CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2013)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3099

Scalable large-scale fluid–structure interaction solvers in the Uintah framework via hybrid task-based parallelism algorithms

Qingyu Meng*,† and Martin Berzins

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

SUMMARY

Uintah is a software framework that provides an environment for solving fluid–structure interaction problems on structured adaptive grids for large-scale science and engineering problems involving the solution of partial differential equations. Uintah uses a combination of fluid flow solvers and particle-based methods for solids, together with adaptive meshing and a novel asynchronous task-based approach with fully automated load balancing. When applying Uintah to fluid–structure interaction problems, the combination of adaptive meshing and the movement of structures through space present a formidable challenge in terms of achieving scalability on large-scale parallel computers. The Uintah approach to the growth of the number of core counts per socket together with the prospect of less memory per core is to adopt a model that uses MPI to communicate between nodes and a shared memory model on-node so as to achieve scalability on large-scale systems. For this approach to be successful, it is necessary to design data structures that large numbers of cores can simultaneously access without contention. This scalability challenge is addressed here for Uintah, by the development of new hybrid runtime and scheduling algorithms combined with novel lock-free data structures, making it possible for Uintah to achieve excellent scalability for a challenging fluid–structure problem with mesh refinement on as many as 260K cores. Copyright © 2013 John Wiley & Sons, Ltd.

Received 31 July 2012; Revised 20 June 2013; Accepted 26 June 2013

KEY WORDS: MPI; threads; Uintah; many core; lock free; fluid–structure interaction

1. INTRODUCTION

A significant part of the challenge in moving from petascale to exascale problems is to ensure that the multiphysics multiscale codes used to solve real application problems can be run in a scalable way on parallel computers with large core counts. The multiscale challenge involved arises from the need to use approaches such as adaptive mesh refinement while the multiphysics challenge is typified by different physics being used in different parts of the domain with different computational loads in these different spatial domains. In addition, it is necessary to couple these domains with their different physics. This coupling is an interesting challenge in itself as the coupling algorithm may be more complex than the algorithms used away from the fluid–structure interface. These challenges are further compounded by the anticipated relative memory per core potentially shrinking [1] and nodal architectures becoming more complex with ever-increasing core counts and with the addition of accelerators on machines such as Titan‡ and Stampede.§

*Correspondence to: Qingyu Meng, Scientific Computing and Imaging Institute, University of Utah, Salt lake City, Utah 84112, USA.

†E-mail: qymeng@cs.utah.edu

‡Titan is a parallel computer under construction at Oak Ridge National Laboratory with about 299K CPU cores now with a large number of attached GPUs to be added in 2012.

§Stampede is a parallel computer under construction at Texas Advanced Computing Center with two petaflops of CPU performance and eight petaflops of accelerator performance.

Copyright © 2013 John Wiley & Sons, Ltd.

This work will start to address some of these challenges by developing scheduling algorithms and associated software that will be used to tackle fluid–structure interaction algorithms in which mesh refinement is used to resolve the structure. The vehicle for this work is the Uintah Computational Framework [2–4]. The broad range and challenging nature of Uintah applications and the software itself are described in [5]. The Uintah code was designed to solve fluid–structure interaction problems arising from a number of physical scenarios. Uintah uses a combination of computational fluid dynamics (CFD) algorithms in its implicit continuous-fluid Eulerian (ICE) solver [6, 7] and couples ICE to a particle-based solver for solids known as the material point method (MPM) [8, 9]. The adaptive mesh refinement (AMR) approach used in Uintah is that of multiple levels of regularly refined mesh patches. This mesh is also used as a scratch pad for the MPM calculation of the movement and deformation of the solid [10, 11]. Uintah has been shown to scale successfully with many fluid and solid problems with adaptive meshes [12, 13]; however, the scalability of fluid–structure interaction problems has proven to be somewhat more challenging. This is at least partly because the particles that represent the solid in Uintah can freely move across mesh patch boundaries in a way that is not known beforehand and partly because not all the domain is composed of only the solid or the fluid. There are two key components in the approach that will be described here to improve the scalability of fluid–structure interaction. The first component concerns access to data stored at the level of a node in Uintah. In Uintah, a multicore node is treated as a miniature shared memory system, and only one copy of Uintah's global data needs to be stored per multicore node in the data warehouse that Uintah uses, thus saving a considerable amount of memory overall [12]. This approach also makes it possible to migrate particles across the cores in a node without using MPI. However, for this approach to be successful, it is necessary to design a single nodal data structure, a data warehouse, that large numbers of cores can simultaneously access without contention.

The second component concerns how the cores on a node themselves request work. In the model proposed by the authors [12], a single centralized controller allocates work to the cores. This approach has worked well on the Kraken[¶] and Jaguar XT5[‖] architectures. This approach breaks down when there are more cores per node, and when communications are faster, as on the Jaguar XK6 machine that consisted of the CPU part of the Titan machine. The solution, as will be shown, is to move to a distributed approach in which the cores themselves are able to request work.

Both approaches will be shown to make a substantial improvement to the scalability of fluid–structure interaction problems. This improvement in scalability will be demonstrated by using a challenging problem that consists of the motion of a solid through a liquid that is modeled by the compressible Navier–Stokes equations. The solid is modeled by particles on the finest part of an adaptive mesh.

The main challenges addressed by this paper (and indeed its novelty) are as follows:

1. The incorporation of a decentralized multithreaded scheduler model to the Uintah framework;
2. Improving the performance of applications by using lock-free data structures in the hybrid multithreaded scheduling approach; and
3. Using a real-world fluid–structure application on a machine with a large core count to demonstrate these improvements in scalability.

Although the challenges addressed here is similar to that addressed in the PRONTO work [14, 15] for the solution of contact problems in which the contact algorithm requires more work than takes place in the rest of the domain, the solution adopted here is very different. There are many examples of other parallel fluid–structure interaction work [16–19], but the approach adopted here is somewhat different to many as it relies heavily upon the asynchronous nature of Uintah.

There are also many other codes that are similar to some parts of Uintah and have been run on large parallel architectures. In the case of adaptive mesh codes, there are many such solvers. Examples of codes that run on large parallel architectures are those of Steensland, Wissink, and

[¶]Kraken is a National Science Foundation (NSF) Cray XT5 computer located at NICS, Oak Ridge, Tennessee with 112K compute cores and a peak performance of 1.17 PF

[‖]Jaguar was until 2012 the Cray XT5 system at Oak Ridge National Lab. with 224K cores and a peak performance of 2.33 petaflops

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Parashar, [20, 21], the Flash code [22–24] based on adaptive octtree meshes and the physics AMR code Enzo, [25, 26]. The Cactus framework [27–29] provides solutions for a broad class of computational physics applications. The highly scalable codes of Ghattas *et al.* [30] also use an octree-based approach for very different problems. There are many codes that use high-speed flow algorithms. On example typical of many recent codes is the Sandia CTH code, [31], which also now uses MPM. Most of these codes do not target the problems that Uintah has been designed for, with large deformations, complex geometries, and massive parallelism through an asynchronous task-based approach.

In the remainder of this paper, the aforementioned challenges are addressed as follows: Section 2 describes the Uintah software and the adaptive fluid–structure methodology used. In Section 3, the details of the fluid–structure algorithm are described. Section 4 describes the existing scalability of Uintah on fixed and adaptive meshes. The challenge of scaling fluid–structure algorithms is explained in Section 5 in the context of a challenging model problem. Section 6 describes the different approaches that may be used in scheduling tasks, whereas Section 7 shows how to redesign the data warehouse to avoid contention when access is attempted by multiple cores. Section 8 shows that these approaches very much improved the scalability of the model problem on the Jaguar XK6 architecture up to 260K cores. Section 9 provides some conclusions and areas for future work. Overall, Sections 2–4 help the reader with Uintah, whereas Sections 5–9 contain new material.

2. UINTAH'S FLUID–STRUCTURE INTERACTION METHODOLOGY

The Uintah computational framework was intended to make it possible to solve complex fluid–structure interaction problems on parallel computers. In particular, Uintah is designed for *full physics* simulations of fluid–structure interactions involving large deformations and phase change. The term *full physics* refers to problems involving strong coupling between the fluid and solid phases with a full Navier–Stokes representation of fluid phase materials and the transient, nonlinear response of solid phase materials that may include chemical or phase transformation between the solid and fluid phases.

Uintah uses a full *multimaterial* approach in which each material is given a continuum description and is defined over the complete computational domain. Although at any point in space the material composition is uniquely defined, the multimaterial approach adopts a statistical viewpoint whereby the material (either fluid or solid) resides with some finite probability. In order to determine the probability of finding a particular material at a specified point in space, together with its current state (i.e., mass, momentum, and energy), multimaterial equations are used. The algorithm that uses a common framework to treat the coupled response of a collection of arbitrary materials is described in the succeeding text. This methodology follows the ideas of Kashiwa *et al.* [32, 33]. Individual equations of state are needed for each material to determine relationships between pressure, density, temperature, and internal energy. Constitutive models are also required to describe the stress for each material based on appropriate input parameters (deformation, strain rate, history variables, etc.). In addition to those parameters, the multimaterial nature of the equations also requires closure for the volume fraction of each material. For a precise description of the algorithm, the reader should refer to [6, 7, 34].

