

SUPERGLUE: A SHARED MEMORY FRAMEWORK USING DATA VERSIONING FOR DEPENDENCY-AWARE TASK-BASED PARALLELIZATION*

MARTIN TILLENIUS[†]

Abstract.

In computational science, making efficient use of modern multicore based computer hardware is necessary in order to deal with complex real-life application problems. However, with increased hardware complexity, the cost in man hours of writing and re-writing software to adapt to evolving computer systems is becoming prohibitive. Task based parallel programming models aim to allow the application programmers to focus on the algorithms and applications, while the performance is handled by a runtime system that schedules the tasks onto nodes, cores, and accelerators. In this paper we describe a task parallel programming model where dependencies are represented through data versioning. The benefits of using this type of model are that it is easy to represent different types of dependencies and that scheduling decisions can be made locally. We implement this as a freely available C++ header-only template library, and show experimental results indicating that our implementation scales well, and performs well in comparison to other related efforts.

Key words. task parallel, data version, dependency, shared memory

AMS subject classifications. 65Y05 65Y10

1. Background and related work. Modern processors for laptop, desktop, and server computers have several computational cores. In order to write efficient software for such processors, software needs to be parallel. Since writing parallel software is known to be difficult and error-prone, it is desirable that the parallelization-specific parts are separated from the rest of the software.

We identify the application-independent parts of a parallel software that can be implemented in a library without too many assumptions or restrictions to be the management of dependencies between different computations, and the mapping of computations to hardware resources. If these parts are moved into a library, it is well worth the effort to carefully design such a library to be efficient, and to test it thoroughly. The library can then be reused in many different applications.

By moving the dependency management and scheduling into a library that exposes a convenient and expressive interface for specifying dependencies, the development of parallel software becomes easier, faster, less error-prone, and is likely to result in more efficient software.

1.1. Dependencies and synchronization. The most common way to write shared-memory parallel software is to parallelize for-loops using OpenMP [6]. While this works well for many applications, it enforces a fork-join structure where the software is divided up into parallel sections that end with a barrier where all threads are synchronized again. These barriers scale poorly as the number of cores increases, and can reduce the performance substantially. To achieve higher performance, synchronization between threads needs to be more fine-grained and reduced to a minimum. To fulfill this, we select a task-based programming model where tasks may have dependencies between them.

*The work was supported by the Swedish Research Council. The work was carried out within the Linnaeus centre of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center.

[†]Uppsala University, Department of Information Technology, Box 337, SE-751 05 Uppsala, Sweden. (martin.tillenius@it.uu.se)

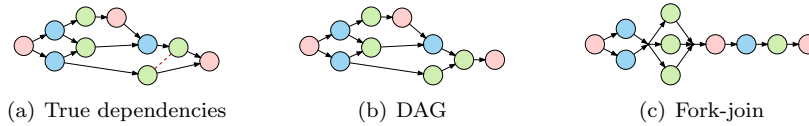


FIG. 1.1. *Examples of how the dependencies in an application can be described. Circles denote tasks and edges denote dependencies. The dashed line in (a) means that the tasks it connects must not execute concurrently.*

A solution to the shortcomings of the fork-join model is to build a directed acyclic graph (DAG) between the tasks. This allows dependencies that are not possible to express in a fork-join model, and reduces unnecessary synchronization points. However, not all kinds of dependencies can be represented by a DAG. An edge means that one task must complete before the next is started. In the case of reduction operations, such as accumulating partial results in a shared variable, the tasks may run in any order but must not run concurrently. To describe this kind of dependency in a DAG, the order of the reduction tasks must either be fixed, which limits the scheduler, the graph must be rewritten at runtime, which is expensive, or these dependencies must be left out and handled by some external construct. That is, a static DAG alone cannot describe the true dependencies.

Figure 1.1 shows the true dependencies of an application and examples of how the dependencies can be expressed in different models. Two of the tasks in this application can run in any order but not at the same time, which is indicated in subfigure (a) by a red dashed line connecting them. In the DAG description in subfigure (b), an artificial must-execute-before dependency has been introduced between the tasks instead, and the order in which they must execute has been fixed. In the fork-join model in subfigure (c), two artificial barriers have been introduced.

In applications where parallelism is scarce, introducing artificial dependencies and unnecessary synchronization may be devastating for performance. A goal of this work is to never overestimate dependencies.

1.2. Scheduling. In our system, we consider all processor cores to be identical. We want to maximize throughput, and do not consider fairness or task deadlines. Since some applications need to be able to create tasks dynamically at runtime, dependency management and task scheduling must happen at runtime. This means that there is very limited information to use for scheduling; the set of tasks to schedule is not known in advance, and the lengths of the tasks are not known either. Also, tasks cannot be preempted. Once the dependencies are met, the only objective we schedule for is locality. That is, tasks that access the same data should preferably execute on the same core, in order to reuse data already present in the caches of that core.

Among the most important properties of the scheduler is that it is fast. In some applications, it is hard to extract parallelism without making the tasks very small. If the time spent in the runtime system between tasks is not negligible compared with the time spent in the tasks, this will limit scalability and simply prevent applications with fine grained tasks from scaling at all. This means that the overhead and the time spent on scheduling must be minimized.

The task scheduling should also scale to large numbers of cores. For this to be possible, scheduling should avoid communication between cores and no central information should be required for scheduling. Scheduling decisions should as far as possible be made locally on each thread.

1.3. Contributions. In this paper, we describe an alternative and flexible way to represent dependencies, using data versions instead of dependency graphs. Tasks in this model depend only on data, but not on other tasks. Tasks are queued at the data they need, which avoids central synchronization, and leads to locality driven scheduling.

The way to represent dependencies was developed in [22], and the system was previously described also in [21]. In this paper, we further develop the programming model, present new experiments with improved performance, and describe the programming model in more detail. We also include experiments comparing our solution to other similar systems, where we show that our solution has less overhead than all the other evaluated systems. Our implementation is open source and freely available at GitHub¹.

1.4. Related work. The question of how to efficiently take advantage of multicore architectures has been investigated in the setting of linear algebra, where performance is key. Task-based, dynamically scheduled approaches are concluded to be well suited for multicore processors in [14] and [5].

There are several different task-based programming models. The main difference between them is how dependencies are managed. Either all spawned tasks are immediately allowed to execute, or there is some way to specify dependencies that must be satisfied before the tasks can execute.

One of the best known task-based programming languages is Cilk [4] (now Intel Cilk Plus [12]). This is a language extension to C (and to C++ in Cilk++ [17] and Intel Cilk Plus) which introduces the `spawn` keyword to generate new tasks, which are immediately allowed to run, and the `sync` keyword, which blocks until all tasks spawned from the current task are finished. This provides the fork-join programming model, which may cause unnecessary synchronization.

The most common programming model for shared memory parallelism is OpenMP. Tasks were introduced into the OpenMP standard with OpenMP 3.0 [2], following the same task model as Cilk. Sections annotated with `#pragma omp task` are spawned as tasks, and are immediately allowed to run while the control flow continues after the task. To synchronize tasks, there is the `#pragma omp taskwait` directive, which blocks until all tasks spawned by the current task are finished. OpenMP also provides parallel sections and parallel for loops. Common for these constructs is that they provide the fork-join model. In OpenMP 4.0 [19], more general dependencies have been introduced through the `depend` clause.

The shortcomings of the fork-join model is pointed out in [14]. In this paper, which deals with the implementation of linear algebra algorithms on multicore processors, it is observed that calling parallel subroutines from a sequential algorithm, which causes a fork-join structure, limits scalability. When the algorithm is implemented as an explicitly parallel code that call sequential subroutines, the number of synchronization points is reduced, and the performance is improved. Different orders of executing the computations are also considered, comparing static and dynamic scheduling techniques, and it is concluded that the best performance is reached when the order of the computations is decided at runtime.

In [5], it was identified that high degree of thread level parallelism requires algorithms to have fine granularity and to be asynchronous. Operations should be split into tasks that work on data sets small enough to fit in the cache, and these tasks

¹<https://github.com/tillenius/superglue>

should be scheduled dynamically based on dependencies between them.

Intel Threading Building Blocks (Intel TBB) [13] is a C++ library for shared memory parallelism. It includes a task scheduler which besides the fork-join model allows dependencies between tasks to be specified explicitly, and arbitrary DAGs of tasks to be built.

A problem with explicitly building a task dependency graph is that it may be demanding for the programmer. In order to realize all dependencies, the programmer must know all data that a task reads and writes, since this will cause dependencies. When dependency graphs are explicitly built, the programmer must also know what other tasks access the same data and declare dependencies between these tasks. This makes building the graphs difficult and error prone. An improvement of this is to let the programmer annotate each task with which data it access. The problem of keeping track of which other tasks access the same data is then left to a runtime system, and the programmer is relieved of this responsibility.

