Parallel Solution of the Time-dependent Ginzburg-Landau Equations and Other Experiences Using BlockComm-Chameleon and PCN on the IBM SP, Intel iPSC/860, and Clusters of Workstations

by E. Coskun and M. K. Kwong
Argonne National Laboratory, with facilities in the states of Illinois and Idaho, is owned by the United States government, and operated by The University of Chicago under the provisions of a contract with the Department of Energy.

---

**DISCLAIMER**

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

---

Reproduced from the best available copy.

Available to DOE and DOE contractors from the Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831
Prices available from (423) 576-8401

Available to the public from the National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161
Parallel Solution of the Time-dependent Ginzburg-Landau Equations and Other Experiences Using BlockComm-Chameleon and PCN on the IBM SP, Intel iPSC/860, and Clusters of Workstations

by

Erhan Coskun\textsuperscript{1} and Man Kam Kwong\textsuperscript{2}

Mathematics and Computer Science Division

September 1995

\textsuperscript{1}Department of Mathematical Sciences, Northern Illinois University, DeKalb, IL 60115. Present address: Karadeniz Technical University, Department of Mathematics, Trabzon, 61080 Turkey. E-mail: erhan@oesf01.bim.ktu.edu.tr

\textsuperscript{2}This author was supported by the Mathematical, Information, and Computational Sciences Division sub-program of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.
## Contents

Abstract

1 Introduction

2 Preliminaries

3 Test Problems

4 Parallel Programs with BlockComm/Chameleon
   4.1 ProgSumBC
   4.2 ProgPiBC
   4.3 ProgPdeBC
   4.4 ProgTdglBC

5 Clusters of Workstations as a Parallel Computing Environment

6 Parallel Programs with PCN
   6.1 ProgPiPCN
   6.2 ProgPdePCN

7 Conclusion

Acknowledgments

Appendix: Program Listings

References
Parallel Solution of the Time-dependent Ginzburg-Landau Equations and Other Experiences Using BlockComm-Chameleon and PCN on the IBM SP, Intel iPSC/860, and Clusters of Workstations

by

Erhan Coskun and Man Kam Kwong

Abstract

Time-dependent Ginzburg-Landau (TDGL) equations are considered for modeling a thin-film finite size superconductor placed under magnetic field. The problem then leads to the use of so-called natural boundary conditions. Computational domain is partitioned into subdomains and bond variables are used in obtaining the corresponding discrete system of equations. An efficient time-differencing method based on the Forward Euler method is developed. Finally, a variable strength magnetic field resulting in a vortex motion in Type II High Tc superconducting films is introduced.

We tackled our problem using two different state-of-the-art parallel computing tools: BlockComm/Chameleon and PCN. We had access to two high-performance distributed memory supercomputers: the Intel iPSC/860 and IBM SP. We also tested the codes using, as a parallel computing environment, a cluster of Sun Sparc workstations.

1 Introduction

In our study of the mathematical modeling of superconductivity, we have developed an efficient algorithm to solve numerically the time-dependent Ginzburg-Landau (TDGL) equations in two dimensions (see [3]). The corresponding problem in three dimensions is, however, very computationally extensive. The study is impractical on a conventional uniprocessor computer, even if the most efficient algorithm is used. The numerical simulation of such Grand Challenge problems (the three-dimensional TDGL in its entire generality) depends on high-performance computing techniques and resources.

We tackled the 3D problem using two different state-of-the-art parallel computing tools: BlockComm/Chameleon and PCN, the development of both involves Argonne scientists. Since the completion of this work, a new tool, the Message Passing Interface (MPI) [9], has emerged. It has an excellent prospect to become the standard message-passing tool. Future extension of our work will definitely include MPI. We had access to two high-performance distributed-memory supercomputers: the Intel iPSC/860 and IBM SP. We also tested the codes using, as a parallel computing environment, a cluster of Sun Sparc workstations in the Mathematics and Computer Science Division of Argonne National Laboratory.

Although our main objective was to develop a parallel code for the forward Euler method (see [3]) to solve the TDGL equations, we started with three simpler warm-up problems. Our experience with these three problems is also described here; they are used as examples to illustrate some of the concepts of parallel programming tools. More in-depth discussion of all the problems considered in this report, together with all complete parallel codes and running procedures, is given in [3].

Dave Levine of Argonne National Laboratory has also developed parallel codes for solving the TDGL using BlockComm, but with a different method of discretizing the equations; consult
[5], [10], and the forthcoming paper [16]. Earlier, two other colleagues, Paul Plassmann and Steve Wright, developed a parallel code for solving the static Ginzburg-Landau equations using optimization techniques; their work is reported in [6].

2 Preliminaries

We begin by introducing some terminology that will be used throughout this report. We also briefly describe the parallel programming tools and environments we used.

When a particular instance of a code or a part of a code is executed on a machine, all of the work needed to execute that portion of the program is referred as a single task, or process. Parallel processing is information processing or numerical computation that emphasizes the concurrent manipulation of data elements belonging to one or more processes in solving a single problem.

Early supercomputers achieved concurrency with the method of pipelining, namely, by dividing a computation into a number of steps that are processed in an assembly-line fashion. More modern architectures use multiple CPUs, each capable of executing instructions entirely independently of others.

How a processor accesses the computer memory (shared memory or distributed memory) affects how a parallel program will be designed and coded. It is generally accepted [18] that shared-memory parallel programming can usually be done through minor extensions to existing programming languages, operating systems, and code libraries. On the other hand, distributed-memory programming is a bit more involved, but it has the advantages of massive parallelism. Our experiments were done exclusively on distributed-memory environments.

A parallel system [17] is the combination of an algorithm and the parallel architecture on which it is implemented. As mentioned in [17], the performance of a parallel algorithm cannot be evaluated in isolation from a parallel architecture. Therefore, it is more appropriate to talk about performance of a parallel system than performance of a parallel algorithm.

Various metrics are used to measure the performance of a parallel system. We mention only a few of them below.

- The parallel run time is the elapsed time from the moment a parallel computation starts to the moment the last processor finishes execution.

- The speedup is defined as

\[ S_p = \frac{\text{serial run time for the best sequential algorithm}}{\text{parallel run time using } p \text{ processors}}. \]

The speedup \( S_p \) represents the benefit of solving a problem in parallel using \( p \) identical processors. A more practical definition (since it is often difficult to determine the best sequential algorithm) is obtained by replacing the expression in the numerator above by “execution time of the same code using a single processor.” We are using the second definition to evaluate our numerical results.

- The efficiency is defined as

\[ E_p = \frac{S_p}{p}. \]

In the ideal case of perfect speedup, \( S_p = p \), and \( E_p = 1 \).
• The cost of solving a problem on a parallel system is defined as the product of parallel run time and the number of processors used. It reflects the sum of time that each processor spends solving the problem.

The generic goal in the development of parallel algorithms is to achieve as high a speedup as possible. The perfect speedup \( S_p = p \), or optimal efficiency \( E_p = 1 \), is obtainable only for essentially trivial problems. All causes of imperfect speedup of a parallel system are collectively referred to as the overhead resulting from parallel processing. Some factors that cause overhead are as follows (see [13], [17], and [18]):

- lack of a perfect degree of parallelism in the algorithm,
- lack of perfect load balancing,
- communication or contention time, and
- extra computation.

In the ideal situation when each computational step of an algorithm can be done independently of the other steps, we say that the algorithm has a perfect degree of parallelism. In reality, this rarely happens. A processor often must wait in the middle of a run until it has received all the data or information from other processors it needs to execute the next computational step.

Load balancing is the assignment of tasks to the processors of the system so as to keep each processor doing useful work for as much of the time as possible. The determination of this optimal assignment is also called the mapping problem. Load balancing may be achieved either-statically or dynamically. In static load balancing, tasks are assigned to processors at the beginning of a computation. In dynamic load balancing, tasks are assigned to processors as the computation proceeds.

In distributed-memory system, each processor can address only its own local memory. Communication between processors takes place by message passing, a process that takes relatively more time than direct access to local memory. In a shared-memory system, all the processors have access to a common memory. Each processor can also have its own local, but limited, memory for program code and intermediate results. Communication between individual processors is through the common memory. A major advantage of a shared memory system is the rapid communication of data between processors. A serious disadvantage is that different processors may wish to use the common memory at about the same time (especially when new values are to be deposited), in which case there will be a delay until the memory is free, or until the proper order of access is established. This delay is called contention time.

An efficient serial algorithm may not lend itself to efficient parallelization because of the dependency of computational steps on results from previous steps. As a consequence, a redesign of the algorithm necessitating extra computation may be required. In an extreme situation, a better serial algorithm may have to be sacrificed in favor of an inferior one.

We close this section by introducing the parallel programming tools Chameleon, BlockComm, and PCN, used in our study.

Chameleon is a library of low-level, comprehensive, and very efficient message-passing routines developed by W. Gropp and B. Smith [11].

BlockComm is a library of high-level message-passing routines designed by Gropp to manage the efficient communication of blocks of data between processors. It provides shortcuts for many common message-passing tasks often found in the computational technique of
domain decomposition. Both packages are still under active development. One can consult [7] for the most current documentation about BlockComm. Although the use BlockComm greatly simplifies the coding of domain decomposition algorithms, it does not provide the data reduction and broadcast routines that are needed in our case. Hence, we have used a combination of Chameleon and BlockComm routines in the same program. Although both packages have both Fortran and C versions, we have chosen Fortran as our programming language (see Section 7).

PCN (Program Composition Notation) is a parallel programming language developed jointly by Argonne (I. Foster), Caltech, and the Aerospace Corporation. It provides a paradigm for composing parallel programs out of modules of parallel or sequential subroutines that may be written either in PCN itself or in more conventional programming languages. The programmer needs to specify only which modules are to be run concurrently and what data communications are needed between modules. The actual assignment of tasks to specific processors and message passing are transparent to the programmer. See [4] for more information and its use for various parallel environments.

The two programming tools we used are highly portable over a wide variety of computer architectures. We have used three different parallel environments in our study: the Intel iPSC/860, the IBM SP, and clusters of Sun Sparc workstations. All of them are distributed-memory multiple instruction multiple data (MIMD) systems. For each problem, the same program (recompiled with the appropriate makefiles) were used in the three systems. The Intel iPSC/860 at Argonne has eight nodes. All processor nodes are identical and are connected by bidirectional links in a hypercube topology. See [1] for its hardware and software specifications. We used this machine mainly for program development because it is freely accessible and there is no limitation on the amount of time one can work on the machine. The Argonne IBM SP³ has 128 nodes. Each node is an RS/6000 model 370 and has 128 MBytes of memory per node, 1 GByte local disk per node, full Unix on each node, and a high-performance Omega switch. The peak performance of each node is 125 MFlops. There are several transport layers on the SP including EUI, EUIH, and p4. EUIH is the low-overhead implementation of the EUI interface. EUI is IBM's message-passing interface to the high-performance switch. See [8] for more current information about the SP and how to use these transport layers.

3 Test Problems

In this section, we describe the four test problems in our experiments. Our ultimate goal is to develop a parallel code implementing the forward Euler algorithm for the TDGL equations. As warm-up trials, we experimented with three simpler but computationally intensive problems.

The first two problems are examples of the partitioning technique known as functional decomposition; the others use the domain decomposition technique.

**Problem 1:** We consider the slowly divergent harmonic series

$$
\sum_{i=1}^{N} \frac{1}{i}.
$$

Mathematicians are interested in investigating its rate of divergence. The extremely slow rate of divergence of the series means that a large number of terms will be needed in numerical

---

³The work described in this report was done during the period of May 1993–May 1994. Since then, the SP system at Argonne has been upgraded, and more efficient communication switches have been installed.
experiments, and this requirement makes the problem an interesting example for parallel programming. A parallel code using BlockComm to compute the partial sums will be presented together with some performance results. The code will be referred to as ProgSumBC.

**Problem 2:** Our second problem is a well-known simple numerical integration problem. It has been the arch-example used in the introduction of many parallel programming tool manuals. The objective is to approximate the integral

\[ \int_0^1 \frac{4}{1 + x^2} \, dx, \]

where

\[ f(x) = \frac{4}{1 + x^2}, \]

by using the rectangular rule:

\[ I_n(f) = h \sum_{i=1}^{n} f(x_i), \]

where \( h = \text{number} \) and \( x_i = (i - \frac{1}{2})h \). One can easily modify ProgSumBC to obtain a parallel BlockComm code for this problem. A parallel PCN code for this problem, named ProgPiPCN, will also be presented.

**Problem 3:** We study the following two-dimensional PDE:

\[ -u_{xx} - u_{yy} + cu - xy(cy^2 - 6) = 0 \quad (3.2) \]

in \((0,1) \times (0,1)\) with the boundary conditions

\[ u(x,0) = 0, \quad u(x,1) = x, \quad u(0,y) = 0, \quad u(1,y) = y^3, \]

where \( c \) is a constant. The exact solution, as one can easily verify, is \( u = xy^3 \). By approximating the second derivatives in the PDE by the usual central difference formulas, we obtain the linear system of equations

\[ \left( \frac{U - 2U + \bar{U}}{\Delta x^2} \right) - \left( \frac{U^{11} - 2U + U^{11}}{\Delta y^2} \right) + cU - x_i y_j (cy_j^2 - 6) = 0, \quad (3.3) \]

for \( i = 1, \ldots, N - 1, \quad j = 1, \ldots, M - 1, \) where \( \Delta x = 1/N, \quad \Delta y = 1 \text{method}, \quad x_i = i \Delta x, \quad y_j = j \Delta y \). We use the notation \( U^{11} \) to denote the value of \( U \) at the point above the current one, and so on.

By expanding the function \( u(x,y) \) as a Taylor series at the point \((x_i, y_j)\), we see that the truncation error involves only the fourth-order derivatives of \( u(x,y) \). Since \( u(x,y) = xy^3 \), both \( u_{xxx} \) and \( u_{yyyy} \) are identically zero. Therefore, the truncation error is identically zero as well. When the parameter \( c \) is greater than approximately \(-2\pi^2\), the coefficient matrix in the linear system is positive definite (see [21]). The SOR (successive overrelaxation) method is, therefore, guaranteed to converge if the relaxation parameter is chosen from the interval \((0,2)\). The parallel codes for this problem with BlockComm and PCN, which we named ProgPdeBC and ProgPdePCN, respectively, are given in the appendix.

**Problem 4:** Mathematical details of the TDGL are given elsewhere (see [3], [14], [15], and the references cited therein). It suffices to say that we are solving a system of (partial differential) evolution equations governing two unknown functions of time and space position: a
complex-valued scalar \( \phi \) (called the order parameter); and a three-dimensional vector \( A \) (called the vector potential). We used an unconventional method (see [14]) to discretize the equations with respect to the space variables. The resulting system is then solved using a forward Euler method. A parallel BlockComm code ProgTdlBC, for implementing this algorithm is given in the Appendix. Since the code itself is rather complicated and specialized, we will present in this report only the performance results, and refer the readers to [3] for a detail discussion of the code. We note that we have also developed a parallel PCN code for this problem, but performance results were less complete. As a consequence, we have decided not to present the code in this report.

4 Parallel Programs with BlockComm/Chameleon

4.1 ProgSumBC

ProgSumBC is the parallel program for Problem 1 written with BlockComm and Chameleon. We give the program listing below and explain its content. The line numbers in the listing have been added for easy reference and are not part of the code. The subroutine calls that begin with the letters \( BC \) are BlockComm routines, while those that begin with \( PI \) are Chameleon routines. The first five lines of the program declare the appropriate function name and variables.

```
1  integer function worker()
2  integer nbytes, PImytid, myid, sx, ex, N
3  integer intsize, msg_int, Psallprocs
4  parameter(intsize=4, msg_int=1, Psallprocs=0, nbytes=8)
5  double precision t1, t2, SYGetElapsedTime
```

Strictly speaking, the name ProgSumBC refers to the file PRofSumBC.f that contains a Fortran subroutine, called worker(), as declared in line 1 above. The worker() subroutine looks very much like the corresponding sequential code for the same problem, consisting of instructions for the numerical computations. In the actual execution of a parallel program, the computer needs some extra overhead instructions, such as initial setup directives (to round up the processors, to establish communication links among them) and clean-up directives (needed after all the computations are finally completed). Many parallel programming tools require the programmer to explicitly include these instructions in their programs. Chameleon also has these instructions, such as PICall used to call worker() in a parallel execution mode, but it provides a convenient alternative that frees a user from this extra effort. Overhead instructions that are common to most programs have been collected in a main subroutine and precompiled into the object files fmain.o (for Fortran codes) and cmain.o (for C codes), the appropriate one of which is to be linked to the computational subroutine when compiling the program. The moderate price to pay is that one no longer thinks in terms of writing a main Fortran code (or a main() routine in C), but just a function, with the mandatory name worker(), as we have done in line 1.
When the code is executed on the computer, every processor is given the same set of instructions contained in ProgSumBC, but not every processor will execute all the steps contained in the program. The program uses the ID number of the calling processor (obtained in line 6 using the Chameleon routine PImytid() and assigned to the variable myid) to determine which segments of codes are appropriate for the processor. Lines 7 to 10 are an example of such a segment. One of the processors, that with $ID \neq 0$, is given the responsibility to obtain (interactively) the user's input of the number of terms in the harmonic series to be summed.

Line 11 calls the Chameleon routine PIbcastSrc to broadcast the value N to all processors. Even though only processor $ID = 0$ is the sender, and all other processors are receivers, this routine must be called by all the processors. Roughly speaking, PIbcastSrc is shorthand for processor $ID = 0$ to send a message to all other processors, and for all other processors to wait for this message to arrive. The arguments of PIbcastSrc are, respectively, the variable (buffer) that contains the message, the size of the buffer, the ID of the processor that broadcast the message, the set of processors that receive the message (by conventions, all processors are involved when this argument is 0), and the data type of the message. For more precise syntax definitions of Chameleon routine calls, consult the Chameleon manual [11].

Now that each processor knows the value of N, the next step is to find out the range of those terms in the harmonic series that it is responsible to work on. This is done in line 12, by calling the subroutine getindex to compute the indices of the starting term sx and the last term ex in the range. The subroutine getindex is given below.

In line 13, a global synchronization call is used to make all the processors begin timing at the same time. Lines 14 and 16 return the elapsed time used by the subroutine compute in line 15, which does the actual summing.
subroutine getindex(mx,sx,ex)
include '/home/gropp/tools.n/blkcm/meshf.h'
integer mx, sx, ex, nd
integer sz(0:9,0:0)
integer myid, nproc, PINumtids, PINytid

nd=1
sz(szmdim,0) = mx
sz(szisparallel,0) = 1
sz(szndim,0) = -1
myid = PINytid()
nproc = PINumtids()
call BCGlobalToLocalArray( nd, sz, nproc, myid )
sx = sz(szstart,0) + 1
ex = sz(szend,0) + 1
return
end

The BlockComm subroutine BCGlobalToLocalArray determines the appropriate data domain that a processor is responsible for, given the decomposition style nd, the number of processors nproc, and the processor ID # myid. The BlockComm call stores its results in the array sz. The precise definitions of each component of sz are given in the manual.

subroutine compute(sx,ex,myid)
integer sx, ex, i, myid
double precision sum, work
sum=0.0
do i=sx,ex
   sum=sum+ld0integral
endo
call PIgdsum(sum,1,work,0)
if (myid .eq. 0)then
   print*, 'sumall=',sum
endif
return
end

The first part of compute finds the partial sum of the series from the term with index sx to the term with index ex, inclusively. The call PIgdsum finds the (g)lobal ((d)ouble precision) sum, by adding up all the results stored in the local variable sum attached to each processor. The other arguments of PIgdsum are, respectively, the length of the array sum (in the current case, sum is a scalar and so the value of this argument is simply 1), a variable work of the same size as sum to be use as a work area to compute the global sum, and the set of processors involved (as mentioned earlier, a value of 0, by conventions, denotes that all processors are to be included). The result of the computation, the global sum, overwrites the local sum originally stored in the variable sum.

Some self-explanatory performance results are illustrated in Figure 1.
4.2 ProgPiBC

One needs only to modify the computation routine `compute` in ProgSumBC to get a parallel code for Problem 2 in BlockComm. As a matter of fact, the only difference between Problem 1 and Problem 2 is the form of the terms in the series to be summed. In other words, the only changes needed are in modifying the line "sum=sum+ld0integral."

We include this example to make the point that once a prototype parallel program has been written, most of it can be reused to write another program. Hence, the initial investment is worthwhile.

4.3 ProgPdeBC

Our method of solution for Problem 3 is to decompose the domain in which the partial differential equation is defined into as many subdomains as the number of processors used. Each
processor is assigned the data of one of the subdomains, called a block, and a share of the computations that involves mainly data in the associated block. At each time step, each processor also requires some extra information from processors associated with neighboring blocks in order to complete the assigned computation. In most domain decomposition algorithms for solving partial differential equations, this extra information is typically data carried by a set of lattice points, the so-called ghost points, that borders the subdomain. The exchange of information among processors is performed by message-passing library calls. A two-dimensional computational domain with a typical subdomain and its ghost points for a five-point stencil is illustrated below.

![Figure 2. A nine-processor decomposition of a 2D domain with ghost points (o)](image)

If only a general-purpose, low-level message-passing tool, such as Chameleon, is used to write a parallel domain decomposition algorithm, one has to include explicit code segments to

1. define each subdomain (i.e., determine the ranges of indices for the lattice points that belongs to the subdomain),
2. map each subdomain to a processor,
3. determine the ghost points and the flow of messages, and
4. send and receive each message explicitly.

BlockComm provides subroutine calls to automate these steps for a wide class of common domain decomposition algorithms for rectangular domains. For example, the call BCGlobalToLocalArray, used earlier in the subroutine compute in Section 4.1, takes care of Steps 1–3. Another subroutine BCexec() can be used to automate Step 4.

The complete ProgPdeBC is given in the Appendix. Some performance results are presented in Table 1. For this particular experiment, \( c = 20, w \) (relaxation parameter) = 1, and we have used 500 grid points and 1000 iteration steps.
Table 1. Performance results for the ProgPdeBC-SP1-EUIH system

<table>
<thead>
<tr>
<th>Num. of Proc.</th>
<th>Parallel Run Time</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1294.95</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>691.55</td>
<td>1.8725</td>
<td>0.9363</td>
</tr>
<tr>
<td>4</td>
<td>536.82</td>
<td>2.4123</td>
<td>0.6031</td>
</tr>
<tr>
<td>8</td>
<td>319.12</td>
<td>4.0579</td>
<td>0.5072</td>
</tr>
<tr>
<td>12</td>
<td>245.38</td>
<td>5.2773</td>
<td>0.4398</td>
</tr>
<tr>
<td>20</td>
<td>224.40</td>
<td>5.7707</td>
<td>0.2885</td>
</tr>
</tbody>
</table>

4.4 ProgTdglBC

The code for ProgTdglBC is rather long and is given in the Appendix. It has been run on the Intel iPSC/860, the IBM SP, and a cluster of Sun workstations without further modification.

Typical performance results for the ProgTdglBC-iPSC/860 and ProgTdglBC-SP1-P4 systems are plotted in Figure 3. The latter uses the version of BlockComm that is based on the p4 macro package, developed by E. L. Lusk at Argonne, and uses the Ethernet transport layer.

The graph suggests that the speedup for the first parallel system is far better than that of the second. This is due to the fact that our test problem has a rather low granularity for the SP. As a result, SP nodes have to spend more time in communication than in computation. This explanation is confirmed by the fact that when we switched to the more efficient transport layer EUIH for the SP, the speedup curve shows a much better performance.
Figure 3. Speedup for the ProgTdg1BC-SP1-P4 and ProgTdg1BC-iPSC system
5 Clusters of Workstations as a Parallel Computing Environment

Due to the low access priority given to parallel jobs running in the background, performance on a cluster of workstations is not consistent, varying according to the demand of other users on the workstations. This environment is, therefore, mainly used for test runs and for debugging. Also, we observed that process creation on remote workstations takes a considerable amount
of time. Typical performance results obtained by running ProgPiBC with \( n = 10,000,000 \) on a collection of workstations are shown below. Here, the real and system times are obtained by the Unix's time command and elapsed time is computed by the program.

Table 2. Performance results for ProgPiBC on a cluster of workstations
(time in sec)

<table>
<thead>
<tr>
<th>Time</th>
<th>Number of Workstations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Real</td>
<td>19.0</td>
</tr>
<tr>
<td>System</td>
<td>0.2</td>
</tr>
<tr>
<td>Elapsed</td>
<td>17.8</td>
</tr>
</tbody>
</table>

6 Parallel Programs with PCN

6.1 ProgPiPCN

The syntax of PCN is similar to that of C. The comma, however, is used as the command terminator, while the semicolon is used to declare a sequential procedure. ProgPiPCN consists of five PCN procedures and a Fortran procedure. The arguments argc and argv of main() have the usual meanings as in C, and rc is used for a return code. But unlike in C, the arguments to main() must be specified in the definition, whether we are planning to pass any command line arguments to the program or not. Line 2 serves a dual purpose: the number of command line arguments is checked, and if that is equal to two, the values of argv[1] and argv[2] are assigned to n_intervals and interval_size. In lines 4–5, PCN’s sys module is used to define ni and li to be the integer values represented by the strings n_intervals and interval_size, respectively. In lines 6–7, the total number of points and the width of
the intervals are computed. Line 8 is a call to the procedure `main_body`; the infix operator `in` is used to specify the map function `vts: array(ni)`, which creates a virtual array topology of size `ni`. This topology guarantees the portability of the program across different computer platforms. See [4] for more on virtual topologies and map functions. Line 9 sets the return code variable to zero. Lines 11–13 print an error message in case the number of arguments supplied is wrong.

```
16   main_body(ni,li,width)
17   port globals[nodes()];
18   {|| rectangle(ni,li,width,globals),
19      display(0,0,globals,ni)
20  }
```

The built-in function `nodes()` determines the number of nodes present. In line 17, a port array `globals` with `nodes()` elements is created. This port array is used for the global operations to be performed later. Lines 18–19 are two procedure calls to be executed in parallel mode. The first procedure call implements the rectangular rule to approximate the value of \( \pi \), and the second displays the results. The role of the arguments passed to these procedures is clear from the context of the program.

```
21   rectangle(ni,li,width,globals)
22   port globals[];
23   {|| i over 0 .. ni-1 :: /* ni intervals */
24      start_interval(i,li,width,globals[i])@node(i)
25  }
```

```
start_interval(i,li,width,globals)
  double sum;
  {;
    compsum_(li,width,sum),
    stdio:printf("li=%d width=%f sum=%f\n",{li,width,sum},_),
    globals={sum},
    stdio:printf("globals=%f\n",{globals},_),
  }
```

The iterative construct in line 23 creates `ni` instances of `start_interval()`, each of which calls the Fortran procedure `compsum` to compute the local contribution to the value of \( \pi \). This value is snapshot by the definitional variable `globals` for use in the procedure `display`.

```
display (count,globsum,globals,ni)
  port globals[];
  {? count<ni ->
    {; display(count+1,globsum+globals[count],globals,ni ) },
    default ->
    {; stdio:printf(" sumall =%16.10f\n",{globsum},_) }
  }
```
We ran this program on the Intel iPSC/860 and on the IBM SP. The performance results for ProgPiPCN are illustrated below using gauge, an execution profiler for PCN programs. This utility provides many options to analyze the performance of a parallel PCN program. Among these are the profile data for the time spent in each procedure on each node, the number of times each procedure is called, idle times, internode message counts and volumes, and various statistical results based on these profile data. The first graph pertains to ProgPiPCN run on the Intel iPSC/860 with eight nodes.

The graph shows the execution time metric of ProgPiPCN by procedures. Those procedure names with the prefix compi belong to our code, and the other procedures are in the built-in PCN modules sys and stdio. Notice that the time spent by the Fortran procedure compsum is much greater than that of other procedures. Displayed below the graph is the total execution times, the number of reductions, and the number of suspensions. A reduction is one completed execution of a process, and a suspension occurs when a process requires value of an undefined definitional variable. A process suspends until the definitional variable is given a value.

6.2 ProgPdePCN

For the code ProgPdePCN we discuss only the procedure named square, which maps each block to a node in a virtual array topology. The other procedures are similar to those of ProgPiPCN. The complete code is given in the Appendix.
square(max_iter,globals)
port N[nodes()],E[nodes()],globals[];
{|| i over 0 .. isize-1 : :
  {|| j over 0 .. jsize-1 : :
    {me=id(i,j),
     start_block(max_iter,i,j,
                 N[me],N[id(i,j-1)],
                 E[me],E[id(i-1,j)],
                 globals[me])@node(me)
    }
  }
}

start_block(max_iter,i,j,N,S,E,W,global_s)
...

The domain is decomposed into isize horizontal and jsize vertical blocks. Each block is assigned an ID number by the function id and mapped to the member node(me) of the array node. The port arrays N[nodes()] and E[nodes()] are used to communicate data on the ghost points (which form the edge). Notice that the north ghost points of block(id(i, j-1)) are the south ghost points of block(id(i,j)). And the north input of a block is the south output of its north neighbor. The procedures send_edge and receive_edge in the Appendix send and receive data on the edge.

Figures 6–8 give the performance results of ProgPdePCN run on the IBM SP with nine nodes. The first graph shows the execution time by procedures. Notice that the time used by the computational procedure compute is about one hundred times those by the communication procedures get_edge and receive_edge.

The second graph shows the time breakdown by nodes. The gray bars represent idle time while the black ones represent the execution time. Notice that each node spends a considerable amount of time waiting for data from other nodes. To improve performance, one must find ways to reduce this idle time.
Execution Time by Procedures

Figure 6. Performance of ProgPdePCN on the SP

Time Breakdown by Nodes

Figure 7. Time breakdown by nodes
The third graph shows the execution time by procedure and nodes. The time is represented by the color (unfortunately, the color cannot be reproduced in this report) of the square that corresponds to the procedure and node.

![Graph showing execution time by procedure and nodes](image)

**Execution Time by Procedures and Nodes**

**Figure 8. Execution time by procedures and nodes**

### 7 Conclusion

The observations given below are based on our limited experience with the tools, and may even be outdated.

- PCN is a programming language, whereas BlockComm is a library of routines. From a user's point of view, this means that to use PCN, one has to master the language syntax, whereas to use BlockComm/Chameleon, one has to learn how and where to use the BlockComm/Chameleon subroutines to modify a sequential code. The new MPI tool is more like the latter.

- For more complicated applications, BlockComm must be supplemented by Chameleon routines (for parallel I/O, data reduction, broadcasting, etc.).

- Although BlockComm has versions for both Fortran and C, writing a domain decomposition code in C is not as convenient, because C arrays cannot be declared with arbitrary index ranges. Indeed, our original sequential TDGL code was written in C, and we have to convert it to Fortran to take advantage of the BlockComm package.

- The current BlockComm documentation is written for Fortran users, whereas that of Chameleon is for C users. Since we need to use Chameleon routines in our Fortran
program, we have to sometimes guess the Fortran syntax for some Chameleon routine calls. It would be of great help to the users if both Fortran and C documentations for the two packages were available.

- To use PCN to rewrite a sequential code in general involves relatively more effort than to use a message-passing tool.

- Since the compilation technology for PCN is still in its infancy (and so is not as good as that of Fortran or C), a program written entirely in PCN usually do not produce the most efficient code. The approach of *multilingual programming* permits us to take advantage of the unique features of PCN, such as mapping, communication, and scheduling, to complement the proven efficiency of Fortran and C programming for sequential computation [4]. This approach calls for dividing up a sequential program into some convenient parts and converting these pieces to procedures to be called by PCN. A Fortran sequential subroutine can be called from PCN directly, except that the suffix "/" has to be appended to the subroutine name to form the correspond PCN procedure name. In the case of C subroutines, arguments (except arrays) passed to a C procedure from PCN must be declared as pointers in the C procedure.

