PROCESS GROUPS AND COLLECTIVE COMMUNICATION
IN A MESSAGE DRIVEN ENVIRONMENT

BY
J. ROBERT NEELY
B.S., State University of New York College at Plattsburgh, 1992

THESIS
Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois
ABSTRACT

Efficient communication among parallel processes is an important and integral part of designing scalable parallel programs. Yet many languages do not support methods for efficiently and easily expressing communication among subsets of these processes. This thesis describes a Process Groups library for the Charm parallel programming system, which allows user programs to easily and efficiently express communication among dynamically configured groups of processes defined by the user. A detailed description of the algorithm is presented, as well as its implementation in the Charm language. Performance figures for key routines are presented, and several small application kernels are described to illustrate usage of the library.
ACKNOWLEDGEMENTS

I would like to thank my advisor Prof L.V. Kale for his guidance along the way and his ability to immediately get me back on track when I had struck out on my own in the pursuit of (eventually unworkable or misguided) solutions to what I would discover later were conceptually simple problems.

Special thanks also to all the Charm group members for their assistance in learning how to program in Charm, and help in understanding the system at a deeper level. I would especially like to thank Attila Gursoy, Amitabh Sinha, Sanjeev Krishnan, and Ed Kornkven for their expert advice when I needed it most.
# TABLE OF CONTENTS

1 INTRODUCTION .................................................. 1

2 THE CHARM ENVIRONMENT ........................................... 3
   2.1 Overview of Charm ........................................... 3
   2.2 The Programming Model: Message Driven Execution .......... 3
   2.3 Portability .................................................. 4
   2.4 The Charm Programming and Development Environment ....... 5

3 PREVIOUS WORK IN GLOBAL COMMUNICATION LIBRARIES ........ 6
   3.1 Portability Libraries ....................................... 6
   3.2 Group Communication ........................................ 6

4 THE PROCESS GROUPS (PG) LIBRARY ............................. 8
   4.1 Specifying Process Groups ................................ 9
      4.1.1 Starting the PG library and Creating an Initial Group .... 9
      4.1.2 Partitioning Into Subgroups .......................... 10
   4.2 Using Process Groups ....................................... 13
      4.2.1 Process Group Communication Functions ................. 14
      4.2.2 Process Group Informational Functions ................. 16
      4.2.3 A Simple Example Using Process Groups ............... 18
   4.3 Reduction Library Group Functions ........................ 19
      4.4 Defining and Using the Reduction Library ................ 19
      4.4.1 Creating a reduction library instance .................. 19
      4.4.2 Reduction Library Enhancements ....................... 21

5 IMPLEMENTATION OF PROCESS GROUPS ......................... 23
   5.1 Hiding Implementation Details From the User ............... 24
   5.2 Implementation of Partition ............................... 25
      5.2.1 The Partition algorithm ............................. 26
      5.2.2 Important Data Structures in Partition ............... 27
      5.2.3 Handling Out of Order Partition Messages ............. 33
   5.3 Implementation of Multicast ............................... 33
      5.3.1 The Multicast algorithm ............................. 33
      5.3.2 Important Data Structures Used in Multicast .......... 34
      5.3.3 Handling Out of Order Multicast Messages ............. 36
   5.4 Implementation of Synchronize ............................. 36
      5.4.1 The Synchronize algorithm ............................ 37
      5.4.2 Important Data Structures and Message Types in Synchronize .. 37

6 PERFORMANCE BENEFITS OF MESSAGE DRIVEN PARTITIONING .... 39
LIST OF TABLES

4.1 Available reduction libraries for use with Process Groups .................. 20
6.1 Advantages of Partitioning in a Message Driven Environment .............. 39
LIST OF FIGURES

4.1 The PG library interface file, pg.int ............................................. 9
4.2 Representation of the copy and partition parameters ....................... 12
4.3 Example of Partition() using groups of rows and columns .................. 13
4.4 Output from example program when run with 8 processors ................... 19

5.1 Advantage of global communication via spanning tree ....................... 25
5.2 The partition() algorithm ........................................................ 27
5.3 The multicast algorithm ......................................................... 35
5.4 The synchronize algorithm ...................................................... 37

7.1 Data distribution of dense matrix example ...................................... 42
7.2 Communication patterns of dense matrix multiplication example .......... 43
7.3 Outline of sorting algorithm based on histogramming ..................... 44
1 INTRODUCTION

In many applications or algorithms, there is often a natural specification of groups within the domain of the problem. This distinction becomes even more natural as a programmer begins to design an algorithm for the problem in a parallel language. Processes in many parallel programs communicate with only a determinable subset of other processes, and do so often. In sequential programs, these groups are usually implicitly defined in the data structures of the program (or instances of objects when using an object oriented programming methodology). A programmer concerned with performance will often take into account the effects of locality among these groups in large data structures. Likewise, a parallel program must be concerned with the additional overhead of efficient communication amongst these group members due to the relatively large overhead of remote memory accesses and/or message passing.

Efficient and scalable communication among these groups in a parallel program is certainly a desirable attribute of any language or architecture. This thesis introduces a program-level process groups library and associated algorithms and data structures, designed for the Charm [24] system, developed at the University of Illinois. This library has the following features:

- Ease of use at the programmer level.
- Efficient communication primitives within groups.
- Automatic overlap of group communication primitives with other useful work.
- No limit to the number of times the groups can be partitioned. i.e. defining a group within a group within a group, ad infinitum.
- No limit to the number of groups each processor may join.
- Scalability. ie, the size of distributed data structures are not a function of the total number of processors.

Correctness is guaranteed because processes are not allowed to perform group communications until it is collectively known that a complete group is formed. Yet, due to the non-blocking,
message driven nature of the underlying language, programs need not block after making a request to join a group. Instead, they are free to continue working on other tasks which are ready and scheduled, and will be notified via a message when the creation of the group is complete.

Groups are dynamically formed in a span tree structure (sometimes called a broadcast tree) for optimal communication amongst group members in $O(\log G_{\text{size}})$ time. Data structures associated with groups are completely distributed across all processors, providing an implementation which is completely scalable.

In addition to the modest (yet powerful) set of communication primitives supplied in the process group library, there is also a reduction library written for Charm which can easily cooperate with a process group, supplying a rich set of further communication primitives which can be performed at the group level.

This thesis will give a basic background on the Charm language and related tools and previous work in Global Communication Libraries will be reviewed. Then, a description of the Process Groups library functions is presented, as well as a detailed description of the implementation. The performance of the partition function is evaluated, and finally some example usages of the Process Groups library are presented.

Because of the “newness” of the library, and lack of example codes which use it, several complete Charm programs are given in the appendix which outline various usage techniques.
2 THE CHARM ENVIRONMENT

In this chapter, a brief description is given of the Charm system, upon which this work is based. In addition, an overview of previous work done with communication libraries is presented.

2.1 Overview of Charm

Charm [30] is a C based language with extensions to support development of programs for use on parallel computers. Developed at the Parallel Programming Laboratory of the University of Illinois, the Charm System consists of compilers, developments tools, and analysis tools. Several important aspects of the Charm language which will be discussed here include: message driven execution (MDE), portability across shared and distributed memory parallel machines, and a language level set of development and analysis tools.

A Charm program consists of a series of tasks called *chares*. A chare is an object based construct which contains *chare data, entry points*, and possibly *private functions*. Multiple instances of a chare can be created by the program and run in parallel, each with their own state.

The Charm language does share some similarities to other programming languages and models, such as Ada, Actors [1], monitors, concurrent objects [8][9][41], etc. Features of Charm which are novel include a rich set of information sharing abstractions [28], support for dynamic load balancing, portability without sacrificing efficiency, and branch office chares (BOC’s), a construct which allows for exactly one task, or chare, per processor. For more information, please see [24].

2.2 The Programming Model: Message Driven Execution

Like many other programming languages designed for efficient use of parallel programming resources, data is explicitly passed between instances of chares via *messages*. However, unlike “message passing languages”, there are no receive statements. Instead, data is sent to a specific chare and entry point, which upon receipt schedules that entry for execution on that processor.
This method, called *Message Driven Execution (MDE)*, has the distinct advantage over “regular” message passing that a processor will never be caught waiting on a blocking receive call. This is especially advantageous when the amount of work each processor is doing is irregular, or imbalances in computation and network speed exist. Often in these cases, it is difficult before run-time to determine the order in which events may happen, and in the standard message passing model, this can result in receive messages blocking to wait for data which has “fallen behind”. With MDE, when a message is sent, the programmer need not worry about whether there is a corresponding receive statement on the receiving end of the message. Instead, the run-time system will queue the message, and execute the entry point corresponding to the message based on built in queueing and scheduling strategies.

Programming *libraries* which involve communication are much easier to use with Charm, as well as potentially much more efficient. From the user program’s point of view, a processor need not even be aware that it is taking part in a library function. That is, the user program need not necessarily ever give explicit control to the library function, since just sending a message to the library on a given processor will guarantee that the library’s entry point will eventually be executed.

For further details on MDE programming model, please see [23][32].

### 2.3 Portability

Charm programs have the unique property that they can run without change to the source program on a variety of machines. And unlike many other message passing based libraries such as P4 [5][6], PVM [18][12][13], PICL [17], and MPI [14][16][11] [10][35], Charm programs will run efficiently on both distributed memory and shared memory machines.

In addition, development of parallel programs can be done without requiring actual time on a parallel machine. A uniprocessor version of the language has been designed which allows the user to perform all testing and debugging on a stock Unix workstation. A uniprocessor-based execution driven simulator developed for Charm takes this one step further by allowing the user to provide input parameters defining the compute power, network latencies, and run-time overhead of a particular machine, and provides an estimate of performance.
2.4 The Charm Programming and Development Environment

In addition to the Charm language, a full set of development and analysis tools have also been developed. This collection of languages, development tools, and analysis tools will collectively be referred to as the *Charm System* throughout this thesis.

In addition to the Charm language, the Charm System includes:

- **Charm++** – A C++ based version of the language [25].
- **Dagger** – A language built on top of Charm to facilitate better coordination and synchronization of messages [21].
- **Dagtool** – A visual (GUI based) tool designed for development of Dagger programs.
- **SummaryTool** – A small performance analysis tool which gives the user a “one page” analysis of the program’s performance.
- **Projections** – A detailed post-analysis tool which uses a trace of a program run to the give the user in-depth information about run-time communication patterns and behavior of the program [27].
- **Charm Libraries** – A growing set of libraries (written in Charm) which provide the user with easy (and modular) access to common complex operations.
- **Simulator/Emulator** – Briefly described in the previous section, this simulator allows the user to predict performance on a distributed memory machine by supplying parameters describing the machine’s characteristics.
3 PREVIOUS WORK IN GLOBAL COMMUNICATION

3.1 Portability Libraries

Global communication in this context refers to a point in the program where all processors partake in a common exercise. This could refer to either a broadcast type of message, such as one-to-all, or a reduction of some sort, where values spread across multiple processors must be collected and then either compared, or have some other common (usually arithmetic) operation performed on all elements.

