Efficient Parallelization for AMR MHD Multiphysics Calculations; Implementaion in AstroBEAR

Jonathan J. Carroll-Nellenback[a], Adam Frank[a], Brandon Shroyer[a], Chen Ding[b]

[a] Department of Physics and Astronomy, University of Rochester, Rochester, NY 14620
[b] Department of Computer Science, University of Rochester, Rochester, NY 14620

Abstract

Current AMR simulations will require algorithms that are highly parallelized and manage memory efficiently. We have attempted to employ new techniques to achieve both of these goals. Patch based AMR often employs ghost cells to decouple the hyperbolic advances of each patch or grid. This decoupling allows each grid to be advanced independently. In AstroBEAR we utilize this independence to allow grids across multiple refinement levels to be advanced independently with preference going to the finer level grids. This allows for global load balancing instead of level by level load balancing and allows for greater parallelization across both physical space and AMR level which can improve performance in deep simulations with many levels of refinement. To improve memory management we have employed a distributed tree algorithm so processors no longer need to store the entire tree structure. We have also implemented a sweep method that pipelines the computations required for updating the fluid variables using unsplit algorithms. This can dramatically reduce the memory overhead required for intermediate variables.

1. Introduction

Larger and larger clusters alone will not allow larger and larger simulations to be performed in the same wall time without either an increase in CPU speed or a decrease in the workload of each processor. Since CPU speeds are not expected to keep pace with the requirements of large simulations, the only option is to decrease the work load of each processor. This however requires highly parallelized and efficient algorithms for managing the Adaptive Mesh Refinement (AMR) infrastructure and the necessary computations. AstroBEAR, like many other grid based AMR codes, utilizes a nested tree structure to organize each individual refinement region. However unlike many other AMR codes (Paramesh, BoxLib), AstroBEAR uses a distributed tree in which no processor has access to the entire tree but rather each processor is only aware of the AMR structure it needs to be aware of in order to carry out its computations and perform the necessary communications. While currently, this additional memory is small compared to the resources typically available to a CPU, future clusters will likely have much less memory per processor similar to what is already seen in GPU's.

Email address: johannjc@pas.rochester.edu (Jonathan J. Carroll-Nellenback)

Preprint submitted to Elsevier *May 31, 2011*

AstroBEAR also utilizes interlevel threading to allow advances to occur on different levels independently. This allows for total load balancing across all refinement levels instead of balancing each level independently. This becomes especially important for deep simulations (simulations with low filling fractions but many levels of AMR) as opposed to shallow simulations (high filling fractions and only a few levels of AMR). Processors with coarse grids can advance their grids simultaneously while processors with finer grids advance theirs. Without this capability, base grids would need to be large enough to be able to be distributed across all of the processors. For simulations with large base grids to be able to finish in a reasonable wall time, only a few levels of AMR can often be used. With interlevel threading this restriction is lifted. In section 3 we will discuss the distributed tree algorithm, in section 4 we will discuss the interlevel threading of the advance, in section 5 we will discuss the load balancing algorithm and in section 6 we will discuss the pipelining of the unsplit integration schemes. We will also briefly discuss attempts to further reduce locally redundant advance computations in section 7 and in section 8 we will present our scaling results and we will conclude in section 9.

2. AMR algorithm

Here we give a brief overview of patch based AMR introducing our terminology along the way. The fundamental unit of the AMR algorithm is a patch or grid. Each patch contains a regular array of cells in which the fluid variables are stored. Patches with a common resolution or cell width Δx_l belong to the same level l and on all but the coarsest level are always nested within a coarser "parent" patch of level $l-1$ and resolution $\Delta x_{l-1} = R \times \Delta x_l$ where R is the refinement ratio. After each generation of level l patches are created they initialize their cells with a combination of prolongated data from their parent patch as well as data from the preceding set of level l patches. They then determine which cells to refine and then lay down their first set of child patches to cover those cells. They then take one step of Δt_l and then wait until their child patches advance R steps of $\Delta t_{l+1} = \Delta t_l / R$. They then merge the restricted data from their child patches with their own updated data before synchronizing fluxes and emfs with any adjacent neighboring level l patches. They then continue to apply overlapping data (although now the data comes from neighbors and the overlapping regions lie only in the ghost zones) before laying down the successive generation of children, advancing a time step, waiting for child advances to complete, applying restricted data from children, and synchronizing fluxes with their neighbors until they have completed R steps at which point they restrict their data to be applied to their parent patch. They then wait for the next generation of level l patches to be created at which point there data is copied onto their succeeding overlaps after which they are destroyed. See figure 1 for a graphical representation. Throughout a patches lifetime it must therefore share data with its parent patch, preceding overlaps, multiple generations of children patches, neighbor patches, and succeeding overlaps. These connections between patches form the AMR tree.

