Performance Analysis of DECK Collective Communication Service

Rafael Silva*, Delcino Picinin*, Marcos Barreto**, Rafael Ávila*, Tiaraú Diverio* and Philippe Navaux*

* Institute of Informatics - UFRGS
Av. Bento Gonçalves 9500 PO Box: 15064 Zip Code: 90501-910 Porto Alegre – RS - Brazil
{resilva, picinin, avila, diverio, navaux}@inf.ufrgs.br
** Centro Universitário La Salle - UNILASALLE
Av. Victor Barreto 2288 Zip Code: 92010-000 Canoas – RS - Brazil
{barreto@lasalle.tche.br}

Abstract

Collective communication is very useful for parallel applications, especially those in which matrix and vector data structures need to be manipulated by a group of processes. This paper presents a performance analysis of collective communication primitives designed for the DECK parallel programming environment, with the aid of different numerical methods used to solve hydrodynamics and mass transportation models.

1. Introduction

One of the parallel processing premises is to divide large and complex tasks in short tasks that are distributed and executed at the same time by many processors in order to achieve good performance and to minimize the execution time. Once they are divided, these tasks need to communicate in order to exchange data and results of their execution.

Communication operations can be divided in two types depending on how many processes are participating. If the communication involves a single source and a single destination, this type is called point-to-point; on the other hand, if communication involves more than one source and/or destination, this type is called group communication.

DECK (Distributed Execution and Communication Kernel) [1-2] is an environment for parallel and distributed programming that provides basic mechanisms for multithreading and point-to-point communication, as well as a set of specialized services[3], such as load balancing and collective communication.

This paper presents the collective communication service developed for DECK [4], in terms of its basic primitives. Such primitives are quite similar to those found in MPI (Message Passing Interface) library [5], as we intend to facilitate the use of DECK’s collective communication service by users of another environment, such as PVM (Parallel Virtual Machine) [6] and MPI.

Additionally, it is presented a case study on numerical methods, such as Conjugate Gradient (GC) and Generalized Minimal Residuals (GMRES) in order to better evaluate DECK in comparison with other environments.

2. Collective Communication

Collective or group communication is a paradigm widely used in parallel applications generally composed by groups of processes that manipulate numerical data arranged in arrays [7] or are in charge of control operations within distributed systems.

2.1. Group Organization

Group communication can be organized in two ways. One is the open group, in which its members can receive messages sent by processes not belonging to the group. The other way is the closed group, in which the communication is allowed only to processes belonging to the group (members).

Another important feature is the internal structure of the group. In some groups, all decisions are made collectively. In other groups, some kind of hierarchy exists. For example, one process is the coordinator and all the others are workers. In this model, when a request for work is generated, either by an external client or by one of the workers, it is sent to the coordinator. The coordinator then decides which worker is best suited to carry it out, forwarding the request.

Each organization has its own advantages and disadvantages. The peer group is symmetric and has no single point of failure. If one process crashes, the group simply becomes smaller, but can continue working. The major disadvantage is that decision is more complicated. To decide anything, a vote has to be taken, incurring some delays and message overheads.

The hierarchical group has the opposite properties. The presence of a coordinator makes the group more fail prone, as the coordinator is responsible to make decisions.

The collective communication service designed for DECK, described later, supports the second pattern of communication, in which a root process is established and the other ones are slave processes.
2.2. Primitives

Group communication is commonly defined in terms of a set of primitives, such as broadcast, multicast and barrier, used to distribute data among processes and to realize global synchronization. Besides these primitives, a group communication service must provide routines for mathematical operations, such as reduce and allreduce. PVM and MPI [8,9,10] are the major examples of programming environments that provide such kind of functionalities.

3. DECK Parallel Programming Environment

DECK – Distributed Execution and Communication Kernel – is a programming environment aimed at high performance computing. It is composed of a runtime system, an user API that provides a set of abstractions and specialized services that are needful in a distributed application. A DECK application runs based on the SPMD model, in which there is one process copy on each node used in the cluster.

