Adaptive Workload Partitioning and Allocation for Data Intensive Scientific Applications

Xin Yang and Xiaolin Li

Abstract Scientific applications are becoming data intensive, and traditional load-balance solutions require reconsideration for scaling data and computation in various parallel systems. This chapter examines state-transition applications, which is a representative scientific application that handles grand-challenging problems (e.g., weather forecasting and ocean prediction) and relates to intensive data. We propose an adaptive workload partitioning and allocation scheme for parallelizing state-transition applications in various parallel systems. Existing schemes insufficiently balance both computation of complicated scientific algorithms and increasing volumes of scientific data simultaneously. Our solution addresses this problem by introducing a time metric to unify the workloads of computation and data. System profiles in terms of CPU and I/O speeds are considered for embracing system diversity, suggesting accurate estimation of workload. The solution consists of two major components: (1) an adaptive decomposition scheme that uses the quad-tree structure to break up workload and manage data dependency; and (2) a decentralized scheme for distributing workload across processors. Experimental results from real-world weather data demonstrate that the solution outperforms other partitioning schemes, and can be readily ported to diverse systems with satisfactory performance.

1 Introduction

Modern scientific applications are becoming data intensive and increasingly rely on various computing systems to analyze data and discover insights quickly. Low-cost sensors and other scientific technologies (e.g., fine-granularity computing models) drive the increase of scale of scientific data. Scientists start to explore new computing systems such as clouds to scale data and computation in an extended deployment while spending reduced cost. As a consequence, solutions effective in traditional computing settings require reconsideration for performance purpose. In this chapter, we use *state-transition scientific application* as an example to

X. Yang (✉) • X. Li
Department of CISE, University of Florida, Gainesville, FL, USA
e-mail: xin@cise.ufl.edu; andyli@ece.ufl.edu

X. Li, J. Qiu (eds.), *Cloud Computing for Data-Intensive Applications*,
DOI 10.1007/978-1-4939-1905-5_6

discuss modifications when porting traditional solutions to new computing settings. Specifically, the adaptive partitioning and allocation methods for processing state-transition applications in a load-balance manner are examined.

State-transition applications tackle grand-challenge scientific problems (e.g., weather forecasting [1] and ocean prediction [2]). They simulate the evolvement of environments, named *states*, such as atmosphere, ocean, and so on. Sensors or other scientific instruments are deployed in environments for collecting *observations*. Ideally, these observations can be used to predict the changes of environments directly. But in reality, errors are generally common in them. For example, measurements from scientific instruments might not be accurate, fluctuations exist in environments, or the underlying mathematical models may be inaccurate. As a result, observations need to be calibrated before describing environmental states. State-transition algorithms are applied here for calibrating observations using previous states.

Most applications of interest in this domain are modeled in a 2D or 3D coordinate space, and need to handle two independent datasets (represented as logic arrays) for observations and states. These two arrays need to follow a same decomposition pattern and distribute partitioned blocks across a parallel system (e.g., a cluster, a virtual organization in a grid, or a virtual cluster in clouds). Observations typically reflect fluctuations of the environment and exist in a few local regions where significant phenomena occur, resulting in a sparse array. To address such dynamism, adaptive partitioning solutions [3–7] are typically used to balance the distribution of computation of observations, generating blocks of different sizes for the data of states. While many state-transition applications increasingly use a growing volume of scientific data for high-resolution, extended-coverage and timely results, solutions that balance the computation of observations need to balance the data of states as well.

In addition, to make solutions portable across various parallel computing systems, performance profiles of systems need to be considered. Modern HPC clusters are often built with different hardware configurations, indicating different CPU and I/O speeds [8]. Newly emerging virtual clusters are built using various types of virtual machines to meet various computing needs. New issues due to virtualization, such as network jitters [9] and processor sharing [10] on Amazon EC2, affect the performance as well. As profiles reflect CPU and I/O speeds, we should adjust workload partitioning and allocation schemes accordingly to match different system profiles.

1.1 Summary of Contributions

In this chapter, we present an adaptive partitioning and load-balancing scheme, called Apala, for balancing state-transition applications in a computer cluster system. Apala consists of an adaptive decomposition scheme for decomposing arrays into blocks to maximize parallelism and a decentralized scheme for

distributing blocks across processors. Based on the quad-tree [11] structure, blocks are decomposed adaptively and recursively. The distributing scheme distributes the blocks across processors by leveraging the linear representation of the quad-tree structure. Techniques of virtual decomposition and finding-side-neighbor are proposed for organizing data dependencies of updating "halos" (i.e., accessing non-local distributed arrays [12, 13]) between adjacent blocks.

An important feature of Apala is that the decomposition decision is based on both computation and data. Balancing either one independently suggests different decomposition patterns. To consider both jointly, Apala introduces a time metric to unify the workloads of computation and data. More specifically, the workload in terms of time is calculated by adding the time of computing observations (i.e., computation amount/system's CPU speed) and that of loading the data of states (i.e., data amount/system's I/O speed).

1.2 Organization

The rest of this chapter is organized as follows. We discuss related work in Sect. 2. The partitioning problems in parallelizing state-transition applications are described in Sect. 3. Section 4 presents Apala, including unifying the workloads of computation and data with the consideration of system profiles, decomposing unified workloads adaptively, and distributing workloads across processors. Experimental results are given for comparing Apala with other partitioning schemes and showing Apala's portability in Sect. 5. We conclude our work in "Conclusion" section.

