SANDIA REPORT

SAND2012-2437
Unlimited Release
Printed April 2012

MiniGhost: A Miniapp for Exploring Boundary Exchange Strategies Using Stencil Computations in Scientific Parallel Computing

Richard F. Barrett, Courtenay T. Vaughan, and Michael A. Heroux

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.osti.gov/bridge

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2012-2437
Unlimited Release
Printed April 2012

MiniGhost: A Miniapp for Exploring Boundary Exchange Strategies Using Stencil Computations in Scientific Parallel Computing

Richard F. Barrett, Courtenay T. Vaughan, and Michael A. Heroux
Computer Science Research Institute
Sandia National Laboratories
P.O. Box 5800 Albuquerque, NM 87175
{rfbarre,ctvaugh,maherou}@sandia.gov

Abstract

A broad range of scientific computation involves the use of difference stencils. In a parallel computing environment, this computation is typically implemented by decomposing the spacial domain, inducing a "halo exchange" of process-owned boundary data. This approach adheres to the Bulk Synchronous Parallel (BSP) model. Because commonly available architectures provide strong inter-node bandwidth relative to latency costs, many codes "bulk up" these messages by aggregating data into a message as a means of reducing the number of messages. A renewed focus on non-traditional architectures and architecture features provides new opportunities for exploring alternatives to this programming approach.

In this report we describe miniGhost, a "miniapp" designed for exploration of the capabilities of current as well as emerging and future architectures within the context of these sorts of applications. MiniGhost joins the suite of miniapps developed as part of the Mantevo project, http://mantevo.org.

Acknowledgment

The authors thank the DOE ASC program for funding this research.

Contents

Figures

Tables

Executive Summary

A broad range of scientific computation involves the use of difference stencils. In a parallel computing environment, this computation is typically implemented by decomposing the spacial domain, inducing a "halo exchange" of process-owned boundary data. This approach adheres to the Bulk Synchronous Parallel (BSP) model. Because commonly available architectures provide strong inter-node bandwidth relative to latency costs, many codes "bulk up" these messages by aggregating data into a message as a means of reducing the number of messages. A renewed focus on non-traditional architectures and architecture features provides new opportunities for exploring alternatives to this programming approach.

In this report we describe miniGhost, a "miniapp" designed for exploration of the capabilities of current as well as emerging and future architectures within the context of these sorts of applications. MiniGhost joins the suite of miniapps developed as part of the Mantevo project, `http://mantevo.org`.

Our work is motivated by recent experiences with new node interconnect architectures, including that used by Cielo [21, 26, 11].

Various experiments involving miniGhost have already been completed, supporting the value of miniGhost as a proxy for represented applications. Additional experiments have been defined, with implementations underway. Future reports will describe the outcome of those experiments, and will also include the definition and use of a performance model incorporating many issues defined by the DOE exascale codesign efforts [22, 3, 14].

This document is released in support of the initial release of miniGhost. As miniapps are intended to be modified, additional supporting documentation is expected to be produced.

1 Introduction

A broad range of physical phenomena in science and engineering can be described mathematically using partial differential equations. Determining the solution of these equations on computers is commonly accomplished by mapping the continuous equation to a discrete representation. One such solution technique is the finite differencing method, which lets us solve the equation using a difference stencil, updating the grid as a function of each point and its neighbors, presuming some discrete time step. The algorithmic structure of the finite difference method maps naturally to the parallel processing architecture and single-program multiple-data (SPMD) programming model. For example, on a regular, structured grid, $O(n^2)$ computation is performed, with nearest neighbor $O(n)$ inter-process communication requirements.

On parallel processing architectures, these sorts of computations require data from neighboring processes. Inter-process communication is typically abstracted into some sort of functionality that may be loosely described as *boundary exchange* (likewise also called ghost-exchange or halo-exchange, *other?*) This notion of mapping a continuous problem to discrete space and the inter-process communication requirement induced by spacially decomposing the grid across parallel processes is illustrated in illustrated in Figure 1.

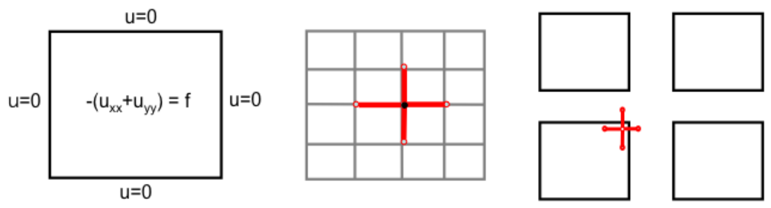


Figure 1. Stencil inter-process communication requirement

The left figure show a partial differential equation (the Poisson equation) described on a continuous domain, with homogeneous Dirichlet boundary conditions. The discretization of this problem is shown in the figure on the middle. The right figure illustrates the inter-process communication requirements when the discretized domain is decomposed across four parallel processes.

This approach adheres to the bulk-synchronous parallel programming model (BSP [25]), arguably the dominant model for implementing high performance portable parallel processing scientific applications. As widely available parallel processing architectures focused node interconnect performance on bandwidth (relative to latency), code developers often aggregated data from various structures into single messages [4]. Although many such applications have continued to perform well even up to peta-scale [2, 6], the situation appears to be changing with the push to exascale [22, 1].

In the BSP/message aggregation (BSPMA) model, data from multiple (logical) memory locations are combined into a user-managed array with other data, then subsequently transmitted to the target process. This step incurs three additional costs, none of which directly advances the computation: memory utilization (the message buffers), on-node bandwidth (copies into the buffer), and synchronization (leading up to and including the data transfer). Further, this model interferes in some sense with the natural mapping of algorithms to programming languages in that the code developer must organize computation with the intent to aggregate and exchange data as a means of maximizing bandwidth and avoiding latency rather than organizing computation in a manner natural to the algorithm.

A renewed focus on non-traditional architectures and architecture features provides new opportunities for exploring alternatives to this programming approach. In previous work [26, 21] we saw that codes configured for the BSPMA model realized an evolutionary improvement in performance. However, codes that sent a large number of small messages realized a significant improvement in performance. This is attributable to the significantly increased message injection rate of the node interconnect and supported by the node architecture, a trend we see continuing as nodes become more powerful and complex.

In order to study the performance characteristics of the BSPMA configuration within the context of computations widely used across a variety of scientific algorithms, we have developed a "miniapp", called miniGhost. As a miniapplication [15], miniGhost is designed for modification and experimentation. It is an open source, self-contained, stand-alone code, with a simple build and execution system. It creates an application-specific context for experimentation, allowing investigation of different programming models and mechanisms, existing, emerging, and future architectures, and enabling investigation of entirely new algorithmic approaches for achieving effective use of the computing environment within the context of complex application requirements.

We begin with a discussion of difference stencils, followed by a description of the miniGhost miniapp, focusing on its computation and communication requirements. We include a discussion of the implementation, with a description of some MPI semantic options for implementing the boundary exchange. We then describe CTH, an application code for which miniGhost is intended to serve as a proxy, including a listing of the intentional differences between miniGhost and the implementation of CTH. Next we present some runtime results that serve to support the claim that intended connection, and conclude with a summary of this initial work and a discussion of future work.

2 miniGhost

Stencil computations form the basis for finite difference, finite volume, and in fact many other algorithms. The basic idea is to update a value as an average of that value and some set of neighboring points. In the simplest case, heat diffusing across a homogeneous two dimensional domain is simply the non-weighted averages of the points surrounding the point to be updated. This can be described using a 5-point stencil defined as

$$u_{i,j,k}^{t+1} = \frac{u_{i,j-1,k}^{t} + u_{i-1,j,k}^{t} + u_{i,j,k}^{t} + u_{i+1,j,k}^{t} + u_{i,j+1,k}^{t}}{5}, \text{for} \quad i,j,k = 1, \ldots n, \quad \text{for} \quad \text{timestep} \quad t.$$

A 9-point stencil would include the (up to four) points diagonally adjacent to

$$u_{i,j,k}^{t} : u_{i-1,j-1,k}^{t}, u_{i-1,j+1,k}^{t}, u_{i+1,j-1,k}^{t}, \quad \text{and} \quad u_{i+1,j+1,k}^{t}.$$

A three dimensional domain might need to include neighbors in adjacent two dimensional "slices", creating 7-point (analogous to 5-point) stencil or 27-point (analogous to a 9-point) stencil.

A Fortran implementation of the 5-point stencil shown in Figure 2 with the notion of decom-

```
REAL, DIMENSION(0:NX+1,0:NY+1,0:NZ+1) ::  X, Y

DO K = 1,NZ
   DO J = 1,NY
      DO I = 1,NX
         X(I,J,K) =                                  &
                        ( Y(I-1,J,K)+                &
            Y(I,J-1,K) + Y(I,J,K) + Y(I,J+1,K) +     &
                          Y(I+1,J,K) )               &
                  * FIFTH
      END DO
   END DO
END DO
```

Figure 2. Five point stencil in Fortran

This code segment implements a five point differencing scheme on a three dimensional ($\texttt{NX} \times \texttt{NY} \times \texttt{NZ}$*) grid. Note the extra (ghost) space allocated for the boundary condition.*

posing across parallel processes illustrated above in Figure 1.

