A Diffusion-Based Processor Reallocation Strategy for Tracking Multiple Dynamically Varying Weather Phenomena

Preeti Malakar[*], Vijay Natarajan[*†], Sathish S. Vadhiyar[†], Ravi S. Nanjundiah[‡]
[*]Department of Computer Science and Automation, Indian Institute of Science, Bangalore, India
[†]Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India
[‡]Centre for Atmospheric and Oceanic Sciences, Indian Institute of Science, Bangalore, India

***Abstract*—Many meteorological phenomena occur at different locations simultaneously. These phenomena vary temporally and spatially. It is essential to track these multiple phenomena for accurate weather prediction. Efficient analysis require high-resolution simulations which can be conducted by introducing finer resolution nested simulations, *nests* at the locations of these phenomena. Simultaneous tracking of these multiple weather phenomena requires simultaneous execution of the nests on different subsets of the maximum number of processors for the main weather simulation. Dynamic variation in the number of these nests require efficient processor reallocation strategies. In this paper, we have developed strategies for efficient partitioning and repartitioning of the nests among the processors. As a case study, we consider an application of tracking multiple organized cloud clusters in tropical weather systems. We first present a parallel data analysis algorithm to detect such clouds. We have developed a tree-based hierarchical diffusion method which reallocates processors for the nests such that the redistribution cost is less. We achieve this by a novel tree reorganization approach. We show that our approach exhibits up to 25% lower redistribution cost and 53% lesser hop-bytes than the processor reallocation strategy that does not consider the existing processor allocation.**

***Index Terms*—redistribution; processor reallocation; data analysis; cloud tracking**

I. Introduction

Accurate and timely prediction of severe weather phenomena such as heavy rainfall, heat waves, and thunderstorms enables policy makers to take quick preventive actions. Such predictions require high-fidelity weather simulations. Multiple similar meteorological phenomena may occur at the same time in different regions of a geographical domain. For example, Figure 1 illustrates the phenomena of tall clouds occurring at multiple regions simultaneously in the Indian region.

Weather simulations need to track these clouds at higher resolutions. Simulating and tracking these multiple regions of interest at high resolutions is important in understanding the weather phenomena and for accurate weather predictions. These phenomena may vary temporally as well as spatially. Some of the regions of interest may disappear in subsequent time steps while new regions of interest may form. For example, some of the cloud systems[1] shown in Figure 1 may produce severe weather conditions such as high winds, intense localized rainfall, floods and storm surges over coastal regions, some clouds may move to different regions and cluster with other clouds, and some may disappear with time.

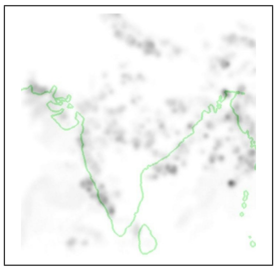


Fig. 1. Tall clouds over the Indian region during the 2005 monsoon season. Image generated from WRF simulation. Darker regions correspond to regions with higher cloud water mixing ratios.

Many meteorological phenomena occur at different locations simultaneously (requiring resolution of finer scales) and simulation methods have to efficiently handle this. Tracking these multiple phenomena incur new challenges which are more difficult to tackle than the simulation of a single phenomenon. We list some of the challenges below.

- Simultaneously tracking the appearance, disappearance and merging of the phenomena at various locations requires dynamic representations and efficient data analysis.
- Simulating multiple events at high resolutions needs efficient processor allocation schemes for these multiple events.
- The dynamic nature of these events require fast data redistribution strategies.

High-resolution nested simulations, *nests*, are spawned within the main parent simulation to simultaneously track these multiple regions of interest. Each nested simulation is executed on disjoint subsets of the total number of processors for high performance [1]. Therefore, with dynamic appearance and disappearance of regions of interest, we need to modify the processor allocation for the nests. The processors allocated to the nests in the previous time step of simulation will have to be freed if the nests do not exist in the current time step. Similarly, a subset of processors needs to be allocated to the

This research is partly supported by Centre for Development of Advanced Computing, India, project ref. no. CDAC/ESE/STV/C014.

[1]Cloud systems are a hierarchical organization of clouds.

new nests formed in the current time step. The removal of old nests and addition of new nests may lead to new processor allocation for the old nests which continue to exist from the previous time step.

Reconfiguration of processor allocation implies data redistribution for the old nests. In this work, we have developed a tree-based hierarchical diffusion algorithm for processor allocation that reduces data movement by considering the old processor allocation. This algorithm results in lower redistribution time as compared to a strategy that does not consider the existing processor allocation.

We have implemented the redistribution algorithm to support resource reconfigurations in an application that detects and tracks organized tropical convective cloud systems which have widespread occurrence of tall *cumulonimbus* clouds as a distinct signature. These clouds are associated with thunderstorms and atmospheric instability, forming from water vapour carried by powerful upward air currents. They can produce heavy rain and flash flooding. Hence, it is important to track these clouds. These clouds may form and disappear with time. We have developed a parallel data analysis algorithm that detects these clouds from simulation output. We spawn nests over these regions of interest within the running simulation, and also dynamically remove nests when the old regions of interest no longer exist. For weather simulation, we use the popular Weather Research and Forecast model (WRF) [2].

The key contributions of this paper are as follows:

1) A parallel data analysis algorithm to detect and track clouds on-the-fly.
2) A framework that supports dynamic nest formation and processor rescheduling within a running simulation.
3) A novel tree-based hierarchical diffusion algorithm for processor allocation that minimizes data redistribution cost.

We performed experiments on Blue Gene/L and Intel Xeon-based clusters using both real data corresponding to Mumbai rainfall of 2005, and synthetic nest formations and deletions for the same period. Our results showed that we were able to reduce the redistribution time by 25% over the scratch method and resulted in 53% lesser hop-bytes on Blue Gene/L.

While we have used the tracking of tall clouds as a case study, our algorithms for data analysis and processor allocation are generic and applicable to other scenarios that involve multiple dynamically varying nested simulations.

Section II describes related work in adaptive mesh refinement and graph partitioning. Section III presents our parallel data analysis algorithm. Section IV presents our data redistribution strategies. Section V presents experimental results to illustrate the performance improvement achieved. Section VI concludes and enumerates our future efforts.

II. Related Work

Many of the existing efforts in mesh adaptation and repartitioning have been in the domain of Adaptive Mesh Refinement (AMR) applications [3], [4], in which repartitioning and dynamic load balancing are critical components. Schloegel et al. present a multilevel diffusion scheme for repartitioning adaptive meshes [5]. A commonly used strategy for mesh repartitioning is to map the problem to graph partitioning and attempt to minimize the edge-cut representing the amount of communication between the partitions [3], [5], [6]. In our case, the partitions are the nests and there is no communication between the nests. Hence the graph partitioning heuristics are not directly applicable in our case.

Moreover, existing efforts in AMR [3], [7] and irregular mesh applications [5] perform repartitioning of adaptively refined meshes primarily to achieve dynamic load balancing. In our work, load balancing is implicitly achieved by partitioning the processor space into multiple disjoint rectangular grids that are assigned to the individual nested domains in proportion to the domain workloads. The primary focus of our work is repartitioning the rectangular processor grid into sub-rectangles while minimizing the data redistribution cost when nests dynamically appear and disappear. In our case, a rectangular process grid needs to be allocated for each nested domain and one processor executes a region of a nested domain. The new processor allocation after the redistribution is also a sub-rectangle of the rectangular process grid for the parent simulation. Thus we require to reform the existing rectangles such that there is maximum overlap between the old and new process grids (sub-rectangles) for the same nest in consecutive adaptation points.

Furthermore, in AMR applications, a grid is refined multiple times and inter-grid operations between the refinement levels are considered while repartitioning using Space Filling Curve (SFC) strategies like Hilbert ordering [8]. In our case the multiple high-resolution nests are formed at different locations in the simulation domain and there is no communication between these nests. We focus on minimizing the data redistribution and maximizing the overlap between the old and new rectangular process grids for the same nest. Hence SFC based repartitioning is not applicable for our domain.

Sinha et al. [4] presented an adaptive system sensitive partitioning of AMR applications that tune the partitioning parameters to improve overall performance. In these efforts, the virtual 2D process topology is not considered to make redistribution decisions. However, in our work we need to allocate a sub-rectangle of the 2D process grid to each nest. This requires us to consider the virtual 2D process topology in our redistribution algorithm for the selection of sub-rectangles.

III. Tracking Organized Cloud Clusters via Parallel Data Analysis

In this section, we describe an algorithm for parallel data analysis of simulation output. The algorithm analyzes the cloud water mixing ratio (QCLOUD) and outgoing long wave radiation (OLR) in WRF simulation output to detect tall clouds in tropical weather systems. These clouds are referred to as *cumulonimbus* clouds. They extend vertically from 1 km above the surface to more than 10 km. QCLOUD is the amount of liquid water contained in a cloud. Generally, high values of QCLOUD correspond to tall clouds. OLR is the infrared radiation at the top of the atmosphere. Coherent patterns of low OLR indicate occurrence of organized cloud systems (such as tropical depressions and cyclones) and would contain tall cumulonimbus clouds. A combination of OLR and QCLOUD better identifies such systems and precludes identification of isolated cumulonimbus (as QCLOUD alone would do) [9]. We use 200 as the upper threshold for OLR [10].

Each process running WRF generates output for its subdomain and writes into a *split* file. These split files are analyzed in parallel as shown in Algorithm 1. This algorithm forms contiguous, non-overlapping, and small clusters whose sizes do not grow uncontrollably. It is simple and fast and hence suitable for online analysis. Let P be the number of processes running WRF and N be the number of processes which analyze the QCLOUD values in the split files. The algorithm takes as input the split files $\{F_1, F_2, \cdots, F_P\}$. These split files are distributed to the N processes. Each of the N processes analyze k files (lines 1–2). The subset S of files, where $|S| = k$, is chosen as a rectangular subset of (Px, Py), where $Px{\cdot}Py = P$ is the rectangular process decomposition in WRF. Thus P is divided into N rectangular subsets.

The value of QCLOUD at each grid point in each split file is aggregated if the outgoing long wave radiation OLR $\leq$ 200 (lines 4–9). The fraction of the grid points which satisfy the above criteria, $olrfraction$, is calculated (lines 7–8). The aggregated QCLOUD values, one value per file, are then sent by all the N processes to a root process, rank 0 in our case. Each process will at most send k values. Note that some of the split files may not have regions with OLR $\leq$ 200, in which case the process owning these split files will send fewer than k values. The root process gathers the aggregated QCLOUD values and the $olrfraction$ values (line 11).

The rest of the algorithm is executed only on the root process. Firstly the aggregated QCLOUD values obtained from the split files are sorted in non-increasing order (line 13). A contiguous region with high cloud cover can span multiple split files processed by multiple processes. To obtain a contiguous region, we perform a variant of nearest neighbour clustering (NNC) (line 14). NNC outputs a set of clusters with each cluster containing a contiguous region of high cloud cover. A rectangle is formed around each cluster (lines 16–19) and these rectangles constitute nests for fine-resolution simulations in WRF.

Nearest Neighbour Clustering: The pseudo code for the NNC algorithm is shown in Algorithm 2. It takes as input the sorted list of QCLOUD values, *qcloudinfo*. Each element in *qcloudinfo* is a tuple of aggregate QCLOUD values for a split file and the corresponding fraction of the split file which has OLR $\leq$ 200. The QCLOUD value of each element in the list represent the cloud cover for a subdomain. The spatial location, i.e. the latitude and longitude extents of a subdomain is used in this algorithm to determine *proximity* between two subdomains.

The algorithm iterates over each element in the input array *qcloudinfo* (lines 2–20). Line 3 checks whether the aggregate QCLOUD value and the fraction of the subdomain that has OLR $\leq$ 200 are greater than a threshold, which is 0.005 in our case. This avoids analyzing smaller cloud-covered regions with a very low QCLOUD value. Clusters are formed based on proximity of the elements (lines 4–18). Each cluster represents a contiguous region of strong cloud cover. An element is added to a cluster if it is either 1-hop or 2-hop away from an existing cluster. Initially, the list of clusters is empty. First, we check if the current element is at 1-hop distance from any element in an existing cluster (lines 6–9). If this does not hold true, then we check if the element is 2 hops away from any element in an existing cluster (lines 10–13).

In lines 6 and 10, the DISTANCE function is invoked to calculate the proximity. If it returns true, the element is added to $list$. If $element$ is within hop distance from $member$, then it is added to the cluster $list$ iff it does not deviate the mean of the QCLOUD values by more than a threshold (30% in our case) (lines 23–29). This ensures that a cluster of contiguous cloud region has low standard deviation and also helps in controlling the size of an existing cluster.

If $element$ is not within 2 hops from any element in any of the existing clusters, then a new cluster $newlist$ is formed. $element$ is added to $newlist$, which is added to the set of clusters $Clusters$ (lines 16–18). NNC outputs $Clusters$ which is the set of clusters representing different contiguous regions of cloud cover.

```
   Input: Per-process simulation output of one time step from $P$
          processes $\{F_1, F_2, \cdots, F_P\}$, Number of processes for
          parallel data analysis $N$
   Output: $Rectangles$: Rectangular regions with high cloud water
           mixing ratio
   /* Divide $P$ files among $N$ processes                    */
 1 $k = P/N$;
 2 Let $S$ be the set of $k$ files assigned to each of the $N$ processes;
   /* Begin analysis of QCLOUD values in the files
      in $S$ by each of the $N$ processes                      */
 3 $count = 0$;
 4 foreach $file \in S$ do
 5     Read QCLOUD and OLR from $file$ for each grid point;
 6     Aggregate $qcloud$ and increment $count$ where
       $OLR[gridpoint] \leq 200 \; \forall \; gridpoint \in file$ ;
 7     Let $area$ be the total number of grid points in the file;
 8     $olrfraction = count/area$;
 9 end
   /* End analysis                                             */
10 $root = 0$;    /* Assume rank 0 is the root rank */
11 Root collects the $qcloud$ and $olrfraction$ information from every
   process in qcloudinfo;
   /* Form rectangular regions in $root$ process               */
12 if (my rank == root) then
13     Sort qcloudinfo in decreasing order of qcloudinfo.qcloud;
14     $Clusters$ = NNC(qcloudinfo);
15     $Rectangles = \emptyset$;
16     foreach ($list \in Clusters$) do
17         Let $item = (minX, maxX, minY, maxY)$ be set of the
           minimum and maximum of x and y coordinates of elements
           of $list$;
18         Add $item$ to $Rectangles$;
19     end
20 end
```

Algorithm 1: Parallel Data Analysis (PDA) algorithm

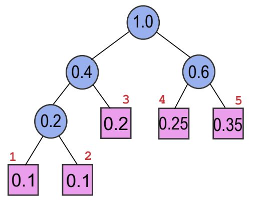
```
Input: Sorted array qcloudinfo
Output: Clusters: List of elements, clustered by proximity
1  $Clusters = \emptyset$;
2  LOOP: foreach element ∈ qcloudinfo do
3      if (element.qcloud ≥ threshold and
       element.olrfraction ≥ threshold) then
           /* Check if this element is physically
              close to any member of any list      */
4          foreach list ∈ Clusters do
5              foreach member ∈ list do
6                  if (DISTANCE (element,member,list,1)) then
7                      Add element to list;
8                      Continue next iteration of LOOP;
9                  end
10                 if (DISTANCE (element,member,list,2)) then
11                     Add element to list;
12                     Continue next iteration of LOOP;
13                 end
14             end
15         end
           /* Form a new list                      */
16         Initialize newlist;
17         Add element to newlist;
18         Add newlist to Clusters;
19     end
20 end
21 Return Clusters;
22 Begin Function DISTANCE (element, member, list, hop)
23 if (distance between member and element == hop) then
24     OldMean = Mean of QCLOUD values of members of list;
25     NewMean = Mean of QCLOUD values of members of list and
       element.qcloud;
26     if (NewMean is within 30% of OldMean) then
27         Return True;
28     end
29 end
30 Return False;
31 End Function DISTANCE
```

Algorithm 2: Nearest Neighbour Clustering (NNC) algorithm

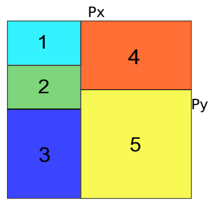
The parallel data analysis algorithm is executed simultaneously on a different set of processors than the processors running the WRF simulation. Hence execution of PDA does not affect WRF execution times. In Algorithm 1, the analysis of QCLOUD values in each split file is done in parallel because this is the most time-consuming step. For a maximum of 1024 split files, experiments show that the number of elements gathered at the root process is less than 200 for most of the time steps. The sequential NNC algorithm (Algorithm 2) takes less than a second to cluster such few values. In this case, parallel clustering would have been an overkill for online analysis. However, we would like to parallelize the NNC algorithm in future for simulations on larger number of processors.

IV. Processor Allocation

The parallel data analysis (PDA) algorithm computes a set of regions of interest (ROI) in the domain, which in our case are the regions with high cloud cover. Nested simulations are spawned over the regions of interest. We simulate these nests at high resolutions for better accuracy. The resolutions of these



(a) Huffman tree for 5 nests with execution time ratios 0.1 : 0.1 : 0.2 : 0.25 : 0.35



(b) Sub-division of the processor grid $Px \times Py$ for the 5 nests.

Fig. 2. Illustration of processor allocation for nests.

nested simulations are thrice that of the parent simulation. We modified the WRF code to spawn nests on-the-fly without stopping the simulation. The initial data for the nested domains are interpolated from the parent domain.

In a recent work [1], it was shown that significant performance improvements can be achieved by executing the nests simultaneously on different subsets of the total number of processors, P. We use the performance modeling and Huffman tree based algorithm in [1] to determine the size of the subset of processors for a nest and the position of the subset in the processor grid $Px \times Py$ where $Px{\cdot}Py = P$. The performance model is used to predict the execution times of nests based on the size and aspect ratio of the nests. The Huffman tree based algorithm is used to determine the initial processor allocation for each nested domain.

An example of processor allocation for 5 nests is shown in Figure 2. Assume that the ratios of the predicted execution times of the nests are $0.1 : 0.1 : 0.2 : 0.25 : 0.35$. These ratios are used as weights in the construction of the Huffman tree, as shown in Figure 2(a). The corresponding processor sub-grid for each nest is shown in Figure 2(b). The 5 sub-rectangles correspond to the set of processors that execute each of the nests. The start rank i.e. the rank of the processor at the north-west corner of the sub-rectangle and the rectangular dimensions of each processor sub-grid for this example configuration are shown in Table I for a maximum of 1024 cores.

TABLE I
Processor allocation on 1024 cores

Nest ID	Start Rank	Processor sub-grid
1	0	13×8
2	256	13×8
3	512	13×16
4	13	19×13
5	429	19×19

The regions of interest may persist in time or disappear in subsequent time steps. Our regions of interest are the regions with high cloud cover. Clouds may form and disappear over a period of time. The PDA algorithm is invoked periodically (every 2 minutes) to detect regions of interest (ROI) in the output of the current simulation time step. A nest is spawned whenever a new ROI is detected. A nest is deleted when an existing ROI is not output by PDA. A *retained* nest is one which was output by PDA in the previous invocation as well as in the current invocation. The insertion, deletion and retainment of nests cause changes in the Huffman tree structure and hence in the processor allocation. Therefore the newly allocated set of processors (*receivers*) executing a retained nest may not be the same as the previously allocated set of processors (*senders*) for the nest. The *senders* need to distribute the nest domain data to the *receivers*. We modified the WRF code to execute this redistribution. First the amount of data to be redistributed is calculated based on the nest size, followed by MPI_Alltoallv to redistribute data for each nest. The processors that are neither senders nor receivers for a nest send and receive 0 value during the MPI_Alltoallv for that nest.

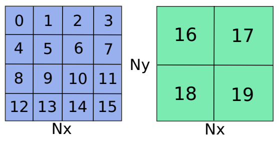


Fig. 3. Data redistribution from old to new set of processors assigned to a nest.

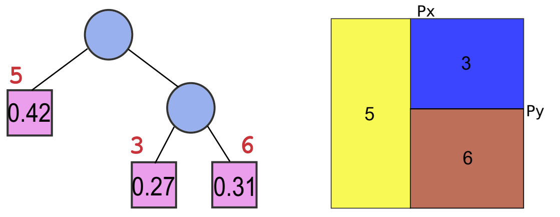
An example is shown in Figure 3 for a nest size of $Nx \times Ny$. A nest is equally subdivided among its allocated processors $0 - 15$ as shown in the left grid. These processors distribute the nest data to the newly allocated processors $16 - 19$ as shown in the right grid of the figure. It can be observed that the region of the nest domain that processor 16 owns was previously owned by $0, 1, 4, 5$. Hence 16 receives the domain data from $0, 1, 4, 5$. Similarly, the other receivers also receive data from 4 senders in this example.

In the above example, the senders and receivers are non-intersecting sets. The communication cost for the data redistribution between the senders and the receivers can be minimized if the senders and receivers overlap. In torus networks, minimizing the number of hops between the senders and receivers can minimize the redistribution cost. We describe two strategies for data redistribution in the next section.

A. *Partition from scratch*

In this approach, we partition the entire process grid $Px \times Py$ for processor allocation based on Huffman tree constructed using the predicted execution times of the nests as weights, as explained in the previous section. The tree construction does not consider the current allocation of processors. Hence this strategy can lead to completely non-overlapping senders and receivers, which will lead to increased redistribution cost.

For example, let us consider the configuration in Figure 2. Assume that in the next invocation, PDA outputs the nests $3, 5, 6$ as regions of interest. So the nests $1, 2,$ and 4 will be deleted and new nest 6 will be formed. Let the ratios of the predicted execution times of the nests $3, 5, 6$ be $0.27 : 0.42 : 0.31$. The corresponding Huffman tree and the processor partition



(a) Huffman tree for nests $3, 5, 6$ with execution times in ratios of $0.27 : 0.42 : 0.31$.

(b) Sub-division of the processor grid $Px \times Py$ for 3 nests.

Fig. 4. Processor allocation for nests using partition from scratch.

are shown in Figure 4. The start rank and the rectangular dimensions of each processor sub-grid for each nest are given in Table II for a maximum of 1024 cores. Comparing the previous and the new allocation for nests 3 and 5 from Tables I and II, we can observe that there is no overlap between senders and receivers. This can increase the redistribution cost.

TABLE II
PROCESSOR ALLOCATION ON 1024 CORES

Nest ID	Start Rank	Processor sub-grid
3	13	19×13
5	0	13×32
6	429	19×19

The redistribution cost may be high in some cases in this approach. However, the rectangular partitions based on the Huffman tree are as square-like as possible owing to the tree construction in the order of increasing weights. The square-like partitions minimize the execution times of the nests.

B. *Tree-based hierarchical diffusion*

In this approach, we try to maximize the overlap between senders and receivers of the *retained* nests. The key idea is to shift the boundaries of rectangular partitions for the retained nests so that the distribution of data is among neighbouring processes and the overlap in the nest data between the old and new set of processes is maximized. This minimizes the redistribution cost, especially on torus networks. An example is illustrated in Figure 5. Figure 5(a) shows the existing processor partitioning for nests $1, 2, 3$. When a new nest is added, the existing partitions are shrunk. In this example, the right boundary of rectangle for nest 1 is shifted to the left and the left boundaries of the nests 2 and 3 are shifted to the right, thereby leaving some processors free for inserting the new nest, as shown in Figure 5(b). This also leads to a large overlap between the old and new processor partitions for nests $1, 2, 3$.

This repartitioning method is based on modifying the tree corresponding to the current allocation, rather than building the Huffman tree from scratch. The positions of the nodes corresponding to the retained nests are kept intact in the tree. Note that the weights of the old nodes, i.e. the retained nests, may be modified because the weights represent the ratios of

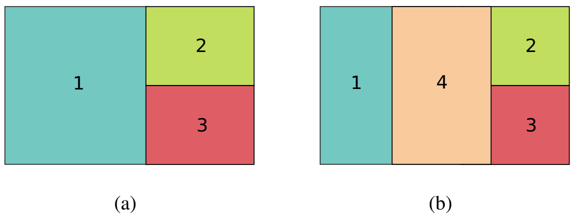


Fig. 5. (a) Existing and (b) new processor allocation in the hierarchical diffusion approach.

the number of processors that will execute each nest. When new nests are added and/or old nests are deleted, the processor shares of the existing nests may change.

When there is no deletion, and there is only insertion of new nodes, they are inserted near those existing nodes whose weights are similar to those of the new nodes. By inserting a new node near a node in the Huffman tree with similar weight, we attempt to obtain rectangular partitions for the nests that are more square-like. However, inserting a new node near a node with large difference in weights will lead to skewed rectangles. As reported in [1], square-like partitions lead to smaller executions times for the nests, while skewed rectangular partition increases the execution time of a nest.

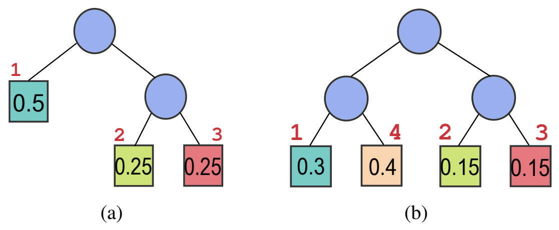


Fig. 6. (a) Existing and (b) new trees in the hierarchical diffusion approach. Predicted execution time ratios of the nests are the weights in the leaf nodes.

This is illustrated in the example shown in Figure 5. The existing and the new trees corresponding to the processor partitions of Figure 5 are shown in Figure 6. The new tree in Figure 6(b) is constructed by inserting node 4 near node 1. This is because the weight of node 4 is closest to that of the new weight of node 1. The size of a partition that each node gets is proportional to its weight. Thus, the nodes 1 and 4 get $\frac{3}{7}^{th}$ and $\frac{4}{7}^{th}$ of the processors allocated to their parent node. Since the difference in weights of nodes 1 and 4 is less, so the resulting rectangles for 1 and 4 will be as square-like as possible. Note that this would not have been the case if node 4 was inserted near node 2 whose weight is 0.15. This is because the corresponding shares for 4 and 2 would have been $\frac{0.4}{0.55} = \frac{8}{11}$ and $\frac{0.15}{0.55} = \frac{3}{11}$. Thereby the rectangle for node 2 would not have been square-like due to the large difference in weights. This is illustrated in Figure 7. One can note that rectangle 2 is skewed as compared to rectangle 4.

When nests are both inserted and deleted, the nodes corresponding to the deleted nests are deleted from the tree. Further, new nodes are inserted in the positions of deleted nests so

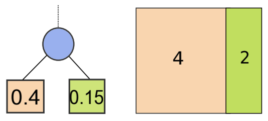


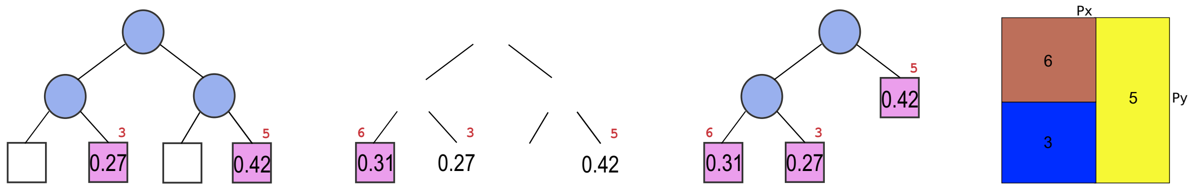
Fig. 7. Skewed rectangle due to large difference in weights of the two nodes.

that the positions of the retained nests remain intact as much as possible. This may enhance the chance of overlapping old and new nest processor allocations for the retained nests. The algorithm for modifying the existing tree for new processor allocation is detailed in Algorithm 3. The inputs are the existing tree $oldtree$, the list of deleted nodes $deletednodes$, the modified weights of the retained nests $rweights$ and the weights of the new nodes $nweights$. The output is the modified tree $newtree$.

```
Input: Existing tree $oldtree$, list of deleted nodes $deletednodes$,
       new weights of retained nests $rweights$, and weights of new
       nests $nweights$.
Output: New tree $newtree$
1  $freenodes = \emptyset$, $siblings = \emptyset$;
2  foreach $node \in deletednodes$ do
3      Mark $node$ as free in the $oldtree$;
4      Add $node$ to $freenodes$;
5      Add sibling of $node$ to $siblings$;
6  end
7  foreach $weight \in rweights$ do
8      Update $weight$ for the corresponding retained node;
9  end
10 Update weights of internal nodes of $oldtree$;
   /* Insert in the positions of deleted nodes,
      near to the nodes with closest weights      */
11 foreach $new\_weight \in nweights$ do
12     if $(|freenodes| > 1))$ then
13         Add $new\_weight$ to the position of $node$, where
           $node \in freenodes \wedge sibnode$ is sibling of $node \wedge$
           $d = Weight(sibnode) - new\_weight \wedge$
           $d = \underset{\forall\ s\ \in\ siblings}{\text{argmin}} (Weight(s) - new\_weight)$
14         Delete $node$ from $freenodes$;
15         Delete $sibnode$ from $siblings$;
16     end
17 end
18 if $(|nweights| \geq |deletednodes|))$ then
19     Build Huffman tree for the remaining new weights rooted at
       $node \in freenodes$;
20 else
21     Delete the remaining nodes in $freenodes$ from $oldtree$;
22 end
23 Copy $oldtree$ to $newtree$;
```

Algorithm 3: Tree-based hierarchical diffusion algorithm

Firstly, nodes from $deletednodes$ are marked as free in $oldtree$ and added to the set $freenodes$ (lines 2–6). The siblings of these nodes are added to the set $siblings$ (line 5). These are used later as insertion points. The weights of the retained nodes are modified (lines 7–9). Based on the deletion and modification of weights of retained nodes, the weights of the internal nodes are updated (line 10). The new weights are added in the positions of the deleted nodes (lines 11–17). As explained above, the new nodes should be inserted near the ones who have closest weights. So, we inspect the



(a) Deleted nodes marked empty and weights of retained nests modified.

(b) Node 6 inserted near node 5

(c) Remaining deleted nodes removed

(d) Sub-division of the processor grid based on the modified tree.

Fig. 8. Steps of the tree-based hierarchical diffusion algorithm for deleting nests $1, 2, 4$, retaining nests $3, 5$ and adding new nest 6.

weights of the sibling nodes of the deleted nodes. Inserting a new node in the place of a deleted node will lead to minimum modification of the existing tree structure. This is shown in line 13. new_weight is inserted in the position of $node$, which was marked empty earlier. $node$ is selected such that the difference between the weight of its sibling $sibnode$ and new_weight is minimum. $node$ and $sibnode$ are deleted from their respective sets (lines 14–15).

Note that the operation in line 13 is done only when there are multiple nodes in the set $freenodes$. This is because when the number of deletions is less than the insertions, we build Huffman tree using the remaining unmatched weights in $nweights$, and this subtree is rooted at the position of the last element in $freenodes$. This is shown in lines 18–20. If there are fewer insertions than deletions, we delete the remaining nodes of $freenodes$ (line 21). The updated $oldtree$ is output as $newtree$.

This approach reduces the data movement between the senders and receivers and hence achieves significant reduction in redistribution time as compared to the partition from scratch method. This is because we attempt to allocate receivers such that there is large overlap between senders and receivers and the receivers are neighbouring processes of the senders.

The processor allocation using tree-based hierarchical diffusion algorithm for the example in Figure 2 is shown in Figure 8. To compare with the *partition from scratch* approach, let us assume the same output of PDA that was considered in Section IV-A (see Figure 4). The nests $1, 2$ and 4 are deleted, nests 3 and 5 are retained and 6 is the new region of interest. Figure 8(a) shows the tree after nodes $1, 2$ and 4 are marked as deleted and weights of 3 and 5 are modified. Note that deleted nodes $1, 2$ have been combined as one empty node because the two free rectangles represented by them can be considered as one free rectangle. Hence there are two free slots available for inserting new node 6 - the weight of one sibling node is 0.27 and that of the other is 0.42. Node 6 is inserted in the position of sibling of node 3 because $0.31 - 0.27 < 0.42 - 0.31$ i.e., the weight of node 3 is closer to weight of node 6. The rectangular partitioning based on this tree is shown in Figure 8(d). Comparing this with the partitioning obtained from the partitioning from scratch method shown in Figure 4(b), we can see that there is considerable overlap between the old and new set of processors for nests 3 and 5, as compared to no overlap in the partition from scratch approach. Also, we observe that the rectangles for 3 and 5 expand to neighbouring processes because we try to keep the positions of retained nests as intact as possible.

Note that the resulting modified tree may no longer be a Huffman tree in this approach. However, the modifications lead to some overlap between new and old processors and redistribution among neighbouring processes. Our techniques are scalable for large number of processors. Also, the maximum number of hops between old and new set of processors is likely to increase for the scratch method with larger total processor count. Therefore the data redistribution time may increase with increase in number of processors for the scratch method. Processor reallocation via Huffman tree construction or reorganization depends on the number of nests and is not affected by increase in processor count.

C. Dynamic Strategy

The performance differences between the two methods, namely, the partition from scratch method and our diffusion-based method, depend on both the execution times of the resulting partitions and the redistribution costs. The execution time ratios of the nests and hence the percentage of total number of processors allocated for the nests are same in both partition from scratch method and our diffusion-based method. However, due to integral sides of the sub-rectangles, the rectangular grids and the aspect ratios of the rectangles for the same nest configuration may not be exactly the same. For example, one method may allocate 16×18 while the other may allocate 17×17. This can lead to slight difference in execution times of the nests for the two methods.

Similarly, while we expect the redistribution costs for our diffusion-based method to be smaller than the partition from scratch method, there may be cases when the redistribution costs are almost same in both approaches. This is because both approaches are based on tree construction using the ratios of predicted execution times of nests as weights. The relative order of the weights affect the construction of the tree, and hence also affects the resulting rectangular processor grid allocated to the nests. Similar relative order of the weights of those nests that persist between reconfigurations may result in similar trees for both approaches, and hence similar redistribution costs. Therefore we propose a dynamic strategy that selects the approach which requires minimum redistribution time and execution time. For this, we need to predict both these times.

1) Performance model for redistribution time: The primary component of the redistribution time is MPI_Alltoallv between the processors. We assume direct algorithm for MPI_Alltoallv [11] between the processors in mesh and torus based networks. We predict MPI_Alltoallv time as the maximum communication time between senders and receivers. First, we find the size of the message that a sender will send to its receiver(s), and then find the number of hops between the sender and its receivers. Using this, we find the communication time for every sender-receiver pair. The maximum of these communication times is predicted as the time for MPI_Alltoallv. For non-mesh networks like switched networks, the times taken for sender to send messages to all receivers can be added to predict the time for MPI_Alltoallv.

2) Performance model for execution time: We profiled the execution times of a small set (size = 13) of domains with different domain sizes on a few (10 in our case) processor sizes within the maximum number of processors (1024 in our case). The actual execution times of these 13 domains are used to interpolate the execution times of the nests formed in our simulation using Delaunay triangulation. The details of these steps are presented in [1]. Additionally, we predict the execution times of the nests for the 10 processor sizes. Using these times, we perform linear interpolation to predict the execution time on desired number of processors. This gives good prediction accuracies as shown later in Section V. The prediction execution times are used for dynamic selection of methods, and also for determining the weights of the nests needed for processor allocation in the partition from scratch and our tree-based methods.

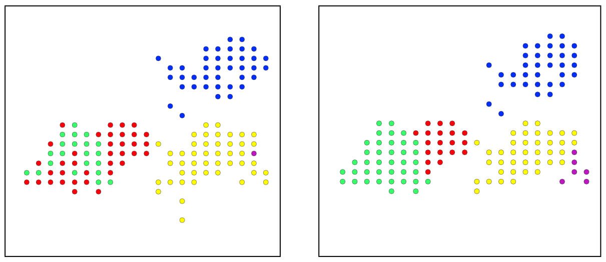
Using the above predictions for redistribution and execution times for both scratch and tree-based approaches, the dynamic scheme selects the one which has lower sum of these times.

V. Experiments and Results

A. Data analysis algorithm

One of the primary components in our work is the data analysis algorithm described in Section III to identify clouds and form nests. We form clusters of contiguous regions with high cloud cover using QCLOUD values in non-increasing order. A QCLOUD value in this list represents the aggregated QCLOUD over a subdomain, where OLR $\leq$ 200. The contiguous regions are clustered based on the proximity between the subdomains.

In this section, we compare our nearest neighbour clustering algorithm described with a simple nearest neighbour clustering approach. In Figure 9(a), we show the clustering using only 2 hop distance criteria. This strategy checks whether the list element is within 2 hops from an existing cluster. We can observe there are some overlapping clusters. In Figure 9(b), we show the clusters formed by our method. It can be observed that the clusters formed by our method are non overlapping because we first check for 1 hop and then 2 hop distance. We check for 2 hop distance only if the list element is not within 1 hop from an existing cluster. This ensures that the list element is added to its nearest cluster. We insert into a cluster only if the mean deviation is not more than 30% to ensure that the cluster size does not grow uncontrollably.



(a) Nearest neighbour clustering using 2-hop distance and no mean deviation criteria. Clusters overlap in space.

(b) Nearest neighbour clustering using 1-hop and 2-hop distances and mean deviation threshold of 30%. Clusters do not overlap.

Fig. 9. Nearest neighbour clustering for our parallel data analysis algorithm.

B. Domain Configurations

We used WRF v3.3.1 [2] for all our experiments. The parent simulation domain in WRF can have multiple child domains, called *nests*. These nests were formed during the simulation over different regions of interest. We modified the WRF source code for dynamic insertion and deletion of nested domains. We simulated over the Indian region from 60°E - 120°E and 5°N - 40°N for the July 2005 Mumbai rainfall event [12]. The period of simulation was from July 24, 2005 18:00 hours – July 27, 2005 18:00 hours. The parent simulation resolution was 12 km and the resolutions of the nested domains were 4 km. We compared our tree-base hierarchical diffusion approach with the partition from scratch method for both real and synthetic test cases. For the dynamic approach, we experimented with synthetic test cases.

Real: Nests were formed over regions with high cloud cover, which were detected by our parallel data analysis algorithm. The maximum number of nests formed during these runs were 7. The maximum and minimum sizes of the nests formed were 202×349 and 175×175. There were approximately 100 reconfigurations of processor allocations for the nests.

Synthetic: The real traces for our application had fewer configuration changes and fewer (4 – 5) nests on average. We generated some synthetic test cases in order to test our algorithm for higher number of nests in a time step and more number of redistributions per adaptation point. We tested with up to 70 random nest configuration changes, with number of nests varying between 2 – 9. Nests were randomly inserted and deleted. The maximum and minimum sizes of the nests formed were 361×361 and 181×181.

C. Experimental Setup

We performed our simulations on two different kinds of systems, a Blue Gene/L system and an Intel Xeon cluster called *fist*. Table III details our experimental configurations. For the experiments on Blue Gene/L [13], we developed a folding-based topology-aware mapping [14] that maps the neighbouring processes to neighbouring processors on the 3D torus. This topology-aware mapping was used for all our experiments so that the processes are one hop away from their neighbours in the process grid. This also benefits the execution times for both the partition from scratch method and diffusion based approach. For all our experiments, visualization was performed on a graphics workstation in Indian Institute of Science (IISc) with a Intel(R) Pentium(R) 4 CPU 3.40 GHz and an NVIDIA graphics card GeForce 7800 GTX.

TABLE III
SIMULATION CONFIGURATIONS

Simulation Configuration	*Maximum Number of Cores*
Blue Gene/L: Dual-core 700 MHz PowerPC 440 processor cores with 1 GB physical memory, 3D torus network	1024
fist: 2 Xeon quad core processors (2.66GHz, 12MB L2 Cache) with 16GB memory, connected by Infiniband switched network	256

D. Improvement in redistribution time

Our tree-based hierarchical diffusion method achieved 14% and 12% improvements in redistribution times on 512 and 1024 Blue Gene/L cores respectively over partition from scratch method for the real test cases.

TABLE IV
AVERAGE IMPROVEMENT IN REDISTRIBUTION TIMES FOR SYNTHETIC TEST CASES

Simulation Configuration	*Improvement*
BG/L 1024 cores	15%
BG/L 256 cores	25%
fist 256 cores	10%

Table IV shows the average percentage improvement in redistribution times for our tree-based hierarchical diffusion method over partition from scratch method for the synthetic test cases. It can be observed that the performance improvement is higher in the case of Blue Gene/L which has 3D torus network. This is because our tree-based hierarchical approach selects the new processor allocation based on the neighbours in the process grid. For Blue Gene/L the neighbours in the process grid are also neighbours in the processor topology because of our topology-aware mapping. However, in the *fist* cluster, there is no regular mesh/torus topology, hence the gains are lower. However, it is important to note that we still achieve 10% improvement over the scratch method because of the overlap between the senders and receivers in our approach. Maximum overlap ensures less data communication during the redistribution. We also observe higher improvement for 256 cores. We assume that this may be because of larger per-core data for redistribution in the case of smaller number of cores.

For both real and synthetic test cases, we observed an average of 4% increase in execution times for our approach over the partition from scratch method. This is because in our approach, the Huffman tree is not constructed from scratch and we try to maximize the overlap. Hence the resulting partitions may not always be square-like. However, when the number of adaptation points is high, it is more important to minimize the redistribution cost.

E. Distance between senders and receivers

Figure 10 shows the average hop-bytes during the sender-receiver communication for partition from scratch and our approach for 70 synthetic test cases on 1024 Blue Gene/L cores. The hop-bytes metric is the weighted sum of message sizes where the weights are the number of hops (links) traveled by the respective messages. Higher hop-bytes is an indication of higher communication load on the network [15]. It can be seen that the average in the case of partition from scratch is 5.25 whereas in our approach the average is 2.44. This is because in our strategy the receiver process grid is placed closer to the sender process grid so that the number of hops between a sender-receiver pair is minimized.

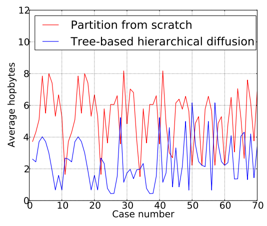


Fig. 10. Average hop-bytes for partition from scratch method and tree-based hierarchical approach. X-axis denotes the test case number and Y-axis denotes the hop-bytes. Tree-based hierarchical approach incurs lesser hop-bytes than scratch method.

Figure 11 shows the percentage of overlap of data points between the senders and receivers for partition from scratch and our approach for 70 synthetic test cases on 1024 Blue Gene/L cores. It can be observed that the overlap is higher for our method and hence it incurs lesser redistribution time.

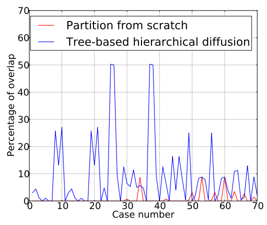


Fig. 11. Percentage overlap between senders and receivers for partition from scratch method and tree-based hierarchical diffusion approach. X-axis denotes the test case number and Y-axis denotes the percentage overlap. Tree-based hierarchical approach has more overlap than scratch method.

In the case of *fist* cluster, we found that there was an overlap of 27% data points between senders and receivers for our tree-based hierarchical approach. For the scratch method, there was 15% overlap. This is because in our method, we try to maximize the overlap between senders and receivers so that there is less data communication during the redistribution.

F. Dynamic Approach

In this section, we present the results for our dynamic scheme which selects either the scratch or the tree-based approach. We tested 12 reconfigurations for synthetic cases on 1024 BG/L cores for a simulation period of 4 hours. The approach with the minimum sum of predicted execution and redistribution times was selected by the dynamic approach. Since the efficiency of dynamic selection approach depends on the ability to predict the execution times of different nest configurations, we calculated the Pearson's correlation coefficient between the actual and predicted execution times. We found that our prediction method yielded Pearson's correlation coefficient of 0.9. This shows linear relationship between the two and hence also shows that our performance prediction for execution times is nearly accurate.

Out of the 12 reconfiguration cases, scratch method was selected two times and tree-based approach was selected ten times. The dynamic approach made correct decisions in 10 out of the 12 cases. In terms of actual execution times, our tree-based diffusion method gave smaller sum of execution and redistribution times than partition from scratch method in 9 cases, while the partition from scratch method gave smaller sum in the remaining 3 cases.

Figure 12 shows the total times including the execution times and redistribution times for tree-based approach, partition from scratch method and dynamic approach. It can be observed that the redistribution time is lowest in our tree-based method, while the execution time is the lowest in the partition from scratch method. The dynamic selection approach combines the advantages of both the methods, with its redistribution time similar to the tree-based approach and its execution time similar to the partition from scratch method. The dynamic scheme resulted in 3% improvement in overall execution time than the next best-performing tree-based approach. It should be noted that more frequent adaptation points seen in our real runs (about 70 adaptation points) will result in higher performance improvement for the dynamic scheme.

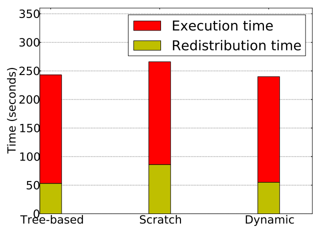


Fig. 12. Execution and redistribution times.

VI. Conclusions and Future work

In this work, we presented a parallel data analysis algorithm and efficient processor reallocation algorithm to detect and track tall clouds in tropical weather systems. Our data analysis algorithm detects organized cloud systems using a variant of nearest neighbour clustering. We performed nested high-resolution simulations for the regions with high cloud cover. The nested simulations were executed on a disjoint subset of the total number of processors. Due to the dynamic nature of the clouds, the nests may form and disappear with time. We proposed a tree-based efficient processor allocation algorithm that reallocates processors for the persistent nests at a low data redistribution cost.

Our approach considers the existing processor allocation and selects a new subset of processors with maximum overlap with the existing rectangular subset of processors. Results showed that we are able to reduce the redistribution times by up to 25% as compared to partition from scratch method with minimum increase in the execution times. We also developed a dynamic scheme that attempts to select the best of the two approaches, namely, partition from scratch and our approach.

Our detection and tracking algorithms are quite generic. In future, we would like to apply these algorithms for other applications which require simultaneous tracking of multiple dynamic events.

References

[1] P. Malakar, T. George, S. Kumar, R. Mittal, V. Natarajan, Y. Sabharwal, V. Saxena, and S. S. Vadhiyar, "A Divide and Conquer Strategy for Scaling Weather Simulations with Multiple Regions of Interest," in *Proceedings of the 2012 ACM/IEEE conference on Supercomputing*.

[2] W. C. Skamarock and et al., "A Description of the Advanced Research WRF version 3," *NCAR Technical Note TN-475*, 2008.

[3] L. Oliker and R. Biswas, "Efficient load balancing and data remapping for adaptive grid calculations," in *Proceedings of the ninth annual ACM Symposium on Parallel Algorithms and Architectures*, 1997, pp. 33–42.

[4] S. Sinha and M. Parashar, "Adaptive System Sensitive Partitioning of AMR Applications on Heterogeneous Clusters," *Cluster Computing*, vol. 5, 2002.

[5] K. Schloegel, G. Karypis, and V. Kumar, "Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes," *Journal of Parallel and Distributed Computing*, vol. 47, pp. 109–124, 1997.

[6] G. Karypis and V. Kumar, "Multilevel k-way Partitioning Scheme for Irregular Graphs," *Journal of Parallel Distributed Computing*, vol. 48, no. 1, pp. 96–129, 1998.

[7] Z. Lan, V. E. Taylor, and G. Bryan, "Dynamic load balancing of SAMR applications on distributed systems," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*.

[8] H. Sagan, *Space-Filling Curves*. Springer-Verlag, 1994.

[9] D. Rosenfeld. and I. M. Lensky, "Satellite-based insights into precipitation formation processes in continental and maritime convective clouds," *Bulletin of the American Meteorological Society*, vol. 79, pp. 2457–2476, 1998.

[10] G. Gu and C. Zhang, "Cloud components of the Intertropical Convergence Zone," *Journal of Geophysical Research: Atmospheres*, vol. 107, no. D21, pp. ACL 4–1–ACL 4–12, 2002.

[11] S. Kumar, Y. Sabharwal, R. Garg, and P. Heidelberger, "Optimization of All-to-all Communication on the Blue Gene/L Supercomputer," in *International Conference on Parallel Processing*, 2008.

[12] S. Sahany, V. Venugopal, and R. Nanjundiah, "The 26 July 2005 heavy rainfall event over Mumbai: numerical modeling aspects," *Meteorology and Atmospheric Physics*, vol. 109, pp. 115–128, 2010.

[13] IBM Blue Gene Team, "Overview of the Blue Gene/L system architecture," *IBM Journal of Research and Development*, vol. 49, 2005.

[14] H. Yu, I.-H. Chung, and J. Moreira, "Topology Mapping for Blue Gene/L Supercomputer," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*.

[15] A. Bhatele, G. Gupta, L. V. Kale, and I.-H. Chung, "Automated Mapping of Regular Communication Graphs on Mesh Interconnects," in *International Conference on High Performance Computing*, 2010.