3. Distributed Tree Algorithm

Many current AMR codes store the entire AMR tree on each processor. This, however, can become a problem for simulations run on many processors. If we assume that each

AMR patch requires m bytes to designate its meta data (ie physical bounds and the processor containing its data), and that each patch requires on average d bytes for the actual data - and that their are on average n patches on each of p processors, then the memory per processor would be $nd + nmp$. The memory requirement for the AMR heirarchy metadata becomes comparable to the local actual data when $p = d/m$. If we assume a 3D isothermal hydro run with an average patch size of 8x8x8 then $p = \frac{8\times8\times8\times4}{(6+1)} \approx 293$. While this additional memory requirement is negligible for problems run on typical cpus on 100's of processors, it can be become considerable on 1000's or 10000's of processors. Since it is expected that efficient memory use and management will be required for HPC (high performance computing) down the road, AstroBEAR uses a distributed tree algorithm in which each processor is only aware of the patches 'surrounding' its own patches. This includes its coarser parent patch and finer children patches, as well as neighbors of the same generation and overlapping patches from the preceding or succeeding generation.

Because of the nested nature of patches, neighbor and overlap relationships will always be inherited from parent relationships. Neighboring level l patches must either have neighboring level $l-1$ parents (or the same level $l-1$ parent). Likewise, the preceding overlaps of a patches first generation of children will be the last generation of children of the patches preceding overlaps and the succeeding overlaps of a patches last generation of children will be the first generation of children of the patches succeeding overlaps. For intermediate generations of a patches children, the preceding (succeeding) overlaps will be the previous (following) generation of children of neighboring patches (or of the patch itself.) Therefore when new patches are created, the parent patches determine which of its neighboring or overlapping patches need to know about each of its children so that the neighboring/overlapping patches can establish the necessary connections between children. The basic algorithm for updating local trees is summarized in table 3. *This should be table 1 but latex is not getting the reference right...

4. Threaded Multilevel Advance

Many if not all current AMR codes tend to perform the necessary computations using a single thread shown in the top panel of figure 1. If we follow the thread of computation we see that the level advances occur in the following order [0, 1, 2, 3, 3, 2, 3, 3, 1, 2, 3, 3, 2, 3, 3]. Good parallel performance then requires each level update to be independently balanced across all processors. Load balancing each level, however, requires each level to contain enough patches to be able to effectively distribute them among all of the processors. This requires each level to be fairly large to naturally have enough patches or for each level to artificially fragment patches into small pieces. The former leads to broad simulations (large base grid leaving resources for only a few levels of AMR), while the later leads to inefficient simulations due to the fair amount of overhead small grids require for ghost zone calculations.

Consider the somewhat idealized case of a 4x4 base level with 3 additional levels of refinement with each level having a centrally refined region as shown in figure 2. The left panel shows the resulting distribution of patches among processors when each level is required to be tbalanced. Each processor has a 2x2 patch on all four levels. The right panel shows the resulting distribution of patches when only global load balancing

