PAPER Special Section on Hardware/Software Support for High Performance Scientific and Engineering Computing

Programming Support for MPMD Parallel Computing in ClusterGOP

Fan CHAN[†], **Jiannong CAO**[†a)], **Alvin T.S. CHAN**[†], ***Nonmembers***, ***and*** **Minyi GUO**[††], ***Member***

SUMMARY Many parallel applications involve different independent tasks with their own data. Using the MPMD model, programmers can have a modular view and simplified structure of the parallel programs. Although MPI supports both SPMD and MPMD models for programming, MPI libraries do not provide an efficient way for task communication for the MPMD model. We have developed a programming environment, called ClusterGOP, for building and developing parallel applications. Based on the graph-oriented programming (GOP) model, ClusterGOP provides higher-level abstractions for message-passing parallel programming with the support of software tools for developing and running parallel applications. In this paper, we describe how ClusterGOP supports programming of MPMD parallel applications on top of MPI. We discuss the issues of implementing the MPMD model in ClusterGOP using MPI and evaluate the performance by using example applications.

key words: *parallel computing environments, graph-oriented programming, ClusterGOP, VisualGOP, MPMD, MPI*

1. Introduction

Many parallel applications are developed using the popular SPMD (single program multiple data) model. Using the SPMD model has advantages on program design for small applications, including efficiency in building and running the applications and convenience with the use of sequential programming techniques. However, because a single program source needs to include all the tasks for all processors in the application, a SPMD program is hard to understand and to change. Therefore, when the application becomes large and complex, especially with heterogeneous computations requiring irregular or unknown communication patterns, MPMD (multiple program multiple data) model is preferred [1]. MPMD separates the application into different functional modules with separate program sources for concurrent tasks, thus promoting code re-use and the ability to compose programs. It is also well suited for metacomputing in a heterogeneous, large-scale environments because MPMD programs are a lot more loosely-coupled than those written using the SPMD model.

However, MPMD programming is difficult in most situations without good support. For example, the Message Passing Interface (MPI) [2] is mainly for SPMD programming. Although it has support for MPMD environment, the support is not sufficient for programmers to take advantages of developing MPMD programs. Currently, most of the MPI implementations do not support automatic runtime configuration for the MPMD environment. Programmers must manually create a processor list and then submit the correct initial arguments each time when creating the application. Moreover, MPI does not provide a complete library to support communication and data sharing among different tasks. MPI communication library mainly supports the SPMD model so that a program using collective operations cannot be re-written as a MPMD program easily. Also, to form a new communication group, the programmer needs to make effort to produce additional code for group initialization and creation.

In our previous work, we have developed a programming environment, called ClusterGOP, for building and developing parallel applications. ClusterGOP is based on the graph-oriented programming (GOP) model [3], which provides a high-level programming abstraction for building message passing parallel applications. Programs can be constructed using a user-defined logical graph. Graph-oriented primitives for structured communications, synchronization and configuration are perceived at the programming-level and their implementation hides the programmer from the underlying low-level programming activities. The programmer can thus concentrate on the logical design of an application, ignoring unnecessary details. ClusterGOP supports both SPMD and MPMD parallel computing. In a previous paper, we described how ClusterGOP implements its SPMD model on top of MPI [4]. In this paper, we describe a methodology for MPMD programming support in the ClusterGOP environment.

We have also developed a visual tool in ClusterGOP, called VisualGOP [5], which provides graphical and intelligent support for the design and construction of parallel programs. It has a visual and interactive user interface, and provides a framework in which the design and coding of GOP programs, and the associated information can be viewed and modified easily and quickly. VisualGOP provides several useful facilities for MPMD programming, such as visualization for data distribution, NodeGroups and resource management. We will describe how to use VisualGOP to build and deploy MPMD applications in CluserGOP. As we will see, with these useful functions, the task of MPMD programming becomes much easier and the efficiency of the application development can be improved.

The rest of this paper is organized as follows. Section 2 describes the background and related work. Section 3 intro-

Manuscript received October 15, 2003.
Manuscript revised January 15, 2004.

[†]The authors are with the Department of Computing, Hong Kong Polytechnic University, Hung Hom, Hong Kong.
[††]The author is with the Department of Computer Software, University of Aizu, Aizuwakamatsu-shi, 965–8580 Japan.
a) E-mail: csjcao@comp.polyu.edu.hk

duces the ClusterGOP programming framework. Section 4 describes how to provide the support for MPMD programming in ClusterGOP. Section 5 shows the experiment results using example applications. Finally, Sect. 6 concludes the paper with the discussion of our future work.

2. Background and Related Work

In parallel computing, the design and implementation of message-passing applications have been recognized as complex tasks. Message passing programming libraries such as PVM [6] and MPI have improved the situation, as they provide support for implementing parallel applications to run independently on underlying architecture. PVM and MPI allow for the general form of parallel computation, as programs may exhibit arbitrary communication dependencies. In general, programs forming tree inter-process communication dependencies, where each process communicates only with its parent and its child processes, are well suited to PVM, while regular ring or grid process communication dependencies are well suited to MPI. However, as we know, PVM and MPI are low-level tools and somewhat difficult to use for building applications. Even with some new and enhanced features in MPI-2 [7], problems still exist. The step from application design to implementation remains a demanding task. Several high-level programming models are being well developed.

