WorkStream – A Design Pattern for Multicore-Enabled Finite Element Computations

BRUNO TURCKSIN, Texas A&M University
MARTIN KRONBICHLER, Technische Universität München
WOLFGANG BANGERTH, Texas A&M University

Many operations that need to be performed in modern finite element codes can be described as an operation that needs to be done independently on every cell, followed by a reduction of these *local* results into a *global* data structure. For example, matrix assembly, estimating discretization errors, or converting nodal values into data structures that can be output in visualization file formats all fall into this class of operations. Using this realization, we identify a software design pattern that we call *WorkStream* and that can be used to model such operations and enables the use of multicore shared memory parallel processing. We also describe in detail how this design pattern can be efficiently implemented, and we provide numerical scalability results from its use in the DEAL.II software library.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: Finite Element Software—*Multicore computing*; G.1.8 [**Numerical Analysis**]: Partial Differential Equations—*Finite element method*

General Terms: Parallel Algorithms, Software Design, Multicore Computing

Additional Key Words and Phrases: Finite element algorithms, assembly, pipeline software pattern

ACM Reference Format:
Bruno Turcksin, Martin Kronbichler, and Wolfgang Bangerth. 2016. *WorkStream* – A design pattern for multicore-enabled finite element computations. ACM Trans. Math. Softw. 43, 1, Article 2 (August 2016), 29 pages.
DOI: http://dx.doi.org/10.1145/2851488

1. INTRODUCTION

Since around 2004, computer processors clockspeeds have not become significantly faster per processing unit, and single-core performance has increased at a much lower speed than before. Instead, processor manufacturers have decided to provide more compute power by increasing the number of cores working in parallel on a single chip. For example, at the time of writing, well-equipped workstations may have 64 cores, and the recently introduced Intel Xeon Phi accelerator card has a similar number; no

B. Turcksin and W. Bangerth were partially supported by the National Science Foundation under award OCI-1148116 as part of the Software Infrastructure for Sustained Innovation (SI2) program; by the Computational Infrastructure in Geodynamics initiative (CIG), through the National Science Foundation under Award No. EAR-0949446 and The University of California – Davis; and through Award No. KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST).
Authors' addresses: B. Turcksin and W. Bangerth, Department of Mathematics, Texas A&M University, College Station, TX 77843; email: {bangerth,turcksin}@math.tamu.edu; M. Kronbichler, Institute for Computational Mechanics, Technische Universität München, Boltzmannstr. 15, 85748 Garching b. München, Germany; email: kronbichler@lnm.mw.tum.de.

DOI: http://dx.doi.org/10.1145/2851488

doubt, the number of cores will continue to grow in the future. To exploit this resource efficiently, modern scientific codes therefore necessarily need to be written in a way that exposes to compilers and support libraries operations that can be executed in parallel.

In the past, attempts to do so have often focused on relatively low-level descriptions of simple loops. In the context of finite element software—the focus of this article—these attempts have often targeted the inner loops of linear algebra operations. For example, through paradigms such as the compiler directives in OpenMP, one can annotate the loop that sums two vectors into a third to indicate that the operations in the loop body are independent and can consequently be executed in parallel. A great deal of literature is available on the potential for acceleration of finite element codes using this approach; see, for example, Mahinthakumar and Saied [2002], Oliker et al. [2002], Nakajima [2003, 2005], Pantalé [2005], and Giannoutakis and Gravvanis [2009].

On the other hand, parallel programming is most scalable if it can identify opportunities for parallelism at the highest levels of a program. For finite element codes, this is frequently at the level of loops over all cells. These loops form the most common operation in finite element codes apart from the solution of linear systems and appear, for example, in the assembly of the stiffness matrix and right-hand sides, the estimation of errors, the computation of postprocessed quantities, and the generation of data for visualization. Rather than addressing each of these places in an ad-hoc way, we have identified a software design pattern that allows us to describe each of these loops over all cells in a common way and to utilize a common software framework for their implementation. The key insight we use in modeling these operations is that these cases are all composed of *an embarrassingly parallel operation followed by a reduction operation*. The reduction operation has to be synchronized and sequenced properly to guarantee correctness and reproducibility. We will describe these requirements and efficient implementations herein. (Obviously, finite element codes also contain many operations that do not follow this pattern and are therefore not addressed by the methods discussed herein. However, as we will show, the operations we target account for about half of the runtime of a typical finite element code and parallelizing them is therefore a necessary component of achieving reasonable overall speedup.)

With this, the goals of this article are as follows:

—To describe a design pattern frequently encountered in finite element codes;
—To discuss ways in which this design pattern can be implemented in a way that allows us to efficiently exploit multicore parallelism on modern computer systems;
—To demonstrate the efficiency of these implementations on real-world testcases taken from our work on efficient numerical solvers for complex problems.

In these goals, this article stands squarely in the tradition of research on software design patterns [Gamma et al. 1994; Mattson et al. 2004]. We would like to emphasize that our goal is not to outline ways to achieve perfect scalability on large parallel machines. The techniques for this are well known and in almost all cases require the distribution of data structures to limit memory access contention to explicit communication sections. Doing so, however, typically requires redesigning existing codes from scratch. On the other hand, our paper presents a way that allows reinterpreting common operations in finite element codes in view of a *design pattern*; if these operations follow the description of this pattern—which typically only requires splitting functions into two parts but no redesign of data structures or the introduction of explicit parallel communication—then they are trivially parallelizable. We demonstrate that with this marginal programming effort, we can achieve sizable speedups between 7 and 44 for significant parts of a finite element code on current workstations. Reformulating

operations in view of the design pattern therefore provides an easy way toward parallelization of legacy codes as well as for the many codes that can benefit from utilizing multiple cores but do not require fully parallel implementations based on MPI that would scale to hundreds of cores.

The remainder of the article is structured in the following way: In Section 2, we will provide an overview of how modern finite element codes are typically structured and what kinds of operations they perform that may lend themselves to parallelization. Section 3 will then be a critique of the existing approaches that are widely used in exploiting parallelism to map finite element codes to shared memory multicore machines. Section 4 introduces a design pattern that will allow us to implement high-level loops in finite element codes using a common software framework. We will discuss three possible implementations of this design pattern in Section 5 and compare their relative efficiency and scalability on large multicore machines in Section 6. We will conclude in Section 7.

2. CHARACTERISTICS OF MODERN FINITE ELEMENT CODES

Historically, the runtime of finite element codes was dominated by the cost of linear solvers. This was a consequence of the use of fixed meshes, low-order elements, and mappings, all of which resulted in fast assembly of linear systems; and relatively simple solvers such as Jacobi or Gauss-Seidel preconditioned Conjugate Gradients, or even just fixed point iterations that resulted in slow linear solves. Graphical data were written into files that often were barely more than memory dumps of solution vectors and mesh data structures. Given this set of techniques, linear solvers often consumed 80% or more of the overall runtime, and their optimization and parallelization presented the most expedient approach to accelerating the overall execution.

However, this workload is no longer representative of the current generation of finite element codes. For example, today's codes use sophisticated, adaptively changing meshes; higher order, mixed, or enriched elements; and higher order mappings to support curved boundaries. All of these make the assembly of linear systems far more complex and expensive. Furthermore, modern codes often write their numerical results in semantically rich data formats (such as the XML-based VTU format [Schroeder et al. 2006] or data stored in HDF5-encoded formats [Folk et al. 1999]) that are relatively expensive to generate but allow for the production of high-quality graphics and other postprocessing steps. Finally, linear solvers have become vastly better: The development of block ("physics-based") preconditioners for coupled systems and algebraic or geometric multigrid for elliptic problems have allowed the solution of problems up to 10^9 unknowns in a number of iterations that is almost independent of the size of the problem, even for complex equations [White and Borja 2011; Kronbichler et al. 2012; Frohne et al. 2015].

A consequence of these developments is that, today, modern finite element codes are no longer dominated by the CPU time spent in straightforward linear solvers (such as those using sparse or dense decompositions or conjugate gradients with relatively simple preconditioners). To give just one example, a recent three-dimensional simulation of time-dependent thermal convection using the ASPECT solver [Kronbichler et al. 2012] (See also http://aspect.dealii.org/) resulted in the breakdown of its overall runtime (summed over more than 10,000 time steps) shown in Table I. This computation used an average of slightly more than 10^7 degrees of freedom. Mesh adaptation and postprocessing are relatively cheap because meshes were adapted only every 15 time steps, and the major part of postprocessing—generating graphical output—happened only every 50 time steps. The computation ran on 64 cores using MPI. It is therefore representative of many medium-sized computations common today.

Table I. Runtime of a Typical ASPECT Run

Stokes assembly	24%	Temperature assembly	19%
Compute Stokes preconditioner	10%	Compute temperature preconditioner	3%
Stokes solve	29%	Temperature solve	9%
Mesh adaptation	2%	Postprocessing	1%

What these numbers show is that, for many complex applications, simple linear solves are no longer the dominant part of the code. Rather, modern codes have many sections that significantly contribute to the overall runtime. If one wanted to make such codes fully utilize multi- or many-core machines, it is therefore necessary to address a much larger subset of functions than just the linear algebra. This is made more difficult by the fact that assembly, the computation of algebraic multigrid hierarchies, or postprocessing are typically not expressed in terms of operations on large vectors but instead require the traversal of complex data structures with many indirections. Parallelizing the innermost—often very short—loops is therefore not going to yield a significant benefit on current architectures, although they may of course lend themselves to vectorization. These arguments regarding utilization of parallel computers hold true even taking into account that the assembly operations used in the preceding example could be accelerated with techniques such as those discussed in Melenk et al. [2001] and Kronbichler and Kormann [2012].

One approach that has commonly been taken to address these difficulties is to parallelize many of these operations using a distributed memory paradigm, using, for example, MPI [Message Passing Interface Forum 2012]. While this has been very successful, it is labor intensive in that it requires a complete redesign of many parts of an existing code; developing codes with MPI also requires specialized knowledge not available in many communities for whom large-scale parallel computations are not necessary. In such cases, providing simple means that can efficiently use shared memory programming parallel processing on a single machine provides an attractive route toward utilizing today's computer architectures. This article is one step in this direction.

