

JSweep: A Patch-centric Data-driven Approach for Parallel Sweeps on Large-scale Meshes

Jie Yan^{*} Zhang Yang^{†‡} Aiqing Zhang^{†‡} Zeyao Mo^{†‡}

[†] Software Center for High Performance Numerical Simulation, CAEP

[‡] Institute of Applied Physics and Computational Mathematics, Beijing, China

^{*} Noah's Ark Lab, Huawei Technologies

Correspondence: {yang_zhang, zhang_aiqing, mo_zeyao}@iapcm.ac.cn

Abstract—In mesh-based numerical simulations, sweep is an important computation pattern. During sweeping a mesh, computations on cells are strictly ordered by data dependencies in given directions. Due to such a serial order, parallelizing sweep is challenging, especially for unstructured and deforming structured meshes. Meanwhile, recent high-fidelity multi-physics simulations of particle transport, including nuclear reactor and inertial confinement fusion, require *sweeps* on large scale meshes with billions of cells and hundreds of directions. In this paper, we present JSweep, a parallel data-driven computational framework integrated in the JAxMIN infrastructure. The essential of JSweep is a general patch-centric data-driven abstraction, coupled with a high performance runtime system leveraging hybrid parallelism of MPI+threads and achieving dynamic communication on contemporary multi-core clusters. Built on JSweep, we implement a representative data-driven algorithm, Sn transport, featuring optimizations of vertex clustering, multi-level priority strategy and patch-angle parallelism. Experimental evaluation with two real-world applications on structured and unstructured meshes respectively, demonstrates that JSweep can scale to tens of thousands of processor cores with reasonable parallel efficiency.

I. INTRODUCTION

In mesh-based numerical simulations, *sweep* is an important computation pattern widely used in solving deterministic Boltzmann Transport Equation (BTE) [4], convection dominated or Navier-Stokes equation [5] [6] and so on. During *sweep* on a mesh, cells are computed from upwind to downwind in the sweeping direction. One cell can compute only when all of its upwind neighboring cells are computed.

General *sweep* computation on large-scale meshes is challenging. For rectangular structured meshes where the data dependencies can regular, the well-known Koch-Baker-Alcouffe(KBA) [8] [9] algorithm which uses a pipelining wavefront way to parallelize multiple sweeps has been very successful. However, for the more general deforming structured meshes and unstructured meshes in which data dependencies among cells are irregular, the KBA method doesn't fit and is typically impossible. Instead, a data-driven approach [15] [16] [20] is often considered. This approach models the cells' data dependencies as a directed acyclic graph (DAG), regardless of the mesh types, then *sweep* on the mesh is equivalent to a topological traversal on the DAG. Unfortunately, although KBA-based sweep on regular structured meshes has scaled to 10^6 CPU cores and billions

of cells in 2015 [22], sweep on ordinary unstructured meshes still doesn't efficiently scale to 10^5 cores.

Meanwhile, mesh-based application programming frameworks [1] [2] [28] [29] have been increasingly critical to today's complex simulations that require to couple multiple multi-physics procedures. On one hand, multiple simulation procedures developed on a unified framework share the same specification of mesh and data layout, and thus are more consistent to work together. Given the fact that coupling different simulation procedures is difficult for both software development and numerical validation, this really makes sense. On the other hand, by providing users a programming abstraction and ensuring reasonable performance, the framework isolates applications from the evolution of underlying HPC systems, and thus achieves good portability. Recently, in areas related to particle transport in which sweep on mesh is the most time-consuming portion, framework-based coupling of multi-physics simulations have led to several successful cases, including the full core reactor simulation based on MOOSE [35] and the ICF (Inertia Confinement Fusion) program LARED-I [34] based on JASMIN [1]. Nevertheless, so far these cases are still on structured meshes only.

In this paper, we focus on the patch-based mesh application framework, specifically JAxMIN [1] [2] (detailed in section II-B), where the mesh is divided into patches. Patch is conceptually a subdomain of the mesh. The patch-based approach has advantages on adaptive mesh refinement, mesh management and parallel computation scheduling. Over 50 real-world applications have been implemented on JAxMIN. However, JAxMIN, like most counterparts, adopts BSP (Bulk Synchronous Parallel) [13] style of parallel computing, in which all subdomains (patches) first compute using previous data of themselves and other subdomains, and then communicate to update their remote copies. Although being efficient and scalable enough for most numerical solvers, BSP is seriously inefficient for data-driven sweep computation where the parallelism is fine-grained. Furthermore, the fact that a patch often can't finish computation at one time and thus has to compute many times, as well as complex factors impacting performance, makes it hard to realize in JAxMIN's BSP-based abstraction.

We propose JSweep, a patch-centric data-driven approach for parallel sweep computation on both structured and un-

structured meshes, embedded in the JAxMIN infrastructure. Fig.1 illustrates the JSweep modules in the abstracted layers of JAxMIN. Specifically, our key contributions are as follows:

- The patch-centric data-driven abstraction, a unified model for general data-driven procedures on both structured and unstructured meshes. The core idea is extending the concept of patch to a logical processing element that supports reentrant computation and communicate with other patches (Sec. III).
- The patch-centric data-driven runtime module for contemporary multi-core cluster systems, featuring hybrid parallelism (MPI+threads) and dynamic data delivery (Sec. IV).
- A sweep component based on the above patch-centric approach, enhanced by vertex clustering, multi-level priority strategy, patch-angle parallelism and coarsened graph. (Sec. V).
- Experimental evaluation with real applications of particle transport on both structured and unstructured meshes, demonstrating JSweep’s reasonable performance and scalability on up to 76,800 processor cores (Sec. VI).

Besides, we present background and motivation in Sec. II, related work in Sec. VII, and finally conclusions in Sec. VIII.

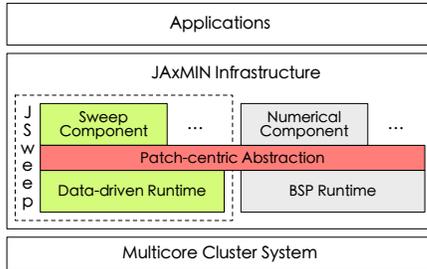


Fig. 1: JSweep Framework Overview

II. BACKGROUND AND MOTIVATION

A. Preliminaries

Throughout this paper, we use a small set of terminologies as illustrated in Fig. 2. Note that we describe them in an abstract view and don’t explicitly differentiate the structured and unstructured meshes unless otherwise stated.

- `mesh/grid`: the generic way of describing the discretized domain.
- `cell`: the smallest unit of a discretized domain.
- `patch`: a collection of contiguous cells.
- `local cells`: cells owned by a patch. They are updated by an operator applied to the patch.
- `ghost cells`: halo cells surrounding local cells. They are needed for computation but not updated by the local operator. They belong to other patches.

