The Design of a Parallel Adaptive Multi-level Code in Fortran 90 *

William F. Mitchell

National Institute of Standards and Technology,
Gaithersburg, MD 20899
william.mitchell@nist.gov
http://math.nist.gov/~WMitchell

Abstract. Software for the solution of partial differential equations using adaptive refinement, multi-level solvers and parallel processing is complicated and requires careful design. This paper describes the design of such a code, PHAML. PHAML is written in Fortran 90 and makes extensive use of advanced Fortran 90 features, such as modules, optional arguments and dynamic memory, to provide a clean object-oriented design with a simple user interface.

1 Overview

Software for the solution of partial differential equations (PDEs) using adaptive refinement, multi-level solvers and parallel processing is complicated and requires careful design. This paper describes the design of such a code, PHAML (Parallel Hierarchical Adaptive Multi-Level). PHAML is written in Fortran 90 and makes extensive use of advanced Fortran 90 features, such as modules, optional arguments and dynamic memory, to provide a clean object-oriented design with a simple user interface.

The primary subroutine of PHAML solves a scalar, linear, second order elliptic PDE in two dimensions. More complicated PDE problems can be solved by multiple calls to the primary subroutine. This includes systems of equations, nonlinear equations, parabolic equations, etc. PHAML also provides for the solution of eigenvalue problems, like the linear Schrödinger equation.

The underlying numerical methods in PHAML are those used in the popular scalar PDE code MGGHAT [11], and are described in [9, 10]. PHAML is a finite element program that uses newest node bisection of triangles for adaptive refinement/derefinement [9] and a hierarchical multigrid algorithm [10] for solution of the linear system. The multigrid method can be used either as a solver or as a preconditioner for conjugate gradients [2] or stabalized bi-conjugate gradients [2].

Several other software packages are optionally used by PHAML to increase its capability. Operation in parallel requires the use of either MPI [5] or PVM [4]. Visualization is provided through OpenGL [17], GLUT [6] and f90gl [12].

PHAML contains one method for partitioning the grid for load balancing, but additional methods are available through Zoltan [3]. Eigenvalue problems require the use of ARPACK [8]. Some operations use BLAS [7] and LAPACK [1]; performance may be improved by using a vendor implementation rather than source code included with PHAML.

PHAML uses the concept of data encapsulation from object-oriented program design. Using Fortran 90 modules with public and private attributes, the user can only manipulate top level data types using only the functions provided by PHAML. This not only removes the burden of knowing the details of the data structures from the user, but it provides for the evolution of those data structures without any changes to the user code, improving upward compatibility of revisions of PHAML.

Simplicity of the user interface is also improved by using the optional argument and dynamic memory features of Fortran 90. The argument list for the primary subroutine is rather long, but nearly all the arguments are optional. The user only provides the arguments for which a specific value is desired; all other arguments assume a reasonable default value. Finally, the use of dynamic memory (allocatable arrays) removes the burden of allocating workspace of the correct size, which is often associated with FORTRAN 77 libraries.

PHAML can run as either a sequential or parallel program. As a parallel program it is designed for distributed memory parallel computers, using message passing to communicate between the processors. It uses a master/slave model of computation with a collection of compute processes that perform most of the work, graphics processes that provide visualization, and a master process that coordinates the other processes. The executable programs can be configured as three separate programs for the three types of processes, or as a single program used by all of the processes. The single program approach is used when all processes are launched at once from the command line, but limits some of multiple-PDE capabilities of PHAML. In the multiple program approach, the master process is launched from the command line and spawns the other processes.

A simplified version of the algorithm of the primary subroutine is

```
initialize coarse grid
repeat
   if (predictive) then load balance
   refine/derefine
   if (not predictive) then load balance
   solve linear system
until termination criterion is met
```

Note there is option of performing *predictive load balancing* before the refinement of the grid occurs, or load balancing the grid after refinement. The numerical methods have been modified for parallel execution using the *full domain partition* approach [13–15]. This approach minimizes the frequency of communication between the processors to just a couple messages for each instance of refinement, load balancing or multigrid, which reduces the amount of time spent on communication, especially in high-latency, low-bandwidth environments like a cluster. Load balancing is performed by partitioning the grid and redistributing the data so that each process owns the data associated with a partition. The k-way refinement-tree partitioning method (RTK) [16] is used by default.

2 Modules

The Fortran 90 module is fundamental to good software design in Fortran 90. A module is a program unit that can contain variables, defined constants (a.k.a. parameters), type definitions, subroutines, and other entities. Each entity in a module can be private to the module or public. The entities that are public are available to any program unit that explicitly uses the module. Modules have many uses. For example, modules can contain global data to replace the old-style common blocks, contain interface blocks for an external library to provide strong type checking, or group together a collection of related subroutines.