Acknowledgments

We thank our colleagues who have made their work on the various parallel programming tools available to us and helped us with many of our questions. This list includes Ian Foster, William Gropp, Ewing L. Lusk, and Steve Tuecke. We also thank Dave Levine for sharing with us his version of parallel TDGL code and Paul Plassmann for his parallel GL code; both provided valuable assistance to get us started in learning BlockComm.
APPENDIX: Program Listings

ProgPdePCN: A PCN Program for Problem 3

```c
#include "grid.h"
#define id(i,j) (((i+isize)%isize)+((j+jsize)jsize)*isize)
main(argc,argv,rc)
{
    argv ?[_,maxnum_of_iterations] ->
    sys:string_to_integer(maxnum_of_iterations,max_iter),
    main_body(max_iter) in vts:array(isize*jsize),
    rc=0,
    },
    default ->
    {
        stdio : printf("usage: %s <max_iter>\n",{argv[0]},_),
        rc=1
    }
}
main_body(max_iter)
port globals[nodes()];
{|| square(max_iter,global_s),
    display(0,0,global_s)
} square(max_iter,global_s)
port N[nodes()],E[nodes()],global_s[];
{|| i over 0 .. isize-1 ::
    {|| j over 0 .. jsize-1 ::
        (me=id(i,j)),
        start_block(max_iter,i,j,
            N[me],N[id(i,j-1)],
            E[me],E[id(i-1,j)],
            global_s[me])@node(me)
    }
} start_block(max_iter,i,j,N,S,E,W,global_s)
double square[bsz*bsz],edge[bsz];
{|| N={Ni,Mo},E={Ei,Es},
    S={So,Si}, W={Wo,Wi}
    -> {
        initialize_(i,j,square),
        start_clock(),
        block(max_iter,i,j,square,edge,{Ni,Mo,Ei,Es},
            {No,So,Es,Wo,global_s,0})
    }
} block(max_iter,i,j,square,edge,Is,Os,global_s,count)
double square[],edge[],error;
{;
    send_edge(square,edge,Os,Os1),
    receive_edge(ni,si,ei,wi,Is,Is1),
    compute_(i,j,square,ni,si,ei,wi,error),
    {? count < max_iter ->
    {||
        block(max_iter,i,j,square,edge,Is1,Os1,global_s,count+1)
    }
}
```
default ->
{; stop_clock(),
  global_s=error,
  stdio:printf("done\n",{),_)
}
}
send_edge(square,edge,0s,0s1)
double square[], edge[];
{
  0s?=(N,S,E,W) ->
  {
    ; getedge_(NORTH,square,edge),
    N=([{edge}|N1],
    getedge_(SOUTH,square,edge),
    S=([{edge}|S1],
    getedge_(EAST,square,edge),
    E=([{edge}|E1],
    getedge_(WEST,square,edge),
    W=([{edge}|W1],
    Os1={N1,S1,E1,W1}
  }
}
receive_edge(ni,si,ei,wi,Is,Is1)
{
  Is?={N,S,E,W} ->
  {
    ;
    {
      {? N?=[(n)|N1_tmp] ->{ni=nn,N1=N1_tmp}},
      {? S?=[(s)|S1_tmp] ->{si=ss,S1=S1_tmp}},
      {? W?=[(w)|W1_tmp] ->{wi=ww,W1=W1_tmp}},
    Is1={N1,S1,E1,W1}
  }
}
display(count.globmax.globals)
port globals[];
{
  count<isize*jsize ->
  {
    ;temp_max=globals[count],
    getmax(globmax,temp_max,new_max),
    display(count+1,new_max,globals)
  },
  default ->{
    ;
    stdio:printf("Max_error=\n",{globmax},_),
    stdio:printf("done\n",{),_)
  }
}
getmax(x,y,z)
{
  ? x>y ->z=x,
  default ->z=y
}
#include "grid.h"

subroutine initialize(i,j,block)
  integer i, j
  double precision block(BSIZE,BSIZE)
  integer ii, jj
  do ii=1, BSIZE
do jj = 1, BSIZE
    block(ii, jj) = 0.0
endo
dono return
dono

subroutine compute(i, j, v, ned, sed, eed, wed, errmax)
dono integer i, j, ii, jj
dono double precision v(BSIZE, BSIZE), u(0:BSIZE+1, 0:BSIZE+1)
dono double precision ned(BSIZE), sed(BSIZE)
dono double precision eed(BSIZE), wed(BSIZE)
dono double precision dx, dy, errmax, err, w, a, x(BSIZE), y(BSIZE)
dono errmax = 0.0
dono dx = 1.0/(isize*BSIZE-1.0)
dono dy = dx
dono w = 1.0
dono a = 20.0
do ii = 1, BSIZE
do jj = 1, BSIZE
todo u(ii, jj) = v(ii, jj)
dono enddo
do ii = 1, BSIZE
todo u(0, ii) = wed(ii)
todo u(BSIZE+1, ii) = eed(ii)
todo u(ii, BSIZE+1) = med(ii)
todo u(ii, 0) = sed(ii)
dono enddo
do ii = 1, BSIZE
todo x(ii) = (BSIZE*i+(ii-l))*dx
todo y(ii) = (BSIZE*j+(ii-l))*dy
todo if (i .eq. 0) u(0, ii) = 0.0
todo if (i .eq. isize-l) u(BSIZE+l, ii) = y(ii)**3
todo if (j .eq. 0) u(ii, 0) = 0.0
todo if (j .eq. jsize-l) u(ii, BSIZE+1) = x(ii)
dono enddo
do kk = 1, 20
todo errmax = 0.0
do jj = 1, BSIZE
do ii = 1, BSIZE
todo u(ii, jj) = u(ii, jj) - \left(-u(ii+1, jj)+2*u(ii, jj)-u(ii-1, jj)\right)/dx**2
todo / +(-u(ii, jj+1)+2*u(ii, jj)-u(ii, jj-1))/dy**2
todo / + a*{u(ii, jj)-x(ii)*y(jj)}*(a*y(jj)**2-6))/(4/dx**2+a)
todo err = abs(u(ii, jj)-x(ii)*y(jj)**3)
todo errmax = max(errmax, err)
dono enddo
dono enddo
do ii = 1, BSIZE
do jj = 1, BSIZE
todo v(ii, jj) = u(ii, jj)
dono enddo
dono return
dono end

subroutine getedge(id, block, edge)
dono double precision block(BSIZE, BSIZE), edge(BSIZE)
dono integer i, id

c North face
todo if (id .eq. NORTH) then
do i=1,BSIZE
    edge(i) = block(i,BSIZE)
  enddo
endif

C South face
if (id .eq. SOUTH) then
  do i=1,BSIZE
    edge(i) = block(i,1)
  enddo
endif

C East face
if (id .eq. EAST) then
  do i=1,BSIZE
    edge(i) = block(1,i)
  enddo
endif

C West face
if (id .eq. WEST) then
  do i=1,BSIZE
    edge(i) = block(1,i)
  enddo
endif
return
end
integer function worker()
  double precision errmax, work, dx, dy, w
  integer nx, ny
  parameter (nx=501, ny=501, a=20)
  double precision u((nx+2)*(ny+2)), x(ny+2), y(ny+2)
  double precision v((nx+2)*(ny+2))
  double precision t1, t2, SYGetElapsedTime
  integer pimytid, pgm, myid, nstep
  integer sx, sxgp, ex, exgp, sy, sygp, ey, eygp

  myid = pimytid()
  call indexcomp(nx, ny, sx, ex, sxgp, exgp,
    + sy, ey, sygp, eygp, pgm)
  errmax = 0.0
  w = 1.0
  call InitDomain(u, nx, ny, sx, ex, sxgp, exgp,
    + sy, ey, sygp, eygp)
  call InitDomain(v, nx, ny, sx, ex, sxgp, exgp,
    + sy, ey, sygp, eygp)

  call Plgsync(0)
  t1 = SYGetElapsedTime()

  dx = 1.0d0/(nx-1)
  dy = 1.0d0/(ny-1)

  call bound(u, x, y, nx, ny, dx, dy, sx, sxgp, ex, exgp,
    + sy, ey, sygp, eygp)
  call bound(v, x, y, nx, ny, dx, dy, sx, sxgp, ex, exgp,
    + sy, ey, sygp, eygp)

  c begin iteration
  nstep = 2000
  do 20 iter = 0, nstep - 1, 2
    call BCexec(pgm, u, u)
    call compute(u, v, x, y, nx, ny, dx, dy, errmax, a,
      + sx, sxgp, ex, exgp, sy, sygp, ey, eygp)
    call BCexec(pgm, v, v)
    call compute(v, u, x, y, nx, ny, dx, dy, w, errmax, a,
      + sx, sxgp, ex, exgp, sy, sygp, ey, eygp)
    call Plgmax(errmax, 1, work, 0)
    if (myid .eq. 0) then
      print 30, w, IFIX(iter/2.0), errmax
    endif
  20 continue

  30 format(5x, f8.2, i10, f16.12)

  t2 = SYGetElapsedTime() - t1
  print *, 'Total time = ', t2, ' on ', pimytid()
  call BCfree(pgm)
  worker = 0
  return
end

