In *Proc. 10th SIAM Conf. Parallel Processing for Sci. Computing* (2001).

Parallel Adaptive Blocks on a Sphere*

Robert Oehmke and Quentin F. Stout

Abstract

We have developed a flexible tool for efficient parallel adaptive mesh refinement on a sphere. Adaptive mesh refinement allows one to concentrate computational resources on regions of interest, and by extending to spheres we have extended their applicability to climatology and other large scale planetary phenomena. The system is effective for our space weather application and as a general purpose object-oriented tool.

1 Introduction

Adaptive mesh refinement is an important technique for saving computational resources when modeling multi-scale phenomena. Rather than using a uniform grid, which would have to be at the resolution of the smallest feature of interest, adaptive grids permit one to use a fine resolution where it is needed, and a coarser resolution elsewhere. This can result in significant time and space savings over simply using a uniform grid. In this paper we present a new variation on this technique: flexible, efficient adaptive refinement on a sphere. Further, our system can provide adaptation on far more general shapes.

We were led to investigate this subject by our involvement in the CSEM project to predict geomagnetic storms [4]. These space weather events, which can damage satellites, astronauts, and power lines, are caused by solar coronal mass ejections. To predict these storms, a simulation involving the heliosphere and several different components of the earth's atmosphere is performed. Modeling the earth's atmosphere necessitated our sphere-based adaptive blocks. Although developed for a specific purpose, this tool could be used to help model large scale planetary phenomena such as climatology or geology.

*Work supported by NSF KDI grant ATM-9980078 and by the Center for Parallel Computing at the University of Michigan.

2 Previous Work

There has been some previous work in parallel adaptive blocks for Cartesian grids. The Cartesian system most similar to our's is BATS-R-US. This project, in which we participated, developed adaptive blocks to model the heliosphere [8]. We used many of the lessons learned on that project to improve this one.

Another Cartesian adaptive block system similar to ours is PARAMESH [6]. There are also some parallel adaptive Cartesian systems which differ from our system more substantially. These include Chombo [3], Dagh [5], Samrai [7], and Kelp [1].

There has also been some other work on adaptive non-rectangular grids. Overture [2], for example, could be used for adaptive mesh refinement on a sphere, but it differs greatly from our project in its mesh structure and implementation of adaption. Overture's meshing of a given object consists of a collection of meshes containing differing numbers of data cells. In our system, in contrast, every block consists of the same number of data cells no matter what their actual spatial size. Also, Overture's adaptation is patch based so that when a region is being refined a new finer mesh is generated which covers the whole region. In our system when a region is being refined all the blocks which currently represent that region are simply broken into four spatially smaller blocks, each containing the same number of data cells as their parent.

3 System Capabilities

The system consists of several flexible components. The sphere is initially gridded as a collection of blocks. The grid has a variety of initial configurations and a user-specified structure at each cell. As refinement or coarsening occurs, the system maintains the adjacency information and provides functions for the user to iterate through all of the blocks and to transfer data efficiently between the blocks.

3.1 Block Structure

Each *block* is a 2-dimensional rectangular array of cells, where the extents of each dimension, and the extents of the ghost cells, are specified by the user. Each block contains the same number of cells no matter how often they have been refined or coarsened. This structure of multiple cells per block, where all blocks have the same structure, provides several efficiencies. For example, the inner loops of the user code operate on an array of fixed structure, which allows various compiler and cache optimizations. Further, the cost of operations such as access and communication are amortized over an entire block, as opposed to a single cell. Thus adaptive blocks are far more efficient than standard quadtrees.

The structure of the blocks is quite flexible, in that the user can attach any type of data in any format to each cell. In particular, this permits user-defined gridding in the altitude. Besides allowing users to incorporate whatever data their application requires, they can also format it to achieve maximum efficiency. For example, a block could be an array of records, or a record of arrays. The user need only provide the appropriate functions to manipulate their data, and our system does the rest. User functions are needed to manipulate ghost regions, including coalescing and interpolating cells at boundaries between blocks of differing resolutions.

Blocks are organized into objects called *block groups*. Every block is a member of exactly one block group. The block group allows a set of blocks to be operated on as a whole. In our system there is, in general, a one-to-one correspondence between a shape and a block group.

3.2 Sphere Structure

The sphere structure is also quite flexible. It may contain any number of blocks. A sphere's poles may be either "on" or "off" independently, allowing the creation of "half" spheres with only one pole or "rings" with none. Also, the spatial distribution of blocks across the sphere is user definable which, in addition to adaptation, is another method to focus computational resources where needed. Note that beyond the creation function our system does not depend on the shape of the region being gridded, so by simply writing a new creation function, it can be extended to operate on a different shape. Adaptation is only in two dimensions, with the structure in other dimensions fixed and specified by the user. However, the same system structure described below can easily be extended to arbitrary dimensions.

4 System Structure

The system is implemented in Fortran 90, and currently uses MPI for communication, so it is quite portable. It is designed using a layered approach where each layer is composed of two modules. All the modules are constructed to have a well defined interface to the rest of the system, which allows our system to be quite flexible. By replacing a module with another with an equivalent interface the system can be given entirely new functionality, or better implementations of old functionality. The system is divided into three layers: the base layer, the block-move layer, and the grid layer.

4.1 The Base Layer

The base layer contains the most basic functionality upon which the rest of the system depends. It is composed of two modules: communication and block.

The *communication module* handles the transfer of system data between the processors. The rest of the system uses it to do communication. This functionality is separated into its own module to allow the communication methods to be easily changed. Currently our communication is done using MPI, but because of our approach we could easily switch to another mechanism, such as OpenMP. This module is never directly accessed by the user.

The *block module* provides all the basic functionality for operating on blocks and block groups. This module is responsible for maintaining the block connection information, and whether a block is currently in use. It also implements the ability to iterate through active blocks.

4.2 The Block-Move Layer

This layer is composed of the transfer and the load-balancing modules.

The *transfer module* allows the user to transfer their data between blocks. It is responsible for maintaining the proper buffers and deciding what goes where. Using the transfer module is advantageous for the user because it allows the transfer to be done transparent to the block location and connections. All messages sent from one processor to another are accumulated, rather than sending one per block, which results in much greater efficiency.

The *load-balancing module* is responsible for distributing the blocks among the processors. Currently this module just strives to maintain an equal number of blocks on all processor, however, in the future we will use a more sophisticated criteria for distribution (see Section 7). This module is not directly accessed by the user, but the sphere creation and adaptation modules both use its functionality.

4.3 The Grid Layer

This layer implements operations that operate on the entire non-uniform grid that our system provides to the user. This layer contains the adaptation and sphere modules.

The *adaptation module* is responsible for implementing the coarsening and refinement of the blocks. This module maintains the information from the user about which blocks should be refined or coarsened. It is also responsible for enforcing restrictions on adaptation, e.g. many codes require that neighboring blocks differ by no more than one refinement level. This module does much of the work for adaptation, using the block module to actually rearrange the block connections and the load balancing module to distribute its new blocks.

The *sphere module* is responsible for creating the sphere shaped connected group of blocks and for maintaining the geometric information describing the sphere. Currently our system only implements the sphere, however, by writing a new module similar to this one an entirely different shape could be easily created. This module calls the block module to connect the blocks into a sphere, and the load balancing module to distribute its new blocks.

5 API

This section gives an overview of the major user functions in the adaptive block API. The subroutines are grouped by the module. The first part of the subroutine names also encode their home module (e.g., subroutines whose names start with AB_BLK belong to the block module). We also include a section which describes the subroutines the user must provide to the system. Note, not every module is represented because not every module has user visible functions. Due to the limited scope of this paper we only sketch the subroutine behavior. A more complete description will accompany the public release of our system.

Another aspect of the API is that the routines can be easily instrumented. To achieve this, the subroutine that the user calls, for example **`AB_ADPT_do`**, calls another subroutine that actually implements the functionality, **`AB_ADPT_do_actual`**. This allows one to redefine the user called subroutines to slip measurement code between the user call and the actual implementation of the subroutine functionality. This is the method used in MPI.

5.1 User Provided Subroutines

These are the subroutines that the user must provide for the system to be able to do inter-block communication and adaptation. Because the block structure can be arbitrary, any operation changing block contents must be supplied by the user. Note that this allows the user to control the interpolation and coalescence operations.

`user_pack_data:` Pack the ghost data for transport to the neighboring block at the same refinement level.

`user_pack_data_split:` Here ghost data is moving from a block to two finer neighboring blocks, so the user must split the data and pack it.

`user_unpack_data:` Unpack the ghost data and insert it into the block.

`user_unpack_data_join:` Here ghost data has been transported from two blocks to a coarser neighbor. Merge the data and insert it into the block.

`user_refine:` Split the block into four sub-blocks.

`user_coarsen:` Merge the four blocks into one block.

5.2 Block Module Subroutines

Note that there isn't a creation subroutine for the blocks themselves, because they are created automatically when an adaptive block group is created. Also, the user never actually deals with the block objects themselves as they are always referred to by index.

`AB_BLK_GRP_create:` Create an object which holds a group of blocks.

`AB_BLK_GRP_destroy:` Remove a group of blocks from the system.

`AB_BLK_ITER_create:` Create an iteration object associated with a block group.

`AB_BLK_ITER_reset:` Reset the iteration object to the beginning of the list of blocks in the associated block group.

`AB_BLK_ITER_next:` Move to the next block in this block group. Returns a flag when the last block in the group has been reached.

5.3 Transfer Module Subroutines

These functions implement a transfer of data between each block and its neighbors.

`AB_XFER_create:` Create a transfer structure associated with a block group. For this subroutine the user specifies the size of ghost data transported between neighbors in every direction, so that proper buffers can be pre-allocated.

`AB_XFER_destroy:` Remove a transfer object from the system.

`AB_XFER_start:` Start a non-blocking transfer between each block and its neighbors, using the user pack routines.

`AB_XFER_finish:` Finish a transfer, using the user unpack routines.

By breaking the communication into start and finish functions, the user may be able to perform calculations during the communication and thus hide some of the cost.

5.4 Adaptation Module Subroutines

AB_ADPT_set_adapt: Mark a block to be coarsened or refined. Currently this subroutine accepts three values: 1, 0, or -1, where 1 indicates the block should be refined, 0 indicates no change and -1 indicates coarsening. We intend to expand this range to indicate the priority of the block's adaptation (see Section 7).

AB_ADPT_do: Do the adaption indicated by the adaption values set by the previous subroutine. This routine also takes as input the user coarsen and refine subroutines.

5.5 Sphere Module Subroutines

AB_SPH_create: Create a sphere object associated with a block group. The user can specify various geometric parameters such as center and radius. The user also specifies some of the connection properties of the sphere, for example if it is connected across the poles.

AB_SPH_destroy: Remove a sphere object from the system.

AB_SPH_get_xyz: Return the rectangular coordinates of a specified data cell in a given block.

AB_SPH_get_sphr: Return the spherical coordinates of a data cell in a given block.

6 Programming Model

One of the goals of our system was to allow the user as much flexibility as possible in the structure of their data. Thus we allow the user to be entirely in charge of their data so they can structure it however they feel is best. This also allows them to take advantage of efficiencies which might not be possible when a system manages the user data. For example, they can use statically allocated arrays. In order for our system to refer to a block we simply use an index, which is more flexible than using a pointer. A pointer would force each block's data to be in a contiguous section of memory, or another level of indirection would be needed. Instead, with the index, if the data is in arrays, as it is in many scientific codes, then it can all be in one array with one of the indices the block index, or it can be in separate arrays, or any combination in between. If the data is in a more exotic pointer-based data structure then an array mapping the index to the pointer can be used.

To use our adaptive blocks, the first step, after calling a couple of setup routines, is to call the block group creation subroutine (**AB_BLK_GRP_create**). This routine takes as input the group's maximum allowed number of blocks on the calling processor and returns an empty block group object. Next the sphere creation routine, **AB_SPH_create**, is used to create a sphere object, i.e., the block group now has a set of blocks with connection and geometric information but no user data.

To loop through a sphere's blocks on a processor, first **AB_BLK_ITER_create** is called to create an iteration object. One can then iterate through the sphere's block indices by using **reset** and **next** subroutines which operate on the iteration object. The indices used may change as adaptation occurs but these details are hidden from the user, while still providing systematic access to the blocks.

To transfer information between blocks, the user first calls **AB_XFER_create** to create a transfer object from the block group object. **AB_XFER_start** can then be used on the transfer object to initiate a non-blocking communication, which is completed by **AB_XFER_finish**. Because the user can have their data in any format they choose, they must provide functions for packing and unpacking their data.

To do adaptation, the user first uses **AB_ADPT_set_adapt** to mark blocks for refinement or coarsening, and then calls **AB_ADPT_do** to do the adaptation. Because we do not know how the user's data is arranged, the user must supply routines to split or merge blocks.

When the user is finished with the sphere and associated objects they can call the various deletion routines to remove them from the system.

7 Future Work

There are several directions in which we intend to advance our work in the future.

One of these is to develop modules to implement non-spherical shapes. More generally it would be useful to add the functionality for creating new shapes without the addition of a new subroutine for each, for example by implementing a module which takes as input a data structure indicating the connections between a group of blocks and then creates the shape.

Another area we intend to pursue is more intelligent block distribution. Currently our system just attempts to maintain an equal number of blocks on each processor without regard to how the blocks are connected. While this achieves good load-balance it can increase the amount of interprocessor communication, because neighboring blocks are often not assigned to the same processor. This is not usually a serious problem because the communication overhead is amortized over each block, but in the future we intend to implement a dynamic block distribution scheme which attempts to put neighboring blocks on the same processor, in addition to balancing the numbers of blocks.

An additional goal is to do prioritized adaptation. Currently the user is only able to indicate whether or not a block should be adapted. In the future we would like to allow the user to be able indicate how crucial a block's adaptation is, since the total number of blocks that can be created by adaptation is limited by memory size. Currently if we do not have enough blocks to perform the user's adaptation request we return an error. We would like to allow the user to assign a priority to blocks which provide a basis for choosing which of a limited number of blocks to refine. This would be very useful for codes which are limited by memory, but is much more complicated to implement when there are restrictions on the relative refinement levels of neighbors. For example, if neighbors can only differ by one level, then refining a fine block forces a neighboring coarse block to also refine.

Another aspect we would like to implement is a method for the inclusion of user information in the block structure for the system to use. For example, suppose the amount of work varied per block. If the user could supply an estimate of the work required by each block, this could be used by load balancing routines. These routines would have be supplied by the user, but perhaps this would be a common enough concern that it would become a standard module. There would have to be a user function for setting the weights and system routines for accessing them, and the block header structure would have to include them. We

would like to develop a general system for the user to be able to choose to include various types of infrastructure at compile time, using a system more sophisticated and cleaner than the normal preprocessor directives.

There are also some new grid-level operations which we would like to add to the system in the future. An example is one which would allow the user to project a regular grid of arbitrary resolution onto our non-uniform grid. An iterator would be provided which would allow the user to step through all the points in their grid contained in the blocks on a given processor. The iterator would return the block and cell in which their grid point was contained. This type of operation would be very useful for graphics and for mapping data onto the grid.

Another potential future operation is one which given a spatial location would return the block and cell containing that location. This would be useful for users trying to interpolate values at specific spatial locations.

We will also eventually provide the functionality for linking blocks (possibly in different groups) so that the adaptation of one triggers the adaptation of the other. This functionality is useful if you have two models which are touching and it is required that the parts that touch have the same resolution.

Lastly, we would like to create interfaces between our system and common science frameworks to maximize the value of our system to the scientific community.

Bibliography

[1] Baden, S.B. (1996), "Software infrastructure for non-uniform scientific computations on parallel processors", *ACM Applied Computing Review* **4(1)**, pp. 7–10

[2] Bassetti, P., et al. (1998), "Overture: an object-oriented framework for high performance scientific computing", *Proceedings of ACM/IEEE SC98*, 9p.

[3] seesar.lbl.gov/anag/chombo/

[4] Clauer, R., et al. (2000), "High performance computer methods applied to predictive space weather simulations", *IEEE Trans. Plasma Sci.*, in press.

[5] www.npac.syr.edu/projects/bh/dagh.html

[6] MacNeice, P., et al. (2000), "PARAMESH: a parallel adaptive mesh refinement community toolkit", *Computer Physics Communications* **126**, pp. 330–354.

[7] www.llnl.gov/casc/SAMRAI/

[8] Stout, Q.F., De Zeeuw, D.L., Gombosi, T.I., Groth, C.P.T., Marshall, H.G., and Powell, K.G., (1997), "Adaptive blocks: A high-performance data structure", *Proc. SC'97*.