This problem definition presumes regular, equally spaced grid points across the global domain. This greatly simplifies the implementation of the algorithm, allowing us to focus in on the performance aspects of interest in our experiments.

2.1 Implementations

The basis of miniGhost is the BSPMA implementation described above. Since miniGhost was originally developed to explore alternative message passing implementations, we include two variants, each also using MPI for parallelization. The three options are:

Bulk synchronous parallel with message aggregation (BSPMA) This version accumulates face data, from all variables as they are computed, into message buffers, one per neighbor. Thus six messages are transmitted, one to each neighbor, each time step. (This implementation is illustrated in Figure 3.)

Single variable, aggregated face data (SVAF) This version transmits data as soon as computation on a variable is completed, face data aggregated. Thus six messages are transmitted for each variable (up to 40), one to each neighbor, each time step. (Looking at Figure 3, this eliminates the inner `END DO` and `DO I = 1, NUM_VARS`.)

Single variable, contiguous pieces, v1 (SVCP) A modification of SVAF, with four of the six faces eliminating intermediate buffering in exchange for a significant increasing in the number of messages. That is, the $x-y$ faces are transmitted as they are found contiguously in memory (thus including ghost space), the $x-z$ faces are transmitted one message per contiguous column, and the rows of the $y-z$ faces are aggregated into individual messages. Thus $2 + (2 \times ny) + (2 \times nz)$ messages are transmitted for each variable (up to 40), one to each neighbor, each time step. whereby columns of data are transmitted as soon as the computation(s) on them are completed.

Skeleton app Although not an "official" implementation, by selecting the "no stencil" option (see Table 1), miniGhost runs in pure communication mode, based on the above configurations.

The abbreviations are used for procedure and file naming, described below.

Optionally, summation of the (grid) elements for each variable may be computed, injecting collective communication into the execution. The MPI collective `MPI_ALLREDUCE` forms the global value, adding a runtime stress point typically seen in codes of this sort.

2.2 Execution and Verifying Correctness

MiniGhost is not configured to solve any particular problem, allowing the user to control running time, by setting the number of time steps executed. The `GRID` arrays are loaded with random values (using the Fortran subroutine `RANDOM_NUMBER`). Because homogeneous Dirichlet boundary conditions are used, the grid values will eventually become zeros, so randomly generated source terms (called *spikes*) can be applied in order to maintain non-zero computation. Each spike will induce the requested number of time steps to be performed. That is, if 10 spikes and 50 times steps are requested, each spike will be inserted every 50 time steps, resulting in $50 \times 10 = 500$ total time steps.

Runtime input parameters are listed in Table 1.

A special problem is configured in order to verify correctness. Setting the runtime parameter `check_diffusion` to `1` (the default) initializes the `GRID` arrays to 0, inserts a source term in the

Parameter	*Description*	*Options*
`--scaling`	Parallel scaling configuration: weak or strong	`SCALING_STRONG` `SCALING_WEAK`*

`--nx`
`--ny`
`--nz`
	Local grid dimension in (x, y, z) directions.	> 0; 10*

`--num_vars`	Number of `GRID` arrays operated on.	1 – 40*
`--percent_sum`	(Approximate) percentage of variables summation reduced	0-100; 0*
`--num_spikes`	Number of source spikes inserted.	> 0; 1*

`--npx`
`--npy`
`--npz`
	Logical processor grid in (x, y, z) directions.	> 0; 1*

`--num_tsteps`	Number of time steps iterated.	> 0; 10*

`--stencil`	Stencil to be applied	`STENCIL_NONE`
		`STENCIL_2D5PT`
		`STENCIL_2D9PT`
		`STENCIL_3D7PT`
		`STENCIL_3D27PT`*

`--comm_method`	Boundary exchange implementation.	`COMM_METHOD_BSPMA`*
		`COMM_METHOD_SVAF`
		`COMM_METHOD_SVCP`

`--send_protocol`	Blocking or non-blocking communication	`SEND_PROTOCOL_BLOCKING`* `SEND_PROTOCOL_NONBLOCKING`
`--check_diffusion`	Ensure correctness of computation.	0 or 1*
`--profiling`	To gather runtime performance data.	0 or 1*

Table 1. Input parameters

**default setting; See* `MG_OPTIONS.F` *for list of all parameterized options.*

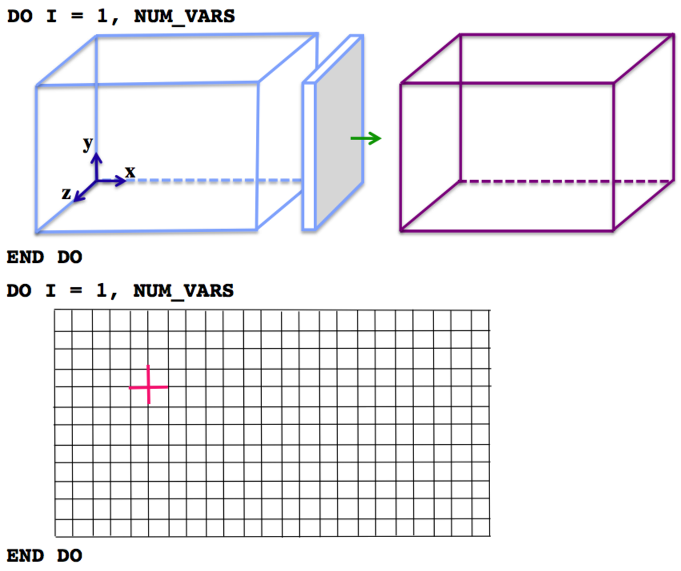


Figure 3. CTH boundary exchange and computation

middle of the global domains, and then tracks the sources as they propagates throughout the arrays. That is, the sum of each `GRID` array should be equal to the inserted source term (within some specified tolerance). The current implementation uses a scaled error check:

$$\frac{\mathtt{GRIDSUM}_{i+1}(j) - \mathtt{GRIDSUM}_i(j)}{\mathtt{GRIDSUM}_{i+1}(j)} < \mathtt{TOL}, \text{for } \ i = 1, \ldots, \frac{\mathtt{NUM\_TSTEPS}}{2}, j = 1, \ldots, \mathtt{NUM\_VARS}.$$

Error tolerance is currently set to 10^{-4}. Note that all variables are checked, each requiring a global summation, which significantly impacts execution time.

2.3 Output

Information describing the problem set is written to a file named `result.txt`. If command line setting `profiling` is set to "1", performance data is included.

Future plans call for a YAML formatting[1].

2.4 Code description

MiniGhost is (mostly) implemented using the Fortran programming language[2], requiring at least a Fortran 90 compliant compiler. Parallelism (further described in Section 2.5) is enabled using

[1]http://yaml.org

[2]The main program is configured using the C programming language, which enables more flexible parsing of command line input.

functionality defined by the MPI 1.2 specification [23]. Each variable (representing for example a material state) is stored in a distinct three dimensional Fortran array (named `GRIDx`, for `x` currently $1, \ldots, 40$), across which the stencil is computed using a triply nested `DO` loop. Type precision is configurable as either single (four bytes) or double (eight bytes, the default), managed in module `MG_CONSTANTS`. Pre-processor compiler directives manage the interface with MPI functionality. That is, `MG_MPI_REAL` is set to `MPI_REAL4` or `MPI_REAL8`, depending on the precision requested. Most other variables are declared using the default `INTEGER` or `REAL`, unless otherwise required or recommended. For example, timing is obtained using `MPI_Wtime`, which is defined in MPI as double precision. Accumulation of profiling data could require increased precision, so eight byte integers are employed.

Figure 4 shows the runtime flow. For the most part, the names refer to the subroutine as well

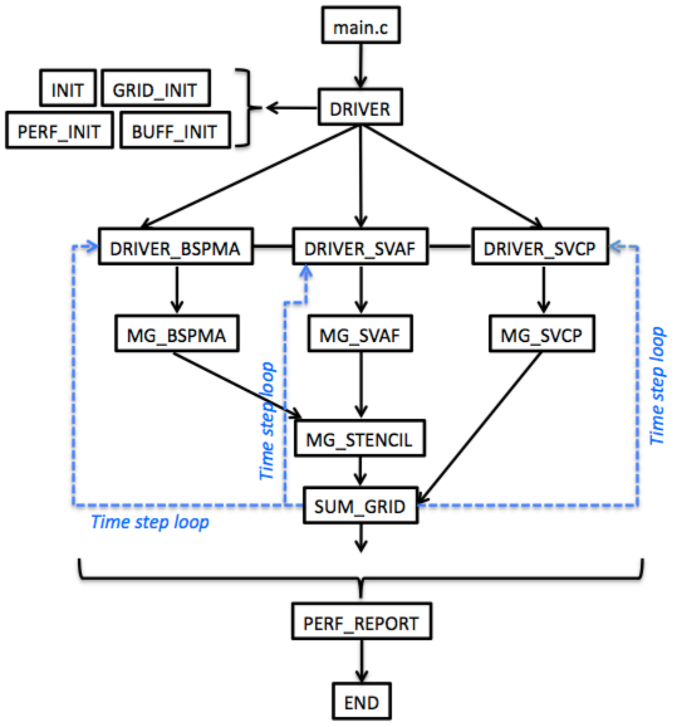


Figure 4. miniGhost code flow diagram

as the file name. Where convenient, names are parameterized. For example, `MG_STENCIL_xDyPT` refers to a y-point stencil in x dimensions. Currently this set includes 5- and 9-point stencils in two dimensions and 7- and 27-point stencils in three dimensions.

