Inc-part: Incremental Partitioning for Load Balancing in Large-Scale Behavioral Simulations

Yu Zhang, Xiaofei Liao, *Member, IEEE,* Hai Jin, *Senior Member, IEEE,* Guang Tan, *Member, IEEE,* and Geyong Min, *Member, IEEE*

Abstract—Large-scale behavioral simulations are widely used to study real-world multi-agent systems. Such programs normally run in discrete time-steps or ticks, with simulated space decomposed into domains that are distributed over a set of workers to achieve parallelism. A distinguishing feature of behavioral simulations is their frequent and high-volume *group migration*, the phenomenon in which simulated objects traverse domains in groups at massive scale in each tick. This results in continual and significant load imbalance among domains. To tackle this problem, traditional load balancing approaches either require excessive load re-profiling and redistribution, which lead to high computation/communication costs, or perform poorly because their statically partitioned data domains cannot reflect load changes brought by group migration. In this paper, we propose an effective and low-cost load balancing scheme, named *Inc-part*, based on a key observation: that an object is unlikely to move a long distance (across many domains) within a single tick. This localized mobility property allows one to efficiently estimate a dynamic domain's load incrementally, based on merely the load changes occurring in its neighborhood. The domains experiencing significant load changes are then partitioned or merged, and redistributed to redress load imbalance among the workers. Experiments on a 64-node (1024-core) platform show that *Inc-part* can attain excellent load balance with dramatically lowered costs compared with state-of-the-art solutions.

Index Terms—Behavioral simulation, load balancing, incremental partitioning, multi-agent system, group migration

1 Introduction

Behavioral simulations are widely used to model real-world systems, in which a large number of autonomous agents interact in complex ways. They are instrumental in characterizing physical, ecological, and societal systems, such as fish schools [1], [2], [3], [4], transportation simulation [5], [6], [7], honeybee groups [1], [8], [9], [10], and outer space objects [11], [12], [13], [14]. For example, transportation simulations are used to predict road condition and help with transportation engineering. This proves to be very helpful in mitigating traffic congestion, which costed \$12.1 billions and produced 56 billion pounds of carbon dioxide (CO_2) pollution [15] in 2011 alone.

In these simulations, the simulated objects move in a homogeneous space, and the time is discretized into *ticks*. In each tick, simulated objects concurrently gather data about the surrounding environment, reason on the data, and update their states for the next tick [16]. In order to achieve parallelism, the simulated space is divided into domains, which are then distributed over a set of workers [16], [17], [18]. An inherent property of behavioral simulations is the phenomenon of *group migration*, in which objects move across the domains in groups, imitating the self-organization behavior of autonomous agents in the real world [19], [20], [21]. Group migration occurs frequently and at large scale, causing continual and significant load imbalance among the workers. For example, some workers may well have loads many times higher than average, yielding so-called *stragglers*, which greatly slow down program execution.

Currently, there are two main approaches to tackling the load imbalance problem. The first approach partitions data into equally loaded domains (tasks) by profiling load costs through a user-defined cost function [17], and then assigns the domains to workers. The second approach uses persistence-based load balancing (PLB) or retentive work stealing (RWS) [22] to partition data into fixed domains, and then redistribute the corresponding tasks or migrate stragglers for load balancing. Applied to behavioral simulations, the first approach needs to re-profile load cost frequently to account for the changes of load distribution brought by group migration. Accordingly, the

• *Yu Zhang, Xiaofei Liao and Hai Jin are with Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China.*
E-mail: zhang_yu9068@163.com; {hjin, xfliao}@hust.edu.cn

• *Guang Tan is with Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, 518055, China.*
E-mail: guang.tan@siat.ac.cn

• *G. Min is with the Department of Computing, School of Computing, Informatics and Media, University of Bradford, Bradford BD7 1DP, U.K.*
E-mail: g.min@brad.ac.uk

entire simulation space needs to be re-decomposed and a large number of objects be migrated, incurring prohibitively high computation and communication costs. The second approach, on the other hand, does not try to balance load among the tasks, but instead focuses on balanced placement of tasks on the workers. Such an approach has to rely on historical information for task load estimation, which easily becomes outdated and even misleading under group migration.

To address these challenges, we propose a novel and efficient method, named *Inc-part* (Incremental Partition), to mitigate load imbalance for behavioral simulation, based on a key observation: that the mobility of objects within a single tick is limited. In other words, an object is unlikely to move across many domains within a single tick. This allows one to estimate a domain's load cost based on merely the load changes of its neighboring domains. Moreover, the load cost recalculation is needed for a domain only when its neighbor domains experience significant joins or leaves of objects. With up-to-date load cost information, those highly overloaded or under-loaded domains are either partitioned or merged, and then redistributed to restore load balance among workers.

We have implemented Inc-part in the Piccolo [23] programming model to support behavioral simulation. Experiments on a 64-machine (1024-core) cluster show that Inc-part attains good load balance with much lower costs compared with state-of-the-art schemes. For example, Inc-part produces load imbalance degree below one fifth of what is achieved by PLB and RWS [22]. This leads to speedup improvements of $4.08\times$ and $2.95\times$ compared with PBL and RWS, respectively.

The remainder of this paper is organized as follows: Section 2 presents the motivation of this work. Section 3 outlines our approach, with implementation details given in Section 4. Experimental results are shown in Section 5, followed by a description of related work in Section 6. Finally, we conclude this paper in Section 7.

2 Background and Motivation

In this section we discuss the performance characteristics of behavioral simulation and explain why traditional approaches are inadequate for load balancing. Four typical examples of behavioral simulation are introduced in Appendix A.