![Figure 1. Internal organization of DECK.](image)

DECK is structured in two layers: µDECK and the service layer, as shown in Figure 1. The µDECK interacts directly with the underlying hardware, operating system and communication protocols in order to provide communication, synchronization and multithreading facilities; therefore, this layer manages the following abstractions: threads, semaphores, messages, mailboxes and shared segments. The service layer implements a set of more specialized facilities, such as group communication and fault tolerance, which are used on demand by the applications.

Currently, we have developed 4 versions of the DECK environment, each one directed to a specific communication protocol, namely TCP/IP, BIP, GM and SISCI. Each implementation has its own characteristics, such as latency and bandwidth curves, issues related to message management, flow control, and so on [11, 12].

3.1. Mail Boxes

The DECK communication model is based on an indirect communication scheme. Data is not sent directly to the destination process; it is first stored in a structure called mail box. This mechanism works as follows: the sender writes the data message in a mail box and then the receiver can access the mail box and read the message.

In all communications using DECK, the mail box that will store the messages must be created before to be used. To send data to other nodes the sender must have previous information about the receiver mail box. These information is obtained by using the primitive `deck_mbox_clone()`, that joins the essential data the sender needs to send a message to the receiver’s mail box.

Only the creator of a mail box is allowed to retrieve messages from it, but any other thread/process knowing the mail box can post to it. To use a mail box the creator must register it in a naming server. There are two ways to obtain a mail box address: fetching it in the name server or receiving it in a message.

Internally the mail box behaves like a FIFO queue, where the first message stored into the mail box will be the first message delivered to the user buffer. In order to send or receive messages, the user must perform the necessary pack and unpack procedures. It is important to notice that a message structure can receive more than one message, but the size of message received inside a mail box can not be larger than the message structure size.

3.2. DECK Collective Communication

The service layer of DECK provides different services to the applications, including the group communication one. This service is responsible to allow the creation of a static group of threads and the communication within the group.

Figure 2 presents the group communication API used by DECK.

```c
int deck_group_create();
int deck_group_destroy();
int deck_collective_barrier();
int deck_collective_bcast();
int deck_collective_scatter();
int deck_collective_gather();
int deck_collective_reduce();
int deck_collective_allgather();
```

![Figure 2. Group communication API.](image)

Within the scope of this paper, the following routines are important:
- Barrier: the routine `deck_collective_barrier()` blocks the caller until all members in the group call the routine.
- Broadcast: the routine `deck_collective_bcast()` broadcasts a message from a process called root to all processes in the group.
Scatter: the routine `deck_collective_scatter()` allows the root process to divide a message according to the number of group members; so that the message size is the same for all processes, and so, to send the respective message piece to each one.

Gather: the routine `deck_collective_gather()` allows each process in the group to send data to the root process.

Reduce: the routine `deck_collective_reduce()` allows the root process to do an operation (such as sum, subtract, multiply and divide) with data from other members.

AllGather: the routine `deck_collective_allgather()` allows all processes to send data to all processes. The idea is to do a gather operation and after broadcasts the result.

AllReduce: this routine allows all processes to do the same operation (such as sum, subtract, multiply and divide) with data from other members. The idea is to do a reduce operation and after broadcasts a message.

Some of these primitives, such as `barrier` and `broadcast` use a binary tree-based algorithm, in which a root node is responsible to send data to its child nodes, and then, each child node forwards the message to its child nodes, and so on, until the data arrives in the leaf nodes (last level in the tree hierarchy).

4. Performance Analysis

In order to evaluate the collective communication service designed for DECK, some experiments were made using the primitives described in the previous section (except for `broadcast`, `allreduce` and `allgather`), comparing its execution time with similar primitives from the MPICH library.

Additionally, an application was developed which uses DECK and MPICH collective communication primitives in order to aid in hydrodynamics and mass transportation models, through numerical methods for domain partitioning in clusters of PCs.