A programming model where task dependencies are deduced from data accesses is the StarSs model. This programming model was introduced in CellSs [3], a system that targets the Cell BE processor architecture, and was applied for general multicore architectures in the SMP Superscalar (SMPs) system [20]. The StarSs programming model is now represented by OmpSs [8], which succeeds SMPs and adds support for heterogeneous architectures. In the StarSs programming model, the task declaration clause in OpenMP is extended with `in`, `out`, and `inout` clauses for declaring data-dependencies. Using these clauses, the programmer specifies which data each task accesses, and how the data is accessed (read, write, or both). OmpSs then uses this information to build a task dependency graph at runtime. The idea to deduce task dependencies from annotations of data accesses is not unique for the OmpSs programming model, but has been used earlier in for instance the Jade programming language [16].

StarPU [1] is a C library for task-based programming, targeting heterogeneous architectures. It is capable of scheduling tasks over (for instance) both CPUs and GPUs. It also manages data transfers and supports distributed memory parallelism. StarPU provides both the possibility to state dependencies between tasks explicitly, and to state them implicitly through data dependencies. StarPU also includes a GCC plugin, which allows tasks to be annotated using pragma directives.

X-Kaapi [10] is a C++ library for task-based programming, with dependencies computed at runtime from memory access annotations. The programming model is borrowed from its successor Athapascan-1 [9]. X-Kaapi includes a compiler that introduces pragma directives which can be used for specifying tasks, as an alternative to the C++ interface.

The Swan programming language [25] is an extension of Cilk which adds the possibility to annotate tasks with data access information, in order to introduce task dependencies. Swan uses an independently developed versioning system similar to the scheme we introduce here.

The state of the art in linear algebra on multicore architectures is the PLASMA package [7], which uses a C library for task-based programming with data dependencies called Quark [26].

1.5. Paper organization. The next section briefly introduces other related efforts. We describe our solution in section 2 and the details in section 3. We highlight the flexibility of our solution in section 4, where we show some features, usage examples, and how to customize the library to the user’s needs. To evaluate the performance

of our implementation, we have conducted a number of experiments. We show the results of some micro-benchmarks in section 5, present more realistic applications in section 6, and compare our implementation against other task-based frameworks in section 7.

2. Our programming model. Our programming model contains two fundamental concepts; handles and tasks. Handles are objects that represent some shared resource for which accesses should be managed. The most common shared resource is data, but handles can be used to represent anything. A handle may represent network communication or file access, and be used to serialize or to order the use of such resources.

Tasks are pieces of the program logic, together with the data needed for its execution. In practice, it is an object that inherits a `Task` class provided by SuperGlue, and implements a method that is called by SuperGlue to execute the task.

The programmer creates handle objects to protect shared data, and writes tasks which operate on this data to perform the logic of the software. The software is then expressed as creating and submitting these tasks to the runtime system. For each task, the programmer must specify which handles are required, and what type of access the task performs, such as if the protected data is only read, or if it is modified. From the order the tasks are submitted, or rather from the order the handles are accessed, together with the access type, the runtime system deduces which tasks can be executed in parallel, and maps the tasks to the available cores in the system.

2.1. Handles. One way to think of handles is that they are similar to ordinary locks in the way that it is up to the programmer to decide what they protect, and that their meaning is unknown to the runtime system. The programmer must correctly specify all handles a task accesses, or the program will be incorrect. The runtime system cannot help with this.

In contrast to programming with locks, it is not possible to have deadlocks, and rather than blocking and waiting for locks, the runtime system uses the handles to decide an order in which access is granted and tasks are executed.

In our model, accesses are registered to handles, not to memory addresses. There are several reasons for this choice. First of all, the memory address is not necessarily a unique identifier. Different tasks can have pointers to the same memory block but update different parts of the block. An example of this is reported in [15], where several tasks of a matrix decomposition algorithm access the same matrix block, but some only touches the lower triangular part and others only access the upper triangular part. Another reason is that when working with memory addresses, the user could expect the runtime system to detect that synchronization is needed between tasks that access the same data through different pointers. This could also be supported, but at an additional cost at runtime. With handles, it is more clear that the library cannot find out if two handles represent overlapping resources. A third reason is that the handles keep the required book-keeping data, so by requiring the user to provide the handles directly, we do not have to map the address to a book-keeping object. The possibility to use memory addresses to represent data, and then map addresses to book-keeping objects is still available, and can be added as a layer upon our solution.

The granularity of the handles is connected to the granularity of the tasks, and is an important factor for performance. If tasks are too large, or similarly, handles too coarse, there will be fewer tasks to run in parallel, and less parallelism. If tasks are too small, too much time will instead be spent on task and dependency management. The granularity of the tasks depends strongly on the granularity of the handles. There

needs to be enough handles for tasks to work on in parallel, but too many handles occupies more memory, and leads to too small tasks.

The granularity should be large enough so that the time spent on task management is negligible. In Section 5 we perform experiments with different task sizes, which gives a hint of how large tasks need to be. SuperGlue does not help the programmer with making these decisions.

The lifetime of a handle naturally needs to be longer than all tasks that access it. Handles can be created dynamically as needed, and can be deleted either when it is certain that all tasks that access it have finished, such as after a global barrier, in a task that does not access the handle but is guaranteed to execute after the last task that does, or in the destructor of the last task that access the handle.

2.2. Tasks. The programmer divides the software into a number of **tasks**, and declares which handles each task requires. A task is a piece of the program logic together with the data needed for its execution. In practice, it is an object that inherits a **task** class provided by SuperGlue, with a callback method that is called to execute the task. When a task is constructed, the data that is required for the task, typically pointers to the data it works on, are stored in the task object, and accesses are registered to the required handles. Tasks are executed as a whole, and cannot be suspended. They are therefore expected not to block.

2.3. Access types. To allow concurrent access when desired, and unique access when required, the programmer must specify an access type for each handle a task accesses. The default access types are **read**, **write**, and a reduction type called **add**, but it is possible to customize these, as will be described in Section 4.

The read access type means that the task must wait for all previous write or add accesses to finish, but that several tasks may read the same handle concurrently.

A write access means that the task must wait for all previous accesses to finish. The task is then allowed to both read and write to the data. There is no access type to describe a write-only access.

The add access type means that a task must wait for all previous read and write accesses to finish. An add access can be reordered with other add accesses, but two add accesses to the same handle cannot run concurrently. The typical use for this type is to accumulate results together. The tasks do not actually need to add the results together. The access type can be used for any tasks that perform an associative and commutative operation, or any other operation where reordering is accepted. The name add is selected because it is the common case, and a simple name.

```

1 #include "superglue.hpp"
2 struct Options : public DefaultOptions<Options> {};
3 struct MatrixBlock {
4     Handle<Options> handle;
5     double *data;
6 };
7
8 void cholesky(size_t dim, MatrixBlock *A) {
9     SuperGlue<Options> sg;
10    for (int k = 0; k < dim; ++k) {
11        sg.submit(new potrf(A[k*dim+k])); // Cholesky factorization
12        // Panel update
13        for (int m = k+1; m < dim; ++m)
14            sg.submit(new trsm(A[k*dim+k], A[m*dim+k]));
15        // Update trailing matrix
16        for (int m=k+1; m < dim; ++m) {
17            for (int n=k+1; n < m; ++n)
18                sg.submit(new gemm(A[m*dim+k], A[n*dim+k], A[m*dim+n] ));
19            sg.submit(new syrks(A[m*dim+k], A[m*dim+m]));
20    } } }
```

```

21 int main() {
22     const size_t dim = 10;
23     MatrixBlock *A = new MatrixBlock[dim * dim];
24     // initialize A with data here
25     cholesky(dim, A);
26     return 0;
27 }

```

LISTING 1

Example SuperGlue code for a tiled Cholesky factorization

2.4. Example. Listing 1 shows an example of an application that uses SuperGlue. It performs a tiled Cholesky factorization to decompose a symmetric positive definite matrix A into a lower triangular matrix L such that $A = LL^T$. The tiled Cholesky factorization is a classic example of an algorithm with nontrivial dependencies, but the details of the algorithm are not important here. By creating a **SuperGlue** object, as on line 9, the SuperGlue runtime system is initialized, and worker threads are created so that there is one thread per available processor core. SuperGlue can be configured to meet the application needs, as will be explained in Section 4. This configuration is done by specifying types in an **Options** struct, which most other SuperGlue classes take as a template parameter. The options struct is defined in line 2, where it in this case only inherits and uses the default options.