2 Related Work

Efficient partitioning and balancing of workloads in parallel systems are both needed to achieve good performance and scalability. The scientific computing community has made significant efforts in partitioning computations using a number of non-overlapping regular blocks while minimizing the maximally loaded block. Adaptive partitioning methods are widely used in the presence of computation skews [3–5, 7, 15].

The GBD [5, 15] partitioning, also called rectilinear partitioning, uses (M−1)*(N−1) lines to decompose a 2D domain into M*N blocks. In case computations distribute non-uniformly in the domain, these blocks are of different sizes but contain the same amount of computations. The GBD partitioning is widely applied [19] due to its approved load-balance effectiveness and easy-to-organize communications of block boundaries. [5] also proposes a semi-GBD partitioning. The semi-GBD first uses M−1 lines to divide a 2D domain into M stripes. Then, it either divides every stripe into N blocks (called M×N-way jagged partitioning), or divides each stripe according to the amount of computations the stripe contains

(called M-way jagged partitioning) that stripes need not to have the same number of blocks. According to [7], the semi-GBD partitioning has a better load-balance effectiveness in some cases. However, synchronization complexity increases. The HB [3] partitioning is an adaptive and recursive partitioning method similar to the quad-tree structure we use. Different from the quad-tree manner, HB uses a split to divide a block into two sub-blocks every time.

These adaptive partitioning methods require a global view of computations to determine the placement of lines or splits. They also imply a significant overhead of finalizing a decomposition pattern for a domain, i.e., the domain is scanned again and again to determine every partitioning decision. In parallel environments, maintaining a global view for intensive computations and data is hardly feasible, and frequent workload scans incur substantial overheads. Apala overcomes these by presenting a distributed partitioning method that every processor partitions a local block independently. There is no need to maintain a global view of workloads in Apala, and the scan is for the local block only.

Although finding an optimal decomposition plan is NP-hard [20], many parallel frameworks use heuristics to integrate these adaptive partitioning methods for handling computation skews at runtime, such as [21, 22] for the adaptive mesh refinement problem. Apala resembles them but uses a decentralized strategy to partition the workload instead of the centralized mechanisms used in these frameworks.

Data is playing a more important role in state-transition applications. Apala explicitly considers data in its partitioning decisions and uses a time metric to unify computation and data according to system profiles. A related work to Apala is Mammoth [13] that processes state-transition applications using a MapReduce system. For the load imbalance issue, Mammoth relies on a runtime management by launching shadow tasks for heavy blocks. SkewReduce [23] is a specially designed system for feature-extracting scientific applications. SkewReduce defines two cost functions to estimate the costs of the partitioning and merging operations. A block will be bi-partitioned if the performance gain of parallelizing the two parts outweighs the partitioning and merging costs. SkewReduce samples the data to estimate the workload and guide the partitioning. Its MapReduce programming model and runtime make it easy-to-use and efficient. In contrast, Apala still addresses the partitioning and load-balance problem in the conventional MPI programming model.

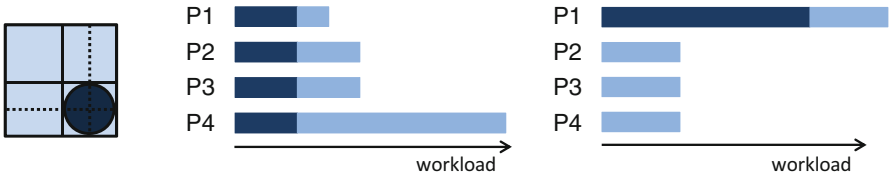

Fig. 1 Example of parallelizing a state-transition application by equally partitioning the computation (*dark blue*) or by equally partitioning the data (*light blue*). For both scenarios, the computation and the data cannot be equally partitioned simultaneously, and consequently the workloads allocated to processors P1, P2, P3, and P4, are not equal (Color figure online)

3 Problem Description

For state-transition applications, the workloads of computation and data are determined by observations and states, respectively. The data of observations is generally of small size and does not bring in data workloads in terms of I/Os. An efficient parallelization scheme needs to partition and balance both workloads. However, the inconsistent distributions of the two workloads complicate the parallelization work. Consider the example in Fig. 1. The state-transition application of the weather data assimilation is modeled in a 2D coordinate space with a bounding domain, and changes of states (e.g., severe regional weather phenomena) are observed in a small region. Balancing the parallelization of this application's computation may result in the partitioning illustrated by the dash lines. We can see that, although the workload of computation is balanced, the workload of data corresponding to block size is not. Likewise, partitioning with the consideration of balancing data (illustrated by lines) suffers from the imbalanced partitioning of computation.

The partitioning problem becomes more complex if porting the parallelization work across different systems. Although the application's computation scale and data volume are constant, the CPU and I/O times spent on computing and loading data vary as systems have different CPU and I/O speeds. The length of the bars indicating the workloads of computation and data in Fig. 1 will change when porting to different systems due to their different profiles. Consequently, the partitioning strategies should be adjusted accordingly.

4 Apala

Apala features three key design merits: (1) it unifies and balances computation and data requirements; (2) it leverages the quad-tree structure to conduct the adaptive and recursive decomposition; (3) it utilizes a decentralized mechanism to distribute workloads across processors for load balance.