SUBROUTINE indexcomp(nx, ny, sx, ex, sxgp, exgp, + sy, ey, sygp, eygp, pgm)

  integer pimytid, pinumtids, iper(2)
  include '/home/gropp/tools.n/blkcm/meshf.h'
  integer myid, nproc, nx, ny, nd, NBYES
  integer pgm, sz(0:9, 0:1)
  integer sx, sxgp, ex, exgp, sy, sygp, ey, eygp
  nd = 2
NBYTES=8
sz(szmdim,0) = nx
sz(szisparallel,0) = 1
sz(szndim,0) = -1
sz(szmdim,1) = ny
sz(szisparallel,1) = 1
sz(szndim,1) = -1
call BCFindGhostFromStencil(nd, sz, 0, 0, 1)
myid = pimytid()
nproc = pinumtids()
if(myid .eq. 0) print*, 'nproc=', nproc
call BCGlobalToLocalArray(nd, sz, nproc, myid)
iper(1)=0
iper(2)=0
call BCSetGhostWidths(nd, sz, iper)
pgm = BCBuildArrayPGM(nd, sz, nproc, myid, NBYTES)
call BCArrayCompile(pgm, 0)
sx = sz(szstart,0) + 1
ex = sz(szend,0) + 1
sxgp = sz(szsg,0)
exgp = sz(szeg,0)
sy = sz(szstart,1) + 1
ey = sz(szend,1) + 1
sygp = sz(szsg,1)
eygp = sz(szeg,1)
return
end subroutine InitDomain(u,nx,ny,sx,sxgp,ex,exgp,sy,sygp,ey,eygp)
integer sx,sxgp,ex,exgp,sy,sygp,ey,eygp
double precision u(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
integer i,j,nx,ny
do j = sy-sygp,ey+eygp
  do i = sx-sxgp,ex+exgp
    u(i,j) = 0.0d0
  enddo
enddo
return
end subroutine bound(u,x,y,nx,ny,dx,dy,
+ sx,sxgp,ex,exgp,sy,sygp,ey,eygp)
integer sx,sxgp,ex,exgp,sy,sygp,ey,eygp,i,j
double precision u(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
double precision x(sx:ex)
double precision y(sy:ey)
double precision dx,dy
integer nx,ny
do i=sx,ex
  x(i)=(i-1)*dx
enddo
do j=sy,ey
  y(j)=(j-1)*dy
enddo
c Bottom (sy = 1)
if (sy .eq. 1) then
  do i=sx,ex
    u(i,sy) = 0.0
  enddo
endif
c Top (ey = ny)
if (ey .eq. ny) then
  do i=sx,ex
    u(i,ey) = x(i)
  enddo
endif

Left (sx = 1)
if (sx .eq. 1) then
  do j=sy,ey
    u(sx,j) = 0.0
  enddo
endif

Right (ex = nx)
if (ex .eq. nx) then
  do j=sy,ey
    u(ex,j) = y(j)*y(j)*y(j)
  enddo
endif
return
end

subroutine compute(u,v,x,y,nx,ny,dx,dy,\n+ ssx,ssy,ex,exgp,ssy,ssygp)
integer ssx,ssy,ex,exgp,ssy,ssygp
integer sx,sxgp,ex,exgp,sy,sygp,ey,eygp
integer sxgp,ex,exgp,sy,sygp,ey,eygp
double precision u(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
integer ssx,ssy,ex,exgp,ssy,ssygp
double precision v(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
double precision x(sx:ex)
double precision y(sy:ey)
double precision dx,dy,\n+ ssx,ssy,ex,exgp,ssy,ssygp,i,j,nx,ny

ssx=sx
ex=ex
ey=ey
ssy=sy

if(sx .eq. 1) ssx=2
if(sy .eq. 1) ssy=2
if(ex .eq. nx) ex=ex-1
if(ey .eq. ny) ey=ey-1
errmax=0.0

do 15 j=ssy,ey
  do 15 i=ssx,ex
    v(i,j)=u(i,j)-w*(-u(i+1,j)+2*u(i,j)-u(i-1,j))/dx**2
    / + (-u(i,j+1)+2*u(i,j)-u(i,j-1))/dy**2
    / + a*u(i,j)-x(i)*y(j)**3)/4/dx**2+a)
    err=abs(v(i,j)-x(i)*y(j)**3)
    errmax=max(errmax,err)
  15 continue
return
end
integer function worker()

INTEGER sx,ex,sy,ey,sxgp,exgp,sygp,eygp
INTEGER nproc,myid,pimytid,pgm, sz(0:9,0:1)
INTEGER nx,ny,nd,np,ns,nsx(2),nsy(2),nxm,nym,count
parameter(nx = 52, ny = 52, nd=2)
double precision a(nx*ny),b(nx*ny)
double precision da(nx*ny),db(nx*ny)
double precision p1(nx*ny),p2(nx*ny)
double precision dp1(nx*ny),dp2(nx*ny)
double precision hh(nx*ny),seed(2)
double precision time,SYGetElapsedTime
double precision dx,dt0,dx0,dx,dt,t,dy,ty,tp,h,rk
double precision dx2,rkx,rk2,dy2

myid=pimytid()
CALL getindex(nx,ny,nd,sz,sx,ex,sy,ey,sxgp,exgp,sygp,eygp,pgm)

CALL checkindex(sz,sx,ex,sxgp,exgp,sy,ey,sygp,eygp, +
               nx,ny,myid,nproc)
if (myid.eq.0) print*, 'Reading parameters.'

CALL main_input(rk,h,tp,nx,ny,np,nr,ns, +
               dx2,dy2,rk2,rkx,ry,dy,txm,nym, +
               dt0,dt,dy,nsx,nsy,seed,myid)
if (myid.eq.0) print*, 'Initializing.'

CALL initialize(p1,p2,a,b,h,dx, +
x,ex,sxgp,exgp,sy,ey,sygp,eygp, +
               nx,nsx,seed,ns,myid,nx,ny)

  t=0
  count=0
  dt=0

   ************** Main loop **************
10 IF (t.lt.tp) THEN

    CALL bound(p1,p2,a,b,h,rk,nx,ny,dx,dy,nxm,nym, +
               sx,ex,sxgp,exgp,sy,ey,sygp,eygp,rkx,rky)

    CALL compf(p1,p2,a,b,da,db,dp1,dp2,dx0, +
x,ny,dx,dy,nxm,nym,dx2,dy2,rkx,ry,rk2,rk,h,dt,count, +
x,ex,sxgp,exgp,sy,ey,sygp,eygp,pgm)

    if ((MOD(count,np).eq.0)) then
        CALL compsum(p1,p2,a,b,h,myid,count,pgm, +
                    dx,dx2,dy,dy2,rkx,ry,rk2,nx,ny,nxm,nym,h,rk,t, +
                    sx,ex,sxgp,exgp,sy,ey,sygp,eygp)
    endif

    dt=min(tp-t,dt0)
    t=t+dt
    count=count+1
GO TO 10
ENDIF
c End of main loop
   if(myid .eq. 0) then
      time=SYGetElapsedTime()
      print*, ' Elapsed Time : ', time
      print*, ' Average Time : ', time/count
   endif
   worker=0
   RETURN
END

SUBROUTINE getindex(nx,ny,nd,sz,sx,ex,sy,ey,
   +sxgp,sygp,exgp,eygp,pgm)
   integer pimytid, pinumtids,iper(2)
   include '/home/gropp/tools.n/blkcm/meshf.h'
   integer myid, nproc,nx,ny,nd,NBYTES
   integer pgm, sz(0:9,0:1)
   integer sx,sxgp,ex,exgp,sy,sygp,ey,eygp
   NBYTES=8
   sz(szmdim,0) = nx
   sz(szisparallel,0) = 1
   sz(szmdim,0) = -1
   sz(szmdim,1) = ny
   sz(szisparallel,1) = 1
   sz(szmdim,1) = -1
   call BCFindGhostFromStencil( nd, sz, 0, 0,1)
   myid = pimytid()
   nproc = pinumtids()
   if(myid .eq. 0) print*, 'nproc=', nproc
call BCGlobalToLocalArray( nd, sz, nproc, myid )
   iper(1)=0
   iper(2)=0
call BCSetGhostWidths(nd,sz,iper)
   pgm = BCBuildArrayPGM( nd, sz, nproc, myid, NBYTES )
   call BCArrayCompile( pgm, 0 )
   sx = sz(szstart,0) + 1
   ex = sz(szend,0) + 1
   sxgp = sz(szsg,0)
   exgp = sz(szeg,0)
   sy = sz(szstart,1) + 1
   ey = sz(szend,1) + 1
   sygp = sz(szsg,1)
   eygp = sz(szeg,1)
   return
end

#include "tools.h"
#include "comm/comm.h"
#include <stdio.h>
#include "blkcm/bc.h"
#include "blkcm/mesh.h"
#include "comm/io/pio.h"
#ifdef rs6000
#define checkindex_ checkindex
#endif

void checkindex_(size,sx,ex,sxgp,exgp,sy,ey,sygp,eygp,
   nx,ny,myid,nproc)

BCArrayPart size[10];
int *sx, *ex, *sxgp, *exgp;
int *sy, *ey, *sygp, *eygp;
int *nx, *ny;
int *myid, *nproc;
{

FILE *pw;
static char filename[] = "blk_rep";
int i, lx, ly;
int glx, gly; /*dimension of blocks with ghosts*/
if (*myid == 0) {
    printf( "Writing report
");
    if ((pw = fopen(filename, "w")) == NULL) {
        printf( "cannot open %s
", filename);
        exit(0);
    }
    fprintf(pw, " Decomposition Report\n");
    fprintf(pw, "**************************\n");
    fprintf(pw, "Total processors : Xd
", *nproc);
    fprintf(pw, "Global size (x,y) : */.d 7.d
", *nx, *ny);
    fprintf(pw, "Block Decomposition : ");
    fprintf(pw, "Processor Distribution (x, y) : 7,d 7,d
", *nx, *ny);
    fprintf(pw, "node\tblock size\tblock endpoints\tblock w/ghosts points
");
    for (i=1;i<=*nproc;i++) {
        if (GTOKEN(0,i)) {
            pw = fopen(filename, "a")
", *myid, lx, ly, *sx, *ex, *sy, *ey);
            /* fprintf(pw,"done\n");*/
            fclose(pw);
        }
    }
}

lx = *ex-*sx+1;
ly = *ey-*sy+1;

SUBROUTINE main_input(rk,h,tp,nx,ny,np,nr,ns,
+ dx2,dy2,rk2,rx,rk,dy,rxm,ny,
+ dt0,dx,dy,nsx,nsy,seed,myid)
integer isz,msg_int,msg dbl,all,dsz
parameter(isz=4,msg_int=4,all=0)
parameter(dsz=8,msg dbl=4)
real*8 dx,dt0,dx,dy,cf1,ylength,xlength
real*8 rk2,dy2,rk,rxm,rx,dk2,dp
integer np,ns,nsx(2),nsy(2)
integer i, nx,ny,myid,nxm,nym
double precision seed(2)
CHARACTER*79 discrp
if (myid.eq.0) then
    OPEN(unit=9,file='defaults',
+ status='old')
    REWIND 9
READ (9,25) discrp
}
c The input file is read by processor 0 and then the data is
c scattered to the other processors
READ (9,*) rk
READ (9,25) discrp
READ (9,*) h
READ (9,25) discrp
READ (9,*) tp
READ (9,25) discrp
READ (9,*) xlength
READ (9,25) discrp
READ (9,*) ylength
READ (9,25) discrp
READ (9,*) np
READ (9,25) discrp
READ (9,*) nr
READ (9,25) discrp
READ (9,*) cfl
READ (9,25) discrp
READ (9,*) ns

DO i=1,ns
  READ (9,25) discrp
  READ (9,*) nsx(i),nsy(i),seed(i)
END DO
CLOSE(9)

FORMAT(A72)

dx = xlength/(nx-2)
dy = ylength/(ny-2)
dxy = dx*dy
dx2 = dx*dx
dy2 = dy*dy
rk2 = rk*rk
rxx = rk*dx
rxy = rk*dy
dt0 = rk*cfl/max(1./dx2/rk2+1./dy2/rk2+(h*xlength)**2,
  + 1./dx2+1./dy2+1.)
nxm = nx-1
nym = ny-1
ENDIF

C scatter the data
CALL PIbcastSrc(np,isz,0,all,msg_int)
CALL PIbcastSrc(nr,isz,0,all,msg_int)
CALL PIbcastSrc(ns,isz,0,all,msg_int)
CALL PIbcastSrc(nxm,isz,0,all,msg_int)
CALL PIbcastSrc(nym,isz,0,all,msg_int)
CALL PIbcastSrc(h,dsz,0,all,msg_dbl)
CALL PIbcastSrc(dt0,dsz,0,all,msg_dbl)
CALL PIbcastSrc(tp,dsz,0,all,msg_dbl)
CALL PIbcastSrc(dx,dsz,0,all,msg_dbl)
CALL PIbcastSrc(dy,dsz,0,all,msg_dbl)
CALL PIbcastSrc(dx2,dsz,0,all,msg_int)
CALL PIbcastSrc(dy2,dsz,0,all,msg_int)
CALL PIbcastSrc(rk,dsz,0,all,msg_int)
CALL PIbcastSrc(rxx,dsz,0,all,msg_dbl)
CALL PIbcastSrc(rxy,dsz,0,all,msg_dbl)
CALL PIbcastSrc(rk2,dsz,0,all,msg_dbl)
CALL PIbcastSrc(dxy,dsz,0,all,msg_dbl)

DO i=1,ns
  CALL PIbcastSrc(nsx(i),isz,0,all,msg_int)
  CALL PIbcastSrc(nsy(i),isz,0,all,msg_int)
  CALL PIbcastSrc(seed(i),dsz,0,all,msg_dbl)
ENDDO
RETURN
END
SUBROUTINE initialize(p1,p2,a,b,h,dx,
+sx,ex,sxgp,exgp,sy,ey,sygp,eygp,
+nsx,nsy,seed,ns,myid,nx,ny)

INTEGER sx,ex,sy,ey,sxgp,exgp,sygp,eygp,ns
INTEGER nsx(ns),nsy(ns)
DOUBLE PRECISION seed(ns),dx,h
DOUBLE PRECISION p1(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
DOUBLE PRECISION p2(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
DOUBLE PRECISION a(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
DOUBLE PRECISION b(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
INTEGER ix,iy,myid,nx,ny

DO iy = sy-sygp, ey+eygp
  DO ix = sx-sxgp, ex+exgp
    p1(ix,iy)=0
    p2(ix,iy)=0
    a(ix,iy)=0
    b(ix,iy)=(ix-1)*dx*h
  END DO
END DO
CALL reinit(p1,sx,ex,sy,ey,sxgp,exgp,sygp,eygp,
+nsx,nsy,seed,ns,myid)
RETURN
END

SUBROUTINE reinit(p1,sx,ex,sy,ey,sxgp,exgp,
+sygp,eygp,nsx,nsy,seed,ns,myid)
INTEGER ns,myid
INTEGER sx,ex,sy,ey,sxgp,eygp,exgp,sygp
DOUBLE PRECISION pi(sx-sxgp:ex+exgp,sy-sygp:ey+eygp),seed(ns)
INTEGER nsx(ns),nsy(ns),i,ix,iy

DO i = 1,ns
  DO ix = sx,ex
    IF ((nsx(i) .ge. sx) .and. (nsx(i) .le. ex)) then
      DO iy = sy,ey
        IF ((nsy(i) .ge. sy) .and. (nsy(i) .le. ey)) then
          p1(nsx(i),nsy(i))=seed(i)
        ENDIF
      ENDDO
    ENDIF
  ENDDO
ENDDO
RETURN
END

SUBROUTINE bound(p1,p2,a,b,h,k,nx,ny,dx,dy,nxm,nym,
+sx,ex,sxgp,exgp,sy,ey,sygp,eygp,kx,ky)

INTEGER nx,ny,i,j,nxm,nym
INTEGER sx,ex,sy,ey
INTEGER ssx,ssy,eex,eey
INTEGER sxgp,exgp,sygp,eygp
DOUBLE PRECISION p1(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
DOUBLE PRECISION p2(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
DOUBLE PRECISION a(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
DOUBLE PRECISION b(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
DOUBLE PRECISION dx,dy,kx,ky,h,k
ssx=sx
ssy=sy
eex=ex
eey=ey
if (ex.eq.nx) eex=nxm
if (ey.eq.ny) eey=nym

32
if (sy .eq. 1) then
  do i=ssx,eex
    pl(i,1)=pl(i,2)*cos(b(i,1)*ky)
     +p2(i,2)*sin(b(i,1)*ky)
    p2(i,1)=p2(i,2)*cos(b(i,1)*ky)
     -p1(i,2)*sin(b(i,1)*ky)
    a(i,1)=a(i,2)+(h-(b(i+1,1)-b(i,1))/dx)*dy
  enddo
endif

if(sy .eq. ny) then
  do i=ssx,eex
    pl(i,ny)=pl(i,nym)*cos(b(i,nym)*ky)
     -p2(i,nym)*sin(b(i,nym)*ky)
    p2(i,ny)=p2(i,nym)*cos(b(i,nym)*ky)
     +pl(i,nym)*sin(b(i,nym)*ky)
    a(i,ny)=a(i,nym)-(h-(b(i+1,nym)-b(i,nym))/dx)*dy)
  enddo
endif

if (sx .eq. 1) then
  do j=ssy,eey
    pl(1,j)=pl(2,j)*cos(a(l,j)*kx)
     +p2(2,j)*sin(a(l,j)*kx)
    p2(1,j)=p2(2,j)*cos(a(l,j)*kx)
     -pl(2,j)*sin(a(l,j)*kx)
    b(l,j)=b(2,j)-(h+(a(l,j+1)-a(l,j))/dy)*dx)
  enddo
endif

if (ex .eq. nx) then
  do j=ssy,eey
    pl(nx,j)=pl(nxm,j)*cos(a(nxm,j)*kx)
     -p2(nxm,j)*sin(a(nxm,j)*kx)
    p2(nx,j)=p2(nxm,j)*cos(a(nxm,j)*kx)
     +pl(nxm,j)*sin(a(nxm,j)*kx)
    b(nx,j)=b(nxm,j)+(h+(a(nxm,j+1)-a(nxm,j))/dy)*dx)
  enddo
endif

RETURN
END

SUBROUTINE compf(phi,ph2,al,a2,fgl,fg2,hgl,hg2,dxy,
              +nx,ny,dx,dy,nxm,nym,dx2,dy2,rxr,ryr,rx2,ry2,h,dt,count,
              +sx,ex,sxgp,exgp,sy,ey,sygp,eygp,pgm)

INTEGER sx,ex,sy,as,sxgp,exgp,sygp,eygp
INTEGER nx,ny,count,i,j,nxm,nym
double precision ph1(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
double precision ph2 (sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
double precision a1(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
double precision a2(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
double precision fgl(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
double precision fg2(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
double precision hgl(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
double precision hg2(sx-sxgp:ex+exgp,sy-sygp:ey+eygp)
double precision wm(60,60)
double precision dx,dy,dx2,dy2,c10,c11,c20,c21,c10,c21
double precision rk2,rkx,rky,dxy
ssy=ssx
ssx=sx
eex=ex
ey=ey
call BCexec(pgm,ph1,ph1)
call BCexec(pgm,ph2,ph2)
call BCexec(pgm,al,al)
call BCexec(pgm,a2,a2)
if (sy .eq. 1) ssy=2
if (sx .eq. 1) ssx=2
if (ey .eq. ny) eey=ny
if (ex .eq. nx) eex=nx
if (j.eq.2) then
  fgl(i,j)= (al(i,j+l)-al(i.j))/dy2 + h/dy
  /*a2(i,j)+a2(i+1,j))/dx2
  /*((phl(i,j)*ph2(i,j)+phl(i,j)*ph2(i,j)+phl(i,j)*ph2(i,j))/s11)/rkx
else if (j.eq.nym) then
  fgl(i,j)= (al(i,j+l)+al(i,j+l))/dy2 - h/dy
  /*a2(i,j)+a2(i+1,j))/dx2
  /*((phl(i,j)+ph2(i+1,j)*phl(i,j)+ph2(i,j)*ph2(i,j)+ph2(i,j))/s11)/rkx
else
  fgl(i,j)= (al(i,j+l)-2.*al(i,j)+al(i,j))/dy2
  /*a2(i,j)+a2(i+1,j))/dx2
  /*((phl(i,j)*ph2(i+1,j)-phl(i,j)*ph2(i+1,j)+phl(i,j)*ph2(i+1,j))/s11)/rkx
end if
if (i.eq.2) then
  lg2(i,j)= (a2(i+1,j)-a2(i,j))/dx2 - h/dx
  /*a(i,j)-al(i,j))/dy2
  /*((phl(i,j)-ph2(i,j)+phl(i,j)-ph2(i,j))/s11)/rkx
else
  lg2(i,j)= (a2(i+1,j)-a2(i,j))/dx2 - h/dx
  /*a(i,j)-al(i,j))/dy2
  /*((phl(i,j)-ph2(i,j)+phl(i,j)-ph2(i,j))/s11)/rkx
end if
SUBROUTINE compsum(pi, p2, a, b, hh, myid, count, pgm, + dx, dx2, dy, dy2, kx, ky, k2, nx, ny, nxm, nym, h, k, t, + sx, ex, sxgp, exgp, sy, ey, sygp, eygp)

INTEGER sx, ex, sy, ey, i, j, pgm
INTEGER sxgp, exgp, sygp, eygp
INTEGER myid, count, nx, ny, ssx, ssy, exx, eyx
double precision pi (sx-sxgp:ex+exgp, sy-sygp:ey+eygp)
double precision p2 (sx-sxgp:ex+exgp, sy-sygp:ey+eygp)
double precision a (sx-sxgp:ex+exgp, sy-sygp:ey+eygp)
double precision b (sx-sxgp:ex+exgp, sy-sygp:ey+eygp)
double precision hh (sx-sxgp:ex+exgp, sy-sygp:ey+eygp)
double precision h, k, t
double precision sum, p2max, work
double precision dx, dy, dx2, dy2, kx, ky, p2m
double precision c1, c2, s2, s

ssy = sy
ssx = sx
eex = ex
eey = ey

call BCexec(pgm, p1, p1)
call BCexec(pgm, p2, p2)
call BCexec(pgm, a, a)
call BCexec(pgm, b, b)

if (sy .eq. 1) ssy = 2
if (sx .eq. 1) ssx = 2
if (ey .eq. ny) eey = nym
if (ex .eq. nx) eex = nxm
sum = 0
p2max = 0

do j = ssy, eey
  do i = ssx, eex
    ph1(i, j) = ph1(i, j) + dt * hg1(i, j)
    ph2(i, j) = ph2(i, j) + dt * hg2(i, j)
    if (i .lt. nxm) a1(i, j) = a1(i, j) + dt * fg1(i, j) * dx
    if (j .lt. nym) a2(i, j) = a2(i, j) + dt * fg2(i, j) * dy
  enddo
endo
RETURN
END
/ -(a(i,j+1)-a(i,j))/(dy)
c1 = cos(a(i,j)*kx)
s1 = sin(a(i,j)*kx)
c2 = cos(b(i,j)*ky)
s2 = sin(b(i,j)*ky)
s=(((pl(i+l,j)-(c1*pl(i,j)-s1*p2(i,j)))**2
/ +(p2(i+l,j)-(c1*p2(i,j)+s1*pl(i,j)))**2)/dx2
/ +((pl(i,j+1)-(c2*pl(i,j)-s2*p2(i,j)))**2
/ +(p2(i,j+1)-(c2*p2(i,j)+s2*pl(i,j)))**2)/dy2)/k2
/ - p2m + 0.5*p2m**2
sum= sum+s+(hh(i,j)-h)**2
end do
end do
sum=sum*dy*dx
p2max=sqrt(p2max)
call PIgdsun(sum,1,work,0)
call PIgdsnax(p2max,1,work,0)
if (myid .eq. 0) then
   write(6,991) t,p2max,sum
991 format('t = ',f10.6,',  max(phi) = ',f12.5,/
   ',  energy = ',f16.10)
endif
RETURN
END
References


Distribution for ANL-95/49

Internal:

J. M. Beumer (5)
F. Y. Fradin
M. K. Kwong (10)
G. W. Pieper
R. L. Stevens
C. L. Wilkinson
TIS File

External:

DOE-OSTI, for distribution per UC-405 (52)
ANL-E Library
ANL-W Library
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
  F. Berman, University of California at LaJolla
  G. Cybenko, Dartmouth College
  T. DuPont, The University of Chicago
  J. G. Glimm, State University of New York at Stony Brook
  M. T. Heath, University of Illinois, Urbana
  E. F. Infante, University of Minnesota
  K. Kunen, University of Wisconsin at Madison
  R. E. O'Malley, University of Washington
  L. R. Petzold, University of Minnesota
  E. Coskun, Karadeniz Technical University (10)
D. Nelson, DOE - Office of Computational and Technology Research
F. Howes, DOE - Office of Computational and Technology Research