2.1 Group migration and its impact

Behavioral simulations often run in discrete time-steps, or *ticks*. To achieve parallelism, the simulated space is partitioned into domains, which are then mapped to tasks and distributed over a set of workers. An important feature of the simulations is the behavior of *group migration*, by which objects move through domains, following the mobility patterns of

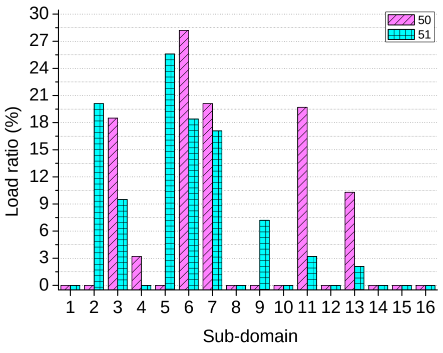


Figure 1. Frequent and high-volume fish migrations among 16 sub-domains across two ticks in the fish school simulation.

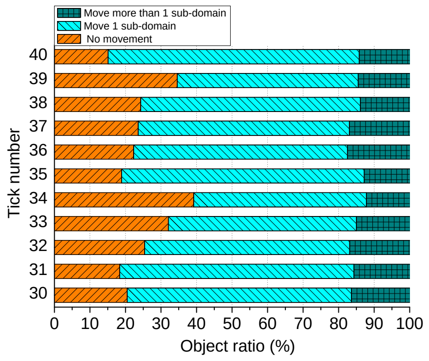


Figure 2. The locality property of group migration in the fish school simulation.

groups of agents in real-world systems. Group migration exhibits three characteristics: high frequency, high volume, and locality.

We run a fish school simulation on a 64-machine (1024-core) cluster to demonstrate the properties of group migration. (More details of the experimental methodology are given in Section 5.) A 2D simulated space comprising 10 million fish is divided into 1024 equal-sized square domains, which are distributed over the 1024 cores. Furthermore, each domain is divided into 16 sub-domains. Note that the sub-domain is only used in this section to explain the phenomenon of group migration. A sub-domain's load ratio over the total load of its owning domain is measured by the percentage of fish it hosts. Figure 1 shows the 16 sub-domains' load ratios in two consecutive ticks, 50 and 51. It can be seen that the load ratios of the various sub-domains vary constantly and widely. For example, 7 out of the 16 sub-domains, namely sub-domains 2, 3, 4, 5, 9, 11, 13, experience significant load changes of at least 50% in tick 51 compared with tick 50. Furthermore, sudden rises and falls of loads appear to be very common. For example, the sub-domains 2 and 5 both see a surge of load ratio from zero to above 20% across tick 50.

The locality property is reflected in Figure 2, where a breakdown of the objects' moved distances in each tick is shown. We can see that in all cases, less than 20% of the objects move more than one sub-domain in a tick, indicating that the objects' movement is largely localized within a short period of time. This property will turn out to be very helpful in load balancing.

2.2 Inefficiency of existing solutions

In Figure 1, at tick 50 there are four sub-domains with load ratios above 18%, while at the same time ten sub-domains have almost zero loads, suggesting a high variation of load ratios. Also, the highest loaded sub-domains, called stragglers, change over time. In tick 50 for example, the straggler sub-domain is 6, while in tick 51, the straggler becomes sub-domain 5. This phenomenon brings great challenges to the existing load balancing schemes. For example, the user-defined cost function based approach [17] attempts to balance load among tasks (sub-domains in our case). Due to the changes of sub-domain loads, this approach will have to profile load cost for each sub-domain at each tick. Furthermore, it needs to decompose the entire simulated space and migrate a large number of objects in each tick in order to create equally loaded sub-domains. Clearly, this will result in a very high cost in both computation and communication.

The persistence-based load balancing [22] and retentive work stealing [22] methods represent another approach to reducing load imbalance. (See details in Appendix B.) They decompose the simulated space into equally loaded domains only in the beginning, and then re-schedule tasks for load re-balancing based on the profiling of the tasks processed by each worker and the duration of each task. Unfortunately, the task rescheduling in a tick is based on an outdated load profile obtained in the previous ticks. This leads to misguided rescheduling of tasks, which in turn generates continual load imbalance in the system.

In summary, the existing solutions cannot cope with group migration efficiently, thus we need to develop new techniques to tackle the load imbalance problem.

3 The Inc-part Approach

In this section, we first present an incremental load cost updating method, and then show how to re-balance load in behavioral simulations using this method.

3.1 Main idea

The idea of Inc-part leverages the locality property of group migration of objects. Since most objects (>80% in Figure 2) stay in the same domain or move no more than one domain in a tick, it is possible to redress the imbalance accumulated in a tick in a more efficient way. Specifically, we *incrementally* calculate

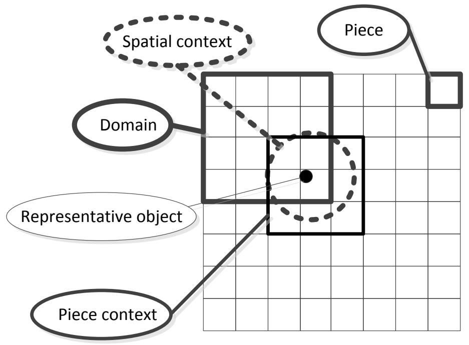


Figure 3. Illustration of main concepts in the cost load model.

a domain's load cost from the load changes of its neighboring domains. This allows us to obtain fairly accurate estimate of load cost for a domain at a low cost. After the load cost update, Inc-part identifies a set of domains that experience most significant changes of load cost, partition or merge them, and redistribute them over the workers for load balance. The selective processing of domains keeps the overall communication cost low.