2.1. The ICE multimaterial CFD approach

In order to represent fluids in its multimaterial CFD formulation, Uintah uses the ICE method [35], further developed by Kashiwa and others at Los Alamos National Laboratory [36]. The use of a cell-centered, finite volume approach is convenient for multimaterial simulations in that a single control volume is used for all materials. This is particularly important in regions where a material volume goes to zero, as by using the same control volume for mass and momentum, if the material volume tends to zero, then the associated mass and momentum also similarly tend to zero. The approach allows the solutions to a broad class of fluid flow problems to be computed.

The Uintah implementation of the ICE technique uses operator splitting in which the solution consists of a separate Lagrangian phase where the physics of the conservation laws are computed

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

and a Eulerian fluid phase in which the material state is transported via advection to the surrounding cells. The general solution approach is well developed and described in [6, 7, 34, 37].

2.2. *The material point method*

Solids in Uintah are represented by the particle method known as the MPM [8, 9]. MPM is a powerful technique for computational solid mechanics and has found favor in applications such as those involving complex geometries, large deformations, and fractures; see [5] for these and many other examples. MPM is an extension to solid mechanics of FLIP [38], which is a particle-in-cell method for fluid flow simulation [39]. Uintah also uses an implicit formulation of MPM [40]. In MPM, Lagrangian particles or material points are used to discretize the volume of a solid material. Each particle carries state information (e.g., mass, volume, velocity, and stress) about the portion of the volume that it represents. The MPM method typically uses a Cartesian grid as a computational scratchpad for computing spatial gradients. This grid may be arbitrary, and in Uintah, it is the same grid used by the accompanying multimaterial ICE CFD component. Particles are usually created on the finest level of the mesh and are always mapped back to the background grid according to their coordinates. The initial physical state of the solid is projected from the computational nodes to the cell centers collocating the solid material state with that of the fluid. This common reference frame is used for all physics that involve mass, momentum, or energy exchange among the materials. This results in a tight coupling between the fluid and solid phases. This coupling occurs through terms in the conservation equations, rather than explicitly through specified boundary conditions at interfaces between materials. Because a common multifield reference frame is used for interactions among materials, typical problems with convergence and stability of solutions for separate domains communicating only through boundary conditions are alleviated. Considerable improvements in MPM and its analysis [41–44] have resulted from work connected to the Uintah code.

3. UINTAH FLUID–STRUCTURE ALGORITHM

The combination of MPM and ICE, MPMICE, is Uintah's fluid–structure interaction component. One of the challenges in multiphysics simulations is that the coupling algorithms involve calculations with each of the methods used in the domains that are coupled. Such calculations impose extra work in the coupling domain and also involve calls to the functions that implement the individual methods being coupled. In the Uintah coupling algorithm, there are 12 steps. Some of these steps apply only to the fluid (as labeled by ICE), others apply only to the solid (as labeled by MPM), whereas other steps apply to both the fluid and the solid (as labeled by MPMICE). If each phase of this algorithm is used as a synchronization point, then the parts of the domain not containing a multiphysics interface will be locked out while the interface calculation proceeds. The 12 steps used in the Uintah coupling algorithm are the following [7]:

1. Interpolate particle state to grid, MPM.
2. Compute the equilibrium pressure, ICE.
3. Compute face centered velocities for the Eulerian advection, ICE.
4. Compute sources of mass, momentum, and energy as a result of phase-changing chemical reactions, MPMICE.
5. Compute an estimate of the time-advanced pressure, ICE.
6. Calculation of face-centered pressure using a density-weighted approach, ICE.
7. Material stresses calculation, MPM.
8. Compute Lagrangian phase quantities at cell centers, ICE.
9. Calculate momentum and heat exchange between materials, MPMICE.
10. Compute the evolution in specific volume due to the changes in temperature and pressure during the foregoing Lagrangian phase of the calculation, ICE.
11. Advect fluids for the fluid phase, ICE.
12. Advect solids for the solid phase, interpolate the time-advanced grid velocity and the corresponding velocity increment back to the particles, and use these to advance the particle's position and velocity, respectively, MPM.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

The difficulties associated with this algorithm from a parallel scalability point of view are twofold. The first difficulty is that the MPM work per patch depends on the number of particles per patch. This value constantly changes as particles enter or leave patches. The second difficulty is that as particles are not distributed throughout the domain, the work associated with MPM only takes place in an irregular and transient manner throughout the patch set.

4. EXISTING UINTAH SCALABILITY

In order to put the work in this paper in context, it is helpful to provide a summary of the evolution of the Uintah code. There have been three main phases of development.

Phase 1: 1998–2005 In this phase, the task graph based framework was written and most of the key applications code that runs on static meshes developed [3, 6, 7]. The task graph was executed in a fixed order and the different components of Uintah scaled to about 2000 cores.

Phase 2: 2006–2010 In the second phase of development, the emphasis was on adaptive mesh refinement and on revising many of the data structures and algorithms while still using an MPI-only approach. These developments resulted in scalability of mesh refinement applications to 98K cores [10, 45].

Phase 3: 2011- The third phase of Uintah development has focussed on a rutime system that scales well on large core counts and extends to computer architectures with accelerators. The start of this work was out-of-order task execution [46] and the move to one MPI process per multicore node [12]. This work provides an extension of these ideas to more complex problems on recent architectures. At the same time, the runtime system of Uintah has recently started to be extended to accelerators [47]. In describing the existing scalability of Uintah, there three topics that need to be covered.

4.1. Adaptive mesh refinement algorithms

Before addressing fluid–structure interaction with AMR, it is important to be sure that the adaptive meshing part of the algorithm scales independently. The importance of AMR for allowing refined and coarsened meshes is now well understood in many areas of computational science. In the case of codes such as Uintah, which solves large systems of partial differential equations on a mesh, refining the mesh increases the accuracy of the simulation. Although Uintah's task graph structure of the computation makes it possible to improve scalability through adaptive self-tuning, the changing nature of the task graph from adaptive mesh refinement poses extra challenges for scalability [45]. In the last 4 years, improvements within Uintah to the regridding and load-balancing algorithms have led to a 40× increase in the scalability of AMR [10, 11]. Previously, Uintah used the well known Berger–Rigoutsos algorithm [48] to perform regridding. However, at large core counts, this algorithm performed poorly. The new regridder [11] defines a set of fixed-sized tiles throughout the domain. A search of each tile is then made for refinement flags without the need for communication, and all tiles that contain refinement flags become patches. This regridder scales well at large core counts because cores only communicate once at the end of regridding when the patches are combined.

4.2. Load-balancing and task execution

When the simulation grid changes as a result of physics or load imbalance, a new grid and mesh patches are created and the Uintah framework automatically migrates the mesh patches to their new homes on the cores. This automation is possible as the framework manages all the simulation variable allocations. The load-balancing algorithm used accurately predicts the amount of work a patch requires by using data assimilation and measurement techniques with feedback as this approach outperforms traditional cost models [10]. In this way, Uintah's load balancer determines a reasonable allocation of patches to nodes using measurement and geometric information. The load balancer attempts to guarantee that an equal amount of work is distributed to each core so as to allow for an

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

optimal scaling of the simulation on multiple cores. Once the AMR regridder generates the hexahedral mesh patches of regular mesh cells, a bounding volume hierarchy (BVH) [49] tree is built or updated to hold all the patches for fast lookup on each independent mesh level. The patches are uniquely assigned to cores by the load balancer. Once a local set of patches is known, each core can run computational tasks on its own patches and communicate with other cores through MPI messages automatically generated by Uintah. A significant amount of development has also been carried out on out-of-order execution of tasks, which has produced a significant performance benefit in lowering both the MPI wait time and the overall runtime [13, 46] on 98K cores and beyond.

4.3. Data warehouse

Uintah's data warehouse is a hashed-map-based dictionary that maps variable names and mesh patch or level id to the memory address of a variable. Each task can obtain its read and write variables by querying the data warehouse with a variable name and a patch id. After each timestep, the old data warehouse is frozen and is set to be read-only. A new data warehouse to store newly computed variables in this timestep is then initialized. Variables can also be carried over from the old to the new data warehouse so as to avoid memory reallocation. Besides acting as a central dictionary for all variables, the data warehouse also manages MPI communication buffers. When an MPI message needs to be received (say for a variable coming from another core), a foreign variable will be allocated by the data warehouse and marked as invalid. An associated MPI buffer for this foreign variable will be provided to hold the incoming message. When the variable is received, the foreign variable will be marked as valid and ready for use by computational tasks. As all Uintah variables are allocated and accessed through the data warehouse, this warehouse must also keep track of the life of any variable. All the intermediate variables are deallocated through a scrubbing process if no longer needed.

The message passing paradigm that Uintah initially operated under was that any data that needed to be communicated to a neighboring core was passed via MPI. For multicore architectures, the process of passing data that is local to a node by using MPI is both time consuming and duplicates identical data that can be shared between cores.

