Computer Science
Technical Report

The CGPOP Miniapp, Version 1.0

Andrew Stone, John M. Dennis, Michelle Mills Strout

July 8, 2011

Technical Report CS-11-103

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

The CGPOP Miniapp, Version 1.0

Andrew Stone, John M. Dennis, Michelle Mills Strout

July 8, 2011

Abstract

Miniapps provide performance proxies for larger applications thus enabling easier evaluation of performance tuning and refactoring techniques. The CGPOP miniapp is the conjugate gradient solver from Los Alamos National Laboratory's Parallel Ocean Program (POP) version 2.0. This paper describes the conjugate gradient algorithm encapsulated in CGPOP, presents the organization of the CGPOP 1.0 release including the implementations of CGPOP that are included in the release, evaluates CGPOP as a performance proxy for POP, and describes how users of CGPOP can verify the correctness of their own CGPOP rewrites.

1 Introduction

Miniapps enable developers of large programs to efficiently investigate the use of new programming models, data structures, and program optimizations without having to initially incorporate such changes into a much larger application. The miniapp [6] or skeleton app [5]

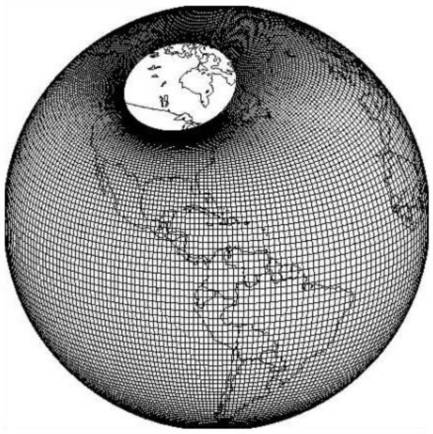

Figure 1: In POP and CGPOP the dipole grid is used to discretize the surface of Earth's oceans.

concept includes an application on the order of 1000 lines of code that incorporates a simple build system and that accurately models the performance of a larger application. Therefore, miniapps also enable high performance computing and parallel programming model researchers to evaluate new techniques within the context of a reasonably-sized version of a real application.

This paper presents CGPOP version 1.0, which is a miniapp for the Parallel Ocean Program (POP) version 2.0 developed at Los Alamos National Laboratory. POP is a global ocean modeling code and a component within the Community Earth System Model (CESM) [1]. CGPOP encapsulates the performance bottleneck of POP, which is the conjugate gradient solver. The CGPOP miniapp is written in Fortran90 with MPI and is about 3000 source lines of code (SLOC), whereas the POP application is 71,000 SLOC. We have been using CGPOP to evaluate different subdomain data structures, computation and communication overlap approaches, and different programming models such as Co-Array Fortran and one-sided MPI. The CGPOP version 1.0 release includes prototypes of each of these variants of the CGPOP miniapp.

In this paper, we overview the algorithms in CGPOP, describe the organization of the CGPOP release, and present an evaluation of CGPOP as a performance proxy for POP. Additionally, we describe how others can create their own CGPOP miniapp variants and verify their correct execution. Our goal in releasing the CGPOP miniapp and some example variants of it are to encourage other researchers to improve the performance of CGPOP in ways that are applicable to POP. One caveat about CGPOP version 1.0 is that it is built on POP version 2.0 that uses the dipole discretization instead of the tripole discretization that is used in the current CESM POP for high resolution runs ($0.1°$). We illustrate the dipole discretization in Figure 1. Future work includes developing a tripole version of the CGPOP miniapp that is a performance proxy for NCAR POP 2.1, which is being used in the CESM 1 release code.

2 Extracting the Conjugate Gradient Solver

Our first step when constructing the miniapp was to identify the most performance impacting part of the POP code. To understand the performance characteristics of POP, we profiled the scalability of three main parts of the application: the barotropic component, the baroclinc solver, and the 3D-update. The barotropic component updates two-dimensional surface pressure data and uses a single production version of a preconditioned conjugate gradient. The POP solver is very sensitive to network latency and OS noise at large core counts [4]. The baroclinc component solves the equations of motion for the ocean model and involves embarrassingly parallel calculations that stream data from the memory hierarchy to the CPU. The 3D-update step is used to update 3-dimensional prognostic variables like temperature and velocity at the end of each timestep.

In Figure 2 we provide a breakdown of the relative computational cost of the three different components of the POP 2.0 $0.1°$ benchmark when run using varying numbers of cores on Kraken. The $0.1°$ has an average of $0.1°$ separation between neighboring grid points

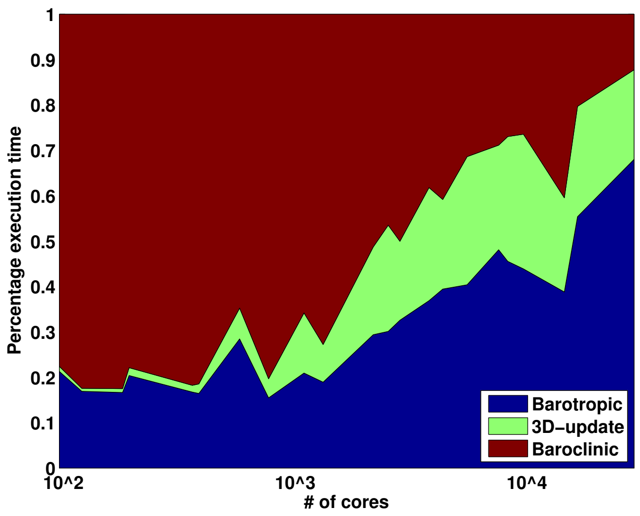

Figure 2: Relative cost of the different components of the POP 0.1° benchmark on Kraken: a Cray XT5

at equator. Kraken is a 99,072 core Cray XT5 system located at the National Institute for Computational Science (NICS). While the relative cost of the baroclinic component is dominant at fewer than 1000 cores, the relative contribution of the communication intensive sections of the code, barotropic and the 3D-update dominate the total cost at greater than 1000 cores. For this reason we have the CGPOP miniapp consists of the conjugate gradient component of POP.

The CGPOP miniapp implements a version of conjugate gradient algorithm, which uses a single inner product [3], to iteratively solve for vector x in the equation $Ax = b$ (see Figure 3). The matrix A, along with the initial guess vector x_0, right hand side vector b, and diagonal preconditioner vector M^{-1} are read from an intermediate state file and passed as inputs into the function CGPOP-solver. The final surface pressure vector x is the output of CGPOP-solver. The CGPOP-solver algorithm consists of a number of linear algebra computations interspersed with two communication steps. The `GlobalSum` function performs a 3-word vector reduction, while `UpdateHalo` function performs a boundary exchange between neighboring subdomains. The `UpdateHalo` function is passed an array that has been distributed across processes using the distribution described in the blocks data-structure of the intermediate state file. The asterisk $*$ indicates a dot product between two vectors involving all entries in the local subdomain, and the `GlobalSum` results in the full dot product being completed.

3 Required Behavior of the CGPOP MiniApp

Broadly, the CGPOP miniapp is defined in terms of its input/output behavior and the algorithm it must conduct. We illustrate this behavior in Figure 4 and list pseudocode for

```
x = function CGPOP-solver($A$,$x_0$,$b$,$M^{-1}$)
!------------------------------------
! Compute initial residual
!------------------------------------
$s = Ax_o$
$r = b - s$

$rr0 = (GlobalSum(r * r))^{1/2}$

UpdateHalo(r)

!------------------------------------
! Single pass of regular CG algorithm
!------------------------------------
$z = M^{-1}r$
$s = z$
$q = As$

UpdateHalo(q)

$\{\rho, \sigma\} = GlobalSum(\{r * z, s * q\})$

!------------------------------------
! Calculate coefficient
!------------------------------------
$\alpha = \rho/\sigma$

!------------------------------------
! Compute next solution and residual
!------------------------------------
$x = x + \alpha s$
$r = r - \alpha q$

do 124 iterations:
    !------------------------------------
    ! Apply preconditioner
    !------------------------------------
    $z = M^{-1}r$
    $az = Az$

    UpdateHalo(az)

    $\{\rho', \delta, \gamma\} = GlobalSum(\{r * z, r * r, az * z\})$

    !------------------------------------
    ! Calculate updated coefficients
    !------------------------------------
    $\beta = \rho'/\rho$
    $\sigma = \delta - \beta^2\sigma$
    $\alpha = \rho/\sigma$
    $\rho = \rho'$

    !------------------------------------
    ! Compute next solution and residual
    !------------------------------------
    $x = x + \alpha(z + \beta s)$
    $r = r - \alpha(az + \beta q)$
    $s = z + \beta s$
    $q = az + \beta s$
```

Figure 3: CGPOP's preconditioned conjugate gradient algorithm.

