2018 IEEE International Parallel and Distributed Processing Symposium Workshops

# A Preliminary Port and Evaluation of the Uintah AMT Runtime on Sunway TaihuLight

Zhang Yang
Institute of Applied Physics and Computational Mathematics
Email: yang_zhang@iapcm.ac.cn

Damodar Sahasrabudhe
Scientific Computing and Imaging Institute
Email: damodars@sci.utah.edu

Alan Humphrey
Scientific Computing and Imaging Institute
Email: ahumphrey@sci.utah.edu

Martin Berzins
Scientific Computing and Imaging Institute
Email: mb@sci.utah.edu

***Abstract*—The Sunway TaihuLight is the world's fastest supercomputer at the present time with a low power consumption per flop and a unique set of architectural features. Applications performance depends heavily on being able to adapt codes to make best use of these features. Porting large codes to novel architectures such as Sunway is both time-consuming and expensive, as modifications throughout the code may be needed. One alternative to conventional porting is to consider an approach based upon Asynchronous Many Task (AMT) Runtimes such as the Uintah framework considered here. Uintah structures the problem as a series of tasks that are executed by the runtime via a task scheduler. The central challenge in porting a large AMT runtime like Uintah is thus to consider how to devise an appropriate scheduler and how to write tasks to take advantage of a particular architecture. It will be shown how an asynchronous Sunway-specific scheduler, based upon MPI and athreads, may be written and how individual task-code for a typical but model structured-grid fluid-flow problem needs to be refactored. Preliminary experiments show that it is possible to obtain a strong-scaling efficiency ranging from 31.7% to 96.1% for different problem sizes with full optimizations. The asynchronous scheduler developed here improves the overall performance over a synchronous alternative by at most 22.8%, and the fluid-flow simulation reaches 1.17% the theoretical peak of the running nodes. Conclusions are drawn for the porting of full-scale Uintah applications.**

***Index Terms*—Asynchronous Many Task Runtime, Asynchronous Scheduler, Uintah, Sunway TaihuLight, Strong Scalability, Performance Evaluation**

## I. Introduction

Progress towards Exascale computing is facing the challenge of needing to obtain performance while keeping power budgets at an economically feasible level. This has driven the latest generation of supercomputers to embrace low frequency many core architectures. Examples of this are current machines based upon GPUs, Intel Knights Landing, and a number of novel architectures such as that of Sunway TaihuLight [1], the current No. 1 in the top 500 list. A central challenge is then how to port software to a wide variety of such machines and to their successors at exascale.

For example the 100PF Sunway TaihuLight features a special architecture consisting of 41K nodes with SW26010 processors, where each processor contains four Core-Groups (CGs). Each CG is made up of one Management Processing Element (MPE) and 64 Computing Processing Elements (CPEs) sharing the same main memory as is described below. Each CPE is equipped with a small user-controlled scratch pad memory instead of data caches. This architecture has made it possible to run many real-word applications at substantial fractions of peak performance, such as the three applications selected as Gordon Bell finalists in SC16 [2]–[4]. However, these performance levels were obtained through extensive and intensive tuning of the code at a level that may not be possible or affordable for general application codes.

One way to partition the porting is to consider using an Asynchronous Many Task (AMT) runtime system such as Uintah [5], Charm++ [6], StarPU [7] or Legion [8], for example, as this may make it possible to consider modifying the runtime task mapping system and the tasks themselves independently. To the best of the authors knowledge, none of these runtime systems has been adapted to take advantage of the Sunway TaihuLight architecture at the present time. Given the use of Uintah on many other top ten systems, this code will be considered here.

After describing the Uintah software, the model problem and the Sunway Architecture, in Sections II, III, and IV, the contributions of this paper in Sections V, VI , and VII are to:

1) Develop an asynchronous scheduler based on MPI and *athread* to take advantage of the shared memory MPE+CPE architecture of Sunway TaihuLight;
2) Port a typical but model fluid-flow simulation to Sunway TaihuLight with the asynchronous scheduler;
3) Show how many FLOPs a preliminary scheduler is able to extract for the model problem on the Sunway TaihuLight.

The main motivation of this work was to understand how to port an asynchronous many task runtime system framework like Uintah to Sunway and to understand how a model application can take advantage of the architectural features of SW26010 with such a system.

## II. The Uintah Framework

Uintah [5], [9] is an open-source software framework that can be used to solve partial differential equations (PDE) on

978-1-5386-5555-9/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPSW.2018.00155



IEEE computer society

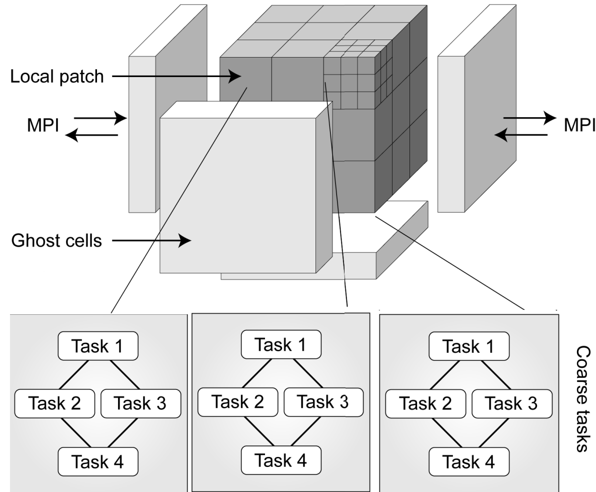


Fig. 1: Patches and user tasks in Uintah

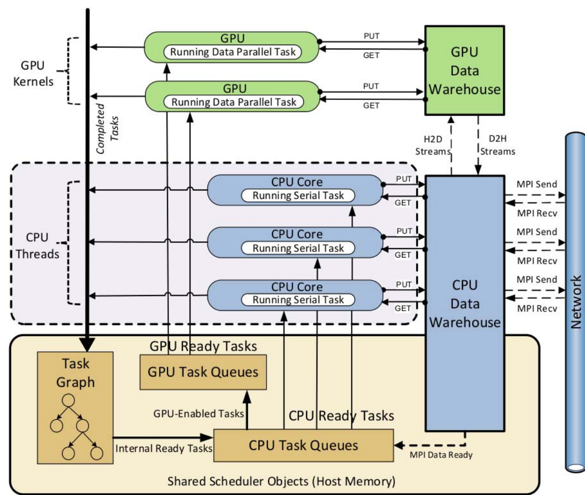


Fig. 2: Architecture of an Uintah scheduler [5]

adaptively refined structured meshes, which in turn can be used for problem solving and simulations in multiple application domains. The Uintah infrastructure is an asynchronous many task runtime system built on top of a patch-centric discretization of structured meshes. Beside predefined simulation components, the Uintah infrastructure provides users APIs to describe their problems as a collection of dependent coarse tasks, to create variables and associate them with the tasks, and to add these tasks into the system. Under the hood, Uintah builds a distributed task graph and uses a scheduler to run them in an out of order manner on available resources, as depicted in Figure 2. In this way, Uintah keeps users insulated from all of parallel executing details and so allows users to focus on solving the problem at hand. This decoupling allows independent and parallel development of infrastructure and user applications.

Uintah's task definition and creation are illustrated in Figure 1. As an adaptive mesh refinement (AMR) framework, Uintah subdivides the computational grid into patches, and assigns groups of patches to distributed memory computing nodes. When a task needs values from neighboring patches, a dependency on neighboring patches is created, and MPI messages are used to transfer the required values. To build the distributed task graph, the user provided coarse tasks that are mapped to these patches, where each combination of a task with a patch defines a task object. The dependencies between task objects are defined by both the coarse-task dependency specified by the user and the neighbor-value dependency among patches. The task graph is naturally distributed since each computing node builds its portion on its own group of patches. More details on the task definition and task graph compiling can be found in [10].

Uintah has built-in support for time-stepping applications. The task graph defined above is built at the first timestep, and remains unchanged until the mesh patch partition needs to be changed, due to load imbalances or adaptively mesh refinements. The "data warehouse" concept is introduced to distinguish data in different timesteps. The fundamental mechanism is that the *old* data warehouse holds the data calculated in the previous timestep. The coarse task takes what it needs from the old data warehouse and produces results that then populate the *new* data warehouse in current timestep. After the timestep is completed, the new datawarehouse becomes the old datawarehouse for the next timestep.

Parallel execution is controlled inside the Uintah framework through a task scheduler. As depicted in Figure 2, the scheduler takes the local portion of the task graph as its input, executes the tasks in ordered or possibly out of order fashion while preserving task dependencies, and issues MPI sends and receives through the data warehouse to drive the progress of dependent tasks on remote computing nodes. As long as the scheduler is properly designed and given enough resources, task computations and communication through the data warehouse can happen concurrently, thus communications and computations are naturally overlapped. This can effectively improve the scalability, and has allowed Uintah to scale up to 768K cores and 16384 GPUs [11]–[13].

The most efficient scheduler in Uintah that is able to overlap communications with computations is the "Unified Scheduler". The Unified Scheduler is built upon a MPI+thread model, where only one MPI process is started on each computing node, and multiple threads are started in the process to make use of multiple CPU cores. Each thread controls a CPU core and executes serially a task fed by the scheduler. Unified scheduler uses atomic variables and lock free pools to fetch tasks and to handle communications, and the lock free implementation reduces synchronization time. The most recent Unified Scheduler allows tasks to be fed to both GPUs and CPUs.

*Challenges arising from the Sunway architecture*: The Unified Scheduler requires multiple threads running on different CPU cores to effectively overlap communications and compu-

tations. However, the Sunway's SW26010 processor has only one MPE on each CG, which would limit the scheduler to one thread. Thus the Unified Scheduler is not able to effectively overlap communications with computations without a new design.

## III. A Model Fluid-flow Problem

While it would be possible to port one of the Uintah application codes mentioned above, in order validate the effort porting the Uintah infrastructure and to quickly understand how to write task codes for the Sunway architecture, a 3D time-dependent model problem was constructed. The 3D model Burgers equation used here is equivalent to many of the equations in the Uintah applications in terms of its computational structure and reflects the types of PDEs in many different applications in areas such as fluid mechanics, molecular acoustics, gas dynamics, and traffic flow. The equation is given by

$$\frac{\partial u}{\partial t} = -\phi(x,t)\frac{\partial u}{\partial x} - \phi(y,t)\frac{\partial u}{\partial y} - \phi(z,t)\frac{\partial u}{\partial z} + \nu\Delta u) \quad (1)$$

The initial and boundary conditions are give by:

$$u(x,y,z,t) = \phi(x,t)\phi(y,t)\phi(z,t)$$

and $\phi(x,t)$ is the solution of a 1D Burgers equation given by: $\phi(x,t) = \frac{0.1e^a+0.5e^b+e^c}{e^a+e^b+e^c}$, with $a = (-0.05*(x-0.5+4.95*t)/\nu$, $b = -0.25(x-0.5+0.75t)/\nu$, $c = -0.5(x-0.375)/\nu$, and $\nu = 0.01$ is the viscosity of the medium. Dividing the numerator and denominator of the expression for $\phi(\dots)$ by the largest value of $e^a, e^b, e^c$ reduces the number of exponentials needed by one.

The problem is discretized within Uintah by using finite differences on a three dimensional regular grid. The first order derivatives are discretized using backward differences and the second order derivatives are discretized using second order central differences. The forward Euler method is used to discretize the equation in time. The solution $u$ values are situated at the centroid of a cell. Given a timestep size $dt$ and the discretized form of the right side of equation 1 being denoted by $du$, the solution is incremented using:

$$u_{t_{n+1}} = u_{t_n} + dtdu$$

where $dt$ is chosen to ensure stability for the forward Euler time integration. The exact solution of the Burgers equation $u(x,y,z,t)$ for $t=0$ is used to calculate the initial condition. The main kernel for solving Equation 1 using the above methods is presented in Algorithm 1. To calculate on a patch, the kernel requires exactly one extra layer of ghost cells.

### A. Performance characteristics

The model problem was so designed that it resembles typical Uintah applications while remains simple enough for experimenting performance optimization techniques. The Burgers kernel combines a low-order stencil structure with complex coefficient evaluations, which is typical in scientific and engineering applications. The exponentials and branching

**Algorithm 1** Pseudo code for the Burgers kernel

1: **for** every cell `(i, j, k)` **in** current patch **do**
2: $u_dudx = \phi(i*dx, dt)*(u_{i-1,j,k} - u_{i,j,k})/dx$
3: $u_dudy = \phi(j*dy, dt)*(u_{i,j-1,k} - u_{i,j,k})/dy$;
4: $u_dudz = \phi(k*dz, dt)*(u_{i,j,k-1} - u_{i,j,k})/dz$;
5: $d2udx2 = (-2*u_{i,j,k} + u_{i-1,j,k} + u_{i+1,j,k})/(dx^2)$;
6: $d2udy2 = (-2*u_{i,j,k} + u_{i,j-1,k} + u_{i,j+1,k})/(dy^2)$;
7: $d2udz2 = (-2*u_{i,j,k} + u_{i,j,k-1} + u_{i,j,k+1})/(dz^2)$;
8: $du = -((u_dudx + u_dudy + u_dudz) + \nu*(d2udx2 + d2udy2 + d2udz2))$;
9: $u^{new}_{i,j,k} = u_{i,j,k} + dt*du$;
10: **end for**

logic in the $\phi(\dots)$ calls in the kernel exclude performance-oriented choices making extensive use of the regular stencil structure, while the low order of the stencil prevents excessive data reuse optimizations. This complexity reflects that of some of the stencils of real applications in Uintah. With the presence of exponentials, it is not possible to count directly the floating point operations of the kernel. Instead, an experiment was carried out to evaluate the floating point operations (FLOPs) per cell with varying problem sizes by counting them directly using precise hardware counters on SW26010. Table I indicates the FLOPs per cell of the model problem is around 311, 215 of which is contributed by the exponentials. Given the 16 byte memory access required per cell as indicated by Algorithm 1, the arithmetic intensity of the kernel is approximately 19.4 Flop/Byte, and is still memory-bounded compared to that of the SW26010 processor.

TABLE I: FLOP per cell for the model problem

| Problem Size | Total Cells | Total FLOPs | FLOPs per Cell |
|---|---|---|---|
| 16x16x512 | 17339400 | 5179553014 | 299 |
| 16x32x512 | 34412040 | 10399936968 | 302 |
| 32x32x512 | 68294664 | 20881857690 | 306 |
| 32x64x512 | 136059912 | 41845438269 | 308 |
| 64x64x512 | 271065096 | 83854642144 | 309 |
| 64x128x512 | 541075464 | 167873049894 | 310 |
| 128x128x512 | 1080045576 | 336073950828 | 311 |

## IV. Architectural features of Sunway TaihuLight

Sunway TaihuLight is currently the world's fastest supercomputer [14], with a theoretical peak performance at 125.4 Pflop/s and HPL benchmark performance at 93 Pflop/s. Like other supercomputers nowadays, Sunway employs an MPP (Massively Parallel Processing) architecture. The system is composed of 40,960 SW26010 processors connected with a proprietary high speed network. Here architectural features relevant to this work are described. Some important system parameters of Sunway are summarized in Table II and more details can be found in [1], [15].

### A. Architectural features of the SW26010 processor

As shown in Figure 3, the SW26010 processor is composed of four identical core-groups (the CGs) connected by an on-chip interconnect network. Each CG is a heterogeneous

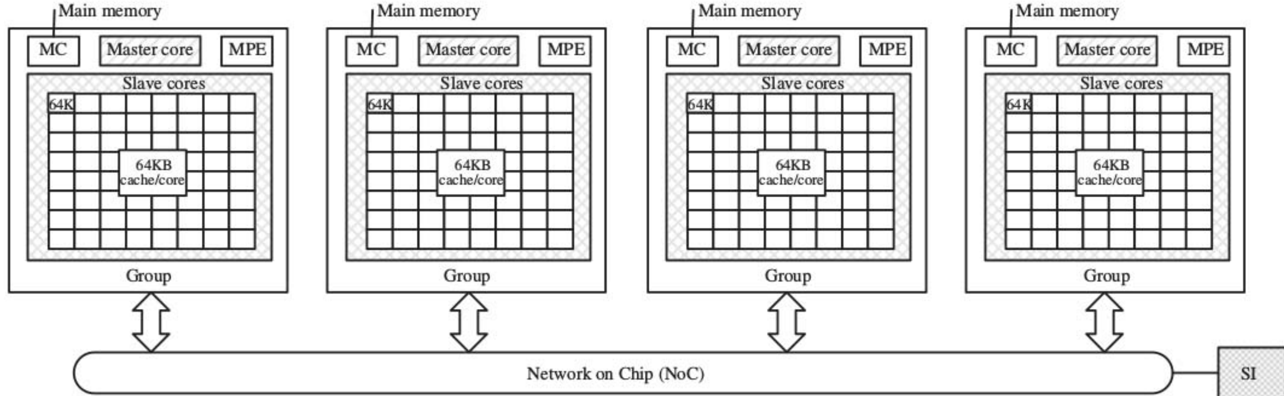


Fig. 3: Sunway SW26010 architecture [1]

| Item | Description |
|---:|---|
| Node architecture | 1 SW26010 processor |
| Node cores | 4 MPEs + 256 CPEs, 260 cores |
| Node memory | 32GB, 4*128bit DDR3-2133 |
| Node Performance | 3.06 Tflop/s |
| Interconnect Bandwidth | Bidirectional P2P 16 GB/s |
| Interconnect Latency | around 1 $\mu s$ |

TABLE II: Major system parameters of Sunway TaihuLight

computing unit consisting of one 128bit-wide DDR3-2133 memory controller (the MC), one Management Processing Element (the MPE), and a cluster of 64 Computing Processing Units (the CPEs). The usual and recommended practice is to use these CGs as separate computing nodes, thus 'CG' and 'computing node' are used interchangeably in this paper.

The most significant architectural feature of SW26010 is the "shared-memory MPE+CPE heterogeneous architecture". The MPE and CPEs share the same memory space connected to the memory controller, thus they can communicate directly via the main memory. This enables light-weight kernel offloading to the CPEs. Although with different ISAs, the MPE and CPEs are based on a RISC architecture, 64 bit, SIMD, and out-of-order microarchitecture, so both can run the user's application. Performance of the MPE is 23.2 Gflop/s, and that is 742.4 Gflop/s for the cluster of CPEs. Since the MPE only contributes 3% of the aggregated performance, little is sacrificed if the MPE is excluded from intensive floating point computing. This architecture is suitable for asynchronous many task runtimes since overlapping communications (as well as other runtime-related tasks) and computations with a low-overhead offloading interface is possible. SW26010 also provides instructions for CPEs to atomically increase a 4 or 8 byte location in the main memory, which helps the MPE to monitor the progress of the CPEs with little overhead.

Another interesting feature of the SW26010 is the introduction of user-controlled scratch pad memory. While the MPE is equipped with 32KB/256KB hardware-controlled L1/L2 data cache, the CPEs are cacheless. Instead, an on-chip 64KB scratch pad memory, dubbed as "Local Data Memory" (the LDM), is attached to each CPE. The CPE can use both the main memory and the LDM in computing, but the LDM enjoys a much higher bandwidth and much lower latency. In order to get the best performance, the application has to move data explicitly between the LDM and the main memory and use only the LDM as working memory. To assist the move, SW26010 provides DMA mechanisms which can move data between the LDM and the main memory asynchronously.

Manually overlapping communications with computations, together with explicitly managing the transfer between the LDM and the main memory, requires significant coding effort for general applications, and even more effort is required to make good use of these features. The expense of this motivates us to explore alternative porting options based on asynchronous many task runtimes such as Uintah.

### *B. The MPI+athread programming interface*

Sunway supports MPI for inter-node communications. On a single computing node, OpenACC is supported to allow the offload of computations to the cluster of CPEs (see [16]). However, the Sunway OpenACC interface does not expose all the features of SW26010 and the current implementation does not support OpenACC runtime functions such as `acc_async_test`. For this reason a more low-level atheads interface is used here.

The use of athreads provides a low-level offloading interface for the cluster of CPEs. Conceptually, an athread is a light-weight thread binding to one CPE. The athread library provides mechanisms for the MPE to create a group of athreads running a given function. APIs are provided for the offloaded function to transfer data asynchronously between the LDM and the main memory through DMA operations, as well as to manage the environment on CPEs. athread provides enough freedom for the runtime system to take full control over the data transfer, the execution, and the scheduling.

## V. ADAPTING UINTAH TO SUNWAY TAIHULIGHT

This section describes how Uintah was adapted to Sunway TaihuLight. After describing how Uintah was ported,

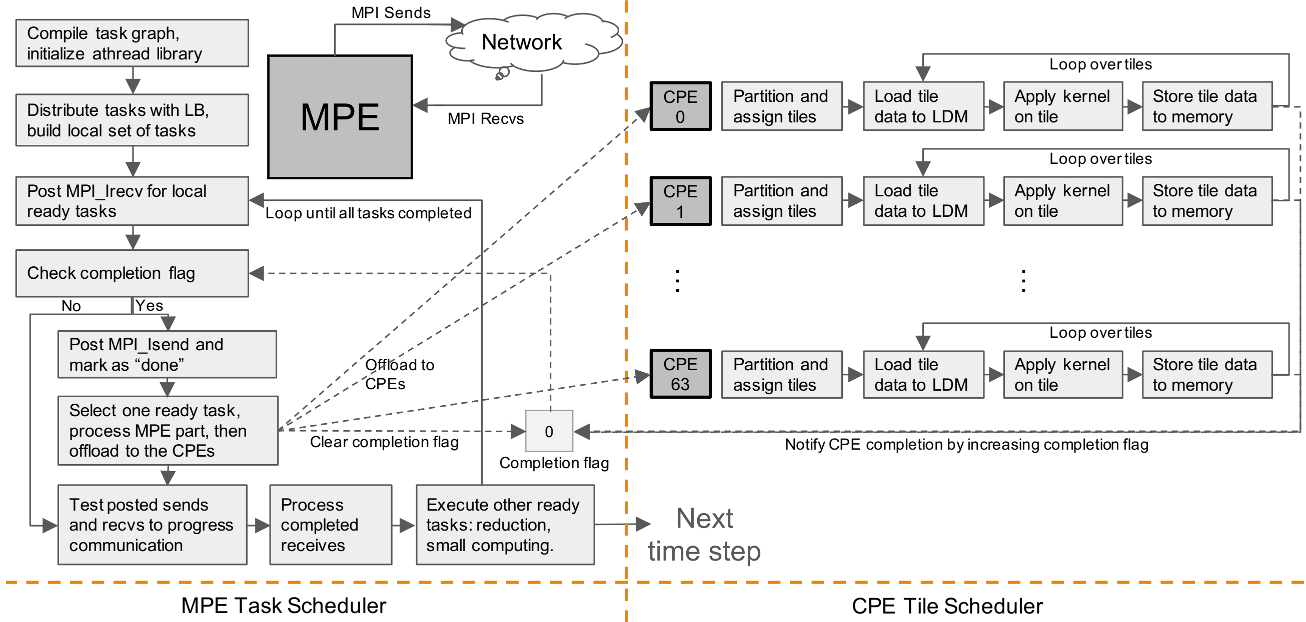


Fig. 4: Uintah's asynchronous scheduler for Sunway TaihuLight

the design of an asynchronous Sunway-specific scheduler is described.

### *A. Porting Uintah to Sunway*

As introduced in Section II, Uintah is a complex framework written in C++ , and uses a variety of C++11 features. Some Uintah components such as Arches include kernels written in Fortran. In order to port Uintah to Sunway, the following steps were taken:

1) Uintah was ported to the MPE using Sunway's GNU compiler since C++11 is needed;
2) Uintah's build system was modified to support compiling C and Fortran for the CPE using Sunway's native compiler for full CPE functionalities such as SIMD;
3) Support was provided for hybrid linking of the MPE and the CPE code together with the MPI, OpenACC, and athread runtime.

Steps 1 and 2 were straightforward. However the complex toolchain needed caused significant problems in Step 3. Since a hybrid toolchain is used, dependent libraries of the toolchain can conflict with each other. The solution was to remove libc and libgfortran to resolve the conflicits, which then excluded all Uintah's gfortran-compiled components.

### *B. Design of an asynchronous Sunway-specific scheduler*

In order to make the best use of the Sunway's architectural features, a Sunway-specific task scheduler was introduced in Uintah. The design of this scheduler originated from the need to be able to:

- Adapt to the shared-memory MPI+CPE heterogeneous architecture and reduce scheduling cost whenever possible;
- Improve kernel performance with the per-CPE scratch pad memory.

Use of Sunway's offload-based programming model made it possible to adapt to the shared-memory MPE+CPE architecture. All tasks except compute-intensive kernels run on the MPE, including main control, task management, and communications. Computing-intensive kernels are offloaded to the cluster of CPEs using athread. The offload is light-weight due to the shared-memory design of Sunway. To further overlap computation with other tasks, an asynchronous offloading mechanism is introduced in the scheduler. The scheduler sets up a completion flag in the main memory just before offloading a kernel and then continues with jobs such as communication and reduction tasks, and checks the completion flag at times. The kernel will update the flag when it finishes computing. When the scheduler detects a kernel is finished, it offloads another ready kernel in the same way until all kernels are completed. Given this approach, the scheduler is "asynchronous".

To make use of the per-CPE scratch pad memory, tiling is built into the scheduler. When a kernel is scheduled to run on the CPEs, the patch is further subdivided into 'tiles' like those in TiDA (see [17]). The tile is defined so that the working memory of the kernel fits in the 64KB LDM. The tiles are then assigned evenly to the CPEs before computing starts. In this way, it is possible to take advantage of the LDM without incurring an extra burden on the application side.

The scheduler is implemented with MPI and athreads. One MPI process is started on each CG and uses the athread library to offload a kernel to the cluster of CGs. A small runtime library was implemented to allow the kernel to transfer the

data on the tiles using `athread_get` and `athread_put`, and to atomically update the completion flag using the `faaw` instruction.

The scheduler schedules both tasks on the MPE and tiles on the CPEs, and is depicted in Figure 4. The details of the scheduler follow.

### C. Scheduling tasks on the MPE

The MPE part of the scheduler (the MPE task scheduler) carries out the following steps:

1) Prepare for scheduling, include: compile the task graph, initialize the athread environment, clear task completion flag, etc.
2) Distribute tasks among different computing nodes (or processes) with the help from the load balancer, and build the set of *local* tasks, i.e., tasks belonging to the running process.
3) Schedule tasks until all are finished:
   a) Post non-blocking MPI receive requests for tasks depending on remote data;
   b) If the completion flag is set:
      i) Post non-blocking MPI sends for the completed task and mark the task as "done".
      ii) Select a ready offloadable task, i.e., a numerical kernel whose remote data is received and dependent tasks are finished.
      iii) Process the MPE part of the selected task.
      iv) Clear the completion flag and offload the CPE part of a task to the cluster of CPEs for *asynchronous execution* and return *immediately* here.
   c) Test the posted MPI sends and receives, update the status of the depending tasks for any finished receives.
   d) Execute any other MPE tasks such as MPI reduce tasks and small kernels if they are ready.
4) Check to see if recompilation of task graph, load balancing or regridding is needed as appropriate then when possible continue to next timestep.

The MPE task scheduler overlaps kernel execution on the CPEs with communication and other tasks on the MPE, thus reduces the overall wait time. Testing MPI communication is done because in most MPI implementations, the non-blocking sends and receives do not progress without the help of the host processor (see [18]).

This MPE task scheduler supports two extra operation modes: the MPE-only mode and the synchronous MPE+CPE mode. In the MPE-only mode, the ready task at step 3(b)iv is executed on the MPE without offloading. In the synchronous MPE+CPE mode, at step 3(b)iv, the scheduler spins until the completion flag is set, thus no overlapping of computation and other tasks is possible.

### D. Scheduling tiles on the CPEs

The CPE part of the scheduler (the CPE tile scheduler) carries out the following steps: for each athread (or CPE) in the group:

1) Calculate assigned set of tiles from the patch and the tile shape. In the current implementation, the tiles are assigned to the CPEs by naturally partition the blocks in the $z$ dimension.
2) Loop over the assigned tiles until every tile is done:
   a) Synchronously read the required data field on the tile into the LDM with `athread_get`;
   b) Apply the numerical kernel on the tile in LDM;
   c) Synchronously write back the modified data field to the main memory with `athread_put`;
3) Increase the completion flag by `faaw`.

At the moment, the CPE tile scheduler does not take into account potential load imbalances among tiles, and does not make use of the fact that the memory-LDM transfer can be asynchronous. These issues will be addressed in the future.

## VI. Optimizing the Burgers Kernel on SW26010

While the scheduler takes care of scheduling tasks and tiles, at the applications level it is only necessary to optimize the numerical kernel. This section describes this optimization for the Burgers kernel 1 on the CPEs of SW26010.

### A. Determining the best tile size

The first step is to choose a proper tile size. Since the Burgers kernel requires one layer of ghost cells, larger and regular tiles are necessary to keep the ratio of ghost cells low. Within the 64KB LDM limit, a tile size 16x16x8 is chosen with the $u$ and $u^{new}$ value requires a close to optimal working memory of 41.3KB.

### B. Vectorizing with SIMD intrinsics

The second step is to vectorize the kernel to take advantage of the SIMD pipelines. At the moment, Sunway does not provide automatic vectorization options in its toolchain. Instead, manual vectorization of the kernel with SIMD intrinsics is needed. Vectorization of the Burgers kernel 1 is done in the $x$ direction, i.e., the $i$ loop. The loop is unrolled with a width of 4 since the SIMD width is 4. An illustration of this vectorization is shown in Listing 2 as Fortran code.

**Algorithm 2** Snippet of the Burgers kernel vectorized with SIMD intrinsics

```
VECTOR256 :: v0, v1, v2, v3, v_d2udz2
! d2udz2 = (-2 * u0(i,j,k) +u0(i,j,k-1) &
!           + u0(i,j,k+1)) * (z_dx*z_dx)
v0 = SIMD_CMPLX(-2d0, -2d0, -2d0, -2d0)
call SIMD_LOADU(v1, u0(i,j,k))
call SIMD_LOADU(v2, u0(i,j,k-1))
call SIMD_LOADU(v3, u0(i,j,k+1))
v0 = SIMD_VMAD(v0, v1, v2)
v0 = SIMD_VADDD(v0, v3)
tmp2 = z_dx * z_dx
call SIMD_LOADE(v2, tmp2)
v_d2udz2 = SIMD_VMULD(v0, v2)
```

### C. Accelerating exponential calculations

The Burgers kernel requires 6 exponentials for each cell. Sunway lacks hardware instructions for exponentials and emulates it in software using one of two libraries: one is IEEE-754 conforming and the other is not. As the IEEE conforming library proved to be slow in tests, the fast library was used. While this introduces some inaccuracy it does not greatly impact this benchmark.

## VII. Performance Evaluations

This section validates our effort porting Uintah and evaluates the performance of the Burgers model fluid-flow simulation on Sunway TaihuLight by measuring strong scalability, effectiveness of the asynchronous scheduler, performance with different optimization options, and floating point efficiency.

### A. Experimental settings

The Burgers equation is discretized on a rectangular grid and is run for 10 timesteps for performance evaluation purposes. The grid is partitioned into equally-sized patches for parallelization. One patch is scheduled for execution on one CG at a time. As only access to the Sunway experimental queue at the moment was available, experiments are limited from 1 to 128 CGs (8320 cores). The grid is partitioned into 128 patches with a fixed 8x8x2 patch layout, i.e., 8 patches along the $x$ and the $y$ axis, and 2 patches along the $z$ axis. A full set of patch sizes is chosen to represent typical cases in the following way: starting from the smallest possible patch, double the size in a round-robin way among the $x$ and $y$ axes each time, until a patch size exceeds the data exceeds the memory limit of one CG. As the tile size used is 16x16x8, and 64 CPEs per CG are used, the smallest patch is 16x16x512. The detailed problem settings are presented in Table III. For each problem in Table III, a

TABLE III: Problem settings in the evaluations

| Problem | Patch Size | Grid Size | Mem | Min |
|---|---|---|---|---|
| 16x16x512 | 16x16x512 | 128x128x1024 | 256MB | 1CG |
| 16x32x512 | 16x32x512 | 128x256x1024 | 512MB | 1CG |
| 32x32x512 | 32x32x512 | 256x256x1024 | 1GB | 1CG |
| 32x64x512 | 32x64x512 | 256x512x1024 | 2GB | 1CG |
| 64x64x512* | 64x64x512 | 512x512x1024 | 4GB | 2CGs |
| 64x128x512* | 64x128x512 | 512x1024x1024 | 8GB | 4CGs |
| 128x128x512* | 128x128x512 | 1024x1024x1024 | 16GB | 8CGs |

strong-scalability experiment is run from the smallest possible number of CGs to 128 CGs. The problem size 64x64x512 crashes with memory allocation errors when using 1 CG so more than 1 CG is used in such cases (stared in the table). For each case in the experiments, different variants of the scheduler and optimization combination as defined in Table IV are run. Every variant is compiled with the `-fPIC -O3 -DNDEBUG` flags to ensure performance. To mitigate the instabilities in the machine, each case is repeated multiple times and the best result is selected. The wall time per time step in each experimental case is used as the performance indicator.

TABLE IV: Experimental variants in the evaluations

| Variant | Scheduler Mode | Tiling | Vectorization |
|---|---|---|---|
| host.sync | MPE-only | No | No |
| acc.sync | synchronous MPE+CPE | Yes | No |
| acc_simd.sync | synchronous MPE+CPE | Yes | Yes |
| acc.async | asynchronous MPE+CPE | Yes | No |
| acc_simd.async | asynchronous MPE+CPE | Yes | Yes |

### B. Strong scalability

In evaluating the strong scalability of this simulation the 'host.sync' variant is excluded since it uses only the MPE. The wall time of different variants on different problems are shown in Figure 5 which indicates that this simulation has good strong scalability on all problems sizes, with both the synchronous and the asynchronous scheduler. Strong scalability persists with the vectorized kernel for which the computing time is reduced by half. This suggests that the async scheduler handles the communication and other tasks in a scalable manner. A quantitative analysis of the strong scalability is

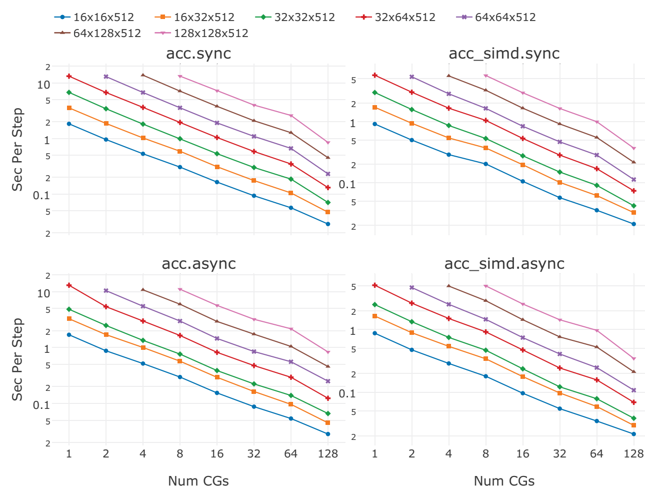


Fig. 5: Wall time of strong scaling different problems

provided by calculating the strong scaling efficiency of each problem from the least to 128 CGs in Table V. This table shows parallel efficiencies from 31.7% to 96.1% for different problems and variants. Especially, for the 'acc_simd.async' variant which is the fastest, a strong scaling efficiency from 31.7% to 57.6% when scaling across two orders of magnitudes the CGs, with small to modest problem sizes is obtained. With large problems, the strong scaling efficiency goes up to at most 89.9% for asynchronous scheduling and 96.1% for the synchronous alternative.

### C. Effectiveness of the asynchronous scheduler

The effectiveness the asynchronous scheduler is calculated using the GFLOP improvement of the scheduler running in asynchronous mode over synchronous mode as $(T_{sync} - T_{async})/T_{async}$.

The performance improvement on the non-vectorized and vectorized variants are presented respectively in Table VI

TABLE V: Strong scaling efficiency of different problems

| Problem | acc.sync | acc.async | simd.sync | simd.async |
|---|---|---|---|---|
| 16x16x512 | 49.7% | 46.8% | 33.7% | 31.7% |
| 16x32x512 | 59.1% | 57.2% | 41.2% | 43.4% |
| 32x32x512 | 75.0% | 57.5% | 55.5% | 50.8% |
| 32x64x512 | 79.3% | 82.5% | 60.6% | 57.6% |
| 64x64x512* | 88.2% | 65.3% | 74.7% | 67.8% |
| 64x128x512* | 95.7% | 73.9% | 80.7% | 72.9% |
| 128x128x512* | 97.7% | 83.1% | 96.1% | 89.9% |

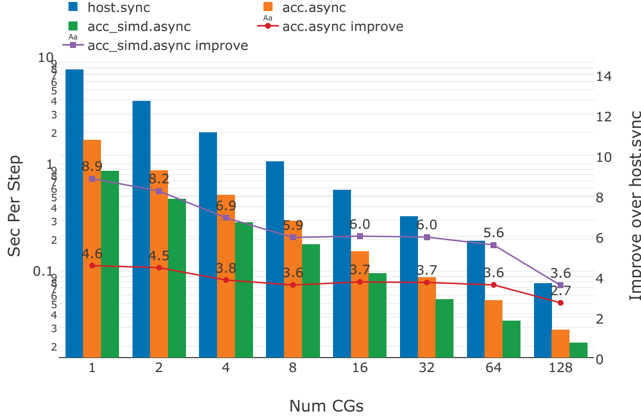


Fig. 6: Performance comparisons of optimizations for the small problem

and VII. The asynchronous mode is a clear winner, as it out-performs the synchronous mode in almost all cases, with an average improvement of 13.5%. The improvement varies for different problems. Medium-sized problems such as 32x32x512 and 32x64x512 get the most improvements. The best improvement is 39.3% for non-vectorized kernel and 22.8% for vectorized kernel. Even with only one CG, performance improvements are still observed. Smaller improvements are seen with the vectorized kernel than the non-vectorized kernel for most of the cases. The results also show that in the 128-CG runs, asynchronous scheduler downgrades the performance in three cases. The cause of this anomaly is under investigation.

### *D. Performance of different optimization options*

Any application port to Sunway must typically go though the following steps: (i) At first port to run on the MPE, then (ii) offload critical kernels to the CPEs, after that (iii) vectorize the kernel, finally (iv) undertake more deep optimizations. The performance gained by the first three steps with the model fluid-flow simulation is now described. These optimization steps are represented by the 'host.sync', 'acc.async', and 'acc_simd.async' variants described above. The trends are illustrated by choosing three typical problems: the small problem 16x16x512, the medium problem 32x64x512, and the large problem 128x128x512. Using 'host.sync' as a baseline, the performance boost of the other variants is calculated as $T_{host}/T_{acc}$.

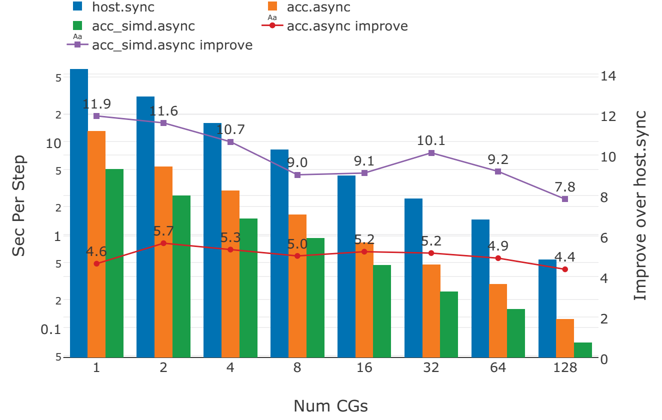


Fig. 7: Performance comparisons of optimizations for the medium problem

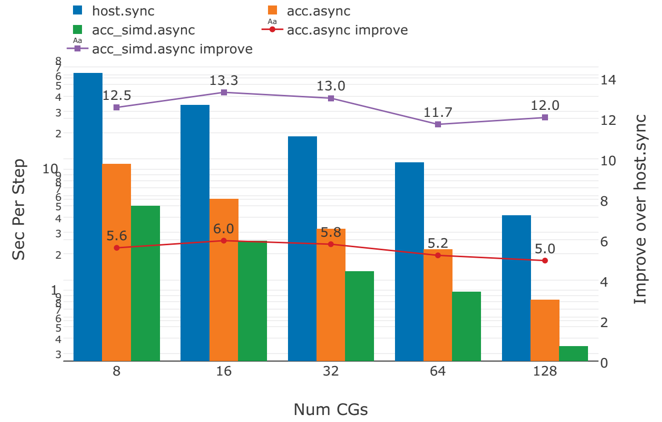


Fig. 8: Performance comparisons of optimizations for the large problem

The results of the three typical problems are shown in Figures 6, 7 and 8. The first observed results is that offloading kernels to the CPEs improve the performance greatly, and vectorizing the kernel improves it further. Offloading kernels provides a 2.7 to 6.0 times performance boost, while vectorization provides another boost of between 1.3 to 2.2 times and gives a final boost of 3.6 to 13.3 times. A higher performance boost is obtained for larger patches. It is not surprising that vectorization is also more effective with larger patches.

### *E. Floating point efficiency*

Now the floating point performance achieved with the asynchronous scheduler is shown with the Burgers model fluid-flow simulation. The 'acc_simd.async' variant is used to represent the best of the options studied here and the floating point performance is calculated as Gflops by $N_{fp}/T_{step} \times 10^{-9}$, where $N_{fp}$ is the floating point operations count per step and $T_{step}$ is the step's wall time. The floating point operations are counted with the performance counters on the CPEs. These counters

TABLE VI: Performance improvement of the asynchronous mode for the non-vectorized kernel

| Num CGs | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|---|
| 16x16x512 | 8.3% | 9.3% | 2.5% | 2.9% | 6.4% | 6.4% | 5.4% | 1.9% |
| 16x32x512 | 8.5% | 9.5% | 1.8% | 3.2% | 5.7% | 6.9% | 8.6% | 4.8% |
| 32x32x512 | 39.3% | 38.0% | 34.9% | 29.7% | 37.3% | 34.5% | 33.7% | 6.8% |
| 32x64x512 | 1.4% | 24.7% | 21.4% | 18.4% | 26.7% | 22.5% | 18.3% | 5.6% |
| 64x64x512 | - | 24.4% | 22.4% | 19.5% | 31.1% | 28.6% | 19.0% | -7.9% |
| 64x128x512 | - | - | 27.5% | 19.2% | 29.1% | 20.0% | 21.3% | -1.5% |
| 128x128x512 | - | - | - | 19.5% | 27.3% | 24.5% | 19.8% | 1.7% |

TABLE VII: Performance improvement of the asynchronous mode for the vectorized kernel

| Num CGs | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|---|
| 16x16x512 | 4.5% | 5.1% | 0.1% | 13.0% | 9.3% | 4.2% | 2.4% | -1.7% |
| 16x32x512 | 4.4% | 5.5% | 0.0% | 8.5% | 11.0% | 4.2% | 4.4% | 9.9% |
| 32x32x512 | 19.5% | 18.2% | 15.3% | 13.0% | 16.7% | 22.8% | 15.2% | 9.4% |
| 32x64x512 | 12.5% | 14.6% | 10.3% | 13.9% | 12.1% | 16.3% | 7.9% | 6.9% |
| 64x64x512 | - | 15.4% | 12.6% | 13.1% | 12.8% | 14.4% | 14.8% | 4.8% |
| 64x128x512 | - | - | 13.0% | 12.9% | 15.5% | 19.8% | 4.5% | 2.1% |
| 128x128x512 | - | - | - | 14.0% | 15.7% | 14.4% | 2.5% | 6.6% |

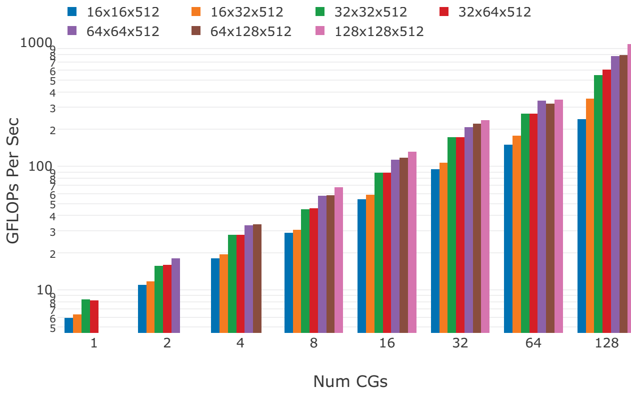


Fig. 9: Floating point performance of different problems

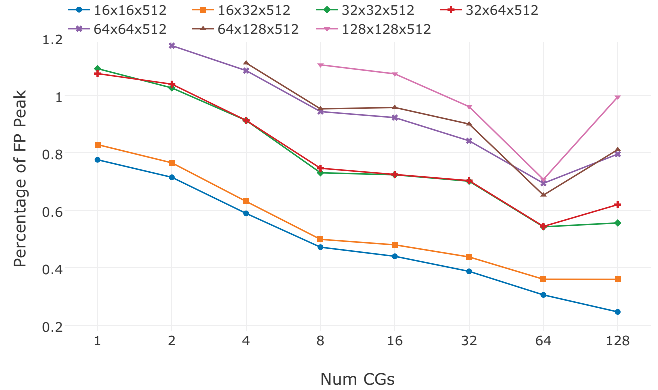


Fig. 10: Floating point efficiency of different problems

are precise on Sunway, but they count division and square root as single operations. Thus while the counter is precise, it does not take into account that divisions requires more cycles, and can under-estimate the floating point efficiency for division-intensive codes.

The floating point performance of different problems is presented in Figure 9. By dividing the observed performance with the theoretical peak of the running CGs, the floating point efficiency in Figure 10 is obtained. The Burgers model simulation reaches 974.545 Gflops with 128 CGs on Sunway, which is 1.0% of the theoretical peak of the 128 CGs. The best performance is 1.17% of the peak obtained by the 64x64x512 problem with 2 CGs. Figure 10 also shows a clear trend that better FP efficiency is obtained with larger problems.

Given the preliminary and primitive nature of the optimizations applied here, further improvement of the floating point efficiency is possible, for example by overlapping computing with memory-LDM transfer and by packing the tiles.

## VIII. Related Work

This research has built heavily upon previous research with Uintah on other modern architectures such as Stampede. Some of the scaling experiments run on Stampede clearly showed need for more sophisticated scheduling and task distribution mechanisms to best utilize architectural features and also to strike balance between computation and communication to hide the communication latency [19]. A number of asynchronous many task runtime systems such as StarPU, Legion, and Charm++ exist and can scale up to hundreds of thousands of cores. These runtime systems support heterogeneous architectures involving CPU, GPU, accelerators and Xeon Phi cores. In Charm++ [6], users create a set of interacting data objects called *chares* (roughly analogous to Uintah tasks). Charm++ provides framework for distributed processing while users need to program to achieve concurrency of shared data resources through proper message structuring. Legion [20] executes concurrent tasks based on the user provided “logical regions” of data. StarPU [7] provides low level multi-task scheduling framework portable across heterogeneous archi-

tectures. However, as far is known, none of these runtime systems is tailored to take advantage of the Sunway TaihuLight architecture at the present time.

## IX. Conclusion and Future Work

The conclusion is that a general runtime system as typified by Uintah may be ported to Sunway TaihuLight through a decomposition of the port into the writing of a modified runtime system with an asynchronous task scheduler based on MPI and athreads and with modified task code tailored to the Sunway CPEs. Experiments show that the design described here obtains good strong scalability with a node range of two orders of magnitude. To the best of the authors knowledge, this appears to be the first attempt to port an asynchronous many task runtime system to Sunway TaihuLight.

Given the preliminary nature of this work, there is much potential for making better use of Sunway's architecture features. For example, the CPEs supports asynchronous DMA transfer between the LDM and the main memory, and so the async scheduler could schedule memory-LDM transfer together with computing kernels to further hide data moving. It is also possible to pack the tiles to improve data transfer performance. It would also be interesting to group CPEs and schedule different patches to different groups, to enable both task and data parallelism on the CGs. With access to more Sunway resources it should be possible to scale beyond 128 nodes of the experimental queue and, building on Uintah's scaling experience with other top ten architectures, possibly up to full capacity of Sunway TaihuLight. The extension of this work to one of the full Uintah applications components would ideally be based on a portability library such as Kokkos as this would automate the loop transformations. As no Sunway option exists in any such libraries at present, the alternative would be a time-consuming but relatively straightforward port based on the knowledge acquired in this research.

## Acknowledgment

The authors would like to thank *National Supercomputing Center in Wuxi* for providing computing resources on Sunway TaihuLight. *Fei Wang, Hongtao You, Qian Xiao, and Xin Liu* from *National Research Center of Parallel Computer Engineering and Technology* provided invaluable support for the porting. Zhang Yang was funded through the National Key R&D Program (No. 2016YFB2021300). The contributions by Damodar Sahasbarude and Martin Berzins are based upon work supported by the National Science Foundation under Grant No. 1337145. Alan Humphrey was funded through the University of Utah.

## References

[1] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang *et al.*, "The Sunway Taihulight Supercomputer: System and Applications," *Science China Information Sciences*, vol. 59, no. 7, Jul. 2016.

[2] F. Qiao, W. Zhao, X. Yin, X. Huang, X. Liu, Q. Shu, G. Wang, Z. Song, X. Li, H. Liu *et al.*, "A highly effective global surface wave numerical simulation with ultra-high resolution," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, pp. 46–56.

[3] J. Zhang, C. Zhou, Y. Wang, L. Ju, Q. Du, X. Chi, D. Xu, D. Chen, Y. Liu, and Z. Liu, "Extreme-scale phase field simulations of coarsening dynamics on the sunway taihulight supercomputer," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 4.

[4] C. Yang, W. Xue, H. Fu, H. You, X. Wang, Y. Ao, F. Liu, L. Gan, P. Xu, L. Wang *et al.*, "10m-core scalable fully-implicit solver for non-hydrostatic atmospheric dynamics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, pp. 57–68.

[5] Q. Meng, A. Humphrey, and M. Berzins, "The Uintah Framework: A Unified Heterogeneous Task Scheduling and Runtime System," in *Proceedings of the International Conference for High Performance Computing, Networking Storage and Analysis*. IEEE Press, Nov. 2012, pp. 2441–2448.

[6] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," *SIGPLAN Not.*, vol. 28, no. 10, pp. 91–108, Oct. 1993. [Online]. Available: http://doi.acm.org/10.1145/167962.165874

[7] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[8] J. Bennett, C. R. *et al.*, "ASC ATDM level 2 milestone #5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms," Sandia National Laboratories, Tech. Rep., 2015. [Online]. Available: http://www.sci.utah.edu/publications/Ben2015c/ATDM-AMT-L2-Final-SAND2015-8312.pdf

[9] B. Peterson, X. Nan, J. Holmen, S. Chaganti, A. Pakki, J. Schmidt, D. Sunderland, A. , Humphrey, and M. Berzins, "Developing uintahs runtime system for forthcoming architectures," SCI Institute, University of Utah, Tech. Rep., 2015.

[10] B. Peterson, A. Humphrey, J. Schmidt, and M. Berzins, "Addressing global data dependencies in heterogeneous asynchronous runtime systems on GPUs," in *3rd International IEEE Workshop on Extreme Scale Programming Models and Middleware*. IEEE Press, 2017.

[11] M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Meng, J. Schmidt, , and C. Wight, "Extending the uintah framework through the petascale modeling of detonation in arrays of high explosive devices," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S101–S122, 2016.

[12] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins, "Applications portability with the uintah DAG-based runtime system on petascale supercomputers," in *Proceedings of the International Conference for High Performance Computing, Networking Storage and Analysis*. IEEE Press, Nov. 2013.

[13] A. Humphrey, D. Sunderland, T. Harman, and M. Berzins, "Radiative heat transfer calculation on 16384 gpus using a reverse monte carlo ray tracing approach with adaptive mesh refinement," in *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 1222–1231.

[14] "The top500 list," 2017. [Online]. Available: https://www.top500.org/lists/2017/06/

[15] J. Dongarra, "Report on the sunway taihulight system," 2016. [Online]. Available: http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf

[16] "Sunway taihulight compiler system user manual (in chinese)," Tech. Rep., 2016. [Online]. Available: http://www.nsccwx.cn/ceshi.php?id=19

[17] D. Unat, T. Nguyen, W. Zhang, M. N. Farooqi, B. Bastem, G. Michelogiannakis, A. Almgren, and J. Shalf, "TiDA: High-Level Programming Abstractions for Data Locality Management," in *High Performance Computing*, J. M. Kunkel, P. Balaji, and J. Dongarra, Eds. Springer International Publishing, 2016, vol. 9697, pp. 116–135.

[18] A. Denis and F. Trahay, "MPI Overlap: Benchmark and Analysis," in *2016 45th International Conference on Parallel Processing (ICPP)*, Aug. 2016, pp. 258–267.

[19] J. K. Holmen, A. Humphrey, D. Sunderland, and M. Berzins, "Improving Uintah's Scalability Through the Use of Portable Kokkos-Based Data Parallel Tasks," in *Proceedings of the PEARC 2017 Conference*. ACM Press, 2017, pp. 1–8.

[20] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2012, p. 66.