Ensemble [8]–[10] supports the design and implementation of message-passing applications (running on MPI and PVM), particularly MPMD and those demanding irregular or partially regular process topologies. Also, the applications are built by composition of modular message-passing components. Ensemble divided the software architecture into two layers: the Abstract Design and Implementation (AD&I), which is the responsibility of the programmer, and the Architecture Specific Implementation (ASI), e.g. MPI implementation, which is generated from the AD&I and transparent to the developer. AD&I consists of three well-separated implementation parts. The first one is virtual component. It is the implementation abstraction of a MP program. For example, it provides abstract names and abstract roots for collective calls and abstract point-to-point interaction. It also uses "ports" to replace the MPI argument types (context, rank and message tag). The second one is symbolic topology. It is an abstraction of a process topology, which specifies the number of processes required from each component, each process's interface and its interaction with other processes. The last one is resource allocation. It is the mapping of processes, as well as the location of source, executable, input and output files in the underlying environment.

There are some other high-level MPMD languages (e.g. Mentat [11], C++ [12] and Fortran-M [13]) and runtime systems (e.g. Nexus [14]), which support combination of dynamic task creation, load balancing, global name space, concurrency, and heterogeneity. Due to the need for crossing program domains, for asynchronously detecting incoming communication, and for potentially spawning new threads, the communication overheads in these systems are often prohibitively high for a multi-computer system. Programming platforms based on the SPMD model (e.g. Split-C [15] and CRL [16]) have a significant performance advantage over MPMD-based ones and are preferred for parallel application development. However, there exists MPMD systems (e.g. MPRC [1], [17]) using RPC as the primary communication abstraction which produce good results compared with the SPMD model.

Many parallel systems use message passing as the basic mechanism for their implementation of communication for MPMD programs. Some systems integrate message passing with other parallel paradigms, such as the data parallel approach, to enhance the programming support and take advantages of different paradigms. The Nanothreads Programming Model (NPM) [18] is a programming model for shared memory multiprocessors. The NPM can integrate with MPI and the runtime system is based on a multilevel design that supports both models (NPM and MPI) individually but offers the capability to combine their advantages. Existing MPI codes can be executed without any changes and codes for shared memory machines can be used directly, while the concurrent use of both models is easy. The major feature of the NPM runtime system is portability, as it is based exclusively on calls to MPI and Nthlib, a user-level threads library that has been ported to several operating systems. The runtime system supports the hybrid-programming model (MPI + OpenMP [19]). Moreover, it extends the API and the multiprogramming functionality of the NPM on clusters of multiprocessors and can support an extension of the OpenMP standard on distributed memory multiprocessors. Another example is Global Arrays (GA) [20] which allows programmers to easily express data parallelism in a single, global address space. GA provides an efficient and portable shared-memory programming interface for parallel computers. The use of GA allows the programmer to program as if all the processors have access to the same data in shared memory.

The MPMD programming support in ClusterGOP to be described in this paper is built on MPI, using MPI as the basic communication facility and adding the GOP construct for high-level programming support. The approach used in GA is adopted to support distributed memory operations. Details can be found in Sect. 4.

3. The ClusterGOP Framework

3.1 Programming Model of ClusterGOP

ClusterGOP is based on the GOP model, which was proposed for message-passing based parallel and distributed computing. In applying GOP to parallel programming, it was our observation that many parallel programs can be modeled as a group of tasks performing local operations and coordinating with one another over a logical graph, which depicts the program configuration and inter-task communication pattern of the application. Most of the graphs are reg-

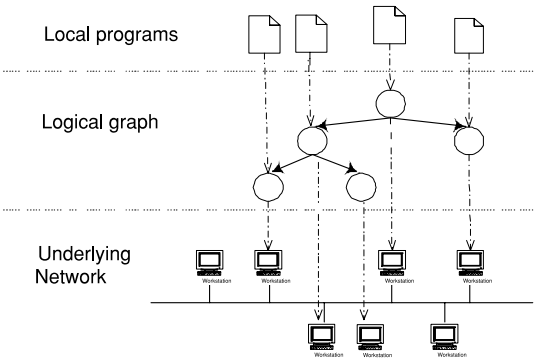


Fig. 1 The GOP conceptual model.

ular ones such as tree and mesh. Using a message-passing library, such as PVM and MPI, the programmer needs to manually translate the design-level graph model into its implementation using low-level primitives. With the GOP model, such a graph metaphor is made explicit at the programming level by directly supporting the graph construct in constructing the program. The tasks of a parallel program are configured as a logical graph and implemented using a set of high-level operations defined over the graph.

In GOP, a parallel program is defined as a collection of local programs (LPs) that may execute on several processors [3], [21]. Parallelism is expressed through explicit creation of LPs and communication between LPs is solely via message-passing. GOP allows programmers to write parallel programs based on user-specified graphs, which can serve the purpose of naming, grouping and configuring LPs. It can also be used as the underlying structure for implementing uniform message-passing and LP co-ordination mechanisms.

The GOP model provides high-level abstractions for programming parallel programs, easing the expression of parallelism, configuration, communication and coordination. The key elements of GOP are a logical graph construct to be associated with the LPs of a parallel program and their relationships, and a collection of functions defined in terms of the graph and invoked by messages traversing the graph. As shown in Fig. 1, the GOP model consists of the following parts:

- A logical graph (directed or undirected) whose nodes are associated with local programs (LPs), and whose edges define the relationships between the LPs.
- An LPs-to-nodes mapping, which allows the programmer to bind LPs to specific nodes.
- An optional nodes-to-processors mapping, which allows the programmer to explicitly specify the mapping of the logical graph to the underlying network of processors. When the mapping specification is omitted, a default mapping will be performed.
- A library of language-level graph-oriented programming primitives.