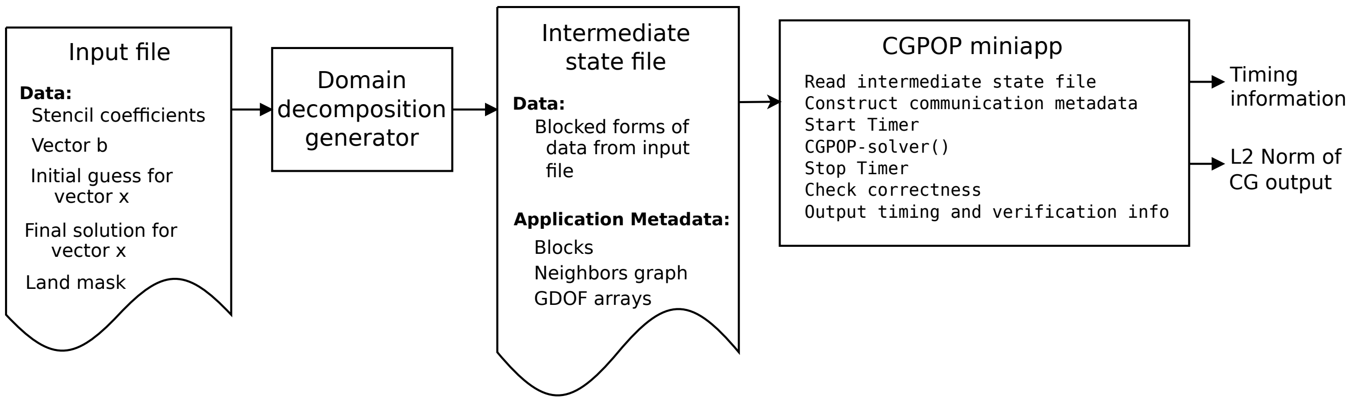

Figure 4: Architecture of the CGPOP Miniapp

the CG algorithm in Figure 3. The CGPOP application is passed an intermediate state file, which is generated by the `cginit` domain decomposition generator.

The `cginit` domain decomposition generator is passed a single input file, which contains a dipole grid stored as a 3600×2400 two-dimensional array and the expected output of the POP conjugate gradient computation for that same grid. The input file also includes the stencil coefficients that are used with the discretization to construct the sparse matrix, a mask to indicate if a grid point is ocean or land, and the initial guess and final solution for vectors x and b of Figure 3. The data stored in the input file is in NetCDF format [7, 2].

The domain decomposition generator breaks the 3600×2400 global domain into subdomain blocks and outputs these blocks into an intermediate state file (also known as a tile file). The generator outputs a unique intermediate state file for each of the block sizes it is configured to work with. By default the generator is configured to work with block sizes of 180x120, 120x80, 90x60, 60x40, 48x32, 36x24, 24x16, and 18x12. Different block sizes are better when different numbers of processors are used, generally smaller block-sizes work better when greater numbers of processors are used.

In addition to the data component of the intermediate state file, there is a metadata that describe the relationship between blocks. There is a set of block information records, a graph of neighbors, and integer arrays that correspond to the global degree of freedom (GDOF) for every point in each block. GDOF values are identifiers that are unique for each grid point in the global domain. The block information records identify the location of each rectangular block within the global domain in terms of two-dimensional indices in the global domain.

After being generated, the intermediate state file can be passed to CGPOP, which will correctness and timing data. Timing information is needed for gathering performance information. The correctness test, which is used for code verification, checks that the difference between L2 norm for x as calculated by the CGPOP miniapp and that calculated by POP is within machine precision.

4 Package Organization

We have implemented several variants of the CGPOP miniapp including the base variant extracted almost directly from POP 2.0, which uses Fortran and MPI. The base implementation serves as a reference that new implementations can be compared against in terms of performance. In this section we describe the existing variants of the miniapp that are packaged in the 1.0 release and overview the organization of the release package.

4.1 Variants of the CGPOP MiniApp Version 1.0

There are several different variants of the CGPOP miniapp included in the 1.0 release package. These variants differ from one another based on how subdomain blocks are stored (either in a one-dimensional or two-dimensional array), the communication mechanism (e.g., two-sided MPI, one-sided MPI, or Co-Array Fortran), and various optimizations such as buffering of data, communication overlap, and whether data is pushed or pulled in for the one-sided versions. We illustrate the variants that use the one-dimensional data-structure in Figure 5, and we illustrate the versions that use the two-dimensional data-structure in Figure 6. The leaf nodes of each tree correspond to implementations.

In both the one-dimensional and two-dimensional data structure variants there is a halo region in the local data array that contains copies of data from neighboring subdomain

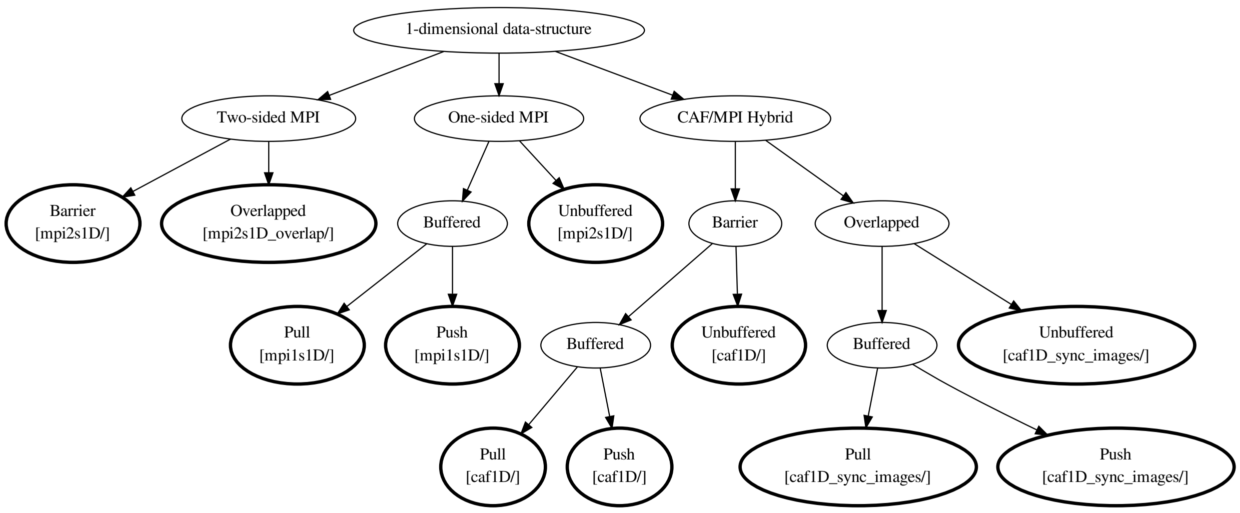

Figure 5: Variants of the CGPOP MiniApp that use a one-dimensional data structure to store each subdomain. The square brackets denote the subdirectory that contains the variant code.