<table>
<tr><td colspan="3">First Step</td></tr>
<tr><td>1</td><td>Receive new patches along with their parents, neighbors, and preceding overlaps from parent processors</td><td>Receive new succeeding overlaps from parent processors</td></tr>
<tr><td>2</td><td colspan="2">Create new children and determine on which child processors they will go</td></tr>
<tr><td>3</td><td>Determine which remote preceding patches might have children that would overlap with its own children and send the relevant children info</td><td>Determine which remote succeeding patches might have created children that would overlap with its own children and send the relevant children info</td></tr>
<tr><td>4</td><td colspan="2">Determine which remote neighboring patches might have children that would neighbor its own children and send the relevant children info</td></tr>
<tr><td>5</td><td colspan="2">Determine which local neighboring patches have neighboring children</td></tr>
<tr><td>6</td><td>Determine which local preceding patches have children that overlap with its own</td><td>Determine which local succeeding patches have children that overlap with its own</td></tr>
<tr><td>7</td><td colspan="2">Receive new children from remote neighboring patches and determine which of the neighbors' children neighbors its own children</td></tr>
<tr><td>8</td><td>Receive children from remote preceding patches and determine which of the patches children overlaps with its own</td><td>Receive children from remote succeeding patches and determine which of the patches children overlaps with its own.</td></tr>
<tr><td>9</td><td>For each remote child, send the child's info as well as information about its parents, neighbors, & preceding overlaps.</td><td>For each remote child, send the child's succeeding overlaps.</td></tr>
<tr><td colspan="3">Successive Steps</td></tr>
<tr><td>10</td><td colspan="2">Create new children and determine on which child processor they will go</td></tr>
<tr><td>11</td><td colspan="2">Determine which remote neighboring patches might have old/new children that would overlap/neighbor its own new children and send the relevant children info</td></tr>
<tr><td>12</td><td colspan="2">Determine which local neighboring patches might have old/new children that would overlap/neighbor its own new children</td></tr>
<tr><td>13</td><td colspan="2">Receive new children from remote neighboring patches and determine which of the neighbors' children neighbors/overlaps its new/old children</td></tr>
<tr><td>14</td><td colspan="2">For each new remote child, send the child's information, and the information about its parent, neighbors, & preceding overlaps.</td></tr>
<tr><td>15</td><td colspan="2">For each old remote child, send the child's succeeding overlaps.</td></tr>
</table>

Table 1: The split rows denote actions taken by the current generation of patches (left) and the previous generation of patches (right). Note that physically disjoint patches can overlap with each other's ghost zones - so the 1st generation of a patch's neighbor's children can overlap with the patch's 2nd generation of children in steps 11-13. See the section on distribution for calculation of parent and child processors.

is required. Now instead of having four 2x2 patches, each processor has one 4x4 patch. While each processor has technically the same number of cells to update, the reduced number of ghost zone calculations in the global load balancing scheme results in a 61% reduction of the computational cost. By removing the need to balance each level, one can more efficiently perform deep AMR simulations with large dynamic ranges.

The problem becomes worse when there are more ghost zones or when a patch has

Figure 1: Plot showing the basic AMR algorithm in both the Serial form and the Threaded. Odd number lines from the top show (O)verlaps, (A)dvances, and (S)ynchronizations from level 0 through 3 with even numbered lines showing (P)rolongations and (R)estrictions connecting the levels. Arrows show direction of execution.

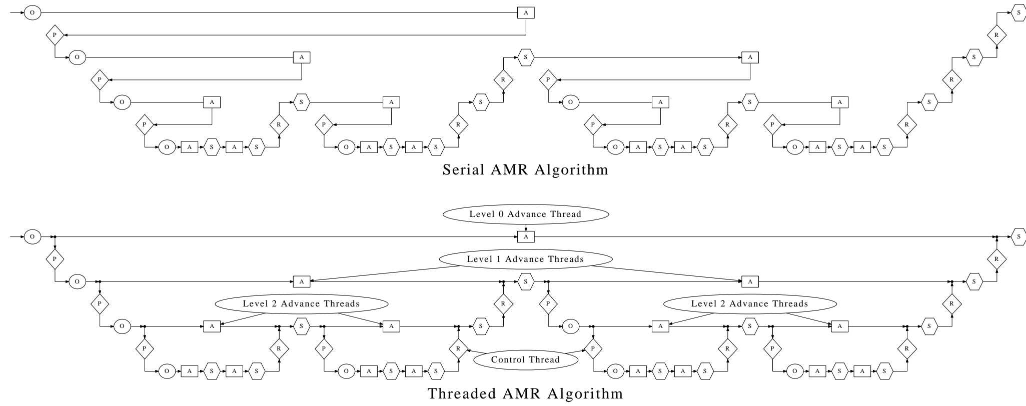
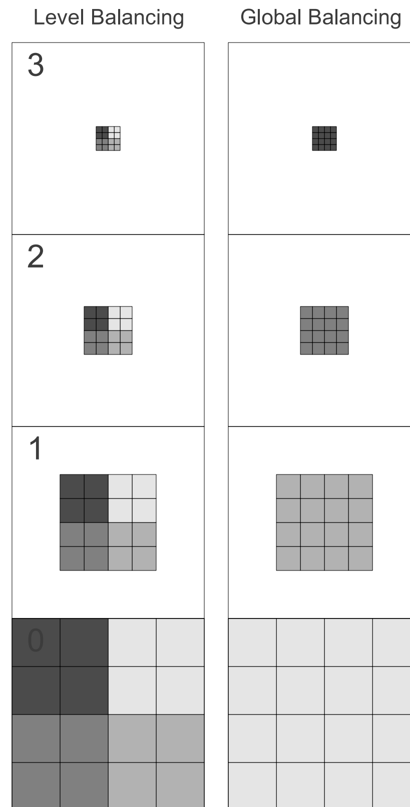