3.2 Load cost update rule

Suppose the simulated space is divided into a set of domains. The load cost of a domain $\mathcal{D}$ is the sum of the load costs of all the objects in it. That is:

$$\mathsf{cost}(\mathcal{D}) = \sum_{o \in \mathcal{D}} \mathsf{cost}(o). \tag{1}$$

In behavioral simulation, objects in close vicinity have similar load costs. We can then obtain a fast approximation of $\mathsf{cost}(\mathcal{D})$ in the following way. We first divide $\mathcal{D}$ into n fixed, equal-sized rectangular or cubic *pieces*, denoted by P, and assume all objects in P are equally loaded. We pick a random object $r(P)$ to represent the piece P, then the load cost of P is $\mathsf{cost}(P) = \mathsf{cost}(r(P)) \times |P|$, where $|P|$ is the number of objects in P. It follows that

$$\mathsf{cost}(\mathcal{D}) = \sum_{P \in \mathcal{D}} \mathsf{cost}(r(P)) \times |P|. \tag{2}$$

For the purpose of clarity, the main concepts are illustrated in Figure 3, and a list of notations is given in Table 1.

Now let us look at the load cost of a particular object o. In behavioral simulation, an object can only interact with its neighbor objects in a local scope, called its *spatial context*. In the fish school simulation, for example, a fish object can only see or move within a limited scope during a tick. Such a scope is then the fish object's context, whose specific size depends on simulation parameters. Moreover, the processing time of each interaction between two simulated objects is almost the same for behavioral simulation. As such, the load cost of an object can be evaluated with the

Table 1
List of notations.

Notations	Descriptions
$\mathsf{cost}(\mathcal{D})$	Load cost of domain $\mathcal{D}$
$\mathsf{cost}(P)$	Load cost of piece P
$\mathsf{cost}(o)$	Load cost of object o
$r(P)$	An arbitrary object in P that represents P
$\|P\|$	The number of objects in piece P
$\mathsf{context}(o)$	The set of pieces that intersect with object o's spatial context

number of objects in its spatial context. We use an approximate measure to represent such a context. Define the *piece context* of object o, $\mathsf{context}(o)$, as the set of pieces that intersect with the spatial context of o, then we have

$$\mathsf{cost}(r(P)) = \sum_{Q \in \mathsf{context}(r(P))} |Q|. \tag{3}$$

Note that P belongs to $\mathsf{context}(r(P))$. Since the spatial context of a piece P is fixed, $\mathsf{context}(r(P))$ is also fixed and can be calculated when domains are partitioned.

The update rule of $\mathsf{cost}(\mathcal{D})$ is as follows:

$$\begin{cases} \mathsf{cost}(\mathcal{D}) &= \mathsf{cost}(\mathcal{D}) + \sum_{P \in \mathcal{D}} \Delta\mathsf{cost}(P) \\ \Delta\mathsf{cost}(P) &= \Delta_1 + \Delta_2 + \Delta_3 \\ \Delta_1 &= \Delta|P| \times \mathsf{cost}(r(P)) \\ \Delta_2 &= |P| \times \Delta\mathsf{cost}(r(P)) \\ \Delta_3 &= \Delta|P| \times \Delta\mathsf{cost}(r(P)) \\ \Delta\mathsf{cost}(r(P)) &= \sum_{Q \in \mathsf{context}(r(P))} \Delta|Q| \end{cases} \tag{4}$$

From Equation 4, we can see that a domain only needs to recalculate load cost for relevant pieces that experience object joins/leaves.

3.3 Domain management

In this section we present a high-level description of domain management. The details will be given in Section 4.

Before the simulation begins, the simulated space is divided into $N_x \times N_y \times N_z$ equal-sized pieces, where N_x, N_y, N_z are the number of intervals along the x, y, z axes and are user-specified parameters. Note that $N_z = 0$ for a 2D space. The piece division brings several benefits. First, the load cost of a piece can be well approximated and easily calculated, as discussed in Section 3.2. Second, the division facilitates partitioning, merging, and redistribution of domains. Third, it becomes easy to determine the data structure of a moving object and to update the load costs of its influenced pieces and domains. Figure 4 shows an example of how to determine the piece owning a particular object.

At the end of a tick, some domains may have a much higher or lower load than average. Upon detecting this, Inc-part attempts to re-balance the loads among domains before the next tick begins. More specifically, when a domain is found to have load cost more than a threshold T_{max}, it will be partitioned into multiple spatially connected sub-domains with the same *baseline* load cost L (except the last sub-domain). On the other hand, when a domain has a load cost smaller than T_{min}, Inc-part seeks to merge it with its neighbor domains. The merging process is repeated, and stops when a new neighbor domain is going to make the merger exceed L in load cost, or when no suitable neighbor domains are available for merging. Note that the partitioning and merging processes all use pieces as basic operational units. In this paper, Let $T_{max} = \alpha \cdot L$ and $T_{min} = \beta \cdot L$, where α and β are used-defined parameters (e.g., $\alpha = 2, \beta = 0.5$).

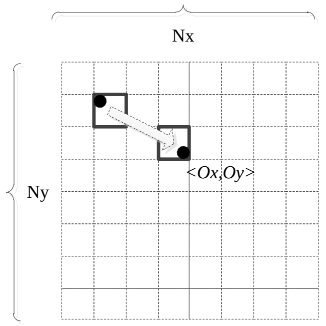


Figure 4. An object moving from a piece to another across a tick. The new location can be obtained from its coordinates $\langle O_x, O_y \rangle$ in the simulated space, given the side length of a piece.

After domain partitioning and merging, all domains will end up having similar load costs. Inc-part then redistributes the domains over workers to maintain load balance for the next tick. The redistribution process takes into account domains' locations and optimizes for small communication and synchronization costs.

4 IMPLEMENTATION

Inc-part is implemented in the Piccolo [23] programming model. Piccolo assumes a master and multiple workers. The master coordinates the workers and monitors their status, and each worker loads a subset of data elements into its memory for processing.

4.1 Data structure

All data elements on a worker are grouped into units of pieces and maintained in a local in-memory key-value state table. Each state table entry corresponds to a piece P and is indexed by P's key. Specifically, it contains four fields: (1) P's key value; (2) the key values of pieces in P's piece context, $\mathsf{context}$(r(P)); (3) the number of objects in P, $|P|$; and (4) a list of objects in P. Note that the second and third fields record the information that is used to calculate load

Algorithm 1 Domain partitioning algorithm

```
Require: Domain $D_1$, Threshold $T_{max}$
Ensure: $D_{set}$ //a set of Domains
 1: while $D_1$.load $>$ $T_{max}$ do
 2:     //Get a piece with load under L from $D_1$'s border
 3:     $P_1 \leftarrow D_1$.GetBorderPiece()
 4:     $D_2$.insert($P_1$)
 5:     //Get a set of pieces connected to $P_1$ from $D_1$
 6:     $P_{set} \leftarrow P_1$.GetConnPieces($D_1$)
 7:     while $P_{set}$ is not empty do
 8:         $P_2 \leftarrow P_{set}$.get()
 9:         if $D_2$.load+$P_2$.load $\leq$ L then
10:             $D_1$.remove($P_2$) //also update $D_1$.load
11:             $D_2$.insert($P_2$) //also update $D_2$.load
12:             $P_{set} \leftarrow P_{set} \bigcup P_2$.GetConnPieces($D_1$)
13:         end if
14:         $P_{set}$.remove($P_2$)
15:     end while
16:     $D_{set}$.insert($D_2$)
17: end while
18: return $D_{set}$
```

cost. Inc-part also uses a table to maintain the information of domains. Each entry in such a table corresponds to a domain, indexed by the domain's key, and contains four fields: (1) the domain's key value; (2) the domain's load cost value; (3) the information of neighboring domains, including their key values and owning workers, used for domain partitioning, merging and redistribution; and (4) the key values of the domain's pieces.

4.2 Domain management details

After an object o has been processed in a tick, Inc-part obtains the piece corresponding to o's new location in the next tick, and inserts o into that piece. If a piece P experiences a significant change of $|P|$ that is greater than a pre-defined threshold T_0, Inc-part recalculates P's load cost, and then updates the load cost of pieces in context(r(P)). Note that in order to reduce the cost of object inserting operations, Inc-part delays them until the end of a tick, and processes them all together. After these updates, some domains may have much higher loads than others. This will be dealt with by domain partitioning and merging below.

4.2.1 Domains partitioning and merging

Inc-part tries to achieve rough load balance among domains by partitioning or merging domains with abnormal load costs before the next tick begins. During this process, the load costs of the influenced domains, including those partitioned/merged ones and their neighbors, will be updated according to the changes of their piece sets. The purpose of this balancing process is to eliminate huge gaps between domains's load costs, which can make subsequent worker-level load balancing difficult. On the other hand, the desired inter-domain balance need not be perfect for the sake of lower costs, as moderate imbalance can be corrected by later domain redistribution.

When a domain's load cost is greater than a predefined threshold T_{max}, it will be partitioned into several spatially connected domains with equal load cost L (except the last domain), as shown in Algorithm 1. The threshold T_{max} restricts the partitioning operations to highly overloaded domains (often a small fraction of all domains), and does not affect others. In addition, pieces in the same domain are ensured to be spatially connected with each other to retain data locality for object processing (Lines 2 to 15 in Algorithm 1).

To partition a target domain D_1, Inc-part tries to extract a sub-domain D_2 with connected pieces from D_1 as follows. It first finds a suitable piece from D_1's boundary, and then repeatedly extracts new pieces connected to the previously extracted ones from D_1, in order to form a new domain D_2, until there are no more suitable pieces. This process is illustrated in Algorithm 1 (Lines 7 to 15). Note that the set P_{set} is a FIFO list. The new neighboring pieces are always inserted to the end of P_{set}. Then through the function get(), the piece connected to the foremost selected piece is first processed. In this way, the pieces in the same domain can be connected with each other more closely, helping to reduce the communication cost.

When a domain's load cost is below a lower bound T_{min}, Inc-part tries to merge it with its neighboring domains, until the merger's load cost is about to be greater than L. Its details are provided in Appendix C.1.

4.2.2 Domains redistribution

Given roughly load balanced domains, Inc-part needs to redistribute the domains among the workers to maintain worker-level load balance before the tick begins. The first consideration for the redistribution is to preserve data locality. To this end, Inc-part always seeks chance to diffuse an excess domain to a worker that hosts domains connected to that excess domain. In this way, much communication and synchronization cost of object processing will be saved in the next tick. If such an arrangement is not possible, Inc-part then diffuses an excess domain to the least loaded worker. The details of domains redistribution algorithm are given in Appendix C.2.

5 PERFORMANCE EVALUATION

In this section, we present experimental evaluation. The impact of system parameters are also evaluated and the results are given in Appendix D.

Platform and benchmarks. The hardware platform used in our experiments is a cluster with 1024 cores residing on 64 nodes, which are interconnected by a 2-Gigabit Ethernet. Each node is a 2-way octuple-core with Intel(R) Xeon(R) CPU E5-2670 at 2.60 GHz CPUs and 64 GB memory, running a Linux operation system with kernel version 2.6.32. A maximum of 16 workers are spawned for each node to run the simulations. Data communication is performed using openmpi version 1.6.3. The program is compiled with cmake version 2.6.4, gcc version 4.7.2, python version 2.7.6 and protobuf version 2.4.1.

To evaluate the performance, four simulation benchmarks are used (detailed in Appendix A), namely the Fish School simulation (FS) [1], the Honeybee Foraging simulation (HF) [1], the N-body simulation (NB) [14], and the Transportation simulation (TS) [6]. The data settings are specified in Table 2. Initially the data are evenly distributed over a 2D simulated space. By default, Inc-part sets $N_x = 2048$, $N_y = 2048$, $N_z = 0$, $\alpha = 4$ and $\beta = 0.25$, and a tick duration of one second.

Table 2
Simulation data sets.

Benchmark	Scale
Fish School (FS)	#Fish: 1 billion
Honeybee Foraging (HF)	#Bees: 1 billion
N-body (NB)	#Points: 1 billion
Transportation (TS)	#Travelers: 1 billion

Compared schemes. Inc-part is compared against three schemes:

1) Partition with user-defined cost functions (PUC) [17], which employs user defined cost functions to guide the division of simulated space into equally-loaded data partitions based on the sample of input data;
2) Persistence-based load balancer (PLB) [22], which is a hierarchical persistence-based rebalancing algorithm that performs localized incremental rebalancing based on the previous task distribution;
3) Retentive work stealing (RWS) [22], which is an active-message-based hierarchical retentive work stealing algorithm. RWS employs split task queues and a portion of tasks in the queue can be randomly stolen by a thief.

PUC represents the approach that focuses on task-level load balancing. It tries to balance load among tasks and then distributes them over workers. PLB and RWS represent the approach that focuses on worker-level load balancing. In these two schemes, the input data is divided into a fixed set of tasks at the beginning, which are then to be redistributed in response to load changes, for load balancing among workers. Note that all these schemes are realized in Piccolo, with the main difference being the balancing methods used. For PLB and RWS, the simulated space is divided into equal-sized domains, and each worker is initially assigned 128 such domains. The sampling rate of PUC is set to 1% as with SkewReduce [17].

Performance metrics. The performance evaluation mainly uses three metrics.

1) *Load imbalance degree* (LID), which reflects the load difference among workers, and is defined as
$$\text{LID} = L_{max}/L_{avg} - 1, \quad (5)$$
where L_{max} and L_{avg} are the maximum and average loads of all the workers, respectively.
2) *Runtime overhead*, which includes computation and communication costs. We measure the processing time $\mathrm{P}_a(i)$ and the real data processing time $\mathrm{P}_r(i)$ of each worker, as well as the number of messages $\mathrm{C}(i)$ sent by each worker. (Note that it does not include the communication to process simulated objects, which is the communication needed by the behavioral simulation.) Then the computation overhead is the average of difference between $\mathrm{P}_a(i)$ and $\mathrm{P}_r(i)$ over all workers:
$$P = \sum_i \frac{P_a(i) - P_r(i)}{i}, \quad (6)$$
and the communication overhead is evaluated by the sum of messages over all workers, that is,
$$C = \sum_i C(i). \quad (7)$$
3) *Speedup*, which is measured by the execution time of a scheme compared against that of a naive implementation without using any load balancing method.

5.1 Accuracy of load cost model

We first examine the accuracy of Inc-part's load cost evaluation method, defined as
$$\text{AR} = 1 - \frac{1}{n}\sum_{i=1}^{n} |L_c(i) - L_t(i)|/L_t(i), \quad (8)$$
where $L_c(i)$ is the load cost of domain i calculated using our load cost evaluation model, and $L_t(i)$ is the true load cost. Figure 5 shows the accuracies of Inc-part and PUC. (PLB and RWS assume a fixed load cost of the domains, so the accuracy metric is irrelevant for them.) It can be seen that Inc-part has an accuracy of up to 91.2% on the TS benchmark, which is higher than PUC's accuracy 85.1%. This is because Inc-part is not based on data sampling as with PUC. Due to the skewed distribution of domain loads, the sampling process is likely to be biased, which negatively affects PUC's accuracy of load estimation.

5.2 Load imbalance degree (LID)

We first confirm the necessity of re-balancing loads every tick by running PUC and measuring the LID

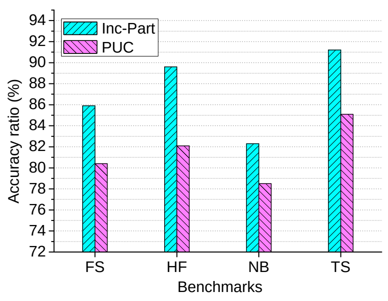


Figure 5. Accuracy of load cost model for Inc-part and PUC on different benchmarks.

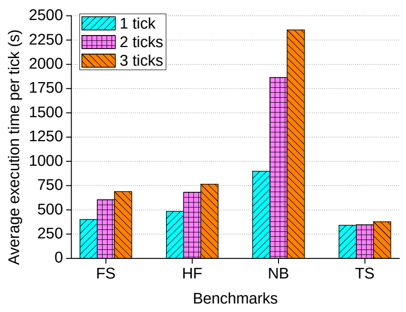


Figure 6. Performance of an epoch-based repartitioning approach with different epoch lengths.

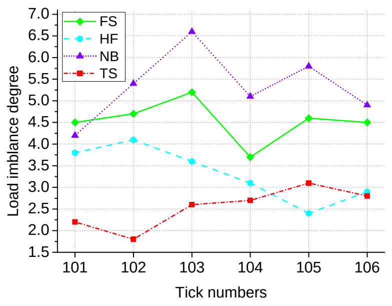


Figure 7. Load imbalance degree of PUC without load balancing, measured at the end of each tick.

at the end of each tick, right before any re-balancing procedure starts.

To demonstrate whether loads need to be re-balanced in every time step and whether the associated overhead can be amortized by muliple ticks, the performance of an epoch-based repartitioning approach with the PUC balancer is evaluated. This epoch-based method can tradeoff between balancing overhead and LID through the parameter epoch length. The results are given in Figure 6. Although PUC's runtime overhead is indeed amortized by multiple ticks in an epoch, the delayed balancing operation significantly increases load imbalance and reduces the performance.

Figure 7 shows the LID results for six successive ticks. It can be seen that the high load imbalance may still arise at the end of the new tick although it is redressed by PUC at the end of the previous tick. For example, for the NB benchmark, the LID is constantly above 4 and can be up to 6.6. This means that in order for the task execution to be balanced in the next tick, load re-balancing must be performed at the end of each tick. In what follows, we will make Inc-part and PUC re-partition and re-schedule the domains at the end of every tick.

