Yan J, Tan GM, Sun NH. Optimizing parallel $S_n$ sweeps on unstructured grids for multi-core clusters. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 28(4): 657–670 July 2013. DOI 10.1007/s11390-013-1366-9

# Optimizing Parallel $S_n$ Sweeps on Unstructured Grids for Multi-Core Clusters

Jie Yan[1,2] (闫　洁), *Student Member, IEEE*, Guang-Ming Tan[1] (谭光明), *Member, CCF, ACM, IEEE* and Ning-Hui Sun[1] (孙凝晖), *Fellow, CCF, Member, ACM, IEEE*

[1] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China*

[2] *University of Chinese Academy of Sciences, Beijing 100049, China*

E-mail: {yanjie, tgm, snh}@ict.ac.cn

Received May 10, 2012; revised February 7, 2013.

**Abstract** In particle transport simulations, radiation effects are often described by the discrete ordinates ($S_n$) form of Boltzmann equation. In each ordinate direction, the solution is computed by sweeping the radiation flux across the grid. Parallel $S_n$ sweep on an unstructured grid can be explicitly modeled as topological traversal through an equivalent directed acyclic graph (DAG), which is a data-driven algorithm. Its traditional design using MPI model results in irregular communication of massive short messages which cannot be efficiently handled by MPI runtime. Meanwhile, in high-end HPC cluster systems, multicore has become the standard processor configuration of a single node. The traditional data-driven algorithm of $S_n$ sweeps has not exploited potential advantages of multi-threading of multicore on shared memory. These advantages, however, as we shall demonstrate, could provide an elegant solution resolving problems in the previous MPI-only design. In this paper, we give a new design of data-driven parallel $S_n$ sweeps using hybrid MPI and Pthread programming, namely `Sweep-H`, to exploit hierarchical parallelism of processes and threads. With special multi-threading techniques and vertex schedule policy, `Sweep-H` gets more efficient communication and better load balance. We further present an analytical performance model for `Sweep-H` to reveal why and when it is advantageous over former MPI counterpart. On a 64-node multicore cluster system with 12 cores per node, 768 cores in total, `Sweep-H` achieves nearly linear scalability for moderate problem sizes, and better absolute performance than the previous MPI algorithm on more than 16 nodes (by up to two times speedup on 64 nodes).

**Keywords** parallel $S_n$ sweep, unstructured grid, data-driven algorithm

## 1 Introduction

Grid-based computational pattern is widely used in scientific computing applications. In a grid-based numeric algorithm, the computational domain is discretized into grids, and computation is conducted on grid cells.

Numerical simulation of radiation transport in high energy density plasma physics is an exemplary application. Statistics[1] show that time devoted to particle transport problem in multi-physics simulations costs 50%∼80% of total runtime on Department of Energy (DOE) systems.

Radiation effects are often modeled by the discrete ordinates ($S_n$) form of Boltzmann transport equation. The standard solving method is source iteration, in which the solution is computed by iteratively repeating two phases. First, compute local scattering source. Second, for each ordinate direction, sweep the radiation flux from the source across the grid in the downstream direction. In the second phase, sweeps from all ordinate directions usually can be carried out in parallel. The second phase is also the most time-consuming portion, which is simply denoted as `Sweep(s)`.

This paper focuses on the parallelization of `Sweeps`, particularly on unstructured grids. During `Sweep` in any given ordinate direction, there exists strict data dependency between neighboring grid cells, i.e., one cell has to wait for its upwind neighbors' newest data and thus cannot start computing until all its upwind neighbors have been computed.

For structured grids, data dependencies are regular and can be decided by simple mathematical calculation. Thus, a domain-based grid partitioning and wave-

Regular Paper

This work is supported by the National Natural Science Foundation of China under Grant Nos. 60925009, 61003062, 61033009, 61272134, and the National Basic 973 Program of China under Grant Nos. 2011CB302502 and 2012CB316502.

front parallelization approach could effectively pipeline the computation, as shown by the success of KBA algorithm[2-3] and Sweep3D[①] benchmark. It is usually implemented in the classic Bulk Synchronization Parallel model[4].

Nevertheless, the above approach generally does not work for the unstructured grid because of its irregular data dependency. Instead, Plimpton *et al.*[5-6] and Mo *et al.*[7] introduced a graph-based data driven approach to address this problem. In their approach, for each sweep direction (or angle), the grid is explicitly transformed into a directed acyclic graph (DAG) according to data dependencies among cells, and then at each time step `Sweep` on the grid is equivalent to a topological traverse across the resulted DAG. For example, Fig.1 illustrates the correspondence between a grid and its associated DAG under a given sweep direction. This approach essentially uses the asynchronous Actors dataflow model[8], i.e., a vertex can compute as soon as its dependent data from upwind vertices is available. Conceptually, it has no explicit synchronization and can expose all potential parallelism. However, it requires massive immediate messages passing through edges among vertices of the DAG, which, as shown later, cannot be supported well in current distributed memory systems. Thus, in practice, compared with the great success on structured grids, `Sweeps` on unstructured grids still suffer problems in performance and scalability.

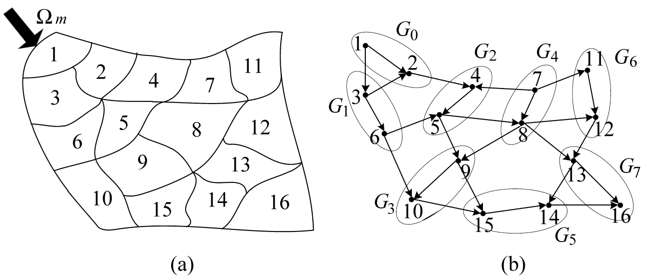


Fig.1. Illustration of an unstructured grid and its associated DAG under a given sweep direction. The DAG is partitioned into 8 subgraphs ($G_0 \sim G_7$).

Meanwhile, the multi-core processor has become the standard computing engine on commodity clusters and high-end supercomputers, which means the system has a physically hybrid memory model (i.e., distributed among computing nodes and shared by cores within a single node). The local shared memory architecture of multi-core is able to effectively support fine parallelism in thread level. Compared with MPI multi-process model, multi-thread model on multi-core eliminates the need of explicit intra-node communication which is replaced by shared memory access with higher bandwidth and lower latency. Besides, as shown in later sections, load balance among shared memory threads is easier to maintain, and local performance is less sensitive to the latency of messages from remote machines. Therefore, to introduce multi-threading parallelism within the MPI process is of great importance to the performance and scalability of $S_n$ `Sweeps` on modern multi-core cluster systems.

In this paper, we target at $S_n$ `Sweeps` on unstructured grids, and redesign the data-driven parallel $S_n$ `Sweeps` algorithm[6-7] to exploit to advantage of contemporary multi-core clusters (up to 12 or more cores in a single node). Specifically, we make three main contributions.

First, we propose a new parallel $S_n$ `Sweeps` algorithm, namely `Sweep-H`, which adopts a hybrid programming model of MPI/Pthreads. In `Sweep-H`, each computing node is assigned only one MPI process that runs multiple threads. By multi-threading, `Sweep-H` replaces MPI communication within a node with direct accesses to shared memory, while significantly reduces the number of processes and increases the subgraph size, which leads to more intra-node parallelism. `Sweep-H` employs the master-workers multi-threading mode, which decouples computation (done by the workers) from both DAG control and communication (done by master). With this strategy, `Sweep-H` provides better supports to instant and asynchronous communication, and thus enables more inter-node runtime parallelism. Besides, we introduce dedicated task queues and a specific load balance policy with which `Sweep-H` works efficiently.

Second, we present a performance model for `Sweep-H` and its MPI-only counterpart `Sweep-MPI`. The model depicts `Sweep-H`'s speculated scalability over problem size and system scale, using parameters of system configuration and problem characteristics.

Third, we implement parallel programs of both `Sweep-H` and `Sweep-MPI` and conduct experiments on a 64-node multi-core cluster with 12 cores per node. Results show that `Sweep-H` obtains high computation efficiency (85.6% ~ 90.2%) and nearly linear scalability on up to 64 nodes. Analysis of the profiling data reveals that `Sweep-H` reduces communication overhead while gets much better load balance.

In the rest of this paper, we first retrospect the previous pure MPI implementation of data-driven parallel `Sweeps` algorithm and introduce our motivations in Section 2. In Section 3 we describe our new algorithm `Sweep-H`. We give the performance model of `Sweep-H` and its MPI alternative in Section 4. Section 5 reports

①Los Alamos National Laboratory. The ASCI Sweep3D Benchmark. http://www.c3.lanl.gov/pal/software/sweep3d/, Jan 2013.

our experimental results of `Sweep-H` with comparison to its MPI alternative, as well as profiling and analysis in detail. Related work is presented in Section 6. Finally, we discuss missed issues of this paper and conclude the paper in Section 7.