One of the most important uses of modules is to provide data encapsulation, similar to a class in C++. The module contains one or more user defined types (a.k.a. structures) and functions that operate on those types. The type itself is made public, so that other program units can declare variables to be of that type, but the internals of the type are declared private, so that nothing outside the module can operate on the individual components of the type. For example

```
public hash_key
type hash_key
   private
   integer :: key
end type hash_key
```

Some of the functions in the module are made public to provide the means of operating on variables of the public type.

PHAML is organized into several modules, each of which contains either the structures and operations for some aspect of a parallel adaptive multi-level code, or global entities. The primary modules are:

`phaml`: This is the only module used directly by the user's code. It makes public the entities that the user needs, and contains all the functions that are directly callable by the user.

`linear_system`: This contains all data structures and operations related to creating and solving the linear system.

`grid`: This contains all operations on the grid, including refinement and redistribution.

`grid_type`: This contains the grid data type. It is separate from module `grid` because other modules also need access to this structure. For example, module `linear_system` needs to use the grid to create the linear system.

`load_balance`: This contains subroutines for partitioning the grid.

message_passing: This contains subroutines for communication between processes. It acts as an interface between a message passing library and the rest of the PHAML code.

hash: This contains operations on a hash table, which translates global IDs (known to all processes) to local IDs (known only to one process).

global: This contains global data which can be used by any program unit.

3 Data Structures

PHAML defines many data structures, most of which are private to a module or have private internals. A complete description of even the main structures is beyond the scope of this paper, and would be fruitless since they continue to evolve as the capabilities of PHAML expand. This section illustrates the flavor of the data structures through a snapshot of the current state of a few of them.

The only data type available to the user is phaml_solution_type, defined as

```
type phaml_solution_type
   private
   type(grids_type) :: grids
   type(proc_info) :: procs
   integer :: outunit, errunit, pde_id
   character(len=HOSTLEN) :: graphics_host
   logical :: i_draw_grid, master_draws_grid, &
              i_draw_reftree, master_draws_reftree, &
              still_sequential
end type phaml_solution_type
```

This structure contains all the information that one processor knows about the computed solution, grid and processor configuration for a PDE. See Sect. 4 for the operations that can be performed on a variable of this type. The first component contains the grid information. type(grids_type) contains the grid(s) corresponding to one or more partitions of the global grid. It allows for more than one partition to be assigned to a processor for possible future expansion to shared memory parallelism and/or cache-aware implementations. type(proc_info) contains information about the processor configuration for message passing. It is defined in module message_passing with private internals, and it's components depend on the message passing library in use. For example, the PVM version contains, among other things, the PVM task ids, while the MPI version contains the MPI communicators.

A slightly reduced version of the grid data type is

```
type grid_type
   type(element_type), pointer :: element(:)
   type(node_type), pointer :: node(:)
   type(hash_table) :: elem_hash, node_hash
   integer :: next_free_elem, next_free_node
```

```
   integer, pointer :: head_level_elem(:), head_level_node(:)
   integer :: partition
   integer :: nelem, nelem_leaf, nelem_leaf_own, nnode, &
              nnode_own, nlev
end type grid_type
```

The first two components are arrays containing the data for each element and node of the grid. These are allocatable arrays (the pointer attribute is used because Fortran 90 does not have allocatable structure components, but does allow a pointer to an array to be allocated), which allows them to grow as the grid is refined. The next two components are the hash tables, which are used for converting global IDs to local IDs. A global ID is a unique identifier for every element and node that may be created through refinement of the initial grid, and is computable by every processor. Global IDs are used for communication about grid entities between processors. The local ID is the index into the array for the element or node component. The next four components provide linked lists to pass through the elements or nodes of each refinement level. `partition` indicates which partition of the global grid is contained in this variable. Finally, the remaining scalars indicate the size of the grid and how much of it is owned by this partition.

Examining one level further, the data type for a node is given by

```
type node_type
   type(hash_key) :: gid
   type(point) :: coord
   real :: solution
   integer :: type, assoc_elem, next, previous
end type node_type
```

4 User Interface

The user interface to PHAML consists of two parts: 1) external subroutines written by the user to define the PDE problem, and 2) the PHAML subroutines that operate on a `phaml_solution_type` variable to solve the PDE problem.