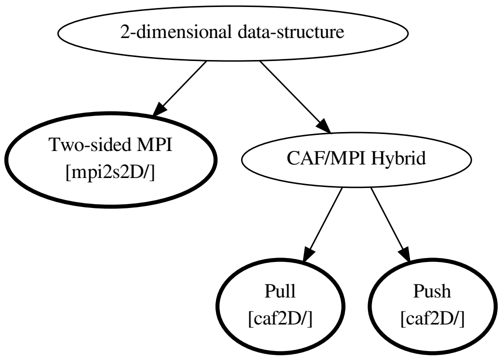

Figure 6: Variants of the CGPOP miniapp that use a two-dimensional data structure to represent each subdomain. Directories for each version are labeled in brackets.

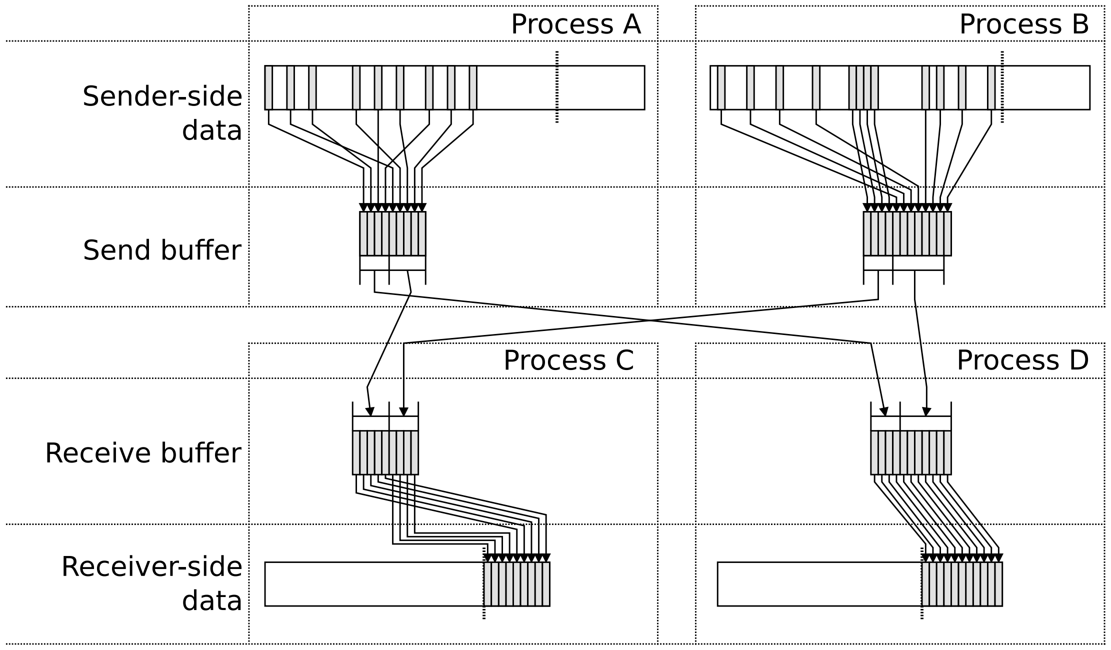

Figure 7: Communication pattern when one-dimensional data structure is used to store blocks.

blocks. (NOTE: the one-dimensional version uses a halo of size 1 and the two-dimensional version uses a halo of size 2). This data is needed in order to update the local subdomain blocks during an iteration of the conjugate-gradient algorithm. In Figure 8 we illustrate the format of the array used in the two-dimensional variants, where two layers of halo cells are used. For the two-dimenaional variants the size of the halo is set by the variable `nghost` in `source/simple_blocks.F90`. The operations to update this halo region are contained in the `UpdateHalo` function. The variants that use a one-dimensional array to store local data will only store data-points that correspond to ocean. This ocean-only data is stored in compressed sparse-row (CSR) format. Note that in the two-dimensional data structure, storage is allocated for points that correspond to land even though these points are not updated.

Among the variants that use the two-dimensional data-structure, there is a two-sided MPI implementation and two CAF/MPI Hybrid implementations. In the CAF/MPI Hybrid version, MPI is used to implement the `GlobalSum` operation. We found it necessary to use MPI for this operation because reduction operations are not currently supported in the CAF standard. For the one-dimensional data structure variants, there are implementations that use two-sided MPI communication, implementations that use one-sided MPI communication, and implementations that use a hybrid of Co-Array Fortran and MPI.