## 2 Background and Motivation

Fig.2 is the complete workflow of data-driven $S_n$ `Sweeps` for static grids, including three main procedures. First, it eliminates cycles in the unstructured grid, and then transforms the grid into a computable DAG[②]. Plimpton *et al.*[6] proposed an algorithm to detect and eliminate the cycles. Second, it partitions the DAG into multiple subgraphs that are then distributed to processes (processors). The graph partitioning has a direct impact on load balance and further parallel efficiency. Fortunately, previous studies[9-11] on graph theory and parallel computing have already provided solutions to this problem. Third, it iteratively carries out `Sweeps` computation for required time steps. Each processor performs sweep computation on its local DAG, and communicates with other processors if there are cutting edges between their subgraphs. As a matter of fact, while the former two procedures can be considered as pre-processing steps, the third one — `Sweeps`, is the main computation part in the $S_n$ `Sweeps` algorithm and thus is our redesigning target in this paper.

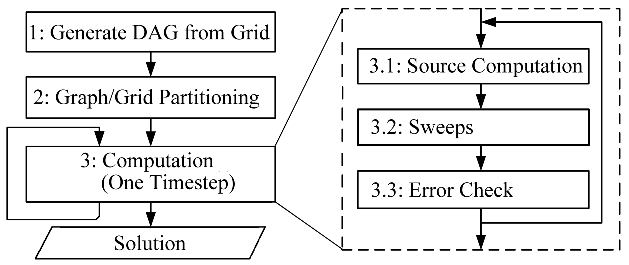

1: Generate DAG from Grid
2: Graph/Grid Partitioning
3: Computation (One Timestep)
Solution
3.1: Source Computation
3.2: Sweeps
3.3: Error Check

Fig.2. Workflow of graph-based approach for parallel $S_n$ `Sweeps`. In this paper, we only consider the most time-consuming part – `Sweeps` in the 3rd procedure.

Besides, note that `Sweeps` for all ordinate directions (angles) are carried out in parallel, and for most problems they are independent. In the DAG-based data-driven approach, sweep from different angles induces different graphs. In this paper, we treat all these subgraphs as a whole and implicitly leverage parallel `Sweeps` from all angles.

### 2.1 Parallel $S_n$ `Sweeps` Algorithm

On the old generations of distributed memory systems, a parallel $S_n$ `Sweeps` algorithm (referred as `Sweep-MPI` in this paper) was presented in [6-7, 12] and implemented using MPI.

Fig.3 is the pseudo-code of a basic `Sweep-MPI` algorithm. Each process holds a subgraph as well as cells' data of the associated piece of grid.

```
Data Structures:
$DAG$: subgraph $(V, E)$, where $V = \{(v_i)\}$, $E = \{(v_i, v_j)\}$
$N[|DAG.V|]$: array of counters for every vertex to record its unfinished upwind vertices
$Cn$: counter of local unfinished vertices
$RQ$: ready queue
Algorithm:
Procedure Sweep-MPI($DAG$)
1:  $RQ \leftarrow \varnothing$, $Cn \leftarrow |DAG.V|$
    //Initialization
2:  for each $v_i \in DAG.V$ do
3:      $N[v_i] \leftarrow Indegree(v_i)$
4:      if $N[v_i] = 0$ then
5:          Enqueue $v_i$ to $RQ$
    //Main loop
6:  while $Cn > 0$ do
7:      //1: Receive
8:      Receive $messages$
9:      for each $message \in messages$ do
10:         $(v_i, v_j$, data$[v_i]) \leftarrow message$
11:         $N[v_j] \leftarrow N[v_j] - 1$
12:         if $N[v_j] = 0$ then
13:             Enqueue $v_j$ to $RQ$
14:     //2: Compute
15:     Dequeue $v_k$ from $RQ$
16:     Compute($v_k$, data[...])
17:     $Cn \leftarrow Cn - 1$
18:     //3: Send
19:     for each $(v_k, v_{k'}) \in DAG.E$ do
20:         $message \leftarrow (v_k, v_{k'}$, data$[v_k])$
21:         Send $message$ to owner $(v_{k'})$
```

Fig.3. `Sweep-MPI` algorithm.

For every vertex $v_i$, one counter $N[v_i]$ is used to record the number of its unfinished upwind vertices. The counter value is initialized to $v_i$'s in-degree, and once the counter becomes zero, $v_i$ is set ready for computing. Another counter $Cn$ records the total number of unfinished vertices in local subgraph. The termination condition of one iteration is that all vertices are calculated (swept), that is, $Cn$ counts down to 0. Besides, a queue $RQ$ (ready queue) is used to record ready vertices during runtime. In the main loop (lines 6 $\sim$ 21), for a process $p$, it repeats the following work:

• *Receive*: Process $p$ polls other processes, and receives incoming messages which contain edge information $(v_i, v_j)$ and vertex data $data[v_i]$ used for computation. The received edge $(v_i, v_j)$ means $v_j$'s upwind

[②]In this paper, as explained later, we treat DAGs induced by sweeping from all angles as a whole.

vertex $v_i$ has been computed remotely, and thus we decrement the counter $N[v_j]$ by one. If $N[v_j]$ equals zero, that is, all upwind vertices of $v_j$ are swept, $v_j$ is appended into $RQ$.

• *Compute*: Process $p$ fetches a vertex $v_k$ from its ready queue $RQ$, and performs computation on $v_k$ with its dependent data. Accordingly, after $v_k$ finishes computation, the counter $Cn$ is decremented by one.

• *Send*: For each $v_k$'s outgoing edge $(v_k, v_{k'})$ , the edge information and data of $v_k$ are packed and sent to the process that owns $v_{k'}$. A communication optimization may be applied if the owner of $v_{k'}$ is the same as $v_k$'s, where we can directly decrement the counter $N[v_{k'}]$ without an extra remote communication.

For simplicity, the pseudo-code of `Sweep-MPI` in Fig.3 omits most of implementation details. In fact, there are two practical optimizations that are critical to performance.

First, the priority strategy should be added to ready queue $RQ$. In parallel `Sweeps`, ordering of ready vertex affects available runtime parallelism, which is critical to whole performance. In this paper, we use the priority of "length of shortest path away from processor boundary"[7,12], where the priority of vertices can be statically calculated in advance. The priority policy is implemented by adding sort operation in *Enqueue* function, while *Dequeue* still simply pop the item in the head of $RQ$.

Second, conceptually, communication in `Sweep-MPI` is asynchronous, instant and always in short messages. However, in practice MPI has much overhead to support this. For this reason, previous implementations of Plimpton *et al.*[6] and Mo *et al.*[7] adopt buffer mechanism as well as asynchronous MPI primitives, in which multiple short messages are aggregated into a larger one before real communication.

## 2.2 Motivation

`Sweep-MPI`, as stressed later, still suffers several performance problems due to the MPI model itself and its imposed restrictions on programming.

We first argue that the message buffering policy hinders available runtime parallelism and further leads to load imbalance. Although buffer policy makes overall data transferring more efficient, it is at the cost of potentially more delay for some messages. These delayed messages would lead to unnecessary remote starvation. We take the simple DAG in Fig.1 as an example in which the graph is partitioned into eight subgraphs ($G_0 \sim G_7$). Subgraph $G_4$ with vertices $v_7$ and $v_8$ is assigned to process $p_4$, and subgraph $G_6$ with vertices $v_{11}$ and $v_{12}$ is assigned to process $p_6$. According to the data dependency, process $p_6$ needs to receive messages from process $p_4$. With the buffer policy process $p_4$ aggregates results of computation on both $v_7$ and $v_8$, and then sends them together as a whole to $p_6$. As a result, process $p_6$ is idle while $p_4$ is computing on $v_7$ and $v_8$. Obviously, this strategy is not aware that the computation on $v_{11}$ does not depend on the results of $v_8$. If $v_7$'s result could be transferred to process $p_6$ as soon as possible, process $p_6$ would immediately compute $v_{11}$ while process $p_4$ is computing on $v_8$, and thus the runtime parallelism would increase.

Another performance issue of `Sweep-MPI` is raised by the mixture of computation and communication. In the data-driven `Sweeps` algorithm, the communication pattern is irregular and asynchronous, such that one process has to periodically poll other processes to check messages. In single thread mode, timely polling for communication would be impeded by computation, while polling operations disrupt computation too. For communication of massive short messages, as revealed by our profiling data for `Sweep-MPI`, this polling mechanism is not efficient enough.

In the next section, we propose a new parallel $S_n$ `Sweeps` algorithm that leverages the multi-core architecture to address the above two issues.

## 3 New Algorithm on Multi-Core Clusters

In this section we present our new parallel $S_n$ `Sweeps` algorithm, `Sweep-H`, that is dedicated to multi-core clusters. We first give the design principles, and then describe our algorithm in details with emphasis on how it addresses critical performance issues in parallelizing $S_n$ `Sweeps`.

### 3.1 Design Principles

Our target platform is a commodity multicore cluster system. Each node of the cluster is composed of multiple multi-core processors sharing memory physically. All nodes are connected by high speed interconnect devices like Infiniband. Data exchanges between two nodes are supported by message passing interface (MPI). We make a key point that hardware support for shared memory multi-threading on multicore could give rise to an elegant solution to the aforementioned problems in `Sweep-MPI`. We present our design principles as follows.

*Use Conserved Cores to Achieve Asynchrony*: For the nature of asynchronous and fine-grained parallelism in data-driven approach, Sweeps through DAG require instant communication to enable parallelism. With MPI communication primitives, as illustrated earlier, each process has to frequently poll incoming messages to make communication as instant as possible. Nevertheless, the frequent polling leads to significant overhead as in Sweep-MPI. Our strategy is to enable multi-threading within the MPI process, and assign a specific thread for communicating while others are dedicated to computing. Potentially this configuration decouples communication from computation and thus provides more advantage in asynchronous and instant communication. Besides, given the importance of instant communication, we conserve an exclusive core for the communicating thread (See Subsection 3.2).

*Transform MPI Communication to Shared Memory Access.* Communication in data-driven Sweeps is latency sensitive rather than bandwidth sensitive, because 1) only boundary vertices of subgraphs need to communicate and thus generally the volume of communicating data is limited, and 2) data-driven Sweeps require fast message delivery to enable future vertex computing. For this situation, MPI communication is much more expensive than shared memory access even if most of current MPI libraries are specifically optimized for shared memory. Looking at lines 20 $\sim$ 21 in Fig.3, one send operation involves actions of packing messages and transferring them through network (or local memory copy). As noted in Subsection 2.1, only a decrement of counter $N(v)$ is involved if the downwind vertex is local. Therefore, in our situation, shared memory accesses have obvious advantage over MPI operations.

*Share a Larger Subgraph to Gain Better Load Balance.* On each node, we should deploy only one process of multiple threads as well as data of its assigned subgraph, which benefits load balance from two aspects. On one hand, the larger local subgraph means that more potential runtime parallelism is decided by local vertices and thus is less sensitive to communication latency. On the other hand, within a node, load balance is easier to maintain among a group of shared memory threads than among a group of processes in logically distributed memory. (See Subsection 3.4)

Fig.4 conceptually demonstrates the above design principles, comparing Sweep-H and Sweep-MPI in terms of the way mapping subgraphs to processors, processor usage and communication. In Sweep-MPI, all data flows among subgraphs (both within the same node and across different nodes) go through MPI communications (red lines), and every processing core performs both computation and communication. Sweep-H conserves one processing core (labeled with $M$) per node for communication and messages processing, while other processing cores (labeled with $W$) perform computations on vertices of the *shared* subgraph. In Sweep-H, we can see that most of red lines are replaced with gray dash lines that represent shared memory accesses. Then a concern on Sweep-H is the potential conflicts of shared memory access, which incur extra overhead. In practice, we address this issue by two techniques, that 1) dividing the data space (see Subsection 3.2) and 2) bringing in dedicated shared queues as data channels among different kinds of threads (see Subsection 3.3).

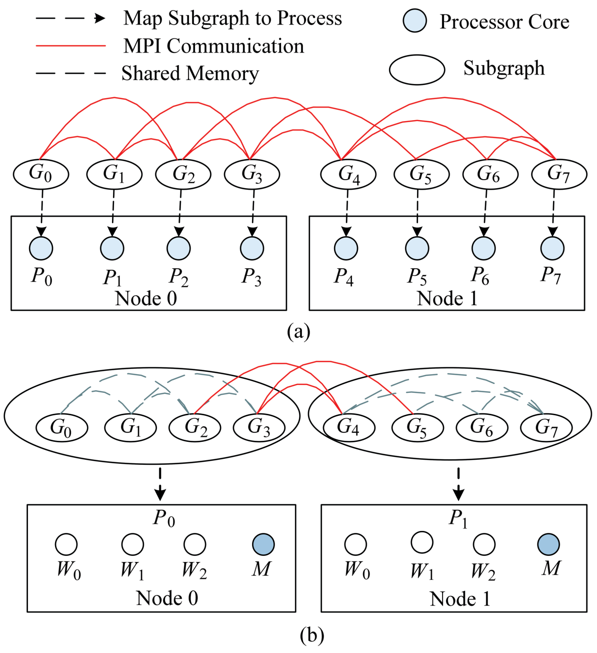


Fig.4. Conceptual demonstration of difference between (a) Sweep-MPI and (b) Sweep-H.

### 3.2 Hybrid Parallel Algorithm

We introduce multi-threads within a process to redesign the data-driven parallel Sweeps algorithm. Particularly, Sweep-H adopts master-workers multi-threading mode. Fig.5 illustrates the algorithmic framework of Sweep-H.

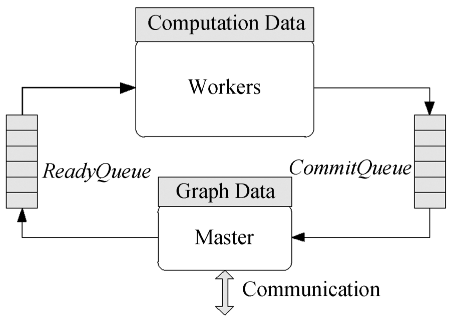


Fig.5. Illustration of Sweep-H algorithm.

First, we decouple computation from both communication and DAG operations. The master performs all MPI communication as well as DAG operations, while workers, which are instantiated of Pthreads, perform only numerical computation on the ready vertices.

Second, we divide the data space of Sweep-H into two disjoint parts, say graph data and computation data. Graph data is the DAG structure which is only used to schedule computation. Computation data, coupled with grid cells, is the data for solving numerical equations. Thus, the master operates only on graph data while workers compute only with computation data of assigned cells, which perfectly eliminates data race.

Extra dedicated data structures, *ReadyQueue* and *CommitQueue*, are designed to connect the master and workers. *ReadyQueue* is the queue of vertices that are ready for computing, where items are produced by the master and consumed by workers. *CommitQueue* is the queue of vertices that have been computed but yet not committed results, where items are produced by workers and consumed by the master.

The master loops the following steps:

1) receive messages and update states of local vertices; for any updated vertex $v$, if all its upwind neighbors (predecessors) are computed, insert $v$ into *ReadyQueue*;

while *CommitQueue* is not empty, repeat steps 2∼3:

2) get a vertex $u$ from *CommitQueue*;

3) for any $u$'s outgoing edge $(u, v)$

a) if $v$ is local, update $v$'s state; if all of $v$'s upwind neighbors (predecessors) are done, insert $v$ into *ReadyQueue*;

b) if $v$ is remote, send a message to its owner;

4) if all vertices are swept, exit; else goto step 1.

Worker threads repeatedly do the following work:

1) get a ready vertex $u$ from *ReadyQueue*;

2) compute on $u$;

3) commit $u$ to *CommitQueue*.

## 3.3 Shared Queues

Fig.6 illustrates our practical implementation of the framework shown in Fig.5.

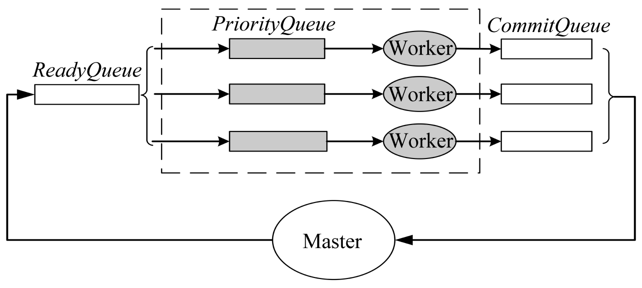


Fig.6. Framework of real Sweep-H implementation.

In Sweep-H, queue data structures play a critical role in vertex schedule and organization. *ReadyQueue* and *CommitQueue* are channels for vertex relay that drives the execution of workers and the master, while *PriorityQueue* is the worker's private vertex queue. During its lifetime, a vertex travels through *ReadyQueue*, *PriorityQueue* and *CommitQueue* in order.

*ReadyQueue* and *CommitQueue* are built with array-based circular queue (simply *RingArray*, as shown in Fig.7). When working as FIFO queue, *RingArray* is simple, fast to index, and with good locality in batch operation.

```
Date Structure:
struct RingArray{
int64_t size;
int64_t head;
int64_t tail;
lock_t lock_tail;
data_t data[*];
}
#define IsFull ($head - tail \geqslant size$)
#define IsEmpty ($head \leqslant tail$)
#define NumVertex ($head - tail$)
Operations:
Procedure Enqueue_SP(src, n)
1: $data[head\%size..(head+n-1)\%size]$
     $\leftarrow src[0..n-1]$
2: $head \leftarrow head + n$
Procedure Dequeue_SC(dest, n)
1: $tail \leftarrow tail + n$
2: $dest[0..n-1]$
     $\leftarrow data[tail\%size..(tail+n-1)\%size]$
Function Dequeue_MC(dest, n)
1: if $Lock\_Try(lock\_tail) = False$ then
2:     return Failure
3: $tail \leftarrow tail + n$
4: $dest[0..n-1]$
     $\leftarrow data[tail\%size..(tail+n-1)\%size]$
5: $Unlock(lock\_tail)$
```

Fig.7. *RingArray*: data structure and operations for building SPSC and SPMC queues.

*ReadyQueue*, conceptually, is an SPMC (single producer multiple consumer) priority queue. However, we design it as a simple FIFO queue in *RingArray*, leaving priority policy to workers' private *PriorityQueue*s. *ReadyQueue* is shared by all native workers, and every worker periodically fetches a batch of vertices from *ReadyQueue* and inserts them into its private *PriorityQueue* according to vertex priority.

The design of *ReadyQueue* is a trade-off of the following three aspects. First, it is shared and frequently accessed by both the master and workers, which means it should be non-blocking. In our design, *Enqueue* operation by the master is lock-free and fence-free, while *Dequeue* operation by workers uses one lock for serialization. In our situation, however, this lock has almost no blocking overhead. This is because 1) *Dequeue* is optimized to fetch data in batch, and 2) unless its own *PriorityQueue* is empty a worker just returns from the failure of getting lock and defers its fetch operation to next time. Second, *ReadyQueue* should be convenient to support implementation of load balance policy among workers. We adopt a central design, i.e., *ReadyQueue* is shared by all native workers, so that workers themselves can maintain load balance by controlling how many items to be fetched from *ReadyQueue*. Third, *ReadyQueue*, as indicated by `Sweep-MPI`, essentially should be a priority queue for maximizing runtime parallelism. However, sort on a shared queue risks serious data access blocking. Instead, we make *ReadyQueue* an FIFO queue, and defer the priority policy to be done later independently by workers. This is a partial priority policy which however is demonstrated effective enough in our experiments.

*PriorityQueue* is introduced as a worker's private data structure to keep prioritized ready vertices. It is implemented as an ordinary priority queue in single-linked list, owned privately by each worker thread, and invisible to the master and other workers. Every worker gets a batch of items periodically from *ReadyQueue* and inserts them into its *PriorityQueue* in vertex priority order. When compute, the worker just picks the vertex with the highest priority in its *PriorityQueue*. In our design, the value of each vertex's priority has been computed statically in advance.

*CommitQueue*, conceptually, is an MPSC (multiple producer single consumer) queue without priority among its items. It buffers the vertices that have been computed by workers and wait for the master to commit. As shown in Fig.6, the MPSC queue is equivalently transformed into a series of SPSC FIFO queues. For each SPSC FIFO queue, there is only one worker being its producer, and the master goes around all the queues to consume vertices in batch. Because the *RingArray*-based SPSC FIFO queue is lock-free and fence-free for both the producer and consumer, so is the resulted MPSC queue.

## 3.4 Schedule Policy and Load Balancing

Scheduling concerns vertices distribution and ordering among processors, and it is a key factor deciding parallel performance[13]. As explained earlier, graph partitioning, distribution of subgraphs among cluster nodes, and calculation of vertex priority are done independently as preprocessing steps. As in `Sweep-MPI`, `Sweep-H` adopts the priority of "length of shortest path away from processor boundary"[7,12], while it is free to use other priority strategies. In rest of this subsection, we only need to consider the case within a process.

In `Sweep-H`, we adopt a partial priority policy, where vertex reordering by priority is executed by each worker on its local *PriorityQueue* rather than the global *ReadyQueue*. This means that, from the view of a process, vertices are swept in a partial priority order. Theoretically, the potential priority inversion could happen among workers, which leads to non-optimal parallelism and thus unnecessary starvation of remote machines. In this paper, we have not yet developed a theoretical model to estimate the price of priority inversion. However, in `Sweep-H`, the price of priority inversion is expectedly very low, because 1) multi-threading makes priority inversion distance shorter, and 2) dynamic vertex fetching from *ReadyQueue* introduces enough randomness to lower probability of serious inversion. In practice, our method works effective enough while keeps runtime overhead of vertex reordering low.

The final issue is the load balance among workers within a process. Once a ready vertex enters a worker's *PriorityQueue*, it becomes invisible to both the master and other workers. Thus, policy like work-stealing does not work here. Instead, we propose a new one that is more effective in our specific situation. In our scheme, the master simply inserts ready vertices into *ReadyQueue*. Meanwhile, the worker periodically fetches items from *ReadyQueue* to fill its *PriorityQueue* according to a rule that is simply called Load Balance principle.

**Definition 1** (Load Balance Principle). *Every time, the number of items a worker fetches from ReadyQueue is at most* $(Rn+Cn)/Wn-Pn$*, where* $Rn$ *is the number of vertices in ReadyQueue,* $Cn$ *is the total number of vertices in all workers' PriorityQueue,* $Wn$ *is the number of workers in the process, and* $Pn$ *is the number of vertices in the worker's PriorityQueue.*

The idea behind Load Balance principle is that: at any time point when a worker tries to fetch items from *ReadyQueue*, it adaptively adjusts its workload to the approximately average level. Omitting noise in execution and given fixed vertex computation grain, this dynamic policy is theoretically ideal for keeping load balance among workers. In our experiments, as will be as shown in Subsection 5.3, its effect is close to the ideal case.

## 4 Performance Model

In this section, we present the performance model of `Sweep-H`, along with `Sweep-MPI` as a comparison. Based on the model, we further reveal why and when `Sweep-H` performs better than `Sweep-MPI`, and gives performance tuning guidelines for `Sweep-H`. Note that focus of our model is the method of parallelization rather than numerical characteristics of $S_n$ `Sweeps`. By a set of assumptions, we actually separate complexity of unstructured grids and numerical solvers.

In the following analysis, we assume 1) the cluster is uniform that all nodes have the same configuration, 2) computation on a vertex is uniform that computing time on a vertex can be reduced to a constant value. Besides, it has to be emphasized that except numerical computation on vertex, all the other costs are treated as overhead. For convenience, Table 1 defines several notations used in our performance model.

**Table 1.** Notations of Variables in Performance Model

| Variable | Description |
|---|---|
| $P$ | Number of cluster nodes |
| $C$ | Number of cores per node |
| $U$ | Computation granularity on one vertex |
| $V$ | Number of vertices in DAG |
| $O_g$ | Overhead of operations on DAG |
| $O_c$ | Communication overhead between any two nodes |

First, given a DAG denoted as $G$, in the ideal case where all overhead is omitted, we have the following equations:

$$T_1 = Work(G) = U \times V,$$

$$T_{n=c\times p} \geqslant \frac{Work(G)}{C \times P}, \tag{1}$$

$$T_\infty = U \times D, \tag{2}$$

where $T_n$ is the execution time with $n$ cores and $D$ is the diameter of DAG. Formula (2) describes the theoretic optimal performance, where the computing resources are infinite and run time is only determined by the critical path.

**Lemma 1.** *The necessary condition of an optimal parallel* `Sweeps` *algorithm is* (3).

$$V/D \geqslant C \times P. \tag{3}$$

*Proof.* In an optimal parallel algorithm, there is no idle processor time, and thus the equality in (1) follows. Meanwhile, $T_n \geqslant T_\infty$ always holds. Hence, by direct mathematical substitution with (1) and (2), we get (3) from the inequality. □

The above analysis gives the upper bound of benefits gained from increasing computing resource for the given problem size. In `Sweeps`, compared with problem size, computing resource is always scarce and thus we assume (3) usually holds. In the following analysis, we further assume that there keeps enough parallelism over the whole lifetime of critical path[3].

Now, we take the overhead of DAG updates and MPI communication into account, and do a further analysis under the assumption of load balance of both inter- and intra-node. For `Sweep-H`, the execution time is decided by the maximum running time of the master and workers, which is described by (4). In (4), we assume the DAG is partitioned by vertex and mapped evenly to machine nodes, which is also the fact in our practice. Besides, we treat the communication overhead, including time of both polling and actual message processing, as being proportional to the number of subgraphs (equal to the number of processes, $P$) with unit cost $O_c$.

$$\begin{aligned} T_{\text{Sweep-H}} &= \max\{T_{\text{worker}}, T_{\text{master}}\} \\ &= \max\left\{\frac{U \times V}{(C-1)\times P}, \frac{O_g}{P} + P \times O_c\right\}. \end{aligned} \tag{4}$$

For `Sweep-MPI`, the expected execution time can be estimated by (5) under the same assumptions with (4).

$$\begin{aligned} T_{\text{Sweep-MPI}} &= (T_{\text{computation}} + T_{\text{DAG}}) + T_{\text{communication}} \\ &= \frac{U \times V + O_g}{C \times P} + C \times P \times O_c. \end{aligned} \tag{5}$$

As shown in (4) and (5), the difference between `Sweep-H` and `Sweep-MPI` is as follows. In `Sweep-H` the master performs DAG operations and communication while workers do numerical computation. In `Sweep-MPI`, however, one process alternates to do all work.

Ideally, the optimal value of $T_{\text{Sweep-H}}$ is $T_{\text{worker}}$, and in this situation all physical cores running workers are fully utilized. Also, for given system scale and problem size, $T_{\text{worker}}$ is a constant value and thus the destination of a perfect `Sweep-H` design. Therefore, the sufficient and necessary condition on which `Sweep-H` gains optimal performance is that $T_{\text{worker}} \geqslant T_{\text{master}}$ or equivalently that the worker rather than the master decides the critical path.

If `Sweep-H` has been optimal (i.e., $T_{\text{Sweep-H}} = T_{\text{worker}}$), in order to beat `Sweep-MPI`, $T_{\text{worker}} < T_{\text{computation}} + T_{\text{DAG}} + T_{\text{communication}}$ is necessary. Compared with `Sweep-MPI`, `Sweep-H` allocates a specific core for each process to run the master thread, which means the number of cores doing computation is reduced. In other words, `Sweep-H` pursuits more efficiency in communication and DAG schedule at the cost of reduced resources on vertex computation.

As `Sweep-H` adopts master-workers mode to decouple the computation from DAG-related control, the optimization of $T_{\text{master}}$ and $T_{\text{worker}}$ are independent. The optimization to vertex computation (the variable $U$) is specific to applications, and is the duty of users. Here, we presume that the computation time $U$ has been optimal and treat it as being constant. Equation (4) gives

[3]Strictly speaking, in the beginning and end of a `Sweeps` iteration, some processors have no ready vertices to compute.

some implications to tune performance in Sweep-H, as follows.

• If $T_{\text{worker}}$ is dominant, then actually we have gotten the optimal result. In this situation, if the performance cannot meet the needs, the only way is to increase computing resources (say, number of nodes), which is always effective until $T_{\text{worker}}$ is no longer dominant.

• If $T_{\text{master}}$ is dominant, we have to tune Sweep-H in two aspects. First, we can reduce $O_g$ and $O_c$ in order to make $T_{\text{master}}$ lower than or at least close to $T_{\text{worker}}$, which is always effective. Second, in the sense of mathematics, given fixed $O_g$ and $O_c$, $T_{\text{master}}$ is a $P$'s non-monotonic function which reaches its lower bound $2\sqrt{O_g \times O_c}$ at $P = \sqrt{O_g \times O_c}$. Thus, if $T_{\text{worker}} < 2\sqrt{O_g \times O_c}$, we increase $P$; and else, we should decrease $P$ until $T_{\text{master}} = 2\sqrt{O_g * O_c}$ or $T_{\text{master}} = T_{\text{worker}}$.

• As long as the inequality $T_{\text{worker}} \geqslant T_{\text{master}}$ holds, both weak and strong scalability of Sweep-H are theoretically linear, which is also demonstrated in our experiments. For the situation that $T_{\text{Sweep-H}}$ is dominated by $T_{\text{master}}$, according to (4), the weak scalability (over increasing problem sizes, or $U \times V$) is linear, while the strong scalability would cross an inflection point.

## 5 Experimental Evaluation

In this section, we experimentally evaluate the performance and scalability of Sweep-H, with a comparison to Sweep-MPI. Further, we give a detailed analysis based on profiling data, and conclude why Sweep-H outperforms Sweep-MPI on scalability.

### 5.1 Experimental Setup

Our multicore cluster consists of 8 cabinets with 10 nodes per cabinet. All 10 nodes in a cabinet are fully connected with infiniband network, and each pair of cabinets are connected by one infiniband channel. In our experiments, we use at most 64 nodes of the cluster. The configuration of one single node is shown in Table 2.

**Table 2.** Experimental Platform Configuration

| Node | SMP |
|---|---|
| Number of processors | 2 |
| Memory size | 24 GB |
| Processor | Intel Xeon X5650 |
| Number of cores | 6 |
| Frequency | 2.67 GHz |
| L1 cache size | 384 K |
| L2 cache size | 1 536 K |
| L3 cache size | 12 M |
| Memory type | DDR3-1333 |
| QPI speed | 6.4 GB/s |
| Interconnection | Infiniband |
| Rate | 40 GB/s |

The operating system on each node is CentOS 5.5 Linux. Besides, all reference codes are implemented in C, compiled by GCC 4.1.2 with default options, and linked with libraries of Pthread and OpenMPI 1.5.1.

Without loss of generality, we consider an example adapted from a realistic application for single-group radiation transport equation with stencil $S_4$ (12 directions) on the rectangular Cartesian grid. The grid consists of $X \times Y$ zones, equivalent to $12 \times X \times Y$ vertices of DAG. The experimental grid is nonconforming and can simulate the irregular data dependency in general unstructured grids. As in Mo[7], we use a horizontal stripe decomposition for the super underlying graph, which actually behaves better than any other block decomposition policies in our experiments. In both Sweep-MPI and Sweep-H, Sweeps from all angles are carried out concurrently. Besides, we set the granularity of computation on any vertex as a fixed value (denoted as $U$). For convenience, we use a triple $(X, Y, U)$ to denote problem size in this section.

### 5.2 Performance and Scalability

In all charts of this subsection, the default horizontal axis is the number of nodes ($P$). One node means 12 processes for Sweep-MPI and 12 threads (1 master plus 11 workers) for Sweep-H. Also, for simplicity of comparison, we use problem size ($X = 1\,\text{K}$, $Y = 8\,\text{K}$, $U = 10\,\text{K}$) as the baseline, and the performance is represented as $1\,000/T$, where $T$ is the execution time in seconds.

Fig.8 shows the performance comparison of Sweep-H and Sweep-MPI on different problem sizes. As shown in Fig.8(a), in small system scale (less than 16 nodes), Sweep-MPI performs better. However, with the number of nodes increasing, Sweep-H catches up and finally outperforms with considerable advantage. We then increase the problem size by doubling mesh size, and as shown in Fig.8(b), the result is similar except that the performance of Sweep-MPI has an obvious drop on 32 nodes. Again, we continue to increase the problem size by doubling the computation unit, and the result in Fig.8(c) shows almost the same performance trend with Fig.8(b), for both Sweep-H and Sweep-MPI.

Obviously, advantage of Sweep-H over Sweep-MPI is due to its better scalability, which is demonstrated in Fig.9. In small system scale ($1 \sim 8$ nodes), both are improved with nearly linear speedup. However, when the system scales to 16 nodes and up, while Sweep-H still maintains linear speedup, speedup of Sweep-MPI becomes smooth.

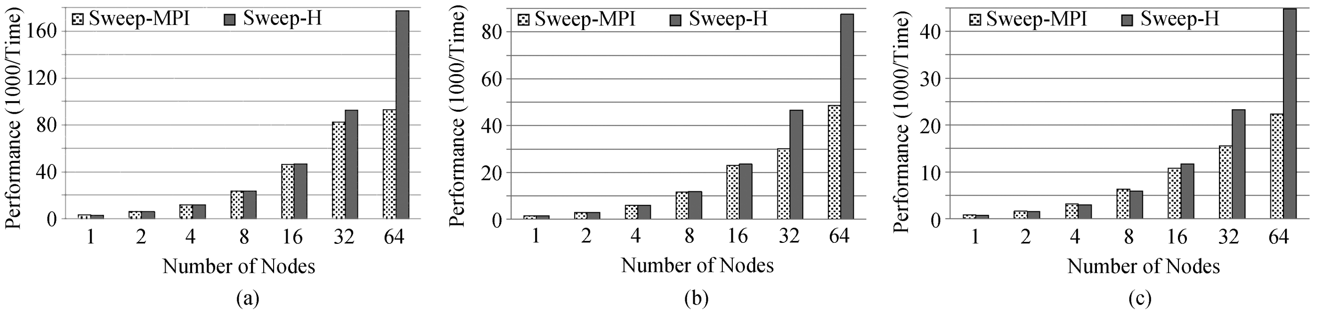


Fig.8. Performance comparison of Sweep-H and Sweep-MPI. Problem size $(X, Y, U)$ means the mesh size is $X \times Y$, and vertex computation grain is $U$. Performance is represented by the reciprocal of execution time, multiplying 1 000. Higher is better. (a) Problem size (1 K, 8 K, 10 K). (b) Problem size (2 K, 8 K, 10 K). (c) Problem size (2 K, 8 K, 20 K).

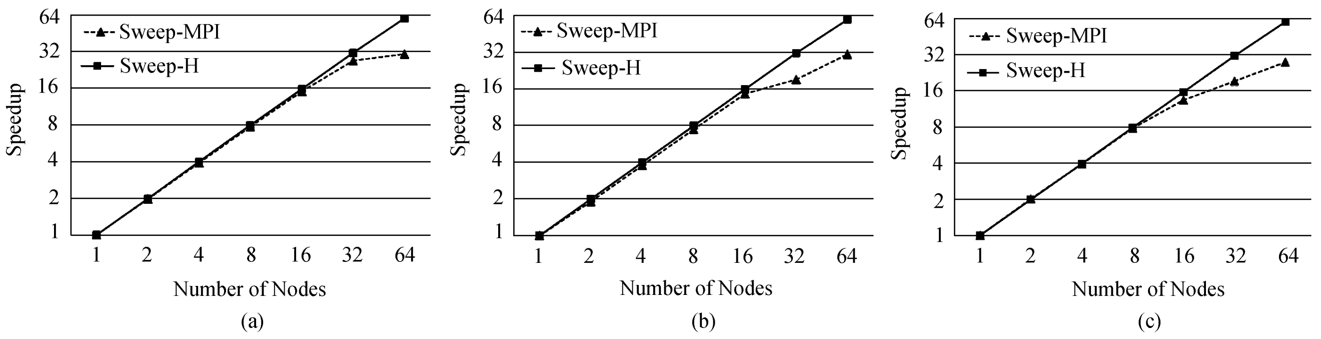


Fig.9. Scalability comparison of Sweep-H and Sweep-MPI. Problem size $(X, Y, U)$ means the mesh size is $X \times Y$, and vertex computation grain is $U$. (a) Problem size (1 K, 8 K, 10 K). (b) Problem size (2 K, 8 K, 10 K). (c) Problem size (1 K, 8 K, 10 K).

In another view, we investigate the computation efficiency (ratio of total vertex computation time to total CPU time). Take the baseline experiment as an example. Efficiency of Sweep-H is 90.2% ~ 85.6%, which is stable and close to its theoretical upper bound (i.e., $11/12 \approx 91.6\%$, as 11 of 12 cores are workers doing numerical computation). As a comparison, Sweep-MPI's actual efficiency ranges from 96.6% to 54.1% with a decreasing trend over incremental cluster nodes, while the theoretical upper bound is 1.

In summary, 1) Sweep-H shows nearly linear scalability over system scale from 1 to 64 nodes, and according to our performance model it can continue to scale until $T_{\text{worker}}$ is no longer dominant; 2) under the same graph partitioning method and priority strategy, Sweep-H outperforms Sweep-MPI when the system scale is moderate or larger.

Additionally, we observe that in our experiments, while the performance of Sweep-H is very stable between different tests of the same problem size and between tests of different problem sizes, Sweep-MPI has obvious fluctuation even for the same problem size especially when the number of processes is large (e.g., no less than 16 × 12). Actually, the performance results of Sweep-MPI shown in the charts of this paper are the best values of all runs.

### 5.3 Detailed Profiling and Analysis

We take the baseline case of ($X = 1\,\text{K}, Y = 8\,\text{K}, U = 10\,\text{K}$) to do profiling in details and analyze the reasons why Sweep-H has better scalability than Sweep-MPI. Note that data given in the charts of this subsection are the average values of all involved processes or threads.

First of all, we take overheads in real implementations into consideration, and rewrite performance formulas of Sweep-H and Sweep-MPI. For Sweep-H, $T_{\text{Sweep-H}}$ is dominated by $T_{\text{worker}}$ in practice, so we rewrite (4) as (6), where $T_{\text{computation}}$ is the time of numerical computation on vertices, $T_{\text{schedule}}$ is the overhead of operating *ReadyQueue*, *CommitQueue* and *PriorityQueue*, and $T_{\text{starvation}}$ is the average idle time that a worker has no ready vertex to compute. Similarly, rewrite (5) as (7), where $T_{\text{computation}}$ is the time of numerical computation, $T_{\text{communication}}$ is the time of all communication related overheads including request polling ($T_{\text{check}}$) and real processing ($T_{\text{mpi}}$), and $T_{\text{other}}$ is time spent on operating DAG and scheduling ready vertices.

$$\begin{aligned} T_{\text{Sweep-H}} &= T_{\text{worker}} \\ &= T_{\text{computation}} + T_{\text{schedule}} + T_{\text{starvation}}, \end{aligned} \quad (6)$$

$$T_{\text{Sweep-MPI}} = T_{\text{computation}} + T_{\text{communication}} + T_{\text{other}}. \quad (7)$$

We dissect (6) and (7) to analyze factors that affect the scalability of `Sweep-H` and `Sweep-MPI`. Obviously, increasing computing nodes means proportionally decreasing subgraph size, and thus good scalability means executing time decreases in the same proportion. In (6), $T_{\text{computation}}$ and $T_{\text{schedule}}$ are proportional to subgraph size, and thus $T_{\text{starvation}}$ decides the scalability of `Sweep-H`. In (7), $T_{\text{computation}}$ is exactly and $T_{\text{other}}$ is approximately proportional to subgraph size, so the scalability of `Sweep-MPI` is determined by $T_{\text{communication}}$. Therefore, in order to get linear scalability, $T_{\text{starvation}}$ for `Sweep-H` and $T_{\text{communication}}$ for `Sweep-MPI` should be either decreased proportionally over increasing nodes or kept in a very low level.

Now, we analyze the detailed profiling data for `Sweep-H` and `Sweep-MPI`. Fig.10 shows the percentage distribution of different parts of execution time defined in (6) and (7). `Sweep-H`, as shown in Fig.10(a), unsurprisingly keeps a very low percentage of $T_{\text{starvation}}$, which exactly explains why `Sweep-H` is linearly scalable in our experiments. In contrast, as shown in Fig.10(b), `Sweep-MPI` has an ever-increasing percentage of $T_{\text{communication}}$ (aggregate of $T_{Comm1}$ and $T_{Comm2}$, being explained later). Scrutiny to Fig.10(b) reveals that scalability of `Sweep-MPI` drops exactly when percentage of communication-related overhead becomes significant. For example, on the system scale of 64 nodes, communication overhead takes more than 40% of run time, which makes performance of `Sweep-MPI` as lower as almost half of `Sweep-H`.

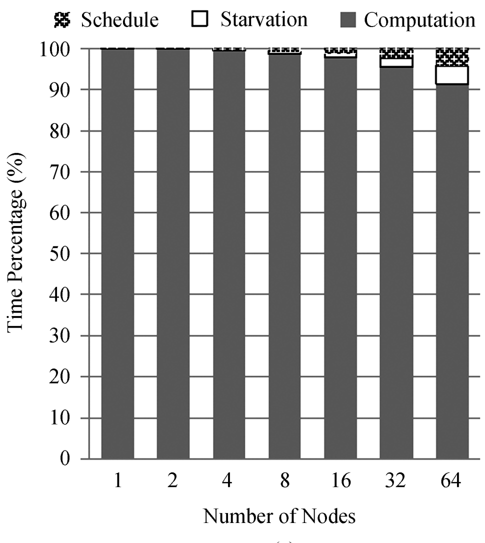
We investigate $T_{\text{communication}}$ to analyze the reasons of its dramatic rise. By triggering sources, communication-related overhead can be divided into two parts, say $T_{Comm1}$ and $T_{Comm2}$. In the real implementation of `Sweep-MPI`, computation and communication are overlapped. Between successive computation of two ready vertices, `Sweep-MPI` routinely uses the time slot to check and do communication, which costs $T_{Comm1}$. Besides, when the sweeping procedure is not over but there is no local ready vertex, i.e., the process is trapped in starvation, `Sweep-MPI` just repeats to check the coming messages and flush all buffered sending messages, which costs $T_{Comm2}$. Thus, $T_{Comm2}$ includes the effect of starvation.

Table 3 gives the data of $T_{Comm1}$ and $T_{Comm2}$ on different number of nodes, as well as real communication time $T_{\text{mpi}}$. As shown, $T_{Comm1}$ is nearly constant over any number of nodes, while $T_{Comm2}$ is variable but keeps in a high level. Both of them are significant and do not decrease with increasing nodes, which limits the

**Table 3.** Communication Time (s) of `Sweep-MPI` for Mesh 1 K × 8 K and Computation Grain 10 K

| Number of Nodes | $T_{Comm1}$ | $T_{Comm2}$ | $T_{\text{mpi}}$ |
|---|---|---|---|
| 1 | 0.580 33 | 6.157 36 | 0.018 58 |
| 2 | 0.519 94 | 4.000 38 | 0.016 16 |
| 4 | 0.491 77 | 4.705 03 | 0.014 96 |
| 8 | 0.482 72 | 2.775 85 | 0.014 55 |
| 16 | 0.489 39 | 1.444 10 | 0.014 08 |
| 32 | 0.521 47 | 1.532 31 | 0.013 89 |
| 64 | 0.537 29 | 3.590 72 | 0.014 83 |

Note: $T_{\text{communication}}$ consists of $T_{Comm1}$ and $T_{Comm2}$, with $T_{\text{mpi}}$ included.



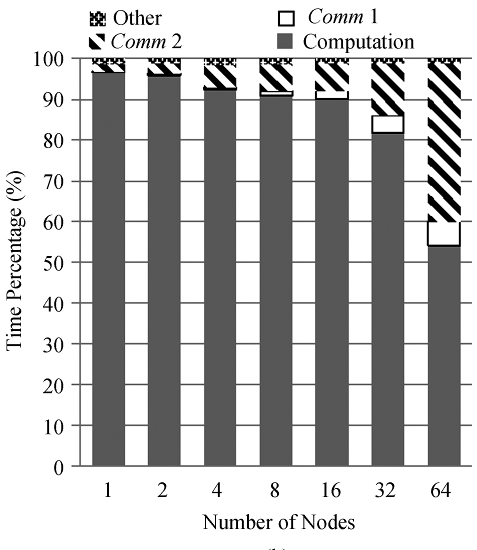


Fig.10. (a) `Sweep-H`: worker's execution time distribution. (b) `Sweep-MPI`: execution time distribution, where communication overhead is divided into *Comm*1 and *Comm*2. The profiling results were collected on mesh 1 K × 8 K and computation unit 10 K.

scalability or speedup over system scale. Besides, from the fact that $T_{Comm2}$ is far higher than $T_{Comm1}$, we can see that the overhead introduced by starvation is more serious than routine communication.

Table 3 also reveals that $T_{\rm mpi}$ is trivial and approximately constant, which indicates that the time of communication check rather than data transfer is dominant in both $T_{Comm1}$ and $T_{Comm2}$. In fact, in order to realize instant communication and further avoid remote starvation, `Sweep-MPI` adopts asynchronous MPI primitives and checks the need of communication as frequent as possible, which results in significant $T_{Comm1}$ and however still could not get expected load balance (low $T_{Comm2}$).

Obviously, the unnatural starvation is due to the latency of message delivery from upwind vertices to downwind vertices. We here experimentally demonstrate that this latency is brought in primarily by MPI itself rather than network latency. In our experiments, we measure the performance of `Sweep-MPI` on two configurations with the same total number of processes. The first configuration runs 12 processes per node while the second runs one process per node, which means the latter uses more network in data transferring. Results in Fig.11 show that for our experimental grids the network has no negative impact on performance.

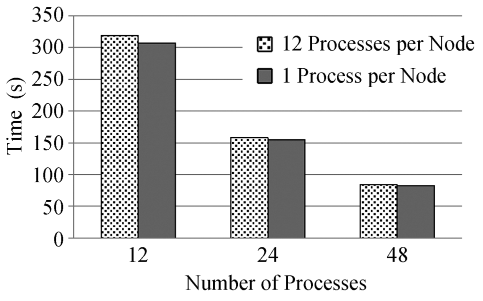


Fig.11. Effect of network to `Sweep-MPI` on mesh 1 K × 8 K and computation grain 10 K.

So far, we can conclude that 1) `Sweep-MPI`'s scalability is limited by communication overhead and load imbalance, and 2) the communication overhead is due to the MPI's own inefficiency to support instant delivery of massive short messages.

As a comparison, `Sweep-H` is successful in overcoming the above problems of `Sweep-MPI`.

- `Sweep-H` significantly reduces communication costs. Table 4 is the profiling data of communication time for `Sweep-H`. $T_{\rm mpi}$ is the time of MPI functions, which can approximately represent the cost of real communication. In `Sweep-H`, only the core running master thread is involved in communication, thus the amortized cost of each core is $T_{\rm mpi}/C$ ($C$ is number of cores per node, here it is 12). Comparison between Table 3 and Table 4 shows that $T_{\rm mpi}/12$ in `Sweep-H` is less than $T_{\rm mpi}$ in `Sweep-MPI` by at least one order of magnitude. (Note that because of the different policies of communication check, $T_{\rm communication}$ in Table 4 is not comparable with $T_{Comm1}$ or $T_{Comm2}$ in Table 3.)

**Table 4.** Communication Time (s) of Master in `Sweep-H` for Mesh 1 K × 8 K and Computation Grain 10 K

| Nodes | $T_{\rm communication}$ | $T_{\rm mpi}$ |
|---|---|---|
| 1 | 0.000 24 | 0.000 00 |
| 2 | 0.004 92 | 0.004 42 |
| 4 | 0.040 27 | 0.007 93 |
| 8 | 0.057 85 | 0.010 58 |
| 16 | 0.056 11 | 0.012 02 |
| 32 | 0.058 80 | 0.012 35 |
| 64 | 0.056 50 | 0.012 15 |

Note: $T_{\rm communication}$ itself includes $T_{\rm mpi}$.

- `Sweep-H` is less sensitive to communication latency. The data-driven parallel algorithms require instant communication, otherwise if there are no more ready vertices in remote nodes the lagged messages will lead to unnecessary starvation. In `Sweep-H`, the size of one process's subgraph is $C$ times as big as that in `Sweep-MPI`. The bigger subgraph means more native runtime parallelism, which can tolerate more communication latency. This fact is indirectly confirmed by the profiling results shown in Table 3 and Table 4 where $T_{\rm mpi}$ in `Sweep-H` is always less than that in `Sweep-MPI`. In our implementations of both `Sweep-MPI` and `Sweep-H`, once a process has no ready vertices, it flushes sending message buffers and checks coming messages repeatedly. Thus higher $T_{\rm mpi}$ usually means more communication of short messages.

- `Sweep-H` has nearly optimal load balance, as illustrated by the low percentage of worker's starvation time in Fig.10(a). This verifies our initial design principle that load balance among shared memory workers is kept by Load Balance principle while load balance among nodes is maintained by more local parallelism and better instant communication. Actually, as the number of nodes increases, load balance among workers becomes more important. As shown in Fig.12, compared with the simple RoundRobin policy in which each worker tries to fetch a fixed number of ready vertices from *ReadyQueue*, Load Balance principle has no obvious effect when the number of nodes is less than 32, but on 64 nodes it shows about 39% performance improvement.

## 6 Related Work

As the most important deterministic method solving Boltzmann transport equation, parallel $S_n$ `Sweeps` has been widely investigated. Most of previous algorithmic

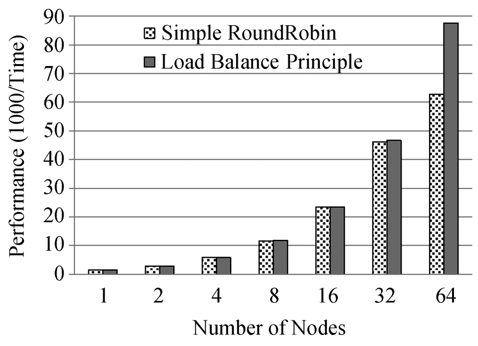


Fig.12. Effect of load balance policies of `Sweep-H` on mesh 2 K × 8 K and computation unit 10 K. Performance is represented by the reciprocal of execution time, multiplying 1 000. Higher is better.

work is for 3-D structured grids, among which the KBA algorithm[2-3] is very successful in parallelizing sweep operations. KBA decomposes 3-D grids in a 2-D columnar fashion and pipelines computation by angles in a way of wavefront, which synchronizes the inter-processor communication. The MPI implementations based on KBA on Cray-T3D and CM-200 in Lawrence Livermore National Laboratory demonstrated good scalability over hundreds to thousands of processors. Later, ASCI Sweep3D benchmark④ and Ardra⑤ further achieved nearly perfect scalability on thousands of processors, with parallel efficiency higher than 90%.

For unstructured grids, however, progress of parallel $S_n$ `Sweeps` is not so optimistic because of the irregular data dependency of computation on cells. Plimpton[6] gave a survey of the earlier work [13-14]. Also, Plimpton *et al.*[5] first presented a graph-based parallel pipeline algorithm of $S_n$ `Sweeps` for 3-D unstructured grids. In the algorithm, geometric dependencies in the mesh are modeled as a DAG, and the full boltzmann problem is to simultaneously perform $M$ sweeps on the DAG, where $M$ is the number of ordinate directions. Later, Mo *et al.*[7] in fact gave a general framework and formal model for Plimpton *et al.*'s method. Implementations of this method on hundreds to one thousand of processor cores[12] show moderate scalability of parallel efficiency.

Most known implementations of above algorithms are MPI-based and originally designed for distributed memory system. Recently, some efforts (e.g., [15-17]) have made to leverage the Cell or GPU architectures to accelerate the Sweep3D. Particularly, some of them[17] explored the potential benefits of accelerating the wavefront computation on multi-GPU clusters.

However, multicore clusters have been the mainstream of today's HPC systems. Previous pure MPI-based algorithms fail to take advantage of multithreading on shared memory within the multicore node. For structured grid, the MPI-based algorithms are efficient enough, while for unstructured girds the existed methods still suffer from problems as stated in this paper. To the best of our knowledge, our work is the first trial to leverage the intra-node multicore architecture to realize problems in MPI implementations of Plimpton and Mo *et al.*'s data-driven algorithm.

## 7 Discussion and Conclusions

### 7.1 Apply `Sweep-H` in Real-World

The focus of this paper is the `Sweep-H` method itself as well as its comparison to `Sweep-MPI`. In order to identify behaviors of the `Sweep` methods, we made two key simplifications to the real-world cases, such that 1) the schedule unit is a cell (vertex), and 2) the computation grain of any cell (vertex) is fixed. Now we discuss the above two issues.

First, in today's particle transportation simulations, generally the computation and schedule unit is often a patch (i.e., a cluster of adjacent cells) rather than a single cell. On one hand, for most applications computation on a single cell is too little to amortize the cost of scheduling it. On the other hand, clustering a group of neighboring cells into a patch can significantly improve the cache locality and partly eliminate the NUMA effect in machines of multi-socket processors.

`Sweep-H` and `Sweep-MPI` can easily handle patches by treating and scheduling one patch as a "super-vertex" consisting of multiple vertices/cells, while with patches the graph partitioning phase needs much extra work to handle the data dependency between super-vertices.

Second, grain of vertex computation would vary for different cells in the same time step or the same cell in different time steps. This is because particles in cells are moving during different time steps and thus the real data dependency between cells is evolving. The direct effect of varied grain of vertex computation is load balance. Compared with `Sweep-MPI`, `Swep-H` has demonstrated better load balance within machine node.

### 7.2 Conclusions

As the kernel of radiation transport simulation, `Sweeps` is critical to the overall performance. In this paper, we proposed `Sweep-H`, a new parallel data-driven

④Los Alamos National Laboratory. The ASCI Sweep3D Benchmark. http://wwwc3.lanl.gov/pal/software/sweep3d, Jan. 2013.

⑤Lawrence Livermore National Laboratory. Ardra: Scalable parallel code system to perform neutron and radiation transport calculations. http://www.llnl.gov/casc/Ardra/, Jan. 2013.

algorithm of Sweeps on unstructured grids. It adopts a hybrid parallel model of both MPI and Pthreads, taking advantage of hybrid memory model and multi-threading on contemporary multicore clusters. Compared with previous Sweep-MPI, Sweep-H improves communication efficiency on message volume and delivery latency, as well as load balance among processes. Our performance model and experiments on up to 64 nodes (768 cores) both demonstrated the nearly linear scalability of Sweep-H under practical problem parameters.

Results of Sweep-H are promising. In the future, we shall deploy and investigate Sweep-H in real situations, with more practical 3D grids on larger system scales. Besides, ideas and technologies we developed in Sweep-H can be generalized to other data-driven parallel applications.

## References

[1] Downar T, Siegel A, Unal C. Science based nuclear energy systems enabled by advanced modeling and simulation at the extreme scale. Report of Workshop on Nuclear Energy, May 2009, http://science.energy.gov/∼/media/ascr/pdf/program-documents/docs/Sc_nework_shop_report.pdf.

[2] Baker R S, Alcouffe R E. Parallel 3-D Sn performance for MPI on cray-T3D. In *Proc. Joint Int. Conf. Math. Methods and Supercomputing for Nuclear Applicat.*, Oct. 1997, pp.377-393.

[3] Baker R S, Koch K R. An Sn algorithm for the massively parallel CM-200 computer. *Nuclear Science and Engineering*, 1998, 28: 312-320.

[4] Valiant L G. A bridging model for parallel computation. *Communications of the ACM*, 1990, 33(8): 103-111.

[5] Plimpton S, Hendrickson B, Burns S *et al.* Parallel algorithms for radiation transport on unstructured grids. In *Proc. ACM/IEEE Conf. Super Computing*, Nov. 2000, Article No.25.

[6] Plimpton S, Hendrickson B, Burns S *et al.* Parallel $S_n$ sweeps on unstructured grids: Algorithms for prioritization, grid partitioning, and cycle detection. *J. American Nuclear Science and Engineering*, 2005, 150(3): 267-283.

[7] Mo Z Y, Zhang A Q, Cao X L. Towards a parallel framework of grid-based numerical algorithms on DAGs. In *Proc. the 20th IPDPS*, Apr. 2006, p.310.

[8] Hewitt C, Bishop P, Steiger R. A universal modular actor formalism for artificial intelligence. In *Proc. the 3rd IJCAI*, Aug. 1973, pp.235-245.

[9] Schloegel K, Karypis G, Kumar V. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 2002, 14(3): 219-240.

[10] Karypis G, Kumar V. Multi-level graph partitioning schemes. In *Proc. ICPP*, Aug. 1995, pp.113-122.

[11] Hendrickson B, Leland R. A multilevel algorithm for partitioning graph. In *Proc. ACM/IEEE Conf. Super Computing*, Dec. 1995, Article No.28.

[12] Zhang A Q. Research on scalable parallel data driven algorithms and applications [Ph.D. Thesis]. China Academy of Engineering Physics, 2009.

[13] Pautz S D. An algorithm for parallel $S_n$ sweeps on unstructured meshes. *Nuclear Science and Engineering*, 2002, 140(2): 111-136.

[14] Nowak P, Nemanic M K. Radiation transport calculations on unstructured grids using a spatially decomposed and threaded algorithm. In *Proc. Int. Conf. Mathematics and Computation, Reactor Physics and Environmental Analysis in Nuclear Applications*, Sept. 1999, pp.379-390.

[15] Gong C Y, Liu J, Chi L H, Huang H W, Fang J Y, Gong Z H. GPU accelerated simulations of 3D deterministic particle transport using discrete ordinates method. *Journal of Computational Physics*, 2011, 230(15): 6010-6022.

[16] Lubeck O, Lang M, Srinivasan R, Johnson G. Implementation and performance modeling of deterministic particle transport (Sweep3D) on the IBM Cell/BE. *Scientific Programming*, 2009, 17(1/2): 199-208.

[17] Pennycook S J, Hammond S D, Mudalige G R, Wright S A, Jarvis S A. On the acceleration of wavefront applications using distributed many-core architectures. *The Computer Journal*, 2012, 55(2): 138-153.

**Jie Yan** is a Ph.D. candidate of Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing. Previously, he obtained the M.S. degree from University of Science and Technology of China, Hefei, in 2010 and B.S degree from Beijing University of Posts and Telecommunications in 2007, both in computer science. His current research interests focus on parallel algorithms and computational model for large-scale graph analysis.

**Guang-Ming Tan** received the Ph.D. degree in computer science from CAS. He is an associate professor in the State Key Laboratory of Computer System and Architecture, ICT, CAS, Beijing. From 2006 to 2007, he was a visiting researcher in the Computer Architecture and Parallel Systems Laboratory, University of Delaware, USA. His research interests include parallel algorithm and programming, performance modeling and evaluation, and computer architecture. He is a member of CCF, ACM, and IEEE.

**Ning-Hui Sun** received his B.S. degree from Peking University in 1989 and M.S. and Ph.D. degrees both in computer science from the CAS in 1992 and 1999, respectively. He is a professor in ICT, CAS. He is the architect and main designer of the Dawning series high performance computers, from Dawning2000 to Dawning Nebulae. His research interests include computer architecture, operating system, and parallel algorithm. He is a fellow of CCF and member of ACM and IEEE.