The user must provide two external subroutines, `pdecoef` and `bcond`, to define the differential equation and boundary conditions, respectively. For problems involving the solution of more than one PDE, multiple interdependent PDEs can be defined in these subroutines, using the global variable `my_pde_id` to determine which one should be evaluated. An example of subroutine `pdecoef` is

```
subroutine pdecoef(x,y,p,q,r,f)

! pde is
!  -( p(x,y)*u ) -( q(x,y)*u ) +r(x,y)*u = f(x,y)
!             x x           y y
```

```
real, intent(in) :: x(:),y(:)
real, intent(out), optional :: p(:),q(:),r(:),f(:)

if (present(p)) p = 1.0
if (present(q)) q = 1.0
if (present(r)) r = 0.0
if (present(f)) f = x**2 + y**2

end subroutine pdecoef
```

Note that the arguments are arrays. This allows PHAML to call the subroutine with many quadrature points to reduce the overhead of calling it many times with one point. But, with Fortran 90's array syntax, in most cases the assignment can be done as a whole array assignment and look the same as the corresponding code for scalars. Also, the return arguments are optional, so the user must check for their existence with the intrinsic subroutine `present`. This allows PHAML to avoid unnecessary computation of coefficients that it does not intend to use at that time.

The user must also provide a subroutine to define the initial grid, which also defines the polygonal domain. At the time of this writing, an example is provided for rectangular domains, but it is difficult for a user to write the subroutine for more complicated domains. It is hoped that in the future PHAML will interface to a public domain grid generation code to define the initial grid.

Optionally, the user may also provide a subroutine with the true solution of the differential equation, if known, for computing norms of the error.

The user provides a main program for the master process. This program uses module `phaml`, and calls the public PHAML subroutines to perform the desired operations. The simplest program is

```
program user_main_example
use phaml
type(phaml_solution_type) :: pde
call create(pde)
call solve_pde(pde)
call destroy(pde)
end program
```

At the time of this writing there are nine public subroutines in module `phaml`. It is not the purpose of this paper to be a user's guide, so only a brief description of the function of the routines is given, except for the primary subroutine where some of the arguments are discussed.

`create, destroy`: These two subroutines are similar to a constructor and destructor in C++. Any variable of type `phaml_solution_type` must be passed to `create` before any other subroutine. Subroutine `create` allocates memory, initializes components, and spawns the slave and graphics processes. Subroutine `destroy` should be called to free memory and terminate spawned processes.

`solve_pde`: This is the primary subroutine, discussed below.

`evaluate`: This subroutine is used to evaluate the computed solution at a given array of points.

`connect`: With multiple PDEs, each one has its own collection of slave processes (see Sect. 5). For interdependent PDEs, these processes must be able to communicate with each other. This subroutine informs two `phaml_solution_type` variables about each other, and how to communicate with each other.

`store, restore`: These routines provide the capability of saving all the data in a `phaml_solution_type` variable to disk, and restoring it from disk at a later time.

`popen, pclose`: These routines provide parallel open and close statements, so that each process opens an I/O unit with a unique, but similarly named, file. This is used for the files in `store` and `restore`, and the `output_unit` and `error_unit` arguments to subroutine `create`.

Subroutine `solve_pde` is the primary public subroutine of PHAML. All the work of grid refinement and equation solution occurs under this subroutine. At the time of this writing it has 43 arguments to provide flexibility in the numerical methods and amount of printed and graphical output. All of these arguments are optional with reasonable default values, so the calling sequence need not be more complicated than necessary. Usually a user would provide them as keyword arguments (the name of the argument is given along with the value), which improves readability of the user's code. For example

```
call solve_pde(pde, &
               max_node = 20000, &
               draw_grid_when = PHASES, &
               partition_method = ZOLTAN_RCB, &
               mg_cycles = 2)
```

For many of the arguments, the acceptable values are given by defined constants (for example, `PHASES` and `ZOLTAN_RCB` above) which are public entities in module `phaml`.

Some of the arguments to `solve_pde` are:

`max_elem, max_node, max_lev, max_refsolveloop`: These are used as termination criterion.

`init_form`: This indicates how much initialization to do. `NEW_GRID` starts from the coarse grid, `USE_AS_IS` starts the refinement from an existing grid from a previous call, and `SOLVE_ONLY` does not change the grid, it just solves the PDE on the existing grid from a previous call.

`print_grid_when, print_grid_who`: These determine how often summary information about the grid should be printed, and whether it should be printed by the `MASTER, SLAVES` or `EVERYONE`. There are also similar arguments for printing of the error (if the true solution is known), time used, and header and trailer information.

`uniform`, `overlap`, `sequential_node`, `inc_factor`, `error_estimator`, `refterm`, `derefine`: These arguments control the adaptive refinement algorithm.