In addition to examining the programming model used, we also look at the impact that optimizations have when applicable for a given model. For the two-dimensional variants that use two-sided MPI implementation and CAF, we include versions that include a barrier synchronization step after performing an `UpdateHalo` and `GlobalSum` , and versions that overlap

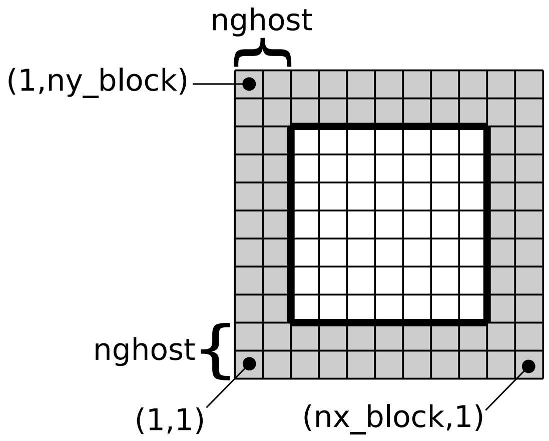

Figure 8: A processor owns a two-dimensional array of points for a block. This array also includes halo cells (shaded), that may have ownership with a different processor. The CGPOP miniapp periodically executes a boundary exchange step to transmit data in order to update halo cells.

communication and computation by only synchronizing between neighboring processes when needed.

Another optimization we look at data buffering. Before data is sent in the two-sided MPI versions, it is aggregated into a buffer so that a single message is sent between a processor and its neighbor. This aggregation is necessary due to the fact that the local data that a neighbor needs may not be stored contiguously in memory. However, for one-sided communication models it is more convenient to not perform such a buffering step and instead work directly with data. A downside to not buffering is that it requires multiple messages to be sent and thus degrades performance. To examine how much impact buffering has we include variants that both do and do not buffer data for one-sided MPI and CAF.

In addition to the decision of whether to buffer data or not, another decision that is necessary with one-sided communication models is whether data is pushed so that each processor issues communication calls to putting data to its neighbors, or pulled so that each processor issues get calls to retrieve data from its neighbors. In two-sided communication models both types of communication calls are specified. In order to examine the performance impact of pushing versus pulling data we include variants that push and variants that pull.

4.2 Common Modules

All existing versions of the miniapp use the modules described in this subsection. These modules can be found under the `source` and `shared` directories. Specifically, modules that are shared between the different implementations are placed in the `source` directory, and modules that are shared between the different implementations and the cginit tile file generator are stored in the `shared` directory.

broadcast	Broadcast routines. This particular version contains MPI versions of these routines
check	Routines to verify that the miniapp produced the correct answer.
communicate	Routines and variables that are necessary for communicating between processors. Routines include `init_communicate`, `exit_message_environment`, `abort_message_environment`, and `get_num_procs`.
constants	Physical and numerical constants used throughout the Parallel Ocean Program.
domain_size	Parameters that define the size of the global model domain and the block decomposition. It is used by the domain and block modules for decomposing the model domain across processors.
exit_mod	Routine to provide a graceful means of exiting from POP when encountering an error.
io_serial	Provides several methods to read and write data needed by the miniapp.
IOUnitsMod	I/O unit manager for tracking, assigning and reserving I/O unit numbers.
kinds_mod	Default numerical data types for all common data types like integer, character, logical, real4 and real8.
simple_blocks	Data types and tools for decomposing a global horizontal domain into a set of blocks. This module contains a data type for describing each block and contains routines for creating and querying the block decomposition for a global domain.
simple_domain	Model domain and routines for initializing the domain. This module also initializes the decompositions and distributions across processors/threads by calling relevant routines in the block, distribution modules.
simple_type	This data structure describes the how blocks are distributed across tasks.
timers	This module contains routine for supporting multiple CPU timers and accumulates time for each individual block and node (task).

In addition to the proceeding modules each of the current CGPOP implementation also include an implementation specific version of the following modules:

broadcast	Broadcast routines. This particular version contains MPI.
cgpop	Main driver file.
linear	This module contains routines for converting to and from the one-dimensional data structure used in the solver in the 1D version of the miniapp.
gshalo	This module includes the boundary exchange routine.
matrix_mod	Routines for multiplying a compressed sparse row matrix and a vector (the linear data structure in the solver).
reductions	This module contains all the routines for performing global reductions like global sums, minvals, maxvals, etc.
solvers	This module contains routines and operators for solving the elliptic system for surface pressure in the barotropic mode.