Figure 2: Sample Distributions for level by level load balancing vs global load balancing on 4 processors.



to take multiple steps. Consider an isolated level l patch of size $mx \times my$ inside of a parent patch on level $l-1$. Let's assume that the coarsening ratio $R = 2$ and that the patch must therefore take two steps each of which requires n ghost cells. On the first step it must update a region that is (mx + 2n × my + 2n) so that on the second step it can update its internal region that is (mx × my). For the example above this results in the global load balancing reducing the computational cost by 67%.

The lower panel of figure 1 demonstrates the use of threads in AstroBEAR. There is a control thread which handles all of the communications and computations required for (P)rolongating, (O)verlapping, (S)ynchronizing, and (R)estricting as well as the finest level (A)dvances. Each coarser level (A)dvance has its own thread and can be done indepedently with preference being given to the thread that must finish first (the finer level threads) . In practice we found that effective scheduling of the various advances gave good performance without requiring additional threading libraries. This pseudo-threading or scheduling algorithm allows each processor to determine how long its control thread will have to wait after completing its own required advances for the other processors to catch up. Then during this time, the processor can switch to the coarser grid advance threads giving priority to the finest coarser grids since these would hold up the other processor's control threads first.

5. Load Balancing

Load Balancing Typically when distributing grids, each processor is aware of the entire amr tree and as such can determine where each new patch fits within some ordering scheme. With a distributed tree this is no longer possible since processors are not aware of all of the patches. However, if a set of parent patches is in Hilbert order, and those patches children are locally sorted among themselves in Hilbert order, then it is expected that sorting the new child patches first by their parents order and then by their local order will give a fairly good approximation to a true Hilbert order for all of the child patches. Additionally if patches on a given level are distributed on different processors in Hilbert order, and each processor locally sorts its patches in Hilbert order, then sorting patches first by processor, then by local Hilbert order would give a fairly good approximation to a true Hilbert order. In this manner if a processor knows the total workload of all child patches on every lower processor and higher processor, it can determine where its new child patches fit within the Hilbert ordering scheme even if it is unaware of the actual extent or location of all of the other child patches.

Shown in figure 3 is a graphical representation of the distribution algorithm. The first two rows show the ordering of new children of each patch in relative Hilbert order, and the ordering of patches within each processor. Each processor p then sums the workload W^p_{l+1} of all of its new level $l+1$ children accounting for all R steps where R is the refinement ratio and p is the processor index (see the third row of figure 3). Next each processor determines its share of the workload. As was mentioned before, threading level advances removes the need for balancing each level and instead allows for global load balancing. It also allows for consideration of the progress of coarser grid advance threads when successively distributing the work load of finer grids in the following manner. Each processor calculates the remaining workload on each coarser level $W^p_{l'}$ where $0 <= l' <= l$ as well as the number of remaining level l advances $n^l_{l'}$ that can be completed before level

l' will need to be completed. Each processer then calculates the work load imbalance per level l step

$$\delta_l^p = \sum_{l'=0}^{l} \frac{W_{l'}^p}{n_{l'}^l} \tag{1}$$

These two quantities W_{l+1}^p and δ_l^p are then shared among all processors. Each processor then calculates its share of the new level $l+1$ work load

$$\epsilon^p = \frac{\sum_{p=1}^{NP} W_{l+1}^p + \delta_l^p}{NP} - \delta_l^p \text{ where } \sum_{p=1}^{NP} \epsilon^p = \sum_{p=1}^{NP} W_{l+1}^p \tag{2}$$

shown in the fourth column of figure 3.