4.1 Unifying Workloads

Apala unifies the workloads of computation and data by quantifying the two in terms of time. Specifically, Apala first estimates the time needed for computing the observations as well as for loading the data of states. The combined time is then considered for partitioning. Ideally, every processor spends an equal amount of time on loading and computing its assigned workload.

Two factors affect the time estimation: the speed of loading the states and the speed of computing the observations. In our work, we mainly consider the system's CPU and I/O speeds. Consider a simple example of processing a block on two

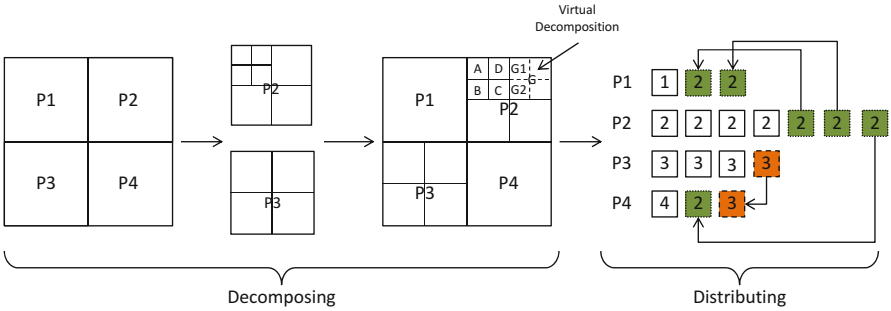

Fig. 2 Apala consists of two major steps: decomposing and partitioning. The decomposing step generates a decomposition plan for the application, and the partitioning step adjusts workload allocations among processors for load balance. The decomposing step involves three detailed substeps. (1) The entire application is uniformly decomposed, and each processor takes one equal-sized block. (2) Each processor independently generates an adaptive decomposition using the quad-tree structure. This decomposition is for the local block only. (3) Some blocks are "virtually" decomposed for finding side neighbors to set up data dependencies across processors

clusters, `cluster1` and `cluster2`. `cluster1` features 2,000 average MOPS (Million Operations Per Second) CPU speed and 100 average MBPS (Megabytes Per Second) I/O speed, while `cluster2` has 1,500 average MOPS CPU speed and 1,000 average MBPS I/O speed. If the block associates with 100 MB data of states and needs 100 million CPU operations to compute observations using state-transition algorithms, its unified workload on `cluster1` is $\frac{100}{100} + \frac{100}{2{,}000} = 1.05\,\text{s}$, while that on `cluster2` is $\frac{100}{1{,}000} + \frac{100}{1{,}500} = 0.17\,\text{s}$.

4.2 Decomposing the Unified Workload

Apala decomposes the unified workload into blocks for exposing parallelism as much as possible. In addition, the decomposition should be well structured so that synchronizations among blocks are organizable. Apala exploits the quad-tree structure in its decomposition scheme. A block with an intensive workload will be decomposed into four equal-sized sub-blocks. Each sub-block is checked to determine if further decomposition is needed. The decomposition continues until every block has a bounded workload.

There are two advantages for the quad-tree structure. First, blocks are regularly decomposed in which every decomposition operation generates four equal-sized sub-blocks. The decomposition shape is critical to simplify synchronization complexity and consequently reduce synchronization overhead. Rectangles are the most preferred shape to decompose 2D workloads for such purpose [7, 14], and quad-decomposing blocks can guarantee this. Second, it is easy to follow the quad-tree manner and decompose blocks adaptively and recursively.

Algorithm 1: Local decomposition

Require: L: single-layer workload (2D) of the state-transition application
N: number of processors
i: processor id
Ensure: P_i: local decomposition at processor i
1: Threshold $T \leftarrow \frac{L}{N}$
2: Decomposition $Q_i \leftarrow \varnothing$, $P_i \leftarrow \varnothing$
3: block $p \leftarrow$ uniform_decompose(L, i, N)
4: $Q_i \leftarrow Q_i \cup \{p\}$
5: **repeat**
6: block $p \leftarrow$choose the first block from Q_i
7: **if** $p.wl > T$ **then**
8: blocks *subs* $\leftarrow$quad_decompose(p)
9: **for all** block p' **in** *subs* **do**
10: **if** $p'.wl > T$ **then**
11: $Q_i \leftarrow Q_i \cup p'$
12: **else**
13: $P_i \leftarrow P_i \cup p'$
14: **end if**
15: **end for**
16: **else**
17: $P_i \leftarrow P_i \cup p$
18: **end if**
19: **until** $Q_i \neq \varnothing$
20: **return** P_i

Although state-transition applications are typically modeled in a 3D coordinate space, the workload decomposing is conducted layer by layer along the vertical direction (z-axis). All layers apply the same 2D decomposition. So we describe the decomposition scheme based on the 2D array of a single layer in the following.

The decomposition is summarized in Algorithm 1. At the beginning, the 2D array is uniformly decomposed into blocks, one for each processor. Every processor independently decomposes its local block in the adaptive and recursive manner. The block's workload is computed, and the block will be decomposed into four equal-sized sub-blocks if it exceeds the threshold. We define the threshold as the average workload across the system, i.e., the number of processors divides the amount of workloads. This recursive decomposing operation continues for every sub-block until each of them meets the threshold. This procedure is analogous to the construction of a quad-tree: the initial local block corresponds to the root; the sub-blocks that contain heavy workloads and are further decomposed correspond to internal nodes; and the final sub-blocks correspond to the leaves.

Synchronizations between adjacent blocks for swapping "halo" updates also benefit from the quad-tree structure. Data dependencies among blocks can be set up by finding side neighbors for the leaves in the quad-tree, and synchronizations occur when adjacent blocks are distributed to different processors. The algorithm introduced in [11] can be used for side-neighbor-finding, but it is restricted to find side neighbors with equal or larger size. As illustrated in Fig. 2, block G's

Algorithm 2: Building block dependencies

Require: P: "virtually decomposed" blocks
r: node (block that finds neighbors)
d: direction where to find neighbors
Ensure: S: set of neighbor nodes
1: set *vdnodes* ← nodes in P decomposed from r
2: **for** node *sub_r* in *vdnodes* **do**
3: node p ← ancestor node of both r and its neighbor
4: *addr* ← address of *sub_r* to p
5: *addr* ← mirror operation
6: node *dest* ← tree traversal from p using *addr*
7: $S \leftarrow S \cup \{dest\}$
8: **end for**
9: **return** S

left-side neighbor will be ambiguous when using this algorithm as there are two such neighbors. We circumvent this restriction by introducing the virtual decomposition technique: large blocks are virtually decomposed as finely as the smallest one. As a result, finding a side neighbor for a large block is split into finding a set of side neighbors for its virtual sub-blocks. The decomposition is virtual because the original block is the minimum unit of distributing workloads. The virtual blocks will not be distributed separately to different processors, nor will there be data dependencies among these virtual blocks.

Building data dependencies for blocks is conducted via the quad-tree traversal. We use node and block interchangeably to ease the understanding of operations related to the quad-tree structure. The concept of *address* is used: except the root, the address of a quad-tree node is defined as its corresponding block's position in its super-block (i.e., *LT* as lefttop, *LB* as leftbottom, *RB* as rightbottom, and *RT* as righttop). Two nodes can concatenate the addresses between them as *path* to refer to each other. The node (or sub-node from the virtual decomposition) locates its side neighbor in three steps: first, finds the path to the common ancestor of the neighbor; second, executes a mirror operation on the path; finally, search down from the ancestor along the mirrored path. The mirror operation replaces every *L*/*R* with *R*/*L* if the direction is *left* or *right*, or every *T*/*B* with *B*/*T* if the direction is *top* or *bottom*, along the path. Algorithm 2 summarizes the procedure of building dependency. An example is given in Fig. 2 that block G is virtually decomposed for finding its two left side neighbors, i.e., C and D.

4.3 Distributing the Unified Workload

Apala's distributing scheme is responsible for mapping blocks to processors with the consideration of load balance. It is based on the linear representation of the quad-tree at every processor and a decentralized scheme for re-mapping blocks from overloaded processors to underloaded ones.

The linear representation is generated using the in-order traversal of quad-tree leaves. A block's four sub-blocks are mapped in the order of "*TL, BL, BR, TR*" (counterclockwise order from topleft). Further decomposition of any sub-block is represented by replacing it with an expanded mapping of its sub-blocks (e.g., decomposing the topleft in "*TL, BL, BR, TR*" will generate the mapping of "[*TL, BL, BR, TR*], *BL, BR, TR*"). This linear representation eases the distributing work of mapping blocks to processors. The first a few blocks with the total workload approximating the threshold will be reserved for the local processor. The rest will be distributed to underloaded processors. Recall that the quad-tree structure is used for building dependencies for blocks. The block distributed to other processors can explore its neighbor blocks quickly, and subsequently processors can set up communication for swapping "halo" updates.

Overloaded processors distribute their extra workloads to underloaded ones in a decentralized manner. Every processor independently checks the available capacity of every underloaded processor, sorts the blocks to be distributed in the ascending order according to their workloads, and maps every such block to the first underloaded processor that is available.

This decentralized mechanism allows an overloaded processor to complete its distributing action quickly. However, it might also lead to a situation that too many overloaded processors push workloads to the same underloaded processor simultaneously, resulting in a new overloaded processor. To overcome this, we introduce a throttle factor γ to control the amount of workload an overloaded processor can push to an underloaded one. Algorithm 3 outlines this decentralized distributing scheme.

Algorithm 3: Decentralized distributing scheme

Require: P_i: the blocks at processor i
L: workload amount
N: number of processors
1: blocks to be distributed $O \leftarrow \varnothing$
2: $P_i^l, P_i^r \leftarrow$ divides P_i that P_i^l are blocks reserved locally and P_i^r are to be distributed.
3: $O \leftarrow O \cup P_i^r$
4: sort(O)
5: $T_a \leftarrow \frac{L}{N}$
6: **for all** underloaded processors j **do**
7: $j.free \leftarrow \gamma * (T_a - j.wl)$
8: **end for**
9: **for** block p **in** O **do**
10: **for all** underloaded processors j **do**
11: **if** $p.wl \leq j.free$ **then**
12: $p.owner \leftarrow j$;
13: $j.wl \leftarrow j.wl + p.wl$
14: $j.free \leftarrow j.free - p.wl$
15: **end if**
16: **end for**
17: **end for**

5 Evaluation

In this section, we present the experimental evaluation. It includes two parts: (1) comparing Apala with other partitioning schemes (i.e., uniform partitioning, Generalized Block Distribution (GBD) [5, 15], and Hierarchical Bipartition (HB) [3]) in terms of the effectiveness of load balance; (2) evaluating Apala's portability of adjusting its partitioning and load-balance scheme according to system profiles; (3) evaluating Apala's overhead of partitioning.

5.1 Setup

Applications and Datasets The state-transition application of weather data assimilation [16] is used. It models states of the environment and observations of the atmosphere in a 3D bounding box. The data assimilation algorithm is performed layer-by-layer along the z-axis and point-by-point within each layer. At every point, the states and the observations are assimilated for new states. These states will not only update the point itself but also the neighbor points in a 4×4 "halo".

We use two datasets (Fig. 3) in the evaluation. The first contains 75 GB data of states and 25 MB data of observations, while the second contains 19 GB data of

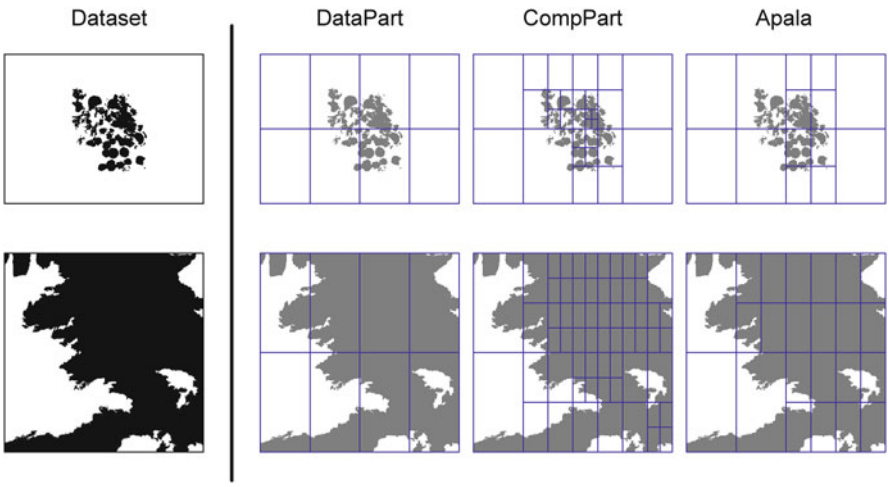

Fig. 3 The datasets we use in the experimental evaluation. The first dataset is from the Center for Analysis and Prediction of Storms at the University of Oklahoma, which described the observations covering the Oklahama state on May 20, 2010. The second one was captured from weather.com, which described the observations covering Gainesville, Florida on January 1, 2012. Both are modeled in the 3D coordinate space but showed with the 2D view. The first is bounded in a $1{,}323 \times 963 \times 10$ box, while the second is bounded in a $400 \times 400 \times 20$ box. The decomposition patterns with different schemes for both datasets are illustrated as well

states and 3 MB data of observations. Both states and observations are stored in the key-value format in plain text files, i.e., *"x,y,z,var1,var2,...,varN"* for states and *"x,y,z,ob"* for observations, where *"x,y,z"* represents the coordinate, *"varI"* represents the environmental variable, and *"ob"* represents the observation.

Computer Clusters We use four clusters for the evaluation: two HPC clusters of Alamo and Sierra at FutureGrid, and two virtual clusters, EC2.Small and EC2.Large, consisting of Amazon EC2 small and large instances, respectively.

Each node at Alamo contains two 2.66 GHz Intel Xeon X5550 processors (four cores per processor) and 12 GB memory and is interconnected via InfiniBand. Alamo uses a NFS-based parallel file system for the shared data storage. Each node at Sierra contains two 2.5 GHz Intel Xeon L5420 processors (four cores per processor) and 32 GB memory and is also interconnected via InfiniBand. Sierra uses a ZFS file system for the shared data storage.

Each small instance in EC2.Small has 1 EC2 compute unit, 1.7 GB memory, and moderate I/O speed, while each large instance in EC2.Large has four EC2 compute units, 7.5 GB memory, and high I/O speed. Due to the virtualization overhead, the two virtual clusters perform much more moderately (particularly the EC2.Small cluster) than the physical HPC clusters. For their data storage, we create an Amazon EBS (Elastic Block Service) volume, attach it to a dedicated instance, and mount it to the cluster using the NFS file system.

Benchmark Tools We use the NAS Parallel Benchmark (NPB) [17] and the IOR HPC benchmark [18] to measure the CPU and I/O speeds for every cluster. The NPB consists of several benchmarks that simulate the computational fluid dynamics applications, and these applications cover various computing patterns. The CPU speeds of these benchmarks are averaged, and the average value is used to represent the CPU speed of the cluster. To measure the I/O speed, the IOR HPC benchmark is used to reproduce the I/O patterns of our data assimilation application, i.e., concurrently and randomly accessing a continuous block of a single file. Table 1 lists the profiles of the CPU and I/O speeds of the four clusters.

Table 1 Profiles of the clusters

	Alamo	Sierra	EC2.Large	EC2.Small
I/O (Read, MB/s)	1233	1099	938	478
I/O (Write, MB/s)	391	70	15	16
CPU (MOPS)	824	351	128	33

Implementation Apala is implemented using C++ with the standard MPI-2 library. It reads the benchmark results describing system profiles (i.e., the CPU speed and the I/O speed) from a configure file. The paths to the data of states and observations are contained in this file as well. Apala generates a quad-tree-based decomposition plan using a *partition* method, and allocates the workload to processors using a *distribute* method. For each processor, its workload allocation is organized as a list of rectangular blocks. These blocks are represented using indexes of arrays (i.e., the

bottom-left coordinate and the top-right coordinate). State-transition algorithms are programmed from the perspective of an individual point. For updates in "halos" that are out of original blocks, Apala has a *synchronize* method for users to manage synchronizations. Indexes of points and data dependencies are built when generating decomposition plans, and they are used to drive the *synchronize* method. An *aggregate* method is opened for users to define how to finalize the result of a point covered by multiple "halos".