5 Installing and Using CGPOP

The CGPOP miniapp version 1.0 can be downloaded from `http://www.cs.colostate.edu/hpc/cgpop`. If you would like to quickly tryout the MPI versions of CGPOP on a Linux machine, then follow these instructions:

1. **Setup environment for mpif90**
   mpif90 must be in your path whenever a new shell is started. If you are using C-shell then you should have the following two lines in your .cshrc (with the path changed to point to your local MPI implementation):

   ```
   setenv PATH /usr/lib64/openmpi/bin:$PATH
   setenv LD_LIBRARY_PATH /usr/lib64/openmpi/lib:$LD_LIBRARY_PATH
   ```

   If you are using BASH then include these two lines in your .bashrc:

   ```
   export PATH=/usr/lib64/openmpi/bin:$PATH
   export LD_LIBRARY_PATH=/usr/lib64/openmpi/lib:$LD_LIBRARY_PATH
   ```

2. **Setup environment for netcdf**
   You need to indicate where netcdf was installed. If you did a standard installation in `/usr/local/`, then just set the `NETCDF_DIR` environment variable to `/usr/local/`. If you did a non-standard installation, then you need to set the `LD_LIBRARY_PATH` environment variable as well. In .csrhc:

   ```
   setenv NETCDF_DIR /fullpath/mstrout/software/
   setenv LD_LIBRARY_PATH /fullpath/mstrout/software/lib:$LD_LIBRARY_PATH
   ```

   In .basrhc:

   ```
   export NETCDF_DIR=/fullpath/mstrout/software/
   export LD_LIBRARY_PATH=/fullpath/mstrout/software/lib:$LD_LIBRARY_PATH
   ```

3. **Build CGPOP miniapp**

   ```
   ./build.linux
   ```

4. **Run CGPOP miniapp (takes about 2 minutes on an 8 core machine)**

   ```
   cd mpi2s1D                  // or another directory
   mpirun -np 24 ./cgpop.linux.180x120

   // The result should be within roundoff error of the following:
   CheckAnswers: MPI2S_1D
   Pressure field: SUM(calc-read):   -7.0421922819799061
   ```

The rest of the section provides more details about building and executing CGPOP on capability computing systems.

5.1 Building the tile files

Tile files include the input data for the CGPOP miniapp blocked into subdomains as well as how the subdomains relate. The appropriate tile file to use when executing CGPOP depends on how many nodes you intend to run it across. Generally, files that use smaller block-sizes are more appropriate when greater numbers of processors are used. For example, a block size 180x120 is recommended when using 358 processors, while a block size of size 120x80 is recommended when using 764 processors. A table of recommended tile sizes is included in `init/domain_size.F90`

The CGPOP version 1.0 release includes the 180×120 tile file (`data/cgpoptile_180x120.nc`), which divides the 3600 × 2400 global domain into 400 subdomains of size 180 × 120. Additionally other tile files are available for download from the CGPOP webpage at `http://www.cs.colostate.edu/hpc/cgpop`, or you can use the provided `cginit.f90` driver to generate the tile files.

Briefly, to generate tile files using the serial `cginit.f90` driver:

1. Setup your environment to use `netcdf` as shown at the beginning of this Section.
2. cd into the `init` directory
3. Set the ARCHDIR environmental variable. The currently supported options are `xt5_serial`, `bgl_fe`, and `linux`, which correspond to a single node of an XT5, a frontend node of a Blue Gene/L, or a generic linux machine that has the GCC compiler. For example, to set this environment variable in bash execute:

   ```
   export ARCHDIR="xt5_serial"
   ```

4. Run `make` to build the serial executable `cginit`.
5. Run the `cginit` executable.

When run, cginit will read in the POP state input file `../data/cgpopState.nc` and generate several tile files for different block sizes. Note that the `cgpopState.nc` file is currently in netCDF4 format. The size of the blocks is set by the following statements that describe three parallel arrays in the `cginit.F90` file:

```
integer(i4), parameter :: nsize = 8
integer, dimension(nsize) :: bsx,bsy,maxb
data bsx  /180, 120,  90,  60,  48,    36,   24,   18/
data bsy  /120,  80,  60,  40,  32,    24,   16,   12/
data maxb /400, 900,1600,3600,5625, 10000,22500,40000/
```

The cginit executable will create the following files

```
../data/cgpoptile_{$bsx}x{$bsy}.nc
```

where {`$bsx`} and {`$bsy`} are block sizes defined by data statements in `cginit.F90`. The {`$maxb`} data statement indicates the maximum number of blocks necessary when a given block-size is used (for example, with a block size of 180x120 the 3600x2400 domain is broken into 400 blocks arranged in a 20x20 grid; in practice fewer blocks are used because some areas correspond solely to land).

5.2 Building CGPOP Executables

There are several ways to build the CGPOP miniapp variants. Note that each driver needs to be built separately for each tile file configuration. We describe a method that simplifies the collection of parallel scaling curves. First, based on the machine you are running on, you need to determine the proper setting for the ARCHDIR environmental variable. The supported options are:

xe6Cray	Cray XE6 with Cray Compiler
xt5Cray	Cray XT5 with Cray Compiler
xt5Pgi	Cray XT5 with PGI compiler
bgl	Blue Gene/L with XLF compiler
linux	Standard Linux machine with GNU compiler

Each of these options corresponds to a `{$ARCHDIR}.gnu` file in each of the implementation's source directories. Additional .gnu files can be created in order to support new architectures. Note that versions that use CAF can only be compiled with the Cray compiler.

5.3 Details of build.{$hostname} scripts

We provide host specific build files that simplify the build and creation of scripts for running scaling curves on certain systems. The build scripts are of the form `build.$hostname`. Running one of these scripts will generate executables for the corresponding `$hostname` in one or more of the directories, `caf1D/, caf1D_sync_images/, caf2D/, mpi1s1D/, mpi2s1D/, mpi2s1D_overlap/, mpi2s2D`.

Another script will build the run scripts for the machine `{$hostname}` and is of the form `buildscript.{$hostname}`.

Inside a `build.{$hostname}` script there is a double nested foreach loop. The outer loop iterates over problem sizes (defined by a variable `szlist`); the inner loop iterates over versions of the CGPOP miniapp to compile (defined by the variable `mdlist`).

Note that `szlist` corresponds to the `{$bsx}x{$bsy}` values described previously. To build a number of CGPOP miniapp executables with different blocks sizes simple edit the `build.{$hostname}` script and execute it.

Build scripts for the following hosts are provided:

frost	A Blue Gene/L at NCAR
kraken	A Cray XT5 at NICS
lynx	A Cray XT5 at NCAR
hopper	A Cray XE6 at NERSC
linux	A standard Linux machine

5.4 Executing

There are two steps to executing CGPOP on a large capability system:

1. Running the appropriate build script
2. Submitting the appropriate script to your system's batch queue (for instance by using the `qsub` command)

The `buildscript.{$hostname}` script creates a number of job submission scripts for particular core counts. For example in the `buildscript.kraken`, we note that for block size 120×80, run scripts for 96, 192, 384, and 768 cores are created. These job scripts are placed in the directory `run/120x80` with the name `kraken.{96,192,384,768}.pbs` respectively. To execute the kraken scripts simply perform a `qsub` command on one or more of the scripts. Note that job output is written to the directory `run/results`. Configuration information is encoded into the name of the output file. An example output file is `cgpop_lynx1_mpi2s1D_xt5Pgi_60x40_576.log.101112-104405`. The name can be decoded in the following manner: CGPOP miniapp on hostname `lynx`, using the `mpi2s1D` version with `ARCHDIR=xt5Pgi`. The block sizes were 60×40 on 576 cores. The string of integers `101112-104405` correspond to a timestamp.

6 Evaluation as a Performance Proxy

One of the characterizations of a miniapp as put forth by Heroux et al. [6] is that a miniapp should be on the order of 1000 SLOC and include a simple build process to enable easy porting. This size and simplicity requirement is critical to the POP developers' need for a miniapp that enables benchmarking in immature environments. CGPOP satisfies these requirements in that each instance of the CGPOP miniapp is approximately 3000 SLOC and we have shown that the build process can be performed on a number of platforms. Additionally, to be of use to scientific application developers a miniapp should serve as a performance proxy for the target full application. To be considered a *performance proxy*, a miniapp should accurately model the performance bottleneck of the full application at the range of cores that the full application typically targets. In this section, we show that CGPOP is a performance proxy for POP by showing that the scalability of CGPOP's base MPI implementation is similar to that of POP.

Scalability can be affected by a number of factors including the machine and compiler used. To ensure that the performance behavior of the CGPOP miniapp matches that of POP we examined the scalability of CGPOP and POP across three different platforms: Hopper, a Cray XE6 located at the National Energy Research Supercomputing Center (NERSC); Frost, a BlueGene/L located at the National Center for Atmospheric Research (NCAR); Lynx, a Cray XT5 also located at NCAR; and Kraken, a Cray XT5 located at the National Institute for Computational Science (NICS). We list technical information about these compute platforms in Table 1. The compilers we used in our examination were PGI Fortran,

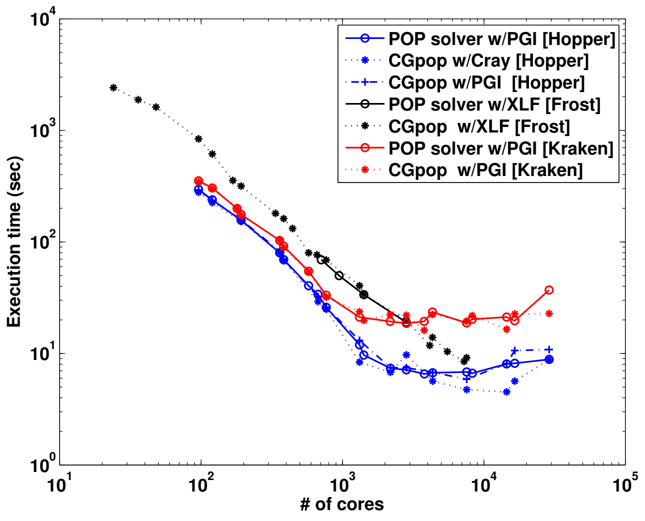

Figure 9: Execution time in seconds for 1 day of the barotropic component of the POP 0.1° benchmark and the MPI-2S version of the CGPOP miniapp on three different compute platforms

Cray Fortran, and XL Fortran. We present our results in Figure 9, and as can be seen by comparing the similarly colored lines for POP and CGPOP, the scalability behavior of the two are comparable when the same compiler and machine are used.

