

An Events Based Algorithm for Distributing Concurrent Tasks on Multi-Core Architectures

David W. Holmes^{*a}, John R. Williams^b, Peter Tilke^c

^aPostdoctoral Fellow, Civil and Environmental Engineering, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139-4307

^bProfessor of Information Engineering, Civil and Environmental Engineering and Engineering Systems, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139-4307

^cScientific Advisor, Department of Mathematics and Modeling, Schlumberger-Doll Research Center, 1 Hampshire Street, Cambridge, MA 02139-1578

Abstract

In this paper, a programming model is presented which enables scalable parallel performance on multi-core shared memory architectures. The model has been developed for application to a wide range of numerical simulation problems. Such problems involve time stepping or iteration algorithms where synchronization of multiple threads of execution is required. It is shown that traditional approaches to parallelism including message passing and scatter-gather can be improved upon in terms of speed-up and memory management. Using spatial decomposition to create orthogonal computational tasks, a new task management algorithm called *H-Dispatch* is developed. This algorithm makes efficient use of memory resources by limiting the need for garbage collection and takes optimal advantage of multiple cores by employing a “hungry” pull strategy. The technique is demonstrated on a simple finite difference solver and results are compared to traditional MPI and scatter-gather approaches. The H-Dispatch approach achieves near linear speed-up with results for efficiency of 85% on a 24-core machine. It is noted that the H-Dispatch algorithm is quite general and can be applied to a wide class of computational tasks on heterogeneous architectures involving multi-core and GPGPU hardware.

Key words: Multi-Core, Port Based Programming, H-Dispatch, Numerical Simulation
PACS: 07.05.Tp, 89.20.Ff, 95.75.Pq

1. Introduction

As noted by Justin Ratner, CTO of Intel Corporation, “...multi-core and many-core processors require a new generation of programming tools” [1]. The shift of computer processor manufacturers, such as Intel and A.M.D. to multi-core / shared memory architectures, means that traditional numerical simulators based on message passing concurrency tools like MPI [2], must be either redesigned or updated to take advantage of the multiple cores and shared memory. In order to take advantage of multi-core architectures it is necessary not only to coordinate multiple threads of execution, but also to be conscious of memory management issues such as cache

*Corresponding Author.

Email address: dholmes@mit.edu (David W. Holmes)

efficiency, garbage collection and thread safety. Here we examine the coordination and synchronization problems involved in a typical numerical simulation that evolves over time. The finite difference (FD) method is used here as an example but the issues covered are relevant to a wide range of numerical techniques.

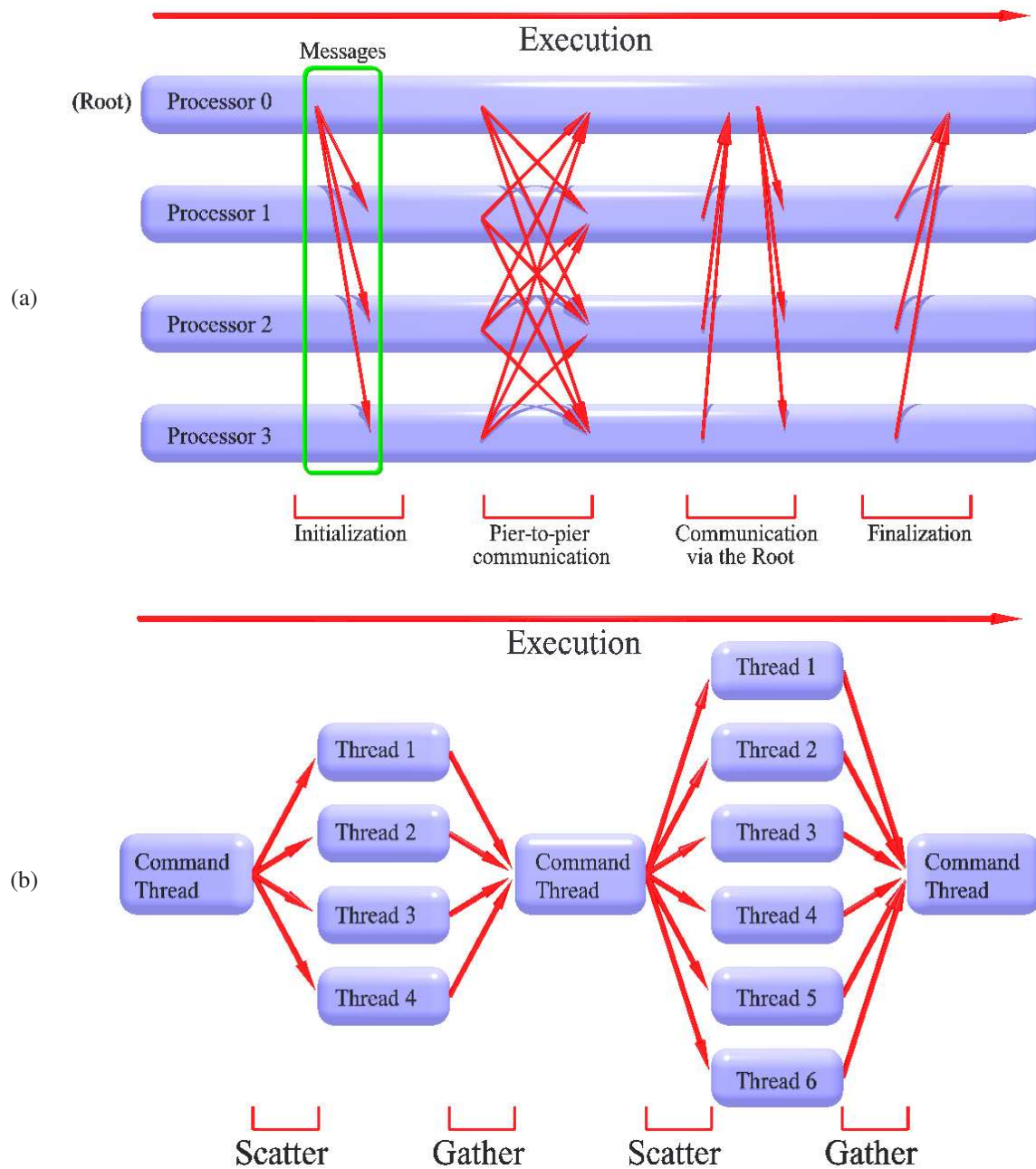
Over the past 30 years, significant effort has focused on developing numerical simulators for distributed memory, cross-machine parallel architectures. MPI [2] has become the standard messaging library used for such applications. Authors such as Eadline [3], Fox [4] and Qiu et al [5] have identified the potential of implementing MPI on shared memory multi-core architectures with the key advantage of such a strategy being the ability to reuse existing, well developed and well benchmarked simulation suites in which industry has significant investment¹. MPI, implemented on Multi-core, will typically accomplish parallelism by creating a separate MPI process for each core, see Fig. 1 (a). Communication is then achieved across process boundaries via serialized message passing. While an advantage of this is the guarantee of memory isolation, the execution overhead related to message packaging can be significant. For classes of problems where communication is minimal, excellent performance can be achieved using such an approach on multi-core (see [3, 4, 5]).

An alternative strategy to using process boundaries to isolate memory (a method that inherently avoids race and deadlock conditions) is to share memory across cores and to provide memory isolation across threads by developing orthogonal computational tasks. For example, a typical loop over an array can often be split into multiple sub-loops over non-overlapping portions of the array. This methodology forms the basis for a multitude of shared memory approaches to concurrent programming.

Developed some time ago for shared memory multi-processors, OpenMP [6] is a concurrency package which makes optimum use of shared memory and is finding wide application on multi-core. OpenMP supplies various command directives that allow multi-threaded execution of process intensive loops. One of the most powerful advantages of such a methodology is in the ability to modify legacy serial code to leverage multiple processing cores by the simple addition of directive commands. A more recent concurrency package, Cilk++ [7], uses a similar approach to asynchronous implementation with several additionally powerful features that intelligently handle thread safety (see [7]). These approaches to asynchronous programming often adopt a *scatter-gather* type programming model, see Fig. 1 (b). While the ability to easily parallelize existing serial code is of significant benefit, the serial portion of a scatter-gather program can strongly inhibit scalability (see Amdahl [8]). Consequently, there is significant advantage to using an alternate programming model which approaches parallelism at the problem level rather than at the loop level.

Where OpenMP and Cilk++ leverage concurrency through the decomposition of loops, the techniques developed in this work rely on a hybridized spatial decomposition more common to MPI approaches to parallel computational physics. Here, orthogonal computational tasks are associated with non-overlapping spatial divisions of the problem domain. Memory isolation is then provided by *spatial localization*. When coordinating tasks of this nature, it will be shown that there are advantages to using an approach to concurrency, which provides greater flexibility than can be achieved with conventional loop-based scatter-gather.

¹Note that we make special mention here of benchmarking. Simulation software which is used to test and verify equipment and components in fields with precise tolerances such as aerospace and big industry must undergo significant benchmarking to ensure the results produced are accurate and indicative of actual physics. As such, it is important to remember that industrial investment in legacy software is not limited to its development, but also in the many years of validation which make them reliable scientific tools.



