Efficient Parallelization for AMR MHD Multiphysics Calculations; Implementation in AstroBEAR

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

[a]*Department of Physics and Astronomy, 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 or grid based AMR often employs ghost cells to decouple the hyperbolic advances of each grid on a given refinement level. This decoupling allows each grid to be advanced independently. In AstroBEAR we utilize this independence by threading the grid advances on each level 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. Threading of level advances can also improve performance by interleaving communication with computation, especially in deep simulations with many levels of refinement. To improve memory management we have employed a distributed tree algorithm that requires processors to only store and communicate local sections of the AMR tree structure with neighboring processors. We have also implemented a sweep method that pipe-lines 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, 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

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

to a CPU, future clusters will likely have much less memory per processor similar to what is already seen in GPU's. Additionally each processor only sends and receives the portions of the tree necessary to carry out its communication.

AstroBEAR also utilizes inter-level 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 inter-level threading this restriction is lifted. In section 3 we will discuss the distributed tree algorithm, in section 4 we will discuss the inter-level threading of the advance, in section 5 we will discuss the load balancing algorithm and in section 6 we will discuss the pipe-lining 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 grid contains a regular array of cells in which the fluid variables are stored. Grids 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" grid of level $l-1$ and resolution $\Delta x_{l-1} = R \times \Delta x_l$ where R is the refinement ratio. The collection of grids comprises the AMR mesh. In addition to the computations required to advance the fluid variables, each grid needs to exchange data with its parent grid (on level $l-1$) as well as any child grids (on level $l+1$). Grids also need to exchange data with physically adjacent neighboring grids (on level l). In order to exchange data, the physical connections between parents, children, and neighboring grids are stored in the AMR tree as a parent, child, or neighbor connection between corresponding nodes where each node represents a grid. Finally since the mesh is adaptive, there will be successive iterations of grids on each level. Since the fluid variables need to be transferred from previous iterations of grids to current iterations, there needs to be overlap connections between nodes of successive iterations corresponding to physically overlapping grids of successive iterations. Additionally, while during a simulation there will be many iterations of grids on each level, only the two most current iterations are stored in the tree. After new grids finish their advances they become old grids at which point the previous old grids are discarded. In summary, nodes on level l have a parent connection to a node on level $l-1$, child connections to nodes on level $l+1$, neighbor connections to nodes on level l of the same iteration, and overlap connections to nodes on level l of the previous or successive iteration.

After each iteration of level l grids are created they initialize their cells(including ghost regions) with a combination of prolongated data from their parent grid as well as data from the preceding set of level l grids that physically overlap. These ghost regions

are needed to update the fluid variables within the grid. They then determine which cells to refine and then lay down their first set of child grids to cover those cells. They then take one step of Δt_l while their child grids advance R steps of $\Delta t_{l+1} = \Delta t_l / R$. They then merge the restricted data from their child grids with their own updated data before synchronizing fluxes and emfs with any adjacent neighboring level l grids. They then successively apply "overlaps" from their neighbors to update their ghost zones, lay down a successive iteration of children, advance another time step, apply restricted data from children, and synchronize fluxes with their neighbors until they have completed R steps at which point they restrict their data to be applied to their parent grid. Notice that neighbors function as both overlaps and neighbors for all steps following the first and that preceding overlaps can be discarded following the first step. They then wait for the next iteration of level l grids to be created at which point there data is copied onto their succeeding overlaps after which they are destroyed. Throughout a grids lifetime it must therefore share data with its parent grid, preceding overlaps, multiple iterations of children grids, neighbor grids, and succeeding overlaps. These connections between grids form the AMR tree. While a node may have many successive iterations of children, only connections to children that belong to the two most current iterations of the child level are kept.

It should also be mentioned that if a grid is isolated (ie. has no neighbors) or even if it is partially isolated, it cannot update all of its ghost zones after each step. There are two solutions to this problem. One is to use the time derivative for the fluid values calculated on the coarse grid to update the ghost zones where needed, although this requires grids to advance before their children. This method has difficulty if shocks are not resolved since spatial discontinuities will lead to delta functions for time derivatives which produce large errors when discretized. The alternative method is to use extended ghost zones that allow grids to successively update smaller and smaller regions so that there is always enough ghost zones to complete the final step. In general if n_{ghost} is the number of ghost zones needed and R is the refinement ratio, then the extended ghost region needs to initially be $R \times n_{ghost}$ cells wide. While only isolated grids need to update extended ghost zones, it can be difficult to implement efficient advance schemes for partially isolated grids, and each grid is often assumed to be isolated. This can, however, result in a large overhead for small grids, especially when the refinement ratio R, or n_{ghost} is large. This does, however, allow coarse grids to be advanced independently of fine grids and can be exploited to allow for more efficient distribution as well as the creation of multiple advance threads described in section 4.

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 grid requires m bytes per node to store its meta data (ie physical bounds and the processor containing the grid), and that each grid requires on average d bytes for the actual data - and that their are on average n grids on each of p processors, then the memory per processor would be $nd + nmp$. The memory requirement for the AMR hierarchy meta data becomes comparable to the local actual data when $p = d/m$. If we assume a 3D isothermal hydro run with an average grid 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 section of the tree containing nodes that connect to its own grids' nodes. Since maintaining these local trees as the mesh adapts is not entirely trivial, we describe the process below.

Because of the nested nature of grids, neighbor and overlap relationships between nodes can always be inherited from parent relationships. First consider the neighbors of the n^{th} iteration of a node's children. The nested nature of the grids restricts each of their neighbors to either be a sibling (having the same parent node and be of the same iteration), or to be a member of a neighbor's n^{th} iteration of children. If we consider the node as being a neighbor to itself, then this can be summarized by saying that the neighbors of a node's n^{th} iteration of children will be a subset of the n^{th} iteration of children of the node's neighbors. Also after a node completes its advances and becomes an old node, its neighbors also become old nodes, however the connections do not change.

For overlaps it is a bit more complicated. As was mentioned before, at any given moment only the last two iterations of grids (and corresponding nodes) on a given level are stored. The old nodes need to have succeeding overlap connections to current nodes that physically overlap. Likewise the current nodes need to have preceding overlap connections to old nodes that physically overlap. Because nodes will create multiple iterations of children, this leads to two different situations. First when the current nodes on level l are the 1st iteration of children of the current nodes on level $l-1$, the old nodes on level l are the last iteration of children of the old nodes on level $l-1$. Second, when the current nodes on level l are the 2nd or greater iteration of current nodes on level $l-1$, the old nodes on level l will be the previous children of the current nodes on level $l-1$.

In the former case, the nested nature of the grids restricts the preceding overlaps of a current level l node to be among the last iteration of children of the node's parent's preceding overlaps. And the succeeding overlaps of an old level l node must be among the first iteration of children of the node's parent's succeeding overlaps. Here the overlap relationships between current and old level l nodes are directly inherited from overlap relationships between their parent nodes on level $l-1$. In the latter case, the current node's preceding overlaps must be among the node's parent's previous iteration of children. Additionally, while children of neighboring grids can not physically overlap, their ghost regions can, and as a such may require copying of overlapping data and so in the latter case we also have that the current node's preceding overlaps may also contain the current node's neighbors' previous iteration of children. Or if we again consider a grid as being its own neighbor, can be summarized by saying that the current node's preceding overlaps must be among the node's parent's neighbors' previous iteration of children and that the old node's succeeding overlaps must be among the node's parent's neighbors' successive iteration of children. These relationships are summarized in table 3

Now the AMR tree needs to be updated after each new iteration of child nodes are created. Since these relationships between children can be inherited from corresponding relationships between parents, parent nodes first share new child information with their neighbors as well as with their preceding overlaps if it is the first iteration of children. Additionally if the current nodes are the first iteration of children, the old nodes would share their last iteration of children with their succeeding overlaps. If the tree is distributed, however, these connections can only be formed if remote neighbors and overlaps