3. A CRITIQUE OF CURRENT APPROACHES TO MULTICORE PARALLELISM FOR FINITE ELEMENT CODES

There is an enormous body of literature on the optimization and parallelization of finite element codes dating back to at least the 1970s; see, e.g., George [1971, 1973], Chang et al. [1984], Law [1986], Farhat [1988], Yagawa and Shioya [1993], Tezduyar et al. [1993], Devine and Flaherty [1996], McKenna [1997], Laszloffy et al. [2000], Löhner and Galle [2002], Remacle et al. [2002], Banaś [2004], Bauer and Patra [2004], Bergen et al. [2005], Nakajima [2005], Pantalé [2005], Paszyński et al. [2006], Paszyński and Demkowicz [2006], Williams et al. [2007], Paszyński et al. [2010], Logg et al. [2012], Löhner and Baum [2013], and Russell and Kelly [2013]. However, most of these contributions—and certainly most of the more recent ones—deal with parallelizing linear algebra operations. This approach has given rise to paradigms such as OpenMP [2014] or OpenACC [2014] that annotate simple loops to allow the compiler to infer mutual independence of loop bodies and thereby allow scheduling them to run on different processor cores. A related research direction has been the use of GPUs to offload computations; this has also typically involved only linear algebra operations [Göddeke et al. 2008; Wadbro and Berggren 2009; Joldes et al. 2010; Williams et al. 2010; Pichel et al. 2012; Markall et al. 2013], explicit time stepping schemes [Klöckner et al. 2009; Komatitsch et al. 2010; Corrigan et al. 2011], or finite element assembly considered in

isolation [Cecka et al. 2011; Knepley and Terrel 2013]. A discussion of the potential for acceleration using GPUs can be found in Vuduc et al. [2010].

We believe that this approach is insufficient to address the more complex structure of a big fraction of current finite element codes. Our critique has essentially three reasons:

—This approach is not a good match for modern finite element codes: Since today's finite element codes no longer operate on simple data structures using tight loops, no single function is typically responsible for more than 20% of run time (see, e.g., the evaluation of DEAL.II's performance as part of the SPEC CPU 2006 suite [Henning 2007]). Furthermore, even "hot" linear algebra functions often contain complex loops such as finding coarsening operations in algebraic multigrid preconditioners. Annotating individual code blocks cannot adequately address these complex data dependencies.
—This approach is too much work: Modern finite element codes can consist of hundreds of thousands of lines of code. It is simply not feasible to identify the potential for parallelism at the level of individual loops. Consequently, on large core counts, the loops that have not been parallelized will become the bottlenecks, providing for an almost endless chase for the next place that needs to be addressed.
—This approach is too low level: If a modestly sized finite element computation today has a million unknowns, then parallelizing operations such as vector-vector additions on machines with 64 cores leaves each core only around half a million instructions to execute. This may be enough to amortize the cost of breaking the vector into chunks, scheduling them to run on 64 threads located on the 64 cores, and then joining execution again in the main thread, but it will not scale to future larger core counts. Even more importantly, most vector-vector operations happen on vectors of much smaller sizes (e.g., the number of degrees of freedom per cell, at most a few tens) for which parallelization by this approach cannot provide an answer. Ultimately, every thread synchronization from annotating a loop to run in parallel (e.g., by marking it as `#pragma omp parallel` with OpenMP) typically requires several thousand CPU cycles and has to be amortized; this is far more difficult to do when simple inner loops are annotated.

These points can only be addressed by exposing parallelism not at the level of the most deeply nested loops, but at the outermost ones. Because of complex data dependencies, relatively simple annotation systems used by language extensions such as OpenMP are unable to adequately describe such loops. Rather, data dependencies and synchronization have to be modeled explicitly in the code, and parallelization happens by exposing independent work in large chunks that are then assigned to available compute resources by a scheduler. Parallelizing large chunks of work also fits today's complex memory hierarchies better and leads to better cache usage.

Remark 1. In this work, we focus on the tasks specific to finite element toolkits and consider linear solvers to be black boxes that are parallelized on their own (a common strategy in many FEM applications). Thus, we will not discuss issues such as the creation of algebraic multigrid levels or combined stages of assembly and matrix factorizations by frontal solvers [Laszloffy et al. 2000; Paszyński et al. 2006; Paszyński and Demkowicz 2006; Paszyński et al. 2010]. On the other hand, matrix-free iterative solvers that perform element integrals [Kronbichler and Kormann 2012] fit into our scheme and could profit from design pattern presented here. Such operations are of the same category as the various "assembly" operations in Section 6.

4. *WORKSTREAM*: A DESIGN PATTERN FOR PARALLEL LOOPS OVER ALL CELLS

4.1. Motivation

Looking at Table I of runtimes, one recognizes that several of the major components are loops over all cells of a finite element mesh:

—When forming a linear system, one typically loops over all cells; evaluates shape functions, their derivatives, and other quantities at the quadrature points of this cell; and forms a local contribution. This *local contribution* is then *added* to the global linear system, with several cells contributing to the same matrix entries.
—In postprocessing, one generates data for files that can be used in creating graphical representations of the solution. For nontrivial cases, this usually involves evaluating the solution on each cell, transforming it to a suitable local representation, and then conjoining the local representations into a global object that can be handed off to functions that write them to disk. The local representation may simply consist of the values of the solution at the vertices of the current cell, but it can also contain derived quantities such as Mach numbers computed from the velocity or seismic wave speeds computed from density, temperature, and chemical composition in simulations of convection in the Earth's mantle.
—Also in postprocessing, one frequently wants to evaluate integral quantities. Examples are the lift and drag coefficients of airfoils in aerodynamics computations, the average heat flux from the Earth's core to the surface, or similar quantities. As before, these are typically computed by a loop over all or a subset of cells where local quantities are computed, and these local quantities are then added up to a global number.

The point of this enumeration is to find common patterns that can then be exploited by design patterns that allow implementations to be generic across specific situations. This approach has led to the identification of a significant number of *software design patterns* [Gamma et al. 1994] and the consequent standardization of many components of software. In the current context, the central realization is that every one of the situations just discussed consists of a *stream of objects* on which we need to *compute a local contribution* followed by a *reduction operation* in which the local contribution is added or merged into a global object. We name this pattern `WorkStream`. It is related to *MapReduce* [Dean and Ghemawat 2008; Lin and Dyer 2010] but works on more structured inputs and makes use of significantly more stringent requirements as outlined in the next section. It is also not intended for distributed memory use.

It is important to recognize that, of course, not all components of a finite element code fit the mold of independent computations followed by reductions. For example, iterative solvers, most preconditioners, and direct solvers do not follow this description. Consequently, while they may offer opportunities for parallelism in other ways, the design pattern described here will not be of use for these parts of codes. On the other hand, as outlined earlier, the design pattern we describe herein fits parts of finite element codes that, taken together, account for a significant fraction (in some cases the majority) of CPU time. The work described here is therefore relevant even though it does not fit *all* parts of finite element codes.

Next, we will systematize the preceding observations. Section 5 then presents a generic framework for the implementation of the corresponding software design pattern.

4.2. Requirements

A sequential implementation of any of the operations listed earlier might look like this in pseudo-code, keeping in mind that the loop over all cells might as well be a loop over the elements of any other kind of sequence as well:

```
GlobalObject       global_object;
LocalContribution  local_contribution;
ScratchData        scratch_object;

for (cell ∈ [c_0, c_N))
{
  local_contribution = compute_local_contribution(cell,
                                                  scratch_object);
  global_object ⊕= local_contribution;
}
```

Here, `LocalContribution` is a data type that stores whatever we compute locally. In the preceding examples, it could be small dense vectors and matrices (for assembly), data for visualization from the current cell (visualization), or a single number (computing integral averages). `GlobalObject` is a type for the object we would like to compute from the contributions of all cells, for example a sparse matrix and vector, or an array of structures containing information for visualization from all cells. $\oplus$ is the operation that adds or joins the local contributions to the global object; the addition to an existing object, $\oplus =$, is generally a reduction operation. In the context of finite element computations, `GlobalObject` is in general a large object that cannot be duplicated or broken down into smaller subentities easily and cheaply to support pair-wise reduction, as one might do for, example, for simple sums over vector elements [Reinders 2007]. In other words, the concept underlying most implementations of MapReduce on multicore machines (like, e.g., Phoenix [Chu et al. 2006]) cannot easily be applied to our situation.

An important practical consideration is that the computation of local contributions typically requires scratch objects holding intermediate computations. For example, in DEAL.II, computing local contributions during the assembly process is done using a class called `FEValues` [Bangerth et al. 2007] that stores the values and derivatives of shape functions and similar information. It is initialized at the beginning of the loop by precomputing these values on the reference cell, and every cell then only needs to transform these values using the mapping from the reference cell to this particular cell. In other operations, one often requires scratch arrays that hold temporary data that can then be further transformed into local contributions. We will assume that all of this temporary data has been collected in a class called `ScratchData`, and the computation of local contributions will thus use an object of this type. It is a common strategy to move as much expensive initialization—computing values on the reference cell, or allocating memory—into the construction of such objects, to make the operations on every cell as cheap as possible.[1] We will assume that the computation of local contributions will not make use of the state of the scratch object carried over from the previous loop iteration.

Having introduced this general approach to computing the global object, let us add a few comments that will guide our considerations when translating the preceding loop into a loop that can be executed in parallel on multicore machines:

—In all of the examples we have considered, the computation of local contributions on different cells are mutually independent and can be run in parallel as long as every independent execution of the `compute_local_contribution` function has its own copy of a scratch object. In other words, this operation is typically trivially parallelizable.

[1]Numerical experiments with the operations discussed in Section 6 show that precomputing data makes all operations 3–10 times faster compared to an approach in which no data are precomputed before the loop starts (i.e., where one would create the equivalent of the `scratch_object` locally inside the `compute_local_contribution()` function).

—On the other hand, the reduction operation using the $\oplus =$ operation reads from objects that can be computed independently but writes into the same global object. To avoid data corruption, the traditional approach is therefore to guard the parallel execution of this line by a mutex. With this, a simple parallelization strategy would be to create P threads, each having its own `ScratchData` object and working on subsets of the range $[c_0, c_N)$ of cells of roughly equal size.
—However, an unfortunate reality is that, in many cases, the reduction operation $\oplus =$ is not associative. The most obvious case is that IEEE floating point addition is not associative and $(a + b) + c$ may differ from $(a + c) + b$ by round-off. The consequence is that the scheme outlined in the previous bullet point produces results that are not reproducible, with the usual catastrophic consequences on programmers' ability to find bugs or simply repeat computations.
—While not always the case, we can generally assume that the reduction operation is significantly cheaper than the computation of local contributions. (We will evaluate this assumption in our experimental results in Section 6.)

Our requirements for a design pattern that models the parallel, shared memory execution of loops such as those described are therefore:

(1) Parallelize as far as possible the computation of local contributions to exploit all available processor cores on a shared memory machine.
(2) Synchronize the reduction operation to avoid race conditions in write accesses to data in `global_object`.
(3) Sequence the reduction operation in such a way that the order in which local contributions are added to the global object is repeatable when running the same executable on the same machine.

We will show in the next section how all of these goals can be achieved.

Remark 2. An alternative approach to avoiding the problem of synchronization in most of the loops mentioned earlier is to consider loops over independent chunks of the global data structure instead of the cells, possibly duplicating some work on adjacent cells. This approach was used, for example, for edge-based CFD solvers [Löhner and Galle 2002], with a loop over vertices or edges instead of cells and where the additional computations could be hidden completely in a memory-intensive loop. Since the number of adjacent elements to a vertex can be more than one layer away for adaptive finite elements with hanging nodes, the overhead in computations is typically too large, and we do not consider that approach.

5. IMPLEMENTING *WORKSTREAM*

Having laid out the kind of operation we want to parallelize on multicore machines, in this section, we provide three implementations of this pattern. They will differ in several implementation details, but will follow a general interface whereby we can write code as follows:

```
GlobalObject       global_object;
LocalContribution  local_contribution = ...;   // initialization
ScratchData        scratch_object     = ...;   // initialization

work_stream ([c_0, c_N),
             &compute_local_contribution,
             ⊕=,
             scratch_object,
             local_contribution,
             global_object);
```

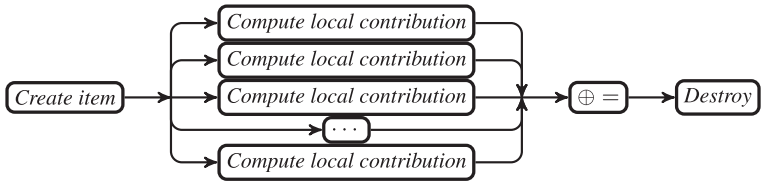

Fig. 1. Visualization of the *Pipeline* upon which we base the implementation of the *WorkStream* pattern as discussed in Sections 5.1 and 5.2.

Here, the first argument in calling `work_stream` denotes the range of objects to work on; the second and third arguments are *function objects* for the computation of local contributions and the reduction operations, respectively; and the fourth and fifth arguments are properly initialized scratch and local contribution objects that serve as templates from which similar objects can be copy-constructed whenever we need them. The last argument is the output argument. At the end of executing this code sequence, `global_object` should contain the same state as if the sequential loop of the previous section had been executed for a defined and reproducible but otherwise arbitrary order of the objects $[c_0, c_N)$.

The following subsections describe three possible implementations of the function above. They assume that an implementation for the *parallel pipeline* design pattern [Mattson et al. 2004] is available. In our experiments, we use the one available through the Threading Building Blocks [Reinders 2007]. The implementations we discuss illustrate our learning process with this pattern, and the final implementation discussed in Section 5.3 is now part of the DEAL.II library and available under an open source license at http://www.dealii.org/.

5.1. A First Implementation

The basis for all implementations here is the *parallel pipeline* pattern [Mattson et al. 2004]. *Pipeline* consists of a set of *stages,* each represented by a function with input and output arguments, where the output of one stage serves as the input of another one. Operations on different stages are assumed to be data-independent of each other. If the function on one stage of the pipeline is *pure* (i.e., it only acts on its input and output arguments but not any global state that might change as the loop is executed), then this stage of the pipeline can run in parallel on multiple cores of a shared memory machine if it has its own set of input and output objects. Making use of these last two statements enables parallelism. Furthermore, using this pattern, none of the functions needs to explicitly lock any resources.

In the current context, the pipeline we use to model *WorkStream* can be visualized as in Figure 1. The pipeline has four stages, of which the second can run in parallel on as many cores as are available. Each of these stages is represented by a function with input and output arguments that we will describe later. In addition, this implementation of *WorkStream* has some state variables common to the entire parallel execution unit of this pipeline (but not shared with any other pipelines that may be running at the same time):

```
class WorkStream {
  list<pair<ScratchData, bool> >          scratch_objects;
  list<pair<LocalContribution, bool> > local_contributions;
};
```

These lists represent scratch objects and objects holding local contributions that can be used by the second and third stages of the pipeline. Because their creation and

initialization may be expensive, we want to reuse them and consequently consider the boolean second part of each pair in these lists an indication of whether the object is currently in use or not. Before starting the pipeline, we initialize the lists with at least as many elements as the maximal number of tasks that can be concurrently processed by the underlying implementation of *pipeline*. We found that limiting this number to two times the number of CPU cores is reasonable. Initialization of the list elements happens by copy construction from the templates passed as fourth and fifth arguments in the call to `work_stream()`.

In addition to the preceding variables, the state of the *WorkStream* object contains pointers to the global object into which results are to be added and function objects representing the `compute_local_contributions()` and $\oplus =$ operations.

Given these, the functions that make up the four stages of the pipeline look like this:

(1) *Create:* The function that implements this stage has no input and outputs a tuple that consists of a cell and pointers to scratch and local contribution objects. Its implementation looks like this:

```
tuple<Cell, ScratchObject*, LocalContribution*>
stage_1 () {
  Cell c = next element of [c_0, c_N);

  ScratchObject *s = pointer to first unused element of
                     'scratch_objects';
  set 'used' flag for 's';

  LocalContribution *l = pointer to first unused element of
                         'local_contributions';
  set 'used' flag for 'l';

  return (c,s,l);
}
```

Because the two lists for scratch objects and local contributions are at least as large as the total number of items currently being worked on, we are guaranteed that each of the two searches in this function will actually succeed.

Note that since this stage of the pipeline writes into global data structures, it can only be run once concurrently. On the other hand, this also implies that no locking is necessary.

(2) *Compute local contribution:* The implementation of the second stage is obvious:

```
tuple<Cell, ScratchObject*, LocalContribution*>
stage_2 (Cell c, ScratchObject *s, LocalContribution *l) {
  *l = compute_local_contribution (c, *s);
  return (c,s,l);
}
```

Because the objects `*s` and `*l` are specifically assigned to an item that is being moved through the pipeline and because `compute_local_contribution` is supposed to be pure, this stage of the pipeline can be executed in parallel. It is typically the most expensive part of the overall work.

(3) $\oplus =$*:* This stage is equally obvious:

```
tuple<Cell, ScratchObject*, LocalContribution*>
stage_3 (Cell c, ScratchObject *s, LocalContribution *l) {
  global_object ⊕= *l;
  return (c,s,l);
}
```

Because this function writes into the global object, it cannot be run in parallel. However, running it on only a single thread at a time also obviates the need for locking.

(4) *Destroy:* The final stage simply marks the scratch and local contribution objects as usable again:

```
void
stage_4 (Cell c, ScratchObject *s, LocalContribution *l) {
  clear 'used' flag for 's' in 'scratch_objects';
  clear 'used' flag for 'l' in 'local_contributions';
}
```

This function may write to flags at the same time as the stage 1 function. However, no race conditions are possible as long as reads and writes to these flags are atomic. In practice, one can avoid this complication by taking scratch and local contribution objects round-robin from a buffer with a sufficiently large number of elements.

Using these functions and a reference to the context object for the state of the algorithm, we can implement the *WorkStream* pattern using an existing implementation of *pipeline*.

Remark 3. The preceding implementation does not, by itself, guarantee reproducible results unless items are processed in a predictable order by the third stage. Fortunately, some implementations of the *pipeline* pattern—notably the one in the Threading Building Blocks [Reinders 2007]—guarantee that sequential stages see items in exactly the same order as they were generated by the first stage. This is sufficient to make results reproducible even if the reduction operation $\oplus$ is not associative.

Remark 4. The overhead for the implementation of this pattern can be reduced by not working on one cell at a time, but instead passing whole batches of cells through the pipeline. This implies that one also has to pass batches of `LocalContribution` objects along. On the other hand, since the scratch objects carry no state between iterations, a single scratch object per batch is sufficient.

We will explore the idea of batches of cells further in Remark 6.

5.2. An Implementation with Thread-Local Scratch Objects

In typical finite element computations, the scratch objects are rather heavy, with many small arrays containing the values and derivatives of shape functions at quadrature points, members providing transformations from the reference cell to the real cell, and the like. The design used in the previous subsection used scratch objects taken round-robin from a global pool. On machines where each processor core has its own cache, this almost certainly leads to a large number of cache misses because a thread working on computing a local contribution rarely will get the same scratch object it had while working on the previous cell. Worse, by preallocating these objects before even starting the pipeline, NUMA machines will typically create the objects in memory associated

with only one processor and consequently far away from those cores trying to load a scratch object into their caches.

This can be avoided if we make the scratch objects thread-local. With this modification, the second implementation of the design pattern uses a global state of the kind:

```
class WorkStream {
  threadlocal list<pair<ScratchData,bool>>        scratch_objects;
  list<pair<LocalContribution,bool>>              local_contributions;
};
```

The `local_contribution` objects are not thread-local since they are passed from the second to the third stage of the pipeline and, consequently, may be accessed from different threads. The implementation then uses the same pipeline as in Figure 1, with the following four functions:

(1) *Create:* The function that implements this stage has no input and outputs a tuple that consists only of a cell and a local contribution. Its implementation looks like this:

```
class WorkStream {
  list<pair<ScratchData,bool>>        scratch_objects;
  list<pair<LocalContribution,bool>> local_contributions;
};
```

(2) *Compute local contribution:* The second stage computes the local contributions. Its implementation now consists of the following code:

```
tuple<Cell,LocalContribution*>
stage_2 (Cell c, LocalContribution *l) {
  ScratchObject *s = first unused element of 'scratch_objects'
                     in the list specific to this thread;
  if (no unused element found) {
    add an element to the list;
    s = newly added element;
  }
  set 'used' flag for 's';

  *l = compute_local_contribution (c, *s);

  clear 'used' flag for 's';

  return (c,l);
}
```

Because the list of scratch objects is thread-local, there is no need to guard access to this list using a mutex. Typical implementations of the *pipeline* pattern pin their worker threads to individual processor cores. Consequently, the scratch objects in this list are not only specific to a thread, but are in fact located in the memory associated with the processor the current thread runs on and will also frequently already be in the cache of the current core.

(3) $\oplus =$*:* This stage uses the same function as before, simply omitting the scratch object from the argument and return value lists.

(4) *Destroy:* The final stage simply marks the local contribution object as usable again:

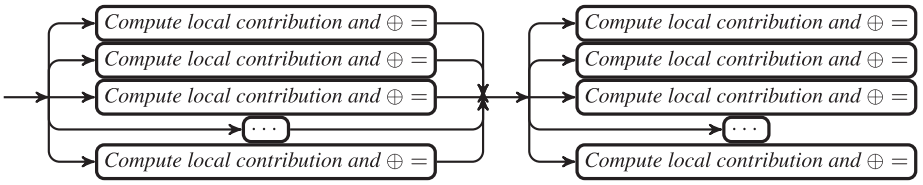

Fig. 2. Visualization of the *parallel-for* for an example with two colors. Colors are treated sequentially but cells within a color are processed independently on as many cores as are available. The implementation of the *WorkStream* pattern as discussed in Section 5.3 uses this scheme.

```
void
stage_4 (Cell c, LocalContribution *l) {
  clear 'used' flag for 'l' in 'local_contributions';
}
```

One may be tempted to think that a single scratch object per thread—rather than a list—is sufficient. After all, the `stage_2` function uses it from the start to the end, and while there may be other instances of `stage_2` running at the same time, they are on different threads and consequently cannot use the thread-local scratch object of the current thread. In other words, because on the current thread only one instance of `stage_2` can run at any given time, there is no need for multiple scratch objects. However, this is not true in general for task schedulers such as the one in the Threading Building Blocks [Reinders 2007]. In particular, whenever something in the implementation of `compute_local_contribution` starts other tasks and then waits for their completion, the task scheduler may run another instance of `stage_2` on the same thread while the first instance sleeps. Consequently, every thread needs a list of scratch objects. In almost all cases, this list will be very short, however.

5.3. An Implementation using Graph Coloring

Having addressed one inefficiency, we are left with the realization that any implementation with a serial third stage running the $\oplus =$ reduction operation cannot scale to large numbers of processors if the reduction operation takes a non-negligible fraction of the runtime of the `compute_local_contribution` function. This is a simple consequence of Amdahl's law [Amdahl 1967; Patterson and Hennessy 2009]. In practice, this condition means that if the number of cores exceeds the ratio `cputime(compute_local_contribution)/cputime(`$\oplus =$`)`, then scalability breaks down.

In actual finite element codes, computing local contributions is often an expensive operation, requiring looking up coefficients in equations, evaluating shape functions, computing integrals by quadrature, and more. On the other hand, adding local contributions to the global object typically requires much less time. As we will show in detail in the next section, the ratio of runtimes for these two functions is typically in the range of 3–100, with some exceptions. Thus, we can expect the previous two implementations to scale on current generation laptops, but not on current workstations with up to 64 cores. Furthermore, we cannot expect these implementations to scale well on future hardware.

To address this deficiency, we need to fundamentally rethink the algorithm. To this end, recall that we run the third stage sequentially because it writes into a globally shared object, and we expect these writes to conflict if they happen at the same time. However, in many cases, we can be more specific. For example, in matrix assembly, not all writes conflict with all others: For continuous elements, the reduction operations for cells conflict with each other only if these cells share degrees of freedom. Cells that are not neighbors write into separate rows of vectors and matrices and consequently do not conflict.

For these kinds of operations, we can introduce a preprocessing step in which we *colorize* the set of cells in such a way that cells for which the reduction operations conflict are assigned different colors [Saad 2003; Kubale 2004]. Cells that do not conflict may have the same color. By consequence, all reduction operations for cells of the same color cannot conflict and may run at the same time. This approach has previously been used in the finite element context for element-by-element matrix-vector products on vector supercomputers [Berger et al. 1982; Carey et al. 1988; Farhat and Crivelli 1989], explicit time stepping in spectral element codes on GPUs [Komatitsch et al. 2009], matrix-free techniques in finite elements [Kormann and Kronbichler 2011], or assembly of matrices [Cecka et al. 2011; Logg et al. 2012]. Coloring is obviously generalizable if the elements of the range $[c_0, c_N)$ we iterate over are not cells but other objects in finite element codes, such as faces for boundary integrals, and the like.

If such coloring is efficiently possible, we can use an implementation where the state of the *WorkStream* is now described by the following variables:

```
class WorkStream {
  threadlocal list<pair<ScratchData,bool>>        scratch_objects;
  threadlocal list<pair<LocalContribution,bool>> local_contributions;
};
```

With this, we can implement the *WorkStream* pattern using a parallel-for as shown in Figure 2. We no longer need a sequential *Create item* stage because, for a given color, the order in which the elements are processed does not matter, and a parallel-for implementation can freely schedule *any* item (not necessarily in any particular order) onto available threads. Thus, we only need the following function:

(1) *Compute local contribution and* $\oplus =$*:* This stage computes the local contributions and adds them to the global object. *Cell* c is an element of $\in [c_0, c_N)$. All the cells worked on simultaneously by the parallel-for must have the same color. The implementation of this stage is obvious:

```
void
stage (Cell c) {
  ScratchObject *s = first unused element of 'scratch_objects'
                     in the list specific to this thread;
  if (no unused element 's' found) {
    add an element to the list;
    s = newly added element;
  }
  set 'used' flag for 's';

  LocalContribution *l = first unused element of 'local_contributions'
                         in the list specific to this thread;
  if (no unused element 'l' found) {
    add an element to the list;
    l = newly added element;
  }
  set 'used' flag for 'l';

  *l = compute_local_contribution (c, *s);
  global_object += *l;

  clear 'used' flag for 's';
  clear 'used' flag for 'l';
}
```

Note that both the scratch and the local contribution object are now thread-local and presumably in the cache of the processor core executing the function. This implementation may execute the reduction operation $\oplus =$ multiple times in parallel, but because we only work on cells of the same color, by definition, these writes do not conflict.

We implement the *WorkStream* pattern by first finding a colorization of the elements of the range $[c_0, c_N)$ and then, for each color, running a parallel-for loop over all elements of this color. This allows a completely parallel execution of all significant operations.

Remark 5. To fully exploit parallelism, it is important that the colorization does not produce colors with a small number of elements. In particular, colors that contain less elements than a small multiple of the number of cores will likely yield poor scaling. To avoid this, we need a colorization algorithm that attempts to equilibrate the sizes of the individual colors. We will describe such an algorithm and show its performance in Appendix A.

Remark 6. As with the other implementations, one can limit the overhead of scheduling items to threads by not working on individual cells but on batches of cells instead. In the experiments we show in the next section, we use a batch size of 8. This also implies that implementation 1 requires memory for 8 (batch size) times the number of items-in-flight `LocalContribution` objects since they can be released again after the reduction stage (which runs concurrently to the local-assembly stage of the pipeline). We limit the number of items in flight to two times the number of CPU cores (`ncpus`), resulting in a total of `16 ncpus` memory slots for `LocalContribution` objects. Because they can be reused on each cell of a batch, we only need `2 ncpus` slots for `ScratchData` objects.

Likewise, for implementation 2, the numbers are `16 ncpus` memory slots for `LocalContribution` objects and slightly more than `ncpus` slots for `ScratchData` objects, taking into account that we may need more than one `ScratchData` per thread, as explained at the end of Section 5.2. Finally, for implementation 3, we need slightly more than `ncpus` slots for both `LocalContribution` and `ScratchData` objects.

In all cases, these numbers are independent of the number of cells we work on (i.e., memory consumption is $O(1)$) and, in general, negligible compared to the sizes of typical mesh and matrix data structures for processor core counts available today.

Using a colorization of all cells typically results in processing cells out of order with regard to the way their properties are stored in memory. This implies that our memory access pattern will not benefit from data locality, and this will have to be balanced with the improvement in parallel efficiency. On the other hand, blocking (as described in Remark 6) can regain some of the data locality.

5.4. Alternative Strategies

In the preceding subsections, we focused on strategies that use shared memory and multiple threads to parallelize work. We would be amiss if we did not mention possible alternatives to achieve the same goal of utilizing available processing power for parallelization. In particular, we discuss data duplication and distributed data approaches in the next few paragraphs. We will also use the distributed data approach to provide baseline compute times for our numerical experiments in Section 6.

Duplicating Entire Data Structures. One of the fundamental difficulties we address in our algorithm designs is that multiple threads want to write into shared data structures concurrently. We have addressed this by sequentializing potentially conflicting writes (implementations 1 and 2), and by graph coloring to avoid conflicting writes

(implementation 3). However, the simplest approach is to simply duplicate the *entire* data structure we write to as many times as there are threads, schedule cells freely to threads, have each thread write into its own copy of the now thread-local object, and have a sequential reduction over this relatively small number of objects at the end.

This approach works as long as the output data structure is relatively small and the final reduction is cheap. For example, later we discuss a "*Depth average*" operation for which the output is a small array, and, in this case, the data duplication approach is completely reasonable—and vastly faster. On the other extreme, in the 3d cases we discuss later, the output data structure for the "*Assemble Stokes system*" operation is a matrix that requires 3.5GB—a size that cannot be replicated 64 times for our largest computations.

Because the memory usage of this approach scales linearly with the number of processor cores, we do not consider it a viable solution for the general cases we want to address in this article and will not consider it further.

Splitting Data Structures Into Pieces. Alternatively, one could duplicate only *parts* of the global data structure; namely, so that each thread's local output object consists of that part of the global object the thread actually writes to. This is the approach typically taken when parallelizing work on distributed memory machines, for example, using MPI: There, every processor stores a part of the triangulation, of the matrix, and the like, with some data duplication where necessary (e.g., for ghost cells and vector ghost elements).

There are downsides to this approach: First, it requires static partitioning of cells onto processors/cores/MPI processes to ensure that each only writes into that part of the partitioned data structure that it actually owns. This often leads to imbalances if operations take different relative amounts of time on cells. An example is that the assembly on some, but not all, cells requires additional boundary integrals, whereas creating graphical output takes the same amount of time on all cells. No static partition of cells onto MPI processes can be optimal for both of these operations at the same time. The implementations outlined in the previous subsections do not suffer from this setback because they dynamically schedule work items onto available compute resources.

Second, the fraction of duplicated data may become prohibitive on systems with many cores but little memory, such as GPUs or the Intel Xeon Phi. To give an example, the 3d examples we show later have 49,152 cells. With 64 cores, each core's partition would consist of 768 cells, corresponding to a block size of roughly $9 \times 9 \times 9$. With one layer of ghost cells, each processor needs to store $11^3 = 1.7 \cdot 768$ cells—a 70% overhead on the mesh data structures (the overhead would be larger if using an unstructured mesh in which the domains for each processor are not as compactly structured).

Finally, to make the approach workable, it is necessary to have efficient strategies to partition *every* data structure in a program. Doing so will yield a program that will likely scale to hundreds or thousands of processor cores. The results we show demonstrate that such a programming model is still the gold standard if pure speed is the only relevant metric. At the same time, it is difficult to redesign existing codes and their data structures, and it is also often not necessary for many applications for which a single shared-memory workstation is large enough to achieve the desired accuracy but for which parallelization would nevertheless provide a welcome speedup. The algorithms described herein do not require modification of data structures and therefore provide a much simpler solution to using parallel machines.

Given these drawbacks, we believe that neither complete duplication nor partitioning of data structures is a complete solution to parallelizing operations such as those discussed in this article and that, therefore, there is a need for the algorithms described

herein. In particular, if the metrics to measure success of a project include programming effort, programmability, debuggability, memory overhead, and other quantities not immediately related to walltime only, then the algorithms provided here are attractive.

6. EXPERIMENTAL INVESTIGATION OF SCALABILITY

Having described three possible implementations of the *WorkStream* design pattern, in this section we evaluate their parallel scalability using some of the use cases we identified in Section 4.1. To this end, we will use a version of the step-32 tutorial program of DEAL.II [Kronbichler et al. 2013] with some additional timers around sections of interest. Step-32 solves the Boussinesq equations describing thermal convection and is the basis for the much larger open source code ASPECT [Kronbichler et al. 2012] for which we showed a breakdown of overall runtime in Section 2. For our computational experiments, we work with step-32 instead of ASPECT since it is much simpler (ASPECT has about 10 times as many lines of code) and the individual operations are much simpler to time separately. However, step-32 does essentially the same computations and is, consequently, a realistic testcase.

Step-32 alternates between solving the Stokes system,

$$-\nabla \cdot (\eta \varepsilon(\mathbf{u})) + \nabla p = \rho(T)\mathbf{g},$$
$$\nabla \cdot \mathbf{u} = 0,$$

for the flow field characterized by velocity $\mathbf{u}$ and pressure p as a function of the temperature-dependent density $\rho(T)$, and the advection diffusion equation for the temperature,

$$\left(\frac{\partial}{\partial t} + \mathbf{u} \cdot \nabla\right) T - \nabla \cdot \kappa \nabla T = 0.$$

Here, η, κ are viscosity and thermal diffusion coefficients, and $\varepsilon(\mathbf{u})$ is the strain tensor. More details on the algorithm can be found in Kronbichler et al. [2012]. While step-32 can also utilize distributed memory parallelism, we here only use it as a reasonably complex testcase for shared memory parallelization on a single multicore machine.

Of the components of this program, we will look in particular at the following operations:

—*Assemble Stokes system:* In this part of the program, we need to assemble the matrix and right-hand side that represent the discretized Stokes operator, namely

$$A_{ij} = (2\eta\varepsilon(\boldsymbol{\varphi}_{i,u}), \varepsilon(\boldsymbol{\varphi}_{j,u})) - (\nabla \cdot \boldsymbol{\varphi}_{i,u}, \varphi_{j,p}) - (\varphi_{i,p}, \nabla \cdot \boldsymbol{\varphi}_{j,u}),$$
$$F_i = (\boldsymbol{\varphi}_{i,u}, \rho(T^{n-1})\mathbf{g}).$$

Here, $\boldsymbol{\varphi}_{i,u}$ and $\varphi_{i,p}$ are the velocity and pressure components of a shape function φ_i. While the right -hand side F is computed in every time step, the matrix A only needs to be rebuilt whenever the mesh changes (as long as the viscosity is constant).

The operation requires the assembly of local contributions on every cell, as well as the addition of these local contributions to sparse matrix and vector objects we use from the Trilinos library [Heroux et al. 2005, 2014]. The $\oplus =$ function adding local contributions to the global object also resolves constraints within the finite element space, for example due to hanging nodes or boundary conditions.

—*Assemble Stokes preconditioner:* The preconditioner we use in this program builds an algebraic multigrid hierarchy using the ML package [Tuminaro and Tong 2000; Gee et al. 2006] of Trilinos, based on the matrix

$$B_{ij} = (\eta \nabla \boldsymbol{\varphi}_{i,u}, \nabla \boldsymbol{\varphi}_{j,u}) + (\eta^{-1}\varphi_{i,p}, \varphi_{j,p}).$$

The velocity-velocity block of this matrix is considerably sparser than that of the full Stokes matrix since it does not couple the individual components of the velocity vector (except, possibly, at boundaries with tangential flow boundary conditions, where we have constraints that do couple different velocity components). This simplifies the construction of multilevel hierarchies. The preconditioner matrix also contains an approximation to the Schur complement in the form of a mass matrix on the pressure space scaled by the inverse of the viscosity. These two matrices form the ingredients for a block-triangular Schur complement preconditioner [Silvester and Wathen 1994].

As before, the two operations we have to implement are the computation of local contributions and adding local contributions to the global matrix while resolving constraints. We do not consider computing the actual preconditioner here.

—*Assemble temperature matrix:* Since step-32 treats the temperature equation semi-implicitly, we need to assemble the matrices

$$C_{ij} = (\psi_i, \psi_j), \qquad D_{ij} = (\kappa \nabla \psi_i, \nabla \psi_j).$$

Here, the ψ_i are the shape functions corresponding to the temperature variable. The matrix to be inverted in a time step will be a linear combination of these two matrices, with coefficients related to the time step. Local operation and reduction are as given earlier.

—*Assemble temperature right-hand side:* While the two temperature matrices only need to be computed every time the mesh changes, the right-hand side needs to be computed in every time step. This involves computing a complicated vector that contains linear combinations of the previous two temperature solutions, an artificial viscosity that depends nonlinearly on a variety of coefficients, and the advection term. The exact formula is not enlightening and can be found in Kronbichler et al. [2012]. Due to the variety of factors and the complexity of computing the artificial viscosity, the computation of local contributions in this operation is more expensive than for any of the other preceding operations. On the other hand, the result is a small local vector that needs to be added to the global vector—a relatively cheap operation.

—*DataOut:* This operation evaluates the solution at the vertices of every cell and converts this information into an intermediate form. This produces a structure for every cell that the reduction operation then inserts into an array. The array is at a later time converted into one of a variety of file formats for graphical visualization. The reduction operation is unnecessary here since each result is simply put into a separate array slot, rather than being added to an existing piece of data.

—*Depth average:* The final operation we describe is a typical post-processing operation. It evaluates the temperature value at the center of the cell and uses this to compute temperature averages over layers of the domain at specific depths. In our examples, we subdivide the domain into 100 such layers and the result of each cell then needs to be added to an element of an array of size 100 that stores the sum of cell-local temperature integrals (approximated by the one-point midpoint quadrature) and the corresponding element of an array of equal size storing the sum of areas of the cells. The final average temperature over each layer is then the ratio of these sums.

As mentioned at the beginning of Section 5.3, the first two implementations of our software design pattern can only scale well if the ratio of run times of the functions implementing the computation of local contributions and the reduction operation is large. To this end, we have measured these ratios for the six areas outlined earlier. Runtimes and ratios for two-dimensional computations are listed in Table II.[2] From

[2]The data shown in the table were measured by running the six operations on a single thread on all cells of the mesh in sequence and dividing the runtime by the number of cells. This workload is not representative

Table II. Runtimes and Ratios of Runtimes for the Computation of Local Contributions and the Reduction Operation, for the Six Operations Discussed in Section 6 and Computations in Two Space Dimensions

Operation	CPU time (local contribution)	CPU time ($\oplus =$)	Ratio
Assemble Stokes system	19 μs	1.0 μs	19
Assemble Stokes preconditioner	30 μs	1.1 μs	27
Assemble temperature matrix	9.0 μs	1.1 μs	8.1
Assemble temperature r.h.s.	39 μs	1.0 μs	39
DataOut	8.1 μs	0 μs	∞
Depth Average	1.3 μs	0.036 μs	36

CPU times are provided as single-processor runtime per execution of the function, averaged over all calls to these functions on all cells of the mesh and over several time steps.

these data, we would expect the first two implementations to scale well for the assembly of the temperature right-hand side and for *DataOut* (and, to a lesser degree, for the assembly of the Stokes system and preconditioner), whereas the remainder of the operations should stop scaling to larger processor numbers relatively quickly. (Computing the depth average shows other peculiarities discussed later.)

To evaluate these claims experimentally, we performed computations on a workstation with 4 16-core AMD Opteron 6378 (Abu Dhabi) processors running at 2.4GHz, for a total of 64 cores. The system has 128GB of memory. The numerical experiments were run in two space dimensions on a mesh with 786,432 quadrilateral cells using quadratic finite elements for the velocity and temperature variables and linear elements for the pressure, resulting in a total of 11,814,912 unknowns. This problem size is large enough to keep the machine busy for sufficiently long times to allow for accurate timing.

Figure 3 presents the actual (walltime) runtime as a function of the available number of processor cores for each of the three implementations discussed in Section 5.[3] To put the achieved speedup in context, we also include numbers for an MPI-only parallelization of the program [Kronbichler et al. 2012] following the data partitioning approach discussed in Section 5.4. We then repeated the same computations for three space dimensions on a test case with 49,152 cells and 1.8 million degrees of freedom, and we show results in Figure 4.

In the following paragraphs, we interpret these results in a number of ways. We will discuss the last two operations (*DataOut* and *Depth average*) separately at the end, since they exhibit significantly different performance characteristics.

Interpreting Implementations 1 and 2. The results shown in the figures indicate that NUMA and cache effects on the scratch data do not appear to be significant issues for any of the operations—in all examples, implementation 2 is slightly faster than implementation 1, but the difference is never very large.

That said, studying the scalability of these two implementations validates the underlying motivation for studying implementation 3: As discussed at the beginning of Section 5.3, we cannot expect the first two implementations to scale beyond a number of cores equal to the ratio of runtimes of the local computation to reduction operation. Indeed, the kinks in the 2d curves (Figure 3) at a number of cores where these implementations stop to scale well are well correlated with the ratios provided in the last column of Table II.

---

of the multithreaded cases since it ensures that all scratch objects and local contributions are always in the cache. As a consequence, we suspect that "real" runtimes are larger than stated in Table II and in particular that the ratios given in the last column would be lower in a multithreaded context.

[3]We show the best of three runs for each data point. All experiments were done on an otherwise empty machine, and we observed very little variation between individual runs.

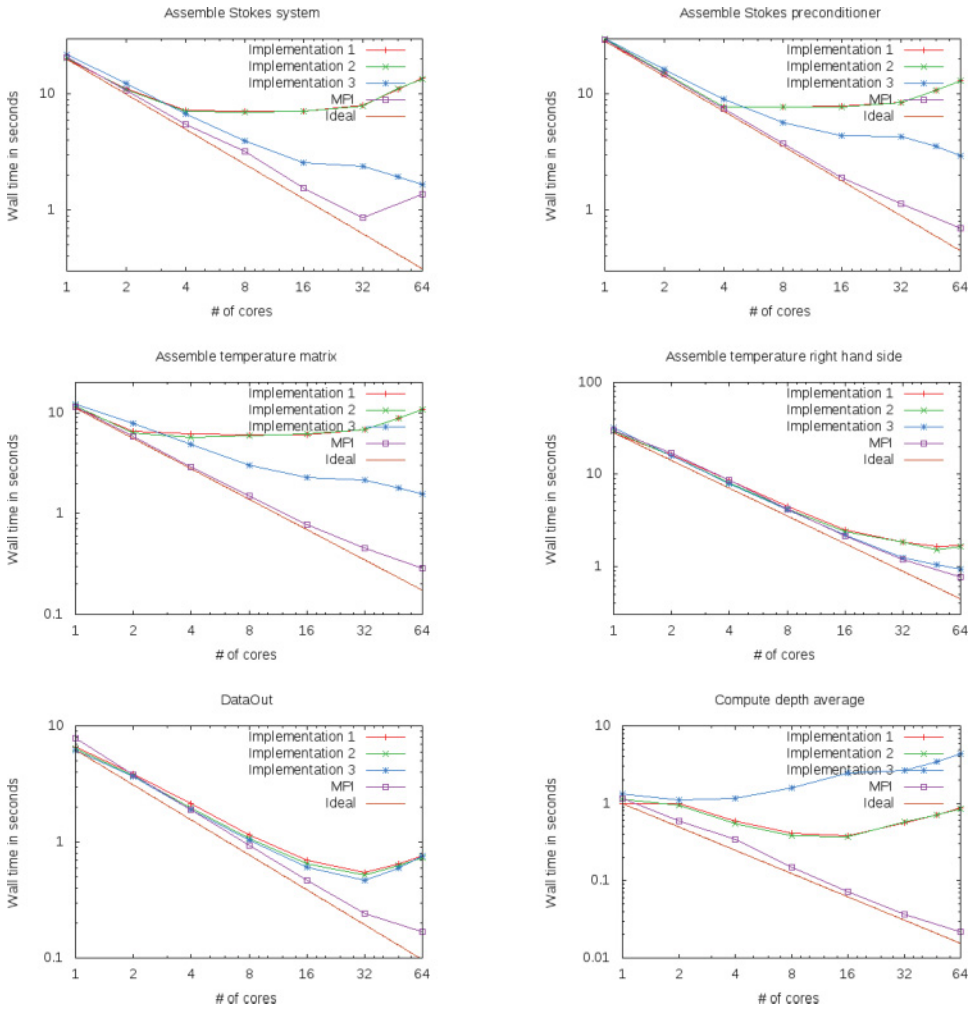

Fig. 3. Walltime for the six operations discussed in Section 6 as a function of the number of processor cores utilized. Computations were performed for a two-dimensional domain and are averaged over multiple time steps.

This consideration already suggests that the first three operations will not be able to gain a significant speedup when using implementations 1 or 2 once the number of processor cores exceeds that of a well-equipped laptop of the current generation.

Interpreting Implementation 3. Compared to the first two implementations, implementation 3 discussed in Section 5.3 does significantly better on the first 4 test cases, achieving overall speedups compared to a single processor in the range of 7.7 to 36 when using up to 64 cores in 2d computations. In 3d, we achieve speedups between 15 and 44. The speedup that results from switching from implementation 2 to implementation 3 is summarized in Table III for the 2d cases. It is obvious that, in several of the cases, further speedup can be expected by going to even more than the 64 cores we had access to.

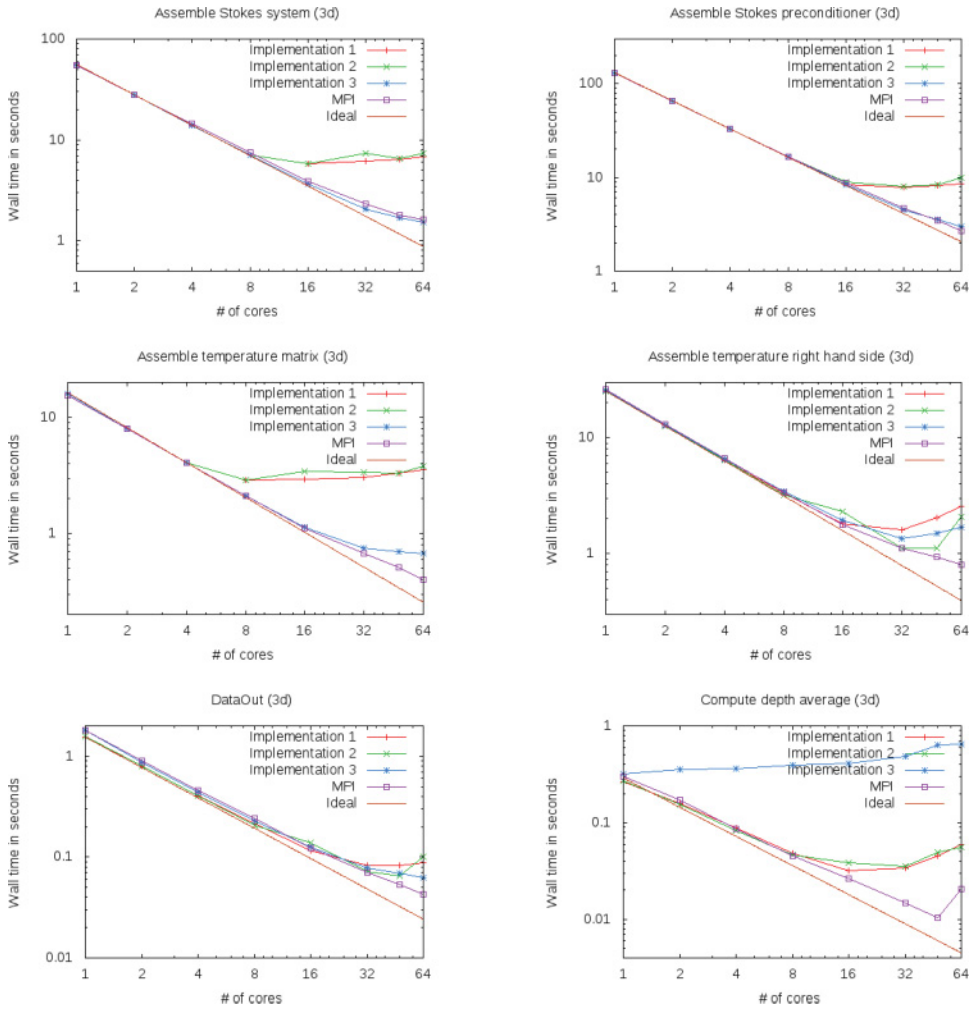

Fig. 4. Walltime for the same six operations as discussed in Section 6 as a function of the number of processor cores utilized. Compared to Figure 3, computations here were performed in a three-dimensional domain.

Table III. Comparison of the Best Runtimes Achieved with Implementations 2 and 3 (Along with the Number of Cores at Which This Time was Achieved) for Two-Dimensional Computations

Operation	Best time implementation 2	Best time implementation 3	Ratio
Assemble Stokes system	7.0s (8 cores)	1.66s (64 cores)	4.2
Assemble Stokes preconditioner	7.75s (8 cores)	2.94s (64 cores)	2.6
Assemble temperature matrix	5.73s (4 cores)	1.57s (64 cores)	3.6
Assemble temperature r.h.s.	1.51s (48 cores)	0.94s (64 cores)	1.6
DataOut	0.52s (32 cores)	0.47s (32 cores)	1.1
Depth Average	0.37s (16 cores)	1.09s (2 cores)	0.33

The last column shows the speedup that can be achieved when replacing implementation 2 by implementation 3.

Interpreting the DataOut and Depth Average Operations. The last two operations show different characteristics that result from how independent the operations are on cells.

In the case of *DataOut*, the work on every cell is completely independent of that on others since the local worker function can already put the processed data into the correct and unique memory location. There is, consequently, no need at all for a reduction operation. However, the scaling is not perfect. Analysis shows that this can be traced back to frequent memory allocation for temporary data and the associated lock contention in the system's memory allocator (data that would ideally go to ScratchData instead) and, to a lesser extent, NUMA effects on the output array.

The *Depth average* operation represents the other extreme. There, all cells produce results that need to be written into an array of only 100 elements. Consequently, each color can only have at most 100 cells, and some colors may have significantly fewer. Given our batch size of 8 (see Remark 6), the best we could possibly hope for is to saturate at most around 12 cores. In reality, given how cheap the operation on every cell is, the more complex scheduling of implementation 3 negates any speedup at all; the obvious remedy—increasing the batch size—is not an option here since the size of colors is already so small. Compared to this, implementation 2 can at least achieve a speedup of slightly over 3, again limited by the ratio of runtimes and the overhead of scheduling.

The conclusion of these considerations is that how well the implementations of the *WorkStream* pattern work depends crucially on how many conflicts there exists between work items (i.e., how many colors we need to partition the cells). In other words, while our design pattern *formally* applies to this situation, it is not a good match *in practice*. As mentioned in Section 5.4, the obvious solution for this operation is to duplicate data structures.

Limits to Scalability. Our experiments show very significant speedups, but no implementation provides ideal speedup. Let us here examine some of the reasons. First, in the 2d examples used earlier, the partitioning of the mesh into colors is almost optimal (6 colors with 131,030, 131,070, 130,988, 131,201, 131,112, and 131,031 cells, respectively). Therefore, a limited amount of available parallelism is not the reason for the lack of scalability. Moreover, the scheduling overhead appears to be relatively small and to have a minor impact: In our implementation, we follow the remarks at the end of Sections 5.1 and 5.3 and do not work on individual cells, but instead on batches of eight. Thus, the first color yields 16,379 batches of cells, for an average of 256 batches that need to be scheduled to each of the 64 cores in the largest configuration. This should still be a sufficiently large number of batches per core to balance cases where some batches require more work than others. Increasing the batch size to 64 decreases the runtime by 5–20% compared to those shown in Figure 3, but when increasing the size even further we get increasing runtimes again. (One could imagine an adaptive choice of batch size here.) In order to quantify the cost of synchronization, we also ran the experiments for successively smaller problem sizes, and parallel speedup can be recorded down to a baseline of approximately 6 to 12 milliseconds for the tasks in panels 1 to 4 of Figures 3 and 4, with saturation when the global mesh has less than 1,000 elements.

Instead, we found the primary reason for the lack of scalability on the assembly test cases to be memory and cache effects when accessing global data structures. This is also obvious by comparing the 2d and 3d data in Figures 3 and 4: The 3d runs require significantly more work both to compute local contributions and to accumulate data into the global data structures, making cache effects less important; consequently, the graphs show much better scalability than in the 2d cases. Since investigating details

of cache usage goes beyond the scope of this article but toward the design of the data structures that local assembly and reduction operations work on, we will not explore this in detail but only provide a limited overview. First, the sparse matrix objects we use (based on the Trilinos library, Heroux et al. [2014]) initialize memory in a single-threaded way and thereby in memory attached to a single processor. When all 64 cores simultaneously accumulate results into the matrix, this nonuniform memory access results in a serious bottleneck and limits the parallel speedup to about 13 for the 2d assembly of the Stokes system, for instance. Using preliminary work replacing this data structure by one in which each processor initializes a part of the matrix memory (first touch), the parallel speedup for the Stokes assembly increases to about 22. Second, cache performance when accessing global objects such as the matrix or the mesh geometry is nonoptimal when coloring single cells. Indeed, the nature of coloring is that neighboring cells are worked on at very different times, and matrix rows, vertex coordinates, and other cell-dependent information between neighboring cells will in general no longer be in caches. The lower performance of implementation 3 for small processor counts in 2d can be attributed to this effect, too. To improve data locality, one can color whole batches of neighboring cells instead of individual cells. Coloring batches of 64 cells, performance is considerably improved. Combined with the NUMA-aware memory initialization, the parallel speedup on 32 cores is 28 and on 64 processors it is 42 for assembling the 2d Stokes system, considerably faster than even the MPI-only implementation.

Furthermore, scaling is affected by the total available memory bandwidth in the system, and an algorithm like matrix assembly that is computation bound in serial gets increasingly memory bandwidth bound in parallel. This is particularly relevant for the assembly of mass and Laplace matrices for the temperature, for which the computations are cheap (see Table II). Last, our speedup numbers may disfavor large processor counts because "turbo" capabilities in modern processors allow computations to run at higher core frequency if only few cores are in use.

Comparison with a Partitioned Data Method. To put the speedup numbers just reported in context, all the graphs also show speedups for an implementation of step-32 that uses MPI only and no threads (see Section 5.4). In this approach, all data structures are completely distributed and all operations on each MPI process can be run in a sequential manner, followed by a global exchange of data between processors. This approach has no write conflicts and has the advantage that MPI processes are pinned to individual processor cores, guaranteeing good cache usage. Consequently, it is not surprising that we achieve good speedup on most operations.

At the same time, we note that this can only be achieved if (i) the static partitioning of cells onto processors is good (which here is the case because the code does not need to compute different things on different cells, as is often the case in practice); (ii) the fraction of data duplicated between processes is small (i.e., if each processor has sufficiently much memory to store large chunks of data); and (iii) if the data structures we write into can be partitioned. The last point is often difficult to achieve in practice; for example, we use heavily tailored wrappers of Trilinos with several thousand lines of code to achieve decent speedup on matrix assembly operations.

Given these limitations, the results we obtain using *WorkStream* show that we can achieve good speedup even on large core counts using algorithms that allow us to dynamically schedule operations onto available resources and without the need to modify data structures. While the comparison—unsurprisingly—demonstrates that MPI provides for a more scalable programming model, our work shows that good speedup can be achieved without the drawbacks discussed in Section 5.4 and at a fraction of the programming cost.

7. CONCLUSION

In this article, we presented a design pattern for the parallel execution of common pieces of finite element computations—namely, the iteration over all cells or other entities—on multicore machines. Our approach addresses contexts that consist of an embarrassingly parallel computation part followed by a global reduction operation into a global object. We describe two simple implementations of the pattern that only parallelize the computation part and a more advanced scheme that uses graph coloring to create nonconflicting operations also in the reduction part. Our numerical results in Section 6 show the scaling properties of these implementations. However, our results also highlight that practical issues like optimal cache usage and nonuniform memory access become important as soon as the underlying algorithm allows for scaling to larger core counts. While these limitations are outside the scope of this article (they require changes in the data structures used), our generic implementation allows us to achieve speedups not far below the theoretical maximum for several of the cases we have investigated.

APPENDIX

A. EQUILIBRATING COLORIZATION OF FINITE ELEMENT MESHES

Implementation 3 of the *WorkStream* pattern required running a pipeline over the elements of each color in a colorized set of elements $[c_0, c_N)$. This can only be efficient if there are no colors with few elements. On the other hand, achieving a minimal number of colors is not important. In the following, we describe an algorithm that realizes these goals.

Our algorithm is composed of three stages: partitioning, coloring, and gathering. It requires a user-defined function that receives an element from the range $[c_0, c_N)$ and returns a set of conflict indices. Conflict indices encode memory locations that will be written to; they could either be pointers to memory or simply the rows of a matrix in the case of assembly. Two elements that have a common conflict index cannot be treated at the same time during the reduction operation. Next, we explain in detail the three stage of the algorithm:

—*Partitioning:* Partitioning divides the elements into sets called *zones* and works similar to a Cuthill-McKee algorithm. The first zone contains only the first element c_0. The second zone contains all the elements that share a conflict index with c_0. The third zone contains all the elements that share a conflict index with those in the second zone and that are not part of any previous zone. This is done until there are no elements left or until the elements left are not in conflict with the elements already associated with a zone. In the latter case, one of the remaining elements is used to create a new zone. Partitioning is done when no elements remain unassigned. In the following, we will call the first, third, fifth, ..., of these zones the *odd zones* and the remaining ones *even zones*. It is important to note that there is no conflict between the elements of one odd zone and those of another odd one (and similarly for two even zones), although there may, of course, be conflicts between the elements within one zone.
—*Coloring:* We color the elements within each zone using the DSATUR algorithm [Brélaz 1979]. Each color consists only of elements that do not conflict. Coloring different zones are independent operations and can be done in parallel.
—*Gathering:* The goal of the gathering stage is to gather colors in the odd (respectively even, zones) to create large colors that have similar number of elements. The gathering is done in the following way:

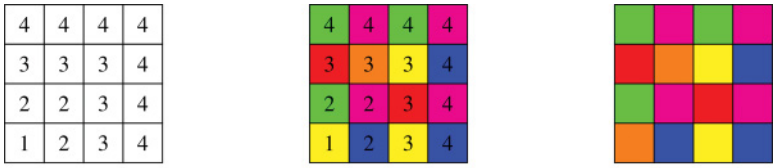

Fig. 5. The three stages of the colorization algorithm applied to a 4×4 uniform mesh.

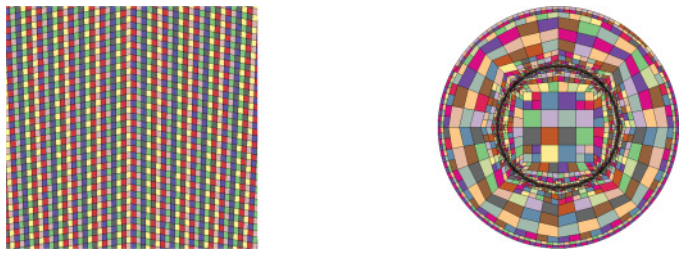

Fig. 6. Examples of colorized meshes.

(1) Search for the odd zone that has the largest number of colors (master colors).
(2) Choose an odd zone and add the elements of the largest color of this zone to the smallest master color.
(3) Search for the second largest color in this zone and add the elements to the smallest master color left (i.e., excluding the master color already used).
(4) Repeat the process for all the colors in all the odd zones.
(5) Apply the same steps for the even zones.

In Figure 5, the three stages or the algorithm are shown for a 4×4 uniform mesh. While not usually done in standard coloring algorithms (see Berger et al. [1982] and Komatitsch et al. [2009]), the initial subdivision of elements into zones allows for the gathering step at the end that can create better balanced color sizes, as well as the possibility of running at least part of the coloring in parallel. The fact that in the example we use seven instead of the minimal number of four colors is not important to us.

Figure 6 shows examples of colored meshes. The left image shows a cutout of the 2d mesh used in Section 6. The domain is a two-dimensional shell discretized using 786,432 cells. Colorization produces six colors of respectively 131,030, 131,070, 130,988, 131,201, 131,112, and 131,031 cells. The right image shows a colorization of an adaptively refined mesh with 3,860 cells (using the step-6 tutorial program of DEAL.II). The colorization yields 14 colors of respectively 275, 274, 277, 278, 274, 264, 273, 265, 301, 302, 272, 267, 267, and 271 cells. Further adaptive refinement results in 16,604 cells and requires 18 colors of respectively 927, 937, 906, 937, 925, 931, 915, 924, 927, 937, 929, 936, 929, 909, 932, 906, 905, and 892 cells. In all of these cases, we can see that the colors are well balanced in size.

ACKNOWLEDGMENTS

The authors would like to thank Timo Heister for invaluable help in finding bugs in our implementation, in particular regarding the need to keep a list of thread-local scratch objects discussed at the end of Section 5.2.

REFERENCES

G. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. *A FIPS Conference Proceedings* 30 (1967), 483–485.

Krzysztof Banaś. 2004. A modular design for parallel adaptive finite element computational kernels. In *Computational Science – ICCS 2004*, Marian Bubak, GeertDick van Albada, Peter M. A. Sloot, and Jack Dongarra (Eds.). Lecture Notes in Computer Science, Vol. 3037. Springer, 155–162. DOI:http://dx.doi.org/10.1007/978-3-540-24687-9_20

W. Bangerth, R. Hartmann, and G. Kanschat. 2007. deal.II – A general purpose object oriented finite element library. *ACM Transactions on Mathematical Software* 33, 4 (2007), 24/1–24/27.

Andrew C. Bauer and Abani K. Patra. 2004. Robust and efficient domain decomposition preconditioners for adaptive *hp* finite element approximations of linear elasticity with and without discontinuous coefficients. *International Journal for Numerical Methods in Engineering* 59 (2004), 337–364.

B. Bergen, F. Hülsemann, and U. Rüde. 2005. Is 1.7×10^{10} unknowns the largest finite element system that can be solved today? In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'05)*. 5:1–5:14. DOI:http://dx.doi.org/10.1109/SC.2005.38

P. Berger, P. Brouaye, and J. C. Syre. 1982. A mesh coloring method for efficient MIMD processing in finite element problems. In *Proceedings of the International Conference on Parallelism (ICPP'82)* August 24–27. Bellaire, Michigan, 41–46.

Daniel Brélaz. 1979. New methods to color the vertices of a graph. *Communication of the ACM* 22, 4 (April 1979), 251–256.

G. F. Carey, E. Barragy, R. McLay, and M. Sharma. 1988. Element-by-element vector and parallel computations. *Communications in Applied Numerical Methods* 4 (1988), 299–307.

Cris Cecka, Adrian J. Lew, and Eric Darve. 2011. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering* 85, 5 (2011), 640–669.

T. Y. P. Chang, J. A. Hulzen, and P. S. Wang. 1984. Code generation and optimization for finite element analysis. In *EUROSAM 84*, John Fitch (Ed.). Lecture Notes in Computer Science, Vol. 174. Springer, 237–247. DOI:http://dx.doi.org/10.1007/BFb0032846

Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, Yuan Yuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. 2006. Map-reduce for machine learning on multicore. In *Proceedings of Neural Information Processing Systems Conference (NIPS)*. Vancouver, Canada.

Andrew Corrigan, Fernando F. Camelli, Rainald Löhner, and John Wallin. 2011. Running unstructured grid-based CFD solvers on modern graphics hardware. *International Journal for Numerical Methods in Fluids* 66, 2 (2011), 221–229. DOI:http://dx.doi.org/10.1002/fld.2254

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* 51, 1 (2008), 107–113. DOI:http://dx.doi.org/10.1145/1327452.1327492

Karen D. Devine and Joseph E. Flaherty. 1996. Parallel adaptive *hp*-refinement techniques for conservation laws. *Applied Numerical Methods.* 20 (1996), 367–386.

Charbel Farhat. 1988. A simple and efficient automatic fem domain decomposer. *Computers & Structures* 28, 5 (1988), 579–602. DOI:http://dx.doi.org/10.1016/0045-7949(88)90004-1

C. Farhat and L. Crivelli. 1989. A general approach to nonlinear finite-element computations on shared-memory multiprocessors. *Computer Methods in Applied Mechanics and Engineering* 72, 2 (1989), 153–171.

Mike Folk, Albert Cheng, and Kim Yates. 1999. HDF5: A file format and I/O library for high performance computing applications. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'99)*.

J. Frohne, T. Heister, and W. Bangerth. 2015. Efficient numerical methods for the large-scale, parallel solution of elastoplastic contact problems. *Submitted* (2015).

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

M. W. Gee, C. M. Siefert, J. J. Hu, R. S. Tuminaro, and M. G. Sala. 2006. *ML 5.0 Smoothed Aggregation User's Guide*. Technical Report SAND2006-2649. Sandia National Laboratories.

Alan George. 1971. *Computer Implementation of the Finite Element Method*. Technical Report STAN-CS-7J-208. Stanford University.

Alan George. 1973. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis* 10, 2 (1973), 345–363.

Konstantinos M. Giannoutakis and George A. Gravvanis. 2009. Design and implementation of parallel approximate inverse classes using OpenMP. *Concurrency and Computation Practice E.* 21, 2 (2009), 115–131. DOI:http://dx.doi.org/10.1002/cpe.1324

D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, and S. Turek. 2008. Using GPUs to improve multigrid solver performance on a cluster. *International Journal of Computational Science and Engineering* 4, 1 (2008), 36–55. DOI:http://dx.doi.org/10.1504/IJCSE.2008.021111

J. L. Henning. 2007. Performance counters and development of SPEC CPU2006. *ACM SIGARCH Computer Architecture News* 35 (2007), 118–121.

M. A. Heroux and others. 2014. Trilinos web page. Retrieved from http://trilinos.sandia.gov.

M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. 2005. An overview of the Trilinos project. *ACM Transactions on Mathematical Software* 31 (2005), 397–423.

Grand Roman Joldes, Adam Wittek, and Karol Miller. 2010. Real-time nonlinear finite element computations on GPU application to neurosurgical simulation. *Computer Methods in Applied Mechanics and Engineering* 199, 49–52 (2010), 3305–3314. DOI:http://dx.doi.org/10.1016/j.cma.2010.06.037

A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven. 2009. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics* 228, 21 (2009), 7863–7882.

Matthew G. Knepley and Andy R. Terrel. 2013. Finite element integration on GPUs. *ACM Transactions on Mathematical Software* 39, 2 (2013), 10:1–10:13.

D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michéa. 2010. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. *Journal of Computational Physics* 229 (2010), 7692–7714. DOI:http://dx.doi.org/10.1016/j.jcp.2010.06.024

D. Komatitsch, D. Michéa, and G. Erlebacher. 2009. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *Journal of Parallel and Distributed Computing* 69, 5 (2009), 451–460. DOI:http://dx.doi.org/10.1016/j.jpdc.2009.01.006

K. Kormann and M. Kronbichler. 2011. Parallel finite element operator application: Graph partitioning and coloring. In *Proceedings of the 7th IEEE International Conference on eScience*. 332–339. DOI:http://dx.doi.org/10.1109/eScience.2011.53

M. Kronbichler, W. Bangerth, and T. Heister. 2013. *The deal.II Tutorial: Step-32*. http://www.dealii.org/developer/doxygen/deal.II/step_32.html.

M. Kronbichler, T. Heister, and W. Bangerth. 2012. High accuracy mantle convection simulation through modern numerical methods. *Geophysics Journal International* 191 (2012), 12–29.

M. Kronbichler and K. Kormann. 2012. A generic interface for parallel cell-based finite element operator application. *Computers & Fluids* 63 (2012), 135–147.

Marek Kubale. 2004. *Graph Colorings*. American Mathematical Society.

Andras Laszloffy, Jingping Long, and Abani K. Patra. 2000. Simple data management, scheduling and solution strategies for managing the irregularities in parallel adaptive *hp* finite element simulations. *Parallel Computing* 26 (2000), 1765–1788.

Kincho H. Law. 1986. A parallel finite element solution method. *Computers & Structures* 23, 6 (1986), 845–858. DOI:http://dx.doi.org/10.1016/0045-7949(86)90254-3

Jimmy Lin and Chris Dyer. 2010. *Data-Intensive Text Processsing with MapReduce*. Morgan & Claypool.

Anders Logg, Kent-Andre Mardal, and Garth Wells. 2012. *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Lecture Notes in Computational Science and Engineering, Vol. 84. Springer.

Rainald Löhner and Joseph D. Baum. 2013. Handling tens of thousands of cores with industrial/legacy codes: Approaches, implementation and timings. *Computers & Fluids* 85 (2013), 53–62. DOI:http://dx.doi.org/10.1016/j.compfluid.2012.09.030

Rainald Löhner and Martin Galle. 2002. Minimization of indirect addressing for edge-based field solvers. *Communications in Numerical Methods in Engineering* 18, 5 (2002), 335–343. DOI:http://dx.doi.org/10.1002/cnm.494

G. Mahinthakumar and F. Saied. 2002. A hyrbid MPI-OpenMP implementation of an implicit finite-element code on parallel architectures. *International Journal of High Performance Computing* 16, 4 (2002), 371–393.

G. R. Markall, A. Slemmer, D. A. Ham, P. H. J. Kelly, C. D. Cantwell, and S. J. Sherwin. 2013. Finite element assembly strategies on multi-core and many-core architectures. *International Journal for Numerical Methods in Fluids* 71, 1 (2013), 80–97. DOI:http://dx.doi.org/10.1002/fld.3648

Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. 2004. *Patterns for Parallel Programming*. Addison-Wesley.

Francis Thomas McKenna. 1997. Object-oriented finite element programming: frameworks for analysis, algorithms and parallel computing. (1997). Ph. D. Thesis, University of California.

J. M. Melenk, K. Gerdes, and C. Schwab. 2001. Fully discrete *hp*-finite elements: Fast quadrature. *Computer Methods in Applied Mechanics and Engineering* 190 (2001), 4339–4364.

Message Passing Interface Forum. 2012. *MPI: A Message-Passing Interface Standard (Version 3.0)*. Technical Report. http://www.mpi-forum.org/.

Kengo Nakajima. 2003. OpenMP/MPI hybrid vs. flat MPI on the earth simulator: Parallel iterative solvers for finite element method. In *High Performance Computing*. Lecture Notes in Computer Science, Vol. 2858. Springer, 486–499.

Kengo Nakajima. 2005. Parallel iterative solvers for finite-element methods using an OpenMP/MPI hybrid programming model on the earth simulator. *Parallel Computing* 31, 10–12 (2005), 1048–1065. DOI:http://dx.doi.org/10.1016/j.parco.2005.03.011

L. Oliker, X. Li, P. Husbands, and R. Biswas. 2002. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Review* 44, 3 (2002), 373–393.

OpenACC. 2014. OpenACC web page. Retrieved from http://www.openacc.org.

OpenMP. 2014. OpenMP Architecture Review Board. Retrieved from http://www.openmp.org.

O. Pantalé. 2005. Parallelization of an object-oriented FEM dynamics code: Influence of the strategies on the Speedup. *Advances in Engineering Software* 36 (2005), 361–373.

M. Paszyński and L. Demkowicz. 2006. Parallel, fully automatic *hp*-adaptive 3D finite element package. *Engineering with Computers* 22, 3–4 (2006), 255–276. DOI:http://dx.doi.org/10.1007/s00366-006-0036-8

M. Paszyński, J. Kurtz, and L. Demkowicz. 2006. Parallel, fully automatic *hp*-adaptive 2d finite element package. *Computer Methods in Applied Mechanics and Engineering* 195 (2006), 711–741.

Maciej Paszyński, David Pardo, Carlos Torres-Verdín, Leszek Demkowicz, and Victor Calo. 2010. A parallel direct solver for the self-adaptive *hp* finite element method. *Journal of Parallel and Distributed Computing* 70, 3 (2010), 270–281. DOI:http://dx.doi.org/10.1016/j.jpdc.2009.09.007

D. A. Patterson and J. L. Hennessy. 2009. *Computer Organization and Design* (4th ed.). Morgan Kaufmann, Burlington.

Juan C. Pichel, Francisco F. Rivera, Marcos Fernández, and Aurelio Rodríguez. 2012. Optimization of sparse matrix–Vector multiplication using reordering techniques on GPUs. *Microprocessors and Microsystems* 36, 2 (2012), 65–77. DOI:http://dx.doi.org/10.1016/j.micpro.2011.05.005

J. Reinders. 2007. *Intel Threading Building Blocks*. O'Reilly.

Jean-François Remacle, Ottmar Klaas, Joseph E. Flaherty, and Mark S. Shephard. 2002. Parallel algorithm oriented mesh database. *Engineering with Computers* 18, 3 (2002), 274–284. DOI:http://dx.doi.org/10.1007/s003660200024

Francis P. Russell and Paul H. J. Kelly. 2013. Optimized code generation for finite element local assembly using symbolic manipulation. *ACM Transactions on Mathematical Software* 39, 4 (2013), 26:1–26:29. http://dx.doi.org/10.1145/2491491.2491496

Y. Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2nd ed.). SIAM, Philadelphia.

W. Schroeder, K. Martin, and B. Lorensen. 2006. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics* (3rd ed.). Kitware, Inc.

D. Silvester and A. Wathen. 1994. Fast iterative solution of stabilised stokes systems. Part II: Using general block preconditioners. *SIAM Journal of Numerical Analysis* 31 (1994), 1352–1367.

T. Tezduyar, S. Aliabadi, M. Behr, A. Johnson, and S. Mittal. 1993. Parallel finite-element computations of 3d flows. *Computer* 26 (1993), 27–36. DOI:http://dx.doi.org/10.1109/2.237441

R. Tuminaro and C. Tong. 2000. Parallel smoothed aggregation multigrid: Aggregation strategies on massively parallel machines. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'00)*, J. Donnelley (Ed.).

R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure. 2010. On the limits of GPU acceleration. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism (HotPar'10)*. USENIX Association, Berkeley, CA, USA, 13–19. http://dl.acm.org/citation.cfm?id=1863086.1863099

E. Wadbro and M. Berggren. 2009. Megapixel topology optimization on a graphics processing unit. *SIAM Review* 51 (2009), 707–721. Issue 4.

J. White and R. I. Borja. 2011. Block-preconditioned Newton-Krylov solvers for fully coupled flow and geomechanics. *Computers & Geosciences* 15 (2011), 647–659.

S. Williams, N. Bell, J. Choi, M. Garland, L. Oliker, and R. Vuduc. 2010. Sparse matrix vector multiplication on multicore and accelerator systems. In *Scientific Computing with Multicore Processors and Accelerators*, J. Dongarra, D. A. Bader, and J. Kurzak (Eds.). CRC Press.

S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. W. Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the High Performance Computing, Networking, and Storage Conference* (SC 2007). 10–16.

G. Yagawa and R. Shioya. 1993. Parallel finite elements on a massively parallel computer with domain decomposition. *Computing Systems in Science and Engineering* 4, 4–6 (1993), 495–503. DOI:http://dx.doi.org/10.1016/0956-0521(93)90017-Q

Received December 2013; revised November 2015; accepted November 2015