GOP programs are conceptually sequential but augmented with primitives for binding LPs to nodes in a graph,

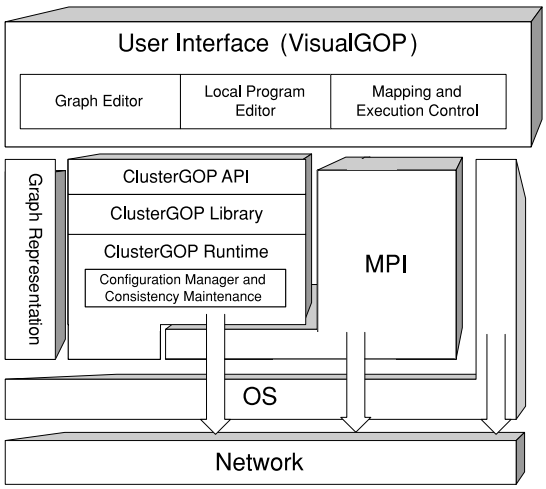


Fig. 2 The ClusterGOP framework.

with the implementation of graph-oriented inter-node communications completely hidden from the programmer. The programmer first defines variables of the graph construct in a program and then creates an instance of the construct. Once the local context for the graph instance is set up, communication and coordination of LP's can be implemented by invoking operations defined on the specified graph. The sequential code of LPs can be written using any programming language such as C, C++ and Java.

3.2 The ClusterGOP Architecture

The ClusterGOP software environment is illustrated in Fig. 2. The top layer is VisualGOP, the visual programming environment [5]. The ClusterGOP API is provided for programmer to use, so that the programmer can build applications based on the high-level GOP model, ignoring the details of low-level operations. The ClusterGOP library provides a collection of routines implementing the ClusterGOP API, with a very simple functionality to minimize the package overhead.

The ClusterGOP Runtime is responsible of compiling the application, maintaining the logical graph structure, and executing the application. On each machine, there exists two runtimes. The first one is the ClusterGOP runtime, a background process that provides graph deployment, update, query and synchronization. When deploying and updating the graph, it will block other machines to further update the graph and synchronize the graph update on all machines. Another is the MPI runtime, which provides a parallel programming library for the ClusterGOP implementation. ClusterGOP uses MPI as the low-level parallel programming facility so that processes can communicate efficiently.

4. Support for MPMD Programming in ClusterGOP

With the MPMD programming model under ClusterGOP,

each LP is associated with separate source code. Data can be distributed and exchanged among the LPs. ClusterGOP also has a better node group management than MPI so that the processes can form groups easily and efficiently. In this section, we describe the methodology for MPMD programming support in the ClusterGOP environment. We focus on the new features added to VisualGOP to support high-level MPMD programming, including forming process groups, data distribution among the processes and deployment of processes for execution. With these new features, programmers can program group communication by using NodeGroup, manage distributed tasks and processors through visual interface, map resources to tasks, and compile / execute programs automatically. The underlying implementation using MPI is hidden from the programmer.

4.1 NodeGroup Management

In ClusterGOP, we use a NodeGroup to represent a group of processes, providing the same functionality as the MPI communicator. NodeGroup helps the programmer to write code for group communications. A NodeGroup consists of the member processes. Each process in the group can invoke group communication operations such as collective communications (gather, scatter, etc).

In the design phase, VisualGOP provides a visual way for representing the NodeGroup involved in a MPMD program. The programmer can highlight the NodeGroup in the logical graph to specify the nodes that belong to the NodeGroup, as shown in Fig. 3. When the programmer wants to use the collective operations, VisualGOP provides the valid NodeGroup for programmer to select. It also hides the programming details that are used for constructing the NodeGroup in the MPMD programs so that the programmer can concentrate on programming the nodes' tasks. As a result, the program is easier to understand.

As shown below, NodeGroup has simple APIs that are easy to use. In forming a group, the programmer first forms a task group, assigns a name to the group, and then adds nodes to or removes nodes from the NodeGroup.

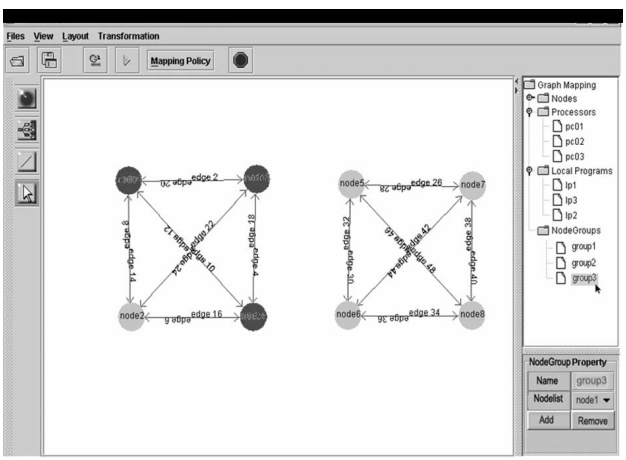


Fig. 3 The NodeGroups in VisualGOP.

```
/* create the NodeGroup from a list of nodes */
NodeGroup InitNodeGroup (char* group_name, Node[] nodelist);
/* add one node to the NodeGroup */
int AddNode (NodeGroup *ng, Node s);
/* remove one node from the NodeGroup */
int RemoveNode (NodeGroup *ng, Node s);
/* clear all nodes in the NodeGroup */
void ClearNodeGroup (NodeGroup *ng);
```

Once NodeGroup is created, it can be used in the ClusterGOP collective communication operations. The following are some example functions:

```
/* sending multicast message */
MsgHandle Msend(Graph g, NodeGroup ng, Msg msg, CommMode m);
/* receiving multicast message */
MsgHandle Mrecv(Graph g, NodeGroup ng, Msg msg);
/* s collect data from all nodes in the NodeGroup */
MsgHandle Gather(Graph g, NodeGroup ng, Msg msg, Node s);
/* s distribute data to all nodes in the NodeGroup */
MsgHandle Scatter(Graph g, NodeGroup ng, Msg msg, Node s);
/* data collection in all nodes in the NodeGroup */
MsgHandle Allgather(Graph g, NodeGroup ng, Msg msg);
/* data distribution in all nodes in the NodeGroup */
MsgHandle Alltoall(Graph g, NodeGroup ng, Msg msg);
/* reduce data to s from all nodes in the NodeGroup */
MsgHandle Reduce(Graph g, NodeGroup ng, Msg msg, Node s);
/* data reduction in all nodes in the NodeGroup */
MsgHandle Allreduce(Graph g, NodeGroup ng, Msg msg);
```

ClusterGOP's NodeGroup is implemented using MPI's communicator. The basic functions of a communicator include managing processes, defining scope of process communication, and communication between communicators. When two processes do not belong to the same communicator, they cannot send or receive information from each other. When the parallel application starts, a default communicator, namely MPI_COMM_WORLD, is created. By default, all processes belong to the MPI_COMM_WORLD can communicate with each other. Programmers can create new communicators in addition to MPI_COMM_WORLD. However, MPI does not provide an easy way to create a new communicator. The example below shows how MPI creates groups and communicators inside the program.

```
MPI_Group MPI_GROUP_WORLD, first_row_group;
MPI_Comm first_row_comm;
int row_size;
int* process_ranks;

process_ranks = (int*) malloc (q*sizeof(int));
for (proc=0; proc<q; proc++)
   process_ranks[proc] = proc;
/* create the group from all processes */
MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
/* select only the process by including the rank no. in the */
/*   process ranks */
```

```
MPI_Group_incl(MPI_GROUP_WORLD, q, process_ranks,
                &first_row_group);
/* create a new communicator according to the group */
MPI_Comm_Create(MPI_COMM_WORLD, first_row_group,
                &first_row_comm);
```

To introduce a new communicator into the application, MPI requires that an MPI group be created to store the neighbors in an array of rank ID. Therefore, the programmer needs to write the corresponding code, remembering every rank ID in the new communicator. This decreases the readability of the program and increases its complexity. In contrast, NodeGroup simplifies the procedure for building the communication group and provides better handling of group communication. The following code segment shows the creation of the NodeGroup.

```
NodeGroup InitNodeGroup(char *group_name, Node nodes[]) {
  int i=0;
  NodeGroup group;

  strcpy(group.name, group);
  while (nodes[i] != NULL) {
    group.nodes[i] = nodes[i];
    group.nodeids[i] = getNodeID(nodes[i]);
    group.size++;
    i++;
  }
  return group;
}
int getNodeID(Node gn) {
  int rank_id=0; /* This variable stores the rank ID */

  for (rank_id=0; rank_id<global_node_count; rank_id++)
    if (strcmp(gn, nodes[rank_id].name) == 0)
      return rank_id;
    return -1;
}
```

InitNodeGroup() is the API for programmer to initialize the NodeGroup from a list of nodes. The function stores the name and rank ID of the each node into the NodeGroup data structure. For getting the rank ID, the function simply invokes getNodeID() to retrieve it.

The programmer can add, remove or clear the nodes in the NodeGroup by invoking the corresponding API functions. The implementation of function AddNode() is shown below.

```
AddNode(NodeGroup *ng, Node s) {
  int i=ng->size;

  strcpy(ng->nodes[i], s);
  ng->nodeids[i] = getNodeID(s);
  ng->size++;
}
```

After a NodeGroup is created, the function createGroup() generates the MPI_Group object (group_world) by providing the NodeGroup information to MPI function MPI_Group_incl(). Then, the MPI_Group object is passed to the function MPI_Comm_create() to form a new MPI communicator.

```
MPI_Comm createGroup(NodeGroup ng) {
  MPI_Group group_world, new_group;
  MPI_Comm new_comm;

  MPI_Comm_group(MPI_COMM_WORLD, &group_world);
  MPI_Group_incl(group_world, ng.size, ng.nodeids, &new_group);
  MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
  return new_comm;
}
```

4.2 Support for Data Distribution

In the MPMD programming model, tasks have different programs to execute but usually need to exchange or share some data. MPI provides API functions for distributing data to different processes, but the programmer still has to write the code for the data distribution procedure. In ClusterGOP, tasks share data by keeping a portion of the global memory in each node that is involved in the communication. The node can update the memory without having to communicate with other nodes.

Using VisualGOP, data distribution can be performed by the programmer through the visual interface. The Distributed Shared Memory (DSM) can be created by selecting an option in the program editor of VisualGOP as shown in Fig. 4. Currently, there are three options of the distributed memory styles: vertical, horizontal and square memory distribution. By default, the memory is distributed to the nodes in a balanced way such that each node will almost share the same size of the distributed data object. VisualGOP also provides a visual interface to allow the programmer to manually specify the memory distribution on each node.

In many DSM systems, the distributed memory object is built-in function and most of the objects are distributed in the whole environment. However, due to its complex design nature, overheads occur frequently which reduce its

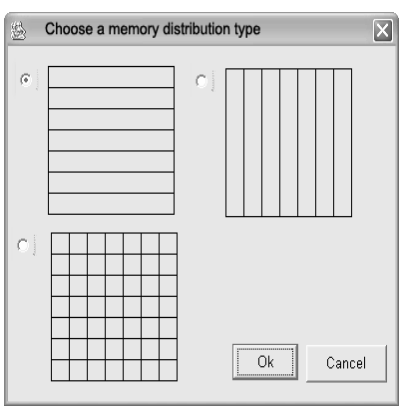


Fig. 4 Choosing memory distribution type.

efficiency. ClusterGOP implements the DSM in a different way that distributed objects are used only if programmer explicitly demands them.

In ClusterGOP, the programmer can use predefined data distribution algorithms to put the distributed objects into different parts of the program. VisualGOP also provides a visual way for the programmers to define the data distribution on tasks. ClusterGOP translates the data distribution specified by the programmer into MPI code. For example, when the programmer wants to distribute the object among several tasks, he / she can insert a distributed data object, and then select which nodes (or NodeGroup) they want to share the object and define the sharing rule. All these can be done with the aid of VisualGOP.

ClusterGOP implements the DSM functions by using the GA toolkit [20], which is based on MPI for communications. Before compiling the application, all distributed objects are converted into Global Arrays (GA) codes. GA provides an efficient and portable shared-memory programming interface. The Global Arrays installs itself on top of the MPI library, allowing programmers to take advantage of the interfaces in the MPI and Global Arrays in the same program.

4.3 Resource Mapping, Automatic Compilation and Execution Support for MPMD Programs

In MPMD programming, managing the mapping of nodes (tasks) to processors could be a complicated task. VisualGOP provides the programmer support to visually map LPs to nodes and nodes to processors (see in Fig. 5). It also provides support for automatic compilation and execution of the applications. This facilitates the development process and simplifies the running of the large-scale MPMD application.

After the program is written, through VisualGOP, the programmer can send the program source codes to the target processors in the system for compilation. When deploying the application, VisualGOP provides information about the target platform such as the address of the machines, the logical graph representation and the compilation arguments. The programmer creates a processor list for the parallel environment so that the nodes-to-processors mapping can be established. After the application has compiled, VisualGOP allows the programmer to start the application for execution. The programmer can provide some pre-defined input argument or data to the application. The final step is the execution of the root process, and VisualGOP monitors the status of the root process and receives the feedback from it.

A ClusterGOP daemon process runs on each target processor to receive the incoming compilation and execution requests and interpret them as system commands.

5. Performance Evaluation

In this section, we describe two example applications to show how ClusterGOP and VisualGOP support the construction and running of MPMD parallel programs. The examples are Parallel Matrix Multiplication and Two-Dimensional Fast Fourier transform (2-D FFT). We also evaluate their performance and compare it with the standard MPI versions of the programs.

5.1 Parallel Matrix Multiplication

In this example, we consider the problem of computing C = A×B, where A, B, and C are dense matrices of size N×N (see Eq. (1)).

$$C_{ij} = \sum_{k=0}^{N-1} A_{ik} * B_{kj} \tag{1}$$

As shown in Fig. 6, a 3×3 mesh is defined. Besides N×N nodes in the mesh, there is an additional node, named "master", which is connected to all the nodes of the mesh. There are two types of programs in this example: "distributor" and "calculator". There is only one instance of "distributor", which is associated with the "master" node. Each mesh node (node1 to node9) is associated with an instance of "calculator". The "distributor" program first decomposes the matrices A and B into blocks, whose sizes are

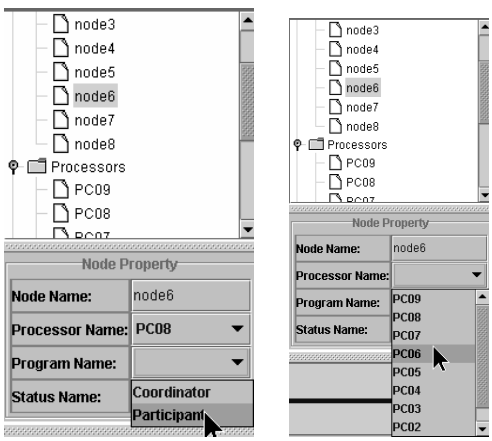


Fig. 5 LPs-to-nodes and nodes-to-processors mapping.

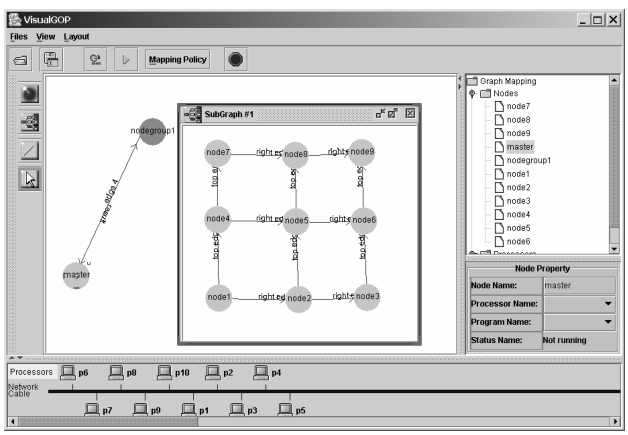


Fig. 6 VisualGOP for parallel matrix multiplication.

determined by the mesh's dimension. It then distributes the blocks to the nodes on the left most column (node1, node4 and node7) and the nodes on the bottom rows (node1, node2 and node3), respectively. Each "calculator" receives a block of matrix A and matrix B from its left edge and bottom edge, and also propagates the block along its right edge and top edge. After data distribution, each "calculator" calculates the partial product and sends it back to the "distributor". The "distributor" assembles all the partial products and displays the final result.

The programmer has support by VisualGOP and ClusterGOP in building this application:

- `Data distribution`. In writing the code for "distributor", the programmer must decompose the matrix A and B into blocks. Instead of writing the code manually, the programmer can select the data distribution option in VisualGOP. VisualGOP automatically generates the distributed object by using the API provided by GA toolkit.
- `MPMD representation`. In MPI, the SPMD version of the program needs to calculate the rank ID of the destination node. The MPMD program written in ClusterGOP, however, can be separated into tasks and each node is associates with an independent program source. In ClusterGOP, the function GetNode() returns the destination node by the edge, so that the program is not required to calculate the rank ID. The code structure is simplified and programmers can have a clear view of the application logic. The following code segments show the difference between the MPI SPMD and the ClusterGOP MPMD programs.

```
/* MPI SPMD program */
/* MPI needs to get the msg tag and size before */
/*   receiving the msg */
  MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
            &status);
  rcv_nid = status.MPI_SOURCE;
  MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
            &status);
  rcv_nid = status.MPI_SOURCE;
  tag = status.MPI_TAG;
  MPI_Get_count(&status, MPI_INT, &num);
  MPI_Recv(buf, num, MPI_INT, rcv_nid, tag, MPI_COMM_WORLD,
            &status);
  if(tag<1000){ /* Matrix A */
    ...  /* processing received data */
    if(nid % msize != 0)
      MPI_Send(buf, num, MPI_INT, nid+1, tag,
               MPI_COMM_WORLD);
  }
  else { /* Matrix B */
      ... /* processing received data */
      if((int)(nid-1)/msize != msize-1)
          MPI_Send(buf, num, MPI_INT, nid+msize, tag,
                   MPI_COMM_WORLD);
  }
```

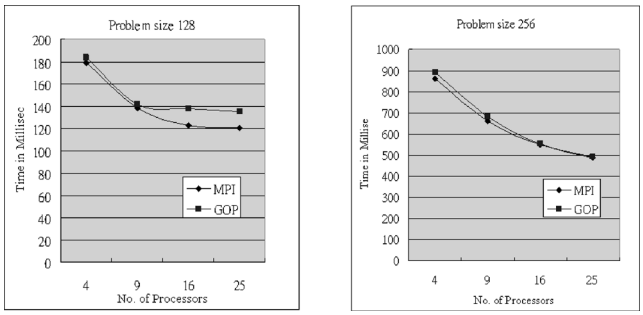


Fig. 7 Execution time per input array for the parallel matrix multiplication application.

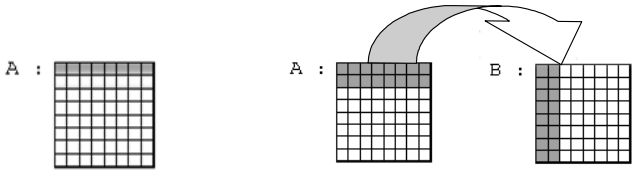
```
/* ClusterGOP MPMD program on node1 */
  for(i=0; i<2; i++){
    Urecv(ggraph, GOP_ANY_NODE, msg1);
    if(tag<1000){ /* Matrix A */
      ... /* processing received data */
      Usend(ggraph, GetNode("rightn1"), msg1, ASYN);
  }
  else { /* Matrix B */
    ... /* processing received data */
    Usend(ggraph, GetNode("top edge"), msg1, ASYN);
  }
}
/* ClusterGOP MPMD program on node9 */
  for(i=0; i<2; i++){
    Urecv(ggraph, GOP_ANY_NODE, msg1);
    if(tag<1000){ /* Matrix A */
      ... /* processing received data */
    }
    else { /* Matrix B */
      ... /* processing received data */
    }
  }
```

The experiments used a cluster of twenty-five Linux workstations, and each workstation is running on Pentium-4 2 GHz. The workstations are setup with MPICH 1.2 and all the testing programs are written in C. Execution times were measured in seconds using the function MPI_Wtime(). Measurements were made by inserting instructions to start and stop the timers in the program code. The execution time of a parallel operation is the greatest amount of time required by all processes to complete the execution. We choose to use the minimum value from ten measurements.

Figure 7 shows the performance result in execution time. We can see that the MPI program runs slightly faster than the ClusterGOP program. This may be the result of conversion overheads (nodes to the rank ID) in the ClusterGOP library. However, there are no significant differences between MPI and ClusterGOP when the problem size and processor number are getting larger.

5.2 Two-Dimensional Fast Fourier Transformation

Figure 8 (a) illustrates that the program calculating 2-D FFT



(a) 2-D FFT mapping *(b) Pipelined 2-D FFT mapping*

Fig. 8 Two implementations of a 2-D FFT, the shading area indicates the elements of the array that are mapped to one processor.

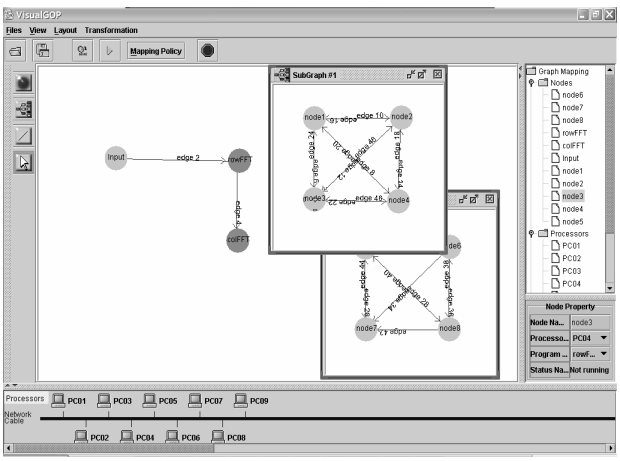


Fig. 9 Diagram for the 2-D FFT program in VisualGOP.

first calls the subroutine rowfft() (row FFT) to apply an one-dimensional (1-D) FFT to each row of the 2-D array A, and then transposes the array and calls rowfft() again to apply a 1-D FFT to each column of A. The 1-D FFTs performed within rowfft are independent of each other and can proceed in parallel. The image data structure needs to be distributed to processors. This distribution allows the calls to the rowfft() routine to proceed without communication. However, the transposition (or FFT in column) involves all-to-all communication.

An alternative pipelined algorithm is often more efficient (see in Fig. 8 (b)). The algorithm partitions the FFT computation among the processors such that $P/2$ processors perform the read and the first set of 1-D FFTs, while the other $P/2$ processors perform the second set of 1-D FFTs and the write. At each step, intermediate results are communicated from the first to the second set of processors. These intermediate results must be transposed on the way; as each processor set has size $P/2$, this requires $P^2/4$ messages. In contrast, the normal 2-D FFT algorithm's all-to-all communication involves P(P-1) messages, communicated by P processors: roughly twice as many per processor. In accordance with the pipelined algorithm, the application is separated into row FFT and column FFT programs. The row FFT can pass the value to column FFT and then continues work for its next data stream. As a result, the network utilization is improved and the application's performance is increased.

Figure 9 shows the logical graph of the application represented in VisualGOP. The input node sends data to the row-fft node, which contains four nodes in the subgraph. The col-fft node also contains four nodes. The corresponding programs are mapped to the nodes, and the nodes are mapped to the processors for execution. VisualGOP and ClusterGOP provide several advantages:

- `MPMD representation`. An FFT program is divided into several parts as mentioned above. Each node is associated with a separated program and mapped to a processor to run. There are totally eight processors involved in the computation. Four processors are used for computing the row FFT and others are used for computing the column FFT.
- `NodeGroup formation`. The nodes are classified into two NodeGroups, namely the row FFT group and column FFT group. The two node groups communicate with each other. The row FFT group collects the result and then sends it to the column FFT group.
- `Data distribution`. To simplify the process communication, a distributed memory object is used in each NodeGroup (row FFT group and column FFT group). It provides a better way to share data between the processes within the same NodeGroup.

In our experiment, the ClusterGOP code is executed as a pipeline of two kinds of tasks in a MPMD model, with an equal number of processors assigned to each task. The MPI code is executed as a single SPMD program. The experiments used a 24-processor SGI Origin 2000 machine, running on IRIX64 6.5, which implements the MPI 1.2 standard. The programs are written using the C language and measurements were made by inserting instructions to start and stop the timers in the program code. Execution times were measured in seconds using the function MPI_Wtime(). The execution time of a parallel operation is the greatest amount of time required by all processes to complete execution and we choose to use the smallest value from 10 measurements.

Figure 10 presents the results of the experiments, which are performed for various program sizes to render pipeline startup and shutdown costs insignificant. Again, the MPI code is faster than the ClusterGOP code when the problem size is small. This is because ClusterGOP has some overheads in processing graph access and communication. However, this effect is eliminated when the problem size increases. As expected, the ClusterGOP code is faster than the MPI code when the number of processors is getting larger.

6. Conclusions

This paper describes a high-level graph-oriented approach for programming parallel application in MPMD model. The ClusterGOP programming environment provides several tools, including VisualGOP, for provoding support to programmers for building MPMD applications. It provides a library of functions for communications, distributed shared data, and mapping of applications to underlying pro-

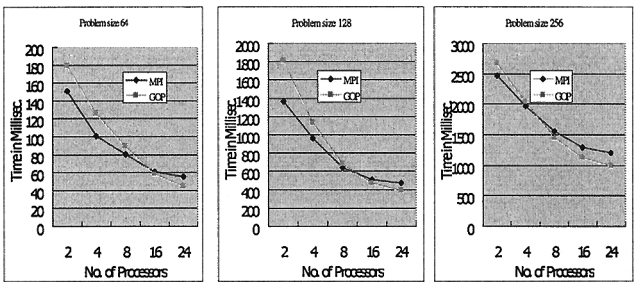


Fig. 10 Execution time per input array for the 2-D FFT application.

cessors for execution. VisualGOP provides visual interface for programmer to access these functions to develop MPMD programs in a flexible and easy way.

Future work can be done in several directions. First, VisualGOP can be enhanced so that it can monitor and manage the processes dynamically. Second, the design of the distributed memory in ClusterGOP can incooperate some data decomposition algorithms so that the application will be more balanced. Finally, more example MPMD programs will be invesigated to see how the proposed model can be applied to different applications.

Acknowledgments

This work is partially supported by Hong Kong Polytechnic University under HK PolyU research grants G-H-ZJ80 and G-YY41.

References

[1] C. Chang, G. Czajkowski, T.V. Eicken, and C. Kesselman, "Evaluating the performance limitations of MPMD communication," Proc. ACM/IEEE Supercomputing, pp.1–10, Nov. 1997.

[2] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, MPI: the complete reference, MIT Press, Cambridge, MA, USA, 1996.

[3] J. Cao, L. Fernando, and K. Zhang, "Programming distributed systems based on graphs," in Intensional Programming I, ed. M. Orgun and E. Ashcroft, pp.83–95, World Scientific, 1996.

[4] F. Chan, J. Cao, and Y. Sun, "High-level abstractions for message-passing parallel programming," Parallel Computing, vol.29, no.11-12, pp.1589–1621, 2003.

[5] F. Chan, J. Cao, A.T. Chan, and K. Zhang, "Visual programming support for graph-oriented parallel/distributed processing," submitted to Software: Practice & Experiences, 2004.

[6] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge, MA, USA, 1994.

[7] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, MPI: The Complete Reference, vol.2, The MPI Extensions, MIT Press, 1998.

[8] J. Cotronis, "Message-passing program development by ensemble," PVM/MPI 97, pp.242–249, 1997.

[9] J. Cotronis, "Developing message-passing applications on MPICH under ensemble," PVM/MPI 98, pp.145–152, 1998.

[10] J. Cotronis, "Modular MPI components and the composition of grid applications," Proc. 10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing 2002, pp.154–161, 2002.

[11] A. Grimshaw, "An introduction to parallel object-oriented programming with mentat," Tech. Rep. 91-07, University of Virginia, July 1991.

[12] K. Chandy and C. Kesselman, "Compositional C++: Compositional parallel programming," Proc. 6th International Workshop in Languages and Compilers for Parallel Computing, pp.124–144, 1993.

[13] I. Foster and K. Chandy, "FORTRAN M: A language for modular parallel programming," J. Parallel and Distributed Computing, vol.26, no.1, pp.24–35, 1995.

[14] I. Foster, C. Kesselman, and S. Tuecke, "The nexus approach for integrating multithreading and communication," J. Parallel and Distributed Computing, vol.37, no.1, pp.70–82, 1996.

[15] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumeta, T.V. Eicken, and K. Yelick, "Parallel programming in split-c," Proc. ACM/IEEE Supercomputing, pp.262–273, 1993.

[16] K. Johnson, M. Kaashoek, and D. Wallach, "CRL: High-performance all-software distributed shared memory," Proc. 15th ACM Symposium on Operating Systems Principles (SOSP), pp.213–226, 1995.

[17] C. Chang, G. Czajkowski, and T.V. Eicken, "MRPC: A high performance RPC system for MPMD parallel computing," Software - Practice and Experience, vol.29, no.1, pp.43–66, 1999.

[18] P. Hadjidoukas, E. Polychronopoulos, and T. Papatheodorou, "Integrating MPI and nanothreads programming model," Proc. 10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing 2002, pp.309–316, 2002.

[19] C. Hu, H. Lu, A. Cox, and W. Zwaenepoel, "OpenMP for networks of SMPs," J. Parallel and Distributed Computing, vol.60, no.12, pp.1512–1530, 2000.

[20] J. Nieplocha, R. Harrison, and R. Littlefield, "Global arrays: A nonuniform memory access programming model for high-performance computers," J. Supercomputing, vol.10, no.2, pp.197–220, 1996.

[21] J. Cao, L. Fernando, and K. Zhang, "Dig: A graph-based construct for programming distributed systems," Proc. 2nd Int'l Conference on High Performance Computing, pp.417–422, New Delhi, India, 1995.

Fan Chan received the BA degree in computing from the Hong Kong Polytechnic University (PolyU) in 2001. He is currently working towards the M.Phil. degree at the Department of Computing, PolyU. His research interests include parallel and distributed computing, software visualization and high-level programming support.

Jiannong Cao received the BSc degree in computer science from Nanjing University, Nanjing, China in 1982, and the MSc and the Ph.D degrees in computer science from Washington State University, Pullman, WA, USA, in 1986 and 1990 respectively. He is currently an associate professor in the Department of Computing at Hong Kong Polytechnic University, Hung Hom, Hong Kong. He is also the director of the Internet and Mobile Computing Lab in the department. He was on the faculty of computer science at James Cook University and University of Adelaide in Australia, and City University of Hong Kong. His research interests include parallel and distributed computing, networking, mobile computing, fault tolerance, and distributed software architecture and tools. He has published over 120 technical papers in the above areas. He has served as a member of editorial boards of several international journals, a reviewer for international journals / conference proceedings, and also as an organizing / programme committee member for many international conferences. Dr. Cao is a member of the IEEE Computer Society, the IEEE Communication Society, IEEE, and ACM. He is also a member of the IEEE Technical Committee on Distributed Processing, IEEE Technical Committee on Parallel Processing, IEEE Technical Committee on Fault Tolerant Computing, and Computer Architecture Professional Committee of the China Computer Federation.

Alvin T.S. Chan is currently an assistant professor at the Hong Kong Polytechnic University. He graduated from the University of New South Wales with a Ph.D. degree in 1995 and was subsequently employed as a Research Scientist by the CSIRO, Australia. From 1997 to 1998, he was employed by the Centre for Wireless Communications, National University of Singapore as a Program Manager. Dr. Chan is one of the founding members and director of a university spin-off company, Information Access Technology Limited. He is an active consultant and has been providing consultancy services to both local and overseas companies. His research interests include mobile computing, context-aware computing and smart card applications.

Minyi Guo received his Ph.D. degree in information science from University of Tsukuba, Japan in 1998. From 1998 to 2000, Dr. Guo had been a research scientist of NEC Soft, Ltd. Japan. He is currently an associate professor at the Department of Computer Software, The University of Aizu, Japan. From 2001 to 2003, he was a visiting professor of Georgia State University, USA, Hong Kong Polytechnic University, Hong Kong. Dr. Guo has served as general chair, program committee or organizing committee chair for many international conferences. He is the editor-in-chief of the Journal of Embedded Systems. He is also in editorial board of International Journal of High Performance Computing and Networking, Journal of Embedded Computing, Journal of Parallel and Distributed Scientific and Engineering Computing, and International Journal of Computer and Applications. Dr. Guo's research interests include parallel and distributed processing, parallelizing compilers, data parallel languages, data mining, molecular computing and software engineering. He is a member of the ACM, IEEE, and IEEE Computer Society. He is listed in Marquis Who's Who in Science and Engineering.