Figure 8 shows how much change in domains contribute to load imbalance degree as described in Figure 7. The results include the ratio of domains which have increased load by more than 0.01% of the total load after a tick and the sum of the increased load ratio for these domains in this tick, namely

$$T_c = \sum_i |L_a(i) - L_b(i)|, \tag{9}$$

where $L_a(i)$ and $L_b(i)$ are the load ratios of domain D_i against the total load of all domains after and before the finish of a tick, respectively. From this figure, we can find that in behavioral simulation a small number of domains may see significant load changes which result in high load imbalance.

Figure 9 shows the LIDs of the various schemes for the benchmarks at tick 100. It shows that without load balancing, the LID can be as high as 18.4 due to some very densely populated domains. PLB and RWS also suffer from serious load imbalance because they only partition the domains in the very beginning of the simulation. Due to group migration, the initial balance will soon be lost and straggler domains arise. Worse still, the straggler domains are not fixed, but rather appear unpredictably among the workers, making it very difficult to balance the loads with PLB and RWS. In contrast, Inc-part keeps the LID below 0.69, since it selectively re-profiles and re-balances the domain loads at the end of every tick. For the same reason, PUC achieves almost the same LID (though with much higher overheads, as discussed later).

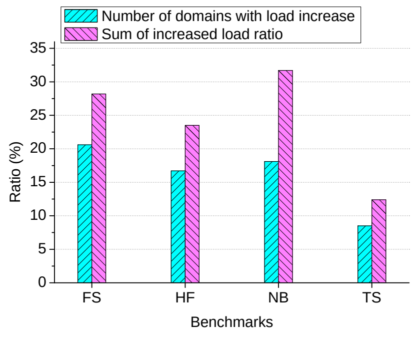


Figure 8. Load changes condition of domains.

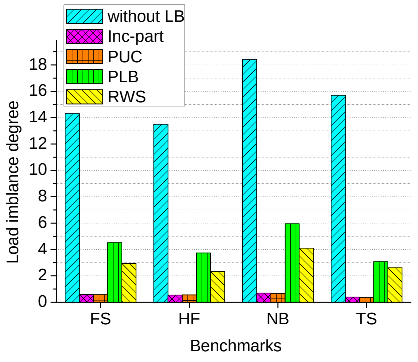


Figure 9. LID of various schemes in tick 100.

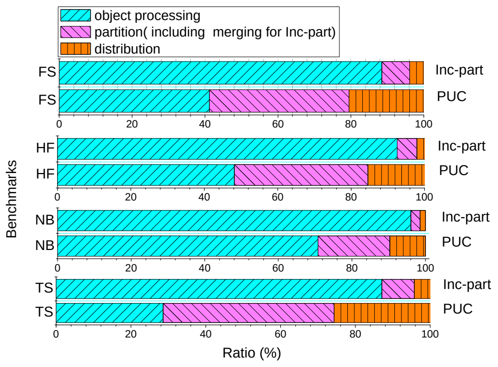


Figure 10. Runtime breakdown of Inc-part and PUC for various benchmarks. The object processing, partition, and distribution sections represent the proportions of time spent on real object processing, load cost profiling and domain partition, and domain balancing among workers, respectively.

5.3 Runtime overheads

Figure 10 shows the proportions of the total execution times spent on three tasks, namely real object processing, load cost update and domain partitioning, and domain distribution, of the various schemes during a tick. This breakdown reflects how much time a scheme spends on actual simulation and how much on load balancing. It can be seen that PUC spends almost 71.4% of its time on domain management for TS, including partitioning and distribution. Furthermore, it spends 45.7% of its time in the partitioning step on parallel sampling on workers, finding partition points and partition planning on workers. Another 25.7% of its time is spent in the redistribution step on notifying every worker of its new partitioning results, as well as on dispatching the new data domains to the workers. Thanks to the incremental nature of Inc-part's load balancing, it spends only 12.9% of its time on load balancing, including 8.65% on partitioning and 4.27% on domain redistribution.

Figure 11 shows the computational and communication overheads of the various schemes divided by Inc-part's respective overheads. Inc-part's computational overheads per tick are 21.5s, 19.3s, 26.4s and 15.2s for FS, HF, NB and TS, respectively. Also, its per tick communication overheads are 21.3K, 15.9K, 27.7K and 14.1K messages for FS, HF, NB and TS, respectively. Note that the overhead of PLB is incurred at the beginning of a tick, while RWS causes overhead within a tick. We can see that PUC incurs at least 16 times as much overhead as Inc-part does for the TS benchmark. This great difference is attributed to two reasons: (1) PUC has to profile load cost and redecompose the entire simulation space at the end of each tick, in order to eliminate stragglers; (2) PUC needs to migrate large numbers of objects for load re-balancing between workers.

In contrast, Inc-part updates the domains' load cost incrementally and conservatively, in the sense that only very dynamic domains update their load costs. Moreover, the domain partitioning only involves a small number of domains, so the incurred cost remains low. Because the load cost distribution changes every tick, the load information profiled from previous ticks will be largely useless at the new tick. As a result, as shown in Figure 11, the load balancing of RWS is rendered ineffective. For the NB benchmark, RWS's computational and communication overheads are as high as 2.72 and 4.13, respectively.