The algorithm works on tiles of a large matrix. To manage dependencies between the different calls to BLAS kernels, a handle is associated with each matrix tile by adding a handle to the struct that represents a tile of the matrix, as shown in line 3. Tasks are submitted to the runtime system using the `submit` method, as shown for instance at line 11.

```

1 // Task to perform a general matrix multiply
2 struct gemm : public Task<Options> {
3     double *a, *b, *c;
4     gemm(MatrixBlock &ba, MatrixBlock &bb, MatrixBlock &bc)
5     : a(ba.data), b(bb.data), c(bc.data) // Store parameters needed by the task
6     {
7         // Register accesses
8         depend(ReadWriteAdd::read, ba.handle);
9         depend(ReadWriteAdd::read, bb.handle);
10        depend(ReadWriteAdd::write, bc.handle);
11    }
12    void run() {
13        // Task body (call to BLAS in this case)
14        dgemm('N', 'T', DIM, DIM, DIM, -1.0, a, DIM, b, DIM, 1.0, c, DIM);
15    } };

```

LISTING 2

Example SuperGlue task

An example task definition is shown in Listing 2. This task performs a matrix-matrix multiplication by calling BLAS. It takes two input matrix tiles `ba` and `bb`, and adds their product into the output matrix tile `bc`. A task in SuperGlue inherits from a **Task** class, as in line 2. The constructor (line 4–11) copies the data needed for the task into the class (line 5), and declares which handles the task accesses. The tasks read from `ba` and `bb`, so read accesses are registered on line 8 and 9, and a write access to `bc` is registered at line 10.

When all the dependencies of the task are met, the runtime system will call the `run` method of the task (line 12), which will invoke BLAS to perform the actual matrix multiplication.

2.5. Data-centric view. Central to our programming model is the data-centric view. In our model, tasks can only depend on data, not on other tasks. One way to

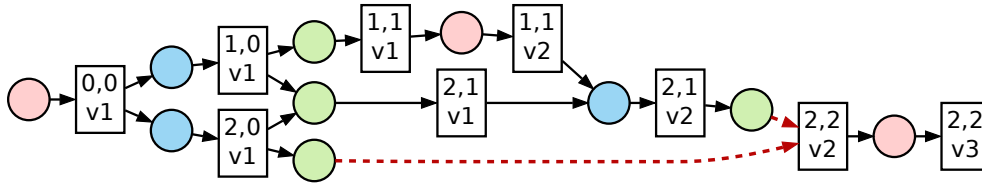


FIG. 2.1. Data dependency graph. Circles are tasks, edges are dependencies, and boxes are handles. The same handle is represented by several boxes, one for each version. The first line in each box is the name of a handle, and the second line is the handle version. The dashed edges means that tasks can access the handle in any order, but not concurrently.

illustrate dependencies in this data-centric view is shown in Figure 2.1. This figure shows the same dependencies as Figure 1.1, but this graph also includes the data.

This view simplifies the implementation. The straight-forward way for a runtime system to insert a new task into a dependency graph using information about which memory addresses the task accesses is as follows:

1. Look up the memory address in a map, to find the book-keeping object
2. Lock the book-keeping object to avoid data races
3. Check which preceding tasks to wait for from the book-keeping object
4. For each preceding task:
 - Lock the preceding task, and add a dependency edge to it
5. Update the book-keeping object with the access from the new task

In our model, we avoid looking up the book-keeping object, and instead require that the user provide the correct handle. By having tasks depend on data instead of on other tasks, we avoid synchronization between tasks, which are transient in nature. Data objects that are shared between the tasks are required to be alive until all tasks are finished, while tasks can be submitted, executed, and deleted at any point. To submit a new task in our model, the following steps are needed:

1. Lock the handle to avoid data races
2. Update the handle with the access from the new task

The actual update is simple. All that needs to be done is to increment one or two integer variables, depending on the access type.

Not only does the data-centric view simplify implementation, it also improves flexibility. In our model, it is possible to register future accesses to a handle. This way it is possible to submit a task that waits for a task that will be submitted some time in the future, in a natural and straight-forward way.

3. The runtime system. The runtime system has one worker thread per core, and each worker thread has its own queue of ready tasks. The worker threads execute tasks from the front of the ready queue. If a task submits new tasks, these are put in the front of the ready queue of the thread where the task is running. If a worker runs out of tasks, it tries to steal tasks the from the end of another thread's ready queue, picked at random.

In addition to the ready queues at each worker, there is also a task queue at each handle, for tasks that have an unfulfilled dependency on that handle. This leads to large number of task queues, and reduces contention on task queues to a minimum.

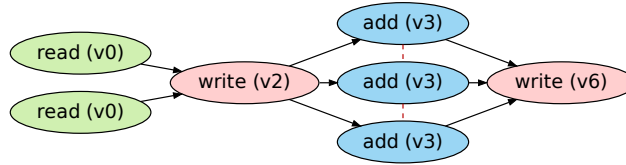


FIG. 3.1. Example to illustrate how dependencies are represented by versioning. Nodes are tasks, the label and color specifies the access type, and the required version number is given in the parenthesis. All tasks access the same handle. The dashed lines between the add tasks denote that they must not run at the same time.

3.1. Dependency management. Dependencies are managed by attaching a version number to each handle, and having the tasks require certain versions of the handles it accesses. The version number works as a counter that keeps track of how many tasks have accessed the handle, and is increased after each access to it. Note that the version number is increased also after read-only accesses. This is needed to detect if all previous reads are finished, before a write can be allowed.

When a task is submitted to the runtime system, it will be assigned a required version number for each handle, which determines when it will be able to execute. The required version of a handle is available when its current version is equal to or greater than the required version. A task is ready to execute when the required versions of all handles it accesses are available.

If a task requires a version of a handle that is not yet available, the task will be put in a queue at that handle. When a worker thread has finished a task, it will increase the versions of the accessed handles, and wake up the tasks that waited for the newly available version.

Figure 3.1 illustrates how dependencies are represented. In this example, all tasks access a single handle. The two first tasks only read from the handle, and can run at the same time. They will both require version 0 of the handle. The next task is a write, and it must wait for the two previous read tasks to finish, so it will require version 2 of the handle. This write is followed by three add tasks, which can run in any order, but not at the same time. They will all require version 3, which means they have to wait for all three previously submitted tasks that access this handle to finish. To ensure that only one of them accesses the handle at a time, the tasks must grab a lock on the handle. If the lock is already taken, the task is queued at the handle, and the worker thread is free to execute other tasks. The last task is a write, and must wait for all previously submitted tasks to finish, and thus requires version 6 of the data. It is worth to point out that we do not keep several copies of the data that a handle represents. This might be desirable in some cases, and is discussed further in Section 4.

The dependency checking is easily implemented. To see if a task is ready to execute, it is enough to compare the required version with the version of the handle. Which version a task should require depends on the access type. If the previous access was of the same access type, and that type allows reordering or concurrent execution, the same version is required. Otherwise, the required version is one more than the total number of accesses registered so far.

It is enough to save the access type, the current version, and the total number of accesses for this book-keeping. For the default configuration, with read, write, and add as the possible access types, it suffices to store only two versions; the version to

require if the next access is a read, and the version to require if the next access is an add. Write accesses need to wait for all previous read and add accesses to finish, and the required version is the next after the maximum of the two stored versions.

```

1 handle::register_access(access_type) {
2     if access_type ≠ last_access_type or not reorderable(access_type)
3         next_version = access_count
4         last_access_type = access_type
5         ++access_count
6     return next_version
7 }

```

LISTING 3

Pseudo-code showing how required versions are deduced from registering accesses

```

1 task::check_if_ready() {
2     for (handle, required_version) in accessed_handles
3         if handle.version < required_version
4             return false
5     return true
6 }

```

LISTING 4

Pseudo-code showing how task dependencies are checked

```

1 task::finish() {
2     for handle in accessed_handles
3         ++handle.version
4 }

```

LISTING 5

Pseudo-code showing how handle versions are updated after a task is finished

To concretize how dependencies are managed through data versioning, Listing 3 shows how a task is assigned a required handle version. This is the most general solution, which requires three variables to store the current state. As mentioned above, it suffices with two variables for the default access types *read*, *write* and *add*, but this is not shown here. Listing 4 shows how tasks are checked for being ready to execute. A task may be checked if it is ready several times, at most once per accessed handle. The actual implementation saves the index of the first handle not available, and continue after that one. This ensures that each handle is checked only once. Listing 5 shows how handle versions are updated after a task is finished.

3.2. Scheduling. The order in which tasks are executed is driven by locality. As mentioned before, a task that cannot run because it requires a version of a handle that is not yet available, is put in a queue at that handle. When a task finishes accessing a handle, it increases the version number of the handle and moves all tasks waiting for the new version of the handle to the front of its ready task queue. In a typical application, most tasks will be queued at the handles they are waiting for. When a worker finishes a task, it will wake up and execute the tasks that wait to access the same data, leading to good locality. Tasks will be executed on the thread where the data they need was most recently used, unless load balancing was needed, and the task was stolen. Tasks that are ready to execute directly when they are submitted are distributed among the workers in a round-robin fashion.