An alternate approach to concurrency and coordination for multi-core, developed by Chrysanthakopoulos and co-workers (see for example [9, 10, 11, 12]), and based on earlier work by Stewart [13], is the use of *port* based abstractions. Such ports allow the exchange of messages between active asynchronous application threads. Where messaging between MPI processes involves posting serialized data packets to TCP/IP ports, the ports we discuss here are simply abstractions implemented in code which give the appearance of port behavior. Messages in port based programming involve the cross-thread exchange of pointers to locations in shared memory. Program events can be aligned with the arrival of such messages on a port, allowing a high level of flexibility in program development. Port based programming is able to take advantage of the speed and efficiency of shared memory, while leveraging the message passing and events based coordination of an MPI type implementation. As such, applications written using ports are not restricted to any single programming model, rather, such a methodology provides the freedom to develop application specific programming models which can be optimized for shared memory, multi-core architectures.

In this work we use such a port based approach to concurrency to develop a programming model capable of distributing spatially based task divisions. While developed for numerical simulation applications, the developed method could be used for any problem where it is possible to divide the numerical work based on space (for example databases, image processing etc).

In what follows, we first identify some major challenges associated with parallel implementation of numerical simulation codes on multi-core. After addressing some of the limitations of existing message passing and scatter-gather type programming models, we present the developed H-Dispatch programming model, which provides asynchronous utilities appropriate for most, if not all, simulation applications. Functionality for this programming model is provided by the H-Dispatch task distribution class library which is then presented. It will be demonstrated that the developed abstractions are beneficial for a wider field of applications than just numerical simulation. Results from performance evaluations of a finite difference code, implemented using the developed programming model, are then presented.

2. Multi-Core Challenges Specific to Numerical Simulation

As we have noted, the implementation of simulation software in parallel is by no means a new topic, with decades of research having been carried out on cross-machine, distributed memory applications. While it may seem reasonable for a numerical simulator written for multi-core to resemble one written using MPI or an equivalent cross-machine concurrency platform, some of the key differences between multi-core and multi-machine mean that software optimized for one platform, may overlook some key advantages of the other. Understanding the contrast in programming priorities between the two hardware architectures is important. In this section we outline some of the general challenges for multi-core numerical simulators that influence the development of the programming model.

2.1. Periodic Synchronization

The need for repeated synchronization within an application has significant implications for the associated programming model used. For cross machine implementations, synchronization requires costly inter-machine communication. As such, emphasis is placed on code structures which minimize communication. For multi-core, synchronization is critical for thread safety as it provides guarantees on the state of threads. For example, it is critical that all results from

a previous step have contributed to update calculations, before the associated memory is overwritten. Fortunately, the overhead associated with synchronization in multi-core programming is many orders of magnitude smaller than that associated with the cross machine equivalent, so synchronization can be used more readily within an application. This represents a fundamental difference between programming models optimized for cross machine architectures and those optimized for multi-core.

2.2. Spatial Reasoning and Load Balancing

In parallel software development load balancing is achieved by evenly distributing numerical tasks across the available processor resources. Assuming a wholly parallel application, an ideal load balancing will remove processor redundancies and provide scalability. For numerical simulation codes, this division of tasks is almost always achieved through some form of a priori spatial division². The way in which space is divided and then allocated to processors is strongly influenced by machine architecture.

Broadly speaking, ‘space’ in numerical simulation terms, refers to a discrete number of integration points such as nodes, elements or particles, arranged in virtual space to represent some region of physical space. Field values calculated at these points provide approximate representations to that which would occur in an actual continuum. In the majority of cases, numerical simulation is used to solve non-local problems in physics where the state of some region in space is mutually dependant on that in some other region or regions. Correspondingly, calculations at integration points may require information from adjacent points to properly reproduce a physical phenomena. Such information may be readily available in shared memory or may have to be retrieved via communication from a remote source (such as from another machine). This can have important implications for spatial reasoning.

Conventional cross-machine type parallel implementations involve dividing the model into specific spatial sub-domains, a methodology called *Domain Decomposition* [15], see Fig. 2 (a). Each sub-domain is then allocated to a machine for processing. The spatial grouping of calculation points is done to maximize the proportion of interacting points on each machine’s local memory and to minimize the number of points along domain boundaries, whose information must be communicated to adjacent sub-domains during the analysis. In practice, each sub-domain carries duplicates or “ghost regions” of adjacent sub-domain boundary points so as to reduce communication frequency to once per time step. From the perspective of load balancing a cross-machine application, the priority is to ensure the computational load of each sub-domain is of similar size. This assumes that each processor is of comparable speed and can be referred to as being *predictive load balancing*. This methodology has been shown to produce reasonably scalable parallel performance on distributed memory architectures (see for example [16, 17]).

Trivially, on multi-core architectures, all memory is locally shared memory. As such, the exchange of information between interacting calculation points is as straight forward as locating and reading the necessary information from the system RAM³. Consequently, a multi-core implementation does not necessitate the same careful grouping of processor tasks that a cross-machine application might. While a similar methodology to Domain Decomposition could be

²although examples of operation based decomposition have been used to some success, see for example [14]

³Note here that while modern multi-core architectures share main memory, data is transmitted from the RAM through a bus into a cache hierarchy on each core. While caching is handled at the hardware level, program structure can be optimized for cache performance (see Tian and Shih [18]). This will be addressed, later in Section 5.5.

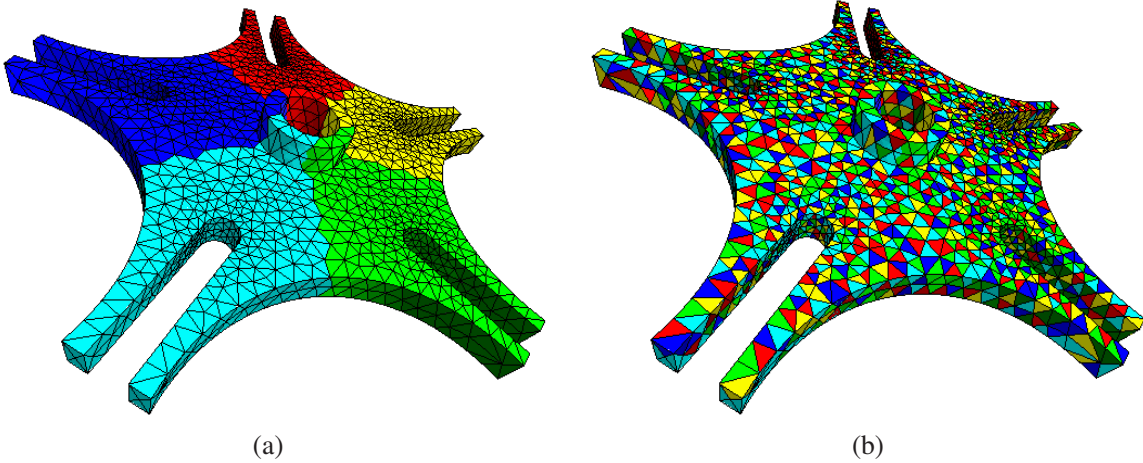


Figure 2: (color online). (a) Diagram of domain decomposition with the 5 different colors each representing a sub-domain to be allocated to a processor in a *predictive load balancing*, (b) diagram of fine grained domain distribution with each of the 5 colors corresponding to an element processed in one of 5 different processing cores, the distribution of which is determined at runtime in an *events based load balancing*.

easily implemented for multi-core, in our experience, the expense associated with the actual decomposition algorithm is not justified. Instead, it would seem advantageous to select a method of task distribution which can achieve a more robust load balancing.

Both an advantage and challenge of multi-core is the potential for variable processor workload. While it is reasonable to assume processors in a dedicated multi-machine cluster system may perform similarly, multi-core introduces the possibility for there to be variable processor efficiency or availability as different operating system threads; threads from other running applications or even output threads from the same application vie for processor resources. As such, an alternate approach to the aforementioned predictive load balancing would seem advantageous. One solution is to use a much finer grained division of space, when compared to domain decomposition. On a shared memory multi-core platform, it is perfectly feasible to consider that small groups of one or more calculation points could be allocated to each processor with each processor handling many such groups over the progress of a single step. Such a methodology can be termed *Fine Grained Domain Distribution* and because of ready access to shared memory, there is no requirement for the groups operated on by any one processor to be in spatial proximity to one another, see Fig. 2 (b). This eliminates the need for complex decomposition algorithms. Additionally, by carefully aligning memory write permissions with such spatial groupings, thread safety can be easily managed.

Given that an appropriate task distribution mechanism is implemented, such a fine grain approach has the potential to allow *events based load balancing*, i.e. tasks get allocated to processors when they become free, in an events based fashion. This is extremely important in dynamic multi-core environments.