Most vendors of parallel machines provide support in the form of message passing libraries for global communication. These libraries can often be linked directly into the user program (usually limited to C and Fortran), and often take advantage of the machine’s characteristics or special hardware. Examples include the Intel NX library [31], the CM-5 library with special support for active messages [38], or the MPL library on the IBM SP series machines [22][37]. Although they are often the most efficient implementation of a particular function, they have the disadvantage of being non-portable to other architectures.

To solve the portability problem, several popular portable message passing libraries were created which were built on top of the vendor message support, and were portable across multiple distributed memory architectures. Some of the libraries include P4 [5][6], PVM [18][12][13], PICL [17], Parmacs [7], Chameleon [20], NX [31], Zipcode [33][34], and most recently a joint effort between vendors and language developers to create a standard message passing interface, MPI [15][14][16][11][19][29].

3.2 Group Communication

For the most part, efficient global communication libraries have concentrated on making communication efficient if all of the processors on the machine (or at least those dedicated to a

\footnote{Although originally an Intel standard, at least one other vendor (Meiko) has made an NX compatibility library available on their machine}
particular partition) are involved in the operation. If a subset of processors wants to communicate, the programmer either has to revert to standard communication techniques (such as a processor performing a multicast by sending point-to-point messages in a loop), or involve all of the processors in the communication, regardless of whether a processor has a contribution to the function or needs the result.

Several of the aforementioned libraries have faced this problem by allowing the user to define process groups. A process group is a closed set of processors which can perform communication exclusively among its members. Any given processor can usually belong to multiple process groups, and a group is usually specified by a unique group identifier.

Most recently, the MPI standard [15] includes a broad range of support for process groups. At the time of this writing, the only MPI implementations available were built on top of P4, and thus are not yet subject to performance evaluation. It does appear that MPI will become a de facto standard in portable message passing libraries, however.

Yet, none of these communication libraries offer the advantages of message driven execution. In this thesis, we will attempt to show why building a group communication library to take explicit advantage of the Charm language is a better and more scalable solution to these problems.

For further information on process groups, see [2][3][36].
There are two basic stages to using the Process Groups library and its communication primitives. First, the program must specify, or define, the group structure. Upon completion of this phase, a unique *group identifier* (gid) is provided to the program which is used to identify groups during subsequent calls to the PG library.

The PG library is implemented as a branch office chare (BOC), and is thus fully distributed across all processors. Because of the message driven programming model which Charm supports, all of the work done during partition or general communication requests can be completed without the user program ever having to give explicit control to the library, except of course during the actual call to the library, and receiving the results. When operations are completed, the user program is notified via a message at a specified BOC and entry point with the result. In addition, any number of requests can be outstanding at one time, while useful computation can be easily overlapped due to the message driven execution model.

All calls in the library are non-blocking. The basic commands within the PG library are:

- **CreateRootGroup()** – Starts up the PG coordinator process, as well as supplies the user with an initial group to work with which contains all processes.

- **Partition()** – Subdivides a group into smaller groups and returns a group identifier to the user program.

- **Multicast()** – One-to-many communication of any user specified message within a group.

- **Synchronize()** – A special case of a *SendMsg* which guarantees that all processes in a group have performed the call before the message is delivered.

In order to use the PG library, the user need only include the proper interface file supplied with the Charm library distribution, and specify the charm library at link time with the `-lcharm` flag. The file `pg.int` simply provides the user with a description of the “external” references which may be made to the PG library and is shown in figure 4.1.

Using the module features of the Charm language, the user need only precede each of these calls (or message structures) with “**PG::**.”
interface module PG {
    CreateRootGroup(); /* Use this when in CharmInit, */
    CreateRootGroupMsg(); /* otherwise, use this (only call from 1 pe) */
    Partition();
    Multicast();
    Synchronize();
    PgMySpanTreeParent(); /* These functions are supplied to allow the */
    PgMyNumSpanTreeChildren(); /* user program (or other libraries) to deter- */
    PgMySpanTreeChildren(); /* mine the structure of the span tree built */
    PgGroupSize();
    PgMyRank();
    message {} CREATE_ROOT ;
    message {ChareNumType rootBocNum ;} ROOT_GROUP_CREATED ;
    message {int newGid ;} PARTITION_CREATED ;
}

Figure 4.1: The PG library interface file, pg.int

4.1 Specifying Process Groups

At the heart of the PG library is the partition() call. This call allows a processor, or more specifically a branch of a BOC, to request to join a particular partition, or subgroup, of a larger group. Every processor of the group being partitioned must join a partition before the partitioning is considered complete, thus guaranteeing that the user cannot inadvertently initiate communication with a partially formed group.

The library can support any number of partition requests, thus allowing a processor to belong to more than one group. In addition, by using reference numbers in each call, a single instance of the Process Group Coordinator can provide group functions for multiple BOC’s and modules.

4.1.1 Starting the PG library and Creating an Initial Group

Since larger groups are broken down into smaller groups, the user must be provided with an initial group to partition, as well as an identifier for the “coordinator” process\(^1\). This is done

\(^1\)The coordinator is implemented as a Branch Office Chare, and thus will return a value of type ChareNumType
via the CreateRootGroup() or CreateRootGroupMsg() functions. These calls are identical in their purpose, and only differ in the method by which a return value is provided.

CreateRootGroup() must be used only inside CharmInit(), and returns the ChareNumType of the PG coordinator immediately. CreateRootGroupMsg() is a non-blocking call which can be made at any point in the program. Only one instance of the coordinator is required, so it is important that only a single processor be designated to make this call.

The function prototypes for these calls are as follows:

ChareNumType PG::CreateRootGroup() ;

PG::CreateRootGroup(ep, boc)
    EntryPointType ep ;
    ChareNumType boc ;

In CreateRootGroup(), the ChareNumType of the coordinator is returned immediately. This call can only be used inside CharmInit(). The returned value must then be passed via a user message to the desired BOC(s).

In CreateRootGroupMsg(), ep and boc specify an entry point inside a BOC at which a message of type PG::ROOT_GROUP_CREATED will be delivered containing only the ChareNumType of the coordinator handling the PG functions in a variable called rootGroupBoc.

After this call has returned (either immediately or via message), the user can now assume that an initial group containing all the branches (or processors) has been created, which has an initial gid value of zero (0). NOTE: Only a single branch should make this call (usually, but not necessarily processor zero), lest P copies of the coordinator be created, potentially incurring errors in the program. Multiple coordinators can be created, but with proper use of reference numbers, this is not necessary.

4.1.2 Partitioning Into Subgroups

Once the CreateRootGroup call has returned, the user can then define and create groups via the Partition call. The function prototype for this call is as follows:
CoordBoc is the value which was returned via the CreateRootGroup() function. gid is the group identifier of the group being partitioned. Initially, this value will be zero, the gid of the original root group. When further subgroups are created, gid will contain the system generated group identifier of that group. (An example of this will be shown in § 4.2.3).

Copy and partition are user specified identifiers which together form a unique differentiation of which subgroup the process wants to join. If only one type of group were allowed, the copy identifier would not be necessary. However, the user will certainly want to partition processes into more than one type – for example, groups of rows, and groups of columns in a two dimensional domain. The partition parameter specifies which group in the specified copy that processor wishes to join.

Ep and boc specify the branch and entry point at which the process will be notified once the group has been created. A message of type PG::PARTITION CREATED will arrive at that entry point with a single field newGid representing the group identifier of the newly created group. All processes in the group being partitioned must make a partition request with the same copy identifier before the group is considered complete. In addition, a reference number can be supplied with the ref parameter. This reference number will be attached to the PARTITION CREATED message, and can be retrieved with the GetRefNumber() call supplied with the Charm language. A reference number is useful when you wish to have a single entry point designated to receive two or more PARTITION CREATED messages in which the next step depends on which group the message is pertaining to. It is important to remember that the gid returned, although it is an integer value, tells nothing about which group the member belongs to, other than the fact that members of the same group will obviously share the same group identifier.

Figure 4.2 shows a graphical representation of how the group structure would look after each processor in a program containing 6 processors had made two calls to partition. For copy
Figure 4.2: Representation of the copy and partition parameters

1, the processors numbered \(< P/2\) join partition 1, and the remaining processors numbered \(\geq P/2\) join partition 2. For copy 2, the odd numbered processors join partition 1, and the even processors join partition 2. Note that partition numbers need not be unique across different copy numbers.

Because of the message driven programming model which Charm supports, is it thus advantageous to perform the partition call as early as possible, such that other useful work (not dependent upon the group being partitioned) can be automatically overlapped by the requesting processor while the group is being completed.

4.1.2.1 Example of partition()

For example, suppose the user wishes to partition the root group (containing all processors) into groups of rows and columns. An example of the code to do this is shown in Figure 4.3.

Since two types of partitions are being performed (rows and columns), there are two distinct copy numbers used. Each processor is also assumed to know what row and column it is representing (denoted by myRow and myCol in the example), which is then used as a unique partition parameter.
This example is similar to the dense matrix multiply example which is fully described in § 7.1, and shown in its completion in Appendix A.3.

4.2 Using Process Groups

Once groups have been created, and their group identifiers have been returned to the user, the user program can profit from the advantages of group functions. In § 4.2.1, two common group functions are described which are provided in the PG library. In § 4.3, we describe how the existing reduction library in Charm was expanded and incorporated into the PG library.
4.2.1 Process Group Communication Functions

In addition to the functions outlined in earlier sections which dealt primarily with the creation of groups, the PG library contains two additional functions which allow the user to take advantage of optimized communication within these groups:

- **Multicast()** – One-to-many communication of any user specified message within a group.
- **Synchronize()** – A special case of a SendMsg which guarantees that all processes in a group have performed the call before the message is delivered.

**One-to-many Communication**  
Multicast() is a general form of the BroadcastMsgBranch() function supplied with the Charm system. It allows efficient one-to-many communication within any user-created group via a spanning tree (sometimes called a broadcast tree). The function prototype for this call is as follows:

```c
PG::Multicast(coordBoc, gid, msg, ep, boc)
    ChareNumType coordBoc ;
    int gid ;
    ANY_MSG_TYPE *msg ;
    EntryPointType ep ;
    ChareNumType boc ;
```

CoordBoc is the value which was returned via the CreateRootGroup() function. gid is the group identifier of the group receiving the message. **NOTE: Currently, a branch is only allowed to multicast to a group of which it is a member.** Of course, a branch can send a message to a member M outside of its group with the intention of multicasting it to the group of which M is a member. Msg is any user defined type, and ep and boc specify the entry point and BOC type which will receive the message.