B. JAxMIN: Patch-based Mesh Application Infrastructure

JASMIN [1] and JAUMIN [2] (denoted as JAxMIN¹ for simplicity), are software infrastructures for programming ap-

¹Abbreviation of J Adaptive Structured/Unstructured Mesh Infrastructure.

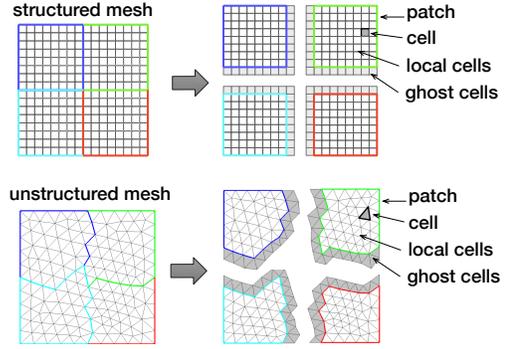


Fig. 2: Illustration of mesh terminologies.

lications of large-scale numerical simulations on structured and unstructured meshes. They share the same design principles, i.e., patch-based mesh management and component-based application programming interface. Although they are different in the way of describing mesh geometry and mesh elements, we omit these details by discussing in an abstract level in this paper.

JAxMIN adopts a patch-based strategy to manage the mesh and data. The computational domain, discretized as mesh, is decomposed into patches. *Patch* is a well-defined subdomain, that (1) each patch has complete information about its own cells as well as other mesh entities, (2) with ghost cells, each patch can explicitly get all adjacency information about its neighboring patches, and (3) it is abstract enough to hide differences of structured and unstructured meshes.

JAxMIN provides users a rich set of components as the programming interface. *Component* here is a generic implementation of any computational pattern. To implement a parallel program, users only need to instantiate a component by defining the application-specific computation kernel. So far, general patterns such as initialization, numerical computation, reduction, load rebalance, particle-in-cell communication, are provided. Besides, JAxMIN implements amounts of physics-specific integration components.

Traditionally, JAxMIN adopts the BSP model to organize computations in a component. The computations consist of a sequence of iterations, called super-steps. During a super-step, each patch executes logically in parallel, as follows: (1) does computation independently without data exchange with others, and then (2) does halo exchange communication with neighbors using newly computed data. Since many numerical algorithms fit well in BSP, the patch-based framework has been successful in many areas.

JAxMIN implements a high performance runtime system supporting hybrid MPI+threads parallelism and accelerators, with underlying optimization on memory management, data layout and buffering communication.

C. Data-driven Parallel Sweeps

Without loss of generality, we consider the sweep computation in discrete ordinates (S_n) transport solvers. Sweep is

the most computationally intensive portion of source iterative methods solving S_n form of Boltzmann Transport Equation [4]. As the name implies, *sweep* in any ordinate direction requires a computational ordering of cells from upwind to downwind. One cell can begin computing only if all of its upwind neighboring cells are computed.

Parallelizing sweep computation is challenging since it can't be efficiently implemented in a BSP manner. For regular structured meshes, the KBA approach, decomposing 3d meshes in a 2d columnar fashion and pipelining the computation for successive angles, is sufficient with BSP. However, for more general deforming structured meshes and unstructured meshes where data dependencies among cells are irregular and thus the pipeline can't be easily determined, the KBA approach is almost impossible to implement.

Alternatively, we focus on the data-driven parallelization which is a general approach for *sweeps* on both structured and unstructured meshes [15] [16]. In this approach, any complex and irregular data dependencies can be explicitly modeled by a directed acyclic graph. As an example, Fig. 3 illustrates a 2d unstructured mesh and the associative directed graph in a given sweeping direction. Then, the *sweep* on a mesh is equivalent to a topological traversal on the directed graph, generalized with the user-defined numerical computations on the vertex.

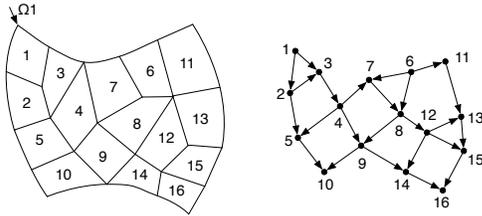


Fig. 3: Illustration of sweeping an unstructured mesh from one direction and the induced data dependency graph [15]

In reality, multiple *sweeps* in different ordinate directions (angles) that are carried out simultaneously. We can model their induced data dependencies in a single graph and implicitly leverage parallelism of *sweeps* from all angles.

D. Motivation

Now we consider the data-driven parallel sweep procedure in the context of the patch-based framework, especially JAxMIN. Unlike other numerical algorithms, patch-level data-driven computation can't be naturally supported in BSP, due to difficulties described below. These difficulties motivate us to develop a new patch-centric data-driven abstraction in the next section.

1) *Partial computation*: In data-driven scenarios, partial computation of the patch is essential. As illustrated in Fig. 4 where one mesh is partitioned into two patches, interleaved data dependencies between the patches means that the patch can't be computed as a whole. In reality, the above zig-zag data dependency can be normal in unstructured meshes. Thus, to be reentrant, partial computation of a patch is necessary.

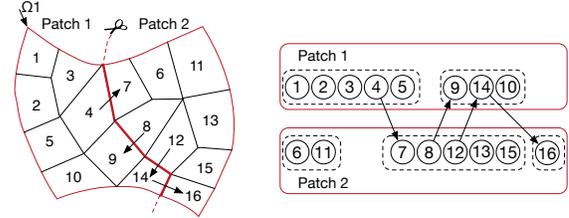


Fig. 4: Illustration of sweeping on a 2d unstructured mesh decomposed into two patches (ghost cells are not shown).

2) *Simultaneous sweeps on a patch*: In real-world applications, sweeps from multiple directions are often performed in parallel. For example, in the S_2 sweeps example illustrated in Fig. 5, one patch would be swept by multiple sweep procedures from 4 different directions. Generally, it is common that some sweeping directions are independent to each other. Thus, to enable such parallelism, simultaneous sweeps on a patch is necessary. In JAxMIN, however, patch is the basic unit of parallel computation, so we need to extend its abstraction.

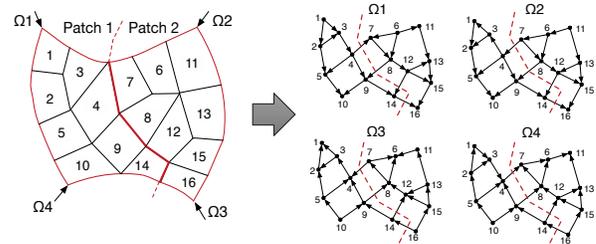


Fig. 5: Parallel sweeps from 4 independent directions(angles)

3) *Priority strategies*: Priority strategies are becoming more important and complex. Previous work [15] [16] have proven that ordering of computing cells (or vertices) is often critical to both parallelism and performance. In their settings, since one process (in MPI) or thread handles only one mesh subdomain, it is sufficient to compute priorities of cells (or vertices in the associated graph). However, in the patch-based framework, one process or thread typically is assigned with arbitrary number of patches, which means patch scheduling is always prioritized than cells within a patch. Thus, we need at least a two-level policy to prioritizing both patches and cells.

III. PATCH-CENTRIC DATA-DRIVEN ABSTRACTION

In this section, we introduce the *patch-centric data-driven abstraction* for mesh-based parallel computations. Its foundation is the completeness and expressivity of the patch concept in JAxMIN described in section II-B. In our abstraction, the concept of patch is further extended as a logical processing element being able to compute on itself and communicate with any other patches. Users should follow a *think-like-a-patch* philosophy to program, and focus on only actions of a single patch, i.e., defining the local computation and inter-patch communication. The abstraction doesn't expose any details of

underlying patch execution details. It should be suitable for all patch-based mesh application frameworks including SAMRAI [28], (part-based) PUMI [29] and especially JAxMIN.

A. Data-driven patch-programs

Data-driven logics on a patch is encoded as a *patch-program*. The patch-program is identified by a $(patch, task)$ pair, indicating *task* is executed on *patch*. Any data communication between two patches is abstracted as a *stream*. The stream contains the user-defined data and description of source and dest patch programs. Fig. 6 presents the interface of patch-program and stream, in which the patch-program is factored into five primitive functions.

```

struct Stream {
  PatchID src_patch; //source patch
  TaskTag src_task; //task on source patch
  PatchID tgt_patch; //target patch
  TaskTag tgt_task; //task on target patch
  ... //user-defined data
};
interface PatchProgram(PatchID p, TaskTag t) {
  void init();
  void input(Stream s);
  void compute();
  Stream output();
  bool vote_to_halt();
};

```

Fig. 6: Patch-program interface

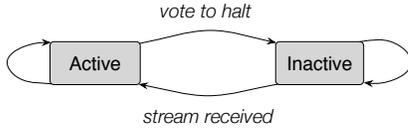


Fig. 7: Patch-program State Machine.

We define patch-program fully reentrant to support partial computation (detailed in the next subsection). At the beginning, each patch-program is set *active*. And, in later execution, the state of a patch-program transits according to the finite state machine given in Fig. 7. If its *vote_to_halt* function is evaluated true, the patch-program becomes *inactive*. Once receiving a *stream*, the patch-program becomes *active*. Conceptually, if there are no active patch-programs globally, the whole program terminates.

The patch-program, identified by $(patch = p, tasktag = t)$, is scheduled to run the semantics in Alg. 1, as follows.

- If runs at the first time, the *init* function is used to initialize a local context.
- Receives all streams sent to (p, t) by others, which is processed by the user-defined *input* function.
- Calls *compute* function with user-defined numerical kernels.
- Sends all *output* streams to and activates targets.

- Calls *vote_to_halt* to evaluate whether there remains ready work to do. If not, deactivates itself. Otherwise, keeps active for being scheduled again.

Algorithm 1: Patch-Program Execution Semantics

```

input: Center patch  $p$ 
input: Task tag  $t$ 
begin
  // Init the state (execute once).
  1 if first time then
  2   | init();
  // Recv data streams.
  3 while Stream  $s = \text{receive}(p, t)$  is not empty do
  4   | input( $s$ );
  // Compute.
  5 compute();
  // Send data streams.
  6 while Stream  $s = \text{output}()$  is not empty do
  7   | activate( $s.tgt\_patch, s.tgt\_task$ );
  8   | send( $s$ );
  // Vote to halt.
  9 if vote_to_halt() then
  10  | deactivate( $p, t$ );

```

1) *Partial computation of patch-program*: Partial computation is an essential property of the patch-program. On one hand, generally a patch-program couldn't finish at one time and thus requires many times of scheduling. As illustrated by the S_n sweeps case (Sec II-D), two patch-programs would depend on data of each other, leading to a dead lock if patch-programs are not reentrant. On the other hand, a patch-program may contain multiple parts of computations that depend on data of different patch-programs, so allowing a patch-program to execute multiple times can benefit from finer grained parallelism.

In our abstraction, *partial computation* of a patch-program is achieved by the following approaches. First, we allow storing of local context so the state are memorized, as illustrated by the implementation of *sweeps* in section V-A. Second, the logics of *finite state machine* in Fig. 7 maintains state transition of a patch-program, ensuring the correctness of termination after arbitrary times of partial execution.

2) *Simultaneous tasks on a patch*: Our abstraction supports multiple tasks on the same patch. Since any patch-program is identified by the pair $(patch, task)$, multiple tasks on a patch naturally execute in parallel, even with possible inter-task communications. Whether and how to decompose work on a patch into patch-programs is the programmers' decision. For the full S_n sweeps discussed in Sec.II-D2, by defining sweep on any patch p from the angle a as a patch-program $(patch = p, task = a)$, sweeps from all directions execute simultaneously.

B. Scheduling patch-programs

The data-driven engine initializes and continues to schedule active patch-programs to run until program termination.

For general patch-centric data-driven computations, the necessary and sufficient condition of program termination is that globally *all* $(patch, task)s$ become inactive. To detect the termination condition in distributed situations, general negotiating protocols [14] are needed. However, in numerical algorithms requiring the data-driven approach, the workload is known in advance. Thus, we can often detect the termination with little or even no distributed negotiation. For example, in *sweeps*, the program termination condition is all $(cell, angle)s$ are computed, which is known by every patch before computation, and termination detection only need local information. In JSweep’s real implementation, we actually allow the patch-program to commit its remained workload (i.e., number of $(cell, angle)s$ in *sweeps*) to a data structure shared by the master and worker threads of local runtime system (detailed in next section). The master thread, as representative of the process, participates distributed terminate negotiation only when there are no longer patch-programs with remained workload.

Priority policies are known critical for scheduling computations, yet is tightly coupled with the properties of the problem itself. In section V-D, we shall discuss several strategies used in parallel S_n *sweeps*.

IV. PATCH-CENTRIC DATA-DRIVEN RUNTIME SYSTEM

In this section, we present the runtime that maps the patch-centric data-driven computation and communication onto underlying resources. Our target platform is the *multicore cluster* widely adopted in contemporary HPC systems. Fig. 8 shows an overview of the runtime system. In particular, our design emphasizes fast stream delivery, load balance, fine-grained parallelism and low schedule overhead.

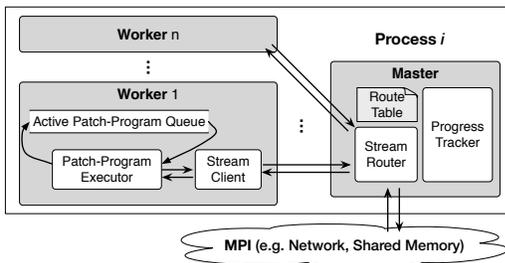


Fig. 8: Data-driven Runtime Overview

A. Hybrid parallelism

The runtime inherits from JAxMIN a hybrid parallel approach of MPI + threads, in which the program is organized with distributed-memory MPI processes and each process consists of multiple threads. On top of this design, JAxMIN have been highly optimized on both domain decomposition and data management. Besides, to reduce NUMA effect in multi-socket systems, by default our runtime launches one MPI process per processor and bind the process to the

processor. Within each MPI process, the master-workers multi-threading mode is adopted. As shown in Fig. 8, the master thread is in charge of scheduling patch-programs, communicating streams and detecting global termination, while each worker thread executes patch-programs and communicates only with master thread.

B. Dynamic stream communication

In the patch-centric abstraction (section III-A), communication conceptually happens between a pair of $(patch, task)s$ and is defined as a *stream*. For data-driven algorithms, the latency of *stream* transmission is critical for performance, since only received the dependent data can a patch-program execute. By definition, *the communication of stream is asynchronous and dynamic*.

The runtime system adopts the routable stream concept and reserves a specific core for master thread to support such timely communication. As defined in section III-A, the *stream* itself carries full information of source and target $(patch, task)s$. By identifying the target patch-program and looking up the route table that maps $(patch, task)$ to $(process, thread)$, the runtime system can deliver any stream to its target place, either locally or in remote process.

The master thread schedules patch-programs by routing streams. At the beginning, all patch-programs are active and assigned to workers evenly. Later as the execution progresses, some patch-programs become inactive. If the master thread receives a stream whose target patch-program is inactive, it chooses and sets a lightest worker as the patch-program’s owner, and then routes it the stream.

C. Distributed progress tracking

The master thread calls the *progress tracker* to detect global program termination. Once temporarily there is no longer work within the process, the progress tracker is activated. A consensus algorithms is implemented to detect distributed termination [14]. Besides the general negotiating protocol, as discussed in section III-B, for known data-driven algorithms special condition detection methods would be preferred for efficiency in practice. Currently, we support both.

V. NEW PARALLEL ALGORITHM OF SN SWEEPS

Now we describe a new parallel sweeps algorithm based on JSweep, the above patch-centric data-driven approach. Further, we explore four optimizations, including scheduling by $(patch, angle)$, vertex clustering, multi-level priority strategy, and coarsened graph, which are natural and efficient to implement thanks to the expressibility of the patch-centric abstraction.

A. Patch-Program Implementation

We assume that the mesh has been decomposed into patches with general spacial domain decomposition methods (for example, the METIS [18] and Chaco [19] for unstructured meshes, Morton and Hilbert space filling curves for structured meshes). Each process is assigned with an arbitrary number of patches, shared by all its threads.

Formally, we define the directed graph induced by sweeping meshes as $G = (V, E)$, where each *vertex* is a $(cell, angle)$ pair, and each *edge* is directed data dependency between two vertices. An edge (u, v) means vertex v depends on vertex u 's data. For any patch p and a sweeping direction t , we denote the induced subgraph as $G_{p,t} = (V_{p,t}, E_{p,t})$, where $V_{p,t}$ is the set of vertices (i.e., $\{(cell, t)\}$) and $E_{p,t}$ is the set of edges.

Listing 1 presents the patch-centric implementation of parallel sweeps. As presented, the patch-program consists of two parts, i.e., local context and interface implementation.

Listing 1: Patch-Program of data-driven parallel sweeps

```

1 //  $G_{p,t} = (V_{p,t}, E_{p,t})$  is the subgraph of patch  $p$  with
2 // task tag (i.e., sweeping angle)  $t$ .
3 SweepPatchProgram(PatchID  $p$ , TaskTag  $t$ )
4 {
5   // Part 1: Local Context
6   int counts[ $|V_{p,t}|$ ];
7   PriorityQueue  $Q$ ;
8   Map<Pair<PatchID, TaskTag>, Stream> outstreams;
9
10  // Part 2: Interface Implementation
11  void init() {
12     $Q$ .clear();
13    for(each Vertex  $v$  in  $V_{p,t}$ ) {
14      counts[ $v$ ] = #.  $v$ 's upwind vertices;
15      if(counts[ $v$ ]==0)  $Q$ .enqueue( $v$ );
16    }
17  }
18  void input(Stream  $s$ ) {
19    while((edge( $u, v$ ), data( $u$ )) =  $s$ .read()) {
20      counts[ $v$ ] = counts[ $v$ ]-1;
21      if(count[ $v$ ]==0)  $Q$ .enqueue( $v$ );
22    }
23  }
24  void compute() {
25    Vector<Vertex> vertices;
26    //  $N$  is the vertex clustering grain
27    while(! $Q$ .empty() and vertices.size()< $N$ ) {
28      Vertex  $v$  =  $Q$ .dequeue();
29      vertices.push_back( $v$ );
30      for(each  $v$ 's downwind Vertex  $w$ ) {
31        if( $w$  is in  $V_{p,t}$ ) {
32          counts[ $w$ ] = counts[ $w$ ]-1;
33          if(counts[ $w$ ]==0)  $Q$ .enqueue( $w$ );
34        } else {
35          Stream&  $s$  = outstreams(patch( $w$ ),  $t$ );
36           $s$ .write(edge( $v, w$ ), data( $v$ ));
37        }
38      }
39    }
40    solve(vertices); //user-defined computation
41  }
42  Stream output() {
43    Stream  $s$  = fetch(outstreams);
44    return  $s$ ;
45  }
46  bool vote_to_halt() {
47    return  $Q$ .empty();
48  }
49 }

```

The local context contains all necessary states required by a reentrant sweep on the patch, including: (line 6) an array

of counters that count the number of unfinished neighbors for each local vertex, (line 7) a priority queue storing ready vertices, and (line 8) streams later sent to other patch-programs.

The interface functions implement DAG-based data-driven sweeps on the patch p in the direction t . The *init* function initializes each vertex's count variable to the number of its upwind neighbors, and collect source vertices into the ready queue Q . The *input* function receives data of vertices from remote patches, updates counts of related local vertices; once a local vertex's count decreases to zero, put it to the ready queue. The *compute* function collect a sequence of ready vertices and computes on them with user-defined numerical computation, updates their downwind neighboring vertices. The *output* function generates streams sent to remote patch-programs. The *vote_to_halt* function evaluates whether the patch-program should deactivate.

B. Optimization: Patch-Angle Parallelism

JSweep naturally supports simultaneous sweeps on a patch, from different angles (i.e., sweeping directions). As shown in Listing 1, we achieve this by setting task tag of the patch-program to the id of the sweeping direction. Consider the example in Fig. 5 again, in which full S_2 transport sweeps are carried on a 2d unstructured mesh of 2 patches. In this example, sweeps from different angular directions are independent, and thus patch-angle parallelism can be fully enabled. This is especially useful for small meshes with large number of angles.