A listing of the source code files that compose miniGhost is shown in Table 2. Table 3 lists the MPI functionality employed by miniGhost. Appendix A provides a breakdown of the source code. The number of lines of code is magnified by the redundancy employed by the implementation as a means of clarity as well as the inclusion of several options to the basic BSP message aggregation model. The basics are captured in the boundary exchange and stencil computation procedures. (Note that for the asynchronous versions, these two functional requirements are combined into a single procedure.)

Functionality	*Subroutine*	*Alternatives*
Setup and support	main.c DRIVER MG_BUFINIT MG_CONSTANTS MG_OPTIONS MG_PROFILING MG_UTILS	
Boundary exchange driver	DRIVER_BSPMA	DRIVER_SVAF DRIVER_SVCP
Boundary exchange	MG_BSPMA MG_BSPMA_DIAGS	MG_SVAF MG_SVCP
Driver for stencil option	MG_STENCIL	
y-point stencil computation in x dimensions	MG_STENCIL_xDyPT	
Async y-point stencil computation in x dimensions		MG_SVCP_xDyPT
Manages the reduction (summation) across a grid	MG_SUM_GRID	
Performs the reduction (summation) across a grid	MG_ALLREDUCE	
Post non-blocking receives	MG_IRECV	
Pack face into message buffer	MG_PACK	MG_PACK_SVAF
Manages send protocol	MG_SEND	
Blocking send protocol.	MG_BSEND	
Non-blocking send protocol.		MG_ISEND
Non-blocking send protocol.		MG_ISEND_SVAF
Message completion	MG_UNPACK_AGG	MG_UNPACK
Unpack message buffer into GRIDs ghost space	MG_GET_FACE	

Table 2. MiniGhost functionality

Subroutine	*Use*
MPI_IRECV MPI_SEND MPI_WAITANY	Core functionality
MPI_ALLREDUCE MPI_ISEND MPI_RECV	Optional Core functionality
MPI_ABORT MPI_BCAST MPI_COMM_DUP MPI_COMM_RANK MPI_COMM_SIZE MPI_ERRHANDLER_SET MPI_INIT MPI_GATHER MPI_FINALIZE MPI_REDUCE	Support functionality

Table 3. MPI functionality employed

2.5 Parallel Programming Model

MiniGhost is configured using the Single Program Multiple Data (SPMD) parallel programming model, with parallelism enabled using functionality defined in the MPI specification [17]. MPI provides a wealth of mechanisms and configurations for point-to-point interprocess communication. Our choice is motivated by that employed by the widest number of applications in our experience, and reinforced in discussions with many MPI implementers. Here, non-blocking receives for all communication partners are posted, followed by all non-blocking sends, followed by completion of these procedures as a whole. Sends are preceded by data copies into message buffers where needed; upon completion, receives are followed by unpacking of data into appropriate data structures where needed. Figure 5 illustrates the idea with a code fragment. We anticipate that different configurations might result in meaningfully different (and perhaps better) performance on different platforms with different MPI implementations, an issue we intend to explore as a general study using this and other miniapps.

2.6 Extensibility

As illustrated in Figure 4, miniGhost is constructed using a modular design. In particular, the separation of the stencil computation and boundary exchange communication lends this miniapp to experimentation in a variety of ways. For example, new stencils, adding weights to the stencils, or alternative MPI functionality could be configured. Significantly different implementations of the required functionality can also be configured, with some examples described in this report's summary Section 5.

```
DO I = 1, NUM_RECVS
   CALL MPI_IRECV ( ..., MSG_REQ(I), ... )
END DO

!  Perhaps some buffer packing

DO I = 1, NUM_SENDS
   CALL MPI_ISEND ( ..., MSG_REQ(I+NUM_RECVS), ... )
END DO

DO I = 1, NUM_RECVS + NUM_SENDS

   CALL MPI_WAITANY ( NUM_RECVS + NUM_SENDS, MSG_REQ, IWHICH, ISTAT, IERR )

   IF ( IWHICH <= NUM_RECVS ) THEN
   !  Perhaps some unbuffer packing
   !  else completed send, no action required.
   END IF

END DO
```

Figure 5. Sketch of miniGhost boundary exchange

2.7 Peer implementations

Mantevo miniapps are designed to serve as a tractable means of describing key performance issues within the context of large scale scientific and engineering application codes. As such, they are purposely written using the most ubiquitous languages (C, C++, Fortran) and parallel programming mechanism (MPI), providing what we refer to as the *reference implementation.* This provides a means for exploring alternative, emerging and future architectures. We anticipate implementations based on compiler directives such as OpenMP [10] andOpenACC[3], Fortran co-arrays, as well as alternative and developing programming models and languages, such as Chapel [8] and X10 [9], and perhaps some functional languages. We also anticipate developing an implementation based on the C programming language.

[3]http://www.openacc-standard.org/

3 A Represented Application, CTH

CTH is a multi-material, large deformation, strong shock wave, solid mechanics code developed at Sandia National Laboratories [16]. It includes models for multi-phase, elastic viscoplastic, porous and explosive materials, using second-order accurate numerical methods to reduce dispersion and dissipation and produce accurate, efficient results. The numerical algorithms used in CTH solve the equations of mass, momentum, and energy in an Eulerian finite difference formulation on a three-dimensional Cartesian mesh. CTH can be used in either a flat mesh mode where the faces of adjacent cells are coincident or in a mode with Adaptive Mesh Refinement (AMR) where the mesh can be finer in areas of the problem where there is more activity. miniGhost is currently implemented using only a flat mesh. Future plans call for incorporating the behavior of an AMR mesh. CTH is used throughout the United States DOE complex, and is part of the US Department of Defense (DoD) High Performance Computing Modernization Program (HPCMP) test suite [7].

The domain is divided into three dimensional regions, which are mapped to parallel processes. Each time step, CTH exchanges boundary data (two dimensional "faces") 19 times, in each of the three dimensions, and 3 calls to propagate data across faces. Figure 3 illustrates this implementation, including a basic view of the computation across a two dimensional slice of the domain, sending an $y-z$ face in the x direction. Each boundary exchange aggregates data from (typically) tens of arrays, each representing say a material state. So with, say, a local domain square domain of dimension 100, with say 40 variables, the message sizes will be $100 \times 100 \times 40 \times 8$ bytes = 3.2 MBytes. Collective communication, called 90 times each time step, is typically a reduction (`MPI_Allreduce`) of small counts.

Computation is characterized by regular memory accesses, is fairly cache friendly, with operations focusing on two dimensional planes. Each parallel process can have a maximum of six communication neighbors; for a typical problem that number is reached (for some processes) once 128 parallel processes are employed.

3.1 Model abstractions of CTH

Miniapps are not intended to capture all aspects of a particular application. Instead, they are designed to represent issues that critically impact the runtime characteristics of large scale application programs. In addition to enabling a stronger focus on a limited set of important issues in a particular application, this approach allows a miniapp to be representative of more than one application, and in some cases, represent a class of applications and algorithms. In the case of miniGhost, attention is focused on the nearest neighbor inter-process communication strategy, with computation mainly serving to provide enough data and separation of the boundary exchanges from some computation. Toward that end, below is a list of some key distinctions between miniGhost and CTH.

stencils CTH is a finite volume code, and thus it has values that are based at cells and at the nodes. CTH does several things with the values of variables in both the cell being computed and the surrounding cells. Portions of the code simply use the values of different variables in a given cell to calculate new values of those and other variables in that cell (for example equation of state calculations). Other portions use the value of variables in a cell and those neighboring cells in some direction to calculate new values (such as advection). And other portions of the code use values from variables in all 26 neighboring cells to calculate values of variables in a cell (such as interface tracking). This can alter cache and other behavior. MiniGhost computations in some sense, though, provide a meaningful context for the interprocess communication, which looks similar to a finite difference code.

ack to ensure receive posted In CTH, prior to sending a message containing boundary data to a neighbor, an MPI process waits for a message from the target processes, which alerts the sending process that the matching receive is posted. The intent is to avoid unexpected messages, potentially a serious issue given the typically large amount of data transmitted. This is a meaningful approach on some architectures, such as Red Storm [24], but is of no help on most, which include a built-in acknowledgement handshake prior to sending messages. Further, the MPI specification provides functionality managing this. Regardless, our intent is to test the capabilities of the MPI implementation on the target architecture external of some means for adapting to the specific capabilities of a particular implementation. That said, one use of a miniapp is to provide a lower impact means for testing other approaches and ideas.

reductions CTH makes several calls to MPI colletives (e.g. 90 for some problem sets, and typically to `MPI_Allreduce` with a small count input) throughout a time step. MiniGhost is configured to include this as an option, with the intent of injecting collective synchronization.

data structure CTH manages material mesh data as sets of two dimensional slices, contained in a single pool of allocated memory. This memory management scheme is a relic of past language constraints. MiniGhost allocates distinct three dimensional arrays, each representing a material.

load imbalance and AMR CTH is a multi-material code, so over time materials enter and exit cells, altering the computational load as well as interprocess communication requirements. Further, CTH also provides the option for adaptive mesh refinement (AMR), which focuses attention on cells under some specified condition. The cell is (evenly) divided into four new cells. If necessary, neighboring cells will be divided in order to maintain at most two neighboring cells. Future plans call for the incorporation load imbalance into miniGhost as a means of studying the effects of this behavior.