`partition_method, predictive`: These arguments control the load balancing algorithm.

solver, preconditioner, mg_cycles, mg_prerelax, mg_postrelax, iterations: These arguments control the linear system solver algorithm.

5 Parallelism

PHAML uses a master/slave model of parallel computation on distributed memory parallel computers or clusters of workstations/PCs. The user works only with the master process, which spawns the slave processes. PHAML also provides for sequential execution and for spawnless parallel execution, but this section assumes the spawning form of the program.

The parallelism in PHAML is hidden from the user. One conceptualization is that the computational processes are part of a phaml_solution_type object, and hidden like all the other data in the object. In fact, one of the components of the phaml_solution_type structure is a structure that contains information about the parallel processes. The user works only with the master process. When the master process calls subroutine create to initialize a phaml_solution_type variable, the slave processes are spawned and the procs component of the phaml_solution_type variable is initialized with whatever information is required for the processes to communicate. If another phaml_solution_type variable is initialized, a different set of slave processes are spawned to work on this one. When the master process calls any of the other public subroutines in module phaml, it sends a message to the slaves with a request to perform the desired operation. When the operation is complete, the slave waits for another request from the master. When subroutine destroy is called from the master process, the slave processes are terminated.

PHAML was written to be portable not only across different architectures and compilers, but also across different message passing means. All communication between processes is isolated in one module, message_passing. This module contains the data structures to maintain the information needed about the parallel configuration, and all operations that PHAML uses for communication, such as comm_init (initialization), phaml_send, phaml_recv, phaml_global_max, etc. Thus to introduce a new means of message passing, one need only write a new version of module message_passing that adheres to the defined API. PHAML contains versions for PVM, MPI 1 (without spawning), MPI 2 (with spawning), and a dummy version for sequential programs.

6 Conclusion

This paper described the software design of PHAML, a parallel program for the solution of partial differential equations using finite elements, adaptive refinement and a multi-level solver. The program is written in Fortran 90 and makes heavy use of modules for program organization and data encapsulation. The user interface is small and makes use of optional and keyword arguments to keep the calling sequence short and readable. The parallelism is hidden from the user, and portable across different message passing libraries.

The PHAML software has been placed in the public domain, and is available at *URL to be determined.*

References

1. Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., Sorensen, D.: *LAPACK Users' Guide*, SIAM, Philadelphia, 1982
2. Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., Van der Vorst, H.: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994
3. Boman, E., Devine, K., Hendrickson, B., Mitchell, W. F., St. John, M., Vaughan, C.: Zoltan: A dynamic load-balancing library for parallel applications, user's guide, Sandia Technical Report SAND99-1377 (2000)
4. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Snderam, V.: *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, 1994
5. Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M.: *MPI: The Complete Reference*, MIT Press, Cambridge, MA, 1998
6. Kilgard, M.: The OpenGL Utility Toolkit (GLUT) programming interface API version 3, http://www.opengl.org (1996)
7. Lawson, C. L., Hanson, R. J., Kincaid, D., Krogh, F. T.: Basic Linear Algebra Subprograms for FORTRAN usage, ACM Trans. Math. Soft. **5** (1979) 308–323
8. Lehoucq, R. B., Sorensen, D. C., Yang, C.: *ARPACK Users' Guide*, SIAM, Philadelphia, 1998
9. Mitchell, W. F.: Adaptive refinement for arbitrary finite element spaces with hierarchical bases, J. Comp. Appl. Math. **36** (1991) 65–78
10. Mitchell, W. F.: Optimal multilevel iterative methods for adaptive grids, SIAM J. Sci. Statist. Comput. **13** (1992) 146–167
11. Mitchell, W. F.: MGGHAT user's guide version 1.1, NISTIR 5948 (1997)
12. Mitchell, W. F.: A Fortran 90 interface for OpenGL: Revised January 1998, NISTIR 6134 (1998)
13. Mitchell, W. F.: The full domain partition approach to distributing adaptive grids, Appl. Num. Math. **26** (1998) 265–275
14. Mitchell, W. F.: The full domain partition approach to parallel adaptive refinement, in *Grid Generation and Adaptive Algorithms, IMA Volumes in Mathematics and it Applications* **113** Springer-Verlag (1998) 151–162
15. Mitchell, W. F.: A parallel multigrid method using the full domain partition, Elect. Trans. Num. Anal. **6** (1998) 224–233
16. Mitchell, W. F.: The refinement-tree partition for parallel solution of partial differential equations, NIST J. Res. **103** (1998) 405–414
17. Woo, M, Neider, J., Davis, T., Shreiner, D.: *The OpenGL Programming Guide*, Addison-Wesley, 1999