2.3. Dynamic Evolution of the Numerical Task

Unlike many conventional parallel applications, numerical simulation codes have a propensity to involve numerical tasks, which dynamically evolve in size. Several examples of this are provided in Fig. 3 with the major feature being that in many applications, accuracy or efficiency

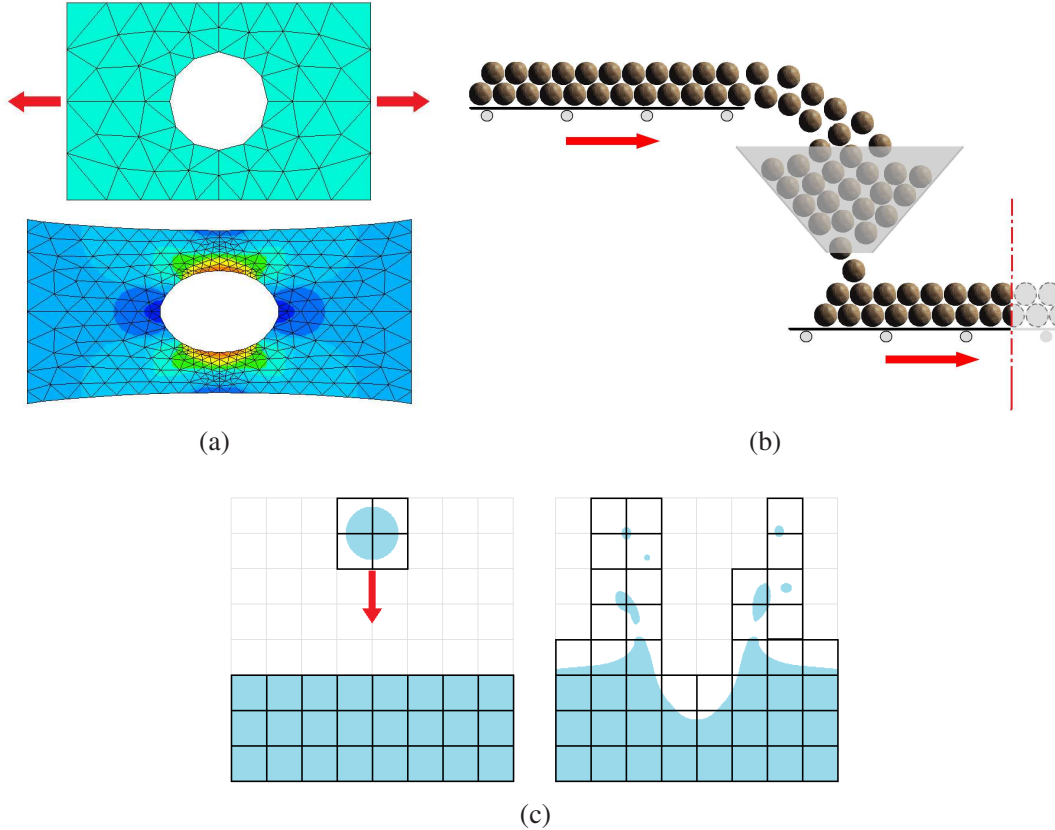


Figure 3: (color online). (a) Adaptive remeshing on an FEM plate deformation analysis, (b) removal of particles outside of the critical zone in a DEM conveyor belt and hopper analysis, (c) increase in hashed cell number necessary to accommodate free surface movement for an SPH falling drop analysis.

dictates that the number of calculation points may differ from step to step in a solution dependent fashion. This has proven to be an extremely challenging feature to incorporate into previous parallel numerical simulation codes (see [19] for an example of parallel adaptive meshing). This is primarily a result of the difficulties involved with load balancing such problems and the inability of conventional scatter-gather type frameworks to accommodate a changing number of executable tasks. In this paper, accommodating such task evolution in a versatile way has been a key objective.

The development of a multi-core programming model, which allows periodic synchronization, events based load balancing and accommodation of dynamic task evolution will be outlined in the sections which follow.

3. Development of the Programming Model

The term ‘Programming Model’ can be used to describe a multitude of different patterns in computer science. In the context of this work, we use the term to describe a programmatic framework, generally appropriate for the implementation of numerical simulation software. Where a

numerical method will have some task to be carried out at a number of points and over some time-frame of interest, the programming model represents all other aspects of a programmatic implementation. It manages time stepping, task distribution and concurrency. The message passing and scatter-gather models of Fig.'s 1 (a) and (b) represent two such programming models. In what follows we identify some of the specific limitations of the scatter-gather programming model for numerical simulation applications and then present a new programming model, capable of addressing these limitations and making optimum use of the advantages of multi-core architectures.

3.1. Drawbacks with Conventional Scatter-Gather Implementations

Perhaps the most obvious drawback of the conventional scatter-gather programming model (Fig. 1 (b)) is its inability to accommodate a dynamically varying number of tasks. As discussed in Section 2.3, a numerical simulation application may require the task count in an asynchronous portion of code to evolve. A conventional scatter-gather type programming model would have great difficulty robustly accommodating such solution dependent changes when they occur between the scatter and gather operations. From a programmatic standpoint, a gather point will generally identify synchronization as having occurred once a predetermined number of threads have completed. Modifying this number during execution can prove to be extremely unreliable. An alternative to evolving the gather process would be to use recursive scatter-gather operations to accommodate the desired task evolution⁴, however, such an approach requires task duplication and increased proportions of serial code and, as such, will reduce performance markedly.

An additional serious but less obvious drawback with such a programming model, implemented using managed memory languages such as C# and Java, is the poor memory efficiency and memory implications for CPU usage that result. Referring to Fig. 4 (a), at each time step, a separate virtual thread is cast for each of $nData$ calculation points. By design, the number of calculation points far exceed the number of available processors and so these virtual threads are enqueued and mapped to processors as they become available⁵. For complex applications, the task carried out in each virtual thread, i.e. $f(d1), f(d2), \dots, f(dnData)$, may involve significant memory allocation for local variables. In a managed language, once the memory associated with local thread variables is no longer needed, it must be collected by the garbage collection mechanism. From experience, the frequency of such collection events is unpredictable, commonly allowing significant local thread memory accumulation between collections. In the worst case, this accumulation can overflow available physical memory and cause failure. Additionally, the execution of a garbage collection event itself, has serious implications for concurrent performance. During garbage collection, the managed heap is reordered and all associated concurrent threads are blocked while memory pointers are updated. Because such events result in synchronous execution, they can diminish scalability markedly, as will be demonstrated later in Section 5.1. Gains in memory efficiency can, however, be achieved by using an alternate programming model.

3.2. The Evolved H-Dispatch Programming Model

A simple solution to minimize the memory footprint of parallel threads is to reduce the thread count to a single active thread for each of the $nProcessor$ available processing cores. Each of

⁴i.e. identify the new task number in one, carry out calculations in another, etc

⁵This mapping can generally be performed with a minimum of numerical overhead, depending on the OS scheduler and concurrency package used.

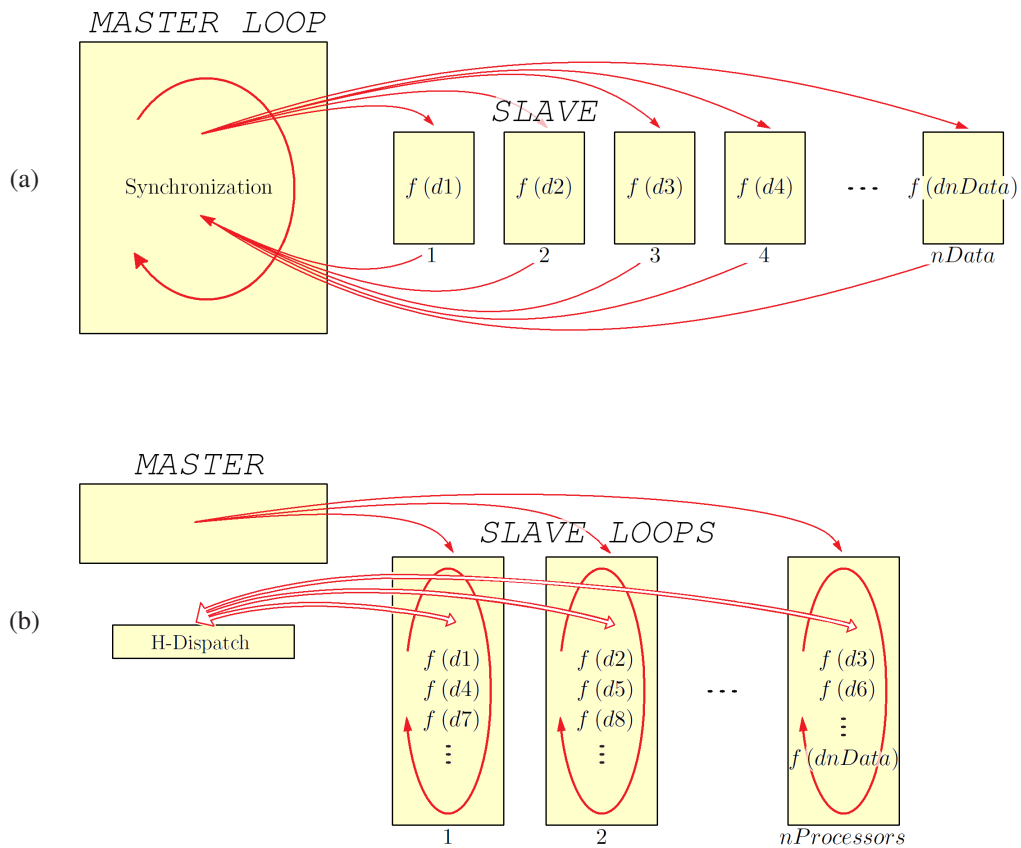


Figure 4: (color online). Structures for scatter-gather parallelization (a) master time loop with $nData$ number of independent slave threads \rightarrow memory and garbage collection intensive, (b) a slave time loop thread for each processing core ($nProcessors$), each processing multiple data values without re-instantiation \rightarrow significant reductions in memory usage and need for garbage collection (looping arrows designate time loops).

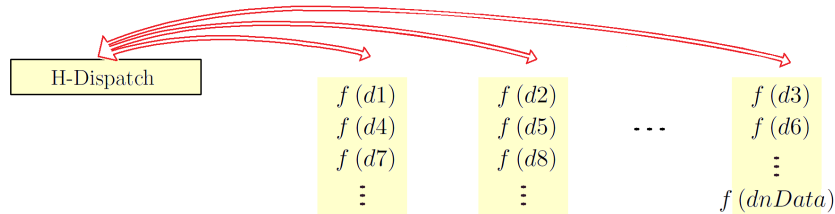


Figure 5: (color online). Portion of Fig 4 corresponding to the developed *H-Dispatch* mechanism.

those ‘real’ threads can then sequentially process multiple data values in an iterative fashion, reusing the same local variable memory allocation, Fig. 4 (b). In this way, the required execution memory will be reduced and the need for garbage collection will be eliminated. Where the scatter-gather programming model relied on the OS to map tasks to threads in a load balanced way, this alternate approach requires a controlling dispatch mechanism which manages the distribution of tasks from a global queue, onto available persistent threads. We have named the dispatch mechanism developed in this work the *H-Dispatch*. As will be addressed in detail in the next section, the H-Dispatch mechanism also controls thread synchronization by way of port based messaging.

An additional point of note is the positioning of the time loops in Fig. 4 (b). By moving the time looping into the concurrent portion of the programming model, a single set of incumbent parallel threads remain active throughout the entire program execution. Removing the need to re-instantiate threads further increases memory efficiency and removes additional garbage collection. It also ensures the programming model is wholly parallel, improving the potential for application scalability.

Compared to the scatter-gather model of Fig. 4 (a), Fig. 4 (b) represents an improved programming model for numerical simulation. The ability of this model to accommodate dynamically varying task number depends on the actual implementation of the dispatch mechanism (Fig. 5). The development of the dispatch component of the programming model to accommodate dynamically varying task number, effective synchronization and events based load balancing, will be the focus of the remainder of this paper.

4. Implementation of H-Dispatch

The functionality of the programming model illustrated in Fig. 4 (b) is wholly provided by the dispatch component. To achieve the features identified in Section 2 as being important for numerical simulation, we have implemented this dispatch mechanism using port based concurrency. Such port based abstractions are conceptual objects which could be developed in a variety of high level languages including C++ or Java, and for a multitude of different operating systems. In this work we have used the concurrency tools provided by Microsoft’s Concurrency and Coordination Runtime (CCR) for C#, however, this is a selection of convenience. Comparable tools are presently in development for Java [20] which could just as easily be used to implement the developed framework⁶. In what follows we discuss some of the specific port based tools used, and outline the H-Dispatch class which provides the functionality to the programming model.

⁶Note also that the .NET version highlighted in this work could additionally be implemented on Linux using the open source *Mono* interface, see [21].

4.1. Tools for Port Based Programming

The programming model developed in this work relies on port and receiver abstractions. The primary innovation of Microsoft's CCR has been to implement such abstractions for use with multi-core concurrency. This provides a high degree of control and versatility in concurrent software development. Here we introduce some of the fundamental features of the CCR which are applied in the programming model⁷.

The CCR is a lightweight library written for C# which performs asynchronous operations using versatile *Port* and *Arbiter* abstractions. A *Port* can be instantiated and have messages posted to it as shown in the following code:

```
Port<type> myPort = new Port<type>();  
myPort.Post((type)message);
```

where the *Port* behaves as a *first in first out* (FIFO) data queue for the posted messages. The CCR primitives associated with message handling are referred to as *Arbiters*. By assigning an *Arbiter* to a port, it is possible to control the handling of posted messages in an 'events based' fashion. *Arbiters* remain inactive until they receive a message that satisfies some specified criterion. Once such a message is received, the *Arbiter* removes it from the port and executes any handling delegate in a separate virtual thread. The CCR manages these virtual threads in a *task queue* and maps them to actual processing threads in an optimally load balanced fashion. A simple receive *Arbiter* takes the form:

```
Arbiter.Activate(_taskQueue,  
    Arbiter.Receive(false, myPort, delegate(type message)  
    {  
        /// handler method to process the message  
    }));
```

The CCR provides several types of *Arbiters* for different circumstances

1. *Receive* arbiter: to handle messages on a single port,
2. *Choice* arbiter: chooses one of two or more available handlers to process a message, dynamically determined based on the message type,
3. *Interleave* arbiter: used to control the level of concurrency of other subset receiver groups and allows for the finalization of persistent handlers,
4. *Join* arbiter: executes a handler delegate only once a message is present on each of two separate ports, and
5. *MultipleItemReceive* arbiter: handles a predefined number of messages on a port only once all the messages have been enqueued.

These 5 *Arbiter* classes account for the majority of concurrency and coordination operations that may be required within an asynchronous application. An additional capability that is desirable for asynchronous programming is the ability to pause a thread until some resume event occurs without having to use recursive sleep commands. The CCR achieves this through a novel application of C# iterators, see Algorithm 1.

⁷For detailed description of the CCR see Chrysanthakopoulos and Singh [9], Chrysanthakopoulos [10], Richter [11] and Lu et al. [22]. The CCR library can be downloaded from <http://msdn.microsoft.com/en-us/robotics/default.aspx>.

Algorithm 1 Demonstration of CCR Iterators.

```
/// Post an empty activation value
_mainPort.Post(EmptyValue.SharedInstance)

/// Spawn an iterator
Arbiter.Activate(_taskQueue,
    Arbiter.ReceiveWithIterator(false, _mainPort, delegate
    {
        return explicitIntegration();
    }));

/// Iterator method
public IEnumerator<ITask> iteratorMethod()
{
    /// Method body
    ...

    /// Yield point
    yield return Arbiter.Receive(false, _yieldPort,
    delegate
    {
        /// the yield return statement pauses
        /// the iterator thread until a message
        /// is received on _yieldPort
    });
}
```

Here, a separate iterator thread is spawned through the `.ReceiveWithIterator` command which can then be paused indefinitely with the `yield return` statement. Only once the criteria on the yield return Arbiter is satisfied will the thread resume. Here we have used a `.Receive` Arbiter however any of the CCR Arbiters could be used. Importantly, messages to the yield return Arbiter can originate from any asynchronous thread, thus providing a versatile means of events driven coordination. This is one of the CCR's most powerful features. Of particular significance to our purpose, this feature greatly simplifies the task of synchronization.

4.2. The H-Dispatch Class

In this work we have developed a H-Dispatch class library to provide the primary functionality for the programming model discussed above. A diagram of the H-Dispatch class is provided in Fig. 6.

Similar to a CCR Port primitive, here the H-Dispatch class manages a queue of data values to be distributed to the operating threads. In the context of this work, these data values may be pointers to fine grained domains as in Fig. 2 (b). Unique to the H-Dispatch class, however, is the means by which this distribution occurs. The principal steps of the process are as follows

1. A thread requests a data value from H-Dispatch,
2. H-Dispatch receives the request and returns a value from the queue,
3. The thread processes the value,
4. When processing is complete, the thread requests another data value.

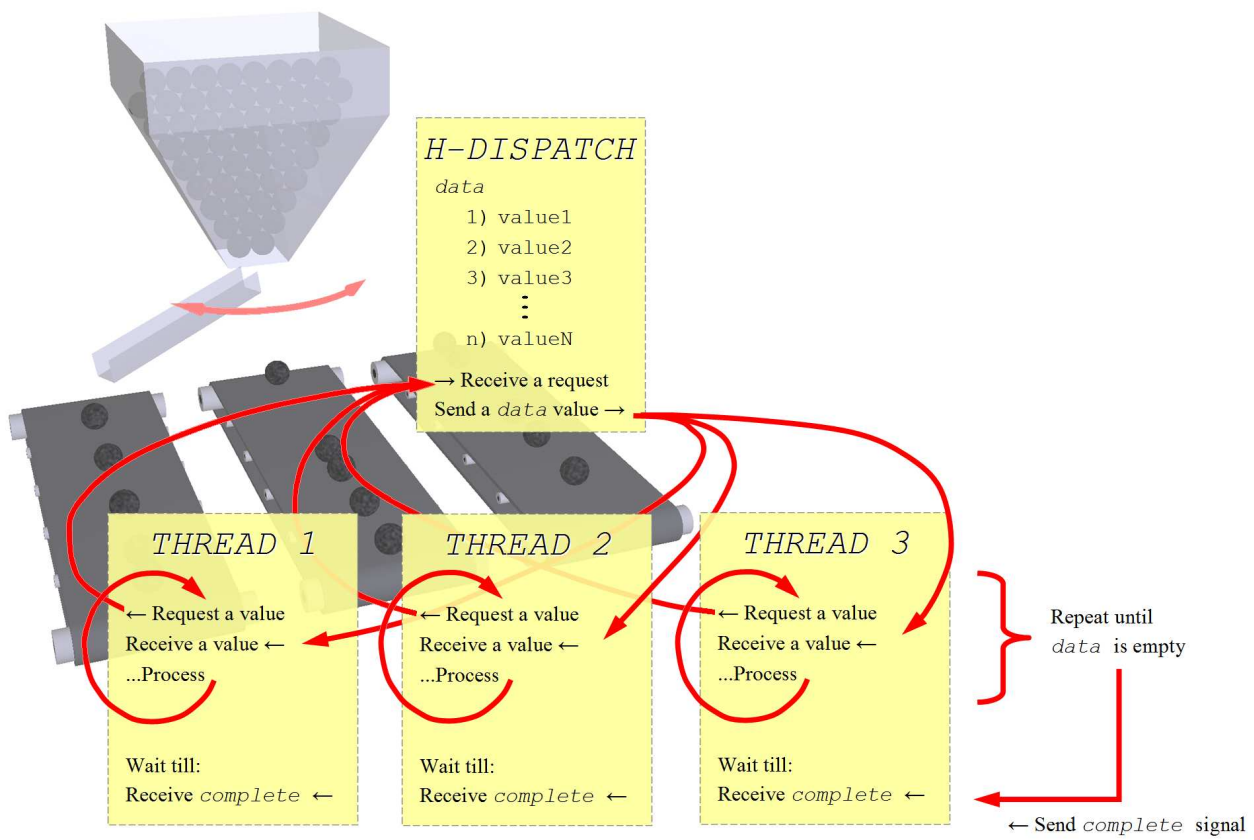


Figure 6: (color online). Implementation of the H-Dispatch class.

While this procedure is executing, the time looping in each thread is paused (i.e. Wait till... in Fig. 6). Synchronization is achieved by removing this pause once all data values have been dispatched and processed.

The H-Dispatch message passing cycle serves several key purposes. The first of these is to ensure an events based load balance. From Fig. 6, threads request data values, only once a previously received value has been processed. In this way, variable processor performance is accounted for intrinsically. Data requests will be received more frequently from faster threads and, as a result, such threads will be allocated a greater proportion of the workload.

The second purpose is to allow dynamic evolution of task number. This is achieved by using messages to identify completion rather than using specific message count, as is the case with the conventional scatter-gather model. While not shown in Fig. 6, the data which is managed by H-Dispatch must be received from one or more source. By ensuring that such sources mark the completion of transfer with a message, H-Dispatch can recognize when all data has been received⁸. H-Dispatch uses two additional criteria to determine when to issue *complete* messages to threads (see Fig. 6) to indicate thread synchronization and return control to the time loops. Firstly, H-Dispatch must have distributed all data values such that the data queue is empty. Secondly, each thread must have finished processing the last data values to ensure a consistent state at synchronization (particularly important for numerical simulation where processing involves path dependent variable evolution). This completion is identified by the presence of an empty pending data request from each thread. As each thread sends a request message to H-Dispatch following a completed processing, the presence of unfilled requests with an empty data queue ensures that final tasks have been completed. Identifying completion with messages requires no explicit knowledge of actual data value count and so means data count can evolve during execution without fault, if required.

The dispatch procedure outlined here shifts the primary data distribution mechanism from the *'Push'* of some command thread (as with scatter-gather and native CCR patterns) to the *'Pull'* of several persistently active, incumbent threads. This has various advantages, not just for numerical simulation applications. One such example is where multiple parallel hardware components, such as General Purpose Graphics Processing Units (GPGPU's), are each assigned a controlling core thread. Due to the nature of GPGPU programming it is particularly difficult to load balance multiple units with a *push* mechanism due to the anonymity of parallel threads in the conventional scatter-gather model [23]. Through the allocation of tasks to a set of identifiable incumbent threads, as with the H-Dispatch mechanism outlined above, it becomes possible for each thread to exclusively manage a single hardware component in a load balanced way.

Alternatively, where in Section 2.2 variable processor efficiency was flagged as being a challenge for load balancing, there are various circumstances (eg. multi-scale, infinite and other special numerical elements [24, 25], sparse matrix operations [26], etc) where the actual computational expense of each task can vary. This poses a similar challenge for load balancing and, in such circumstances, a pull mechanism is again the superior strategy.

The range of applications where a significant advantage is provided by such an alternate approach to multi-core programming extends well beyond that touched on here. Testing on the programmatic implementation of the H-Dispatch class is presented in the next section.

⁸Note that this allows the dispatch process to commence while data is still being acquired, thus allowing for greater concurrency.

5. Testing

In what follows, a variety of test results on the H-Dispatch programming model are presented as demonstration of its performance. These results illustrate several key areas where H-Dispatch provides improvements over conventional approaches to parallelism.

5.1. Garbage Collection Overhead

In Section 3.1 we highlighted a drawback of scatter-gather programming patterns implemented using managed languages related to the detrimental effect of garbage collection on parallel performance. The differences between the scatter-gather and developed H-Dispatch models can be quantified using a simple memory assignment study. For comparison purposes, the scatter-gather code has been implemented in C# using the CCR library (see Algorithm 2). To achieve recursive scatter and gather phases, as might be encountered during a time looping, the code of Algorithm 2 was executed within a loop, inside a CCR iterator method such as that in Algorithm 1.

Algorithm 2 Example of CCR scatter-gather.

```
/// Initialize tasks to be distributed (TypeOf(task) = T)
T[] task = T[nTasks] { d1, d2, ..., dnTasks }

/// Scatter tasks to be processed asynchronously
for (int i = 0; i < nTasks; i++)
{
    _taskPort.Post(task[i]);
}

/// Handler to receive tasks and process in parallel
Arbiter.Activate(_taskQueue,
    Arbiter.Receive<T>(true, _taskPort,
        delegate(T localTask)
        {
            /// Process
            ...

            /// Indicate thread completion
            _yieldPort.Post(EmptyValue.SharedInstance);
        }));

/// Gather point
yield return Arbiter.MultipleItemReceive(false,
    _yieldPort, nTasks, delegate
    {
        /// All threads have completed processing tasks
    });
```

To investigate the sensitivity of performance to the amount of local memory allocated to each task, a problem with 1000 tasks for distribution was created. Each task involved the assignment of values to an array of bytes, i.e.

```
/// Allocate memory
byte[] byteArray = new byte[size];
```

```

/// Assign to array
for (int i = 0; i < byteArray.Length; i++)
{
    byteArray[i] = (byte)i;
}

```

The effect of byte array size on performance was studied, simulating problems with different local memory allocations.

In the scatter-gather implementation, the anonymity of the 1000 created threads, required a new byte array to be allocated for each task. For the H-Dispatch version, a single array was instantiated for each persistent thread (i.e. 1 per core) and the memory was then overwritten by each allocated task. To develop a reasonable problem size the task distribution was repeated over 100 time steps.

Results for execution time, peak physical memory commit, and parallel efficiency⁹ are plotted against task memory sizes up to 1.0×10^6 bytes in Fig. 7 (a), (b), and (c). The tests were carried out on a 24-core Dell Server PER900 with Intel Xeon CPU, E7450 @ 2.40 GHz running 64-bit Windows Server Enterprise 2007 operating system.

From Fig. 7 (a) it can be seen that for small task memory sizes (i.e. $< 0.03 \times 10^6$ bytes) the scatter-gather model outperforms H-Dispatch as a result of the slightly greater messaging overhead of the H-Dispatch mechanism. After this point, however, H-Dispatch is seen to be the better of the two approaches. Most particularly, at a task memory size of 0.08×10^6 bytes there is a significant jump in the execution time of the scatter-gather model. This can be identified as the point at which the C# garbage collector begins to initiate memory cleanups during asynchronous segments of code. As noted earlier, this corresponds to blocking of asynchronous code portions while the managed heap is reordered and performance is seen to deteriorate rapidly as a result.

The peak physical memory usage of the two schemes in Fig 7 (b) provides further insight. The physical memory used by the H-Dispatch model increases gradually with task memory size as might be expected. Both the scatter-gather and H-Dispatch models have an idle physical memory commit of approximately 44×10^6 bytes (i.e. when task memory = 0) so on a 24 core machine, operation with a single byte array per core would be expected to result in a total operational physical memory of:

$$\text{Memory} = 44 \times 10^6 + 24 \times \text{byteArray.Length} \quad (\text{bytes}) \quad (1)$$

This is plot in Fig. 7 (b) as the *Theoretical* line and can be seen to coincide closely with the H-Dispatch results as would be expected. The peak physical memory of the scatter-gather model exceeds this figure for all by the smallest of task memory sizes. From the figure it can be seen that, after the point where the garbage collector initiates, the scatter-gather model uses closer to $44 \times 10^6 + 100 \times \text{byteArray.Length}$ bytes in physical memory. This means that, while the garbage collector is clearly operational¹⁰, it is only collecting once per every 100 completed tasks. This provides a trade off between memory usage and execution sacrifice, however, by eliminating the need for garbage collection, the H-Dispatch approach can be seen to be superior in both respects.

Fig. 7 (c) demonstrates that for H-Dispatch task memory sizes larger than 0.1×10^6 bytes, messaging overhead is overcome and local memory operations have a neutral effect on application scalability (i.e. 100% efficient). For the scatter-gather model, however, in similar ranges of

⁹Note, parallel efficiency will be described in detail, later in Section 5.4

¹⁰Physical memory would be closer to $44 \times 10^6 + 1000 \times \text{byteArray.Length}$ bytes if the garbage collector was not operating at all.

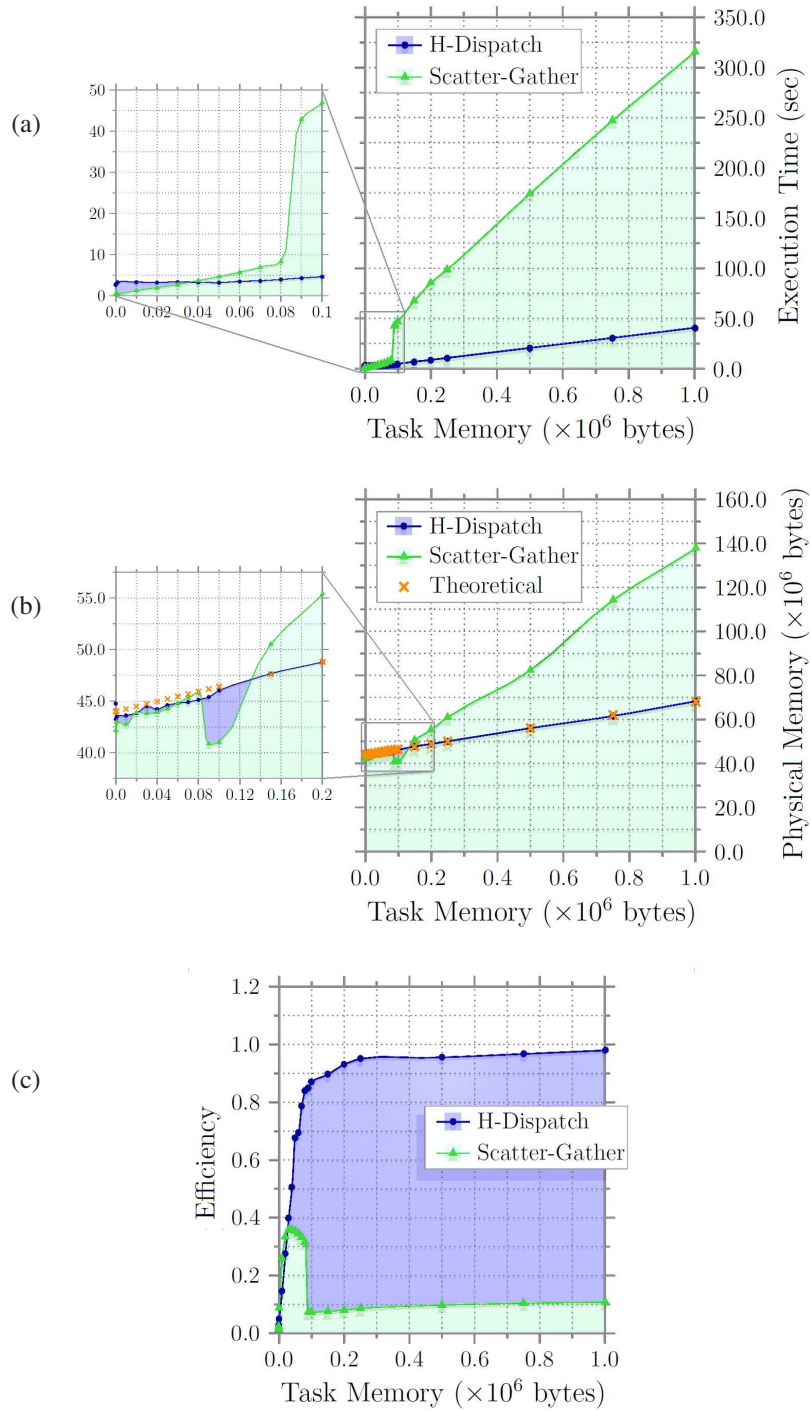


Figure 7: (color online). Results for memory allocation tests on the H-Dispatch and scatter-gather programming models. (a) execution time vs. task memory, (b) physical memory vs. task memory and, (c) parallel efficiency vs. task memory.

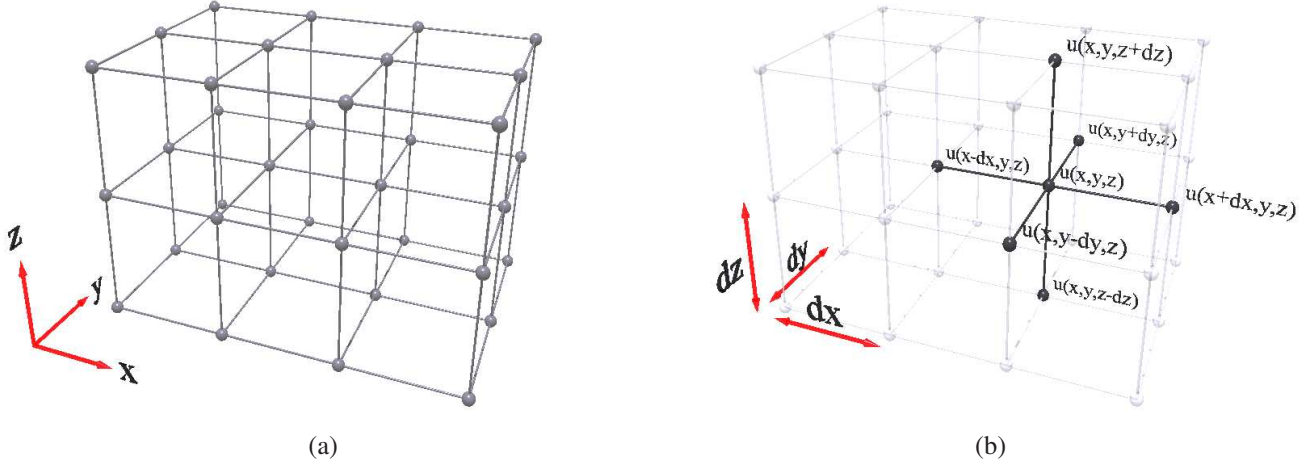


Figure 8: (color online). (a) Cubic finite difference grid, (b) the finite difference *stencil* for some grid node (x, y, z) .

local memory, garbage collection has a detrimental effect on scalability. It's important to note here that, while the poor performance of this scatter-gather model is a direct result of its implementation within a managed language, scalability will still be adversely affected in non-managed implementations by malloc and dealloc operations. In most cases such effects will be more subtle but may still warrant a H-Dispatch type approach.

5.2. Finite Difference Simulator

For the remainder of this section, we confine our attention to the implementation of a 3D Finite Difference (FD) simulator as demonstration of the H-Dispatch programming model's performance for numerical simulation applications. While a more advanced numerical method may take specific advantage of one or more of the characteristics which give the H-Dispatch algorithm its edge over existing programming models, FD provides a strong baseline for a general comparison.

The finite difference method has been used to numerically solve the 3D heat equation

$$\begin{aligned} \frac{\partial u}{\partial t} &= k \nabla^2 u \\ &= k \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \end{aligned} \quad (2)$$

within some region of space, discretized into a cubic grid, see Fig. 8 (a).

Referring to Fig. 8 (b), the simple first order finite difference scheme approximates the gradient of the heat at some grid node (x, y, z) as a function of its own heat and that at the adjacent nodes. This contributing set of nodes are referred to as being the *Stencil* for the grid point in question [15]. The FD stencil equations for the point (x, y, z) are correspondingly

$$\frac{\partial^2 u(x, y, z)}{\partial x^2} = \frac{u(x + \Delta x, y, z) - 2u(x, y, z) + u(x - \Delta x, y, z)}{\Delta x^2} \quad (3)$$

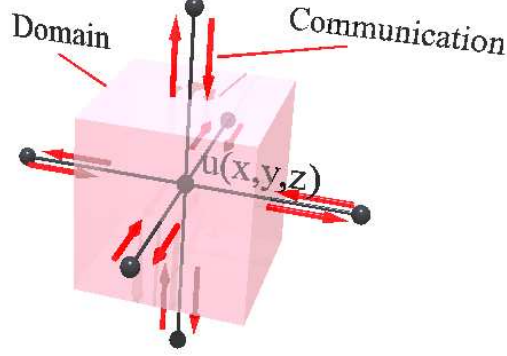


Figure 9: (color online). A domain for parallel execution containing a single grid point. Cross thread communication is represented with arrows.

$$\frac{\partial^2 u(x, y, z)}{\partial y^2} = \frac{u(x, y + \Delta y, z) - 2u(x, y, z) + u(x, y - \Delta y, z)}{\Delta y^2} \quad (4)$$

$$\frac{\partial^2 u(x, y, z)}{\partial z^2} = \frac{u(x, y, z + \Delta z) - 2u(x, y, z) + u(x, y, z - \Delta z)}{\Delta z^2} \quad (5)$$

Implementation of equations (3), (4) and (5) in (2) and solution to the differential equation using a simple explicit time integration scheme gives the heat update equation for a step in time $\Delta t = t_{n+1} - t_n$ as

$$u_{t_{n+1}} = u_{t_n} + k\Delta t \left(\frac{\partial^2 u}{\partial x^2} \Big|_{t_n} + \frac{\partial^2 u}{\partial y^2} \Big|_{t_n} + \frac{\partial^2 u}{\partial z^2} \Big|_{t_n} \right) \quad (6)$$

Correspondingly, given boundary and initial conditions, the time evolution of heat in some body can be determined.

5.3. FD in Parallel

Parallel implementation of the FD scheme of Section 5.2 is a straight forward process. The mesh is decomposed into fine grained spatial domains or *tasks* relating to one or more grid point. These tasks are then distributed across the available processors at each time step to determine the updated heat at each constituent grid point using equation (6). In the context of this paper, this distribution is carried out using the developed H-Dispatch class library. The stencil of a grid point may include points from other domains (Fig. 9) whose information must be communicated across threads during the analysis. Synchronization involves ensuring heat has been updated for all points in all domains before incrementing time for the next step. The H-Dispatch library takes care of this intrinsically.

5.4. H-Dispatch Performance Results

To test the performance of the H-Dispatch algorithm, a heat diffusion problem with 1,000,000 nodes was simulated. The measure of parallel performance is provided in terms of *Speed-Up* and

Efficiency [17]. Speed-up is the ratio of the execution time for a serial implementation of the code, to that observed for a parallel implementation across N parallel cores, i.e.

$$\text{Speed-Up} = \frac{t_{(1 \text{ processor})}}{t_{(N \text{ processors})}} \quad (7)$$

where we have determined execution time as being the time for a single step to complete, averaged over 100 steps.

For ideal application scalability, increasing the processor number by some factor should decrease the execution time by the same factor (i.e. double processors, halve execution time). Correspondingly, speed-up will ideally be equal to the number of processors. Efficiency is the ratio of actual speed-up to this ideal value

$$\text{Efficiency} = \frac{\text{Speed-Up}}{N} \quad (8)$$

with an efficiency of 1 representing the ideal case.

Simulations were carried out on the 24 core server outlined earlier in Section 5.1. Tests were run using 1, 2, 4, 8, 12, 16, 20 and 24 cores and speed-up and efficiency were calculated in each case. During testing it quickly became apparent that the relationship between task size and task count was critical to performance. Fig. 9 shows a domain whose task corresponds to a single grid point, however, for such a circumstance, the H-Dispatch mechanism must distribute 1,000,000 such tasks and message passing quickly becomes the dominant computational expense. Increasing the number of grid points within a domain acts to counterbalance the overhead associated with message passing. Fig.'s 10 (a) and (b) provide results for speed-up and efficiency for the FD simulation as a function of both the processor number and the number of grid points operated on per task.

Fig. 10 illustrates the importance of careful design of task size and number within a parallel application. For small task size, i.e. less than approximately 500 grid points per task, performance is poor due to the message passing overhead. As the number of points per task approach the total number of available points, there become fewer threads than processors and again performance drops off. Between these ranges, task size is large enough to negate the overhead associated with message passing and high performance is observed, peaking at just over 85% efficiency for 24-cores.

For the simplistic FD case presented, the range of peak parallel performance is relatively narrow. This can, however, be attributed to the fact that the necessary calculations per grid point are relatively trivial (i.e. equation (6)). For more complex numerical simulation methodologies, particularly particle methods such as the Discrete Element Method (DEM) and Smooth Particle Hydrodynamics (SPH), the increase in computation results in a far wider range of task sizes where peak parallel performance can be achieved. In fact, research we have carried out implementing SPH using the H-Dispatch mechanism has provided efficiency results approaching 70% for 24-cores with cells containing as few as 27 particles¹¹.

5.5. Cache Implications

It is pertinent here, to briefly address cache implications for program development. As a means to address the widening disparity between processor capabilities and main memory band-

¹¹Complete results to be published elsewhere.

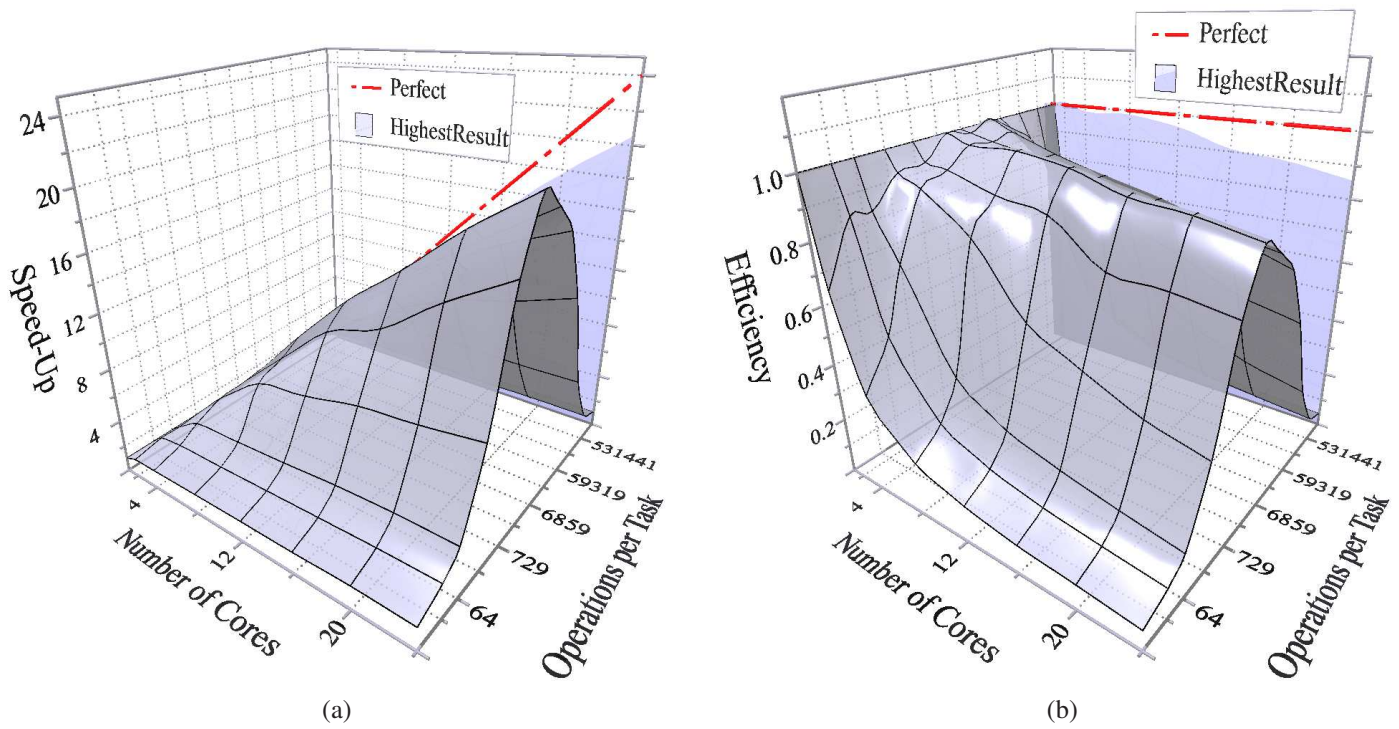


Figure 10: (color online). (a) Speed-up and (b) efficiency for the 1,000,000 node FD simulation using the H-Dispatch class library, plotted against both processor number and grid point count per task. Peak performance for each is projected onto the rear plot wall and compared with the ideal.

width, today’s multi-core architectures adopt a *hierarchical memory* arrangement which can involve as many as 3 levels of on-chip cache memory [27, 28]. How effectively this cache is utilized, is strongly influenced by the structure of a programming model.

A well recognized technique for cache optimization is referred to as *cache blocking*, or *spatial locality* (see Pohl et al [27], Lam et al [29] and Tian and Shih [18]). The aim of this technique is to complete small blocks of a larger task sequentially to maximize cache reuse. An advantage of the fine grained domain distribution technique developed in this work is that it naturally provides spatial locality and so is inherently cache optimized. If desired, the size of a domain can be controlled so as to fit completely into L2 or L3 cache. Additionally, we observed the spatial arrangement of points within a domain or ‘block’ to be important. For the FD study above, an improvement in efficiency of over 10% for 24-cores was achieved by simply using a cubic, rather than row-wise, domain configuration. This can be attributed to the increased number of interacting nodes present in each block, thus reducing cache miss rates. A minor level of *temporal localization* (see Lam et al [29]) was also provided by carrying out gradient and update calculations for each domain in a single task block, corresponding to a further improvement in performance. These points will be addressed in more detail in a later publication.

5.6. Comparison to Conventional Approaches to Parallel Simulation

For comparison purposes, the heat diffusion problem of Section 5.2 has been implemented using several conventional approaches to parallel simulation. A scatter-gather version (following the form of Fig. 1 (b)) utilizing the C#/CCR programming model of Algorithm 2 has been implemented, as has a message passing model (following Fig. 1 (a)), also written in C# using the recently developed .NET implementation of the MPI standard, MPI.NET [5, 30]. Each version of the heat diffusion simulator made optimal use of both spatial and temporal locality for cache optimization (see Section 5.5), and the sub-domains for the MPI version were assumed static and so were calculated only once at the beginning of each run.

Results for speed-up and efficiency for the H-Dispatch, scatter-gather and MPI programming models are compared to the theoretical ideal in Fig. 11. An investigation into the effect of task size on the performance of the scatter-gather model produced speed-up and efficiency surfaces similar in shape to that of the H-Dispatch model in Fig. 10. Peak values are reported in Fig. 11. For the MPI case, task size is dictated by the required number of sub-domains and so no such study was conducted.

Peak values of speed-up and efficiency for both the H-Dispatch and scatter-gather FD simulators occurred when 2197 nodes (i.e. $13 \times 13 \times 13$) were operated on in each task domain. At this number of nodes, the local memory associated with an FD task was determined to be approximately 0.3×10^6 bytes. Recalling the findings of Section 5.1 and Fig. 7, this level of local memory corresponds to a measurable level of garbage collection overhead in the case of the scatter-gather model. As a result of this, the scatter-gather model is seen to under-perform the H-Dispatch model in the FD performance testing¹².

Fig. 11 additionally shows the H-Dispatch and scatter-gather FD implementations to outperform the MPI instance. The performance results of the MPI simulator compare well with those reported by authors such as Lu and Shen [31], De Angeli et al [32] and Ramadan [33] for various implementations of cross-machine FD simulators. Cross-process communication in the MPI

¹²Note that the local task memory for a particle based method such as DEM or SPH can approach levels closer to 10×10^6 bytes, at which point the garbage collection overhead becomes significantly more detrimental.

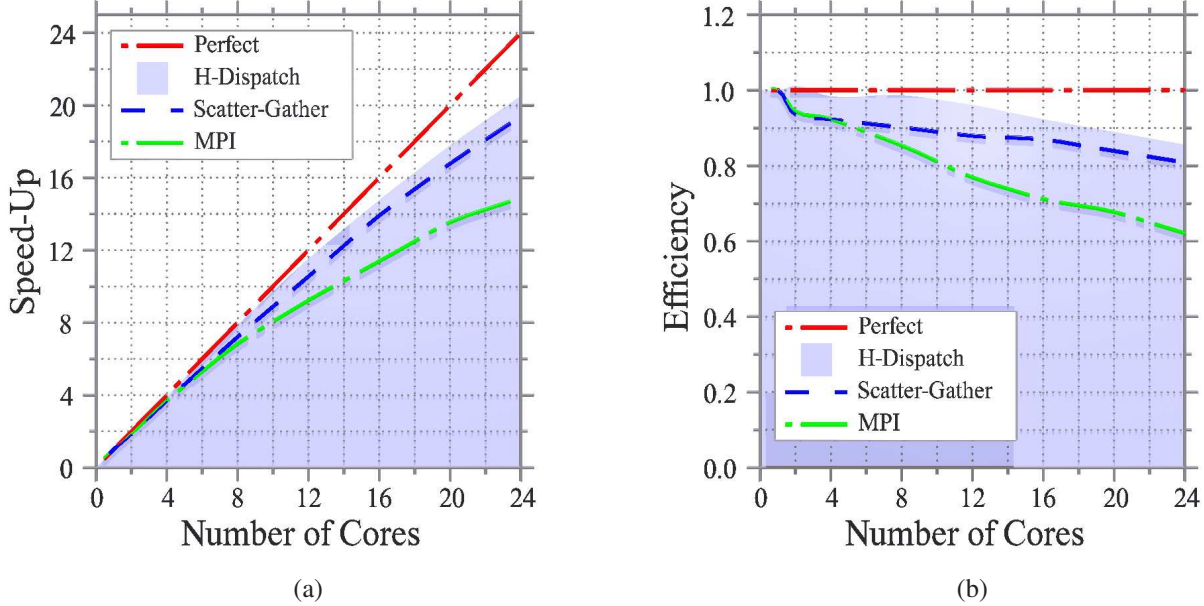


Figure 11: (color online). (a) Peak speed-up and (b) peak efficiency for the 1,000,000 node FD simulation. Results for the H-Dispatch parallel model, conventional scatter-gather model and MPI message passing model are given and compared to the ideal results.

case was a significant contributor to its lesser performance. Additionally, due to the maintenance of ‘ghost regions’ and other factors, the MPI implementation required approximately 10× the physical operating memory to that of the other two methods. This was observed to contribute additional sacrifices in performance related to main memory bandwidth limitations and cache efficiency.

In order to achieve optimum performance from the MPI FD implementation, the sub-division of space was considered constant and all data structures were reduced to arrays to minimize communication overhead. While numerically efficient, such a coding style lacks the development ease and clarity of *object oriented* programming, an important factor which can be overlooked when assessing a program based solely on numerical performance. This was not an issue for the other two methodologies.

It’s additionally pertinent to note that upon testing algorithm performance following the external monopolization one or more processors, the H-Dispatch and scatter-gather models adjusted accordingly while execution of the MPI model was blocked indefinitely. This demonstrates the importance of flexible load balancing on multi-core architectures.

The observed excellent performance of the H-Dispatch algorithm for the simple FD simulator demonstrates that it is not simply a programming model for specialized numerical methods. Rather, it is a generally applicable programming model which provides a high level of parallel performance on multi-core architectures and has the capability of handling various specialized numerical complexities which traditional methodologies find challenging. We leave demonstration of H-Dispatch’s more specialized capabilities to later publication.

6. Closure

A new programming model, suitable for parallel implementation of numerical simulation software on multi-core hardware architectures has been presented. It has been shown that the H-Dispatch algorithm provides a flexible and efficient mechanism for distributing tasks for multi-core. The H-Dispatch algorithm controls parallel synchronization and allows for an events based load balance and the dynamic evolution of parallel task number. These capabilities are a significant improvement over those of conventional scatter-gather and message passing type programming models. The programming model has been tested on a finite difference (FD) heat diffusion simulation with 1,000,000 grid points and has demonstrated excellent performance with efficiency figures as high as 85% for a 24-core machine.

We have presented examples which show the H-Dispatch programming model to be appropriate for a wider range of applications than just numerical simulation codes. Additionally, some of the findings of this work have particular significance for all parallel applications written for multi-core hardware architectures. The most critical of these is the observed importance of optimizing the tradeoff between task number and task size within a parallel application. One of the key advantages of multi-core over cross machine computing is the speed with which cross thread communication occurs. As fast as this communication may be, it must still be offset by numerical tasks large enough to negate the message latency. Fig. 10 illustrates this point and includes a dimension of analysis commonly overlooked in the literature. Multi-core allows for application speed-up but requires parallel tasks to be optimized.

Acknowledgments. The authors would like to acknowledge the Schlumberger Doll Research Center and Saudi Aramco for their joint financial support of this research. Also George Chrysanthakopoulos and Henrik Nielsen of Microsoft Research for their assistance with Robotics Studio and the CCR.

References

- [1] Anthes, G., Computerworld UK (2007), [Online] Available: <http://www.computerworlduk.com/technology/hardware/processors/in-depth/index.cfm?articleid=957>.
- [2] Gropp, W., Lusk, E., and Skjellum, A., *Using MPI: Portable Parallel Programming With the Message-Passing Interface*, MIT Press, Cambridge, 1999.
- [3] Eadline, D., Linux Magazine (2007), [Online] Available: <http://www.linux-mag.com/id/4608>.
- [4] Fox, G., Distributed and parallel programming environments and their performance, in *Proceedings of the 2008 IEEE eScience Conference: Accelerating Time to Scientific Discovery*, 2008.
- [5] Qiu, X. et al., Parallel data mining on multicore clusters, in *7th International Conference on Grid and Cooperative Computing GCC2008*, 2008.
- [6] Chandra, R. et al., *Parallel Programming in OpenMP*, Morgan Kaufmann Publishers, San Francisco, 2001.
- [7] Leiserson, C. E. and Mirman, I. B., *How to Survive the Multicore Software Revolution (or at Least Survive the Hype)*, Cilk Arts, Cambridge, 2008.
- [8] Amdahl, G. M., The validity of the single processor approach to achieving large-scale computing capabilities, in *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 30, pages 483–485, Reston, Va, 1967, AFIPS Press.
- [9] Chrysanthakopoulos, G. and Singh, S., An asynchronous messaging library for C#, in *Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages*, pages 89–97, San Diego, 2005, OOPSLA 2005.
- [10] Chrysanthakopoulos, G., Channel9 Wiki Microsoft (2007), [Online] Available: <http://channel9.msdn.com/wiki/concurrencyruntime/>.
- [11] Richter, J., MSDN Magazine (2006), [Online] Available: <http://msdn.microsoft.com/en-us/magazine/cc163556.aspx>.

- [12] Qiu, X. et al., High performance multi-paradigm messaging runtime integrating grids and multicore systems, in *Third IEEE International Conference on e-Science and Grid Computing*, edited by Fox, G., Chiu, K., and Buyya, R., pages 407–414, Los Alamitos, 2007, IEEE Computer Society.
- [13] Stewart, D. B., Volpe, R. A., and Khosla, P. K., *IEEE Transactions on Software Engineering* **23** (1997) 759.
- [14] Liu, M. B. and Liu, G. R., *Computational Mechanics* **35** (2005) 332.
- [15] Foster, I., *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley, New York, 1995.
- [16] Pohl, T. et al., Performance evaluation of parallel large-scale Lattice Boltzmann applications on three supercomputing architectures, in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, pages 21–33, Washington, DC, USA, 2004, IEEE Computer Society.
- [17] Wu, J. S. and Tseng, K. C., *International Journal for Numerical Methods in Engineering* **63** (2005) 37.
- [18] Tian, T. and Shih, C.-P., Intel Software Network (2007), [Online] Available: <http://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems/>.
- [19] Bank, R. E. and Holst, M., *SIAM Journal on Scientific Computing* **22** (2000) 1411.
- [20] Jula, A., 2009, Personal Communication.
- [21] Dumbill, E. and Bornstein, N., *Mono: A developer's Notebook*, O'Reilly Media, Inc., Sebastopol, CA, 2004.
- [22] Lu, W., Gunarathne, T., and Gannon, D., Developing a concurrent service orchestration engine in ccr, in *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 61–68, New York, 2008, ACM.
- [23] Stuart, J. A. and Owens, J. D., Message passing on data-parallel architectures, in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, 2009.
- [24] Irving, G., Guendelman, E., Losasso, F., and Fedkiw, R., *ACM Trans. Graph.* **25** (2006) 805.
- [25] Zienkiewicz, O. C., Bando, K., Bettess, P., Emson, C., and Chiam, T. C., *International Journal for Numerical Methods in Engineering* **21** (1985) 1229.
- [26] Tuminaro, R. S., Shadid, J. N., and Hutchinson, S. A., *Concurrency: Practice and Experience* **10** (1998) 229.
- [27] Pohl, T., Kowarschik, M., Wilke, J., Iglberger, K., and Rde, U., *Parallel Processing Letters* **13** (2003) 549.
- [28] Kuppuswamy, R. et al., Over one million TPCC with a 45nm 6-core Xeon CPU, in *Solid-State Circuits Conference - Digest of Technical Papers, 2009. ISSCC 2009. IEEE International*, pages 70–71, 71a, 2009.
- [29] Lam, M. D., Rothberg, E. E., and Wolf, M. E., *ACM SIGOPS Operating Systems Review* **25** (1991) 63.
- [30] MPI.NET: High-performance C# library for message passing, 2008, [Online] Available: <http://www.osl.iu.edu/research/mpi.net/>.
- [31] Lu, Y. and Shen, C. Y., *IEEE Transactions on Antennas and Propagation* **45** (1997) 556.
- [32] Angeli, J. P. D., Valli, A. M. P., Jr., N. C. R., and Souza, A. F. D., Finite difference simulations of the Navier-Stokes equations using parallel distributed computing, in *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'03)*, pages 149–156, Los Alamitos, CA, USA, 2003, IEEE Computer Society.
- [33] Ramadan, O., *Parallel Computing* **33** (2007) 109.