Child iteration	Neighbors are a subset of ...	Preceding overlaps are a subset of ...	Succeeding overlaps are a subset of ...
1	Parent's neighbors' 1^{st} children	Parent's preceding overlaps' R^{th} children	Parent's neighbor's 2^{nd} children
2	Parent's neighbors' 2^{nd} children	Parent's neighbors' 1^{st} children	Parent's neighbors' 3^{rd} children
3	Parent's neighbors' 3^{rd} children	Parent's neighbors' 2^{nd} children	Parent's neighbors' 4^{th} children
...	...	...	....
R-1	Parent's neighbors' R^{th}-1 children	Parent's neighbors' R^{th}-2 children	Parent's neighbors' R^{th} children
R	Parent's neighbors' R^{th} children	Parent's neighbors' R^{th}-1 children	Parent's succeeding overlaps' 1^{st} children

are aware of these new (and old) children. Each processor uses information about its local nodes' neighbors and overlaps to determine what remote nodes would need to know about new (and old) children. These children are then communicated between remote nodes and the connections are made within the sub-trees on each processor. Then when children are distributed, each new node also receives information about their preceding overlaps and neighbors from their parent nodes so that the assigned processor will have the necessary tree information. One subtlety is that when the current iteration of level l nodes are the 1st iteration of children of level $l-1$ nodes, old level l nodes must also receive succeeding overlaps from their parent nodes. 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 grid updates across all levels in a prescribed order (0, 1, 2, 2, 1, 2, 2, 0...) 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 grids to be able to effectively distribute them among all of the processors. This requires each level to be fairly large to naturally have enough grids or for each level to artificially fragment grids 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 required 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 1. The left panel shows the resulting distribution of grids among processors when each level is required to be balanced. Each processor has a 2x2 grid on all four levels. The right panel shows the resulting distribution of grids when only global load balancing is required. Now instead of having four 2x2 grids, each processor has one 4x4 grid. 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

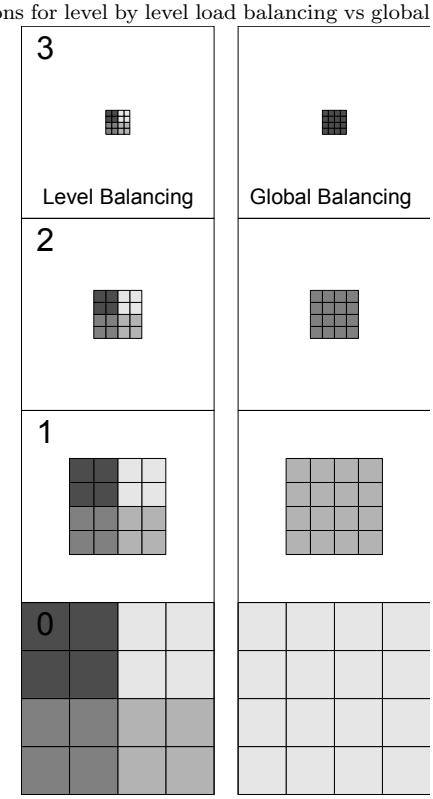
First Step		
1	*Receive* new grids & nodes along with their parents, neighbors, and **preceding** overlaps from parent processors	*Receive* **succeeding** overlaps from parent processors
2	Create new children and determine on which child processors they will go	
3	Determine which remote **preceding**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

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

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 grid has to take multiple steps. Consider an isolated level l grid of size $mx \times my$ inside of a parent grid on level $l-1$. Let's assume that the coarsening ratio $R = 2$ and that the grid must therefore take two steps each of which requires n ghost cells. On the first step it must

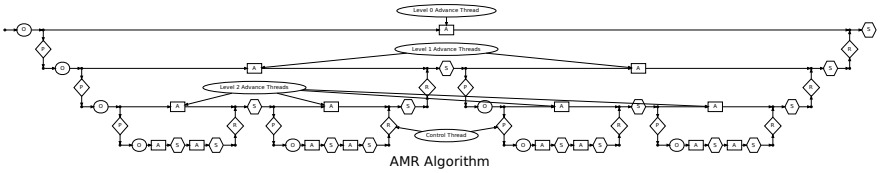
Figure 1: Sample Distributions for level by level load balancing vs global load balancing on 4 processors.



update a region that is $(\mathrm{mx} + 2\mathrm{n} \times \mathrm{my} + 2\mathrm{n})$ so that on the second step it can update its internal region that is $(\mathrm{mx} \times \mathrm{my})$. For the example above this results in the global load balancing reducing the computational cost by 67%.

In figure 2 we show a schematic of the AMR algorithm and the various threads of computation. 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 independently 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 allowed each processor to determine how long its control thread would have to wait after completing its own required advances for the other processors to complete their required advances. Then while waiting, each processor operates on their coarser grid advance threads giving priority to the finest coarser grids since these would hold up the other processor's control threads first.

Figure 2: Plot showing threads of AMR algorithm



5. Load Balancing

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 advancing coarser grids when successively distributing the work load of finer grids in the following manner. When distributing children of level l grids each processor first calculates the remaining workload on each coarser level $W_{l'}^p$ 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 processor then calculates the work load imbalance per level l step

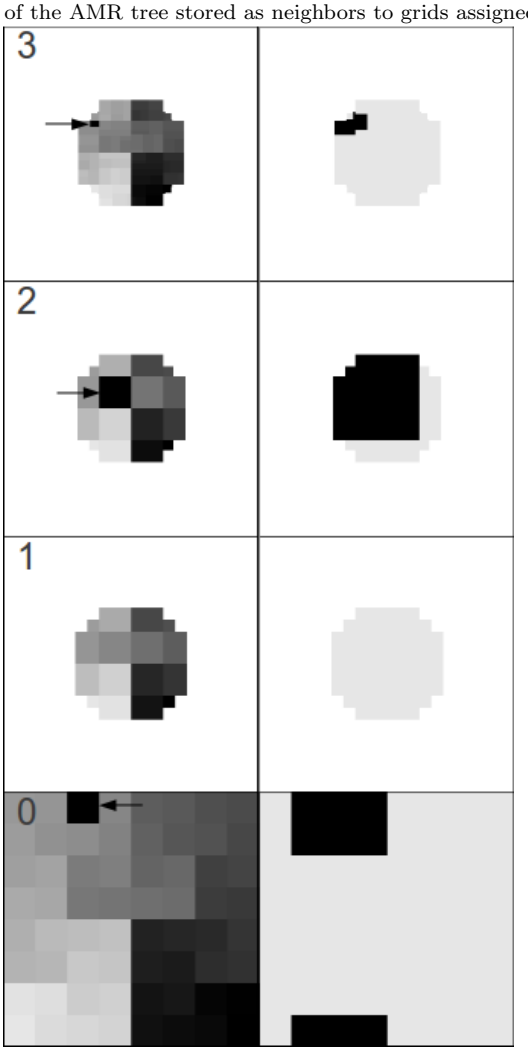
$$\delta_l^p = \sum_{l'=0}^{l} \frac{W_{l'}^p}{n_{l'}^l} \tag{1}$$

as well as the new child load per level l step W_{l+1}^p. Then these two quantities δ_l^p and W_{l+1}^p are distributed over all processors. Then each processor calculates its share of the new level $l+1$ work load

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

Then the workloads per processor W_{l+1}^p are partitioned over ϵ^p so that each processor can determine from which processors it should expect to receive new child grids from as well as which processor it should give new grids to. It then sorts its children in a Hilbert order before distributing them in order among its child processors. If the load of the next child grid is larger than the load assigned to the next child processor then the algorithm can optionally attempt to split (artificially fragment) the child grid into proportionate pieces before assigning them to the child processors. Usually this is avoided and the first child processor is given the entire child grid since future distributions of finer grids can compensate for this inequality. If we however are distributed finest level grids, then this splitting is used. The splitting is also used on the coarsest root grid since its size can in general be very large.

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



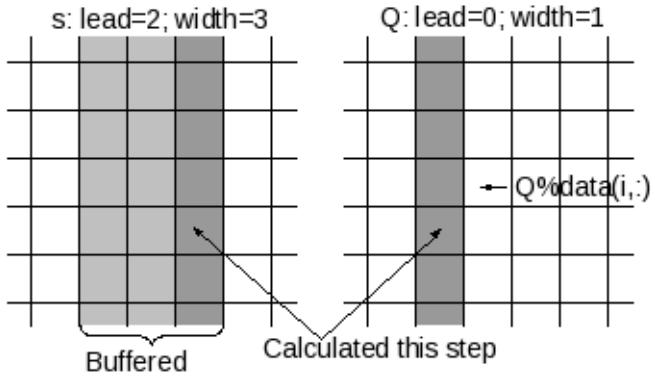
6. Hyperbolic engine

Currently AstroBEAR's hyperbolic solver uses a Godunov 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 that could be updated. In AstroBEAR we implement a sweep method that essentially pipe-lines 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.

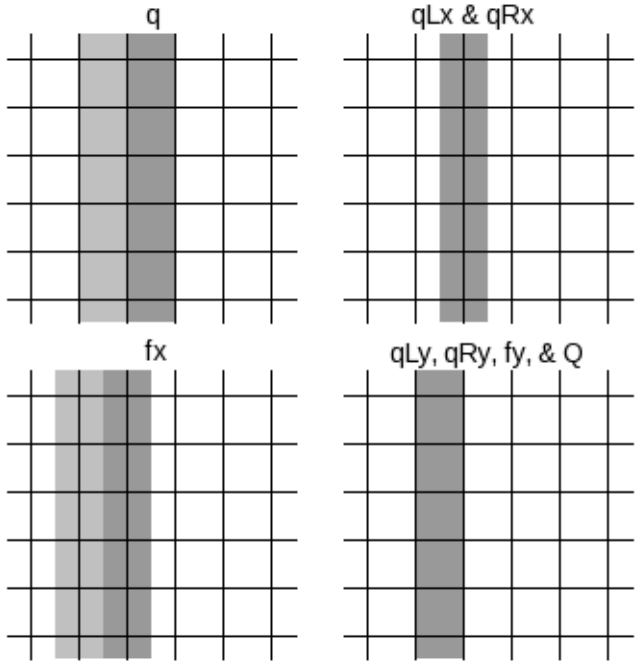
The pipe-lining is done automatically provided that the dependencies between stencil pieces is explicitly stated. For example consider a simple 2D 1st order Godunov method in which the initial state is stored in q, and the updated fields are stored in Q. The x and y fluxes are stored in fx,fy, and the 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 pipe-line 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 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.

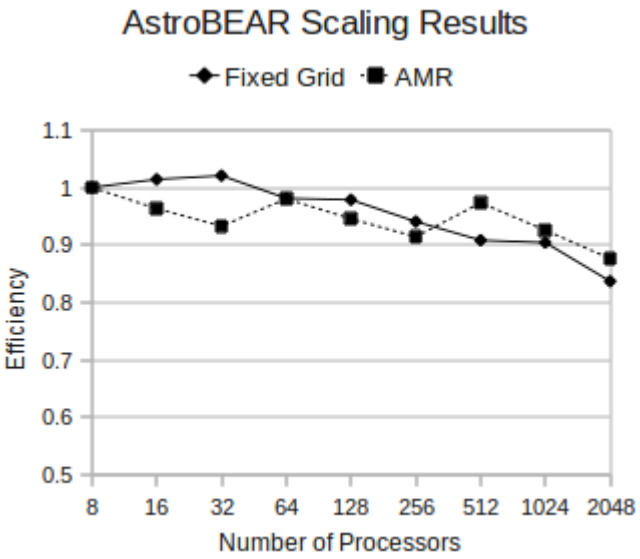
We then modified the above algorithm to allow for processors to prioritize computations to better overlap communication with computation as follows:

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

References