Then the workloads per processor W_{l+1}^p are partitioned over ϵ^p to give the new workload allocations shown in the fifth row of figre 3 so that each processor can determine from which 'parent processors' it should expect to receive new child grids from as well as which 'child processors' it should give new grids to. In the example shown in figure 3, processor one would expect to potentially receive grids from processor two and processor two would likewise expect to receive grids from processor three. The processor allocations are then mapped onto the new child patches (in Hilbert order).

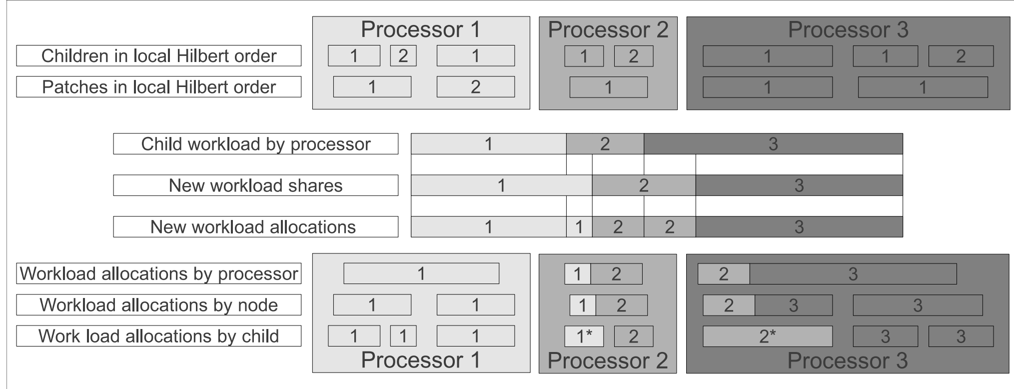
Occasionally a single child will extend over multiple processors. When this happens on all but the finest level, the child is always assigned to the lowest ranking processor and the lowest ranking processor ends up being overallocated while the higher ranking processor(s) are left under allocated. Often the higher ranking processor(s) are under allocated somewhere else to balance out on the average, but small imbalances in allocation can usually be corrected in the next distribution of finer patches as long as the filling fractions are reasonable. Sometimes, however, a large base patch can lead to a small filling fraction in which case most of the workload is on the base level. To prevent this large patch from being assigned to a single processor, splitting is initially used on the base patch. Splitting is also used on the highest level of refinement in order to balance the allocations.

Splitting is achieved by recursively bisecting the child grid into proportionate pieces choosing directions that maintain the largest Hilbert distance between the two pieces. In this manner a large fixed grid allocated among 64 processors will be split into 64 pieces that will trace out a third order Hilbert curve as seen in the level 0 distribution in figure 4. Processors also account for the workload of the split pieces when choosing the direction so that an 8x4 patch with two equal allocations, will be split into two 4x4 patches instead of two 8x2 patches.

6. Hyperbolic engine

Currently AstroBEAR's hyperbolic solver uses a Gudonov type unsplit integrator that utilizes the CTU+CT integration scheme (Stone & Gardiner 08). Unsplit integrators however, often require many intermediate variables be stored globally during a grid update which can require 10-50x the space required for storing the array of conservative variables. For GPU type systems, this would considerably restrict the size of a grid

Figure 3: Plot showing sample distribution of a 64 processor run with 3 levels of refinement. The left column shows the processor ID with arrows pointing to the grids assigned to processor 25. The right column shows the extent of the AMR tree stored as neighbors to grids assigned to processor 25



that could be updated. In AstroBEAR we implement a sweep method that essentially pipelines any sequence of calculations into a one dimensional pass across the grid where variables are only stored as long as they are needed. This is ideally suited for GPU calculations in which a CPU could constantly be sending in new field values and retrieving updated field values while the GPU uses a minimum of memory.

The pipelining can be used for any computational stencil provided that the dependencies between stencil pieces is explicitly stated. For example consider a simple 2D 1st order Gudonov method in which the initial state is stored in q, and the updated fields are stored in Q. The x and y fluxes are stored in fx,fy, and the left and right interface states are stored in qLx,qLy and qRx,qRy respectively. We also adopt the convention that stencil pieces stored on cell edges (ie qLx, qRx, fx) at position i-1/2 are stored in their respective arrays with the index i. The stencil dependencies can then be expressed as:

Variables		Range	Calculation
qLx	q	[-1,-1,0,0]	qLx(i,j)=q(i-1,j)
qRx	q	[0,0,0,0]	qRx(i,j)=q(i,j)
qLy	q	[0,0,-1,-1]	qLy(i,j)=q(i,j-1)
qRy	q	[0,0,0,0]	qRy(i,j)=q(i,j)
fx	qLx	[0,0,0,0]	fx(i,j)=RM(qLx(i,j),qRx(i,j))
fx	qRx	[0,0,0,0]	
fx	qLy	[0,0,0,0]	fy(i,j)=RM(qLy(i,j),qRy(i,j))
fx	qRy	[0,0,0,0]	
Q	fx	[0,1,0,0]	Q(i,j)=q(i,j)
Q	fy	[0,0,0,1]	+ fx(i,j) - fx(i+1,j)
Q	q	[0,0,0,0]	+ fy(i,j) - fy(i,j+1)

Now given the size of Q we want updated we can work backwards to determine over what range of indices we need to calculate fx, fy, qRx, qLx, qRy, qLy & q. To pipeline the calculation we then construct a sliding window for each stencil piece that passes across the grid from left to right. The dependencies determine the window's lead (ie how far in

Figure 4: Plot showing sample distribution of a 64 processor run with 3 levels of refinement. The left column shows the processor ID with arrows pointing to the grids assigned to processor 25. The right column shows the extent of the AMR tree stored as neighbors to grids assigned to processor 25

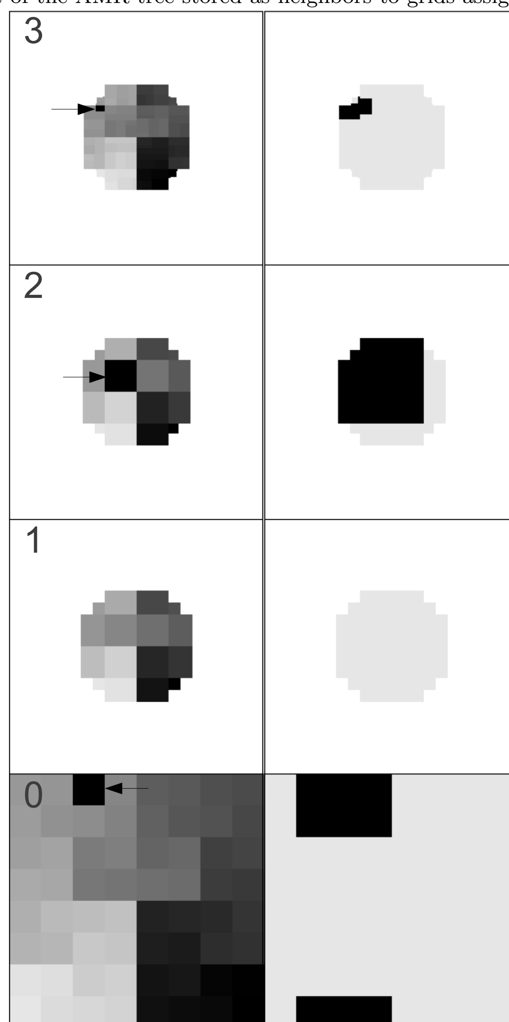


Figure 5: Sweep Window

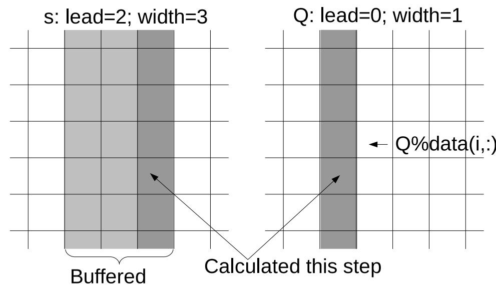
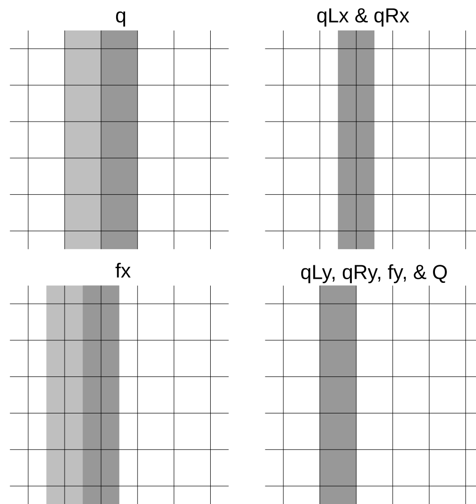


Figure 6: Sweep Window



front of updating Q must we calculate the stencil's value) as well as the window's width (ie how long we need to hold on to values for other stencil pieces to use. For example in the following figure we have a stencil piece s that leads the update of Q by two columns and whose window is 3 cells wide.