A task with an *add* access requires exclusive access to a handle, but compete with other tasks for executing first. For this, handles have a flag to indicate if some task has been granted exclusive access. When a task needs exclusive access but the lock is held by another thread, the task is enqueued at the handle, just like when dependencies are not met, and woken up and put in the front of the ready queue when the other task is finished.

If *add* accesses are not used, each dependency is checked exactly once. A task is queued in at most as many queues as the number of handles it accesses, before it is moved into a ready queue. Once a task is moved into a ready queue it is guaranteed to be ready to run, and no more checks are needed. From a ready queue it might be stolen at most once, since a successfully stolen task is executed directly. However, when *add* accesses are used, a task can try to acquire exclusive access to a handle any number of times.

There are no global scheduling decisions or global data structures in this scheme. Tasks that are ready to execute when they are submitted will be distributed among the threads, unless they are submitted from within a task directly to the ready queue of the current thread. Except for that, communication between threads only happens when load balancing is needed (stealing), when tasks on different threads access the same data, or when there is a global barrier.

4. Customization and Features. SuperGlue is written as a research tool for experimenting with task-based models. Because of this, several classes can be overridden to change the default behavior, and features can be enabled or disabled to experiment with different solutions. Configuration is done by defining types in an `Options` struct, which is given as a template parameter to all SuperGlue classes. This means that disabled features are disabled at compile time, and cause no memory or runtime overhead. This section is indented to give an idea of what can be customized, and how, by showing a few examples, but the list is not complete and the details are suppressed.

```
1 struct Options : public DefaultOptions<Options> {
2     typedef Disable Stealing;
3     typedef Enable Subtasks;
4 };
```

LISTING 6

Example options struct. This example disables task stealing, and enables the possibility for a task to create subtasks and wait for all subtasks to finish before the task is finished.

Listing 6 shows an example options struct. This struct is used as a template parameter to all SuperGlue classes, which allows settings to be made at compile time. The default options class defines the types `Enable` and `Disable`, which are used to indicate that features should be enabled or disabled. In this example, task stealing is disabled, and the Subtasks feature is activated. The subtask feature enables a task to submit subtasks and wait for all subtasks to finish before the task itself finishes.

```
1 // Extend the default handle class with a data pointer and a size field
2 template <typename Options>
3 struct MyHandle : public HandleBase<Options> {
4     double *data;
5     size_t size;
6 };
7
8 // Configure SuperGlue to use our customized Handle struct
9 struct Options : public DefaultOptions<Options> {
10     typedef MyHandle<Options> HandleType;
11 };
```

LISTING 7

Example extending the `Handle` class to append a data and a size field to all handles.

The `Task` and `Handle` classes can be overridden to be extended with user-defined functionality or data. Listing 7 shows an example where all `Handles` are extended with a data pointer and a size field. This is just an example, and not the encouraged programming style. It might be a better design to keep the original handle class as is, and define an application specific class to contain the data, its size, and the handle.

4.1. Scheduling. It is possible to override the default behavior to affect the scheduling in three places:

- Which task to select for execution from the local ready queue.
- Which task to steal from the ready queue of another thread.
- Which other task queue to try to steal tasks from.

The default behavior is to pick the first task of the local ready queue, and to steal the last task from a foreign ready queue. This can be customized by overriding the default task queue with a custom one where the `pop_front` and `pop_back` operations select tasks using the desired criteria. The task queue class can also define additional data fields (possibly none) that should be included in the task class. This is used to attach a priority field to all tasks, or to implement intrusive linked lists, for example.

The default order in which stealing should be attempted is to pick a queue at random, and then attempt to steal from the queues in order, starting from that one, until a task is successfully stolen, or all task queues have been attempted. This behavior can be customized by implementing a class that defines a `steal` method that attempts to steal tasks from other task queues, and redefining the `StealOrder` type in the options struct to use this new implementation instead of the default one.

Related to scheduling is setting thread affinity. The default behavior is to pin each thread to its own core, in the order the cores are enumerated by the operating system. This can be overridden to implement a different pinning strategy, or to disable pinning entirely, by writing a new class and redefine the type `ThreadAffinity` in the options struct.

4.2. Memory management. Memory management is avoided in SuperGlue whenever possible, and is only done in two places; to free tasks that have finished, and in the container used in handles to keep tasks waiting for future versions. The container uses a C++ standard library allocator, whose type is defined in the options struct. To free tasks, the options struct contains a `FreeTask` type, that has to provide a `free()` method, which will be called with the finished task as a parameter. These types can be overridden to use custom memory allocators instead.

4.3. User-defined access types. It is possible for SuperGlue users to use a different set of access types than the default ones. When defining an access type, the programmer needs to declare whether it can be reordered with other accesses of the same type and, if it can, also whether the accesses need to be exclusive or not.

As an example, if some tasks perform an additive operation, while other tasks perform a multiplicative operation, tasks performing the same access types can be reordered, while order must be preserved among tasks with different access types. One might also want to add an access type that allows several tasks to modify data concurrently. This allows data-races, or lets the user manage data-races manually.

```

1 struct MultiBuffered {
2     Handle<Options> h[NUM.BUFFERS];
3     double *data[NUM.BUFFERS];
4     int index = 0;
5     // ...
6     Handle<Options> &get_handle() { return h[index]; }
7     double *get_buffer() { return data[index]; }
8     void next() { index = (index + 1) % NUM.BUFFERS; }
9 };

```

LISTING 8

Example of using multiple buffering to avoid write-after-read dependencies. This implementation is not thread safe, but works if all tasks are submitted from a single thread, and can easily be extended to be thread safe.

4.4. Renaming. There is no write-only access type in SuperGlue. A write-only access would signal that a new instance of the data could be created, and a task could start working on it immediately, without having to wait for previous tasks that need the old data to finish first, avoiding a write-after-read dependency. This is called *renaming*, for its similarities with register renaming performed in processors. SuperGlue does not support automatic renaming, since it has no control over the buffers the handles represent. Also, automatic renaming is not always desired. If there already is enough parallelism, it is better to let write-only tasks wait and reuse the same buffer. If the runtime system creates new buffers automatically, there is both a risk of completely running out of memory, and a risk that the larger working set causes a loss of performance due to more cache misses.

One way to avoid write-after-read dependencies is to use multiple buffering explicitly. Listing 8 illustrates the idea. Each buffer gets its own handle, and write-only operations call the `next` method before registering their write access.

Note that this solution will always alternate buffers, even if all tasks that accessed the old buffer have already finished.

4.5. Parallel reduction. There is some support for automatic renaming for *add* accesses, to allow parallel reductions. When this feature is enabled, a task that needs exclusive access to a handle that is locked will still be able to execute. The task must then write its partial results into another buffer instead. When the task is finished, it attaches this buffer to the handle. The handle can only keep a single reduction buffer. If another buffer is already attached, it is detached and merged with the new buffer using a user defined function. The process is then repeated, to attach the merged buffer to the handle.

This allows tasks that use the *add* access to run in parallel. The partial results are merged lazily, and are merged in parallel if several threads attach partial results concurrently. The buffers are not guaranteed to be merged until a task accesses the handle with a different access type.

The user must provide functions for creating new buffers and initializing them, merging two buffers, and applying a buffer to the real destination. Since the merge and the apply operation are two separate functions, it is possible to have different representations in the temporary buffers and in the real destination buffer. When a task is declared to support this feature, the user must program the tasks to check where the output should be written. The tasks must support three possibilities; write to the real destination, write to an existing temporary buffer, or create and initialize a new temporary buffer and write into this.

4.6. Thread workspace. For tasks that require temporary work space buffers, there is an option to enable each thread to preallocate a buffer of a determined size. Tasks can then request work space memory from this buffer, which is guaranteed to be local to the thread, and need not be allocated and freed for each task.

4.7. Visualization and instrumentation. A positive effect of the task-based model is that the beginning and end of each task execution are natural locations to instrument the software. By storing the start and end time of each task, and on which thread it was executed, we can draw a task execution trace as in Figure 4.1. This is a trace from executing a program with dependencies such as illustrated in Figure 1.1(a) and Figure 2.1. From such an execution trace it is possible to see when tasks are started, for how long they execute, and on which thread they were executed. We use triangles instead of rectangles to illustrate tasks as we find that it makes

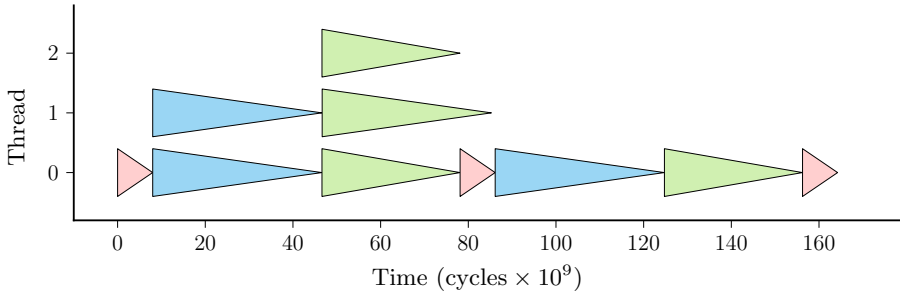


FIG. 4.1. An execution trace illustrating how a task-based application is executed. Each triangle represents a task, and colors distinguish different task types.

it easier to separate several small tasks from a large task, and easier to detect idle time between tasks. From an execution trace, it is possible to directly see if there is enough parallelism, if any cores are idle, in which parts of the program most time is spent, and if there is any unexpected behavior. We will use these execution traces to evaluate how well programs are scheduled. In [23], execution traces are used to detect tasks that exhibit performance degradation due to shared resources such as memory bandwidth. This information is then used to insert scheduling constraints that prevents contention-sensitive tasks from being executed concurrently.

