Master Project Report
Performance Optimization for Regridder Component of Uintah based on Dynamic Analysis

Nan Xiao

April 28, 2015

1 Abstract

The Uintah software is a set of libraries and applications for simulating and analyzing complex chemical and physical reactions. These reactions are modeled by solving partial differential equations on structured adaptive grids using hundreds to thousands of processors. But when encountering full-physics problems on large domain using tens of thousands of cores, poor scalability of some existing data structures and algorithms becomes a bottleneck for the performance. Adaptive mesh refinement is one of the challenges which requires frequent changing of grid. This report will explain in details the analysis and optimizations for regridder component which reduce the overall timecost of regridding process by 45%.

2 Introduction

Partial Differential Equations (PDEs) are mostly modeled and solved by discretizing the problem domain into set of cells which is also referred to as a mesh. The size of cells determines the final error, where the smaller the size is, the better result we will get. But decreasing size of cells also increases time cost and memory cost for computations. Adaptive mesh refinement (AMR) is a method that attempts to reduce the error while limits the overall workload for computations which refines cells with larger error to a higher resolution and coarsens smaller error cells to a lower resolution. However, due to frequent changes of grid, AMR operations often have a poor scalability.

Regridding is required once current timestep exceeds a preset number or a refinement flag is found inside any patch. Several processes need to be done to change the grid which generally includes regridding, load balancing and scheduling, since grid and task graphs need to be recreated every time regrdding occurs. Based on previous experiments, scalability of regridder component is identified as the bottleneck for the overall AMR performance. In order to analyse which parts inside the regridder component lead to the poor scalability, a dynamic profiler was implemented for monitoring the memory and time cost for each function. The implementation details about the profiler will be explained in the next section.

Based on the analysis of results returned from the profiler, several data structures are identified to be dominated in the overall timecost, such as BVH tree structure inside selectPatches function which is used frequently for querying patches within a given index range, red-black tree of STL lib which is widely used for eliminating duplicates and return elements in order. We also found some redundant sorting processeses. After replacements and optimizations to these data structures, the overall timecost for regridding is reduced by 45% and the final performance got improved by approximately 82%.

3 Uintah Dynamic Profiler

3.1 Main Idea

The main idea for this profiler is to discrete the whole function process by sampling current stack state. Since Uintah has its own parallel memory allocation component, dead lock may occur when multiple threads request current stack state check. So apart from common Uintah memory allocator, each thread is designed to own a 1 MB buffer for personal use to check stack state.

For sampling procedure, we can set the sampling rate by specify a constant time interval. Once the program runs into the manually inserted timer block, sampling process is triggered. The sampling component will then gradually establish a dependence graph structure of all components inside this timer block by backtracing current program stack, and also trace memory usage by storing program break locations (which is the first location after the end of the uninitialized data segment) at each sampling moment. When the program runs out of the inserted timer block, the sampling process is set to be disabled and output is printed.

3.2 Sampling Data

Each sampling data we get is a list of function names which looks like this:

```
===
dbg/lib/libCCA_Components_LoadBalancers.so
(Uintah::LoadBalancerCommon::createNeighborhood(HandleINS_4GridEEES5_+0x454)
dbg/lib/libCCA_Components_Schedulers.so
(Uintah::TaskGraph::createDetailedTasks::DetailedTasks(HandleINS_4GridEEES7_+0x9e)
dbg/lib/libCCA_Components_Schedulers.so
```

```
(Uintah::SchedulerCommon::compile+0x1da)
dbg/lib/libCCA_Components_SimulationController.so
(Uintah::AMRSimulationController::recompile(HandleINS_4GridEEEi+0x7e2)
dbg/lib/libCCA_Components_SimulationController.so
(Uintah::AMRSimulationController::run+0x665)
./sus(main+0x1160)
===
```

As we can see the calling dependency is from the top to the bottom, where main function called AMRSimulationController::run function, and run function next called AMRSimulationController::recompile function and so on. Each "===" indicates boundary between samplings, and dbg/lib/*_*.so indicates the the path to a compiled library file, the line below the path indicates the detail of the calling function and its parameters inside that file.

3.3 Analysis Procedure

We can generate a directed graph based on the collection of sampling data lists above.

```
Data: Collection of Sampling Data
Result: Dependency Graph
stack = [];
for line in fin do
    if line ≠ "===" then
        stack.append(line)
    else
        for depth = 1 to len(stack) + 1 do
            cur_stack = stack[-depth:];
            cur_key = stack_to_id(cur_stack);
            prv_key = stack_to_id(cur_stack[1:]);
            count[cur_key] = count.get(cur_key) + 1;
            if prv_key not in edge: edge[prv_key] = [];
            edge[prv_key].append(cur_key);
        end
    end
end
```

Algorithm 1: Generate Dependency Graph

Where *stack* stores a sampling data list, and *count* is a map which stores how many times each function is called, and *edge* here stores all the calling

dependencies. Since the Naming of each function could be really long and it would be really low efficient to use these long string as key index, a mapping from string to an integer is made to assign each function a unique ID.
In order to know the time cost for all components inside the timer block, the following formula is used:

$$F_{tp}(cur\_fun, par\_fun) = \frac{cnt_{cur\_fun}}{cnt_{par\_fun}} \tag{1}$$

$$cnt_{fun} \approx \sum_{i=1}^{n} cnt_{calling\_fun_i} \tag{2}$$

Where $F_{tp}(cur\_fun)$ represents percentage of time cost of current function cur_fun inside the function cur_par which calls it. And cnt_{cur_fun} represents number of samplings we got which contain name of cur_fun. In order to get a clearer view of all dominant components, any components with $F_{tp}(cur\_fun)$ smaller than 1% are filtered out.
Based on the number of samplings of each function, we can calculate the approximate time cost with the following:

$$t_{fun} = cnt_{fun} \times t_{interv} \tag{3}$$

Where t_{interv} is a preset value defining time interval between samplings.
We can also get the time cost for each component overall based on formula below:

$$F_t(cur\_fun) = \sum_{i=1}^{n} F_t(par\_fun_i) \times F_{tp}(cur\_fun, par\_fun_i) \tag{4}$$

Where par_fun_i represents ith function calling current function (ith parent node of current node in the graph).

4 Profiling Results Analysis

Figure 1 represents the profiling results for regridder before optimization. As we can see from the figure that there are mainly two dominant components inside regridder take up 99.33% time cost: createNeighborhood (node 35) for 47.09% and createDetailedDependencies (node 402) for 52.24%.

4.1 createNeighborhood

For function createNeighborhood (node 35), the inputs parameters are old and new grids, the outputs are a STL set (Red-black tree), d_neighborProcessors, containing all the neighboring processors' ID and another STL set (Red-black tree), d_neighbors, containing all the neighboring patches' ID.
This function iterates through all patches of current grid and finds all patches that are assigned to itself (current processor). For each self-owned patch, it calls a selectPatches function (node 44) to find all of its neighboring patches and push these patch IDs into set d_neighbors (node 53) and their corresponding processor IDs into set d_neighborProcessors (node 63).
As we can see from the profiling graph, insertion of d_neighborProcessors takes up 21.43% time cost of createNeighborhood, and 29.91% for d_neighbors, 33.98% for selectPatches. Total time costs spent in these three data structures sum up to 85.32%, giving us rather large space to optimize its performance.

4.2 createDetailedDependencies

Function createDetailedDependencies (node 402) loop through all the tasks assigned to the current processor and create all the dependencies for each task . During this process, function getPatchesUnderDomain is called to get all patches that have dataflow with current task patches, where patches may exist in either current level or other levels. Function selectPatches is called again for querying all neighboring patches.
As we can see from the profiling results, the time costs for all components of createDetailedDependencies (node 436) are rather uniformly distributed, the only components that utilize time cost more than 10% are: LoadBalancer::inNeighborhood (node 444) for 13.68%, Task::getPatchesUnderDomain (node 409) for 15.25% and DetailedTasks::possiblyCreateDependency (node 459) for 17.67%, giving us less space to do optimization. But we can also notice that function inNeighborhood and getPatchesUnderDomain both frequently use sorting inside STL::set, it is still possible to improve the performance by implementing constant querying data structures or eliminating redundant sorting.

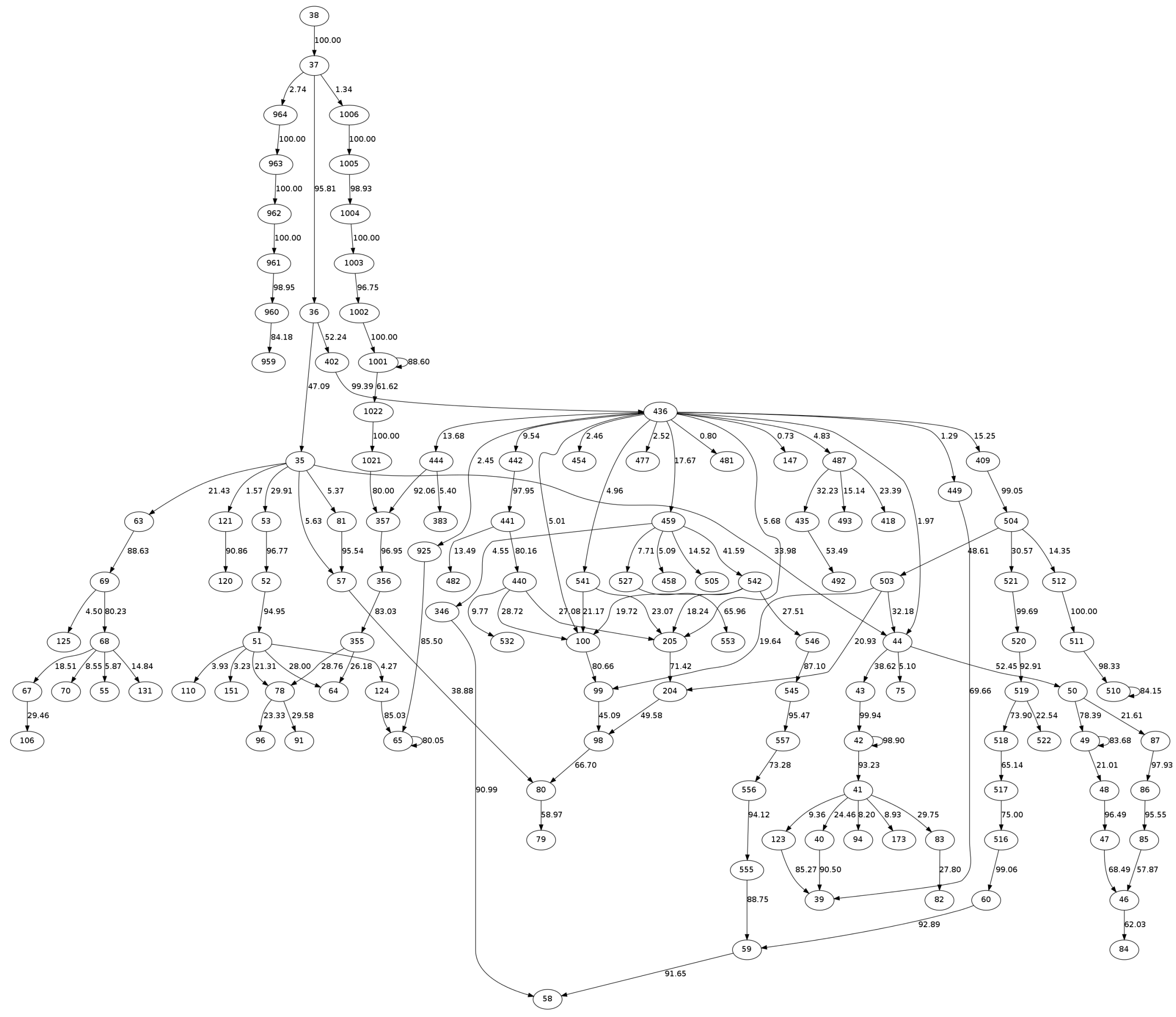

Figure 1: Component Dependencies before Optimization

5 Optimization

5.1 selectPatches

As we can see from analysis above, function selectPatches (node 44) is widely used in both createNeighborhood function and createDetailedDependencies function which utilizes 18.3% of total time cost.

5.1.1 BVH Tree

The previous data structure implemented in selectPatches is Boundding Volume Hierarchy (BVH) tree structure.
Main idea of BVH tree is rather straightforward like KD tree. In order to establish a data structure for a quick query for patches within a given index range, patches are stored in a tree structure where nodes in each level are divided in the maximum range dimemsion. Based on this idea, it has to do a patches sorting in each level which costs $O(n \log n)$ time and equally divides these patches into two sets. There are $\log n$ levels in total, so the constructor time complexity for this structure will be $O(n(\log n)^2)$. And each patches query has time complexity of $O(k \log n)$, where k represents number of patches returned. In some components of Taskgraph like function createDetailedDependencies, it requires all patches returned in order, from small ID to large one. This requires another sorting process for BVH tree since all patches store inside are disordered.

5.1.2 Hash Tree Structure

In order to decrease time cost for patches query to constant time, a Hash tree structure is implemented. Since in most cases, input patches are uniformly distributed, we can simply discretize and project patches into continuous space according to their indexes. Moreover, IDs of patches are generally assigned based on patches' indexes, from small to large coordinate. So in most cases, we do not need to do another sort for returned patches. However, there might be some rare cases where finer levels may not be continuous across the domain, a tree structured hash table was designed to solve this issue.
The constructor for this data structure mainly includes three parts: division of patches, hash function, and indexing. Division algorithm is used to cluster patches into several sets based on whether they are adjacent to each other

to eliminate empty space between clusters and guarantee memory utilization rate. A hash table is then implemented for each patches set. Due to the fact that there may still exist some empty buckets inside hash table, an indexing method is used for improving query performance.

1. Clustering patches

Algorithm for clustering patches is as follow:

```
Data: one vector of patches
Result: vectors of continuous patches
initialization;
create a box to enclose all patches;
generate histograms of all patches along x,y,z axis;
calculate empty ratio of current domain based on histograms;
if empty ratio less than 50% then
    split current domain;
    fit new boxes to each subdomain;
end
```

Algorithm 2: Clustering continuous patches

.

2. Hash function

Hash function is simple to design and calculate due to the cube structured domain of each level. We only need to do a projection for each patch's 3D index to 1D index in our hash table.

$$hash\_value_{x,y,z} = x \times area_{yz} + y \times length_z + z \tag{5}$$

For each query with a given index range, we just need to locate cell with the lowest index and cell with the highest index, and use a loop to output all patches between these two cells. However, there exists cases of varied patch sizes when size of whole domain is not divisible by number of patches in each dimension. For 1D case example, with domain length of 7 and number of patches of 7, we have to divide the domain into three parts: 2, 2, 3. In order to do the projection from 3D to 1D, we first need to project 3D coordinates of patches into uniform 3D Grids.

If it's not divisible, in case of x-axis:

$$uniform\_index_x(idx_x) = \lfloor number\_patches_x \times (idx_x - low\_bound_x + 1) \rfloor \tag{6}$$

When it's divisible, also in case of x-axis:

$$uniform\_index_x(idx_x) = \lfloor number\_patches_x \times (idx_x - low\_bound_x) \rfloor \tag{7}$$

3. Quick indexing

Due to the patches division method, we can guarantee that the maximum empty ratio of the whole domain is less than 50%, but there may still exists space in our structured hash table. In order to reduce the redundant time cost for scanning through empty buckets, a indexing method was used. A pointer is contained in each bucket (grid cell) of the hash table, it points to a valid patch when there exists a valid projection, and null when the projection does not exist. A integer is also contained in each bucket which indicates a local ID for a valid patch, this local ID will then be used in a bitmap for duplicate elimination; or indicates an offset to the next valid grid cell in scanning path so that we could skip a column of consequent empty grids.

4. Query function

In order to return patches in order, a heap structure is implemented. Each time query function is called, all sets (leaves of hash tree root node) search matched patches and push the first of them (the smallest index) into the heap. When the heap is not empty, it pops its top node into the return vector and pushes the next matched patch from the same set. This could guarantee all patches returned are sorted. Since number of sets is small, size of heap could be treated as constant number which indicates the time cost for query is $O(k \log s)$ where k represents number of patches returned and s represents total number of patches sets.

```
Data: index range
Result: vector of returned patches
setup return vector V;
setup heap H;
for i=1 to number of sets do
    push first matched patch into heap H;
end
while heap H not empty do
    push top node of H into return vector V;
    pop H and push corresponding set's next matched patch into H if
    exists;
end
```

Algorithm 3: Query function

5.2 createNeighborhood

As we can see from the analysis results, STL set (red-black tree) is another data structure that is called frequently in regridding process. Both search and insert operations of it have time complexity of $O(\log n)$. In function createDetailedDependencies, STL set is mainly used for duplicates elimination and query whether a given element has been pushed. These features can be fully support by some more efficient data structure, such as bitmap.

5.2.1 d_neighbors

d_neighbors is a variable that is setup during creating neighborhood process, which is used to query whether a patch is in neighborhood of current processor. Since each patch has a unique ID, suppose the maximum number of patches is 1 million, the range of patch ID is from 0 to 999,999. Based on range of patch ID and function requirement, bitmap is the best choice in this situation for replacement of STL::set. If we assign 1 bit for each patch, the total memory cost will be approximately to only 1MB, which can be tolerant compared with total memory size. However, if we have to clean the bitmap each time we do a query, the time cost for constructor and deconstructor of bitmap will be dominant compared with time cost for query. So instead of assign 1 bit for each patch, 8 bits are assigned which extends number of function calls between deconstructor to $2^8 = 256$.

5.2.2 d_neighborProcessors

d_neighborProcessors is another variable used in function createNeighborhood. This variable is used to query whether a processor ID is in neighborhood, and store this ID if it's not existed. Since it is not required that processor IDs returned must be sorted, we can again optimize $O(\log n)$ time complexity for both query and insert to $O(1)$ using bitmap based vector.
Similar bitmap method is used in this case, each time we first query bitmap to find if corresponding processor ID exists, and set corresponding bit to true and push this ID into return vector if not exists, or skip if exists.

6 Results and Improvements

6.1 New Dependency Graph

Based on optimization of data structures described above, another dynamic analysis was executed and new component dependency graph is shown in Fig. 2:

As we can see from the Fig. 2, in function createNeighborhood, time cost for selectPatches (node 52) reduces from 33.98% to 21.83%, time costs for both insertion of d_neighbors and d_neighborProcessors reduce from 21.43% and 29.91% to less than 1% (filtered by threshold of 1% cannot be seen in the graph). Optimization of these three data structures reduces time cost for function createNeighborhood (node 40) from 47.09% of regridding to 18.95%. For function createDetailedDependencies, time cost for function getPatchesUnderDomain reduces from 15.25% to 11.47%, and selectPatches reduces from 4.3% to 3.6%.

Of overall time cost of regridding, time cost for selectPatches (node 52) reduces from 18.3% of total time cost for regridding to current 7%, and d_neighbors and d_neighborProcessors reduce from 14.1% and 10.1% to less than 1.0%.

6.2 Performance Improvements

A weak scalability test (CMCRT test) is executed from 520 patches with 17461008 cells on 1 processor to 66560 patches with 2192199696 cells on 128 processors. Time cost for all selectPatches (Fig. 5), createNeighborhood (Fig. 4) and overall regridding process (Fig. 3) are represented at the end of this report.

7 Conclusions and Future Work

7.1 Conclusions

Using dynamic analysis, we could easily identify which components are dominant in overall timecost so that we can take further actions to optimize the overall performance. Meanwhile, since Uintah framework's component design

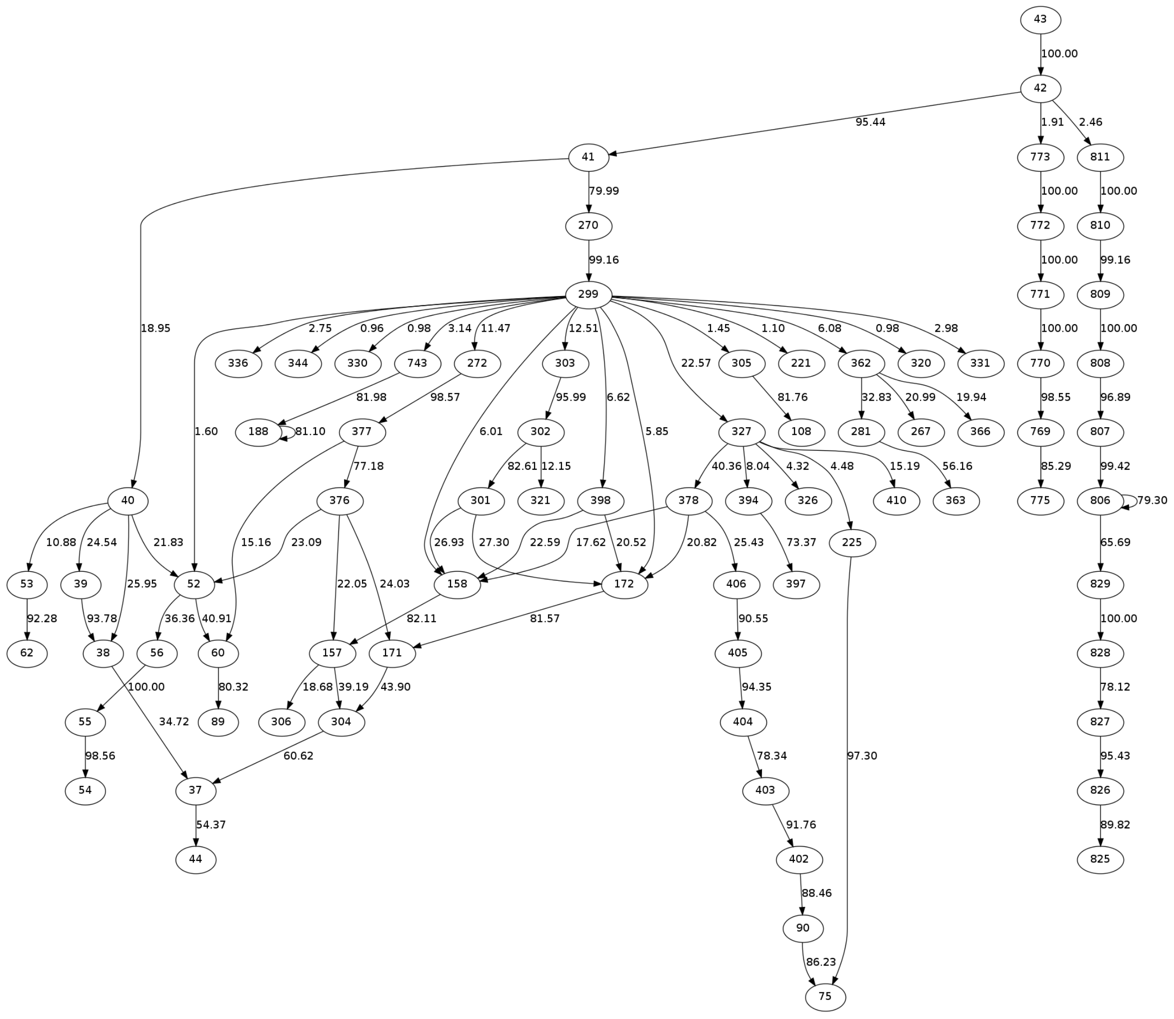

Figure 2: Component Dependencies after Optimization

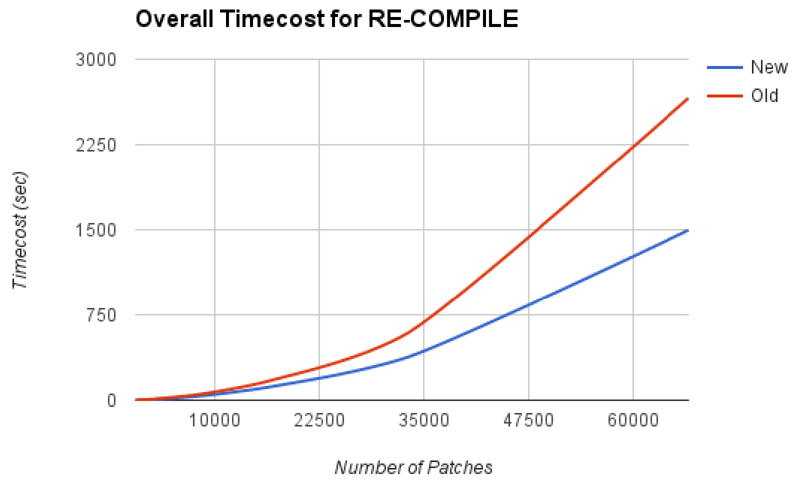

Figure 3: Timecost of regridding process

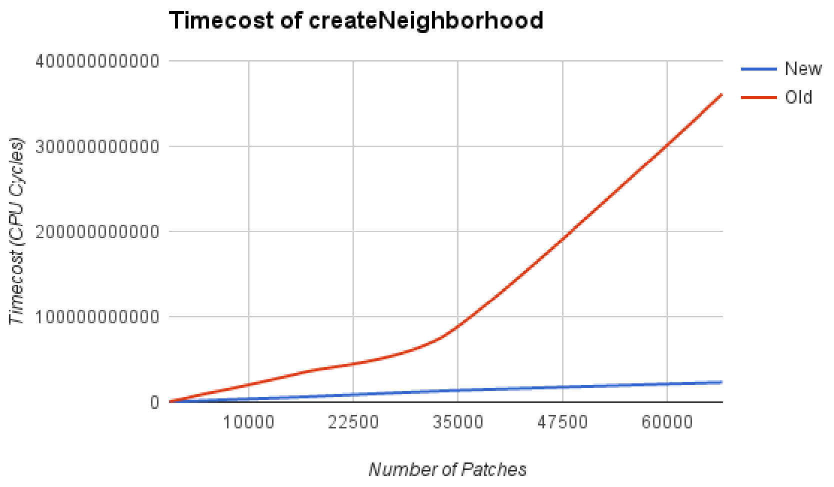

Figure 4: Timecost of createNeighborhood

allows us to replace its original data structures without changing user's interfaces or codes, it's convenient to replace previous structures or components with more efficient ones. And based on the analysis graph and performance tests, data structures do improve the scalability of regridding process.

7.2 Future Work

Current Uintah framework stores a copy of all patches in all levels on each processor which could lead to a intolerable metadata memory usage when dealing with large number of patches. We are now developing a new mechanism for storing patches where all patches are compressed and distributed across all processors. So each thread only has a partial view of the whole domain and communication requests will be sent when information on other patches is needed, like during regridding process.

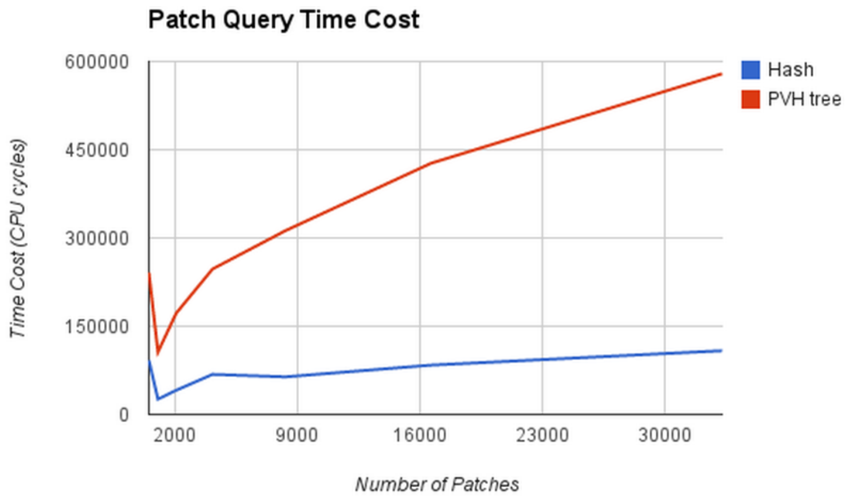

Figure 5: Timecost of selectPatches