4 Toward Validation

A first step in ensuring that a miniapp is predictive of some key performance characteristic of a full application is to build up a knowledge basis of the runtime characteristics of both codes. Some basic profiling, combined with a strong understanding of the proxy and an application representing the sorts of application it is intended to represent, can provide some level of confidence.

We examined two common problems modeled by CTH. The meso-scale impact in a confined space problem is computationally well-balanced across the parallel processes. This problem involves 11 materials, inducing the boundary exchange of 75 variables. The shaped charge problem, illustrated in Figure 4, involves four materials, inducing the boundary exchange of 40 variables.

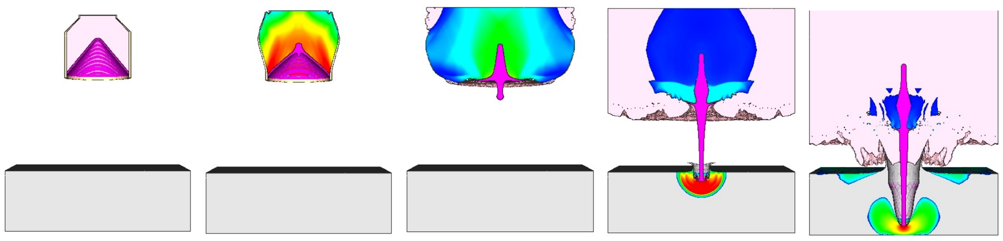
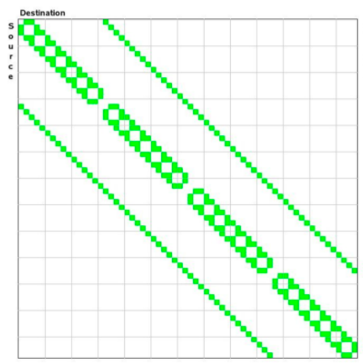


Figure 6. CTH shaped charge simulation

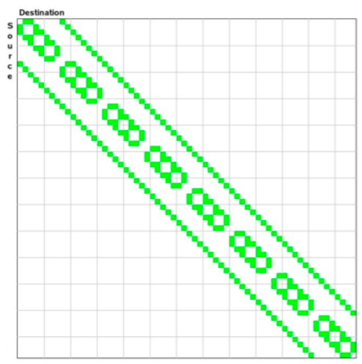
Time progresses left to right.

(This problem was used in the acceptance testing for the NNSA ASC campaign's latest capability computer, Cielo [11].)

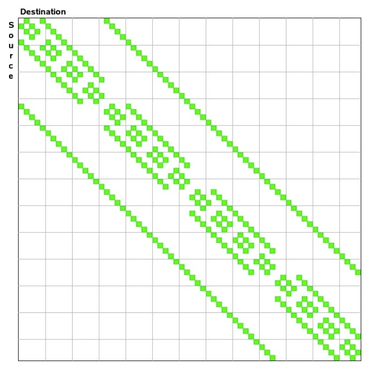
We begin with a comparison of the inter-process communication patterns. Point-to-point communication for *need 128 ranks!* is illustrated in Figure 7. Toward the end of the time step, the



(a) CTH shaped charge



(b) CTH mesoscale

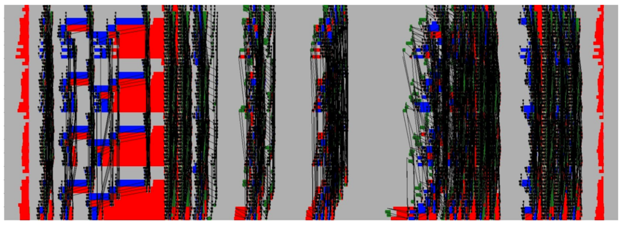


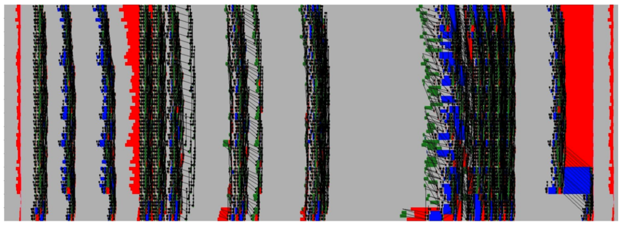
(c) miniGhost

Figure 7. Communication patterns.

shaped charge problem shows some additional communication that is a result of the high explosive calculations. (Recall that 128 MPI processes are required to fully capture the inter-process communication patterns.)

The runtime profile[4] for each implementation is shown in Figure 8. Black represents point-to-


(a) CTH shaped charge


(b) CTH mesoscale

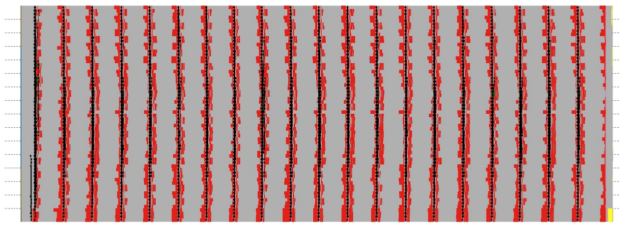

(c) miniGhost

Figure 8. Runtime profiles

point communication, red represents synchronization time waiting for collectives or for communication to finish, green represents sends, blue represents receives, and gray represents computation. These figures show that both problems behave similarly overall, but have different amounts of synchronization time which occur at different points in the time step. CTH has several points during a time step where boundary information is aggregated and exchanged with up to six neighbors in the grid of processors. For the shaped-charge problem these messages average 4.1 MB and for the meso-scale problem these messages average 10.4 MB. Process 0 and a couple of other processors near it for the shaped charge problem have more work since they are the genesis of the explosion and thus have additional work relative to the other processes. The runtime trace shows significantly less waiting time than the other cores.

Note that CTH includes several reductions (mostly `MPI_Allreduce` on 8-byte data) during each time step. Further, CTH implements an acknowledgement system whereby messages are

[4]Generated using CrayPAT and visualized using Apprentice2.

not transmitted until the receive posts the matching receive. The goal is to prevent unexpected messages due to the size of the transmitted data (an issue further discussed in Section 3.1).

We compared performance of miniGhost and CTH on a Cray XT5. Results are shown in Figure 9. Run in weak scaling mode on up to 1,024 processor cores (this XT5 is a dual-socket AMD

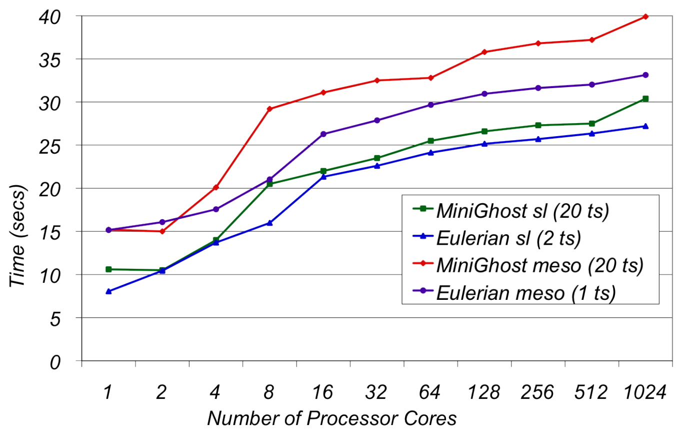


Figure 9. CTH and miniGhost performance comparison

Runtime performance on a Cray XT5.

Opteron Istanbul hex-core node based machine with SeaStar interconnect [26], miniGhost tracks CTH performance reasonably well.

These data provide a justification for applying a more detailed study. Toward that end, we are developing a general methodology for assessing the validity of the relationship between for use across the Mantevo suite [5]. This methodology adheres to the spirit of experimental validation as described in [19, 18, 12, 13] because validation referents are intended to be representative of the empirical (that is, "real") performance of the full applications. Our focus is strictly on the computational runtime characteristics. Beyond this issue we also stress that miniapps are not intended to reproduce specific physics and mathematics represented by the full application. To some degree, we therefore have an operating assumption that a valid miniapp can approximate the runtime performance characteristics of a full application to a useful degree without reproducing the mathematics and physics of the full application to a useful degree. This may be an assumption that is worthy of fuller consideration also, but in the current context it is beyond the scope of our near term priorities.

5 Summary

MiniGhost is a miniapp developed within the scope of the Mantevo project. It is designed to provide a means to explore the Bulk Synchronous Parallel programming model, supplemented with message aggregation, in the context of exchanging inter-process boundary data typically seen in finite difference and finite volume computations. This programming model is employed across a breadth of science domains, typically for solving partial differential equations. MiniGhost was inspired by the multi-decades experiences by the authors with these sorts of parallel programs, and the desire to explore alternative configurations on current, emerging, and future computing environments. It also provides a means for exploring alternative programming languages as well as alternative semantics of MPI.