C. Optimization: Vertex Clustering

We adopt vertex clustering in the patch-program. As shown in Listing 1, the *compute* function collects and computes on multiple ready vertices, rather than a single vertex. For example, in Fig. 4 vertices in the same dashed rectangular are clustered together.

Benefits of this optimization is two-fold. On one hand, vertex clustering can dramatically reduce the scheduling overhead by reducing execution times of a patch-program. For example, in Fig. 4 the patch 1 and patch 2 need only two and three executions respectively, compared to eight times of no clustering. On the other hand, vertex clustering aggregates multiple streams to the same target into a single stream and thus reduces communication overhead. For example, in Fig. 4, the inter-vertex communications of $8 \rightarrow 9$ and $12 \rightarrow 14$ can be combined into one message.

Nevertheless, we need to choose the clustering grain carefully. While reducing overhead of schedule and communication, vertex clustering also has potentially negative effect on parallelism since it may defer the communication thus delay scheduling of other patch-programs. Excessive clustering can lead to long communication delay and thus longer execution time. To illustrate this, consider SnSweep-S, an example in JAXMIN package, which implements a S_n solver for neutron transport equations on 3d structured meshes. The experimental results (mesh cells: $160 \times 160 \times 180$, patch size: $20 \times 20 \times 20$, S_2 ordinates, 8*12 CPU cores) are shown in Fig. 9a.

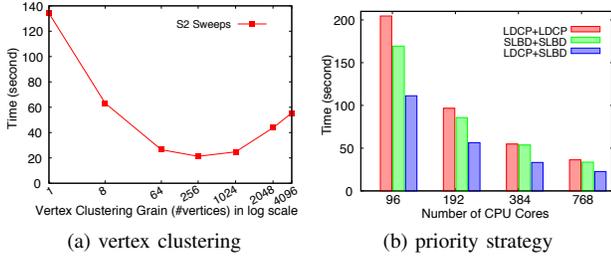


Fig. 9: Performance effect of optimization parameters

D. Optimization: Priority Strategy

We adopt a two-level hierarchical priority strategy, i.e., (*patch*, *angle*) priority and vertex priority.

The (*patch*, *angle*) priority is used for JSweep runtime to schedule patch-programs. For patch p and angle a , its priority is calculated by the following formula:

$prior(p, a) = prior(a) * C + prior(p)$, where C is a constant factor. In S_n sweep, to avoid waiting of downwind patches, we want patch-programs with the same angle are continuously scheduled to execute such that the data streams are delivered to the nearby patches as quickly as possible. Thus we set the importance of $prior(a)$ always higher than $prior(p)$ in the formula, by multiplying a large factor C over $prior(a)$. Meanwhile, with respect to $prior(p)$, however we can't reduce a single objective. On one hand, we hope the upwind patches are computed as earlier as possible such that more parallelisms are available. On the other hand, we also want the patches neighbouring other unfinished patches are computed earlier, but these preferred patches have typically on the downwind of a sweeping direction. Based on the first objective, we develop two priority strategies: LDCP (Longest Distance on Critical Path) for structured meshes and BFS (Breadth First Search) for unstructured meshes. Based on the second objective, we develop the priority strategy SLBD (Shortest Local Boundary Distance, a DFS variant that prefers vertices most close to patch boundary) for both structured and unstructured meshes.

The vertex priority is used within a patch-program to order local ready vertices in PriorityQueue Q in Listing.1. As $prior(p)$, vertex priority also has to trade off more parallelism and earlier communication. The strategies proposed for $prior(p)$, i.e., BFS, LDCP (for structured meshes only) and SLBD, are also suitable for vertex priority. In practice, however we observed that SLBD performs constantly best for unstructured and especially unstructured meshes, as illustrated by SnSweep experiments in Fig.9b.

E. Extra Optimization: Coarsened Graph

Coarsened graph, not presented in Listing.1 for length limit, can be treated as an extension to vertex clustering. In reality, the mesh structure and its data dependencies are always constant in most or even all sweeping iterations. Thus we can cache the vertex clustering results to build a reusable coarsened graph. For example, in Fig. 10, the directed graph

(left) is transformed into a much smaller coarsened graph (right) according to the previous clustering results in Fig. 4.

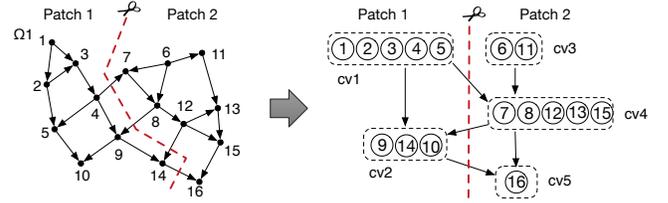


Fig. 10: Graph Coarsening

Formally, we define a coarsened graph as the property graph: $CG = (CV, CE, P(CV), P(CE))$ where CV is set of coarsened vertices derived and CE the set of coarsened edges. CG is the topology of vertex clusters and the directed communication relationships of vertex clusters. Property of a coarsened vertex $cv \in CV$, $P(cv)$, is the series of corresponding clustered DAG vertices, while property of a coarsened vertex $ce \in CE$, $P(ce)$, is the combined edges of source vertices and target vertices in DAG. For example, in Fig. 10, $P(cv2) = (9, 14, 10)$, $P(cv4) = (7, 8, 12, 13, 15)$, $P(cv4 \rightarrow cv2) = (\{8, 12\}, \{9, 14\})$. Since CG is the task graph generated in the scheduling process, we have the computability theorem:

Theorem 1: If a directed graph G is acyclic, its derived coarsened graph CG is also acyclic.

Besides, CG is distributed at the beginning of its construction. We implement it in same technologies presented in agent-graph [40]. With coarsened graph, sweep is carried on DAG in the first iteration and on CG in all subsequent iterations until the mesh changes. In our practice with JSNT-S [25], the cost of building CG is less the one DAG-based sweep iteration itself while the speedup of sweeps on CG over DAG can be 7 – 10 folds.

VI. EVALUATION

Platform All experiments were carried on Tianhe-II, the world's fastest supercomputer in 2015 [3]. We use at most 3200 nodes. Each node has two Intel Xeon E5-2692v2 12-core processors, equipped with 64GB memory and Tianhe-Express-II network of 40GB/s bandwidth. The operating system is Kylin Linux. All applications are compiled with Intel C Compilers (icc13) and customized MPICH2.

Applications We use two real JAXMIN-based S_n applications, JSNT-S [25] and JSNT-U [26], to investigate the efficiency of JSweep on structured and unstructured meshes respectively. The used meshes are visualized in Fig. 11.