System				
Name	Kraken	Hopper	Lynx	Frost
Company	Cray	Cray	Cray	IBM
System Type	XT5	XE6	XT5	BG/L
# of cores	99,072	153,408	912	8192
Processor				
CPU	Opteron	Opteron	Opteron	PPC440
Mhz	2600	2100	2200	700
Peak Gflops/core	10.4	8.4	8.8	2.8
cores/node	12	24	12	2
Memory Hierarchy				
L1 data-cache	64 KB	64 KB	64 KB	32 KB
L2 cache	512 KB	512 KB	512 KB	2 KB
L3 cache	6 MB (shared)	12 MB (shared)	6 MB (shared)	4 MB (shared)
Network				
topology	3D torus	3D torus	2D torus	3D torus
# of Links/per node	6	6	4	6
Bandwidth/link	9.6 GB/s	26.6 GB/s	9.6 GB/s	0.18 GB/s

Table 1: Description of compute platforms used for this study.

For Kraken, the execution time of the barotropic section of POP and CGPOP matches well for all core counts. Note that considerable variability in execution time is observed on

Kraken for configurations with greater than 1,000 cores. This is an expected result of running an OS noise sensitive application on a shared production supercomputer. Additionally the loss of scalability in the CGPOP miniapp at large core counts on Kraken is a reflection of number of factors including the cost of global reduction within the solver. At greater than 4,000 cores on Kraken, approximately 50% of the execution time is due to a three-word global reduction.

On Frost the execution time for CGPOP and POP are also in close agreement. Not surprisingly the execution time of the CGPOP continues to scale on Frost, which is a Blue Gene/L system and therefore provides hardware support for global reductions.

On Hopper, the execution time of the POP barotropic solver and CGPOP for both the Portland Group (PGI) and Cray Compiler agree to within the variability of the system. Interestingly, the execution time for CGPOP using the Cray compiler appears to be depending on setting the following environment variable:

```
 export HUGETLB_DEFAULT_PAGE_SIZE=512K
```

This environment variable enables the use of 512 KByte pages. By default the Cray compiler uses 2 MByte large pages while the PGI compiler uses the much small 4 KByte pages. Preliminary performance timings on Hopper indicated that use of the Cray compiler had a significant negative impact on scalability at large core counts. Subsequent tests with 512 KByte pages revealed significantly smaller execution time for CGPOP. While this suggests that reducing the page size positively impacts performance by a factor of 5, the performance timings for the two different page sizes were not measured at the same time. Further because our access has been during the pre-acceptance phase of Hopper, which potentially involves modifications to the system OS, other system issues may be at play. We intend to closely work with Cray and the technical staff at NERSC to resolve these issues. Regardless of the outcome, this experience clearly demonstrates the utility of the CGPOP miniapp at helping to identify potential scalability issues within compilers.

7 Conclusions

Miniapps that serve as performance proxies can be used to evaluate and compare parallel programming models within the context of a larger application. In this paper we introduce and document CGPOP, a miniapp that models the conjugate-gradient solver in the Parallel Ocean Program (POP). We establish that CGPOP is a performance proxy for POP by comparing the scalability of POP and CGPOP on thousands and tens of thousands of cores on a Cray XT5, Cray XE6, and BlueGene/L. We argue that CGPOP can serve as a programmability proxy because it includes important code from the full application, is incremental in that a modified version of the miniapp can be reincorporated into the full application, and is representative of the implementation details that affect programmability in the full application.

Acknowledgements

This work was supported by Department of Energy Early Career Award #DE-SC3956. This work was financially supported through National Science Foundation Cooperative Grant NSF01 which funds the National Center for Atmospheric Research (NCAR), and through the grant: #OCI-0749206.

References

[1] Community Earth System Model.
`http://www.cesm.ucar.edu/`.

[2] NetCDF (Network Common Data Form) – website.
`http://www.unidata.ucar.edu/software/netcdf/`.

[3] E. F. D'Azevedo, V. L. Eijkhout, and C. H. Romine. Conjugate gradient algorithms with reduced synchronization overhead on distributed memory multiprocessors. Technical Report 56, LAPACK Working Note, August 1993.

[4] K. B. Ferreira, P. Bridges, and R. Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proc. of the 2008 ACM/IEEE Conf. on Supercomputing*, pages 1–12, 2008.

[5] W. Foster, P. H. Worley, and I. T. Foster. Parallel spectral transform shallow water model: A runtime–tunable parallel benchmark code. In *Proceedings of the Scalable High Performance Computing Conference*, pages 207–214. IEEE Computer Society Press, 1994.

[6] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[7] R. K. Rew and G. P. Davis. The unidata netCDF: Software for scientific data access. In *Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*, pages 33–40, Anaheim, California, American Meteorology Society, February 1990.