A validation methodology was applied to demonstrate the predictive capabilities of miniGhost with regard to a well-known and heavily used application of interest, CTH. This also let us demonstrate how different application problem sets may be mimicked in a miniapp.

Toward that end, two alternative boundary communication strategies are included for the boundary exchange, designed to explore the capabilities of computer node inter-connects. Additionally, collective communication may be inserted throughout the time steps, adding an additional level of realism for many application programs. Further, computation may be "turned off", providing a skeleton app capability whereby inter-process communication requirements may used as a "stress test" to explore inter-connect capabilities external of computation.

References

[1] S. Ahern, S.R. Alam, M.R. Fahey, R. Hartman-Baker, R.F. Barrett, R. Kendall, D. Kothe, O.E. Messer, R. Mills, R. Sankaran, A. Tharrington, and J.B. White III. Scientific Application Requirements for Leadership Computing at the Exascale. Technical Report TM-2007/238, Oak Ridge National Laboratory, December 2007.

[2] S. Alam, R. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. S. Vetter, P. Worley, and W. Yu. Early Evaluation of IBM BlueGene/P. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 23:1–23:12, Piscataway, NJ, USA, 2008. IEEE Press.

[3] K. Alvin, R. Brightwell, and S. Dosanjh et al. On the Path to Exascale. *International Journal of Distributed Systems and Technologies*, 1(2), 2010.

[4] R.F. Barrett, S. Ahern, M.R. Fahey, R. Hartman-Baker, J.K. Horner, S.W. Poole, and R. Sankaran. A Taxonomy of MPI-Oriented Usage Models in Parallelized Scientific Codes. In *The International Conference on Software Engineering Research and Practice*, 2009.

[5] Richard F. Barrett, Paul S. Crozier, Simon D. Hammond, Michael A. Heroux, Paul T. Lin, Timothy G. Trucano, Courtenay T. Vaughan, and Alan B. Williams. Assessing the Validity of the Role of Mini-Applications in Predicting Key Performance Characteristics of Scientific and Engineering Applications. Technical Report SAND2012-TBD, Sandia National Laboratories, 2012.

[6] B. Bland. Jaguar: The World's Most Powerful Computer System. In *The 52nd Cray User Group meeting*, 2010.

[7] Laura Brown, Paul M. Bennett, Mark Cowan, Carrie Leach, and Thomas C. Oppe. Finding the Best HPCMP Architectures Using Benchmark Application Results for TI-09. *HPCMP Users Group Conference*, 0:416–421, 2009.

[8] B.L. Chamberlain, D.Callahan, and H.P. Zima. Parallel Programming and the Chapel Language. *International Journal on High Performance Computer Applications*, 21(3):291–312, 2007.

[9] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA)*, October 2005.

[10] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46 –55, 1998.

[11] D. Doerfler, Mahesh Rajan, Cindy Nuss, Cornell Wright, and Thomas Spelce. Application-Driven Acceptance of Cielo, an XE6 Petascale Capability Platform. In *Proc. 53rd Cray User Group Meeting*, 2011.

[12] M. Pilch et al. Guidelines for Sandia ASCI Verification and Validation Plans - Content and Format: Version 2.0. Technical Report SAND2000-3101, Sandia National Laboratories, 2000.

[13] T. G. Trucano et al. General Concepts for Experimental Validation of ASCI Code Applications. Technical Report SAND2002-0341, Sandia National Laboratories, 2002.

[14] Al Geist and Sudip Dosanjh. IESP Exascale Challenge: Co-Design of Architectures and Algorithms. *Int. J. High Perform. Comput. Appl.*, 23:401–402, November 2009.

[15] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James W. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, September 2009. https://software.sandia.gov/mantevo/.

[16] E. S. Hertel, Jr., R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. Petney, S. A. Silling, P. A. Taylor, and L. Yarrington. CTH: A Software Family for Multi-Dimensional Shock Physics Analysis. In *Proceedings, 19th International Symposium on Shock Waves*, 1993.

[17] MPI Forum. *MPI: A Message Passing Interface Standard, Version 2.2*, 2009. http://www.mpi-forum.org/docs/mpi22-report.pdf.

[18] W. L. Oberkampf and C. J. Roy. *Verification and Validation in Scientific Computing.* Cambridge University Press, 2010.

[19] W. L. Oberkampf and T. G. Trucano. Verification and validation in computational fluid dynamics. *Progress in Aerospace Sciences*, 38, 2002.

[20] University of Southern California Center for Software Engineering. CodeCount^{TM}toolset. www.sunset.usc.edu/research/CODECOUNT, 1998.

[21] M. Rajan, C.T. Vaughan, D.W. Doerfler, R.F. Barrett, P.T. Lin, K.T. Pedretti, and K.S. Hemmert. Application-driven Analysis of Two Generations of Capability Computing Platforms: Purple and Cielo. *Computation and Concurrency: Practice and Experience*, 2012. To appear.

[22] H. Simon, T. Zacharia, and R. Stevens. Modeling and Simulation at the Exascale for Energy and the Environment: Report on the Advanced Scientific Computing Research Town Hall Meetings on Simulation and Modeling at the Exascale for Energy, Ecological Sustainability and Global Security (E3). Technical report, Office of Science, The U.S. Department of Energy, 2007.

[23] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference: Volume 1 - 2nd Edition.* The MIT Press, 1998.

[24] James L. Tomkins, Ron Brightwell, William J. Camp, Sudip Dosanjh, Suzanne M. Kelly, Paul T. Lin, Courtenay T. Vaughan, John Levesque, and Vinod Tipparaju. The Red Storm Architecture and Early Experiences with Multi-core Processors. *International Journal of Distributed Systems and Technologies (IJDST)*, 1(2), April – June 2010.

[25] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33:103–111, August 1990.

[26] Courtenay Vaughan, Mahesh Rajan, Richard Barrett, Doug Doerfler, and Kevin Pedretti. Investigating the Impact of the Cielo Cray XE6 Architecture on Scientific Application Codes. In *Workshop on Large Scale Parallel Processing, at the IEEE International Parallel & Distributed Processing Symposium (IPDPS) Meeting*, 2011. SAND 2010-8925C.

A Code categorization

This section provides a breakdown of the source code, as reported by CodeCount [20]. Cross referencing to Figure 4 should provide a basic understanding of the source code for miniGhost.

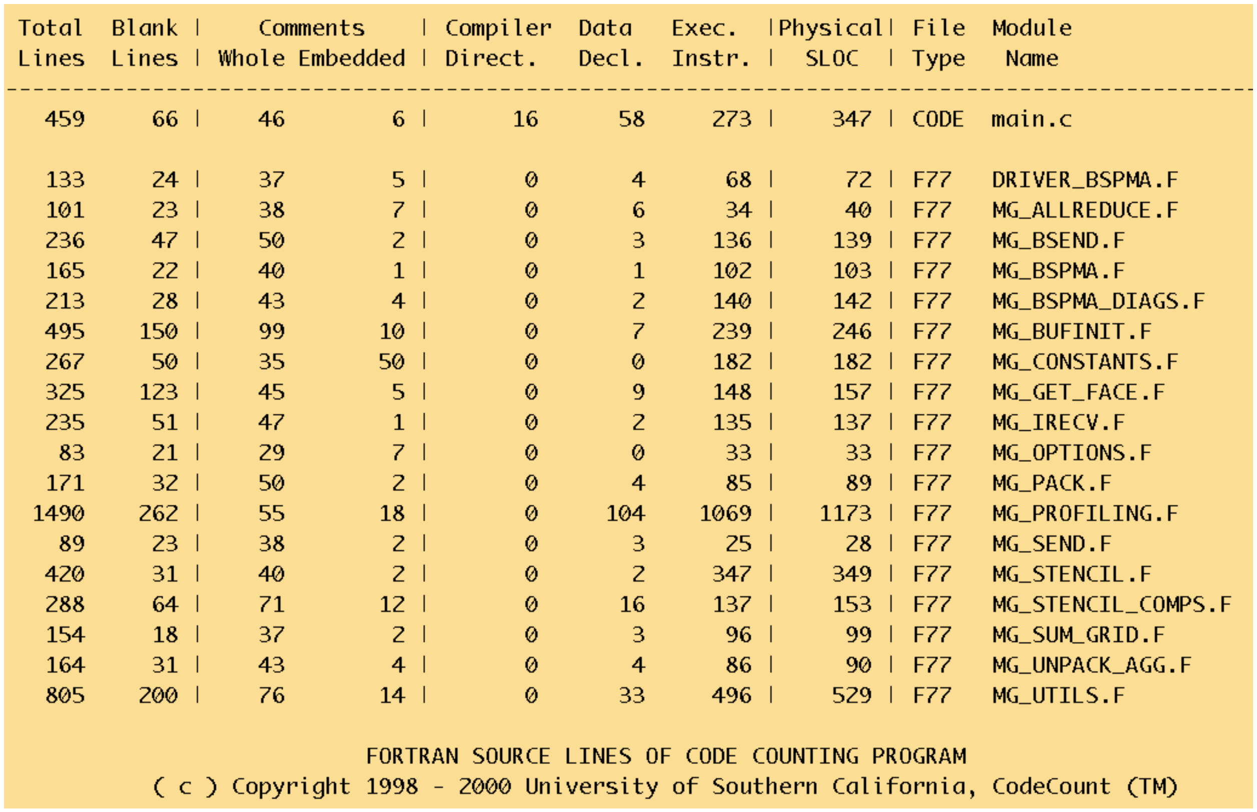
The distinction between a physical and logical line of code is discussed in the READ file (`USC_Read_Me-TXT.txt`) distributed with the CodeCount package:

"PHYSICAL SLOC : The number of physical SLOCs within a source file is defined to be the sum of the number of physical SLOCs (terminated by a carriage return or EOLN character) which contain program instructions created by project personnel and processed into machine code by some combination of preprocessors, compilers, interpreters, and/of assemblers. It excludes comment cards and unmodified utility software. It includes job control language (compiler directive), format statements, and data declarations (data lines). Instructions are defined as lines of code or card images. Thus, a line containing two or more source statements count as one physical SLOC; a five line data declaration counts as five physical SLOCs. The physical SLOC definition was selected due to (1) compatibility with parametric software cost modeling tools, (2) ability to support software metrics collection, and (3) programming language syntax independence.

LOGICAL SLOC : The number of logical SLOC within a source file is defined to be the sum of the number of logical SLOCs classified as compiler directives, data lines, or executable lines. It excludes comments (whole or embedded) and blank lines. Thus, a line containing two or more source statements count as multiple logical SLOCs; a single logical statement that extends over five physical lines count as one logical SLOC. Specifically, the logical SLOC found within a file containing software written in the PL/I programming language may be computed by summing together the count of (1) the number of terminal semicolons, (2) the number of terminal commas contained within a DECLARE (DCL) statement, and (3) the number of logical compiler directives that do not terminate with a terminal semicolon, i.e., JCL directives. The logical SLOC definition was selected due to (1) compatibility with parametric software cost modeling tools, and (2) ability to support software metrics collection. The logical SLOC count is susceptible to erroneous output when the analyzed source code file contains software that uses overloading or replacement characters for a few key symbols, e.g., ';' ."

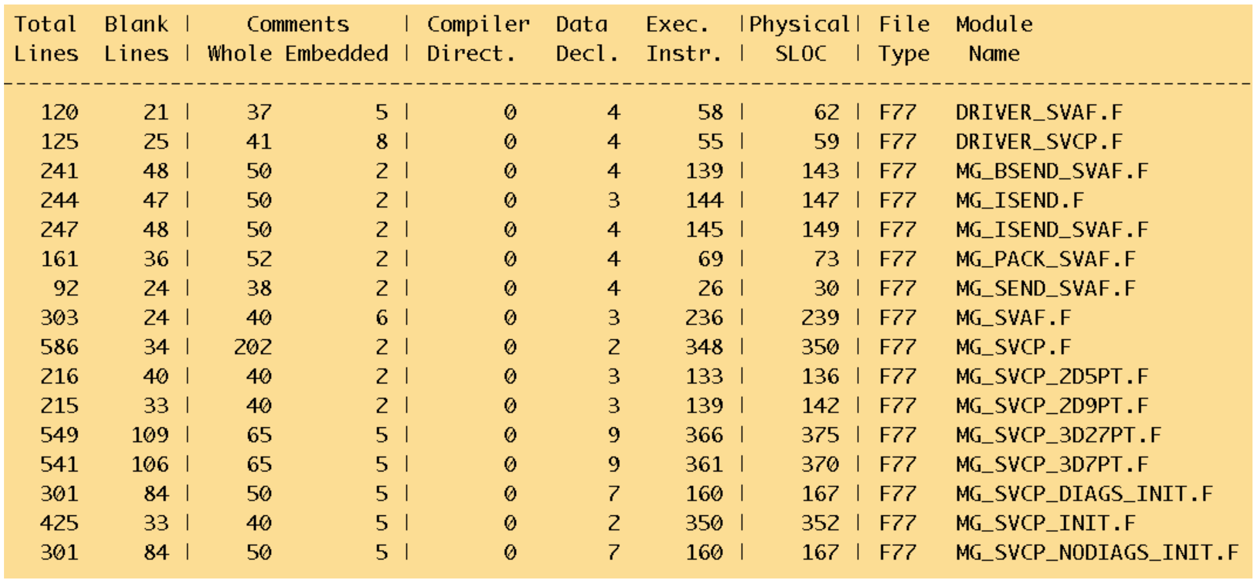
– End quoted text.

Table 3 above listed the MPI functionality employed by miniGhost. Figure A.1 lists the general SLOC for the BSPMA and alternative implementation, respectively, broken down by file. Figures A.2 and A.3 list details of the SLOC for the BSPMA and alternative implementation, respectively.



```
Total  Blank |     Comments     | Compiler  Data   Exec.  |Physical| File  Module
Lines  Lines | Whole Embedded | Direct.   Decl.  Instr.  |  SLOC  | Type   Name
---------------------------------------------------------------------------------
  459     66 |    46         6 |      16      58     273 |    347 | CODE  main.c

  133     24 |    37         5 |       0       4      68 |     72 | F77   DRIVER_BSPMA.F
  101     23 |    38         7 |       0       6      34 |     40 | F77   MG_ALLREDUCE.F
  236     47 |    50         2 |       0       3     136 |    139 | F77   MG_BSEND.F
  165     22 |    40         1 |       0       1     102 |    103 | F77   MG_BSPMA.F
  213     28 |    43         4 |       0       2     140 |    142 | F77   MG_BSPMA_DIAGS.F
  495    150 |    99        10 |       0       7     239 |    246 | F77   MG_BUFINIT.F
  267     50 |    35        50 |       0       0     182 |    182 | F77   MG_CONSTANTS.F
  325    123 |    45         5 |       0       9     148 |    157 | F77   MG_GET_FACE.F
  235     51 |    47         1 |       0       2     135 |    137 | F77   MG_IRECV.F
   83     21 |    29         7 |       0       0      33 |     33 | F77   MG_OPTIONS.F
  171     32 |    50         2 |       0       4      85 |     89 | F77   MG_PACK.F
 1490    262 |    55        18 |       0     104    1069 |   1173 | F77   MG_PROFILING.F
   89     23 |    38         2 |       0       3      25 |     28 | F77   MG_SEND.F
  420     31 |    40         2 |       0       2     347 |    349 | F77   MG_STENCIL.F
  288     64 |    71        12 |       0      16     137 |    153 | F77   MG_STENCIL_COMPS.F
  154     18 |    37         2 |       0       3      96 |     99 | F77   MG_SUM_GRID.F
  164     31 |    43         4 |       0       4      86 |     90 | F77   MG_UNPACK_AGG.F
  805    200 |    76        14 |       0      33     496 |    529 | F77   MG_UTILS.F

                    FORTRAN SOURCE LINES OF CODE COUNTING PROGRAM
         ( c ) Copyright 1998 - 2000 University of Southern California, CodeCount (TM)
```

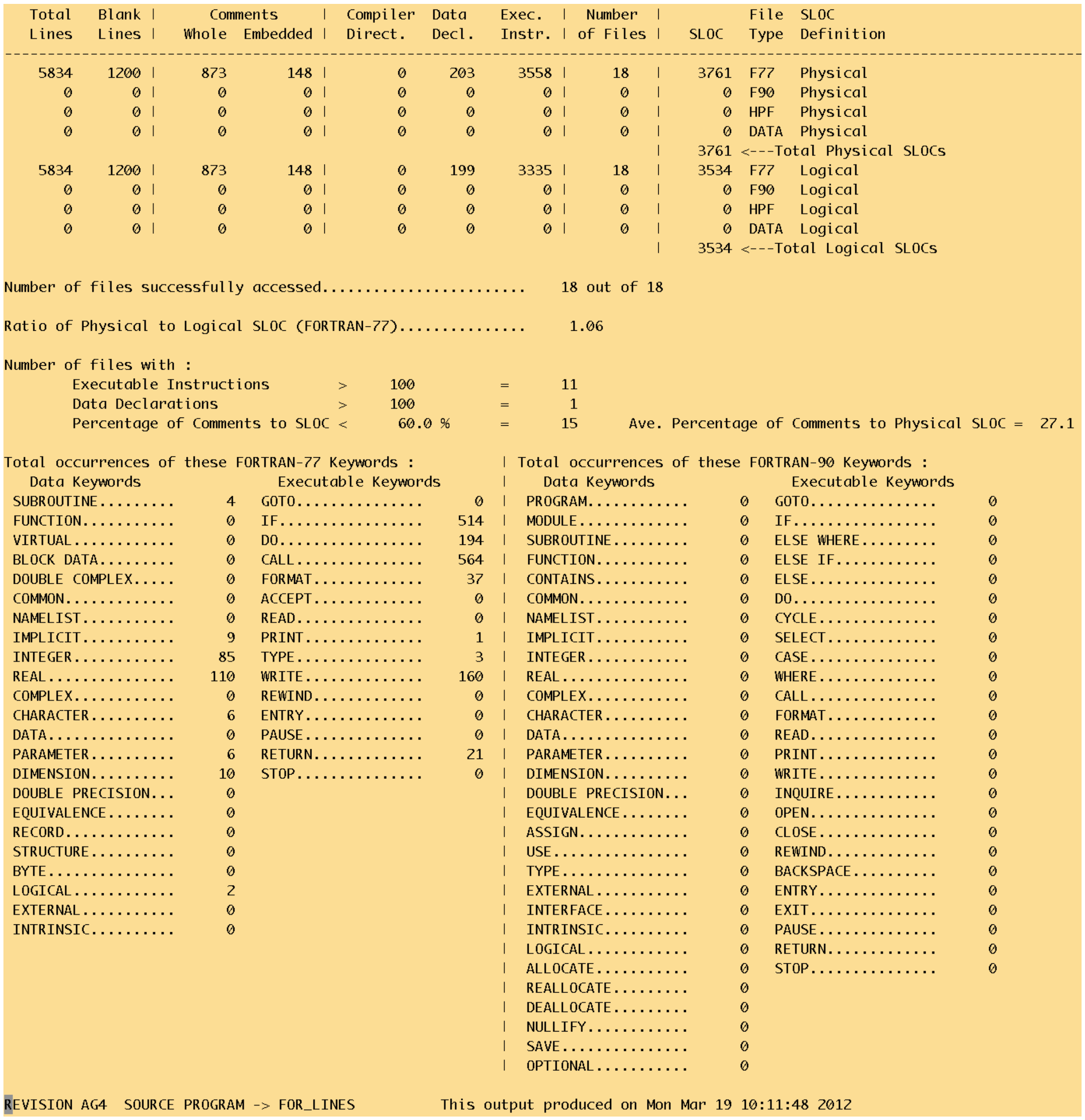
(a) Basic functionality



```
Total  Blank |     Comments     | Compiler  Data   Exec.  |Physical| File  Module
Lines  Lines | Whole Embedded | Direct.   Decl.  Instr.  |  SLOC  | Type   Name
---------------------------------------------------------------------------------
  120     21 |    37         5 |       0       4      58 |     62 | F77   DRIVER_SVAF.F
  125     25 |    41         8 |       0       4      55 |     59 | F77   DRIVER_SVCP.F
  241     48 |    50         2 |       0       4     139 |    143 | F77   MG_BSEND_SVAF.F
  244     47 |    50         2 |       0       3     144 |    147 | F77   MG_ISEND.F
  247     48 |    50         2 |       0       4     145 |    149 | F77   MG_ISEND_SVAF.F
  161     36 |    52         2 |       0       4      69 |     73 | F77   MG_PACK_SVAF.F
   92     24 |    38         2 |       0       4      26 |     30 | F77   MG_SEND_SVAF.F
  303     24 |    40         6 |       0       3     236 |    239 | F77   MG_SVAF.F
  586     34 |   202         2 |       0       2     348 |    350 | F77   MG_SVCP.F
  216     40 |    40         2 |       0       3     133 |    136 | F77   MG_SVCP_2D5PT.F
  215     33 |    40         2 |       0       3     139 |    142 | F77   MG_SVCP_2D9PT.F
  549    109 |    65         5 |       0       9     366 |    375 | F77   MG_SVCP_3D27PT.F
  541    106 |    65         5 |       0       9     361 |    370 | F77   MG_SVCP_3D7PT.F
  301     84 |    50         5 |       0       7     160 |    167 | F77   MG_SVCP_DIAGS_INIT.F
  425     33 |    40         5 |       0       2     350 |    352 | F77   MG_SVCP_INIT.F
  301     84 |    50         5 |       0       7     160 |    167 | F77   MG_SVCP_NODIAGS_INIT.F
```

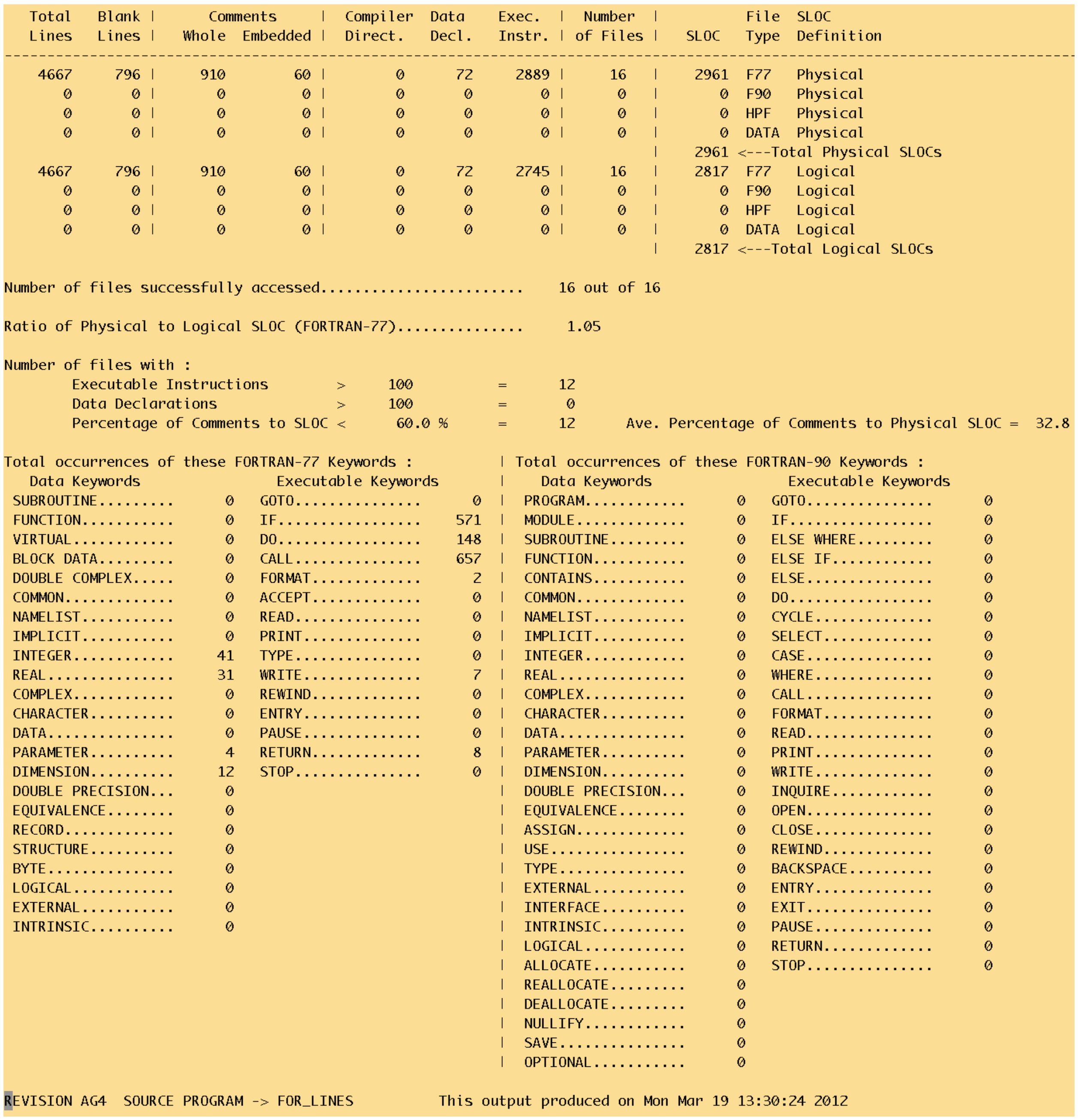
(b) Alternative functionality

Figure A.1. miniGhost SLOC summary by file



```
   Total   Blank |      Comments     |  Compiler  Data    Exec.  |  Number  |         File  SLOC
   Lines   Lines |   Whole  Embedded |  Direct.   Decl.   Instr. | of Files |   SLOC   Type  Definition
----------------------------------------------------------------------------------------------------------------
    5834    1200 |     873       148 |        0     203     3558 |      18   |   3761  F77   Physical
       0       0 |       0         0 |        0       0        0 |       0   |      0  F90   Physical
       0       0 |       0         0 |        0       0        0 |       0   |      0  HPF   Physical
       0       0 |       0         0 |        0       0        0 |       0   |      0  DATA  Physical
                                                                            |   3761 <---Total Physical SLOCs
    5834    1200 |     873       148 |        0     199     3335 |      18   |   3534  F77   Logical
       0       0 |       0         0 |        0       0        0 |       0   |      0  F90   Logical
       0       0 |       0         0 |        0       0        0 |       0   |      0  HPF   Logical
       0       0 |       0         0 |        0       0        0 |       0   |      0  DATA  Logical
                                                                            |   3534 <---Total Logical SLOCs

Number of files successfully accessed........................    18 out of 18

Ratio of Physical to Logical SLOC (FORTRAN-77)...............     1.06

Number of files with :
        Executable Instructions       >     100          =      11
        Data Declarations             >     100          =       1
        Percentage of Comments to SLOC <      60.0 %      =      15       Ave. Percentage of Comments to Physical SLOC =  27.1

Total occurrences of these FORTRAN-77 Keywords :          | Total occurrences of these FORTRAN-90 Keywords :
   Data Keywords                 Executable Keywords      |    Data Keywords                 Executable Keywords
 SUBROUTINE.........      4   GOTO...............     0   |  PROGRAM...........      0   GOTO...............      0
 FUNCTION...........      0   IF.................   514   |  MODULE............      0   IF.................      0
 VIRTUAL............      0   DO.................   194   |  SUBROUTINE........      0   ELSE WHERE.........      0
 BLOCK DATA.........      0   CALL...............   564   |  FUNCTION..........      0   ELSE IF............      0
 DOUBLE COMPLEX.....      0   FORMAT.............    37   |  CONTAINS..........      0   ELSE...............      0
 COMMON.............      0   ACCEPT.............     0   |  COMMON............      0   DO.................      0
 NAMELIST...........      0   READ...............     0   |  NAMELIST..........      0   CYCLE..............      0
 IMPLICIT...........      9   PRINT..............     1   |  IMPLICIT..........      0   SELECT.............      0
 INTEGER............     85   TYPE...............     3   |  INTEGER...........      0   CASE...............      0
 REAL...............    110   WRITE..............   160   |  REAL..............      0   WHERE..............      0
 COMPLEX............      0   REWIND.............     0   |  COMPLEX...........      0   CALL...............      0
 CHARACTER..........      6   ENTRY..............     0   |  CHARACTER.........      0   FORMAT.............      0
 DATA...............      0   PAUSE..............     0   |  DATA..............      0   READ...............      0
 PARAMETER..........      6   RETURN.............    21   |  PARAMETER.........      0   PRINT..............      0
 DIMENSION..........     10   STOP...............     0   |  DIMENSION.........      0   WRITE..............      0
 DOUBLE PRECISION...      0                               |  DOUBLE PRECISION...     0   INQUIRE............      0
 EQUIVALENCE........      0                               |  EQUIVALENCE.......      0   OPEN...............      0
 RECORD.............      0                               |  ASSIGN............      0   CLOSE..............      0
 STRUCTURE..........      0                               |  USE...............      0   REWIND.............      0
 BYTE...............      0                               |  TYPE..............      0   BACKSPACE..........      0
 LOGICAL............      2                               |  EXTERNAL..........      0   ENTRY..............      0
 EXTERNAL...........      0                               |  INTERFACE.........      0   EXIT...............      0
 INTRINSIC..........      0                               |  INTRINSIC.........      0   PAUSE..............      0
                                                          |  LOGICAL...........      0   RETURN.............      0
                                                          |  ALLOCATE..........      0   STOP...............      0
                                                          |  REALLOCATE........      0
                                                          |  DEALLOCATE........      0
                                                          |  NULLIFY...........      0
                                                          |  SAVE..............      0
                                                          |  OPTIONAL..........      0

REVISION AG4  SOURCE PROGRAM -> FOR_LINES         This output produced on Mon Mar 19 10:11:48 2012
```

Figure A.2. miniGhost basic functionality SLOC data



```
   Total   Blank |      Comments      |  Compiler  Data     Exec.  |  Number  |          File  SLOC
   Lines   Lines |    Whole  Embedded |  Direct.   Decl.    Instr. |  of Files |   SLOC   Type  Definition
---------------------------------------------------------------------------------------------------------------------------------
    4667     796 |      910        60 |         0      72     2889 |      16   |    2961  F77   Physical
       0       0 |        0         0 |         0       0        0 |       0   |       0  F90   Physical
       0       0 |        0         0 |         0       0        0 |       0   |       0  HPF   Physical
       0       0 |        0         0 |         0       0        0 |       0   |       0  DATA  Physical
                                                                               |    2961 <---Total Physical SLOCs
    4667     796 |      910        60 |         0      72     2745 |      16   |    2817  F77   Logical
       0       0 |        0         0 |         0       0        0 |       0   |       0  F90   Logical
       0       0 |        0         0 |         0       0        0 |       0   |       0  HPF   Logical
       0       0 |        0         0 |         0       0        0 |       0   |       0  DATA  Logical
                                                                               |    2817 <---Total Logical SLOCs

Number of files successfully accessed.......................    16 out of 16

Ratio of Physical to Logical SLOC (FORTRAN-77)..............     1.05

Number of files with :
        Executable Instructions       >     100         =      12
        Data Declarations             >     100         =       0
        Percentage of Comments to SLOC <      60.0 %      =      12     Ave. Percentage of Comments to Physical SLOC =  32.8

Total occurrences of these FORTRAN-77 Keywords :          | Total occurrences of these FORTRAN-90 Keywords :
   Data Keywords                Executable Keywords       |    Data Keywords                Executable Keywords
 SUBROUTINE.........      0  GOTO...............     0  |  PROGRAM...........      0  GOTO...............      0
 FUNCTION...........      0  IF.................   571  |  MODULE............      0  IF.................      0
 VIRTUAL............      0  DO.................   148  |  SUBROUTINE.........     0  ELSE WHERE.........      0
 BLOCK DATA.........      0  CALL...............   657  |  FUNCTION...........     0  ELSE IF............      0
 DOUBLE COMPLEX.....      0  FORMAT.............     2  |  CONTAINS...........     0  ELSE...............      0
 COMMON.............      0  ACCEPT.............     0  |  COMMON.............     0  DO.................      0
 NAMELIST...........      0  READ...............     0  |  NAMELIST...........     0  CYCLE..............      0
 IMPLICIT...........      0  PRINT..............     0  |  IMPLICIT...........     0  SELECT.............      0
 INTEGER............     41  TYPE...............     0  |  INTEGER............     0  CASE...............      0
 REAL...............     31  WRITE..............     7  |  REAL...............     0  WHERE..............      0
 COMPLEX............      0  REWIND.............     0  |  COMPLEX............     0  CALL...............      0
 CHARACTER..........      0  ENTRY..............     0  |  CHARACTER..........     0  FORMAT.............      0
 DATA...............      0  PAUSE..............     0  |  DATA...............     0  READ...............      0
 PARAMETER..........      4  RETURN.............     8  |  PARAMETER..........     0  PRINT..............      0
 DIMENSION..........     12  STOP...............     0  |  DIMENSION..........     0  WRITE..............      0
 DOUBLE PRECISION...      0                             |  DOUBLE PRECISION...     0  INQUIRE............      0
 EQUIVALENCE........      0                             |  EQUIVALENCE........     0  OPEN...............      0
 RECORD.............      0                             |  ASSIGN.............     0  CLOSE..............      0
 STRUCTURE..........      0                             |  USE................     0  REWIND.............      0
 BYTE...............      0                             |  TYPE...............     0  BACKSPACE..........      0
 LOGICAL............      0                             |  EXTERNAL...........     0  ENTRY..............      0
 EXTERNAL...........      0                             |  INTERFACE..........     0  EXIT...............      0
 INTRINSIC..........      0                             |  INTRINSIC..........     0  PAUSE..............      0
                                                        |  LOGICAL............     0  RETURN.............      0
                                                        |  ALLOCATE...........     0  STOP...............      0
                                                        |  REALLOCATE.........     0
                                                        |  DEALLOCATE.........     0
                                                        |  NULLIFY............     0
                                                        |  SAVE...............     0
                                                        |  OPTIONAL...........     0

REVISION AG4  SOURCE PROGRAM -> FOR_LINES          This output produced on Mon Mar 19 13:30:24 2012
```

Figure A.3. miniGhost alternative functionality SLOC data

DISTRIBUTION:

1 Dorian Arnold, Dept. of Computer Science, MSC01 1130, 1 University of New Mexico, Albuquerque, NM 87131-0001

1 Jeanine Cook, Klipsch School of Electrical and Computer Engineering, Thomas and Brown 127, New Mexico State University, Las Cruces, NM 88003

1 John Gustafson, Intel Corporation, 964 Piemonte Dr., Pleasanton, CA 94566-2139

1 Sharon Hu, Dept. of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556

1 Sameer Shende, ParaTools, Inc., 2836 Kincaid St., Eugene, OR 97405

1 Tim Tautges, 1500 Engineering Dr., Madison, WI, 53706

1 Paul Woodward, The University of Minnesota, Laboratory for Computational Science and Engineering, ECE Department, 499 Walter Library, 117 Pleasant St SE, Minneapolis MN 55455

1 Patrick Worley, Oak Ridge National Laboratory, PO Box 2008, MS 6173, Oak Ridge, TN 37831-6173

1	MS 1319	James Ang, 1422
1	MS 1322	Sudip Dosanjh, 1420
1	MS 0321	Robert Leland, 1400
1	MS 9159	James Brandt, 8953
1	MS 9152	Robert Clay, 8953
1	MS 9158	David Evensky, 8966
1	MS 9152	Ann Gentile, 8953
1	MS 9159	Gilbert Hendry, 8953
1	MS 1320	James Willenbring, 1426
1	MS 0899	Technical Library, 9536 (electronic)

Sandia National Laboratories