Instrumentation is activated by defining a type in the options struct with member functions that will be called before and after each task invocation. In addition to generating execution traces as in Figure 4.1, SuperGlue includes example instrumentation classes to read performance counters before and after each executed task, and there is also an option to generate task dependency graphs in Graphviz DOT format, for illustrating the dependencies in an application.

5. Micro-Benchmarks. In this section we evaluate the performance of the proposed programming model, by running experiments on our SuperGlue implementation.

Time is measured by reading the time stamp counter using the `rdtsc` instruction. This counter has a constant rate, and behaves like a wall-clock timer. Each thread is pinned to its own core or hardware thread in all experiments. For experiments when not all cores are used, we first allocate one thread per module and per core, and only when all modules or cores are already occupied, we assign workers to share a module or a core.

There are no parameters in the SuperGlue implementation that are tuned for the specific hardware used in the experiments.

5.1. Scaling with respect to number of cores. To ensure that our solution scales well to large numbers of cores, we have performed tests on a Xeon Phi 5110P, with 60 cores and 240 hardware threads. The experiment computed a tiled Cholesky factorization of a matrix of size 16384×16384 , divided into blocks of 64×64 elements. The execution trace is shown in Figure 5.1. In the execution trace, it can be seen that some time is needed to start up and reach full speed, which is not reached until at about time 1. In the end of the execution trace, there is less parallelism in the algorithm, which leads to tasks being executed more sparsely. Just after time 2, before time 6, and at time 7, there are short intervals where the application runs out of tasks to execute and worker threads are idle. The reason for these imperfections are not

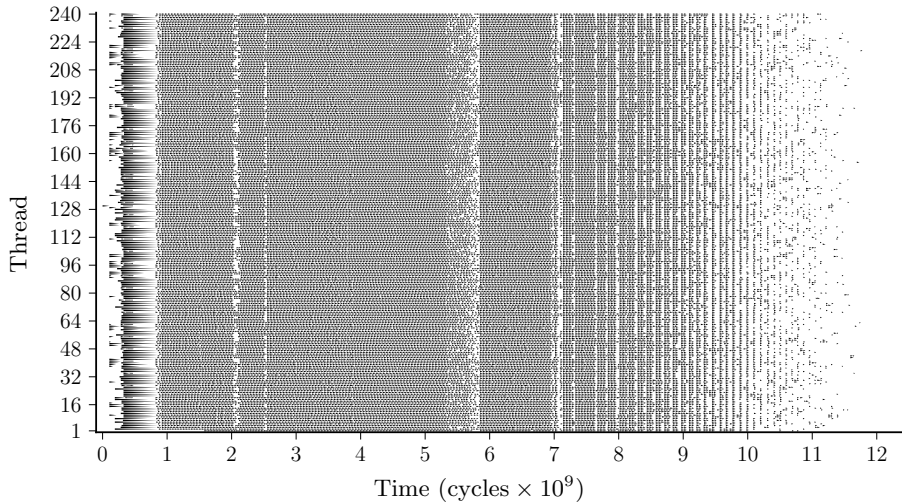


FIG. 5.1. Execution trace of a Cholesky factorization run on the 240 hardware threads of a 60 core Intel Xeon Phi. The matrix size is 16384×16384 , blocked in 64×64 blocks of 256×256 elements each,

clear, but we believe it is due to suboptimal scheduling. Introducing task priorities and prioritizing tasks on the critical path, which in this case are the tasks that work on the diagonal blocks, did not improve on this behavior.

In this experiment, only 131 GFlop/s was reached while the theoretical peak is 1011 GFlop/s^2 . The kernels used in this experiment only reached 2.35 GFlop/s when running on a single thread. The parallel version was 55.8x faster, close to the ideal 60x. The purpose of this experiment is to verify that the runtime system scales up to large numbers of cores. After the start-up effects and before the algorithm runs out of parallelism, the execution trace shows that tasks are scheduled densely and as desired, and that SuperGlue successfully scales up to at least 240 threads.

5.2. Scaling with respect to task size. To investigate how well our solution scales for different task granularities, we measure the time it takes to execute a number of identical and independent tasks that perform no work, but only wait for a number of cycles. The tasks perform no work in order to isolate the behavior of the runtime system, and to have higher precision in the control of the granularity. This means that we investigate the ideal case when tasks do not affect other running tasks by competing for shared resources such as cache or memory bandwidth. The time is measured from when the first task starts until the last task finishes, and does not include time for startup or shutdown.

Memory management of tasks impacts the performance. To make the experiment less dependent on the memory allocator, all memory to store tasks is allocated in advance and returned first after all tasks are executed, so that no memory management of the tasks is included in the timing. This experiment is repeated in Section 7, where we compare our solution to other projects, with standard memory allocation included in the timing.

²The theoretical peak was calculated as 8 doubles wide vector registers \times 2 operations per instruction with fused multiply and add \times 60 cores \times 1052.63 MHz.

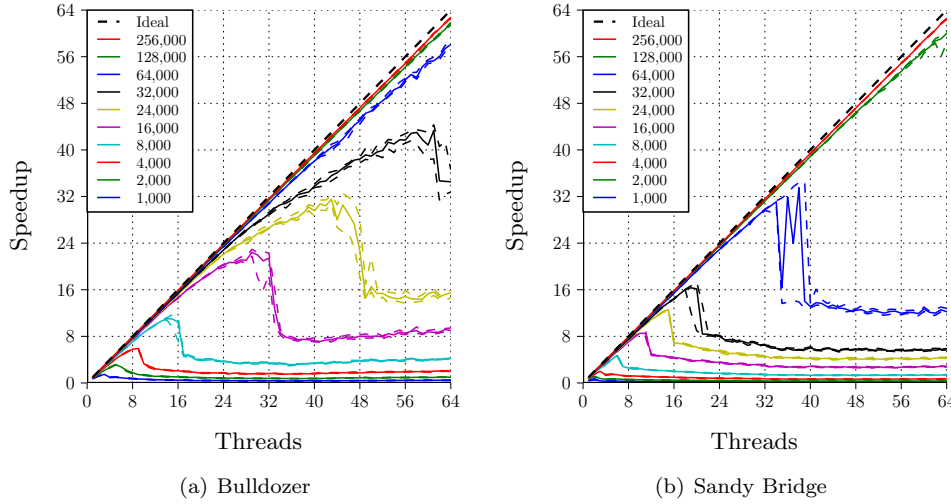


FIG. 5.2. Scaling with respect to task size, where task size is the execution time of a single task measured in cycles. The graph shows the speedup over the ideal execution time (number of tasks \times task size / threads). Each line corresponds to a task size. The solid lines show the median, and dashed lines show the upper and lower quartiles of 15 runs. The experiment is performed on two different computer architectures; Bulldozer and Sandy Bridge.

The experiment is conducted on two different computer systems. The first is called *Bulldozer* and has four AMD Opteron 6276 processors based on the Bulldozer architecture. Each processor has 8 modules, and each module contains two cores and a single floating point unit that is shared between the two cores. This results in 64 cores running at 2.3 GHz. The second is called *Sandy Bridge* and has four Intel Xeon E5-4650 based on the Sandy Bridge architecture, giving a total of 32 cores running at 2.7 GHz. The tests were run with hyper-threading enabled, which results in 64 hardware threads.

The number of tasks is selected to 6,400, which is large enough to fill all the threads, and large enough to make the steady state dominant and the irregular behavior at startup and shutdown less significant. Increasing the number of tasks did not affect the results in a qualitative way. The experiment is repeated 15 times for each task granularity to capture dispersion in execution time.

Figure 5.2 shows speedup over the ideal execution time on one thread, where the ideal execution time is the number of tasks times the number of cycles per task. Note that it is impossible to reach perfect efficiency, since that would require every single clock cycle to be spent inside a task, and leave no cycles for task management.

On Bulldozer, good scaling is reached when tasks are at least 64,000 cycles, corresponding to just below 30 μ s. On Sandy Bridge, tasks should take at least 128,000 cycles, corresponding to just below 50 μ s on that architecture.

Increasing the number of threads when the task size is too small can lead to a sharp drop in performance. This is due to contention on the locks protecting the task queues. These locks are simple test-and-test-and-set spinlocks with no back-off. We have performed experiments with more advanced locks, but the benefits turned out not to outweigh the increased cost of acquiring or releasing the locks, at least in our implementations. Adding a back-off helps against the sharp drop in performance,

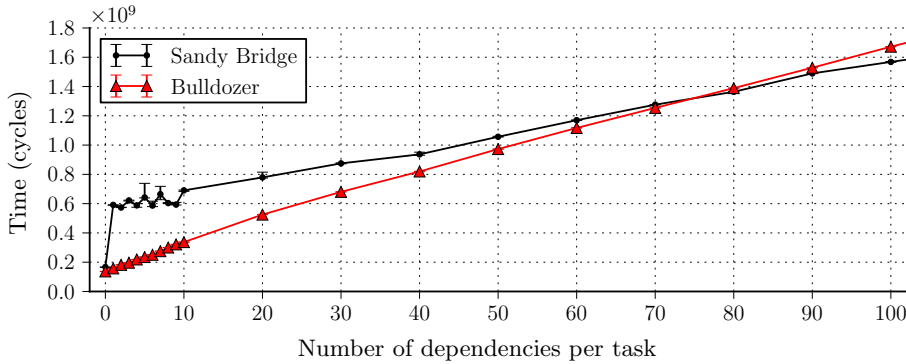


FIG. 5.3. *Scaling with respect to number of dependencies. 64,000 tasks executed on 64 cores, each waiting for 128,000 cycles, and accessing a varying number of handles. Error bars show the first and third quartiles of 15 runs, but are barely visible due to low dispersion.*

but it is unclear how to select good parameters that work for all platforms. Since it would be unfair to tune the locks for the specific machines that we use to perform our experiments on when we compare against other systems in Section 7, we opted to keep the simple, parameter free, spinlocks.

5.3. Scaling with respect to number of dependencies. The aim of this benchmark is to measure how overhead for dependency management scales with number of dependencies. 64,000 tasks are submitted, which is a large enough number to make the long-term behavior dominant and startup and shutdown anomalies negligible. Each task waits for 128,000 cycles, selected to be long enough to avoid contention according to the results from section 5.2. Each test was repeated 15 times in order to capture dispersion in the execution times. Memory is allocated in advance, and not freed until after the experiment. All tasks are submitted before any task is allowed to execute, and both task submission and execution is included in the timing.

The experiment was run on both the Sandy Bridge and the Bulldozer machine, and the results are shown in Figure 5.3. On Bulldozer, the execution time is close to linear in the number of dependencies, but the slope changes slightly from about 300 cycles per dependency to 250 cycles per dependency. On Sandy Bridge, there is a larger cost for introducing dependencies, which we have no explanation for. For large number of dependencies, the overhead approaches 217 cycles per dependency.

We executed the test with up to 10,000 dependencies on Bulldozer, at which point all 64,000 tasks each must keep a list of its 10,000 dependencies, meaning that 640 million items are stored in the memory. This is more than is practically useful. The mean overhead when each task had 10,000 dependencies was about 214 cycles per dependency.

From these tests we verify that our solution scales linearly with the number of dependencies, and that it does not collapse when the number of dependencies is large. The results varied between the two machines, but can be summarized as that there might be an initial cost to use dependencies, that was measured to about 6500 cycles on Sandy Bridge, but was not seen on Bulldozer. When dependencies are used, the overhead to expect from adding another dependency was measured to between 200 and 300 cycles.

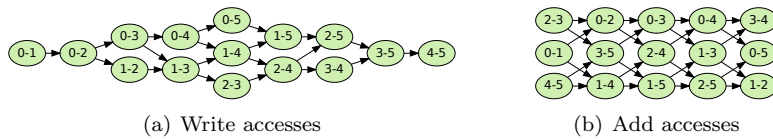


FIG. 6.1. Comparison of the dependency graphs arising from using write respectively add accesses for updates that can be reordered.

6. Applications. In order to show the usability of SuperGlue for real applications, this section contains a couple of more realistic applications that we have parallelized using SuperGlue. In addition to the applications presented here, SuperGlue has been used to target heterogeneous platforms in [11], where a fast multipole method is parallelized for hybrid CPU and GPU systems.

6.1. Direct n -body simulation. This application performs a n -body simulation, that simulates how n neutral atoms or molecules interact with each other, modeled by the Lennard-Jones potential. The simulation consists of two steps; calculating all the forces that act upon each particle, and moving the particles according to the forces acting upon them. The forces are calculated by the direct sum of all pair-wise interactions as modeled by the Lennard-Jones potential. This method does not scale to large numbers of particles, but it is the computational kernel needed also in more efficient methods such as the fast multipole method or Barnes-Hut.

This application is interesting because during the force calculations, there are two tasks that update the force acting upon each particle. If these tasks are submitted in a straight-forward order and the tasks are registered to *write* to the force field of the particles, the tasks are constrained to perform the updates in the order the tasks were submitted, which may allow very little parallelism. This effect was studied in for example [18].

```

1 struct BufferProxy {
2     Access<Options> &a;
3     double *buffer;
4     BufferProxy(Access<Options> &a, double *org_buffer) : a(a.) {
5         if (!a.is_busy())
6             buffer = org_buffer;           // case 1: use original buffer
7         else if ( !(buffer = a.get_tempbuf()) ) // case 2: reuse temporary buffer
8             buffer = allocate_and_initialize(); // case 3: new temporary buffer
9     }
10    ~BufferProxy() {
11        if (a.is_busy())
12            a.attach_tempbuf(buffer);       // attach temporary buffer to handle
13    }
14 };
15
16 struct ForceTask : public Task<Options> {
17     ...
18     bool is_renaming_supported() { return true; } // renaming is supported
19     void run() {
20         BufferProxy proxy(get_access(0), force_buffer[i]);
21         // perform computations here, write output to "proxy.buffer"
22     } };

```

LISTING 9

Simplified code example, showing how renaming is implemented in the n -body simulation application.

When the force field updates are registered as *add* accesses instead, tasks can be reordered and more parallelism can be extracted. This effect is illustrated in Figure 6.1. Here, numbers denote blocks of the force data. Two tasks may not update the same

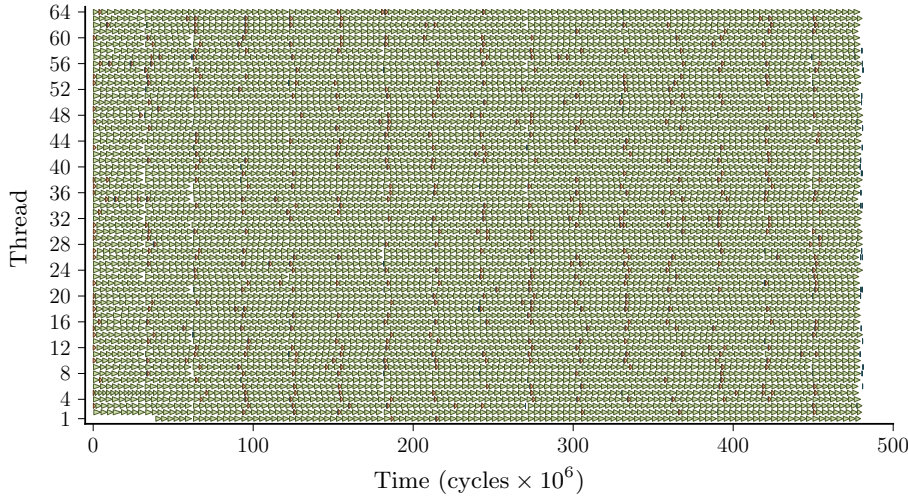


FIG. 6.2. Execution trace of a n -body simulation of 8192 particles blocked into blocks of 256 particles, stepped for 16 time steps.

block at the same time, meaning that any two tasks that have a number in common have a dependency between them. Figure 6.1 (a) shows the case when accesses are registered as writes, and cannot be reordered, while Figure 6.1 (b) shows a possible dependency graph, with more parallelism, arising from registering the updates as adds, which allows tasks to be reordered.

In this application, we used the renaming feature of SuperGlue. We simulated 8192 particles grouped into 32 blocks of 256 particles each. The problem size and the number of blocks is on the limit of what is enough work for 64 threads, and was selected to show how SuperGlue behaves when parallelism is scarce. Most force calculation tasks (taking over 95% of the execution time) write force contributions to two blocks of particles at once, which means that only 16 such tasks can run at the same time. One way around this problem is to create smaller blocks, but this would also increase the overhead. By using the renaming feature as described in Section 4.4, force calculation tasks can run even if the output buffer is used by another task, in which case the results are written into a temporary buffer. Any such temporary buffers are then merged into the original buffer before it is read the next time.

Listing 9 shows a slightly simplified version of how renaming is implemented in the n -body simulation. The `Access` class keeps a handle and the required version, and can be retrieved from a task using the `get_access()` method. The user must implement a `is_renaming_supported()` method that returns `true` to indicate that SuperGlue can start tasks even when exclusive access could not be granted to all handles. To find out if exclusive access was acquired for a handle, the `Access` class has an `is_busy()` method that returns `true` if renaming is required for its handle.

Figure 6.2 shows an execution trace of the n -body simulation application, executing 16 force calculation and particle movement steps. Over 95% of the time is spent on force calculation tasks that without renaming, would be limited to at most 16 concurrent tasks. Thanks to renaming, we can see that all 64 threads are executing tasks during the whole execution. There is an exception for thread 0 which executes no tasks in the beginning of the execution as it is busy creating and submitting the

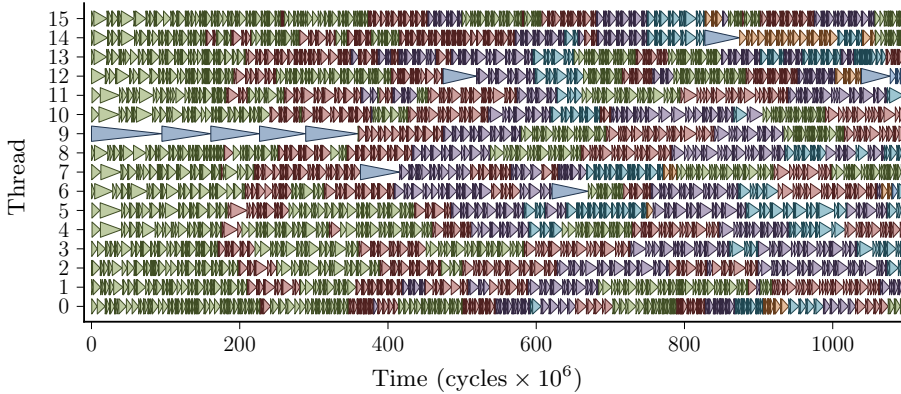


FIG. 6.3. Part of an execution trace of a shallow water equations solver. Different colors represent tasks working on different time steps.

tasks. During this period, it can be seen that thread 4 executes its tasks faster, since that thread shares the floating point unit with thread 0. The dependencies between the time steps can be seen by small gaps in the schedule between most time steps. Running on 64 threads on the Bulldozer system, was about 35 times faster than our best serial implementation. Since the system only has 32 floating point units, and the application is dominated by floating point calculations, a speedup much higher than 32 cannot be expected.

6.2. Shallow water simulation. In [24], we describe an implementation of a shallow water simulation code that uses the SuperGlue library. Just like the n -body simulation, this is a time dependent problem, where each time step depends on the previous. It uses the fourth order Runge-Kutta method for time stepping, where each function evaluation includes two sparse matrix-vector multiplications, which is where over 90% of the execution time is spent.

The problem size is 655,362 unknowns, and the matrices have 20,316,222 non-zeros each. For parallelization, the solution vector is sliced into 131 blocks of 5,000 elements each, except for the last one which is slightly larger. The matrices are blocked in tiles to match the vector blocks. The non-zeros are centered along the diagonal, so many off-diagonal tiles are empty and discarded, leaving only 391 non-empty tiles. The number of non-zeros varies significantly between different tiles.

In order to handle large numbers of time steps, it is not feasible to submit all the tasks at once. Instead, we introduce a `GenerateTask` task, that submits all tasks needed to take one time step. In the end of each `GenerateTask`, it submits another copy of itself which depends on one of the tasks from the time step. Initially, five of these `GenerateTask` tasks are submitted, to generate tasks to take the first five time steps. When each time step is finished, a new `GenerateTask` task will have been submitted. This way, tasks will be submitted continuously. This continues until the end time of the simulation is reached, after which the `GenerateTask` tasks will do nothing.

Figure 6.3 shows the beginning of an execution trace of the shallow water simulation. The `GenerateTask` tasks can be seen as longer than the others, and the five first are executed by thread 9. More `GenerateTask` tasks can be seen throughout the trace, on other threads. Tasks working on different time steps have different colors,

TABLE 7.1
Compared frameworks

Cilk Plus ³	SMPSs 2.4
Intel TBB 4.2	StarPU 1.1.0
OmpSs ⁴	SuperGlue
OpenMP ⁵	Swan ⁶
QUARK 0.9.0	XKaapi 2.1

and the trace shows that tasks from different time steps are mixed freely. Because SuperGlue is driven by locality, it will prefer to take the next time step as soon as possible rather than finishing all tasks from a time step first. When the last task from the first time step finished in this trace, the first tasks from the eleventh time step had already been executed.

Sparse matrix-vector multiplication is a memory bound operation, and is difficult to parallelize efficiently on multicore systems. The parallel version was 5 times faster than the best serial when run on a server with two 8-core AMD Opteron 6220 processors. On this architecture, each pair of cores share a single floating point unit, and the ideal speedup is therefore 8. In this run, less than 2% of the execution time was spent on task management (including executing the task generation tasks) or on threads being idle. The reason the speedup is not closer to 8 is due to contention when all threads compete for the memory bandwidth.

7. Comparison with other efforts. In order to verify that our programming model is competitive, we perform a number of performance experiments comparing our implementation against other efforts. All the experiments measure time from the first tasks start to the last task finishes. This includes both the overhead induced by task management and the quality of the scheduling, but excludes time to start up and shut down the runtime system. As in the experiments in Section 5, we let each task wait for a number of cycles instead of performing actual computations, in order to isolate the task management from application specific behavior. The frameworks we have compared are listed in Table 7.1. They were all compiled with the `-O3` optimization flag. The `-O2` optimization level was evaluated too, but there was no qualitative difference, and `-O3` was selected.

We were not able to install the prerequisites to compile OmpSs on the Bulldozer server, and we had problems with crashes from XKaapi when too many tasks were submitted. Because of this, these two frameworks are missing from the experiments on Bulldozer. The default settings were used for all frameworks, except for StarPU which issues a warning that the user should select a scheduling strategy. StarPU is targeting heterogeneous platforms, and its main strength is to determine whether tasks should run on the GPU or CPU. For this it uses performance models that estimate task execution time and time to move data. Since we have a slightly different objective and consider only homogeneous architectures, simpler scheduling will suffice and the additional time StarPU spends on scheduling will only become overhead. All experiments with StarPU uses the work stealing scheduler, which was found out to give best results in an initial experiment.

³Using the Cilk Plus branch from GCC 4.8

⁴Built using Nanos++ 0.7a (2013-09-05) and Mercurium 1.99.0 (2013-09-10)

⁵Compiled with GCC 4.7.2 on Bulldozer, and GCC 4.8.1 on Desktop

⁶Built from latest version on GitHub, last commit 2013-08-10

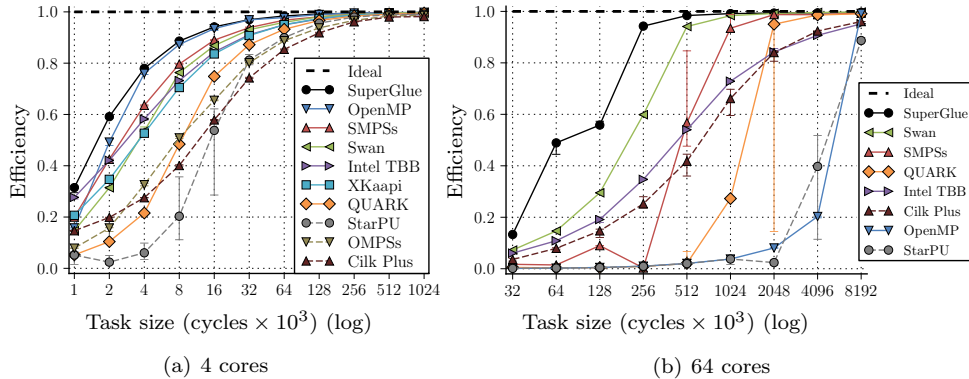


FIG. 7.1. Independent tasks; comparison of efficiency between different task frameworks. Error bars show the upper and lower quartiles.

7.1. Performance: Independent tasks. This benchmark investigate the behavior when there are no dependencies between the tasks. The experiment is basically the same as in Section 5.2, except that task allocation is now included for SuperGlue, and that we always run on all available CPUs. Each task only waits for a predefined number of cycles, and the time to submit and execute all tasks is measured. The test was executed on two different machines, the 64-core Bulldozer server (described in Section 5.2), and a 4-core desktop machine called Desktop, that is equipped with an Intel Core i7 2600K processor running at 3.4 GHz.

In this benchmark, we create 12,800 tasks when running on 64 threads, and 2400 tasks when running on 4 threads, which turned out to be enough work for all threads and avoids anomalies at startup and shutdown. We repeat the experiment for different number of cycles in each task wait, and plot the efficiency, which we define to the ratio of the ideal execution time (number of tasks × cycles each task waits / number of CPUs) to the measured execution time. This gives a measure of how large tasks need to be in order to make task management negligible and the execution efficient. For each wait time, the experiment was repeated 50 times on Desktop, and 5 times on Bulldozer in order to capture dispersion in execution times.

Figure 7.1 shows the outcome. SuperGlue is the most efficient framework on both 4 and 64 cores. OpenMP is about as efficient as SuperGlue on 4 cores, but ended up among the least efficient on 64 cores. We see here that StarPU is among the frameworks which requires the largest task sizes in order to reach high efficiency, because it spends more time on scheduling as discussed above.

7.2. Performance: Tasks with dependencies. In order to compare the performance of the dependency management and scheduling, we use the dependencies from a block Cholesky factorization, which is selected because it is a standard benchmark to represent realistic dependencies. In this experiment, we make use of data-driven dependencies, and frameworks that do not provide this, i.e. OpenMP, Intel TBB, and Cilk Plus, are not included. The tasks perform no actual work, but only wait for a determined number of cycles.

The experiments are run both on the 4-core Desktop machine, where the matrix is blocked into 20×20 tiles, and on the 64-core Bulldozer server, where its blocked into 50×50 tiles. This corresponds to 1540 tasks on 4-cores and 22,100 on 64 cores.

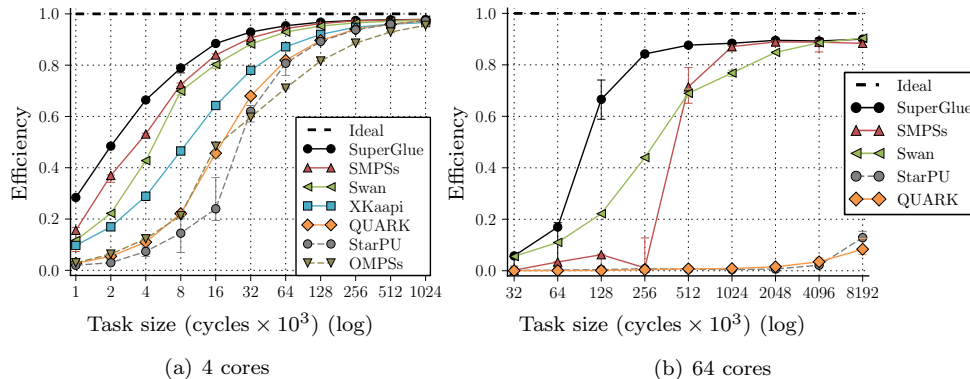


FIG. 7.2. Tasks with dependencies; comparison of efficiency between different task frameworks. Error bars show the upper and lower quartiles.

The test was repeated 20 times on Desktop, and 10 times on Bulldozer in order to capture dispersion in execution time.

Figure 7.2 shows the results. SuperGlue reaches high efficiency for smaller task sizes than other frameworks. On 4 cores, the results are similar to the results from the comparison with independent tasks, but all frameworks have slightly lower efficiency. This is both because of dependency management costs, and because the dependencies limit the parallelism. The ideal efficiency does not take task dependencies into account, so the best possible efficiency is actually slightly lower. On 64 cores, results are also similar, except for QUARK and StarPU that now require much larger tasks to reach good efficiency. When moving from 8,000 tasks to 16,000 tasks, there is a jump in the figure for StarPU, and the overhead doubles. In the experiments on the 64-core Bulldozer server in Section 7.1 and Section 7.2, 12,800 respectively 22,100 tasks were submitted, which is just around this jump, and this is part of the explanation why StarPU did not reach higher efficiency in the previous tests.

7.3. Performance: Scaling with respect to number of tasks. To verify that our solution scales with the number of tasks, this benchmark submits different amounts of tasks, each waiting for 100,000 cycles, and measure how much time is spent on overhead. To find the overhead, we compare the time it takes to submit and execute all tasks to the ideal time (100,000 cycles per task \times number of tasks / 4 cores).

The results are shown in Figure 7.3. Most frameworks are not affected by the number of tasks. The runtime for OmpSs varies for low numbers of tasks, but reaches a steady level for larger numbers of tasks. StarPU is the only framework that is sensitive to large numbers of tasks. When moving from 8,000 tasks to 16,000 tasks, the overhead is doubled. SuperGlue scales well with the number of tasks, and has the lowest overhead among all compared frameworks.

8. Discussion. The most promising approach to program for multicore processors is to divide the software into tasks, and let a runtime system schedule these tasks for execution in parallel. In real applications, the tasks have dependencies, and these dependencies must be handled correctly and efficiently. Introducing artificial dependencies or unnecessary synchronization can be devastating for performance. Instead, fine-grained synchronization is needed. Also, task scheduling must be fast, or software

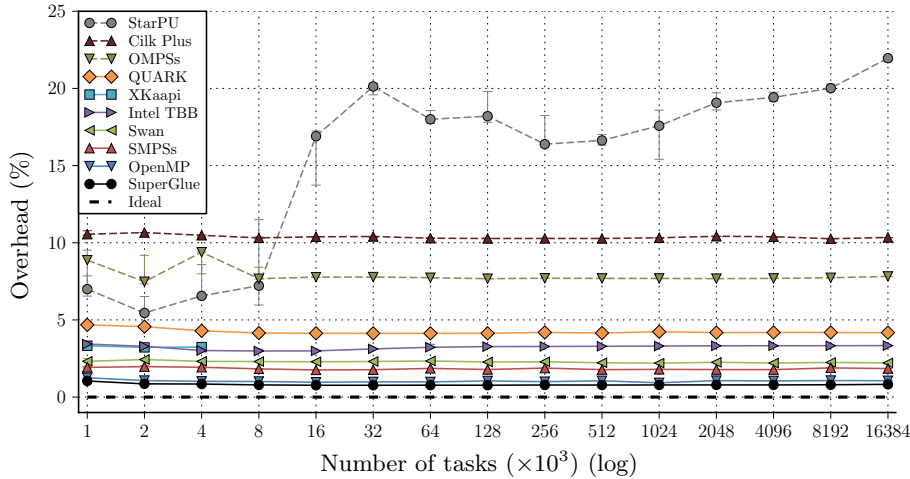


FIG. 7.3. Scaling with respect to number of tasks when running on 4 cores. Error bars show the first and third quartile of 5 executions.

consisting of very small tasks will be dominated by the scheduling overhead and will not scale at all.

In this paper we present a runtime system that can represent dependencies in a flexible way. The scheduling is driven by locality, and is distributed with no central information. We provide several micro-benchmarks showing that our implementation is efficient and scales well. We also show that it can be practically applicable by presenting a few example applications that use SuperGlue. Finally, we compare against a large number of other task-based libraries, and show that our implementation is equally or more efficient than the others.

REFERENCES

- [1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.
- [2] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, March 2009.
- [3] P. Bellens, J. M. Pérez, R. M. Badia, and J. Labarta. CellSs: a programming model for the Cell BE architecture. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing, SC '06*, New York, NY, USA, 2006. ACM.
- [4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, August 1995.
- [5] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.
- [6] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, January–March 1998.
- [7] J. Dongarra, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, A. YarKhan, W. Alvaro, M. Faverge, A. Haidar, J. Hoffman, E. Agullo, A. Buttari, and B. Hadri. PLASMA users’ guide. <http://icl.cs.utk.edu/plasma/>.
- [8] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.
- [9] F. Galilée, J.-L. Roch, G. G. H. Cavalheiro, and M. Doreille. Athapascan-1: on-line building

- data flow graph in a parallel language. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 88–95, Washington, DC, USA, 1998. IEEE Computer Society.
- [10] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. XKaapi: a runtime system for data-flow task programming on heterogeneous architectures. In *2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1299–1308, 2013.
 - [11] M. Holm, S. Engblom, A. Goude, and S. Holmgren. Dynamic autotuning of adaptive fast multipole methods on hybrid multicore CPU & GPU systems. *CoRR*, abs/1311.1006, 2013.
 - [12] Intel Cilk Plus. <http://www.cilkplus.org/>.
 - [13] Intel Threading Building Blocks. <http://threadingbuildingblocks.org/>.
 - [14] J. Kurzak and J. Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. In B. Kågström, E. Elmroth, J. Dongarra, and J. Waniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 147–156. Springer Berlin Heidelberg, 2007.
 - [15] J. Kurzak, H. Ltaief, J. Dongarra, and R. Badia. Scheduling linear algebra operations on multicore processors. