] D. Arapov, A. Kalinov, and A. Lastovetsky. Managing the Computing Space in the mpC Compiler. *Proceedings of the 1996 Parallel Architectures and Compilation Techniques (PACT'96) conference,* Boston, October 1996.