Efficient communication is provided via a spanning tree built across the members of the group during the partition phase. Details of this structure will be described in chapter 5.

---

2This was an implementation trade off which saves potentially substantial communication by not involving processors outside the group being partitioned in the partition phase.
Efficient Barrier Synchronization Within a Group  

`Synchronize()` is a special form of a barrier synchronization. It is designed to allow the user to perform a synchronization of all processors in a group, after which the user specified message will be delivered to the specified chare id and entry point. Note that unlike the `Multicast()` function, the destination of the message can be either a BOC or a regular chare. This requires the user to supply the `ChareIDType` of the destination rather than the BOC number. This is useful for example when the user wishes to synchronize a program after which the messages are delivered to a single chare (for example, the `main` chare.)

The function prototype for this call is as follows:

```c
PG::Synchronize(coordBoc, gid, refnum, msg, ep, id)
   CoordNumType coordBoc ;
   int gid ;
   int refnum ;
   ANY_MSG_TYPE *msg ;
   EntryPointType ep ;
   ChareIDType id ;
```

Again, `coordBoc` is the value which was returned via the `CreateRootGroup()` function, and `gid` is the group identifier for the group being synchronized. `refnum` is a user specified type which identifies which synchronization this particular call is taking place in. Since the call is non-blocking, it is possible that each processor can have several outstanding `Synchronize()` calls at any given time. This reference number allows the PG library to differentiate among them. `ep` and `id` specify the entry point and chare which will receive the message.

Note that unlike traditional barrier operations, useful work can still be performed since the synchronization does not block the program waiting for all members to reach the barrier. Instead, the message to be delivered is buffered, and only when all branches of the group have reached the barrier is the message retrieved and sent to the destination. If other entries have previously been scheduled for execution at the time of the synchronization (and are thus not required to execute after the barrier), they will execute while the other members of the group are still arriving at their respective `synchronize()` call for the same reference number. This is especially useful and efficient when the amount of work done across processors varies, and thus all processors may not reach the barrier within a small time frame.
A common usage of a synchronize or barrier point might be between phases of an algorithm, where all members must complete the first phase before the subsequent phase is undergone. Charm has several ways of doing this without the `synchronize()` call.

Possibly the most common way of doing this is via *quiescence detection*, in which the system waits for all outstanding messages to be completely serviced before a user-specified entry point is called. Conceptually this is a nice way to divide phases of a program, however it is undesirable for two main reasons:

1. Quiescence detection is an expensive algorithm, and it takes a relatively long amount of time after messages have “drained” from the system before all processors are notified.
2. Quiescence can only be detected across all processors, not a subset or group of processors.

Another possible solution is by using the `barrier()` call from the reduction library supplied with Charm. `barrier()` is implemented as a special case reduction in which no actual values are reduced and no value is returned. Again conceptually this is a nice solution, but the fact that no useful data is contained in the call requires the user to create an entry point specifically for receiving the barrier completion notification, at which point the message for the next “phase” must be built and then sent.

### 4.2.2 Process Group Informational Functions

In addition to `Multicast()` and `Synchronize()`, there are several functions provided which immediately return information about the structure of an already created group. These functions were provided as “hooks” into the Process Groups library so that other libraries (such as the reduction library, described in § 4.3) which need more detailed information about the structure of the group can be built on top of the process group library. Details about the spanning tree structure used by the library are fully described in chapter 5.

For all of the functions described in the rest of this section, the group must be fully created (the user has both the `ChareNumType` of the PG coordinator as well as the `gid`), and the calling processor *must* be a member of the group defined by `gid`. For example, a processor cannot make a `PgGroupSize()` call to find out the size of a group of which it is not a member. This limitation is due to the implementation of the library, and was a trade off to gain additional
efficiency by not involving processors outside a group being defined during the partition phase, thus incurring a potentially high overhead.

The function prototypes for these “informational functions” are shown below, along with a description of what each function returns.

```c
int PgMySpanTreeParent(coordBoc, groupID)
    ChareNumType coordBoc ;
    int groupID ;

PgMySpanTreeParent() returns the processor number (as would be given by McMyPeNum()) of the spanning tree parent of the calling processor. If -1 is returned, then the calling processor is the root of this group.

int PgMyNumSpanTreeChildren(coordBoc, groupID)
    ChareNumType coordBoc ;
    int groupID ;

PgMyNumSpanTreeChildren() returns the number of processors which the calling processor has as immediate children in the span tree. If 0 is returned, then the processor is a leaf node in the tree. If 0 < n <= MAX_SPAN_CHILDREN is returned (MAX_SPAN_CHILDREN is defined at compile time of the library, and is 2 by default), then the processor is an intermediate node in the tree.

This is not the same as the number of descendants under a node. ie, each processor knows only about the children directly connected to it, not anything about grandchildren, etc.

void PgMySpanTreeChildren(coordBoc, groupID, children)
    ChareNumType coordBoc ;
    int groupID ;
    int *children ;

PgMySpanTreeChildren() fills an array with the processor numbers of the calling processors’ children in the span tree. It is up to the user to be sure that the children array being passed in is at least as large as n, where n is the value returned by PgMyNumSpanTreeChildren(). And of course, only the first n entries in the resultant array are valid.
int PgGroupSize(coordBoc, groupID)
    CharNumType coordBoc ;
    int groupID ;

PgGroupSize() returns the size of the group specified by groupID. As noted above, this function can only be called from a processor which is a known member of the group specified by groupID.

int PgMyRank(coordBoc, groupID)
    CharNumType coordBoc ;
    int groupID ;

PgMyRank() returns a unique identifier for each processor in a group in the range 0…G, where G is the value returned by the function PgGroupSize(). This number corresponds to the order in which that processor was added to the group, although this information is typically not of interest to the user program.

4.2.3 A Simple Example Using Process Groups

A simple example is described in this section of how the previously defined functions all work together. This complete Charm program, shown in Appendix A.1 partitions the processors into high and low numbered groups (based on processor numberings from 0…N − 1). Each of these groups is then partitioned into subgroups containing odd and even processor numbers, for a total of four distinct groups.

After the groups are completely formed, each processor then builds a message based on what it knows about itself, and multicasts it within its group at which point a short message is written to verify that only group members are communicating with each other.

Output from this program when run with 8 processors is shown in figure 4.4.

As we can see from the example output (which has been sorted for clarity – a real parallel program would produce answers in a more nondeterministic ordering), there are exactly 4 groups formed, and communication occurs only between group members, where each group is of size 2. This program will work for any sized group, as the sizes of groups being formed need not necessarily all be of the same size. For example, if this program were run with 10 processors instead of 8, the “lower and even” group would have 3 members (as processor 4 would now be considered in the lower group), but the “lower and odd” group would remain as size 2.
4.3 Reduction Library Group Functions

In addition to the basic set of group functions provided with the PG library, the Charm system reduction library has been extended and improved to work with process groups as a part of this thesis research. Table 4.1 shows the global operations available to the user program, via the reduction library.

4.4 Defining and Using the Reduction Library

The reduction library available with versions of Charm prior to version 4.1 were usable only across all processors (or more specifically, all branches of a BOC). Since reduction-style operations are often desired across defined process groups as well, modifications were made to the existing reduction library to support this, as well as several improvements to the overall usability and efficiency of the library.

4.4.1 Creating a reduction library instance

The reduction library works by first creating an instance of the reduction library, which is actually implemented as a BOC. An instance of a reduction library must be formed for each processor.
<table>
<thead>
<tr>
<th><strong>Data type and function</strong></th>
<th><strong>InterfaceFile</strong></th>
<th><strong>ModuleName</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>integer maximum</td>
<td>imaxredn.int</td>
<td>IMaxRedn</td>
</tr>
<tr>
<td>integer minimum</td>
<td>iminredn.int</td>
<td>IMinRedn</td>
</tr>
<tr>
<td>integer sum</td>
<td>isumredn.int</td>
<td>ISumRedn</td>
</tr>
<tr>
<td>integer product</td>
<td>iprodredn.int</td>
<td>IProdRedn</td>
</tr>
<tr>
<td>integer count</td>
<td>icountredn.int</td>
<td>ICountRedn</td>
</tr>
<tr>
<td>float maximum</td>
<td>fmaxredn.int</td>
<td>FMaxRedn</td>
</tr>
<tr>
<td>float minimum</td>
<td>fminredn.int</td>
<td>FMinRedn</td>
</tr>
<tr>
<td>float sum</td>
<td>fsumredn.int</td>
<td>FSumRedn</td>
</tr>
<tr>
<td>float product</td>
<td>fprodredn.int</td>
<td>FProdRedn</td>
</tr>
<tr>
<td>double maximum</td>
<td>dmaxredn.int</td>
<td>DMaxRedn</td>
</tr>
<tr>
<td>double minimum</td>
<td>dminredn.int</td>
<td>DMinRedn</td>
</tr>
<tr>
<td>double sum</td>
<td>dsumredn.int</td>
<td>DSumRedn</td>
</tr>
<tr>
<td>double product</td>
<td>dpodredn.int</td>
<td>DProdRedn</td>
</tr>
</tbody>
</table>

*Table 4.1: Available reduction libraries for use with Process Groups*

- Data type and reduction type (ie, integer/sum, float/max).
- The group over which it will perform the reductions.

To create a reduction instance over all processors, the `Create()` call is used (from inside `CharmInit()` with no parameters. To create a reduction instance over a previously defined group, the `CreateOverGroup()` command is used, with four parameters: the *group identifier* of the process group, the `ChareNumType` of the Process Group coordinator (returned by the `PG::CreateRootGroup` call), and the return entry point and `ChareIDType` of the user program to which the new BOC identifier will be returned. Only one processor should make the call to `CreateOverGroup` (for example, the processor whose rank equals 0 in the new group), but the return BOC identifier will be automatically multicast to all members of the group.

The function prototypes for these calls are shown below.
ChareNumType Create()

ChareNumType CreateOverGroup(gid, pgBoc, ep, id)
    int gid;
    ChareNumType pgBoc;
   EntryPointType ep;
   ChareIDType *id;

Once the instance has been created, the user makes calls to the reduction library via DepositData or DepositDataMsg calls (shown below). It is up to the user to guarantee that only valid group members use a particular instance of the reduction library.

DepositData(boc,x,z,nel,refnum,fptr,id)
    ChareNumType boc;
    DataType x[],z[];
    int nel;
    int refnum;
    void (*fptr)();
    void *id;

DepositDataMsg(rednboc, x, nel, refnum, ep, id)
    ChareNumType rednboc;
    DataType x[];
    int nel;
    int refnum;
    EntryPointType ep;
    void *id;

Details on use of the reduction library are not relevant to this thesis, and the reader is referred to the Charm Language Reference Manual [30].

4.4.2 Reduction Library Enhancements

In previous Charm reduction library implementations, not only was a reduction library instance constrained to the type of reduction it performed and the members it served, but the size of the reduction (for use with concurrent reductions) was determined at instance creation time. Also,
the same instance of a reduction library could not allow overlapping reductions to take place. That is, before the instance could be used to perform another reduction, the answer from the previous reduction had to be returned.

Neither of these constraints took away from the expressiveness of the language though, since multiple instances could be created for either different sized reductions, or for overlapping reductions. However, the small cost of overhead incurred by having possibly many instances of the same type reduction could play a factor in program performance, as well as unnecessarily require the user to keep track of multiple instances of libraries, opening the avenue for possible mistakes in reduction specification.

In order to improve the usability and efficiency of the reduction library, the size parameter was moved from the Create() call into the actual DepositData{Msg}() call. Thus, using the same instance of a reduction, the user could perform a reduction over a single operator, or a concurrent reduction over many values in a one-dimensional vector.

In order to allow overlapping reductions with the same instance, a refnum parameter was also added to the DepositData{Msg}() calls. This is a user defined integer value which specifies which reduction that particular call was meant for. Hence, overlapping calls to the reduction instance can be performed without the user having to worry about whether another reduction is currently executing in the same reduction library instance.
5 IMPLEMENTATION OF PROCESS GROUPS

The PG library is implemented such that the relationship between group members provides for efficient and scalable communication via a spanning tree (also sometimes called a broadcast tree). The span tree is built dynamically during the partition phase, with newly joining members being added to the leaves of the tree. The root of this spanning tree is also assigned to be the root group member of that particular group. By default, processor zero is the root member of the original group containing all processors.

Communication via a spanning tree provides two main advantages:

1. The length of the critical path of communication is bound by $\log N$, where $N$ is the number of group members.

2. Messages being sent are spread evenly over the network, preventing network traffic from building up near the sender.

As an example, let us suppose the user wishes to implement one-to-many communication within a group. Using communication primitives built into the Charm language, a portion of code to do this may look something like the following:

```charm
for(i=0 ; i<groupSize ; i++) {
    msg = (USER_MSG *)CkAllocMsg(USER_MSG) ;
    msg->field1 = someValue ;
    /* The rest of the message field is filled in */
    penum = groupList[i] ;
    SendMsgBranch(ep, msg, penum, boc) ;
}
```

Assuming the partition phase is completed, the equivalent effect can be achieved with the PG library call:

`PG::Multicast(coordBoc, gid, msg, ep, boc) ;`

---

1 Reallocating and filling the message each time could be replaced by a CkCopyMsg() call, a new system call as of v4.1

23
In the first example, \textit{groupSize} messages are sent from the processor initiating the communication, incurring \textit{groupSize} message startup latencies. On most distributed memory systems, startup latencies are a dominating factor in communication costs when message sizes are small to moderate sized. Users may write their own span tree based multicast, but it must then be written for each application independently.

Using the \textit{Multicast} library function, the message is sent to the processor assigned as the root of the spanning tree, which in turn sends the message to its children, who in turn forward the message to their children, until the leaf nodes of the span tree are reached. The number of children of each node in the span tree is usually two, but can be adjusted for optimal performance on various architectures.

As the number of members in the group increases, the \textit{Multicast} method will prove to outperform the original, as the length of the critical path of communication is bound by $\log_x N$, where $x$ is the number of children each node of the span tree has, and $N$ is the number of members in the group. In figure 5.1, we show a diagram of how a span tree optimizes one-to-many communication. Time is shown along the $x$ axis, with an arrow pointing from the source to destination processor signifying a message being sent (and received) at that time. Notice that in the span tree method, as soon as the root processor delivers the message to its children, communication begins to happen in parallel, with communication spread out evenly across the network.

### 5.1 Hiding Implementation Details From the User

All of the calls to the PG library described in this thesis are in the form of regular functions and procedures with parameters, not message types. These functions are implemented in the PG module, and perform some of the more “gory details” which the user need not see.

Each of these functions in the PG module actually calls a public function inside the coordinator BOC via the \textit{BranchCall} statement. Thus, the first level of abstraction is to hide the verbose \textit{BranchCall} statement. These functions then call public functions inside the coordinator which initiate the operations inside the PG library, and build messages automatically out of a combination of user supplied parameters and system generated variables. The public functions initiated by the \textit{BranchCalls} are hidden from the user due to the modular features of Charm,
which requires a function name to be “exported” in the interface file for it to be available for
outside use.

All of the names and data structures which the user needs to know about are described in
the \texttt{pg.int} interface file. This file is shown in figure 4.1, and has purposefully been kept as
simple as possible for ease of use.

5.2 Implementation of Partition

In this section we go into detail on the implementation of the partition function of the PG
library. In § 5.2.1, we describe the basic algorithm used during partition to correctly form the
groups. In § 5.2.2, we describe the main data structures used during the partition phase.
5.2.1 The Partition algorithm

During the partition phase, the span tree of each group is dynamically built as new members join. The algorithm outline is as follows, and each step will be presented in more detail below:

1. The partition request is sent to the root group member.

2. The root determines if a group has been started for the copy and partition of the parent group specified.
   - If so, then the request is forwarded to the root group member for the newly forming group.
   - Otherwise, the requesting processor is designated as the new root group member for the forming group.

3. When a processor receives a forwarded request, it first determines if it already has all of its children (usually two, but compile-time definable) in the spanning tree being formed.
   - If so, then it forwards the request down the spanning tree to one of its children (based on which side of the tree will hold the new leaf node).
   - Otherwise, it makes the requesting processor one of its children

4. The requesting processor will eventually receive a message from the processor which will become its parent in the spanning tree, and becomes a new leaf. As the tree keeps growing, the node may eventually become an internal node of the spanning tree.

5. The root group member of the group being partitioned keeps track of how many requests have been made for each copy. When it determines that all the members of the group being partitioned have requested to join some partition, it sends a done message down the spanning tree of each newly formed subgroup.

6. When each processor receives the done message, it sends a message containing the new group identifier to the local branch at the entry point and BOC which the user specified in the partition call. At this point the partition phase for this copy is considered completed.
Figure 5.2: The partition() algorithm

This algorithm is shown in Figure 5.2.

Note that due to the message driven execution style of Charm, and the non blocking nature of the partition call, other work will be automatically overlapped with the partition phase as it becomes available. It is then advantageous to perform the partition phase early on in the computation (relative to where other global group communication primitives will be used) so as to allow for as much possible computational overlap as possible.

5.2.2 Important Data Structures in Partition

This section outlines the major data structures used during the partition phase of the library. These data structures are contained in the file pglib.h.

5.2.2.1 Maintaining Unique Group Identifiers

One of the initial problems to overcome when implementing the library was the need to assure that group identifiers assigned by the library were unique across all processors. The solution to
this was to allow each processor which was assigned as new group root node to only be able to
assign numbers which are a multiple of that processor’s ID number (PE num).

Each processor has a variable called $\text{NextAvailableGid}$ which is initialized to that processors
PE number – except for processor zero which initializes it to $P$ – where $P$ is the total number of
processors. This is because gid 0 is assumed to be the initial root group containing all processors.

Whenever a processor becomes the root for a new group, it uses the $\text{NextAvailableGid}$
variable, and then increments this value by $P$. This assures that no two processors will ever
attempt to use the same gid, and that no processor will assign the same gid twice.

The one problem with this solution is the possibility that a gid value can potentially exceed
the maximum value a standard 32-bit integer can hold. This is highly unlikely to happen
however. Assuming we have $P$ processors, in the ideal case if $P$ distinct groups were formed,
each processor would be the root for only 1 of those groups. Even in the undesirable case that
every group formed assigns processor $p$ as the root, $2^{32}/P$ groups would need to be formed
before the gid values began to “roll over”. In the future when 64-bit integers are standard, the
likelihood of this happening becomes infinitely small.

**Debugging Programs Which Use Process Groups** The group identifier is an integer
type, but the actual value will be meaningful only to the PG library. The user typically need
not know the value of the gid, except in the case of the original group (described in § 4.1)
when the value must be specified as zero (0). However, by printing out the gid along with
other identifying information (namely the processor number), the programmer can debug their
program to determine which processors belong to the same group.

**5.2.2.2 Group Lists**

The process group library is designed to be *scalable*. Large numbers of groups can be formed
without data structures running out of space, or seeing a significant loss of efficiency. To achieve
this, all data structures related to the formation of process groups are created dynamically.
Structures which define the group structure, once created, remain intact throughout the run
of the program. Other data structures are temporary (such as the “multicast control” and
“synchronize control” messages, see § 5.3.2.1 and § 5.4.2) and are deleted once they are done
with.
Each processor maintains a group list: a list of all the groups which that processor is a member of. The information stored for each group includes:

- The value of gid.
- The root processor of this group.
- The span tree structure, i.e. the parent and children of the node for this group.
- The number of descendants in the span tree.
- The total group size thus far.

In addition, if that node is designated as the root member of that list, then it contains additional information about the partitions which it is the root of. This will be described in further detail later in this section.

Note that the data storage requirements are not dependent upon $P$, where $P$ is the number of processors. Storage is dependent only upon the number of groups each processor has joined. This becomes important as the number of processors grows large. Other group libraries require storage of $O(P)$.

**The Hash Function** The corresponding data structures which make up the list are stored in a dynamically created linked list, built on top of a small hash table to keep the lists from getting too large. The hashing function was designed to be simple and fast. It works by using an array of pointers to group identifiers which all hash to that table entry. The hash function is the simple $\mod$ operation, and computes the hash table entry by taking $\mod(gid, TABLE\_SIZE)$. Since group identifiers are given out in an orderly manner based on the processor number (see § 5.2.2.1), the TABLE\_SIZE should be a prime number. As a backup, the library checks to see if the current number of processors is equal to TABLE\_SIZE, and if so chooses another value (the next largest prime value) as the table size. This helps to prevent collisions in the hash table due to gids assigned by the same processor from hashing to the same position.

Once the gid is hashed to a table entry, the linked list is traversed linearly until a matching gid is found in the list. If no gid is found, then there is an error in the user program in which they are trying to use a gid value which was not returned to that branch.
**Root Group Lists**  For every group created, there is a root group member assigned to it. By default, the initial root group created has processor zero as the root group member. Thus, any time gid 0 is partitioned, the request will go through processor 0. The first processor \( p \) to request joining a particular copy/partition will automatically be designated as the root group member of the newly forming group \( g \), after which all requests to further partition \( g \) will go through processor \( p \). These root group members must thus maintain information about all of the groups which have been partitioned under it.

This data structure is maintained by keeping a “list of lists”. The first list is a “copy” list, and contains a structure for each unique copy number which is partitioning the group. This structure’s main function is to keep track of how many processors have joined the copy (via any number of partitions less than or equal to the group size), and notify the group after all processors have joined, thus completing the partition.

Each copy structure also keeps a list of partitions for that copy. For each partition structure, it maintains the number of processors which have joined that partition, the gid assigned to the newly forming group, and the root group member of the newly forming group.

**Differences Between the “Group List” and the “Root Group List”**  It is important at this point to clearly distinguish between the two types of lists being maintained here.

The group list is a list of groups for which this processor is a member. These group entries are stored in linked lists, with each list’s head accessed via a hash table. They contain important information about the structure of a group; namely the root processor of the group (for initiating group communication), and the parent and children in the span tree structure.

The root group list is actually directly attached to the group list. One, and only one, processor in each group formed will be designated as the root group member. Any time a partition request for that group is made, the root group member orchestrates the layout of the newly forming subgroups. For each copy/partition pair, the root group member must be able to store information about the root member of the newly forming group. In addition, a count is kept to keep track of how many members have joined a given copy. Only when all the members of the group being partitioned have made a partition request for a given copy, will the group be considered complete and the done message initiated.
5.2.2.3 Messages Used During the Partition Phase

The messages used internally by the PG library are defined in the file `pglib.h`. In this section, we will describe most of the messages, and their corresponding function.

The message type `GENERIC_MESSAGE` is used whenever the message type is not known, and is roughly equivalent to using a void pointer in C. For example, the user can pass any type of message into the library via the `multicast()` and `synchronize()` functions, thus requiring a generic message handler type.

Creating the Root Group  The user can start the process group library at any time during the program’s execution. If the request is made from within CharmInit, then the PG coordinator BOC is immediately created and the BOC number is returned.

If the request is made from outside CharmInit, however, the user must supply a BOC number and entry point to return the message to. This request should only be made from one processor however, lest multiple copies of the PG coordinator be created, which is usually not desirable or necessary. However, in order to make sure that the new coordinator BOC number is distributed to all of the branches of the BOC (without the user having to explicitly broadcast it after the calling processor receives it), a temporary chare is created which “intercepts” the BOC number after it has been created, and broadcasts its value to all of the branches of the user program.

The `ROOT_GROUP.Created` message type is used to return the new BOC value to the user program. It contains a single field called `newGid` which holds this value.

Creating a Partition  When a partition request is made by the user program, the root group member of the group being partitioned is located, and a message of type `PARTITION_AT_ROOT` is sent. This message contains all of the information the library will need to form the group, including the copy and partition numbers, the processor which is making the request, and the return BOC and EP to send the new gid to.

When the root group member receives the partition request, it decides where to forward the message. If this is the first processor to join this particular partition, then that requesting processor is designated as the root group member of the newly forming group. If somebody
has already joined this partition, then the message is forwarded to the already designated root group member for further handling.

The message type used during this phase is NEW_MEMBER_MSG, and contains information used by the library such as a count of the number of descendants each node has, the new group identifier (which will be different from the group identifier which specified the group being partitioned, as well as the BOC and EP to return the new gid to once the group is completely formed.

At any given node in the tree, the node will know if it has all of its span tree children yet. If it does not, then the current requesting processor will become one of those children. This is handled via the library entry point NewMessageLandAndPin, which signifies that the message will stop its descent through the span tree at that point and be “pinned” in place. If both children are present at a node, then the message is forwarded to one of the children via the entry point NewMemberFwd. This descent of the message through the span tree goes on until the message finally “lands” in an available spot as a leaf in the span tree. Each node in the tree keeps track of only its parent and children, thus giving a completely distributed description of the tree structure.

Deciding which direction to send the message when forwarding it is done via a small algorithm which makes sure that the tree leaves are filled in from “left to right” as the tree is built. Thus, instead of alternating the forwarding message from one child to the other, a node will send a message to one child until that next level of the tree is filled up, at which point it begins to forward messages to the other child. This algorithm was used to match the structure of the default span tree built over all processors at the time the coordinator is created.

Marking a Partition as Done The root group member of a group being partitioned is in charge of determining when all the members of the group have joined a certain copy. It does this by keeping a running total of the number of processors which have already joined a given copy, and comparing it to the number of members in that group, sometimes referred to as the number of descendants. When these two numbers match, the root group considers the new group completed, and initiates a DONE_MSG message to each of the root group members of the new groups. (Recall that multiple groups, or partitions, are formed for each copy).
The DONE_MSG is used to notify all of the groups that they may now notify their respective processors (branches) of the group’s completion. When the DONE_MSG reaches a group, the gid is looked up in the group list, and the BOC and EP which the user requested the new gid to be returned to is looked up. At this point, a message of type PARTITION.CREATED is sent to the user program.

5.2.3 Handling Out of Order Partition Messages

It is possible that due to either random delays, retransmissions, or stack based queueing strategies, that the DONE_MSG could actually arrive before the NEW_MEMBER_MSG has arrived. In this case, there will not yet be an entry in the group list for the gid in question.

In this case, the group structure is added to the group list, awaiting the information contained in the NEW_MEMBER_MSG, and the DONE_MSG is halted at that node and buffered, since none of the children of that node could have received their NEW_MEMBER_MSG message yet either. The DONE_MSG is buffered in a special list dedicated to storing these messages.

When the NEW_MEMBER_MSG arrives, the system recognizes that it has already received the DONE_MSG, forwards the NEW_MEMBER_MSG to its children (if applicable), and starts the DONE_MSG down the span tree again.

5.3 Implementation of Multicast

In this section we go into detail on the implementation of the multicast function of the PG library. In § 5.3.1, we describe the basic algorithm used to multicast within a group. In § 5.3.2, we describe the main data structures used during the multicast phase.

5.3.1 The Multicast algorithm

Once the partition phase is completed, the distributed structures are in place for efficient communication across all members of the group. One-to-many communication is performed by simply sending the message down the spanning tree, with each processor delivering the message to the specified entry point and BOC on the local processor, and forwarding the message down to its children.
Each message in Charm contains an *envelope* which is used by the system to store the final destination of the message, among other things. Because the actual user message is passed through the PG library functions before it is actually delivered, the original envelope containing the desired destination entry point and chare identifier would be overwritten by delivery information for the PG library. Thus, some method must be incorporated to tell the message where to go once it has reached the destination processor.

To solve this, a *control message* is sent ahead of the original user message through the span tree with a system generated reference number which will notify each processor where the message should be sent once it is ready to be sent across the library boundary and back to the user program. The reference number field of the envelope is used by the library to keep track of incoming messages of unknown type, but is restored to its original value before being forwarded to the user program (in case the user had set the reference number in the original message).

When the control message arrives at a node, the information it holds is stored in a hash table based on the reference number. When the actual multicasted message then arrives at the node, its reference number is extracted, and the delivery information is looked up in the hash table. At this point, the original tag (possibly supplied via the `SetRefNumber()` function by the user) is restored to the message, the message envelope is supplied with the destination BOC and EP, and the message is delivered on that processor.

It is possible that the actual multicast message will arrive at any given node in the span tree prior to the control message. In this case, the message is buffered and its traversal down the span tree halted until the control message “catches up”. At this point, since both the message and destination information are known, the message can be delivered to that node, and both the control and user messages are forwarded down the tree.

The algorithm for multicast is shown in Figure 5.3

### 5.3.2 Important Data Structures Used in Multicast

This section outlines the major data structures used during the multicast operation of the Process Groups library. These data structures are contained in the file `pglib.h`. 
5.3.2.1 Control Messages and Data Structures

As explained in § 5.3.1, a separate control message is sent ahead of each user message being multicasted which contains the information needed to deliver that message once it has arrived at the destination processor in the group.

Before a control message is sent, it is assigned a unique reference number by the system which will allow the library to later match the control data with the incoming user message. Uniqueness among reference numbers is kept by allowing each processor (or more specifically, each root group member) to only assign a non-decreasing set of numbers which are unique to that processor. The same technique is used for assigning group identifiers, and is outlined in detail in § 5.2.2.1.

A message of type MULTICAST_CONTROL is then sent to the root group member of the group via the MulticastAtRoot entry point, and contains the group identifier, the reference number, the original reference number of the user message, and the BOC and EP to which the corresponding message is to be delivered. The message is then forwarded down through the span tree to all the members of the group via the MulticastControl entry point. Upon arriving at a processor, the information in the MULTICAST_CONTROL message is stored in a control
structure in a linear linked list. This temporary structure will store the delivery information required when the user message arrives, and is unique by its reference number.

After the control message is sent, the user message is then sent to the root group member as a GENERIC_MESSAGE type – analogous to the void pointer used in C – to the MulticastDeliver entry point. When the user message arrives at a processor, the reference number which is stored in the envelope of the user message is retrieved, and the corresponding control structure is looked up in the control list. The user message is then redirected to the BOC and EP specified by the multicast call and stored in the control structure, and the message is delivered. At this point, the control structure for that particular reference number is no longer needed, and is deleted from the control list.

5.3.3 Handling Out of Order Multicast Messages

As with the partition phase, it is possible that the user message may arrive at a node prior to the control message. In this case, the user message does not yet have the delivery information needed to complete the multicast on that processor.

At this point, the control structure is created, just like it would be upon arrival of the control message, but the delivery information is left blank until the control message arrives. The user message is buffered in a special list dedicated to these buffered messages, and is halted at that node. When the control message “catches up”, the buffered message is retrieved, delivered to that processor, and the pair of messages are both restarted down the span tree in the “correct” order.

5.4 Implementation of Synchronize

In this section describe the synchronize algorithm, which effectively allows the user to barrier synchronize within a group, without fully stopping useful work from progressing. In § 5.4.1, we describe the basic algorithm used to efficiently synchronize within a group. In § 5.4.2, we describe the main data structures used during synchronization.
5.4.1 The Synchronize algorithm

Synchronize also works by using the spanning trees built up via the partition phase of a process group definition. The user specifies a message and destination just as in a `SendMsg()`, but they also specify a group identifier (gid), and a “key” used to uniquely reference overlapping synchronizations going on.

Upon calling `synchronize`, the user’s message is buffered on that processor for later delivery, and a special message is sent up the span tree to the calling node’s parent. When the root group member receives the synchronize request from all of its children (who in turn must have received the request from all of their children, etc.), a special multicast is sent to the group members via the PG library. When this message is received, the buffered messages are then sent to the intended destination.

The algorithm for `synchronize()` is shown in figure 5.4

5.4.2 Important Data Structures and Message Types in Synchronize

Much like the multicast algorithm, a series of “control messages” and data structures are used to orchestrate the synchronize function. Unlike the partition and multicast algorithms, how-
ever, the user must supply the reference number which is used to match corresponding control messages and buffered user messages. This is because multiple processors in a group are participating in the event, and guaranteeing that the system properly handles assigning reference numbers unique to a particular synchronize call becomes difficult and expensive, if not impossible.

Also unlike the multicast algorithm, the user message is immediately buffered on the calling processor, awaiting a signal that it can be “released” for delivery to the specified chare and entry point. NOTE: Unlike the partition and multicast calls, the delivery point is specified by a Chare Identifier and entry point, instead of a BOC and entry point. This is because the ChareIDType is more general than the BOC number, and due to the nature of the function, it is quite possible that the user may wish to have the message delivered to a regular chare, instead of a BOC.

Again, a control list is used on each processor to store multiple outstanding synchronize requests at any given time. Once the user’s message has been buffered, a message of type SYNCHRONIZE_MESSAGE is sent up the span tree, where the root then initiates a message (again of type SYNCHRONIZE_MESSAGE) which is forwarded down the span tree to all members of the group via the SynchronizeDeliver entry point.

When this message is received, the control number is extracted, and the buffered message and its delivery information are retrieved from the control list indexed by that control number. The message is then delivered to the specified chare and entry point, and the control structure is deleted from the list.

The same reference number can be used multiple times, as long as the user can guarantee that no overlapping synchronization requests with the same reference number can occur. In the event that this does happen, an error message will be given by the library.
6 PERFORMANCE BENEFITS OF MESSAGE DRIVEN PARTITIONING

The main advantage of the Partition algorithm is that due to the message driven nature of Charm, many partition requests can occur concurrently, with the communication from each request naturally overlapping with other requests. This effect would be very hard to coordinate in a standard message passing model, and thus partition requests would need to be performed in a more sequential (non-overlapping) manner. As the number of partitions and processors increases, the ability to interleave partition requests becomes obvious.

To show this advantage, a program was written which performs a variable number of partition requests across varying numbers of processors. On each processor, the clock was started before the first partition request, and stopped after the last group identifier had been returned to the user program. The maximum time across all the processors in a group was then taken, since processors at the “top” of the span tree will receive notification before those at the leaves. These timings, gathered on a CM-5, are shown in table 6.1.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Number of Partitions</th>
<th>Overlapping (ms)</th>
<th>Sequential (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>10</td>
<td>81</td>
<td>87</td>
</tr>
<tr>
<td>32</td>
<td>100</td>
<td>2284</td>
<td>2480</td>
</tr>
<tr>
<td>32</td>
<td>500</td>
<td>19856</td>
<td>56100</td>
</tr>
<tr>
<td>64</td>
<td>10</td>
<td>418</td>
<td>397</td>
</tr>
<tr>
<td>64</td>
<td>100</td>
<td>5095</td>
<td>5019</td>
</tr>
<tr>
<td>64</td>
<td>500</td>
<td>27129</td>
<td>53822</td>
</tr>
<tr>
<td>128</td>
<td>10</td>
<td>753</td>
<td>765</td>
</tr>
<tr>
<td>128</td>
<td>100</td>
<td>8513</td>
<td>9446</td>
</tr>
<tr>
<td>128</td>
<td>500</td>
<td>53441</td>
<td>107389</td>
</tr>
<tr>
<td>256</td>
<td>10</td>
<td>1619</td>
<td>1528</td>
</tr>
<tr>
<td>256</td>
<td>100</td>
<td>19589</td>
<td>18994</td>
</tr>
<tr>
<td>256</td>
<td>500</td>
<td>101312</td>
<td>216577</td>
</tr>
</tbody>
</table>

Table 6.1: Advantages of Partitioning in a Message Driven Environment
Each processor performed a test to determine whether it belonged in a group or not. Thus, each *copy* was divided into two partitions – one partition containing the processors which passed a test, and the other containing processors which failed. In this case, the test was a modulo function which determined if the processor number was divisible by the current partition number.

The program used in this chapter is shown in its entirety in Appendix A.2. Notice the use of reference numbers in the partition calls to allow a single entry point to handle multiple partitioning responses.
7 EXAMPLE USAGE OF PROCESS GROUPS

In this chapter, several algorithms are outlined which could be implemented efficiently using the Process Groups Library. A dense matrix multiply solution is outlined, as well as a comparison based parallel sorting algorithm.

For further examples of usage, the user is directed to the Appendix, which contains several complete working programs. These programs are included in this thesis as examples of proper usage of the library, as the current base of programs using process groups is small.

7.1 Dense Matrix Multiplication Using Process Groups

This example implements the matrix multiplication algorithm, with both the initial (\( A \) and \( B \)) and result data (\( C \)) fully distributed across all processors. For reference, the complete working program used in this example is shown in Appendix A.3.

7.1.1 Problem Description

Initially, each processor contains a square portion of each of the initial matrices \( A \) and \( B \). The size of each portion is a function of both the total size of the matrix and the number of processors applied to the problem. Since each processor contains an equal sized portion, the block can be assigned and identified by a “coordinate” system. Although each block will likely contain partial data from several rows and several columns, from this point forward we are going to refer to “rows” and “columns” as \( x \) and \( y \) coordinate blocks respectively, unless otherwise specified. See figure 7.1 which shows the breakdown and distribution of the matrices.

After each processor determines which row and column it is in, it performs two partition calls. In the first, the copy number is specified as ROW and the partition number is determined by the row coordinate of that processor. The result of this partition is that all processors which contain a block of data for the same row (or group of rows) are in a common group. Likewise, a partition request is made to group processors which share a column number.

The matrix multiply algorithm specifies that to compute element \( C_{i,j} \) of the result matrix, the entire row \( A_i \) and the entire column \( B_j \) are required. Thus, after the groups are completed,
each processor communicates its data to all the other processors in their groups via the Process Groups Multicast. In turn, they will receive data from all the other processors in their same row and column, filling in a work array as the data is gathered. See figure 7.2.

At this point, each processor has exactly enough information to compute its portion of the \( C \) matrix without further communication. One each processor computes \( C_{i,j} \), it sends its partial result back to the main chare, which prints out the results of the total matrix.

7.1.2 Advantages of Process Groups and MDE

Due to the message driven nature of Charm, the two partition calls can be easily overlapped with each other, instead of issuing the first, waiting for a reply, and then issuing the second. This automatic overlapping of computation and communication improves program efficiency, as shown in Chapter 6. In addition, the order in which the various blocks arrive at a processor does not affect the performance of the algorithm.

This algorithm is by no means an optimal solution for solving a matrix multiply on distributed matrices. For one, it only works on square matrices and thus also requires that a
7.2 A Comparison Based Sorting Algorithm

Another possible application of process groups is in a parallel comparison based sorting algorithm, completely described in [26]. This algorithm uses the dynamic creation of groups and subsequent group communication extensively to implement a fast, scalable, and general purpose comparison based sorting program.

Although the details of the algorithm are beyond the scope of this thesis (the reader is referred to [26] for the complete description), the basic algorithm is shown in figure 7.3.

The Process Groups library would be ideal for this algorithm for several reasons:

1. During histogramming, both reductions and broadcasts are used across a span tree for efficient communication.

2. During each phase, the root multicasts its quintuple set down to the processors in its group.

3. At the end of each phase, groups are reconfigured dynamically.
for (phase = 1 to $\log_k P$)
    do
        Generate histogram probes and broadcast them.
        On each processor, find key counts for probes.
        Send counts up spanning tree to root.
        At root processor, use new set of counts to refine
        current best values of partition boundaries.
    while (key counts for each partition are equal)
        At root, generate $k$ quintuples per subtree.
        Send quintuples down spanning tree. At each internal
        node, split each quintuple among subtrees.
        On each processor, use $k$ quintuples to find what data
        to send to which processors.
        On each processor, send keys to other processors, then
        merge keys received from other processors.
        Reconfigure tree into $k$ separate spanning trees,
        one for each partition.
    endfor

Figure 7.3: Outline of sorting algorithm based on histogramming

As the utility of the Process Groups library was not available at the time the program was implemented, the sorting algorithm used an ad hoc implementation specific to the program.
8 FUTURE WORK

This chapter outlines several of the limitations of the process groups library, and possible future work to overcome them.

8.1 Moving Process Groups Into the System

If the Process Groups library proves to be an integral part of programming in the Charm language, performance could be increased by moving what is currently the Process Groups library into the Charm runtime system. A small amount of overhead is incurred by any user message which enters the runtime system.

By encapsulating the algorithms into a lower level, including group message data inside the message *envelope*, and giving the runtime system complete knowledge of the group structures (versus just the user program), certain performance benefits could be realized. In addition, coupling process groups with a load balancing strategy (applicable only if regular chares are allowed to form groups, see § 8.4), might be easier and more efficient. These points are all especially true for the network of workstations implementation of Charm, in which computation-to-communication ratios are greatly reduced, thus exaggerating the effects of communication overhead.

8.2 Combining Multicast messages

The current multicast implementation suffers from a slight performance loss because two messages must be sent for each multicasted message (see § 5.3.1). An alternative to this scheme is to copy the multicasted message, envelope and all, into the body of another message which would have its own envelope for traveling through the PG library.

The advantages to this scheme are that message startup costs are reduced by about 50% for each multicast, and the small overhead of buffering control and user messages in a hash table while waiting for the other to arrive is eliminated.
The main disadvantage to this scheme is the additional overhead of copying a message entirely into the body of another message. However, it is probable that the savings realized by reducing remote communication would far outweigh the cost of an additional memory copy.

8.3 Providing a Full Complement of Set Functions

Each group can be considered a set (by the mathematical definition). Each group, if properly formed, will contain only one instance of each processor. The partition phase of the PG library is essentially used to take all of the members of a set and divide them into subsets.

Other set functions could prove useful in extending the power of the Process Groups library. For example, providing the library with an arbitrary number of group identifiers, and returning the union or intersection of the resultant groups formed along with a new group identifier. Addition and subtraction of group members from one set into another is also a possibly useful function.

Not having these functions is not taking away from the expressivity of the Process Groups library, however, since these functions could actually be built “on top of” the existing Process Groups library (a “meta-library”). For example, the union of two groups could be achieved by multicasting a “join” message to both of these groups. After a synchronization, all processors would then issue another partition request based on whether they had received the initial “join” message (thus indicating that they appear at least once in one of the sets being unioned).

8.4 Ability to Work With Regular Chares

One of the advantages of Charm is the ability to create tasks in the form of regular chares, which are not dependent upon the number of processors, versus branch office chares. One limitation of the process groups library is that it only works across branch office chares. For example, the dense matrix program outlined in § 7.1 could be written with regular chares performing the work at each block, thus allowing any number of processors to be applied to the problem, with the Charm system automatically taking care of load balancing issues and scheduling of tasks on each processor.

It is feasible to adapt the Process Groups library to work in this manner. The ability to treat chares as first class (globally nameable) objects is a key factor in this case. In the
current implementation, processes are completely identified by their unique processor number. By replacing occurrences of these with chare identifiers, it is possible to build a Process Groups library which is generally applicable to chares and well as BOC’s.

Because a process group is built “on the fly”, determined by the order in which partition requests are issued to the root of each group, the locality of processes in a group is indeterminable before run-time, and in fact is likely to be different each time the program is run. To minimize the number of remote message calls issued during group communication, it would of course be desirable for processes which are neighbors in the span tree be located on the same processor. This is not an issue in the current implementation, since there is only one process per processor (an inherent feature of Branch Office Chares). In order for an efficient implementation of the Process Groups library to work with regular chares, the library would have to work with the load balancer built into Charm to help insure that this locality is optimized.

8.5 Ability to Set Priorities of Process Group Messages

Programs which use the message priority features of Charm will not be able to control the priority of messages inside the PG library. By default, messages which don’t have a priority attached receive the highest priority by the Charm system. In many cases, this effect is desirable, as collective communication is often required to be the most efficient phases of a program. However, this may not be desirable in all cases. Consider the case where collective communication is not performed until late in the program’s execution. By moving the partition requests to the beginning of the program, the user can ensure that other earlier phases of the program can efficiently overlap with the partitioning. By having the ability to lower the priority of the partitioning phase, the user could also ensure that the partitioning does not take away from other work being performed in the earlier phases.

One possible solution to this is to allow a “generic” priority to be applied to various modules by the user. However, this still may not be adequate, as it is possible that the user may wish to apply low priority to partition requests, while leaving the collective communication messages (multicast, reductions) at a high priority.

1On machines which have noticeably different remote access times bases on processor location, such as a store-and-forward hypercube architecture, there is a slight issue with locality in the current implementation.
The “module” feature of Charm is designed to abstract the functionality of a library from the user program, and hide information about the library. An ideal solution to this problem would be to allow the user to specify at load time (via something akin to the interface files) priorities to be applied to each message type.

8.6 Performance Degradation Due to non-FIFO Queueing Strategies

The process group library is also optimized to work with FIFO queue ordering. Although it will completely handle the possibility of out of order messages (such as a multicast arriving at a process before the group identifier has arrived, due to messages “passing” each other either on the network or by LIFO queue strategies), this requires some additional buffering of messages, and a slight overhead to handle these cases. If the user’s program is linked with a queueing strategy other than FIFO, the chances of these out of order messages occurring is much higher, thus incurring a slight performance degradation during the partition, multicast, and synchronize phases.

Currently, when a program is linked, a single queueing strategy is chosen for the entire system. It has been shown that different modules may take advantage of different queueing strategies. For example, Process Groups is optimized to work with FIFO queueing, while the user program may require LIFO or random queueing strategies to perform at maximum efficiency.

The ability to override the default “global” queueing strategy (chosen at link time) would be beneficial in cases such as this. This requires a new, more complex, queueing strategy. In future versions of Charm, users will be allowed to define their own queueing strategies, thus alleviating this problem.
9 CONCLUSIONS

This thesis has described a complete collective communication library written in the message driven environment of Charm. The main features discussed include:

- Advantages of message driven execution for collective communication.
- Efficiency of creating and using Process Groups.
- The dynamic nature of process group creation, and the algorithms used.
- The ease of use of the library.
- Possible applications of the library.

For programs which have a natural breakdown of processes (or branch office chares) across an algorithm, the Process Groups library first and foremost makes the creation and use of groups easy and natural for programmers using the Charm language. In addition, the dynamic building of Process Groups across optimized broadcast (span) trees gives the programmer an efficient base upon which to apply process groups in an algorithm.

Chapter 1 gave an introduction to why the ability to have process groups is advantageous both from an ease-of-programming standpoint, as well as efficiency. Chapter 2 gave an overview of the Charm system, including message driven execution. Chapter 3 outlined previous and ongoing work in the field of collective communication and message passing libraries. Chapter 4 gave a complete overview on how to use the Process Groups library, the available functions, a small example program, and an overview of the Charm reduction library, which was adapted to work with process groups as a part of this thesis. Chapter 5 went into the details and algorithms used in implementing the Process Groups library, important data structures for those wishing to modify the library, and why certain choices were made during the implementation. Chapter 6 applied the process group library to some kernel benchmark programs to show the performance benefits of using the Process Groups library. Chapter 7 described how the Process Groups library could be applied to real world applications, in this case describing how process groups can be used to implement a general class of matrix solutions. Chapter 8 outlined some of the
current limitations of the process groups library, and some future work which could be applied
to both the library and the Charm system to improve these limitations.
A EXAMPLE PROGRAMS

This appendix supplies the reader with several complete, working Charm programs. The first program, in § A.1, was described in § 4.2.3, and uses most of the functionality supplied with the library in a trivial example. The second program, in § A.2, was used in Chapter 6 to gather performance numbers from a CM-5 related to the partition algorithm, and its inherent advantages due to message driven execution. Finally, the third program, in § A.3, was described in § 7.1 as a possible solution to solving a distributed matrix multiply.

A.1 Odd/Even: Subgroups of Groups

```charm
#include "pg.int"
module example {
    message {} START_MESSAGE ;
    message { int sender ;
        char lowup[10] ;
        char evenodd[10] ;
    } TEST_MESSAGE ;
    BranchOffice TestBoc {
        int me, lower, odd ;
        ChareNumType coordBoc ;
        entry start : (message START_MESSAGE *inMsg) {
            me = McMyPeNum() ;
            if (me < McMaxPeNum() / 2) lower = 1 ; else lower = 0 ;
            if (me % 2 == 0) odd = 0 ; else odd = 1 ;
            if (me == 0)
                PG::CreateRootGroup(StartPartitioning, MyBocNum() ) ;
        }
        entry StartPartitioning : (message PG::ROOT_GROUP_CREATED *inMsg) {
            coordBoc = inMsg->rootBocNum ;
            PG::Partition(coordBoc, 0, 0, lower, LowerUpperPart, MyBocNum() ) ;
        }
        entry LowerUpperPart : (message PG::PARTITION_CREATED *inMsg) {
            int newGid ;
            TEST_MESSAGE *outMsg ;
            newGid = inMsg->newGid ;
            PG::Partition(coordBoc, newGid, 0, odd, OddEvenPart, MyBocNum() ) ;
        }
        entry OddEvenPart : (message PG::PARTITION_CREATED *inMsg) {
            TEST_MESSAGE *outMsg ;
            int newGid = inMsg->newGid ;
            outMsg = (TEST_MESSAGE *)CkAllocMsg(TEST_MESSAGE) ;
            outMsg->sender = me ;
            if (odd) strcpy(outMsg->evenodd, "odd") ;
            else strcpy(outMsg->evenodd, "even") ;
            if (lower) strcpy(outMsg->lowup, "lower") ;
            else strcpy(outMsg->lowup, "upper") ;
        }
    }
}```
entry deliverEp : (message TEST_MESSAGE *msg) {
    int ref = msg->sender;
    CkPrintf("**** This is # %d and I am %s and %s, as is my sender %d
        me, msg->lowup, msg->evenodd, ref); }
} /* end boc */

chare main {
    entry CharmInit: {
        START_MESSAGE *startMsg;
        startMsg = (START_MESSAGE *)CkAllocMsg(START_MESSAGE);
        CreateBoc(TestBoc, TestBoc@start, startMsg);
    } /* end chare main */
} /* end module */
A.2 Partition Benchmark

```c
#include "pg.int"
#define ROOT_GID 0
#define MAX_PART 5000  /* If you create more than this many partitions, 
                     reference numbers will clash */

module partbench {

message {ChareNumType pgCoord ; int np ;} START_MESSAGE ;
message {int total;} TIME_MSG ;
message {} SHOOTIN_BLANKS ;

chare main {
    ChareNumType pgCoord, overlapBOC, seqBOC ;
    int phase1done, phase2done, worstTime1, worstTime2 ;
    entry CharmInit :
    {
        START_MESSAGE *smsg1, *smsg2 ;
        SHOOTIN_BLANKS *dmsg ;
        int num_partitions ;

        CkPrintf("Enter number of partitions: ");
        CkScanf("%d", &num_partitions);

        pgCoord = PG::CreateRootGroup() ;
        phase1done = McTotalNumPe();
        worstTime1 = 0 ;
        worstTime2 = 0 ;
        smsg1 = (START_MESSAGE*)CkAllocMsg(START_MESSAGE) ;
        smsg1->pgCoord = pgCoord ;
        smsg1->np = num_partitions ;
        overlapBOC = CreateBoc(Overlap, Overlap@InitPhaseOne, smsg1) ;
        smsg2 = (START_MESSAGE*)CkAllocMsg(START_MESSAGE) ;
        smsg2->pgCoord = pgCoord ;
        smsg2->np = num_partitions ;
        seqBOC = CreateBoc(Sequential, Sequential@InitPhaseTwo, smsg2) ;

        /* start Phase One */
        dmsg = (SHOOTIN_BLANKS*)CkAllocMsg(SHOOTIN_BLANKS) ;
        BroadcastMsgBranch(Overlap@StartPhaseOne, dmsg, overlapBOC) ;
    }

    entry PhaseOneDone : (message TIME_MSG *msg)
    {
        START_MESSAGE *msg;
        SHOOTIN_BLANKS *dmsg ;
        if(msg->total > worstTime1) worstTime1 = msg->total ;
        CkFreeMsg(msg) ;
        if(--phase1done == 0) {
            CkPrintf("Worst Time, Phase1 = %d\n",worstTime1) ;
            phase2done = McTotalNumPe() ;
            /* Start Phase Two */
            dmsg = (SHOOTIN_BLANKS*)CkAllocMsg(SHOOTIN_BLANKS) ;
            BroadcastMsgBranch(Sequential@StartPhaseTwo, dmsg, seqBOC) ;
        }
    }
}
```
entry PhaseTwoDone : (message TIME_MSG *msg)
{
    if(msg->total > worstTime2) worstTime2 = msg->total;
    CkFreeMsg(msg);
    if(--phase2done == 0) {
        CkPrintf("Worst Time, Phase2 = %d\n",worstTime2);
        CkExit();
    }
}

BranchOffice Overlap {
    ChareNumType pgCoord, myBoc;
    ChareIDType mainChare;
    int amIDone;
    int me, divides;
    int startT, stopT;
    int num_partitions;
    entry InitPhaseOne : (message START_MESSAGE *sMsg)
    {
        pgCoord = sMsg->pgCoord;
        num_partitions = sMsg->np;
        CkFreeMsg(sMsg);
        me = McMyPeNum();
        myBoc = MyBocNum();
        MainChareID(&mainChare);
    }
    entry StartPhaseOne : (message SHOOTIN_BLANKS *dmsg)
    {
        int i;
        amIDone = num_partitions;
        CkFreeMsg(dmsg);
        startT = McTimer();
        /* Create partition based on dividing by i */
        for(i=2;i<num_partitions+2;i++) {
            divides = (me % i == 0) ? 1 : 0;
            PG::Partition(pgCoord, ROOT_GID, i, divides, i,
                Phase1Back, myBoc);
        }
    }
    entry Phase1Back : (message PG::PARTITION_CREATED *pMsg)
    {
        int ref;
        private void allDone();
        ref = GetRefNumber(pMsg);
        CkFreeMsg(pMsg);
        if(--amIDone == 0) {
            stopT = CkTimer();
            PrivateCall(allDone());
        }
    }
    private void allDone()
    {
        TIME_MSG *doneMsg;
        doneMsg = (TIME_MSG*)CkAllocMsg(TIME_MSG);
        doneMsg->total = stopT-startT;
        SendMsg(main@PhaseOneDone, doneMsg, mainChare);
    }
}
BranchOffice Sequential {
    CharNumType pgCoord, myBoc;
    CharIDType mainChare;
    int me, divides;
    int startT, stopT;
    int num_partitions;
    entry InitPhaseTwo : (message START_MESSAGE *sMsg) {
        pgCoord = sMsg->pgCoord;
        num_partitions = sMsg->np;
        CkFreeMsg(sMsg);
        me = McMyPeNum();
        myBoc = MyBocNum();
        MainChareID(&mainChare);
    }
    entry StartPhaseTwo : (message SHOOTIN_BLANKS *dmsg) {
        int ref;
        CkFreeMsg(dmsg);
        startT = McTimer();
        /* Create partition based on dividing by 2 */
        ref = 2;
        divides = (me % ref == 0) ? 1 : 0;
        PG::Partition(pgCoord, ROOT_GID, ref+MAX_PART, divides, ref,
                      Phase2Back, myBoc);
    }
    entry Phase2Back : (message PG::PARTITION CREATED *pMsg) {
        TIME_MSG *doneMsg;
        int ref;
        ref = GetRefNumber(pMsg);
        CkFreeMsg(pMsg);
        if(ref == num_partitions+1) {
            stopT = McTimer();
            doneMsg = (TIME_MSG*)CkAllocMsg(TIME_MSG);
            doneMsg->total = stopT-startT;
            SendMsg(main@PhaseTwoDone, doneMsg, mainChare);
        } else {
            ref++;
            divides = (me % ref == 0) ? 1 : 0;
            PG::Partition(pgCoord, ROOT_GID, ref+MAX_PART, divides, ref,
                          Phase2Back, myBoc);
        }
    }
}
A.3 The Matrix Multiply Example

#include "pg.int"
module Matmul {
extern double sqrt() ;
#define ROWS 0
#define COLS 1
#define SEED 54321 /* Some random number */
#define GSIZE 2
#define MAX_BLOCKS 23 /* 230x230 is largest matrix solvable and 529 processors is largest configuration */

message {
    int gSize ;
    int rootP ;
    ChareNumType pgCoord ;
} INIT_MSG ;
message {
    int rowBlock ;
    int colBlock ;
    float Chunk[GSIZE][GSIZE] ;
} BLOCK_MSG ;
chare main {
    float C[GSIZE*MAX_BLOCKS][GSIZE*MAX_BLOCKS] ;
    int getBack, rootP, gSize ;

    entry CharmInit: {
        INIT_MSG *msg ;
        ChareNumType pgCoord ;
        gSize = GSIZE ;
        rootP = ((int) sqrt((double)McMaxPeNum())) ;
        if(rootP > MAX_BLOCKS) {
            CkPrintf("Increase value of MAX_BLOCKS\n") ;
            CkExit() ;
            return ;
        }
        getBack = McMaxPeNum() ;
        if (McMaxPeNum()/rootP != rootP) {
            CkPrintf("Please choose a square-rootable number of processors.\n") ;
            CkExit() ;
            return ;
        }
        pgCoord = PG::CreateRootGroup() ;
        msg = (INIT_MSG*)CkAllocMsg(INIT_MSG) ;
        msg->rootP = rootP ;
        msg->gSize = gSize ;
        msg->pgCoord = pgCoord ;
        CreateBoc(Compute, Compute@Init, msg) ;
    }

    entry collectResults : (message BLOCK_MSG *resultMsg) {
        private void PrintFinalResults() ;
        int fromRowBlock, fromColBlock, i,ii,j,jj, n ;
        int realRowStart, realColStart ;
        fromRowBlock = resultMsg->rowBlock ;
        fromColBlock = resultMsg->colBlock ;
        realRowStart = (gSize*fromRowBlock) ;
        realColStart = (gSize*fromColBlock) ;
    }
}
for(i=0,ii=realRowStart; i < gSize; i++,ii++)
    for(j=0,jj=realColStart; j < gSize; j++,jj++)
        C[ii][jj] = resultMsg->Chunk[i][j] ;
CkFreeMsg(resultMsg) ;
if (--getBack == 0)
    PrivateCall(PrintFinalResults()) ;
}
private void PrintFinalResults() {
    int i,j ;
    for(i=0;i<gSize*rootP;i++) {
        for(j=0;j<gSize*rootP;j++)
            CkPrintf("%5.3f ",C[i][j]) ;
        CkPrintf("\n") ;
    }
    CkExit() ;
}

BranchOffice Compute {
    int me ;
    ChareIDType myID ;
    int rootP, gSize ;
    int rowBlock, colBlock ;
    int rowGid, colGid ;
    ChareNumType pgCoord, myBocNum ;
    int joinedRow, joinedCol, rowBlocksLeft, colBlocksLeft ;
    float A[GSIZE][GSIZE] ;
    float B[GSIZE][GSIZE] ;
    float C[GSIZE][GSIZE] ;
    float Arows[GSIZE*MAX_BLOCKS][GSIZE] ;
    float Bcols[GSIZE][GSIZE*MAX_BLOCKS] ;
    entry Init : (message INIT_MSG *msg) {
        int i,j ;
        rootP = msg->rootP ;
        gSize = msg->gSize ;
        pgCoord = msg->pgCoord ;
        me = McMyPeNum() ;
        MyBranchID(&myID) ;
        myBocNum = MyBocNum() ;
        joinedRow = joinedCol = 0 ;
        rowBlocksLeft = rootP ;
        colBlocksLeft = rootP ;
        rowBlock = (me/rootP) ;
        colBlock = (me%rootP) ;
        /* A and B will hold the data originally assigned to each branch.
         * Arows and Bcols will hold the row data coming in from the other
         * members of the same row or column group.
         */
        srand(SEED+McMyPeNum()) ;
        for(i=0;i<gSize;i++) {
            for(j=0;j<gSize;j++)
                A[i][j] = rand() / 10000000. ;
                B[i][j] = rand() / 10000000. ;
        }
        CkFreeMsg(msg) ;
        /* Join the row partition */
        PG::Partition(pgCoord, 0, ROWS, rowBlock, ROWS, RowJoined, myBocNum) ;
    }
}
/* Join the column partition */
PG::Partition(pgCoord, 0, COLS, colBlock, COLS, ColJoined, myBocNum);
}

entry RowJoined : (message PG::PARTITION_CREATED *msg) {
private void DistributeData() ;
rowGid = msg->newGid ;
CkFreeMsg(msg) ;
joinedRow = 1 ;
if (joinedRow && joinedCol)
    PrivateCall(DistributeData()) ;
}

entry ColJoined : (message PG::PARTITION_CREATED *msg) {
private void DistributeData() ;
colGid = msg->newGid ;
CkFreeMsg(msg) ;
joinedCol = 1 ;
if (joinedRow && joinedCol)
    PrivateCall(DistributeData()) ;
}

private void DistributeData() {
    int a, i, j ;
    BLOCK_MSG *bMsg ;
    for(a=0; a<2; a++) {
        /* Bundle up and send off the data to the other members in
        * my row or column partition */
        bMsg = (BLOCK_MSG*)CkAllocMsg(BLOCK_MSG) ;
        bMsg->rowBlock = rowBlock ;
        bMsg->colBlock = colBlock ;
        for(i=0;i<gSize;i++) {
            for(j=0;j<gSize;j++) {
                if(a==0)
                    bMsg->Chunk[i][j] = A[i][j] ;
                else
                    bMsg->Chunk[i][j] = B[i][j] ;
            }
        }
        if(a==0)
            PG::Multicast(pgCoord, colGid, bMsg, CollectRowData, myBocNum) ;
        else if (a==1)
            PG::Multicast(pgCoord, rowGid, bMsg, CollectColData, myBocNum) ;
    }
}

entry CollectRowData : (message BLOCK_MSG *rMsg) {
private void Multiply() ;
int fromRowBlock, realRowStart ;
int ii, i, j ;
fromRowBlock = rMsg->rowBlock ;
realRowStart = (gSize*fromRowBlock) ;
for(i=0, ii=realRowStart ; i<gSize ; ii++, i++) {
    for(j=0; j<gSize; j++)
        Arows[ii][j] = rMsg->Chunk[i][j] ;
}
CkFreeMsg(rMsg) ;
if ((--rowBlocksLeft==0) && (colBlocksLeft==0))
    PrivateCall(Multiply()) ;
}

entry CollectColData : (message BLOCK_MSG *cMsg) {

private void Multiply() {
int fromColBlock, realColStart;
int jj, i, j;
fromColBlock = cMsg->colBlock;
realColStart = (gSize*fromColBlock);
for(i=0; i<gSize; i++)
  for(j=0; jj=realColStart; j<gSize; jj++, j++)
    Bcols[i][jj] = cMsg->Chunk[i][j];
CkFreeMsg(cMsg);
if ((--colBlocksLeft==0) && (rowBlocksLeft==0))
  PrivateCall(Multiply());
}

private void Multiply() {
int i, j, k, n;
BLOCK_MSG *bMsg;
ChareIDType mainChare;
for(i=0; i<gSize; i++)
  for(j=0; j<gSize; j++)
    for(k=0; k<gSize*rootP; k++)
      C[i][j] += Arows[k][i] * Bcols[j][k];
bMsg = (BLOCK_MSG*)CkAllocMsg(BLOCK_MSG);
bMsg->rowBlock = rowBlock;
bMsg->colBlock = colBlock;
for(i=0; i<gSize; i++)
  for(j=0; j<gSize; j++)
    bMsg->Chunk[i][j] = C[i][j];
MainChareID(&mainChare);
SendMsg(main@collectResults, bMsg, mainChare);
}

}/* end: boc */
}/* end: module */
BIBLIOGRAPHY


[19] William Gropp and Ewing Lusk. An abstract device definition to support the implementation of a high-level point-to-point message-passing interface. Technical Report MCS-


[30] Parallel Programming Laboratory, Department of Computer Science, University of Illinois. 