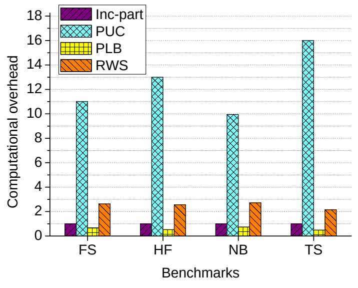
5.4 Speedup

Let the execution time of a simulation without using any load balancing schemes be T_{raw}. The speedup of a scheme is defined as T_{raw} divided by its execution time. Figure 12 shows the speedup of the various schemes for the four benchmarks. It can be seen that Inc-part produces a significantly higher speedup than other schemes. With Inc-part, the per tick's execution times for FS, HF, NB and TS are 186.7s, 260.1s, 665.8s and 117.6s, respectively. The superiority of Inc-part is due to its efficient utilization of the locality property of group migration, and its careful actions to preserve the locality of domains while re-balancing the load costs. Though PUC also yields a fairly high speedup of up to 6.8, this comes at a great price of computational and communication costs, as shown in the previous section. Notice that the speedups of PLB and RWS are only 2.28 and 3.16, respectively, because of their high LID.

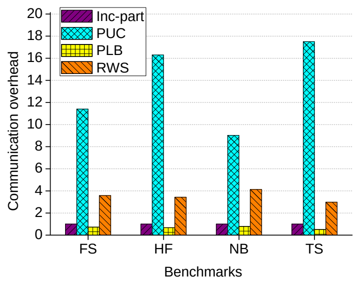
For the FS benchmark, Inc-part generates a speedup of up to 7.34, more than twice higher than that of PUC. For the TS benchmark, the speedup of PUC is even lower than those of PLB and RWS.

Figure 13 shows the scalability of Inc-part under the FS benchmark, measured by the number of processed fish objects per second. It can be seen that Inc-part attains good scalability.

Figure 14 shows the impact of the number of domains for the FS benchmark. In this experiment, we aim to demonstrate that finer grained decomposition of data does not always mean the better performance for PLB and RWS. It can be seen from the figure that the speedup first increases, reaching the maximum at a certain point before it declines afterwards. Though more domains help reduce the LID due to more fine-grained load scheduling, the benefit is gradually canceled out by the increased cost of managing the over-segmented domains. For RWS, when the number of domains grows, stealing operations become more frequent, and the invalid load information profiled from the previous ticks will cause a high miss ratio for the stealing operations, which brings down the performance. Figure 14 also shows that PLB performs



(a) Computational overhead.



(b) Communication overhead.

Figure 11. Computational and communication overheads of the various schemes against Inc-part's respective overheads.

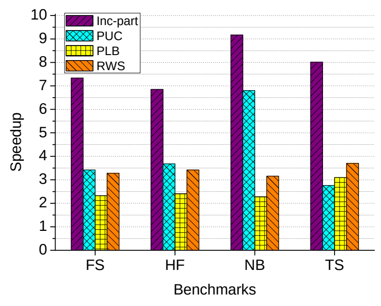


Figure 12. Speedup of the various schemes.

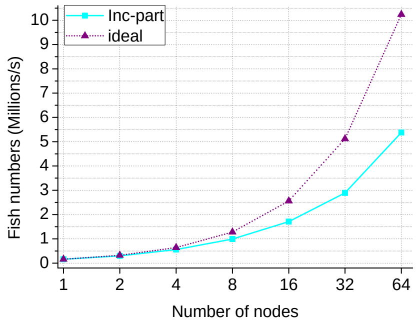


Figure 13. Scalability of Inc-part for the FS benchmark.

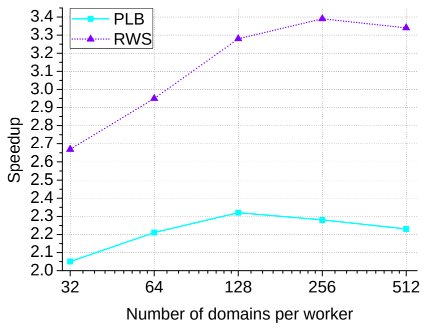


Figure 14. Impact of number of domains on the speedup of PLB and RWS for the FS benchmark.

worse than RWS because the invalid load information profiled from the previous ticks leads to ineffective task redistribution in PLB. In contrast, RWS can mitigate this problem to some extent, via task stealing.

6 Related work

Load balancing is an important problem in behavioral simulations. Current solutions for this problem either focus on task-level load balancing, or on worker-level balancing. The main related works are summarized as follows. Others are provided in Appendix E.

Task-level load balancing. SkewReduce [17] is a state-of-the-art solution for reducing load imbalance among tasks, in view that in some scientific applications, different partitions (domains) of data take vastly different amounts of time to run even if they have an equal size. It proposes to employ the user defined cost function to guide the division of simulated space into equal-loaded, rather than equally-sized, data partitions.

Worker-level load balancing. Persistence-based load balancers (PLB) [22] and retentive work stealing (RWS) [22] represent the approaches to balancing loads among workers for iterative applications. PLB redistributes the work to be performed in a given iteration based on measured performance profiled from previous iterations. RWS is used for applications with significant load imbalance within individual phases, or applications with workloads that cannot be easily profiled. RWS records the task information of previous iterations for work stealing to achieve higher efficiency. Both of these two approaches are based on statically partitioned domains, whose loads tend to change dramatically over time and exhibit high imbalance. The load profiling process also fails to track the load changes. These problems give rise to stragglers that appear unpredictably in the system, seriously slowing down the execution of programs.

Other load balancing strategies. On the other application domains, force-decomposition and equivalent approaches were proposed to balance load. For example, NAMD [25] not only employs persistence-based load balancer but also considers decomposition of forces rather than particles to achieve greater levels of strong scaling than allowed by just partitioning the domain amongst processor cores. Such force decomposition would be, by definition, balancing the computation load rather than the number of particles. Moreover, it also takes into account the distance cut-off similar to what is employed in our approach as locality-cognizant costs.

7 Conclusion

This paper focuses on the load balancing problem for large-scale behavioral simulations. Traditional solutions cannot efficiently balance loads among the workers because of the group migration phenomenon. We propose a new approach, Inc-part, that solves the problem by taking advantage of the effect of localized mobility found in typical behavioral simulations. Experiments on a 64-node (1024-core) cluster show that Inc-part can keep load imbalance within a very low level, with dramatically lower overheads compared with state-of-the-art approaches.