We set the throttle factor γ to 0.5, implying at most half of the processors can push their workloads to an underloaded one. Note that every result in the following figures is the average value of five runs. The error bars are small, indicating performance fluctuations are marginal, even on virtual clusters.

5.2 Decomposition Patterns

Two static partitioning strategies, "DataPart" and "CompPart", are used for the comparison purpose. The "DataPart" partitions the workload according to the data of states only, while the "CompPart" accords to that of observations only. Figure 3 illustrates the decomposition patterns of using "DataPart", "CompPart", and Apala to decompose both datasets for eight processors. We can see that, "DataPart" generates a uniform decomposition as the data of states indicating I/Os distributes evenly, "CompPart" generates a much finer decomposition for the regions with dense observations. In comparison, the decomposition from Apala is in between due to its comprehensive consideration of computation and data.

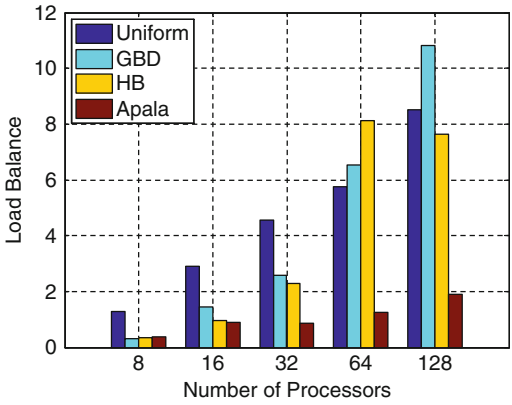

Fig. 4 Comparison of the load balance effectiveness among Uniform, GBD, HB, and Apala, with the first dataset

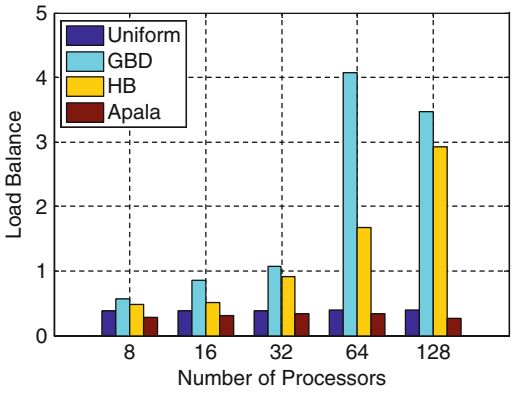

Fig. 5 Comparison of the load balance effectiveness among Uniform, GBD, HB, and Apala, with the second dataset

5.3 *Effectiveness of Load Balance*

The effectiveness of load balance is measured according to $(Max - Avg)/Avg$, where Max is the execution time of the longest processor, and Avg represents the average execution time of all the processors. Smaller values indicate better load balance.

The results comparing Apala with other partitioning schemes are presented in Figs. 4 and 5 for the two datasets, respectively. We can see that, Apala outperforms the partitioning schemes of uniform, GBD, and HB for both datasets by at most ten times. Uniform, GBD, and HB partition the workload according to the computation only. Ignoring the workload of loading the large volume of states results in load imbalance. The first dataset shows more significant imbalance. That is because the observations in the first dataset mainly concentrate on hot spots, and the block sizes are more diverse after decomposition. However, Apala performs stably because it accounts for both computation and data.

5.4 *Portability*

When porting to a new computer cluster, the CPU and I/O speeds of the system change, and Apala will adjust its workload estimation and the subsequent partitioning and load-balance scheme. To evaluate Apala's portability, we deploy Apala to the four clusters and compare its performance to static (not portable) partitioning strategies. Due to space limit, only results about the first dataset is presented in this paper.

According to the CPU and I/O speeds listed in Table 1, Alamo and Sierra show excellent CPU and I/O speeds. EC2.Large and EC2.Small are expected to present poor I/O speeds due to using the shared EBS volume. All the I/Os for the data of

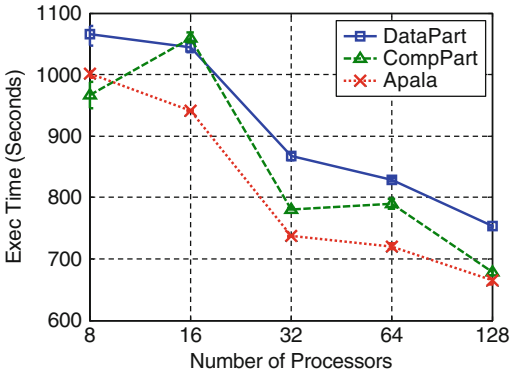

Fig. 6 Comparison of the execution time of "DataPart", "CompPart", and Apala on the Alamo cluster, with the first dataset

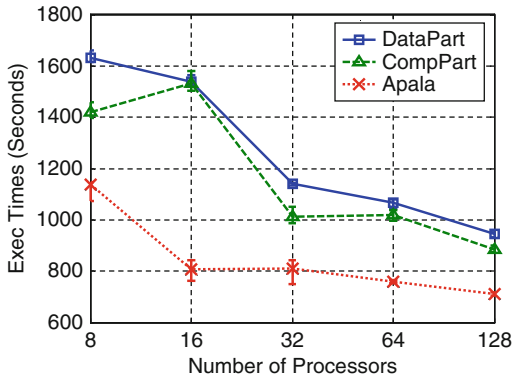

Fig. 7 Comparison of the execution time of "DataPart", "CompPart", and Apala on the Sierra cluster, with the first dataset

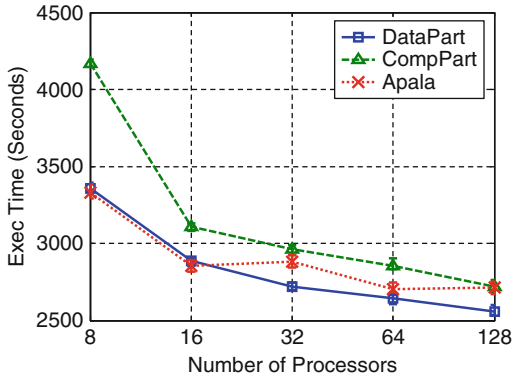

Fig. 8 Comparison of the execution time of "DataPart", "CompPart", and Apala on the EC2.Large cluster, with the first dataset

states will be directed to the shared EBS storage. EC2.Small also shows a poor CPU speed due to the shared use of physical processors [10].

Intuitively, for the system that has a fast I/O speed, the time spent on loading the data of states is relatively short in the workload estimation, and partitioning using "CompPart" that depends on observations is preferred (and vice versa for the system with a fast CPU speed).

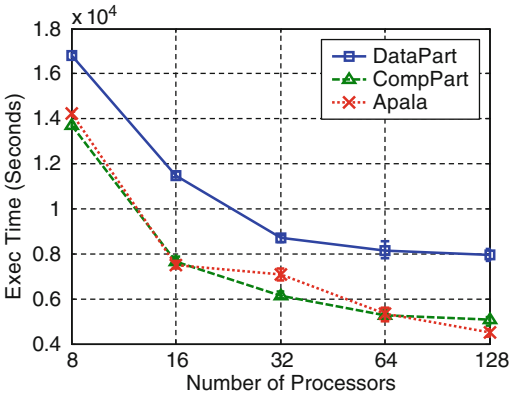

Fig. 9 Comparison of the execution time of "DataPart", "CompPart", and Apala on the EC2.Small cluster, with the first dataset

We use "DataPart", "CompPart" and Apala to partition the first dataset for the evaluation of portability. Results are presented in Figs. 6, 7, 8, and 9. As "DataPart" and "CompPart" are static, their partitioning and load-balance schemes are constant across systems. We can observe that, partitioning according to observations outperforms that based on states for Alamo, Sierra and EC2.Small, while partitioning according to states performs better on EC2.Large. This matches our expectation that partitioning according to the weak factor yields a better load balance. In contrast, Apala always shows excellent performance on all systems. It outperforms both static schemes significantly on the HPC clusters and almost performs best on the virtual clusters (ties with "DataPart" on EC2.Large and "CompPart" on EC2.Small). The reason is straightforward, jointly considering computation and data always benefits the partitioning and load-balance scheme.

5.5 Overhead of Partitioning

We measure the partitioning overhead for Apala with the two datasets and show the results in Figs. 10 and 11. Since the results cover all the four clusters, they are normalized to be plotted in the same figure. For each cluster, the execution time with 8 processors is set to 1, and the values with 16, 32, 64, and 128 processors are adjusted proportionally. The "CPU+I/O" parts for Apala include scientific computations and data loads. The "Synchronization" parts represent the time Apala spends on MPI_Isend, MPI_Irecv, and MPI_Barrier. The "Partitioning Overhead" means processors are decomposing their local blocks and distributing the workload to (or receiving from) others for load balance. At each tick, the four bars from the left to the right represent Alamo, Sierra, EC2.Large, and EC2.Small, respectively.

We can see that the partitioning overhead in Apala is minimal for both datasets on all the clusters. This is reasonable as each processor only estimates its local workload, and such estimation is merely based on the data size, the computation

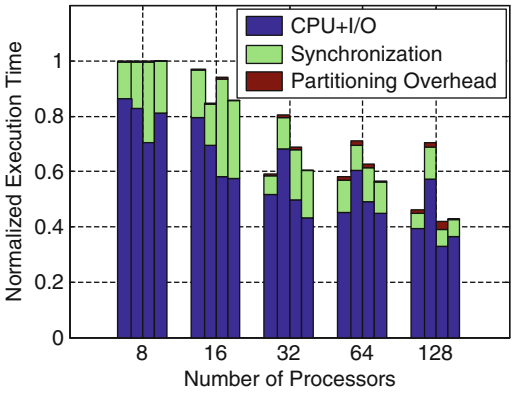

Fig. 10 Comparison of the CPU+I/O, the synchronization, and the partitioning overhead in terms of time on the four clusters, the first dataset

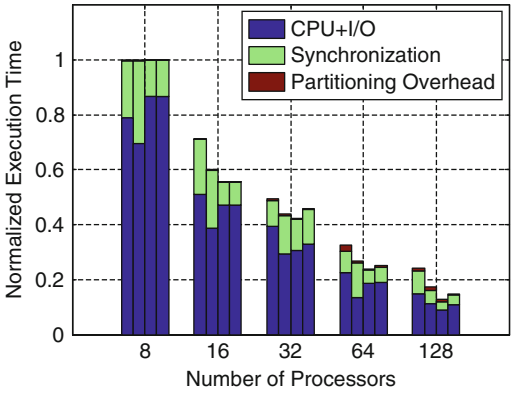

Fig. 11 Comparison of the CPU+I/O, the synchronization, and the partitioning overhead in terms of time on the four clusters, the second dataset

scale, and system profiles. The partitioning overhead increases with respect to the number of processors, as distributing the workloads involves more processors, and the communication complexity increases accordingly.

Conclusion

In this chapter, we presented Apala, an adaptive workload partitioning and allocation scheme for parallelizing data intensive state-transition applications in various parallel systems. State-transition applications are representative data-intensive scientific applications. They generally tackle grand-challenge problems (e.g., weather forecasting, ocean prediction) and involve extremely complex algorithms. Apala considers both computation and data in its workload partitioning and allocation scheme. It introduces a time metric for unifying the workloads of computation and data and profiles systems for accurate workload estimation. The quad-tree structure is used to represent

(continued)

the procedure of breaking up arrays into blocks, and techniques of virtual decomposition and finding-side-neighbors are introduced to organize data dependency. A decentralized distributing strategy is applied for distributing blocks across processors. Experimental results from the real-world data show that, Apala outperforms other partitioning schemes in terms of the effectiveness of load balance by at most ten times. Moreover, it shows excellent portability on diverse systems from HPC clusters to virtual clusters in clouds and incurs marginal overhead of partitioning.

References

1. M. Fisher, J. Nocedal, Y. Trémolet, and S. Wright, "Data assimilation in weather forecasting: a case study in pde-constrained optimization," *Optimization and Engineering*, vol. 10, no. 3, pp. 409–426, 2009.
2. A. Robinson and P. Lermusiaux, "Overview of data assimilation," *Harvard reports in physical/interdisciplinary ocean science*, vol. 62, 2000.
3. M. Berger and S. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors," *ToC*, vol. 100, no. 5, pp. 570–580, 1987.
4. D. Nicol, "Rectilinear partitioning of irregular data parallel computations," DTIC Document, Tech. Rep., 1991.
5. F. Manne and T. Sørevik, "Partitioning an array onto a mesh of processors," *Applied Parallel Computing Industrial Computation and Optimization*, pp. 467–477, 1996.
6. O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Matrix multiplication on heterogeneous platforms," *TPDS*, vol. 12, no. 10, pp. 1033–1051, 2001.
7. E. Saule, E. O. Bas, and U. V. Catalyurek, "Partitioning spatially located computations using rectangles," in *IPDPS*. IEEE, 2011.
8. N. Wright, S. Smallen, C. Olschanowsky, J. Hayes, and A. Snavely, "Measuring and understanding variation in benchmark performance," in *DoD High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC), 2009*. IEEE, 2009, pp. 438–443.
9. T. Zou, G. Wang, M. Salles, D. Bindel, A. Demers, J. Gehrke, and W. White, "Making time-stepped applications tick in the cloud," in *SoCC*. ACM, 2011, p. 20.
10. G. Wang and T. Ng, "The impact of virtualization on network performance of amazon ec2 data center," in *INFOCOM*. IEEE, 2010, pp. 1–9.
11. H. Samet, "The quadtree and related hierarchical data structures," *ACM Computing Surveys (CSUR)*, vol. 16, no. 2, pp. 187–260, 1984.
12. B. H. A. Hoekstra and R. Williams, "High-performance computing and networking."
13. X. Yang, Z. Yu, M. Li, and X. Li, "Mammoth: autonomic data processing framework for scientific state-transition applications," in *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*. ACM, 2013, p. 13.
14. J.-R. Sack and J. Urrutia, *Handbook of computational geometry*. North Holland, 1999.
15. B. Aspvall, M. M Halldórsson, and F. Manne, "Approximations for the general block distribution of a matrix," *Theoretical computer science*, vol. 262, no. 1, pp. 145–160, 2001.
16. M. Xue, D. Wang, J. Gao, K. Brewster, and K. Droegemeier, "The Advanced Regional Prediction System (ARPS), storm-scale numerical weather prediction and data assimilation," *Meteorology and Atmospheric Physics*, vol. 82, no. 1, pp. 139–170, 2003.

17. R. Van der Wijngaart and P. Wong, "Nas parallel benchmarks version 2.4," NAS technical report, NAS-02-007, Tech. Rep., 2002.
18. "IOR HPC Benchmark," http://sourceforge.net/projects/ior-sio/.
19. H. P. F. Form, "High performance fortran language specification," 1993.
20. M. Grigni and F. Manne, "On the complexity of the generalized block distribution," *Parallel Algorithms for Irregularly Structured Problems*, pp. 319–326, 1996.
21. M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *Journal of computational Physics*, vol. 53, no. 3, pp. 484–512, 1984.
22. X. Li and M. Parashar, "Hybrid runtime management of space-time heterogeneity for parallel structured adaptive applications," *TPDS*, pp. 1202–1214, 2007.
23. Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skew-resistant parallel processing of feature-extracting scientific user-defined functions," in *SoCC*. ACM, 2010, pp. 75–86.