A. Evaluation on structured meshes

JSNT-S [25] is a JASMIN-based S_n package for structured meshes, which implements most functionalities of TORT [24]. We use the well-known Kobayashi benchmark to evaluate JSweep. In particular, we focus on the strong scalability. In all the following experiments, JSweep is configured as follows:

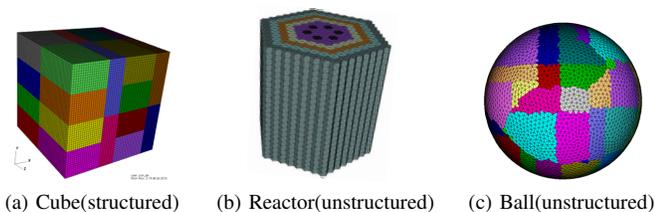


Fig. 11: Shapes of tested meshes

patch size = $20 \times 20 \times 20$, vertex clustering grain = 1000, and the priority strategy is SLBD+SLBD.

We first evaluate JSweep with the original Kobayashi benchmark (Kobayashi-400). It solves the single energy group S_n transport equations with scattering, on a cubic mesh (Fig.11a) of $400 \times 400 \times 400$ cells with 320 angular directions. As presented in Fig.12a, with increasing number of cores, JSweep shows reasonable scalability constantly, with a speedup 14.3 (or parallel efficiency 44.7%) on 24,576 cores compared to 768 cores.

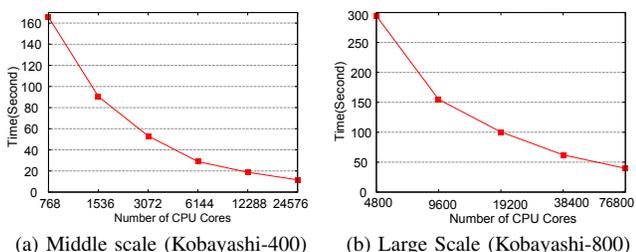


Fig. 12: Runtime of JSNT-S for Kobayashi Benchmark

We then evaluate JSweep on more CPU cores with a larger problem by modifying the mesh of Kobayashi input to $800 \times 800 \times 800$ proportionally, namely Kobayashi-800. As shown in Fig. 12b, JSweep scales to 76,800 cores with a reasonable speedup 7.4 (or parallel efficiency 46.3%), normalized to performance on 4,800 cores.

B. Evaluation on unstructured meshes

JSNT-U [26] is a JAUMIN-based S_n package for unstructured meshes, primarily used for numerical simulations in high energy physics. We evaluate JSweep on two shapes of unstructured meshes, reactor core (Fig.11b) and ball (Fig.11c). Unless otherwise stated, default configurations of experiments are as follows: priority strategy SLBD+SLBD, patch size = 500 cells, vertex clustering grain = 64, #angles = 24 (S_4) and #energy groups = 4.

1) *Hyper-parameters' effect to performance*: We change and investigate three hyper parameters respectively in order, i.e., patch size, vertex clustering grain and priority strategy, while keeping others default. As shown in Fig.13a (left), with increasing patch sizes (i.e., #cells of a patch), the runtime first decreases quickly since the larger patch size reduces total communication between patches, and then slightly increases since the larger patch size also leads to longer waiting time

of downwind patches whose execution is driven by data from this patch. Fig.13a (right) shows the effect of maximum vertex clustering grains. With increasing vertex clustering grain, the runtime decreases quickly and then keeps steady. Unlike on structured meshes (Fig.9a), however, the runtime no longer increases with a large clustering grain. By profiling, we found that the actual number of available vertices is between 16 and 64 at most time, which means the real clustering grain is limited by parallelism. With respect to priority strategies, as shown in Fig.13b, their effect to performance is not so significant as that on structured meshes (Fig. 9b).

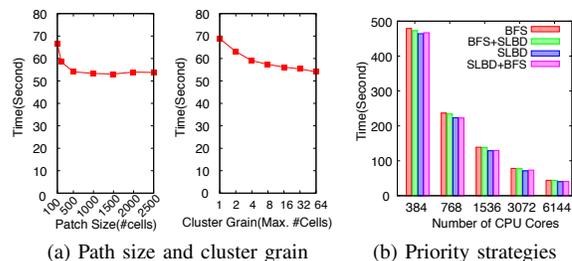


Fig. 13: Hyper parameter's effect in JSNT-U (mesh: reactor)

2) *Strong Scalability*: As shown in Fig. 14, JSweep performs good strong scalability on both small and large meshes. For the small scale problem (ball of 482,248 cells), JSweep shows a speedup of 11.5 (parallel efficiency 72%) at 384 cores and goes to a speedup of 75.8 (parallel efficiency 30%) at 6,144 cores, normalized to the 24-core base. For the large scale problem (ball of 173,197,768 cells), JSweep shows a speedup of 9.9 (parallel efficiency of 62%) at 49,152 cores, normalized to the 3,072-core base. Given that our tested mesh is a ball constructed with tetrahedrons, the above scalability should be reasonably good.

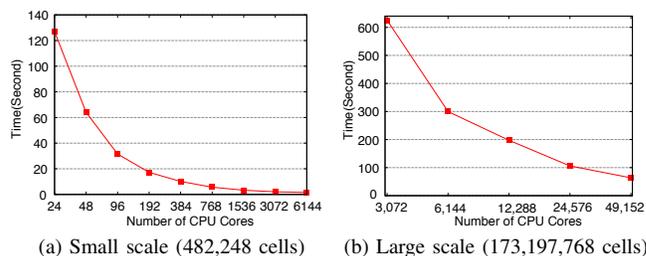


Fig. 14: Strong scalability of JSNT-U on ball meshes

3) *Weak Scalability*: Fig. 15 presents the results of weak scalability evaluated on the ball (originally 482,248 cells) and reactor core (originally 64,479 cells) meshes. In particular, mesh size is increased in a normal approximate refinement method. As shown, the weak scalability of JSweep is not good enough, although reasonable. For reactor, the parallel efficiency at 12,288 cores is about 40%, while for ball it is lower than 20%. One possible reason is that in JAXMIN, the original small mesh is first partitioned and distributed to processes, and then each process refines the assigned subdomain,

leading to *thick* subdomains that dramatically increase length of critical path in the sweeping direction.

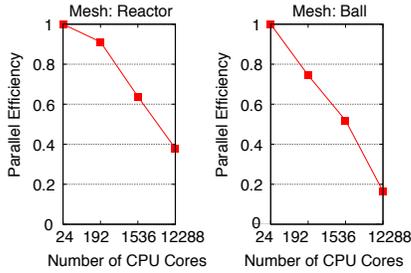


Fig. 15: Weak scalability of JSNT-U

C. Runtime Overhead Analysis

JSweep employs a runtime-based approach, thus the overhead is essential for performance. To investigate the overhead, we carry out a detailed profiling of JSNT-S on small scale Kobayashi benchmark. In particular, the problem has a $200 \times 200 \times 200$ mesh. All optimizations are enabled and all hyper parameters are the same with that in Sec.VI-A. We present one sweep iteration using coarsened graph.