4.1. Hardware

Simulations were done at LabTeC/Dell/UFRGS. The features of the cluster used were: one server with 2 Xeon 1.8 GHz processors and 20 nodes with Dual Pentium III 1.13 GHz, both with 1 GB of RAM memory. The operating system is Debian Linux, with kernel 2.4.

4.2. Simulations

Pairs of similar applications, one with MPI primitives and the other one with DECK primitives were executed, in order to get a comparative of performance. Both implementations use the TCP/IP stack. Each simulation was executed around a hundred times to get better accuracy. The number of nodes was increased from 2 to 20, and we have chose 4 message sizes to analyze the application's behavior, respectively 10, 100, 1K and 10K bytes.

<table>
<thead>
<tr>
<th></th>
<th>MPI</th>
<th>DECK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Msg size (bytes)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>11</td>
<td>100</td>
</tr>
<tr>
<td>1K</td>
<td>16</td>
<td>1K</td>
</tr>
<tr>
<td>10K</td>
<td>52</td>
<td>10K</td>
</tr>
</tbody>
</table>

4.3. Scatter and gather evaluation

Table 1 shows the results obtained from a sequence of scatter and gather primitives considering only one node (primitive's processing time). It is possible to observe that DECK presents better results than MPI, possibly due to the internal organization and structures used in each environment.

Figure 3 shows the results from a sequence of scatter and gather operations with a message of 1K bytes considering different number of nodes. It is possible to observe that the behavior of both libraries is very similar, with a small advantage to DECK.
4.4. Reduce evaluation

Table 2 shows the results obtained from reduce primitive considering only one node (primitive’s processing time). It is possible to observe that DECK presents results very close of MPI.

Table 2. Processing time for reduce primitive.

<table>
<thead>
<tr>
<th></th>
<th>MPI</th>
<th></th>
<th>DECK</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Msg size (bytes)</td>
<td>Time (µs)</td>
<td>Msg size (bytes)</td>
<td>Time (µs)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>100</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1K</td>
<td>3</td>
<td>1K</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>10K</td>
<td>23</td>
<td>10K</td>
<td>29</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4 shows the results from reduce operation with a message of 10 bytes considering different number of nodes. It is possible to observe that DECK presents better performance than MPI.

Figure 5 shows the results from reduce operation with a message of 100 bytes considering different number of nodes. Again, DECK presents better performance than MPI.

4.5. Barrier evaluation

The last measure is the barrier synchronization time, varying from 2 to 20 nodes. Figure 8 shows the obtained curves. It is possible to observe that DECK presents results very similar to MPI for a number down to 6 nodes. After this, a small disadvantage for DECK could
be observed, as a consequence of its barrier implementation.

When a node calls a `deck_collective_barrier()` primitive, it sends to its father node an empty message. The father node does a local barrier, to receive messages from its children; and after that, forwards an empty message to its father node, until the message arrives in the root process. At this time, the root process starts to unblock all group processes.

![Figure 8. Execution time for barrier.](image)

5. Case study on numerical methods

In order to better evaluate the collective communication service designed for DECK, we have used it in the implementation of two parallel numerical methods running on clusters of PCs [13]. The goal is the development and use of techniques and structures to parallelize Krylov subspace iterative methods, which are Conjugate Gradient (GC) and Generalized Minimal Residuals (GMRES). They can be used, respectively, to the solution of linear equations systems SDP (Symmetric Definite Positive) and not symmetrical. Such systems arise from the discretization of hydrodynamics and mass transportation models being developed in GMCPAD (Group of Computing Mathematics and High Performance Computing) [14]. The use of these methods is justified by the fact that, being iterative, they are well suitable to the solution of sparse equations systems.

In the solution of these systems through parallelized iterative methods, it is necessary to divide the problem domain, which is done looking for a good load balancing and the minimization of frontiers between the sub-domains. The data structure developed for this application allow them to be adopted to the solution of equation systems generated from any kind of partitioning, as the format used to store the data supports any type of data dependency. The Guaíba lake was taken as a case study domain (Figures 9 and 10).