Uintah now stores only one copy of global data per multicore node in its data warehouse. The task scheduler now spins off tasks to be executed on, say, nc cores using a threaded model, [12] which results in the global memory used in a single shared data warehouse being a fraction of nc^{-1} of what is required for multiple MPI tasks, one per each of the nc cores. This memory saving expands the scope and range of problems beyond those that it has been possible to explore until now [12, 50]. This approach is also being extended to spin off tasks to be executed on other types of processors, such as GPUs. A working prototype that uses as many as 1000 GPUs has recently been tested on the TitanDev project at Oak Ridge as well as full capability jobs being run on the Keeneland GPU system at NICS, which has multiple GPUs per node [51].

Through this research, Uintah has been able to show strong and weak scalability up to 196K cores on Department of Energy's Jaguar for ICE with AMR as seen in [10–13]. Finally, the use of Uintah with scalable linear solvers such as hypre has led to good weak scalability up to about 200K cores [52].

4.4. Uintah software engineering and sustainability

The Uintah framework has over 700K lines of code, and it is natural to ask what testing procedures are in place to ensure that the code executes correctly. Code development inside Uintah follows a formal structure. Each Uintah component is a C++ class that must implement several virtual methods: **problemSetup()**, **scheduleInitialize()**, **scheduleComputeStableTimestep()**, and **scheduletimeAdvance()**. Each scheduling task, that is, **scheduleTimeAdvance**, contains a callback pointer to a function implementing the actual work of the function. Extensive Uintah documentation describes each of these methods in detail (http://www.uintah.utah.edu/trac/wiki/Documentation). For example, the purpose of **scheduleInitialize** is to initialize the grid data with values read in

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

from the **problemSetup** and to define what variables are actually computed in the time advancement stage of the simulation. Similarly, the purpose of **scheduleTimeAdvance** is to schedule the actual algorithmic implementation. In general, the best way to schedule the algorithm is to break it down into individual tasks. Each task will have its own data dependencies and function pointers that reference individual computational methods. In order to ensure the quality of new codes and the underlying software base, Uintah has had a fifteen-year history of running daily build and test scripts. The Uintah project uses a continuous integration testing buildbot (http://buildbot.net/trac) to constantly build and regression test every source code modification checked into the subversion repository. The Buildbot system provides an automated compile and test cycle that is triggered each time the Uintah subversion repository is updated. After the compile checks, a suite of 100 regression tests are performed. Any failures are reported, and any potential problems are quickly identified and reported to the individual developer via email and to the internal developer mailing list. In addition, a publicly available web server runs and records the status of the build and tests for the individual check-in and the more comprehensive nightly tests. Finally, each night, the buildbot does a comprehensive build and test for both debug and optimized builds. This process provides a level of confidence to developers that the Uintah code base will always be in a compilable and tested state at any given point in time. The latest Uintah release is always available for download at www.uintah.utah.edu, and the Uintah User Guide and application guides [53, 54] are also available on our web site, together with an installation guide that covers installation of Uintah and all supporting libraries including PETSc, hypre, MPI, and VisIt.

Although this process is similar to the working code provided and used by other frameworks for the last 15 years, it does not ensure that the code, although demonstrably executing correctly in many cases, is necessarily formally correct. This may be potentially problematic when running at the very largest core counts. For this reason, we have started to look at more formal approach for ensuring correctness and have outlined our approach in [55]. We have not yet found additional bugs in the runtime system with this approach.

5. A MOTIVATING PROBLEM AND THE CHALLENGES OF SCALING ON FLUID–STRUCTURE PROBLEMS

The motivating fluid–structure interaction problem used here arises from the simulation of explosion of a small steel container filled with a solid explosive (PBX-9501). The explosive ignites and begins to burn, converting the solid into a high temperature gas which in turn causes the container to pressurize. As the pressure increases, the container expands and eventually ruptures violently. The benchmark problem used for this scalability study is the transport of a small cube of steel container piece inside of the PBX product gas at an initial velocity of Mach two. The simulation used an explicit formulation with the lock-step time-stepping algorithm that advances all levels simultaneously. This problem exercises all of the main features of ICE, MPM, and AMR and amounts to solving eight partial differential equations, along with two pointwise solves and one iterative solve as described in [10, 13]. The ICE method [32, 33, 41] is used to model the gas, and the MPM [8] method is used to model the solid. The interactions between the gas and the solid that cause the block of material to move are modeled using the algorithm described in Section 3 [6, 7, 34]. This benchmark also included a model for the deflagration of the explosive and the material damage model ViscoSCRAM [56] in order to be representative of the type of target calculations that are aiming at [13]. For the purposes of weak scaling, each successively larger version of this problem is about four times as large as the previous one. The simulation utilized three mesh refinement levels with each level being a factor of four more refined than the previous level. The starting solution with fluid velocity arrows and the refined spatial mesh around the moving block of material to this problem is shown in Figure 1 as is the solution at the end of the run again with fluid velocity arrows and the refined spatial mesh around the moving object. The refinement algorithm used tracked the interface between the solid and the fluid causing the simulation to regrid often while maintaining a fairly constant-sized grid, which allows the scalability to be more accurately measured. This criteria led to each problem being about four times as large as the previous one.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

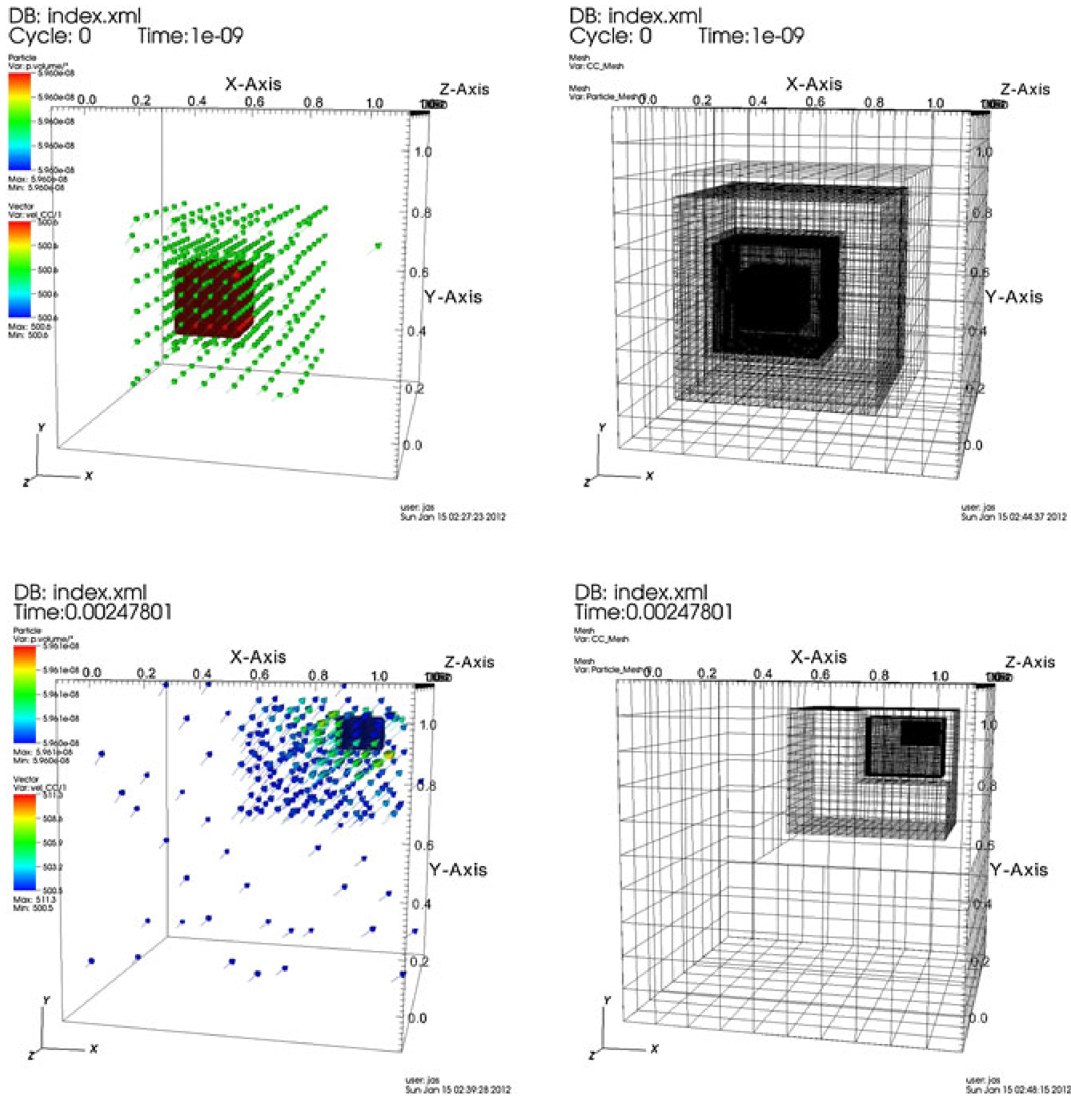

Figure 1. initial solution with velocity arrows (top left) and mesh (top right) and final solution with velocity arrows (bottom left) and mesh (bottom right) of fluid–structure interaction problem.