Fig.16 shows the time breakup in a strong-scaling fashion. The overhead introduced by JSweep (i.e., the graph-op and pack/unpack) is moderately low (approx. 23%), and the major performance loss comes from idling of CPU cores (22%-46%). Communication takes 13%-19% the total time. With more deep optimization and advanced priority strategy, we expect to lower both the overhead and the idle time.

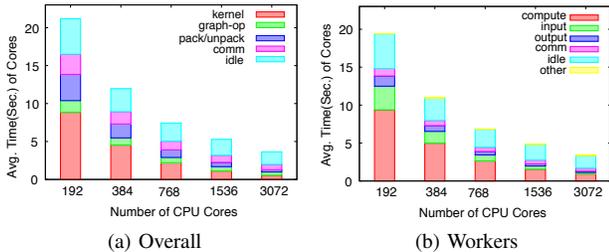


Fig. 16: Runtime breakup of JSNT-S

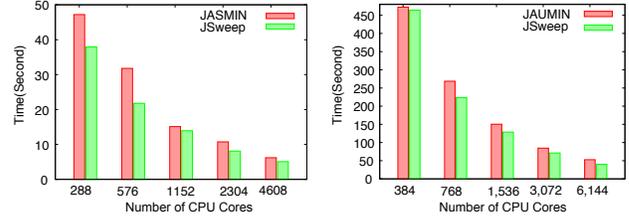
D. Performance Comparison with other systems

We first compare JSweep’s performance with previous JAxMIN, which already implement efficient algorithmic optimizations [32] and achieve good performance. Nevertheless, we show that with innovations on data-driven abstraction and runtime system design, JSweep outperforms them on both structured and unstructured meshes.

Fig.17a presents results of JSweep and JASMIN-based SnSweep program (a data-driven implementation of Sweep3D [12]). We choose SnSweep because it has been optimized manually with all techniques introduced in Sec.V, including a coarsened graph variant which caches the vertex clusters and

their communication relationships by MPI tags. As shown, JSweep’s runtime is constantly less than JASMIN.

Fig.17b presents runtime comparison of JSweep and JAUMIN-based JSNT-U. Again, JSweep shows constant runtime reduction, and with increasing number of cores the comparative advantage becomes slightly bigger.



(a) JSweep vs JASMIN (koba400) (b) JSweep vs JAUMIN (Ball)

Fig. 17: Performance comparison of JSweep vs JAxMIN

Besides, Table-I compares parallel efficiency of JSweep with other work in literatures. We can see that for Kobayashi problem, JSweep demonstrates comparable scalability with Denovo’s KBA-based implementation. For unstructured ball (sphere) mesh of tetrahedrons, JSweep scales worse than the manually implemented data-driven algorithm PSD-b [27]. However, note that JSweep is a solution of general framework. Due to the lack of common public problems and availability of the systems, it is difficult to compare frameworks directly.

TABLE I: Performance comparison with literatures

Application	Problem	Par. Eff.	#cores (max. vs base)
Denovo [31]	Kobayashi-400	77.8%	3,600 vs 144
JSweep	Kobayashi-400	89.6%	6,144 vs 384
PSD-b [27]	sphere, 151,265, S_4	88%	1,024 vs 128
JSweep	sphere, 482,248, S_4	66%	1,536 vs 192

VII. RELATED WORK

The idea of patch-centric abstraction is partly inspired by the *vertex-centric* models [38] in graph-parallel frameworks [38] [39]. In a vertex-centric model, user defines a vertex-program for a single vertex and the framework lifts the vertex computation to the whole graph, conceptually in parallel. However, unlike vertex in graph, patch is not the basic element of mesh, which means patch-centric data-driven abstraction has fundamental difficulties including partial computation, priority inversion and multi-tasks on a single patch. In this paper we comprehensively addresses these issues and formalize a general patch-centric data-driven approach. In fact, the patch-centric abstraction can be seen as a straightforward extension to existing patch-based frameworks (see a survey in [23]), such as SAMRAI [28], (part-based) PUMI [29] and especially JAxMIN [1] [2], in which the mesh is decomposed into and managed by patches.

Task-based programming models, such as PaRSEC [36] and more general Charm++ [37], are also suitable to implement

location-based data-driven computations. For example, a recent work [22] has implemented a ParSEC-based S_n sweep solver on 3d cartesian meshes, demonstrating high efficiency with 34% of peak performance at 384 cores. Compared to JSweep, however, these task-based frameworks are not well-abstracted for mesh-specific parallel computation and thus require users to remap many conceptions.

VIII. CONCLUSIONS AND FUTURE WORK

We have presented JSweep, a generic patch-centric data-driven framework integrated in the JAXMIN infrastructure. In particular, we propose the patch-centric data-driven abstraction whose essential idea is extending the concept *patch* as a logical processing element that is fully reentrant. Also, our abstraction supports multiple tasks on a single patch and arbitrary patch priority strategies. Further, targeting contemporary HPC systems of multicore cluster architecture, we implemented a high performance runtime system to map the patch-centric data-driven computation to underlying system resources. Based on the above approach, we implemented a new parallel sweeps algorithm as a component in JAXMIN, featuring patch-angle parallelism, vertex clustering and hierarchical priority strategy. Evaluation with two real S_n software packages demonstrates that JSweep can scale to at least 49,152 cores for unstructured meshes and 76,800 cores for structured meshes with reasonable parallel efficiency.

What distinguishes JSweep from most counterparts is that we consider *sweep* computations in the context of general mesh-based application frameworks which would be critical in future coupling of multiple multi-physics simulations. Besides S_n transport sweeps, our abstraction also supports other data-driven algorithms well, e.g., particle trace which we have implemented as another component in JAXMIN. Given the increasing importance of data-driven computation and demands on coupling multi-physics simulations, we believe efficient support to both BSP and asynchronous data-driven models are necessary to construct high performance applications.

REFERENCES

- [1] Z. Mo, A. Zhang, X. Cao, Q. Liu, X. Xu, H. An, W. Pei, S. Zhu, JASMIN: a parallel software infrastructure for scientific computing, *Frontiers Computer Science of China*, 2010, 4(4):480–488.
- [2] Q. Liu, W. Zhao, J. Cheng, Z. Mo, A. Zhang and J. Liu, A programming framework for large scale numerical simulations based on unstructured meshes, in *Proc. HPSC*, New York, Apr. 6–8, 2016.
- [3] <http://top500.org/2015-nov>, Nov., 2015.
- [4] R. L. Bowers, J. R. Wilson, *Numerical modeling in applied physics and astrophysics*, Jones and Bartlett publishers, 1991.
- [5] J. Bey, G. Wittum, On the robust and efficient solution of convection diffusion problems on unstructured grids in two and three space dimensions, *Applied Numerical Mathematics*, 1997, 23(1):177–192.
- [6] F. Wang, J. Xu, A cross-wind strip block iterative method for convection-dominated problems, *SIAM Journal of Computing*, 1999, 21:646–665.
- [7] T. Downar, A. Siegel, C. Unal. Science Based Nuclear Energy Systems Enabled by Advanced Modeling and Simulation at the Extreme Scale. *White Paper on Integrated Performance and Safety Codes*, 2009.
- [8] R. Baker, R. Alcouffe. Parallel 3-D S_n Performance for MPI on Cray-T3D. In *Proc. Joint International Conference on Mathematics Methods and Supercomputing for Nuclear Applications*, New York, Oct., 1997.
- [9] R. Baker, K. Koch. An S_n algorithm for the massively parallel CM-200 computer. *Nuclear Science and Engineering*, 1998, 28: 312–320.
- [10] Lawrence Livermore National Laboratory, Amda: Scalable parallel code system to perform neutron and radiation transport calculations, <http://www.llnl.gov/casc/amda>.
- [11] W. Hawkins, et al., Efficient Massively Parallel Transport Sweeps, *Trans. Am. Nucl. Soc.*, 107, 477, 2012.
- [12] Los Alamos National Laboratory. The ASCI Sweep3d Benchmark. <http://www.ccs3.llnl.gov/pal/software/sweep3d>.
- [13] G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 1990, 33(8):108–111.
- [14] J. Misra. Detecting termination of distributed computations using markers. In *Proc. PODC*, pages 290–294, 1983.
- [15] S. Plimpton, B. Hendrickson, S. Burns, W. McLendon. Parallel algorithms for radiation transport on unstructured grids. In *Proc. SC*, Dallas, Nov., 2000.
- [16] Z. Mo, A. Zhang and X. Cao. Towards a parallel framework of grid-based numerical algorithms on DAGs. In *Proc. IPDPS*, Greece, 2006.
- [17] Hewitt C, Bishop P, Steiger R. A Universal Modular Actor Formalism for Artificial Intelligence. In *Proc. IJCAI*, San Francisco, 1973.
- [18] G. Karypis, V. Kumar. Multi-level graph partitioning schemes. In *Proc. of ICPP*, Urbana-Champaign, Aug., 1995, pp.113–122.
- [19] B. Hendrickson, R. Leland. A multilevel algorithm for partitioning graph. In *Proc. of SC*, San Diego, 1995.
- [20] S. Pautz. An Algorithm for Parallel S_n Sweeps on Unstructured Meshes. *Nuclear Science and Engineering*, 2002, 140(2): 111–136.
- [21] M. Mathis, D. Kerbyson. A General Performance Model of Structured and Unstructured Mesh Particle Transport Computations. *Journal of Supercomputing*, 2005, 34:181–199.
- [22] S. Moustafa, M. Faverge, L. Plagne and P. Ramet. 3D Cartesian Transport Sweep for Massively Parallel Architectures with ParSEC, In *Proc. IPDPS*, 2015.
- [23] A. Dubey, et al., A survey of high level frameworks in block-structured adaptive mesh refinement packages. *J. Parallel Distrib. Comput.* (2014), <http://dx.doi.org/10.1016/j.jpdc.2014.07.001>.
- [24] W. A. Rhoades and D. Simpson, The TORT Three-Dimensional Discrete Ordinates Neutron/Photon Transport Code, ORNL/TM-13221, Oct., 1997.
- [25] T.P. Cheng and L. Deng, JSNT-S manual, IAPCM, 2015.
- [26] J.X. Wei, JSNT-U (3DSn) manual, IAPCM, 2010.
- [27] G. Colomer, R. Borrell, F. Trias, and I. Rodriguez, Parallel algorithms for S_n transport sweeps on unstructured meshes. *Journal of Computational Physics*, vol. 232, no. 1, pp. 118–135, 2013.
- [28] SAMRAI. <https://computation.llnl.gov/casc/SAMRAI>, May 31, 2010
- [29] D. A. Ibanez, E. S. Seol, C. W. Smith and Mark S. Shephard, PUMI: Parallel Unstructured Mesh Infrastructure. *ACM Transactions on Mathematical Software*, 2015.
- [30] G. Davidson, T. Evans, J. Jarrell, S. Hamilton and T. Pandyan, Massively Parallel Three-dimensional Transport Solutions for the k-eigenvalue Problem. *Nuclear Science and Engineering*, 2014, 177:111-125.
- [31] T. Evans, A. Stafford, R. Slaybaugh and K. Clarno, Denovo: a new three-dimensional parallel discrete ordinates code in scale. *Nuclear Technology*, 2010, 171(8), 171-200.
- [32] Z. Mo, A. Zhang, and Z. Yang. A new parallel algorithm for vertex priorities of data flow acyclic digraphs, *The Journal of Supercomputing*, vol. 68, no. 1, pp. 49-64, 2014.
- [33] S. Pautz and T. Bailey, Parallel Deterministic Transport Sweeps of Structured and Unstructured Meshes with Overloaded Mesh Decompositions, *Proc. Joint International Conference on Mathematics and Computation (M&C), Supercomputing for Nuclear Applications (SNA) and the Monte Carlo (MC) Methods*, Nashville, TN, April 19-23, 2015.
- [34] P. Song, C.L. Zhai, S.G. Lee, et.al, LARED-I: The Integrated Code for Laser-driven Inertial Confinement Fusion. *High Power Laser and Particle Beam* (in Chinese), 2015, 27(3):54-60.
- [35] D. R. Gaston, C. J. Permann, J. W. Peterson, A. E. Slaughter, D. Andrsie, Y. Wang, M. P. Short, D. M. Perez, M. R. Tonks, J. Ortensi, Ling Zou and R. C. Martineau, Physics-based multi-scale coupling for full core nuclear reactor simulation, *Annals of Nuclear Energy*, 2015, 84:45–54.
- [36] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier and J. Dongarra, DAGuE: A generic distributed DAG engine for High Performance Computing, *Parallel Computing*, vol.38, no.1-2, 2012.
- [37] L.V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth and G. Zheng, Programming peta-scale applications with Charm++ and AMPI, *Peta-scale Computation: Algorithms Appl.* 1(2007): 421–441.
- [38] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

- [39] J. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin. PowerGraph: distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.
- [40] J. Yan, G. Tan, Z. Mo, N. Sun, Graphine: programming graph-parallel computation of large natural graphs for multicore clusters, In *IEEE Transaction on Parallel and Distributed Systems*, 27(6):1647-1659, 2016.