For the 2D example above this would give the following windows:

7. The Super-Gridding experiment

One of the goals in designing AstroBEAR was to keep the algorithms simple and to have all data manipulation routines operate on individual grids (or pairs of grids). However once the stencil dependencies are explicitly stated, it becomes possible to modify the hyperbolic advances to be much more sensitive to the available data. This can allow each processor to perform computations required by neighboring grids only once - instead of twice and allows for processors to skip computations needed to update coarse cells only to replace those values with data from fine cells. For small adjacent grids or completely

refined grids this can reduce the amount of computation by 50-100%. To this end we implemented a super-gridding scheme which was basically as follows:

1. Collect physically adjacent grids on each processor into super-grids.
2. For each supergrid, flag the cells that needed to be updated.
3. Using the stencil dependencies work backwards to flag the locations where each stencil piece needs to be calculated
4. Then sweep across the supergrid performing only the necessary computations.

Of course storing global mask arrays over the entire supergrid for each stencil piece is memory intensive - so instead we implemented a sparse mask storage that was essentially a collection of non-intersecting boxes marking the regions to calculate.

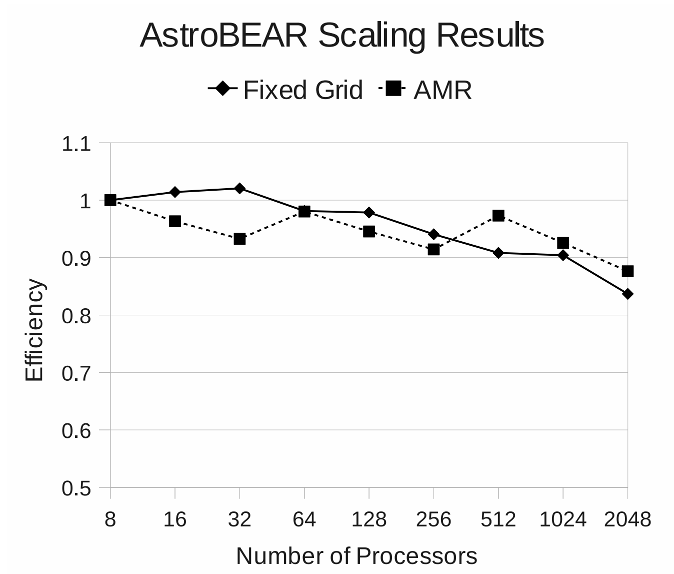
We then modified the above algorithm to allow for processors to prioritize computations to better overlap communication with computation as follows:

1. Collect physically adjacent grids on each processor into super-grids.
2. For each supergrid, flag the cells that needed to be updated.
3. Using the stencil dependencies work backwards to flag the locations where each stencil piece needs to be calculated
4. Then using the old grids with the previous level's data determine which of those calculations can be performed prior to receiving data from overlaps and begin performing those while waiting for overlap data.
5. Determine which fluxes and emf's will need to be synchronized with neighboring processors.
6. Work backwards to determine which remaining calculations need to be performed prior to sending neighbors data and perform those calculations.
7. Send fluxes and then perform remaining calculations that can be done before receiving data from children while waiting for child data.
8. Using data from children continue performing calculations that can be done before receiving data from neighbors while waiting for neighbor data.
9. Using neighbor data complete all required calculations to advance grids.

Unfortunately the computational cost associated with keeping track of all of these logical arrays as well as the additional shuffling of data back and forth became comparable to the savings in the number of reduced stencil computations. It may be possible, however, to improve the algorithms for managing the sparse logical arrays and to design an efficient algorithm that avoids redundant computations on the same processor for unsplit integration schemes.

8. Performance Results

For our weak scaling tests we advected a magnetized cylinder across the domain until it was displaced by 1 cylinder radius. The size of the cylinder was chosen to give a filling fraction of approximately 12.5% so that in the AMR run, the work load for the first refined level was comparable to the base level. The resolution of the base grid was adjusted to maintain 64^3 cells per processor and we found that our weak scaling for both fixed grid and for AMR is reasonable out to 2048 processors.



9. Conclusion