![Figure 9. Guaíba lake.](image)

![Figure 10. Discrete domain representation of Guaíba lake.](image)

5.1. Partitioning

There are two kinds of domain partitioning models adopted in this work. The first one is based in band decomposition, a model that results in simple and low computational cost algorithms; and the other one is an approach for irregular block structured domains. From the former class, we used two examples: S-STRIP (Straight Stripwise Partitioning) and STRIP (Stripwise Partitioning); and from the latter class, we used the RCB (Recursive Coordinate Bisection) algorithm.

The RCB algorithm got better results in load balancing, number of frontiers and minimization size generated. This algorithm admits a little unbalancing, acquires better relation among sub-domains areas and perimeters, but has an expensive computational cost.

The RCB idea consists of analyze the domain or sub-domain and divide it in the orthogonal direction of the larger dimension, reducing the frontiers size. Consequently, this division will produce two news sub-domains with similar load. Figure 11 illustrates four phases of domain partitioning by 2 to 8 sub-domains. An enlarged view of the final result produced by the partitioning showed in Figure 11 can be seen in Figure 12.

![Figure 11. RCB application in the Guaíba domain.](image)
For matrix storage, CSR (Compressed Sparse Row) form is used, which has good performance in large and sparse systems generated by the discretization method. Linear algebra operations including vectors and matrix comprise the GC and GMRES methods. Therefore, the parallelization has to be done in these operations [15]. The dot product and vector operations demand process communication in order to supply data dependency for different sub-domains. For dot product operations it is required communication among all processes involved. Each process calculates the dot product with its data and, after that, a reduce operation including all process is executed.

Nevertheless, for multiplication of sparse matrix by vector, each process needs information about parts of the vectors residing in other processes. It is necessary because, during the matrix generation process, data dependencies related to the frontier cells are generated. The solution adopted to solve this problem was to keep the matrix elements that are multiplied by remote elements (i.e., elements from other domains) in disjoint control structures, used to store input and output data.

In this way, this multiplication can be divided in two parts: $Ax = Ax_{int} + Ax_{ext}$, where $Ax_{int}$ corresponds to the part that considers only local elements, and $Ax_{ext}$ corresponds to the part that considers remote (not locally available) elements.

### 5.2. Performance of Krylov subspace

The execution time versus number of nodes is showed in Figures 13 and 14. It is a comparative analysis among DECK, MPI (blocking) and MPI (nonblocking). Tests were done changing the number of machines from 1 up to 16 nodes. DECK achieves better performance than MPI for any number of nodes. Another result observed is that GC method has better speedup than the GMRES method. This is a consequence of the communication overhead presented by GMRES, as it is necessary several operations of global communication in order to manipulate scalar products from Gram-Schmidt’s process.

Figures 15 and 16 present the speedup obtained for DECK and MPI. It is possible to observe that DECK achieves better performance than MPI for any number of nodes. Another result observed is that GC method has better speedup than the GMRES method. This is a consequence of the communication overhead presented by GMRES, as it is necessary several operations of global communication in order to manipulate scalar products from Gram-Schmidt’s process.
6. Conclusion

Group communication is quite useful in a wide range of parallel applications that need to manipulate numerical data. PVM and MPI libraries are major examples of programming environments that provide primitives for collective communication, and they are used by several researchers to application programming.

The intention of this work is to evaluate a collective communication service proposed to the DECK environment, by means of a direct comparison with MPI primitives and the development of a real application.

For reduce operations, DECK has obtained better results than MPI, as the implementation of both libraries differ enormously; however, for barrier synchronization and scatter/gather operations, this evaluation has showed that DECK’s primitives need to be reviewed and enhanced when the message size pass 10K bytes. For parallel numerical methods, such as GC and GMRES, DECK has showed a little better execution time than MPI.

Currently, we are working on barrier and scatter/gather tuning and also in the integration of the group communication and the fault tolerance services, in order to add reliable and fail-safe characteristics to the group communication service.

7. References