5.1. Initial computational results on Kraken using MPI only

This test problem was originally run on NSF's Kraken system [57] in 2011 with the scaling results shown in Figure 2. In this figure, the average time per timestep is compared with the number of cores used. The benchmark involved four strong scaling runs of varying size. Each of these runs uses an initial coarse mesh with doubled resolution with respect to the previous run. The refinement criteria used in MPMICE refines the mesh anywhere that particles exist. This refinement criteria led to each problem being about eight times as large as the previous one. In the four cases shown, the number of mesh cells were 619K, 3.8M, 21.3M, and 152M, respectively, whereas the number of MPM points used to represent the solid were 2.1M, 16.8M, 134M, and 1.1B in each of the strong scaling cases associated with the four solid lines. In the case of the run with the largest number of points, the rightmost solid line, the strong scaling clearly breaks down as the line turns up. In this case, the code has only 16% relative efficiency [57] at the final point with respect to the run in the top leftmost corner. The dashed lines show weak scaling information in which the lines should be horizontal if weak scaling occurs when moving to a larger core count with the same amount of work per core. In the case of this problem at larger core counts, both weak and strong scaling

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

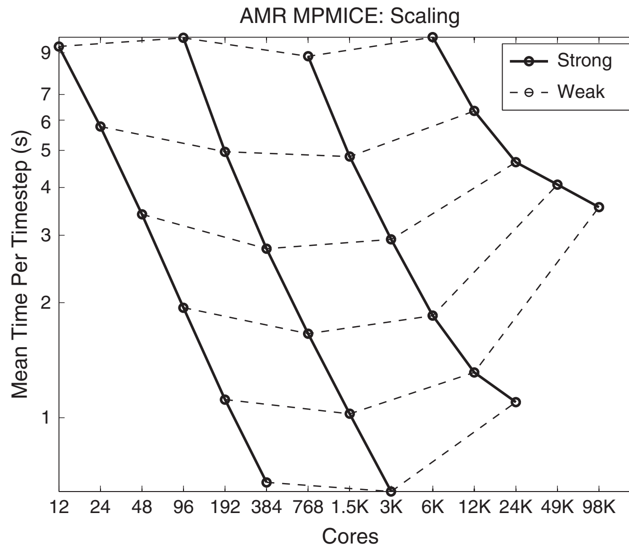

Figure 2. Initial poor scaling of Uintah on the fluid–structure interaction problem. The four solid lines show the average time per timestep for the strong scaling runs with each run having a problem size that is eight times larger than the solid line to its left. The dashed lines show weak scaling information for problem sizes that increase in proportion to the cores used.

break down. In this example, as the solid moves through the domain, the number of particles in a certain area changes significantly. Figure 3 shows how the number of particles and execution time changes during the simulation at two different locations. Location A is at the front of the object, whereas location B is at one of the edges of the object. Figure 3 shows that the computing cost of a patch with particles is about six times as great as the cost of a patch without particles. This causes a serious load imbalance issue that leads to poor scaling results as shown in Figure 2. Even with the measurement-based load balancer, the computational work in a region with particles is hard to precisely predict. When running with pure MPI scheduler, if one core has finished its assigned work faster than the other cores, it has to stay idle and wait for them to finish even if they are in the same node. With the same code, adaptive mesh refinement scaled well to 98K cores [12]. The challenges that arose in the scaling of this fluid–structure interaction problem directly motivated and influenced the work presented here.

6. UINTAH TASK GRAPH AND SCHEDULERS

In order to describe how Uintah schedules task in executing the task graph, an explanation is needed on how the task graph is generated as well as a brief description of the approaches used to execute it. Uintah uses both data parallelism and task parallelism to achieve high scalability when running with the hybrid multithreaded/MPI scheduler in [12]. After patches are assigned to nodes and detailed tasks are created on local and neighboring mesh patches, Uintah computes dependencies between tasks by comparing each task's computed (output) and required (input) variables. If a task's computed and required variables are read from and sent to tasks on a local patch, then an internal dependency is detected and marked as such on the task graph. If a task requires variables from (or computes variables for) a task on a patch held by another core or node, then an external dependency is detected in the task graph compilation process. In this case, a unique MPI message tag is assigned on both the node that computes the variable and the node that requires them.

Once all dependencies are detected, a directed acyclic graph that governs task execution is compiled. Uintah's directed acyclic graph based approach in which the dependencies of all tasks are analyzed before task execution and if any memory conflicts exist in two tasks, this analysis prevents those tasks from running at the same time. The Uintah scheduler uses the task graph to determine the order of execution, assigns tasks to local computing resources (CPU/GPU), and ensures that the correct interprocess communication is performed.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

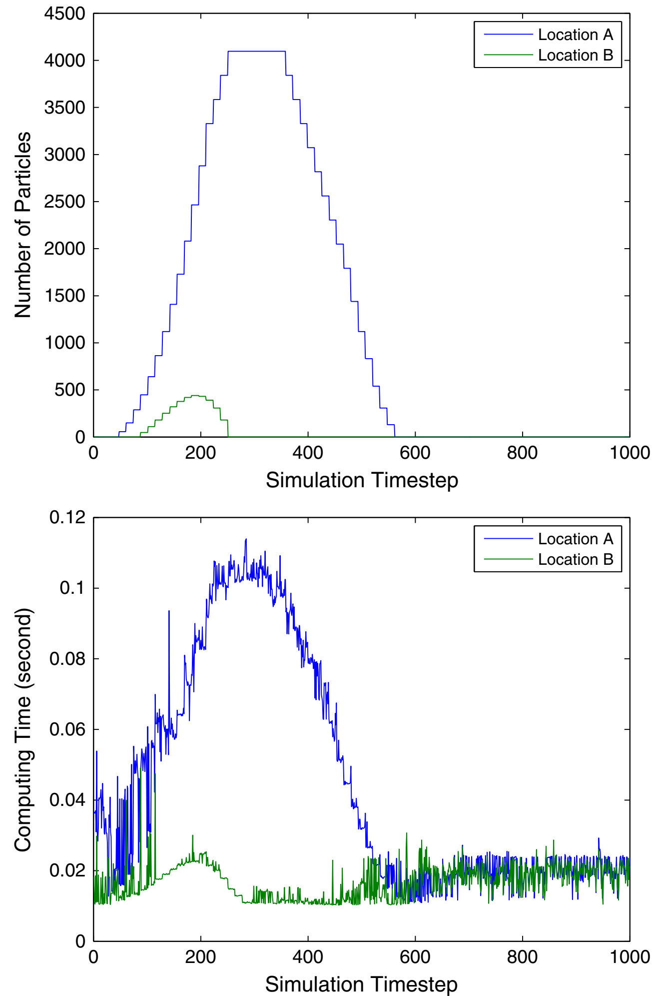

Figure 3. Number of particles and task execution time at two different locations A and B in the spatial domain.

6.1. Dynamic scheduler

Originally, Uintah used a static scheduler in which tasks were executed in a predetermined order. A limitation of this approach is that a single task waiting for messages causes the calculation on a core to sit idle. In order to address this issue, a new Uintah dynamic scheduler was developed so as to better overlap communication and computation by using out-of-order task execution [46]. Reference [46] also shows how Task queues were added to allow the scheduler to keep track of the different states of all tasks. An internal ready queue stores tasks whose local dependencies have been satisfied and are waiting for external MPI messages to arrive. An external ready queue stores tasks whose internal and external dependencies have all been satisfied and are now ready for execution. As long as the external ready queue is not empty, a core can always be given tasks to execute. Once a task is completed, the scheduler will use the task graph to find the subsequent internally satisfied tasks and add them to internal ready queue. In order to support dynamic scheduling of tasks, the Uintah data warehouse was changed from a hashed map to a hashed multimap to allow the saving of multiple variables under the same variable name and a patch id key. This change was necessary as when a task is running out-of-order, multiple versions of a variable may exist at the same time. This scheduler produced a significant performance benefit in lowering both the MPI wait time and the overall runtime [46].

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

6.2. Multithread scheduler: master–slave model

Although the out-of-order execution model of [46] worked well for many cases, one limitation of its pure MPI scheduling is that tasks which are created and executed on different cores on the same node cannot share data. A new multithreaded MPI scheduler was used [12] to solve this problem by dynamically assigning tasks to worker threads during execution. This multithreaded MPI scheduler used a master–slave model that had one control thread and several worker threads per MPI process. The control thread processed MPI receives, managed tasks queues, and assigned ready tasks to worker threads. The worker threads simply executed their task and then asked the control thread to assign the next task. The control thread and worker threads communicated through pthread conditional variables. Uintah variables can be accessed freely by any thread as the task graph guarantees that there are no memory conflicts on any two ready tasks. However, many shared data structures, such as data warehouse and task queues, need to be thread safe and consistent across threads. As the Uintah framework manages tasks input and output, most tasks are already thread safe and can be supported by the multithreaded scheduler without rewriting any task code. Experimental results on AMR ICE simulations showed 50% to 90% savings on memory usage by using the multithreaded scheduler. This approach still did not result in the scalable execution of the fluid–structure problem described in Section 5, however.

6.3. Multithread scheduler: decentralized model

A potential bottleneck in the master–slave model is that a worker thread may become idle if the control thread cannot respond to its next ready task request quickly enough. In order to guarantee a short response time, the control thread was assigned to a dedicated core. This approach led to this core being underutilized when running with small number of cores, as there was not enough work for the control thread. The solution to this was to design a new decentralized multithreaded scheduler to allow all threads to process MPI sends and MPI receives or to execute tasks concurrently without a control thread. Instead of asking the control thread for a ready task, the threads in the decentralized multithreaded scheduler directly pull tasks from the two ready queues. When a thread pulls a task from the internal ready queue, then MPI nonblocking receives are posted. Also, when a thread pulls a task from the external ready queue, then a task's callback function is executed and MPI sends are posted after task execution. Furthermore, each thread must keep checking the task queues and MPI receives when it is idle as there is no longer a central controller. As this could lead to busy-locking on those shared resources when multiple threads become idle and keep acquiring read locks, a two-stage execution approach is used.

The first stage is to check if any work is available, either to process MPI receives or to execute a task for the current thread. When there is a ready task or a pending MPI receive, the scheduler will switch to the second stage to execute the task or to process MPI receives concurrently. If no work is available, a mutex needs to be acquired before checking all the task queues and MPI receives again. The scheduler will not release this mutex and so will prevent other idle threads from checking task queues and MPI receives until a new ready task is available or a new MPI receive is posted. In this way, when multiple threads become idle, checking for shared resources will be slowed down by this mutex and priority is given to threads that are able to update the task queue or post MPI receives.

Table I shows a performance comparison between the decentralized and master–slave models in a 32-core single node. In this case, the decentralized model outperforms the master–slave model on all runs up to 32 cores per node. By monitoring the CPU utilization of each core, it was confirmed that the decentralized model solved the issue of underutilization of the core that runs the control

Table I. Execution time: master–slave versus decentralized.

Number of cores	2	4	8	16	32
Master–slave	57.28	20.72	9.4	4.81	2.95
Decentralized	29.8	15.84	8.2	4.59	2.78

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

thread. Furthermore, when one master control thread with 1, 3, 7, 15, and 31 worker threads are used, the CPU loads on the control thread increases linearly and are about 0.3%, 0.7% 1.7% 3.0%, and 6.9%, respectively. There is an increase in master control thread CPU usage with increasing numbers of worker threads. This makes it difficult to balance the control thread workload with the worker threads, although the master–slave model will likely hit a control thread bottle neck when number of cores per node increases. In contrast, the decentralized scheduler is able to fully utilize all available cores on-node, regardless of the number of cores.

7. UINTAH HYBRID PARALLELISM IMPROVEMENT

This section will discuss how the hybrid multithreaded/MPI approach was used to overcome the challenges with regard to scaling fluid–structure problems. Our previous published work used the multithreaded MPI approach for fluid problems without particles by using the ICE algorithm [12]. Unlike all the tasks in ICE components that were thread safe, several race conditions were found to exist in the MPM simulation components. Most of these race conditions were due to the use of global temporary data structures instead of local ones and were easily fixed. However, much of the existing framework related to particles had to be rewritten to guarantee thread safety.

7.1. Reducing particle relocation costs

As noted earlier, in Uintah, solid objects are represented by MPM particle variables. During a timestep, each particle's new location coordinates and other physical attributes will be computed by MPM tasks and saved in the data warehouse. If a particle's position moves from one patch to another, it is the infrastructure's responsibility to map those particle variables back into the background grid according to their new locations. This is carried out by inserting a relocation task into the task graph. During this process, each particle's new 'owner' patch will be located in the patch BVH tree according to its new coordinates. If the new 'owner' patch is located on the same node, only a simple reindexing of the patch's particle variable array is required. However, if the new 'owner' patch is located on another node, the particle information must be transmitted using MPI. In particle relocation, only the sending node's MPI process has knowledge of the new particle location and how many particles and their variables need to be transferred. The destination node must know the source node id first, as MPI nonblocking receives cannot be posted without a source rank id and a prepared buffer. The sender side packs all particles that have the same destination into a Uintah-defined scatter record and then sends this record though an MPI message. At the receive side, an empty scatter record buffer is prepared by calling MPI_Probe on each the neighboring rank ids. The framework then uses MPI rank id and buffer size pairs for later MPI nonblocking receive calls to actually obtain the moved particles. Thus, the cost for moving a particle off a node requires one MPI call on the sending side and two MPI calls on receive side, which is much more expensive than that of moving a particle inside a node or core. This last point is illustrated in Figure 4, when using our hybrid multithreaded/MPI approach, the number of particles that need to be sent is significantly reduced when using one MPI process per node as opposed to one MPI process per

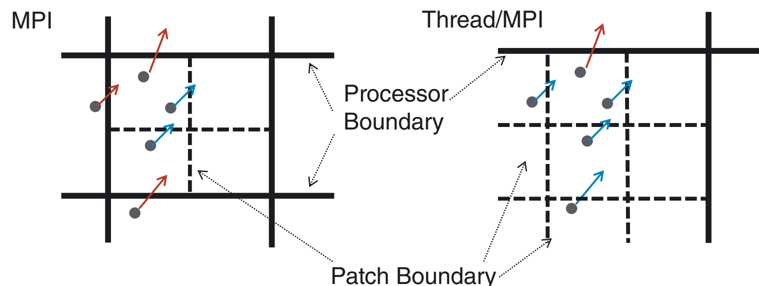

Figure 4. Particle relocation in the MPI and thread/MPI cases showing that there is less MPI communication in the thread MPI case because of the larger domain per MPI process.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

core, as all transfers internal to a node no longer use MPI. In Figure 4 left, of the processor only has four cells and when a particle crosses a solid line, it must be transferred using MPI to another processor. In contrast, in Figure 4 right, the single multicore MPI process can send particles to other patches without using MPI, as the processor boundary is now that of all the patches on the multicore node.

7.2. Load-balancing improvements

By using the hybrid scheduling approach described earlier, all tasks in the same node can be executed by any idle cores on that node. Moreover, the load balancer now profiles and predicts workload per node instead of per core. The accuracy of prediction is improved as changes of workload over a larger region are generally more stable than those over a small region. The average core load imbalance value was reduced from about 60% to 25% when running with nearly one patch per core at 100K cores on the Jaguar XK6 by using this approach.

7.3. Using lock-free data structures

A major overhead of the multithreaded scheduler approach [12] is due to the use of locking to protect shared data structures. This overhead keeps increasing with the number of cores per node as contention for acquiring locks also increases. A significant reduction in task waiting was obtained by eliminating large amounts of locking overhead through a redesign of some of the shared data structures so as to make them lock free. In particular, by using hardware-based atomic operations [58], which are supported by modern CPUs, these new data structures can be made to be more efficient than using the traditional pthread read–write lock and mutex. Although the use of these operations is common, we are not aware of their use in this way to make a data warehouse better able to handle requests for multiple core on a large high-performance computing platform.

Uintah has three types of variables:

1. Grid variables exist everywhere in the simulation grid for flow simulations. Grid variables on the same node can share a combined 3D array with different memory windows. As both the memory window and the 3D array are reference-counted, the associated memory will be deallocated when no longer referenced.
2. Particle variables exist only at a certain point for solid MPM simulations in Uintah. Particles in a Uintah patch are saved in a simple vector and indexed by a subset of that vector. By saving recent query ranges, particle sets are cached so as to speed up later queries.
3. Reduction variables are designed to combine multiple values provided by different tasks. The reduction variable operator must be associative, so that the order of computation will not alter the final value, apart from rounding errors. When the same reduction variable is computed multiple times on the same node, the locally reduced value will be updated immediately and stored in the data warehouse. After all local tasks have computed reduction variables, the global value of the reduction variable will be calculated through a call to MPI_AllReduce. This means that the cost of MPI_AllReduce depends on only the number of multicore nodes and not the number of cores, as is the case if each core has its own MPI process.

As mentioned earlier, Uintah variables can share the same memory window and 3D array. In fact, most of Uintah objects such as grid level, variable label, and MPI buffers are also reference-counted so that they can be easily deallocated when no longer needed. When running in multithreaded mode originally, reference counters were protected by a fixed size array of mutexes to ensure correctness. Once a new reference-counted object was created, a mutex from this array was assigned to it in a round-robin fashion. This design allowed many objects to share a mutex instead of each object creating its own as there may be thousands of reference-counted objects and dynamically creating thousands of mutexes is too expensive. However, potential false conflicts may exist when accessing two unrelated objects that happen to share the same mutex. This reference counting lends itself to the use of atomic operations [58]. A new reference counting class was implemented in Uintah by using *add_and_fetch* and *sub_and_fetch* atomic operations to replace the pthread mutex vector.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Table II. Shared data structure read/write locking times in a total wallclock runtime of 162 sec.

Data structure	Data warehouse	Four other main data structures
Read lock(s) times	7.86	0.166
Write lock(s) times	12.14	0.076

As described earlier, all tasks depend on the hashed multimap data warehouse to look up and save variables by using a patch id and variable name key. Each data warehouse has a pthread read/write lock to protect the hashed multimap. A read-only lock needs to be acquired when a task looks up a variable's memory address from data warehouse. A write lock needs to be acquired when a new variable need to put result into the data warehouse either from a computational task or from MPI message. Based on our timing results on Uintah read–write locks in Table II, the data warehouse lock was seen to be the largest single source of overhead far exceeding the four other main data structures that used locks on a node. For this reason, the hashed multimap data warehouse was redesigned to be lock free by using a two-step look-up strategy. During task-graph compilation, a hash map containing keys of all locally computed and required variables is created on-the-fly. This hash map will not change during task execution and can also be shared by multiple data warehouses until the task graph needs to be recompiled. The actual container of variables in each data warehouse will be a pre-allocated vector that has exactly the same size as this hash map. The value of this precomputed hash map will be the index to the container vector. When accessing a variable, the data warehouse will first use the hash map key to locate the variable from its private container vector. As the precomputed hash map is read-only during the task execution; no lock is needed to protect it. When updating the container vector, atomic operations are used to achieve consistency among threads.

A simplified version of the variable-inserting algorithm for putting a new variable into the data warehouse using the *compare_and_swap* atomic operation is shown in Algorithm 1. This algorithm inserts a variable to a linked list atomically. The head of this linked list is saved in the container vector. A variable-combining algorithm for reducing reduction variables using the *test_and_set* atomic operation is shown in Algorithm 2. This algorithm combines the new value of a variable with the existing value in the data warehouse. When multiple threads try to update this value, any two threads can compute the combined reduction value without being serialized. As these atomic operations are used on our redesigned data structures, pthreads locks are no longer needed to protect a long critical session when accessing the hashed multimap data structure.

8. EXPERIMENTAL RESULTS

This section will consider whether or not hybrid multithreaded/MPI approach can improve the performance and scalability of fluid–structure interaction problems in Uintah. We will also examine the performance difference between using a lock-free data structure and a traditional lock-protected data structure. The prototypical simulation study used in Section 5 was used to compare the hybrid

Algorithm 1 Variable inserting: put a variable into the data warehouse

function ADD(key, var)
  $idx \leftarrow hash_map[key]$
  $di \leftarrow$ **new** $dataitem()$
  $di^\wedge.var \leftarrow var$
  repeat
    $di^\wedge.next \leftarrow vector[idx]$
  until $compare_and_swap(\&vector[idx], di^\wedge.next, di)$
end function

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Algorithm 2 Variable combining: reduce a variable into the data warehouse

```
function REDUCE(key, var)
    idx ← hash_map[key]
    di ← new dataitem()
    di^.var ← var.clone()
    repeat
        old ← test_and_set(&vector[idx], 0)
        if thenold = 0
            old ← di
        else
            old^.var^.reduce(di^.var)
            delete di
        end if
        di ← test_and_set(&vector[idx], old)
    until di = 0
end function
```

multithreaded/MPI approach, the multithreaded/MPI with lock-free data warehouse approach, and the MPI approach.

8.1. Single node performance improvement

The first comparison is between the hybrid multithreaded/MPI approach and the lock-free data warehouse on a single shared memory node. This test was run on a Jaguar Cray XK-6 external node with 32 AMD interlagos cores. Three sets of strong scaling benchmark results were gathered by using the dynamic MPI scheduler, the decentralized hybrid multithreaded/MPI scheduler with the old pthread locking data warehouse and the decentralized hybrid multithreaded/MPI scheduler with the lock-free data warehouse. The input files for all three runs are identical and generate 887K particles on an AMR grid. Figure 5 shows the speedups when running with 2, 4, 8, 16, and 32 cores. The multithreaded scheduler with the lock-free data warehouse is 1.4× faster than with the pthreads locking data warehouse and 2.4× faster than when using MPI only.

Table III shows the execution time comparison results when running with different numbers of MPI processes and with different numbers of threads per MPI process. CPU affinity was used to

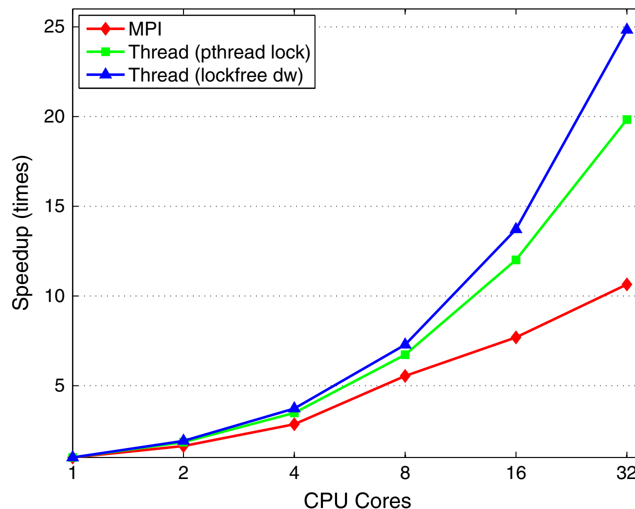

Figure 5. Performance comparison between the MPI, pthreads, and threaded lock-free approaches.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Table III. Mixed MPI and threads performance (total 32 cores).

Number of MPI process	32	16	8	4	2	1	1	1	1
Number of threads	1	2	4	8	16	32	64	128	1024
Execution times	5.18	3.77	3.05	2.79	2.62	2.22	2.40	2.41	2.47

guarantee that each thread is assigned to a dedicated core. Threads in the same MPI process were assigned to nearby cores so as to limit any possible cache coherence overheads. These benchmark results show that the optimized performance for this problem is achieved when running with the maximum number of available threads per MPI process. Also, when the number of threads was increased to be larger than the number of cores per node, the results of using 64 and 128 threads on a 32 core node are about 9% slower than when using 32 threads. As Uintah threads are lightweight, we are able to run 1024 threads per node, which is much larger than the 32 available cores and the additional scheduling overhead is about 11%. With MPI only, the execution times for 64 and 128 MPI processes on a 32-core node are 6.60 and 8.63 sec, 27% and 67% slower than 32-MPI process. We are unable to obtain a result for 1024-MPI processes per node as this exceeds available memory due to the heavy memory footprint of the MPI processes.

8.2. *Scalability improvement*

The scalability benchmark involved four runs with varying problem sizes using a similar approach to that in Section 5 and the same test problem but with larger particle and mesh sizes so as to make possible strong scalability to larger core counts. Each run uses a mesh that was refined by a factor of two in each dimension with respect to the previous run. As the mesh is refined where particles exist, eight times as many particles will be created than on the original coarse mesh. The number of particles created in the four runs are 7.1 million, 56.6 millon, 452.9 millon, and 3.62 billion, respectively. This leads to each run being approximately eight times as large as the previous run. As we will also generate weak scaling results at the same time, the problem size per node needs to be constant. Therefore, eight times as many cores were used from one run to the next. These tests were run on the Jaguar Cray XK-6 machine with up to 256K cores and with 16 cores per node. Figure 6 shows the scaling results for four benchmark tests. Weak scalability is represented by the almost-horizontal dashed lines and strong scalability by the almost-straight diagonal solid lines in the four runs. The strong scaling efficiency of the largest problem (the rightmost solid line) is 68% at 256K cores relative to the base case of 16K cores. It is worth remarking that the slight problem with

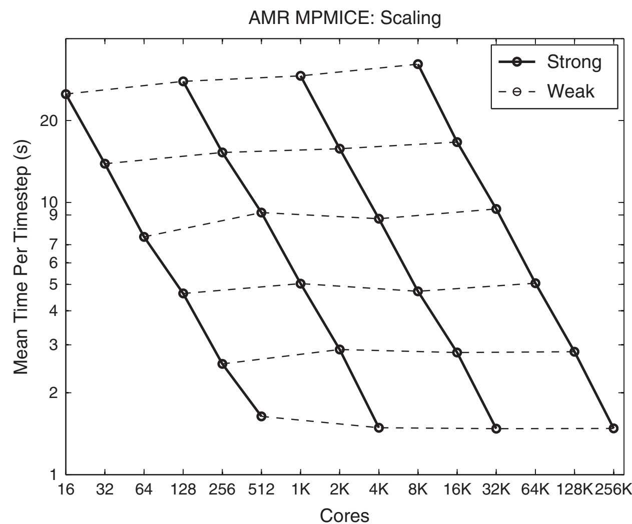

Figure 6. Strong and weak scaling on Jaguar XK-6 for the benchmark problem of Section 5 and Figure 2.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

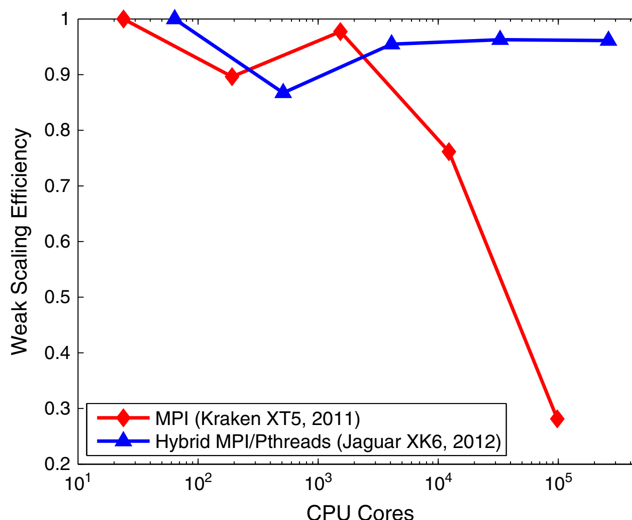

Figure 7. Weak scaling efficiency comparison between the MPI and threaded/MPI approaches.

weak scalablity at 1000 or so core comes from difficulties in assigning each core the same (limited) number of patches.

By using optimized hybrid multithreaded/MPI algorithms, the number of MPI ranks involved in communication is reduced without much performance overhead. For the MPMICE problem, overheads as percentages of local computations are as follows: 1.2% for allocation variables and put in the data warehouse, 6.9% for querying and assembling variables from the data warehouse, 0.6% for querying particle sets from data warehouse, 1.5% for reading tasks from work queue, and 1.2% for inserting tasks into the work queue with priority. The major single overhead (6.9%) of the Uintah framework is querying and assembling variables, in particular when (i) mapping patches between different AMR levels and (ii) allocating new memory and copying data to this new memory to hold both patch center and halo regions when pre-allocation is not possible.

This approach leads to better weak scaling. Figure 7 shows the weak scaling efficiency compared with the previous MPI only benchmark result from Section 5. The base cases to calculate efficiencies are 24 cores for MPI runs on Kraken and 64 cores for the hybrid multithreaded/MPI runs on Jaguar XK6. These two series are the same as the second from bottom dashed weak scaling line in Figure 2 and the bottom dashed weak scaling line in Figure 6. The results show significant improvement of weak scaling efficiency when using hybrid multithreaded/MPI approach especially on runs with large core counts.

9. CONCLUSIONS AND FUTURE WORK

These results presented here show great performance improvements and also show good scalability so far on 256K cores on Jaguar Cray XK-6 by using the hybrid multithreaded/MPI scheduler and the new lock-free data structures. However, alongside the lock-free data warehouse, many other shared data structures such as task queues, particle subset caches, and patch BVH trees still need to be redesigned to be lock free. Even though these data structures are accessed less frequently than the data warehouse, the waiting time for acquiring locks related to these structures is going to grow as the number of core per node increases. The future removal of all these locks and making Uintah fully lock free will improve the scaling here further on present and future many-core and multicore machines. The present decentralized scheduler will also be used instead of the current master–slave model used in the Uintah GPU scheduler [51]. This will require the current GPU controller thread routines to be rewritten to guarantee thread safety but will allow these scalability results to be extended to heterogeneous CPU–GPU architectures. A start on this has already been made while this paper has been under review [47].

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation under subcontract No. OCI0721659, the NSF OCI PetaApps program through award OCI0905068 and by DOE INCITE award CMB015 for time on Jaguar. Uintah was originally written by the University of Utah's Center for the Simulation of Accidental Fires and Explosions and funded by the Department of Energy, subcontract No. B524196. We would also like to thank Alan Humphrey, Todd Harman, and all those presently and previously involved with Uintah, Justin Luitjens, in particular. Finally, we would like to thank the reviewers for helping to improve the paper by their helpful and perceptive comments.

REFERENCES

1. Scientific grand challenges: crosscutting technologies for computing at the exascale report from the workshop held February 2–4, 2010. (Available from: http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Crosscutting_grand_challenges.pdf [Accessed July 31, 2012]).
2. Germain JD, McCorquodale J, Parker SG, Johnson CR. Uintah: a massively parallel problem solving environment. *HPDC'00: Ninth IEEE International Symposium on High Performance and Distributed Computing*, IEEE Computer Society: Washington, DC, USA, 2000; 33.
3. Parker SG. A component-based architecture for parallel multi-physics PDE simulation. *Future Generation Computer Systems* 2006; **22**(1):204–216.
4. Parker SG, Guilkey J, Harman T. A component-based parallel infrastructure for the simulation of fluid–structure interaction. *Engineering with Computers* 2006; **22**(3):277–292.
5. Berzins M. Status of release of the Uintah computational framework. *SCI Technical Report, No. UUSCI-2012-001*, SCI Institute, University of Utah, 2012. (Available from: http://www.sci.utah.edu/publications/SCITechReports/UUSCI-2012-001.pdf [Accessed at July 31, 2012]).
6. Guilkey JE, Harman TB, Xia A, Kashiwa BA, McMurtry PA. An Eulerian–Lagrangian approach for large deformation fluid–structure interaction problems, Part 1: algorithm development. In *Fluid Structure Interaction II*, Vol. 36. WIT Press: Cadiz, Spain, 2003; 143–156.
7. Harman TB, Guilkey JE, Kashiwa BA, Schmidt J, McMurtry PA. An Eulerian–Lagrangian approach for large deformation fluid–structure interaction problems, Part 2: multi-physics simulations within a modern computational framework. In *Fluid Structure Interaction II*, Vol. 36. WIT Press: Cadiz, Spain, 2003; 157–166.
8. Sulsky D, Chen Z, Schreyer HL. A particle method for history-dependent materials. *Computer Methods In Applied Mechanics And Engineering* 1994; **118**:179–196.
9. Sulsky D, Zhou SJ, Schreyer HL. Application of a particle-in-cell method to solid mechanics. *Computer Physics Communications* 1995; **87**:236–252.
10. Luitjens J, Berzins M. Improving the performance of Uintah: a large-scale adaptive meshing computational framework. *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS10),* IEEE, Atlanta, GA, USA, April 2010; 1–10.
11. Luitjens J, Berzins M. Scalable parallel regridding algorithms for block-structured adaptive mesh refinement. *Concurrency And Computation: Practice And Experience* 2011; **23**(13):1522–1537.
12. Meng Q, Berzins M, Schmidt J. Using hybrid parallelism to improve memory use in the Uintah framework. *Proceedings of the 2011 TeraGrid Conference,* ACM, ACM, July 2011; 24.
13. Berzins M, Luitjens J, Meng Q, Harman T, Wight CA, Petersonr J. Uintah a scalable framework for hazard analysis. *Proceedings of the 2010 TeraGrid Conference,* ACM, Pittsburgh, Pennsylvania, USA, July 2010; 3.
14. Attaway SW, Heinstein MW, Swegle JW. Coupling of smooth particle hydrodynamics with the finite element method. *Nuclear Engineering and Design* 1994; **150**:199–205.
15. Brown K, Attaway S, Plimpton SJ, Hendrickson B. Parallel strategies for crash and impact simulations. *Computer Methods in Applied Mechanics and Engineering* 2000; **184**:375–390.
16. Dostl Z, Vondrk V, Hork D, Farhat C, Avery P. Scalable FETI algorithms for frictionless contact problems. In *Domain Decomposition Methods in Sciences and Engineering XVII*, Vol. 60, Langer U, *et al.* (eds), Lecture Notes in Computational Science and Engineering (LNCSE). Springer: Berlin, 2008; 263–270.
17. Bungartz H-J, Benk J, Gatzhammer B, Mehl M, Neckel T. Partitioned simulation of fluid–structure interaction on Cartesian grids. In *Fluid–Structure Interaction—Modelling, Simulation, Optimisation, Part II of LNCSE*, Vol. 73, Bungartz H-J, Mehl M, Schfer M (eds). Springer: Berlin, Heidelberg, 2010; 255–284.
18. Grinberg I, Wiseman Y. Scalable parallel simulator for vehicular collision detection. *2010 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*, 15; 116–121.
19. Gotz J, Iglberger K, Sturmer M, Rude U. Direct numerical simulation of particulate flows on 294912 processor cores. *2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC),* ACM/IEEE, Vol. 13, New Orleans, LA, USA, November 2010; 1–11.
20. Wissink AM, Hornung RD, Kohn SR, Smith SS, Elliott N. Large scale parallel structured AMR calculations using the SAMRAI framework. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing,* ACM/IEEE: Denver, Colorado, USA, November 2001; 22–22.

Copyright © 2013 John Wiley & Sons, Ltd.

21. Steensland J, Söderberg S, Thuné M. A comparison of partitioning schemes for blockwise parallel SAMR algorithms. In *Proceedings of the 5th International Workshop on Applications Parallel Computing, New Paradigms for HPC in Industry and Academia*. Springer-Verlag: London, Uk, 2001; 160-169.
22. Fryxell B, Olson K, Ricker P, Timmes FX, Zingale M, Lamb DQ, Macneice P, Rosner R, Rosner JW, Truran JW, Tufo H. FLASH an adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *The Astrophysical Journal Supplement Series* 2000; **131**:273–334.
23. Daley C, Bachan J, Couch S, Dubey A, Fatenejad M, Gallagher B, Lee D, Weide K. Adding shared memory parallelism to FLASH for many-core architectures. In *TACC-Intel Highly Parallel Computing Symposium*, April 2012.
24. Daley C, Vanella M, Dubey A, Weide K, Balaras E. Optimization of multigrid based elliptic solver for large scale simulations in the FLASH code. In *Concurrency and Cmputation: Practice and Experience*. John Wiley & Sons, Ltd: Chichester, UK, 2012.
25. O'Shea B, Bryan G, Bordner J, Norman M, Abel T, Harkness R, Kritsuk A. Introducing Enzo, an AMR Cosmology Application in "Adaptive Mesh Refinement—Theory and Applications". In *Lecture Notes in Computational Science and Engineering*, Vol. 41, Plewa T, Linde T, Weirs VG (eds). Springer: London, UK, 2005; 341-350.
26. Wise JH, Abel T. Enzo+Moray: radiation hydrodynamics adaptive mesh refinement simulations with adaptive ray tracing, 15th March 2011. arXiv:1012.2865v2.
27. Seidel EL, Allen G, Brandt S, Lraffler F, Schnetter E. Simplifying complex software assembly: the component retrieval language and implementation. *Proceedings of the 2010 TeraGrid Conference*, ACM New York, NY, USA, 2010.
28. Oliker L, Carter J, Beckner V, Bell J, Wasserman H, Adams M, Ethier S, Schnetter E. Large-scale numerical simulations on high-end computational platforms. In *Performance Tuning of Scientific Applications, Chapter 6*, Bailey DH, Lucas RF, Williams SW (eds). Chapman & Hall/CRC Computational Science Series: London, UK, 2011.
29. Schnetter E, Ott C, Allen G, Diener P, Goodale T, Radke T, Seidel E, Shalf J. Cactus framework: black holes to gamma ray bursts. In *Petascale Computing: Algorithms and Applications, chapter 24*, Bader DA (ed.). Chapman & Hall/CRC Computational Science Series: London, UK, 2008.
30. Burstedde C, Wilcox LC, Ghattas O. p4est: scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing* 2011; **33**(3):1103–1133.
31. Schraml SJ, Kendall TM. Scalability of the CTH Shock Physics Code on the Cray XT. *DoD High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC)* June 2009; **2009**:450,453, 15–18.
32. Kashiwa BA, Rauenzahn RM. A multimaterial formalism. *Technical Report LA-UR-94-771*, Los Alamos National Laboratory, Los Alamos, 1994.
33. Kashiwa BA, Gaffney ES. Design basis for cfdlib. *Technical Report LA-UR-03-1295*, Los Alamos National Laboratory, Los Alamos, 2003.
34. Guilkey JE, Harman TB, Banerjee B. An Eulerian–Lagrangian approach for simulating explosions of energetic devices. *Computers and Structures* 2007; **85**:660–674.
35. Harlow FH, Amsden AA. Numerical calculation of almost incompressible flow. *Journal of Computational Physics* 1968; **3**:80–93.
36. Kashiwa BA, Rauenzahn RM. A cell-centered ice method for multiphase flow simulations. *Technical Report LA-UR-93-3922*, Los Alamos National Laboratory, Los Alamos, 1994.
37. Tran LT, Kim J, Berzins M. Solving time-dependent PDEs using the material point method, a case study from gas dynamics. *International Journal for Numerical Methods in Fluids* 2009; **62**(7):709–732.
38. Brackbill JU, Ruppel HM. FLIP: a method for adaptively zoned, particle-in-cell calculations of fluid flow in two dimensions. *Journal of Computational Physics* 1986; **65**:314–343.
39. Brackbill JU. Particle methods. *International Journal for Numerical Methods in Fluids* 2005; **47**:693–705.
40. Guilkey JE, Weiss JA. Implicit time integration for the material point method: quantitative and algorithmic comparisons with the finite element method. *International Journal for Numerical Methods in Engineering* 2003; **57**:1323–1338.
41. Tran LT, Berzins M. IMPICE Method for Compressible Flow Problems in Uintah. *International Journal for Numerical Methods in Fluids* 20 June 2012; **69**(5):926–965. DOI: 10.1002/fld.2620. Published online 20 July.
42. Steffen M, Kirby RM, Berzins M. Decoupling and balancing of space and time errors in the material point method (MPM). *International Journal for Numerical Methods in Engineering* 2010; **82**(10):1207–1243.
43. Wallstedt PC, Guilkey JE. An evaluation of explicit time integration schemes for use with the generalized interpolation material point method. *Journal of Computational Physics* 2008; **227**:9628–9642.
44. Sadeghirad A, Brannon RM, Burghardt J. A convected particle domain interpolation technique to extend applicability of the material point method for problems involving massive deformations. *International Journal for Numerical Methods in Engineering* 2011; **86**(12):1435–1456.
45. Luitjens J, Berzins M, Henderson TC. Parallel space-filling curve generation. *Concurrency and Computation Practice and Experience* 2007; **19**(10):1387–1402. DOI: 10.1002/cpe.1179.
46. Meng Q, Luitjens J, Berzins M. Dynamic task scheduling for the Uintah framework. *Proceedings of the 3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS10),* IEEE, New Orleans, LA, USA, November 2010; 1–10.
47. Meng Q, Humphrey A, Berzins M. The Uintah framework: a unified heterogeneous task scheduling and runtime system. *Digital Proceedings of The International Conference for High Performance Computing, Networking, Storage*

Copyright © 2013 John Wiley & Sons, Ltd.

and Analysis, SC12 2nd International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC 2012, IEEE, Salt Lake City, UT, USA, November 2012. (to appear).

48. Berger MJ, Colella P. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics* 1989; **82**:64–84.
49. Ericson C. Real-time collision detection, Page 236-237.
50. Berzins M, Meng Q, Schmidt J, Sutherland J. DAG-based software frameworks for PDEs. *Proceedings of the 2012 Workshop on Algorithms and Programming Tools for Next-Generation High-Performance Scientific Software(HPSS)*, Springer: Bordeaux, France, 2012; 324–333.
51. Humphrey A, Meng Q, Berzins M, Harman T. Radiation modeling using the Uintah heterogeneous CPU/GPU runtime system. *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment,* ACM, Chicago, Illinois, USA, July 2012; 4.
52. Schmidt J, Berzins M, Thornock J, Saad T, Sutherland J. Large scale parallel solution of incompressible flow problems using Uintah and hypre. *SCI Technical Report, No. UUSCI-2012-002*, SCI Institute, University of Utah, 2012. (accepted for Proceedings of CCGrid13, Delft, May 2013).
53. Guilkey J, Harman T, Luitjens J, Schmidt J, Thornock J, de St. Germain JD, Shankar S, Peterson J, Brownlee C. Uintah user guide version 1.1. *SCI Technical Report, No. UUSCI-2009-007*, SCI Institute, University of Utah, 2009.
54. Schmidt JA. Uintah application development. *SCI Technical Report, No. UUSCI-2008-005*, University of Utah, 2008. (Available from: http://www.sci.utah.edu/publications/SCITechReports/UUSCI-2008-005.pdf [Accessed at July 31, 2012]).
55. de Oliveira DCB, Humphrey A, Rakamaric Z, Meng Q, Berzins M, Gopalakrishnan G. Crash early, crash often, explain well: practical formal correctness checking of million-core problem solving environments for HPC. *Proceedings of the 5th International Workshop on Software Engineering for Computational Science and Engineering (SE-CSE),* IEEE, San Francisco, CA, USA, May 2013; 75–83.
56. Bennett JG, Haberman KS, Johnson JN, Asay BW, Henson BF. A constitutive model for the shock ignition and mechanical response of high explosives. *Journal of the Mechanics and Physics of Solids* 1998; **46**:2303.
57. Luitjens J. The scalability of parallel adaptive mesh refinement within Uintah. *Doctoral Thesis*, University of Utah, 2011.
58. Fraser K. Practical lock-freedom. *Technical Report UCAMCL-TR-579*, University of Cambridge Computer Laboratory, 2004.
59. Attaway SA, Barragy EJ, Brown KH, Gardner DR, Hendrickson BA, Plimpton SJ, Vaughan CT. Transient solid dynamics simulations on the Sandia/Intel teraflop computer. *ACM/IEEE 1997 Conference on Supercomputing,* ACM/IEEE, San Jose, California, USA, November 1997; 58.
60. Barker AT, Cai X-C. Scalable parallel methods for monolithic coupling in fluid–structure interaction with application to blood flow modeling. *Journal of Computational Physics* 2010; **229**(3):642–659.
61. Kale LV, Bohm E, Mendes CL, Wilmarth T, Zheng G. Programming petascale applications with Charm++ and AMPI. *Petascale Computing: Algorithms and Applications* 2007; **1**:421–441.
62. Dedner A, Klöfkorn R, Nolte M, Ohlberger M. A generic interface for parallel and adaptive scientific computing: abstraction principles and the DUNE-FEM module. *Computing* 2010; **90**:3–4, 165–196.

Copyright © 2013 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe