Dynamic Task Scheduling for Scalable Parallel AMR in the Uintah Framework

Qingyu Meng, Justin Luitjens, Martin Berzins

UUSCI-2010-001

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

March 10, 2010

Abstract:

Uintah is a computational framework for fluid-structure interaction problems using a combination of adaptive mesh refinement(AMR) and MPM particle methods. Uintah uses domain decomposition and a task graph based approach for asynchronous communication and automatic message combination . The original task scheduler for Uintah ran computational tasks in a predefined order. To improve the performance of Uintah for petascale architecture, a new dynamic task scheduler allow better overlapping of the communications and computations is designed in this study. The new scheduler supports asynchronous, out of order scheduling of computational tasks by putting them in a distributed directed acyclic graph(DAG) and isolating task memory. The effectiveness of this new approach is shown on large scale fluid-structure examples through an analysis of the performance of the software.

Dynamic Task Scheduling for Scalable Parallel AMR in the Uintah Framework

Qingyu Meng, Justin Luitjens, Martin Berzins
{qymeng, luitjens, mb}@cs.utah.edu
School of Computing, University of Utah, Salt Lake City, Utah

May 10, 2010

Abstract

Uintah is a computational framework for fluid-structure interaction problems using a combination of adaptive mesh refinement(AMR) and MPM particle methods. Uintah uses domain decomposition and a task graph based approach for asynchronous communication and automatic message combination . The original task scheduler for Uintah ran computational tasks in a predefined order. To improve the performance of Uintah for petascale architecture, a new dynamic task scheduler allow better overlapping of the communications and computations is designed in this study. The new scheduler supports asynchronous, out of order scheduling of computational tasks by putting them in a distributed directed acyclic graph(DAG) and isolating task memory. The effectiveness of this new approach is shown on large scale fluid-structure examples through an analysis of the performance of the software.

1 Introduction

Most large-scale scientific applications are expressed as a workflow running on a huge domain. In a distributed memory running environment, the whole domain is usually divided into individual data sets for local computation. While the application in general is typically structured as a arbitrary sequence of computational tasks, where each sequence is executed on a different data set. Every task has its own communication and computation requirements: It reads an input form the pervious task, processes the data, and outputs a result to next task. Initial data are input to the first task and finial results are obtained as the output from the last task. The pipeline operates in synchronous mode: After some initialization delay, a new task on each data set is completed every period. The period is determined by the longest execution time of all synchronized tasks across the whole domain.

In the mean time, the data sets and therefore the task execution time may change during the simulation process. This is even more likely to happen in applications using new generation techniques such as adaptive mesh refinements(AMR) [3]. As a result, when the execution time of synchronized tasks become very diverse, the task wait times in a period are increasing. Hence, the application's performance and parallel efficiency may suffer.

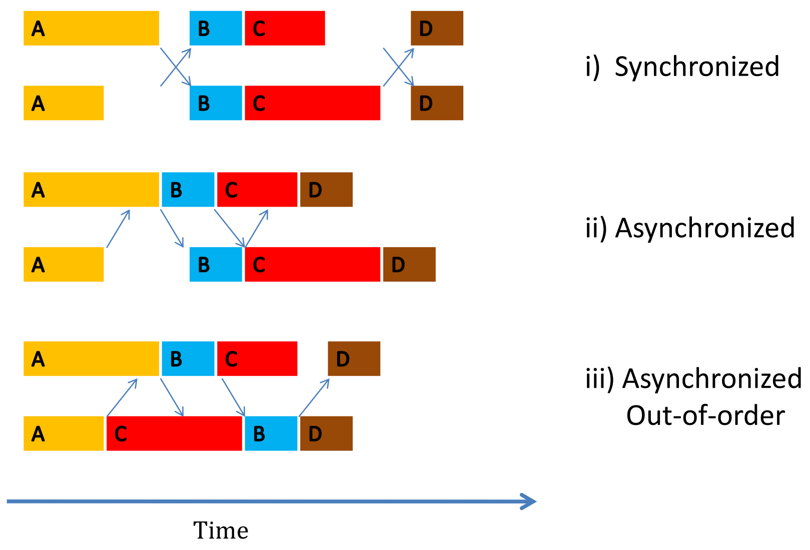


Figure 1: Workflows running on two data sets. Task B and C do not have data dependencies .

Figure. 1 illustrates workflows running on two data sets, each data set is distributed to different processor and has four tasks A, B,C and D working run on it. Using asynchronous communication can certainly help the application to hide data transmit time when it is doing computations. But asynchronous communication will not save the waiting time if there are data dependencies. In this example, task B must wait for task A to complete, as it needs task A's result. This part of time is usually counted into MPI waiting time, but it can be reduced by moving later task to fill that wait time, as it is shown on part (iii).

In this paper, we attempt to design a dynamic task scheduling mechanism in Uintah [4, 10, 13, 14] by allowing the tasks to run not in a sequential order as they are in the algorithm, but out of order according to the runtime information. As Uintah is a general computational framework, it supports various tasks which may have asynchronous communications to different neighbors, write global variables, or even call third party libraries such as PESTc. The dynamic scheduler must be robust enough to grantee all kinds of these tasks are processing correct result.

We accomplished this by putting fin-grained computational tasks in a directed acyclic graph(DAG) and isolating the task memory. To achieve high scalability, we use a decentralized scheduling scheme for distributed memory system. That is, each node schedules its tasks privately and communicates with other nodes regarding data dependencies only when necessary. Further more, Uintah's scheduler respects task priorities and supports scheduling global synchronization tasks. To create as many independent tasks as possible to prevent processors from becoming idle, we allow multiple versions of memory by adding a variable version table. This can help the system to remove certain task dependencies and generate more independent tasks.

2 Uintah

Uintah is designed by the Center for Accidental Fires and Explosions (C-SAFE) [4], originally for simulation of fires and explosions and later became a multi-physics computational framework. The primary objective of Uintah is to provide a software system in which fundamental chemistry and engineering physics are fully coupled with nonlinear solvers and visualization tools. The framework is build upon a set of parallel software components and libraries using the DOE Common Component Architecture (CCA) that facilitate the solution of partial differential equations (PDEs) on structured AMR grids.

The simulation scenario is that a metal container filled with plastic bonded explosive(PBX) embedded in large hydrocarbon fires explosives. Simulating this problem requires expertise from a wide variety of disciplines including combustion, structural mechanics, and fluid dynamics. In this simulation, Uintah uses ICE algorithm [1, 6, 7]to simulate flows and MPM [16] algorithm to simulate solids. MPM (solids) and ICE (fluids) exchange data several times per timestep, which are not limited to boundary condition exchange.

ICE is a multi-material CFD algorithm that was developed by Kashiwa and others at LANL. This technique can be used in both incompressible and compressible ow regimes, which is necessary when modeling fires and explosions. The cell centered, finite volume solution technique is convenient in that a single control volume is used for all materials. Conserving mass, momentum and energy, and the exchange of these quantities between the materials is simplified by use of a common control volume.

The Material Point Method(MPM) is a particle method that is used to evolve the equations of motion for the solid materials. MPM is a powerful technique for computational solid mechanics, and has found favor in many applications involving complex geometries, large deformations, and fracture. Originally described by Sulsky, et al., MPM is an extension to solid mechanics of FLIP, which is a particle-in-cell (PIC) method for fluid flow simulation.

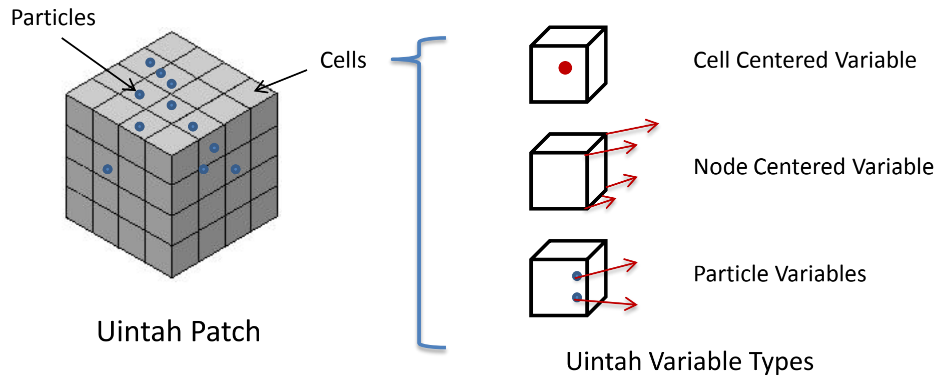


Figure 2: An Uintah patch contains 4x4x4 cells and several particles

Such a problem requires a large amount of processing power necessitating the need for both parallelism and adaptive mesh refinement(AMR).Uintah achieves parallelism by dividing the grid into hexahedral mesh patches, which are uniquely assigned to processing processors. Figure 2 shows an Uintah patches which contains 64 cells. Each cell owns several types of variables: i)node centered variables, such as as velocity, mass, volume, and temperature; ii)cell centered variables, such as density, internal energy, momentum; iii) particles in cell, which also has their own variables like mass, volume, temperature, and velocity. All these variables are stored in ondemand datawarehouse, a directory based hash map. Each variable is indexed by name, type, and patch id it belongs.

AMR focuses the computational resources where needed by adding refinement in areas where rapidly evolving physical processes are occurring. Uintah currently contains three main simulation components released together with the framework, that are capable of using AMR: i) the ICE compressible multi-material CFD formulation, ii) the particle-based Material Point Methodfor structural mechanics, and iii) the combined fuid-structure interaction algorithm MPMICE. In addition, Uintah integrates numerous sub-components including equations of state, constitutive models, and reaction models.

Uintah programming model is based on a far-sighted design which complete separate of the user code and parallelism. This allows the aspects of parallelism such as schedulers, load-balancers, grid refinement, parallel input/output, checkpoint and restart, to operate independently of the simulation code. The scientists are concerned only on their area of expertise, working on on the simulation components design without fully understanding complexities outside of their domain. This has led to a highly flexible simulation package which has been able to simulate a wide variety of problems including shape charges, stage-separation in rockets, the biomechanics of microvessels , the properties of foam under large deformation, and the evolution of large pool fires caused by transportation accidents, in addition to the exploding container scenario described above.

Meanwhile, experts from computer science can focused on parallel infrastructure and are given the maximum possible freedom to improve the performance and scalability. The component design can also help us to design and test new infrastructure components. For example, we can easily integrate Zoltan as our new load-balancers and compare its performance to Uintah's buildin load balancing algorithm. Recently, a new regrider is also implemented for Uintah framework with high efficiency refinement algorithm. This enabled us to design a new dynamic task scheduler for Uintah framework without changing too much on other components.

3 Background

In moving Uintah to petascale machines, it was initially observed that there was a substantial increase in MPI communication time at larger numbers of cores. The time spent waiting for communication comes from the dependencies between computing tasks distributed to different processors. This wait time is a combination of time spent waiting for data to be computed on other task and time spent waiting for the data transmit through the network. In Uintah, the scheduler is responsible for computing the dependencies of tasks, determining the order of execution and ensuring that the correct inter-process communication via MPI is made when necessary.

3.1 Uintah Task Scheduler Design

Uintah uses a call back task based design. To understand how it works, Figure3 shows part of the pseudocode that implements ICE algorithm. The subroutines of scheduleComputePressure, scheduleComputeTempFC, etc will generate tasks by defining input variables, output variables and call back functions, then add them to the scheduler. Scheduler will then determine the task execution order.

```
if(d_turbulence)
    d_turbulence->scheduleComputeVariance( sched, patches,ice_matls);
  scheduleMaxMach_on_Lodi_BC_Faces(        sched, level,  ice_matls);
  scheduleComputeThermoTransportProperties(sched, level,  ice_matls);
  scheduleComputePressure(                 sched, patches,d_press_matl,..);
  scheduleComputeTempFC(                   sched, patches,ice_matls_sub,..);
  scheduleComputeModelSources(             sched, level,  all_matls);
  scheduleUpdateVolumeFraction(            sched, level,  d_press_matl,..);
  scheduleComputeVel_FC(                   sched, patches,ice_matls_sub,..);
  scheduleAddExchangeContributionToFCVel(  sched, patches,ice_matls_sub,..);
```

Figure 3: ICE algorithm written with the task-based subroutines

The original task scheduler in Uintah use asynchronous MPI communications and combine messages which have the same source and destination. These techniques can certainly overlap some communications and computations and reduce the data transmit time through the network. For example, the processor can move to next task while sending messages after finished a task, but it must wait for the messages arrival before running a task as the computation can not start without the data coming with that messages. Original Uintah task scheduler generate a task graph to statically analysis task dependencies and combine MPI messages. The task graph is a directed acyclic graph where each node in the represents a task. Directed edges are used to represent a data dependency or MPI communications. After the static analysis, the task execution order is determined, the scheduler will run tasks based on this order.

If all tasks in the same period cost same amount time to execute, there will not likely major waiting time for data to be computed will take place, because all the data are globally ready at the same time when the whole simulation process are synchronized. But Uintah has different situations: i) Uintah supports AMR, the workloads for different patches may not equal, ii) Particles are moving from a cell to another cell during the simulation, task workload with particle variable is not stable. These cause the time spent waiting for data ready is majority of Uintah's MPI Wait. Measurements show that this type of wait cost as high as 80 percent of total MPI waiting time in Uintah.

To reduce the task wait time and further improve the performance of Uitah simulations, we are researching into better scheduling algorithm which can dynamically execute tasks.

3.2 Similar DAG Design

DARPA released a software public report in 2009, which proposed a sliver model for Exascale architectures. The sliver mode suggests that a Exascale software should provide an abstraction of parallel computation that exposes and exploits a high degree of algorithm concurrency, particularly that available from dynamic directed graph structure based applications.

Most of work that use DAG based scheduling has a global view of task graph. The runtime system map the tasks to multiple threads on shared-memory systems, such as Cick [2], or to multiple nodes through migration such as Charm++ [5] on distributed memory systems. Cick is a multi-threaded parallel programming language for SMP. It schedules tasks by using a provably good work-stealing algorithm on the task graph. Charm++ has a global object graph contains numbers of medium-grained processes which interact with each other via messages. The runtime system of Charm++ will map these medium-grained processes to appropriate processors to balance the load by migrating the data.

PLASMA [12] is a new parallel linear algebra library which also represent its algorithm as DAG . Their programming model enforces asynchronous, out of order scheduling of operations. The current PLASMA release is scheduled statically with a trade off between load balancing and data reuse. TBLAS [15] is another task based parallel linear algebra library. It uses a dynamic scheduler with decentralized task graph to archive high scalability.

4 Distributed Task Graph

One of the primary goals of Uintah is that the scientists can develop large scale parallel simulation components with limit understanding of the underlying parallelism. The simulation component programmer implements an algorithm within the Uintah framework by specifying the algorithm as a serial of tasks which run on a hexahedral mesh patch. Uintah infrastructure analyzes the components and automatically enables mesh refinement and load balancing, then uses distributed task graph for scheduling and communications.

4.1 Tasks

To create a Uintah task, the programmer needs specify a call back function where the computation to be performed, variables which are required for the computation and variables it computes. Flowing example equation shows the algorithm of the fourth stage of ICE, explicit pressure, single material, reaction model simulation. This equation is to compute face-centered velocity.

$\vec{U}^{*f} = f(\Delta t, P_{eq}, \vec{g}, \rho, \vec{U})$

By specifying a task name (`ICE::computeVel_FC`) and a call back function pointer (`&ICE::computeVel_FC`), an Uintah task can be created as below.

```
t = scinew Task("ICE::computeVel_FC",
           this, &ICE::computeVel_FC);
```

Then, the input variables need to be added to the task. In this example algorithm, the requirements of `ICE::computeVel_FC` include following variables

1. Δt : $delT$ global variable from last timestep
2. P_{eq} : $press\_equil\_CC$ cell centered variable from current timestep
3. $\vec{g}$: sp_vol_CC cell centered variable from current timestep
4. ρ: rho_CC cell centered variable from current timestep
5. $\vec{U}$: vel_CC cell centered variable from last timestep

Where Δt is per level global variable. P_{eq}, $\vec{g}$, ρ, $\vec{U}$ need one extra cell data from neighbor cells.

Variables in Uintah have various types. Just like integer and double are distinct types, the variables has distinctly different types such as FaceCenter, CellCenter, Global. During the simulation, variables are stored in a dictionary data structure called DataWarehouse. Variables that from a previous timestep are stored in the Old DataWarehouse, called OldDW, and variables that are computed in current timestep are stored in the New DataWarehouse, called NewDW. At the end of each timestep, the variables in NewDW are mapped to the OldDW for the next timestep in the simulation and a new NewDW is initialized. The detail of data warehouse will be introduced later. That is to say, variables from last timestep should be queried from OldDW; variables from current timestep should be queried from NewDW. In this example, the requirements for task `ICE::computeVel_FC` can be set up as:

```
Ghost::GhostType  gac = Ghost::AroundCells;
Task::DomainSpec oims = Task::OutOfDomain;  //outside of ice matlSet.
t->requires(Task::OldDW, lb->delTLabel, getLevel(patches));
t->requires(Task::NewDW, lb->press_equil_CCLabel, press_matl, oims, gac,1);
t->requires(Task::NewDW, lb->sp_vol_CCLabel,    /*all_matls*/ gac,1);
t->requires(Task::NewDW, lb->rho_CCLabel,       /*all_matls*/ gac,1);
t->requires(Task::OldDW, lb->vel_CCLabel,         ice_matls,  gac,1);
```

From the algorithm, this task compute $\vec{U}^{*^f}$ on all three faces of the cell: $uvel\_FC$, $vvel\_FC$, $wvel\_FC$. They are all face centered variables. All output variables can only be stored in NewDW.

```
t->computes(lb->uvel_FCLabel);
t->computes(lb->vvel_FCLabel);
t->computes(lb->wvel_FCLabel);
```

Finally, the task is added to the scheduler component with specifications about what patches and materials are associated with the actual computation.

```
sched->addTask(t, patches, all_matls);
```

For more complex problems involving multiple materials and multi-physics calculations, a subset of the materials may only be used in the calculation of particular tasks. The Uintah framework allows for the independent scheduling and computation of variables associated with an individual material within a multi-physics calculation.

4.2 Patch Assignment

Uintah's grid is divided to numbers of small patches during the regriding process. As the simulation progresses, individual grid cells are tagged for refinement, using some criterions. The regridder will take flags, and, wherever there are refinement flags, patches are constructed around them on a finer level. If there are relatively large patches being constructed, the regridder also needs to divide them into smaller patches for easier distribution and scheduling.

After regridding, these patches are partitioned and assigned to different processing resources according to load balance algorithm. Uintah's load balancer [9, 11]determines a reasonable allocation of patches to nodes using measurement and geography information. The load balancer attempts to guarantee that an equal amount of works is distributed to each processor allowing for optimal scaling of the simulation to multiple processors. The weight for each patch is predicted through certain criteria, such as history weights, number of particles, number of cells, etc. In additional to reducing the communication cost, the load balancing algorithm clusters neighboring patches together as communication in predominantly local that only a small area around each patch must be communicated.

4.3 Generate Detailed Tasks

In Uintah framework, processes running on different nodes execute the same program and load the same simulation component. Each patch will create it own instance of task called *detailed* task. Suppose an Uintah component designed M tasks, and there are total N patches in the grid, total $M \times N$ detailed tasks will be created globally. It is not trivial to generate a centralized directed acyclic graph(DAG) by creating one edge per dependencies between detailed tasks.

4.3.1 Centralized Version

$Definition$ 1. A centralized task graph is a two-tuple $G_{Global} =< T_g, D_g >$, where T_g is a set of nodes and D_g is a set of direct edges. Each node $t_i \in T_g$ is a detailed task associated to a patch in the global mesh and a task. These is an edge $d < t_i, t_j > \in D_g$ if there is a dependency that t_i need to be executed before t_j.

The complexity to create a centralized task graph will be nearly $O(|T_g| \log |T_g|)$. Since the number of task on a patch M is a constant, the complexity can be written as $O(N \log N)$. There will be thousands to millions of patches created in global depending on what problem size we are running. A centralized version of task graph will not scale on large simulations with high resolutions. Therefore, Uintah uses distributed algorithm to generate task graphs.

4.3.2 Distributed Version

After patches assigned to processors, each processor creates its own and neighbors' instances of tasks. The neighbors' detailed tasks are created here only for dependencies computation and will not actually run after. Suppose the number of processors is P, each processor approximately has N/P local patches.

$Definition$ 2. A distributed task graph is a two-tuple $G_{Global} =< T_l \bigcap T_n, D_d >$, where T_l is a set of local detailed tasks and T_n is a set of neighbor detailed tasks. Each node $t_i \in T_l$ is a detailed task associated to a local patch. Each node $t_i \in T_n$ is a detailed task associated to a patch in its neighborhood. These is an edge $d < t_i, t_j > \in D_d$ if t_i need to be executed before t_j and $t_i \in T_l$ or $t_j \in T_l$.

The complexity to create a distributed task graph in Definition 2 will be nearly $O(|T_l| \log |T_l + T_d|) = O(\frac{N}{P} \log \frac{N^2}{p})$. The distributed version of task graph can scale if we use large number of processors.

These detailed tasks contains all the necessary information for scheduler to analyze data dependencies and execute the tasks on a single processor. Figure 4 shows the data structure of a detail task in Uintah scheduling system.

A detailed task contains following information:

- Patch: The patch current detailed task will process. Assigned by load balancer.
- Task-related information such as task name, task type, call back function.
- Input: Variables required for computation in this task. Variables may come from its own patch or neighbor patches.
- Output: Variables computed from this task. Must write to its own patch.

After task graph is compiled, detailed task also contains:

```
/* Task information*/
Task* task; //Original task
PatchSubset* patches; //Patches running on
list reqs; //input variables
list comp; //output variables

/*Dependency Information*/
list internalDependencies;
int externalDependencyCount;

/*Runtime Information*/
bool initiated;
bool externallyReady;
```

Figure 4: Data structure of detailed task

- Internal dependency pointer: Link to tasks that require variables from this task.
- External dependencies counter: number of MPI messages need to be received from other processor.

During the run time, there are also some fields for task status such as Internal Ready, External Ready, Running and Completed.

By using this design, computing patches and variables are not owned by individual tasks. They are stored in OnDemand DataWarehouse, a directory based data structure, so DataWarehouse can do the allocation and deallocation work automatically. Also there are no MPI calls inside tasks. All the MPI communication buffers are also handled automatically by DataWarehouse. A detailed task is essentially a runtime object for a task on a specific patch, and the smallest schedulable unit in Unitah.

4.4 Task Dependency

As the simulation component programmer writes tasks sequentially and does not explicitly define dependencies between tasks, to ensure the task will run in a correct order, the Uintah's scheduler will automatically detected these dependencies.

If there exists a data dependence, the scheduler can determine which task precedes another. There are two types of dependencies in Uintah framework: internal dependencies and external dependencies. Internal dependencies are between patches on the same processor and external dependencies are between patches on different processors. Thus internal dependencies imply a necessary order where external dependencies also specify necessary communication.

4.4.1 Internal Dependency

Figure5 illustrates how to detect a RAW(read after write), WAR(write after read) and WAW(write after write) dependencies based on the task inputs and outputs. The Uintah task always has the input and output variables defined through requires and computes function. Therefore the scheduler can go through all the detailed tasks to match the patch and variables information. Whenever two tasks access the same variable in the same patch, the scheduler detects a data dependency and update the detailed tasks to put an dependency link between them. Since WAR and WAW dependencies can be removed by renaming, we only consider the true dependencies.

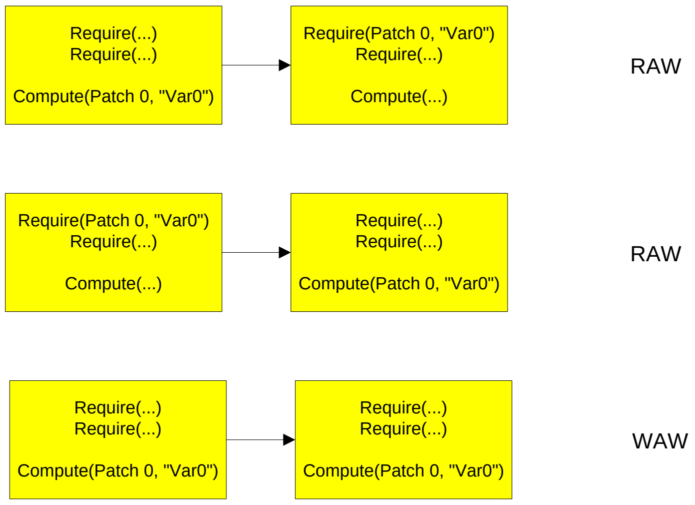


Figure 5: Detecting internal data dependencies

4.4.2 External Dependency

External dependency always comes from input variables with ghost cell requirement, a task may require the variable from one or two additional layers of cells around its patch. Figure6 shows two patches are assigned to two different processors, the task on patch 1 requires one additional lay of ghost cells which are on patch 0. Since all detailed tasks of neighbors are also created, whenever a task requires ghost cells, we can always find the corresponding origin task which computes that variable. If the origin task has been assigned to the same processor, an internal dependency will be added, otherwise an external dependency batch object will be created. The external dependency batch objects will later be used for MPI messages combination and tag assignment.

Since the external dependencies of detailed tasks are computed in distributed environment, each node only computes its own side of sends and receives. The Uintah's distributed task graph can guarantee that those sends and receives will match each other without additional communications.

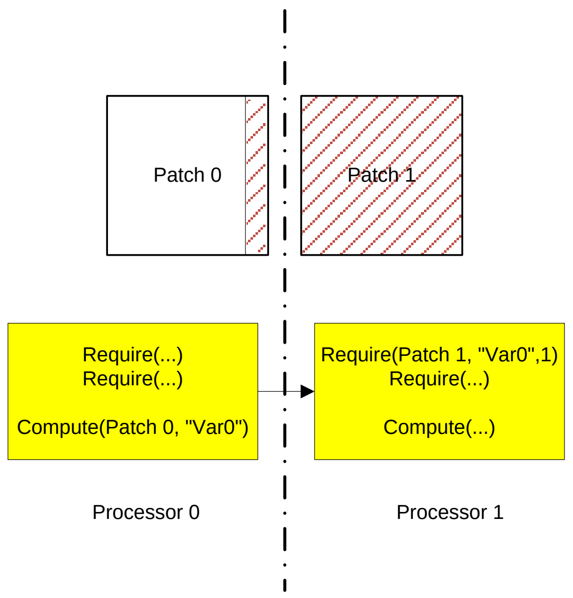


Figure 6: Detecting external data dependency

4.5 Task graph compile

With all tasks and data dependencies detected, each processor creates a distributed directed acyclic graph(DAG) by creating one edge per variable dependency between tasks. For example, in the $computeVel\_FC$ task of pervious example:

1. $delT$: computed from task $ComputeStableTimestep$ of pervious timestep
2. $press\_equil\_CC$: computed from task $ComputePressure$
3. sp_vol_CC: computed from task $ComputePressure$
4. rho_CC: computed from task $ComputePressure$
5. vel_CC: computed from task $ConservedtoPrimitive_Vars$ of pervious timestep

An initial graph is actually generated once we processed all data dependencies. But this graph (shown in Figure 7 middle) has a lot of redundant dependencies. A task graph needs to be simplified due to it will be executed many times. The three requirements of $computeVel\_FC$ task, $press\_equil\_CC$, sp_vol_CC, rho_CC edges can be combined, as they are all dependencies from task $ComputePressure$ to $computeVel\_FC$ and any of them can enforce a order that task $ComputePressure$ must be executed before task $computeVel\_FC$.

A special system task called $CopyOlddata$ will be generated by Unitah infrastructure to prepare old DataWarehouse variables. As a result, all dependencies from pervious time step will be replaced by dependencies from $CopyOlddata$ task.

Dependency is also removed if it can be recursively represented by other dependencies. In this example, the sp_vol_cc edge from task $ComputePressure$ to $ComputeContributionToFC$ is removed, due to $ComputePressure$ is already a predecessor of $ComputeContributionToFC$ via task $computeVel\_FC$.

Figure 7 right shows part of the compiled task graph.

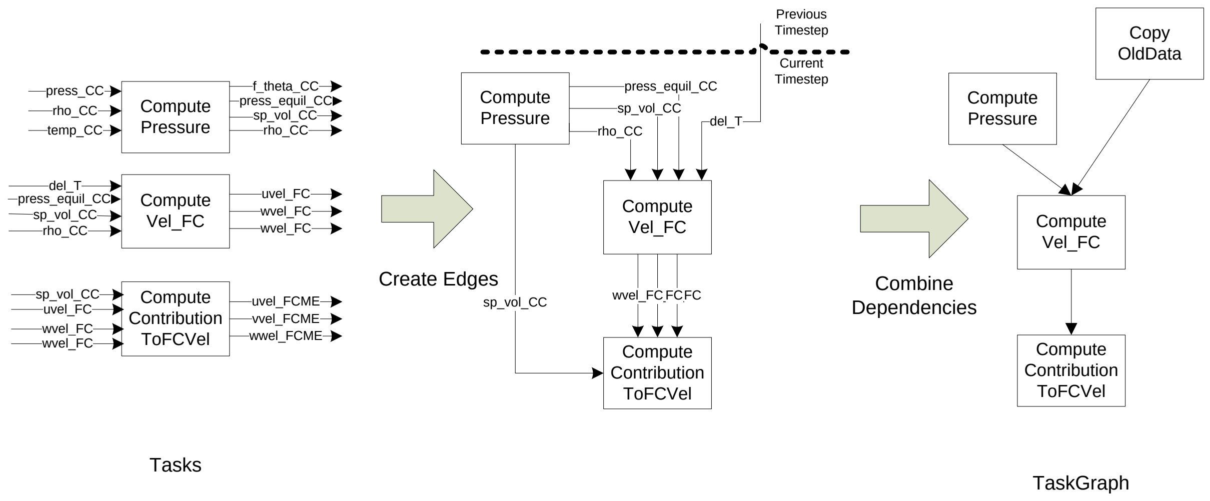


Figure 7: Task graph Compiling

The external dependencies are also combined if they will send to a same destination detailed task. MPI message tags will be assigned after message combination. Each detail task will then initial an external dependency counter to trace the outstanding MPI messages. Figure 8 shows a example of task MPI communications and the external dependency counter. After a task graph is compiled, the scheduler will continues to execute the same task graph on each time step until the grid is refined or the simulation component decides the task list is no longer valid.

4.6 Ondemand DataWarehouse

All the Uintah variables are stored in a distributed dictionary data structure called Ondemand DataWarehouse with the illusion that all memory is global. The DataWarehouse is an abstraction of a global single-assignment memory, with automatic data lifetime management and storage reclamation. The dictionary uses three elements to index a variable: variable name, variable type and patch id. The Ondemand DataWarehouse not only contains local patch variables but also contains foreign variables from other processors. A task can read the variables from all local and foreign patches but can only write to its own patch. In this way, a task is limited to work on its own memory. If task set up its input and output variables correctly, the variables of related patches will be ready in DataWarehouse for read and write when the task is being scheduled. In addition, DataWarehouse will also track the life span of all variables, so it can clean up the variable memory if no local tasks are going to use it.

5 Dynamic Runtime System Design

Results from preliminary scaling studies on petascale machines such as Kraken showed that there was a substantial increase in MPI communication time at larger numbers of cores. We discovered that the

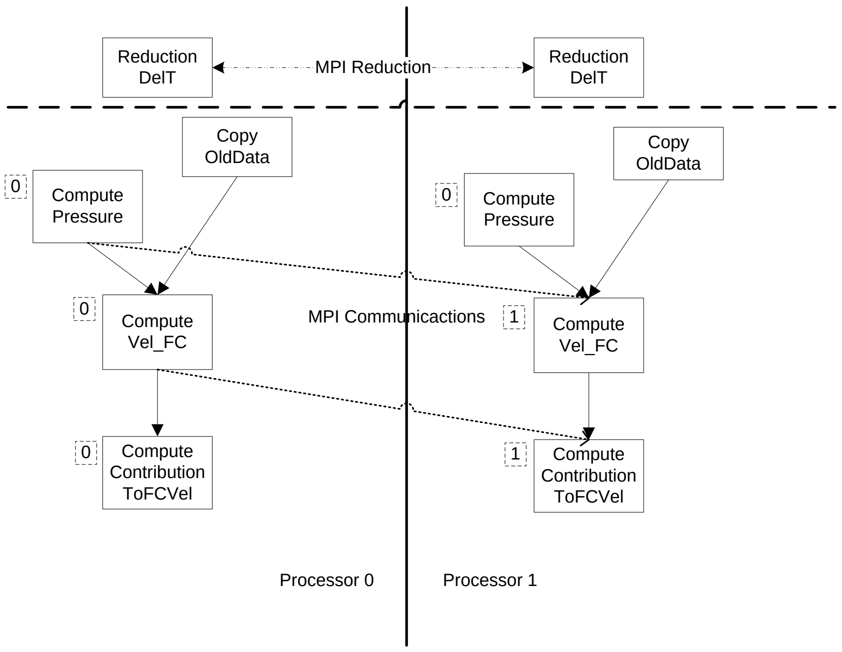


Figure 8: Task communications and external dependency counters

time spent waiting for communication is due to dependencies between computing tasks distributed to different processors.

Originally, Uintah used a static scheduler in which tasks were executed in a pre-determined order. Figure 9 illuminates the original static scheduler architecture. After statically analysis of the task graph, a sorted task list is created. The scheduler then execute tasks from this sorted list. A limitation of static scheduling is that a single task waiting for messages will cause the whole the simulation to sit idle. Measurements showed that this type of wait cost nearly 80 percent of total MPI waiting time in Uintah. The new scheduler solved this problem by dynamically determine the order during the execution to overlap communication and computation. The architecture of runtime system are improved to support the out-of-order execution.

5.1 Tasks Ready Queues

Comparing to Uintah's static scheduler, the new dynamic scheduler has two task queues(Figure 10): the internal ready queue and external ready queue. After task graph is compiled, all the pre-satisfied tasks will be placed in the internal ready queue. The value of counter for tracking outstanding MPI messages is set according to the task graph. When this counter reaches zero, the communication is complete and the task is ready to be executed. At that point it is placed in the external ready queue. When scheduling a task the scheduler chooses a task in the external ready queue based on a priority algorithm.

When a task is completed, the task graph will check if there are any task's all local dependencies are satisfied. New satisfied tasks will also be putted in the internal ready queue and have their external

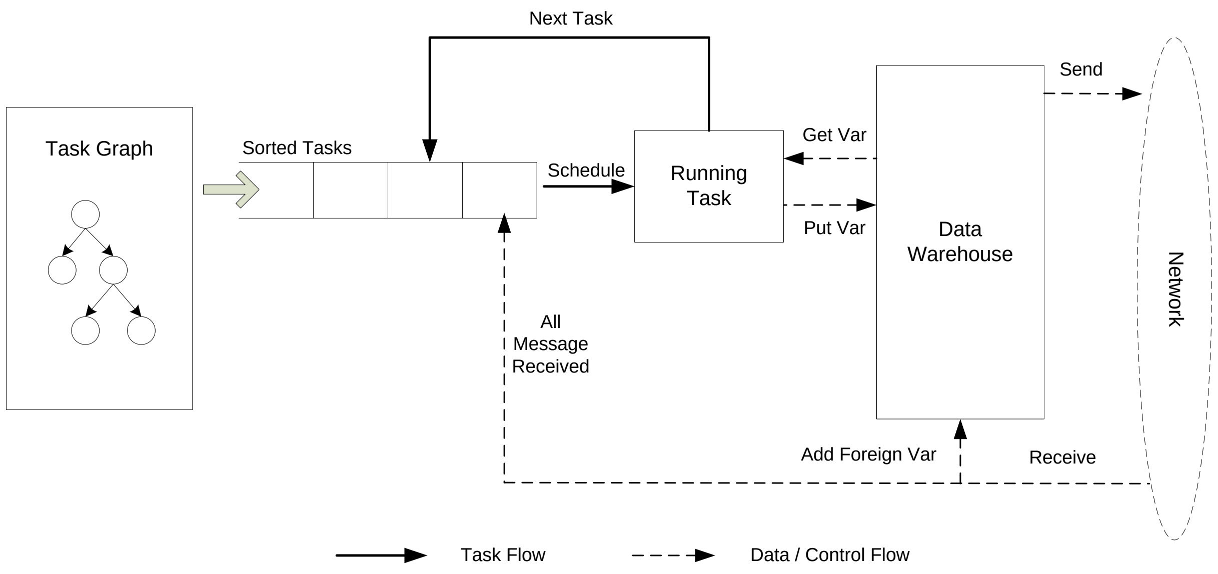


Figure 9: Architecture of Uintah static task scheduling system

dependency initialed. Due to the new scheduler now allows multiple tasks waiting for communications at the same time, a task can be executed when another tasks are receiving foreign variables. To prevent accessing uncompleted foreign variables, a foreign variable now need to set to valid before it can be accessed by tasks.

When the scheduler begins to run, task will be first waiting all internal dependencies and then waiting for MPI messages. If a task finally reaches the external ready queue that means it can be executed immediately. All the variables it request are satisfied and memories to store the results are allocated in the DataWarehouse . As long as the external queue is not empty, the processor always has tasks to run. This can help to overlap the task wait time for communications.

5.2 Variable versioning

Because different tasks may request a same variable at different regions or at different time in an out-of-order way, we created multiple versions of variables under the same key. The DataWarehouse is improved to automatically select a proper version of variable according to the task's requests.

For example, in Figure 11, patch 0 is assigned to processor 0, patch 1,2,3 are assigned to processor 1. If three tasks on patch 1,2,3 all require 1 ghost cell of variable $varname$, three regions A,B,C of variable $varname$ on patch 0 need to be send to and stored at processor 1. Combine all the three regions and send through a single message will create new data dependencies. It removes the possibility that task on patch 1 may run when region A is received and region B and C are still in transmit. To solve this problem, DataWarehouse now use variable versions to store all the regions on the same patch.

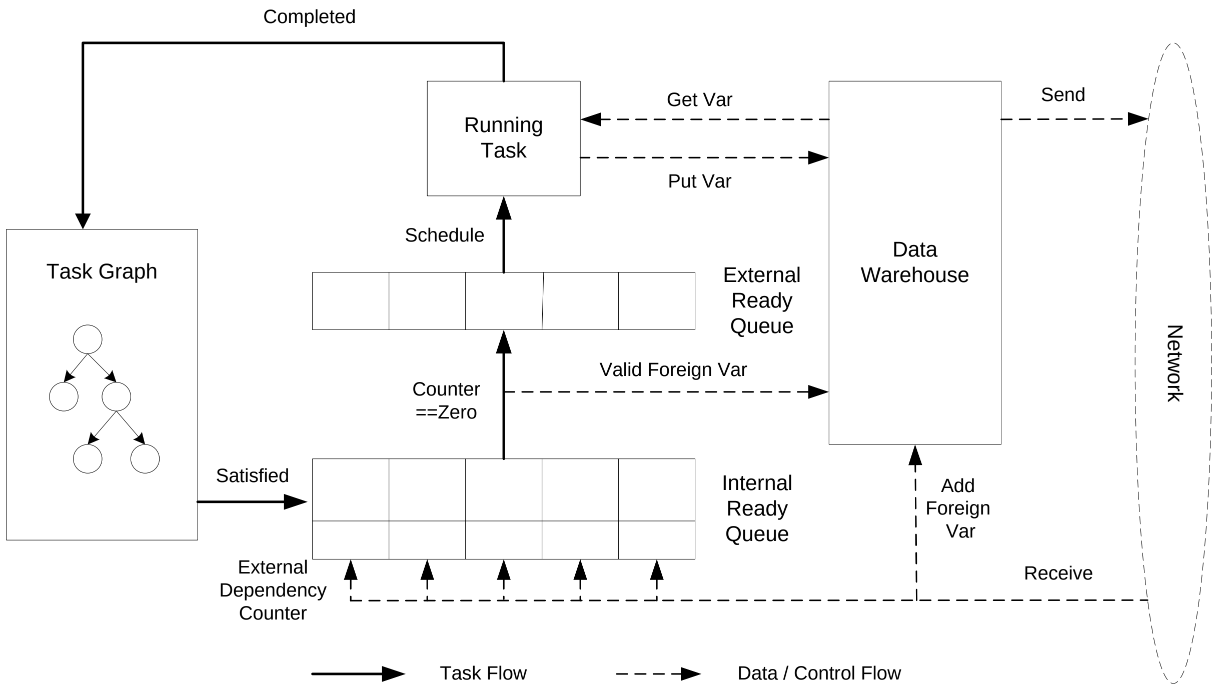


Figure 10: Architecture of Uintah dynamic task scheduling system

5.3 Synchronization Phases

Tasks that require a global communication require a specialized scheduling mechanism when task are running out of order. Those global tasks are created when: a) A task computes a global variable which need to be updated through the whole grid. i.e., computation of the total mass of the system. b) A task calls a third party library which need the MPI communicator as an argument. i.e., calling PETCs library. The global tasks will create one instance on each processor instead of one on each patch and need to be scheduled everywhere in the system at the same time. In a static scheduler, as all tasks are executed on a fixed order, the global tasks do not need specially treatment. But when task runs out order, two issues are noticed: deadlock and load imbalance.

Due to the limit of MPI library, there are no nonblocking reduction operations provided to us. If the global tasks run in an out of order way, a deadlock may occur. In Figure 12 up, R1 and R2 are two global tasks. If processor 0 runs R1 first then R2 and processor 1 runs in a reverse order, a deadlock occurs. Both processors will not make progress due to they are both blocked by the MPI library.

The load imbalance problem shows in Figure 12 down, task 1,2,3,4 are non-global tasks and R is a global task. No matter what execution order processor 0 and 1 choose, they have to sync when running task R. For example, if processor 0 runs tasks in the order: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow R$, processor 1 runs tasks in the order: $1 \rightarrow R \rightarrow 2 \rightarrow 3 \rightarrow 4$. As they are synchronized at task R, when processor 0 finished all 5 tasks, processor 1 still have to run task 2,3,4. Hence the load imbalance is observed.

To solve these two problems, the tasks are divided into different phases where each phase contains

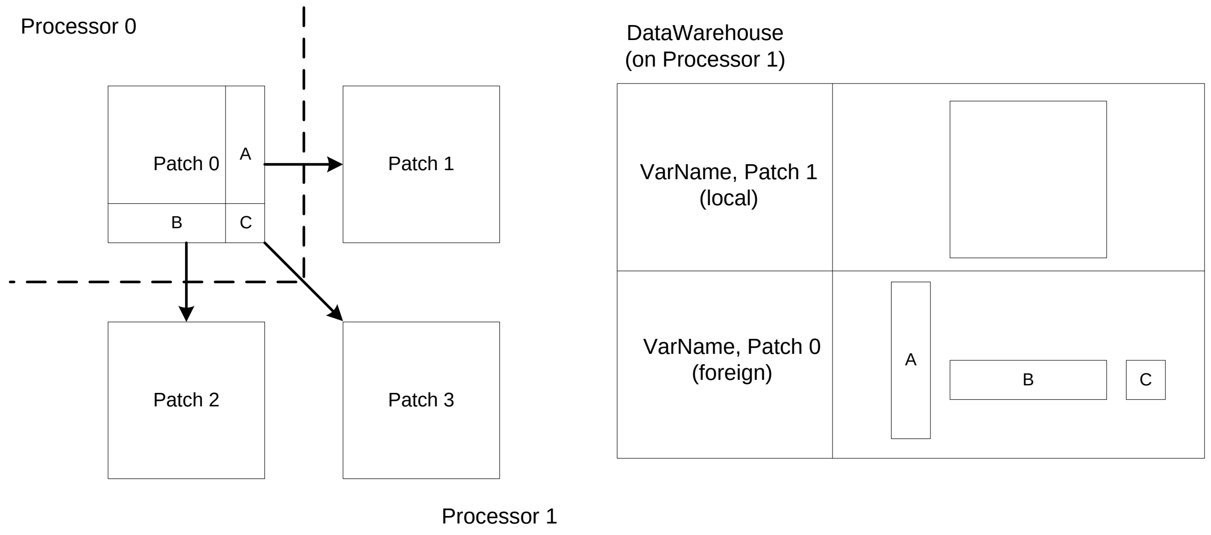


Figure 11: region versions of a same foreign variable

only one global task. The scheduler only executes the global task if all of other tasks in its phase have completed then move to the next phase. In this way, global task will be execute in a fixed order. In addition, the scheduler allows non-global tasks to be executed in an earlier phase but not a later phase.

6 Performance Evaluation

The dynamic scheduler produced a significant performance benefit in lowering both the MPI wait time and the overall runtime. In this section, we present performance results of dynamic scheduler with various benchmark to demonstrate and analyzes its advantage. These tests were preformed at the Ranger Cluster with 3,936 four AMD Barcelona quad-core processors nodes, full-CLOS InfiniBand interconnect at Texas Advanced Computing Center and the Kraken Cluster with 99,072 two AMD Istanbul six-core processors nodes, Cray SeaStar2+ interconnect at National Institute for Computational Sciences.

6.1 Dynamic Scheduling Speedup

The component timing result shows that our new dynamic scheduler significantly reduced the task communication wait time. Figure 13 shows the percent reduction of both wait time (which is as high as 90% in some cases) and total execution time on Ranger and Kraken. The example problem used is a two material compressible Navier Stokes type problem that models the movement of one material through another at high speed as in [8].

The result on Ranger (left) are produced on a fixed problem size (strong scaling) with 24578 patches of 16^3 cells. Task wait time from 512 to 4096 processors are reduced by about 65% to 90% . The overall execution time is reduced up to 50% on larger processors, as when we use more

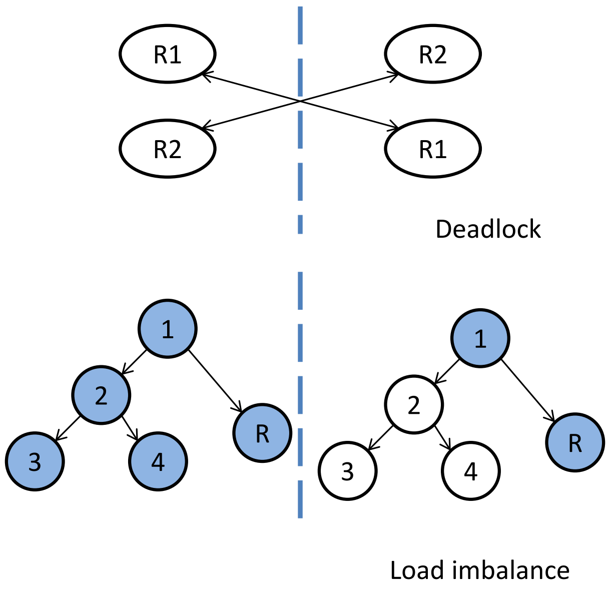


Figure 12: Out of order execution issues on global tasks

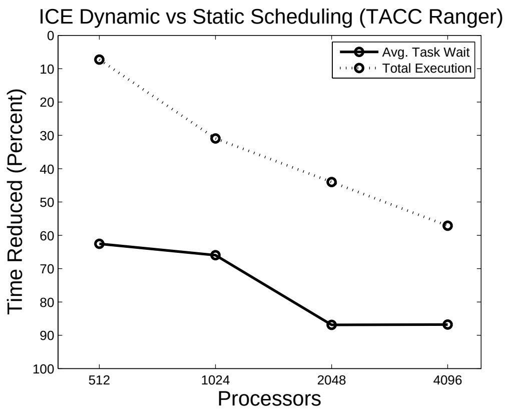
processors the part of MPI wait is increasing.

The result on Karken (right) are produced on a fixed problem size per processor (week scaling) with 8 patches of 16^3 cells on each processor. Task wait time from 192 to $48K$ processors are reduced constantly by about $50\ 60\%$. The MPI wait time is small part of total run time on Kraken, due to the benefit of a faster communication network. The overall execution time is reduced constantly to 10% on larger processors.

6.2 Task Priority

The performance of dynamic scheduling depends on how well the task executions overlapping the communications. The scheduler will have more opportunity to reduce wait times if the external ready queue is larger. One way to enlarge the external ready queue to give the priority to the task which can generate more ready tasks. Uintah calculates task priorities during the runtime. The priority algorithms we tested here are: i) Random, pick a random task from ready queue; ii) First Come First Serve(FCFS), pick the most early ready task; iii) Patch Order, pick the task which runs on the smallest patch ID (usually the most right patch) from the ready queue iv) MostMessages, pick the task which has can satisfy most external dependencies(a.k.a sending most MPI messages)

The ready queue length, wait time and overall runtime on an ICE problem with above task algorithms are in Table 1. Results show that dynamic scheduler needs a good priority algorithm to perform well. A better priority algorithm led to a larger queue length and lower wait time. We chose $MostMessages$ as our default priority algorithm as it favors the MPI sending tasks, which can reduce the MPI waiting time of neighbors nodes.



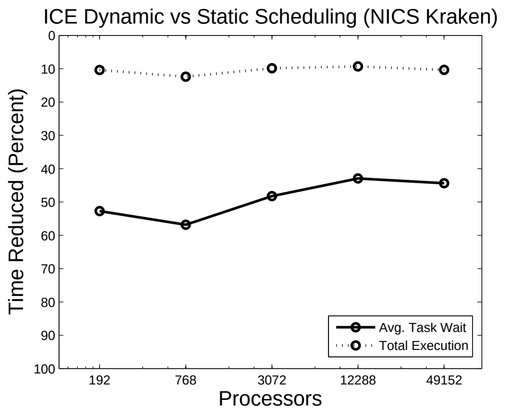


Figure 13: ICE simulation results, Dynamic vs Static scheduling

Priority Algorithm	Random	FCFS	PatchOrder	MostMessages
Avg. Queue Length	3.11	3.16	4.05	4.29
Avg. Wait Time	18.9	18.0	7.0	2.6
Total Run Time	315.35	308.73	187.19	139.39

Table 1: Same ICE problem running with different task priority algorithms

6.3 Granularity Effects

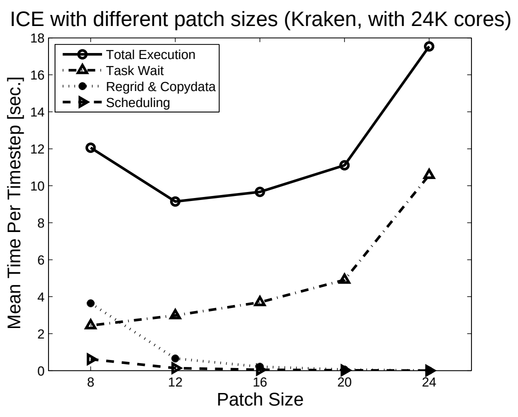
We can also increase the size of external ready queue by reducing the path size. Uintah's patch design that allow users to easily change the size and data layout which can affect performance. When the patch size is smaller, there are more patches per processor. Therefore, more tasks are created per processor and the size of external ready queue increases. Figure **??** are generated from a fixed ICE problem running with 24Kcores on Kraken. Right shows if patches are smaller, there are more patches per processor, the average length of task ready queue does increase and task wait time is lower.

Left shows that if we use smaller patches, the task wait time is small, but the overhead of regridding, data copying and scheduling is relatively large. As a result, the program's overall execution time will decrease first and then increase, depending on which part of the effects dominate.

7 Future work

The new dynamic task scheduler significantly reduce dthe communication wait time in Uintah simulations. This new scheduler's out-of-order execution strategy gives the framework's new ability adaptively overlapping communication and computation by determining the execution order of tasks according to both task graph and runtime information.

Based on the work we have done for out-of-order task execution, we can develop a multi-threaded scheduler to enable the advantage of multi-core architecture. Since all tasks in external ready queue



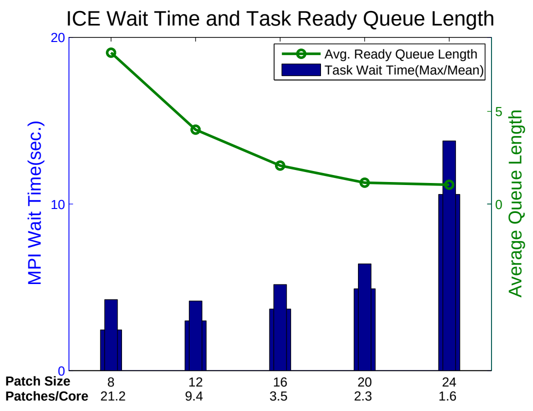


Figure 14: Granularity effects results: same problem running with different patch sizes.

can be executed at the same time, a multi-thread scheduler can easily dispatch these tasks to different threads. This will give us an additional dimension of parallelization and also reduce the overhead of load balancing and MPI communications.

8 Acknowledgement

This work was supported by the University of Utahs Center for the Simulation of Accidental Fires and Explosions (C-SAFE) and funded by both the Department of Energy, under subcontract No. B524196 and the National Science Foundation under subcontract No. OCI0721659

References

[1] M. Lewis B. Kashiwa and T. Wilson. Fluid-structure interaction modeling. Tech. Rep. LA-13111-PR, Los Alamos National Laboratory, Los Alamos, 1996.

[2] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *SFCS '94: Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Washington, DC, USA, 1994. IEEE Computer Society.

[3] Phillip Colella, John Bell, Noel Keen, Terry Ligocki, Michael Lijewski, and Brian van Straalen. Performance and scaling of locally-structured grid methods for partial differential equations. *Journal of Physics: Conference Series*, 78:012013 (13pp), 2007.

[4] J. Davison, St. Germain, John Mccorquodale, Steven G. Parker, and Christopher R. Johnson. Uintah: A massively parallel problem solving environment, 2000.

[5] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[6] B. Kashiwa. A multifield model and method for fluid-structure interaction dynamics. Tech. Rep. LA-UR-01-1136, Los Alamos National Laboratory, Los Alamos, 2001.

[7] B. Kashiwa and E. Gaffney. Design basis for cfdlib. Tech. Rep. LA-UR-03-1295, Los Alamos National Laboratory, Los Alamos, 2003.

[8] J. Luitjens and M. Berzins. Improving the performance of Uintah: A large-scale adaptive meshing computational framework. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS10)*, page (accepted), 2010.

[9] J. Luitjens, M. Berzins, and T. Henderson. Parallel space-filling curve generation through sorting: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(10):1387–1402, 2007.

[10] J. Luitjens, B. Worthen, M. Berzins, and T.C. Henderson. Scalable parallel amr for the uintah multiphysics code. In D. Bader, editor, *Petascale Computing Algorithms and Applications*. Chapman and Hall/CRC, 2007.

[11] Q. Meng, J. Luitjens, and M. Berzins. A comparison of load balancing algorithms for amr in uintah. SCI Technical Report UUSCI-2008-006, University of Utah, 2008.

[12] University of Tennessee. PLASMA. `http://icl.cs.utk.edu/plasma`, 2009.

[13] Steven G. Parker. A component-based architecture for parallel multi-physics pde simulation. *Future Gener. Comput. Syst.*, 22(1):204–216, 2006.

[14] Steven G. Parker, James Guilkey, and Todd Harman. A component-based parallel infrastructure for the simulation of fluid structure interaction. *Eng. with Comput.*, 22(3):277–292, 2006.

[15] Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[16] Deborah Sulsky, Shi-Jian Zhou, and Howard L. Schreyer. Application of a particle-in-cell method to solid mechanics. *Computer Physics Communications*, 87(1-2):236 – 252, 1995. Particle Simulation Methods.