In the future work, we will investigate how to automatically adjust parameters for Inc-part to obtain the best performance under general conditions, and will consider the applicability of incremental partitioning to other iterative parallel applications. With Inc-part, large-scale behavioral simulation may suffer from network jitter in the cloud for global synchronization. It might be helpful to integrate the communication-avoidance method with our approach in order to make such applications more efficient in the cloud.

ACKNOWLEDGMENTS

This paper is supported by China National Natural Science Foundation under grant No. 61272408, 61322210, National High-tech Research and Development Program of China (863 Program) under grant No.2012AA010905, CCCPC Youngth Talent Plan, Doctoral Fund of Ministry of Education of China under grant No. 20130142110048. Guang Tan's work was partly supported by NSFC under grants 61103243 and 61379135.

REFERENCES

[1] "Biological modeling and simulation," http://zool33.uni-graz.at/schmickl/index.html, 2012.

[2] S. Camazine, J. Deneubourg, N. Franks, J. Sneyd, G. Theraula, and E. Bonabeau, *Self-organization in biological systems*. Princeton University Press, 2003.

[3] C. Hemelrijk and H. Kunz, "Density distribution and size sorting in fish schools: an individual-based model," *Behavioral Ecology*, vol. 16, no. 1, pp. 178–187, 2005.

[4] A. Huth and C. Wissel, "The simulation of the movement of fish schools," *Journal of theoretical biology*, vol. 156, no. 3, pp. 365–385, 1992.

[5] B. Raney and K. Nagel, "Iterative route planning for large-scale modular transportation simulations," *Future Generation Computer Systems*, vol. 20, no. 7, pp. 1101–1118, 2004.

[6] N. Cetin, K. Nagel, B. Raney, and A. Voellmy, "Large-scale multi-agent transportation simulations," *Computer Physics Communications*, vol. 147, no. 1, pp. 559–564, 2002.

[7] B. Raney and K. Nagel, "Truly agent-based strategy selection for transportation simulations," in *82nd Annual Meeting of the Transportation Research Board, Washington, DC*, 2003.

[8] J. Bartholdi, T. Seeley, C. Tovey, and J. Vate, "The pattern and effectiveness of forager allocation among flower patches by honey bee colonies," *Journal of Theoretical Biology*, vol. 160, no. 1, pp. 23–40, 1993.

[9] J. Biesmeijer and H. de Vries, "Exploration and exploitation of food sources by social insect colonies: a revision of the scout-recruit concept," *Behavioral Ecology and Sociobiology*, vol. 49, no. 2, pp. 89–99, 2001.

[10] H. de Vries and J. Biesmeijer, "Modelling collective foraging by means of individual behaviour rules in honey-bees," *Behavioral Ecology and Sociobiology*, vol. 44, no. 2, pp. 109–124, 1998.

[11] S. Aarseth, M. Henon, and R. Wielen, "A comparison of numerical methods for the study of star cluster dynamics," *Astronomy and Astrophysics*, vol. 37, pp. 183–187, 1974.

[12] Y. Kwon, D. Nunley, J. Gardner, M. Balazinska, B. Howe, and S. Loebman, "Scalable clustering algorithm for n-body simulations in a shared-nothing cluster," in *Scientific and Statistical Database Management*. Springer, 2010, pp. 132–150.

[13] J. Makino and H. Daisaka, "Grape-8 – an accelerator for gravitational n-body simulation with 20.5gflops/w performance," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society Press, 2012, pp. 1–10.

[14] V. Springel, S. White, A. Jenkins, and et al., "Simulations of the formation, evolution and clustering of galaxies and quasars," *nature*, vol. 435, no. 7042, pp. 629–636, 2005.

[15] D. Schrank, B. Eisele, and T. Lomax, "Tti's 2012 urban mobility report," 2012.

[16] T. Zou, G. Wang, M. Salles, D. Bindel, A. Demers, J. Gehrke, and W. White, "Making time-stepped applications tick in the cloud," in *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)*. ACM, 2011, p. 20.

[17] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skew-resistant parallel processing of feature-extracting scientific user-defined functions," in *Proceedings of the 1st ACM symposium on Cloud computing (SOCC)*. ACM, 2010, pp. 75–86.

[18] G. Wang, M. Salles, B. Sowell, X. Wang, T. Cao, A. Demers, J. Gehrke, and W. White, "Behavioral simulations in mapreduce," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 952–963, 2010.

[19] E. Bonabeau, M. Dorigo, and G. Theraulaz, *Swarm intelligence: from natural to artificial systems*. Oxford University Press, USA, 1999, no. 1.

[20] S. Camazine, J.-L. Deneubourg, N. R. Franks, J. Sneyd, G. Theraula, and E. Bonabeau, *Self-organization in biological systems*. Princeton University Press, 2003.

[21] N. Luhmann, *Social systems*. Stanford University Press, 1996.

[22] J. Lifflander, S. Krishnamoorthy, and L. Kale, "Work stealing and persistence-based load balancers for iterative overdecomposed applications," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing (HPDC)*. ACM, 2012, pp. 137–148.

[23] R. Power and J. Li, "Piccolo: building fast, distributed programs with partitioned tables," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)*. USENIX Association, 2010, pp. 1–14.

[24] N. Francez, "Distributed termination," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 2, no. 1, pp. 42–55, 1980.

[25] S. Kumar, C. Huang, G. Zheng, E. Bohm, A. Bhatele, J. C. Phillips, H. Yu, and L. V. Kalé, "Scalable molecular dynamics with namd on the ibm blue gene/l system," *IBM Journal of Research and Development*, vol. 52, no. 1.2, pp. 177–188, 2008.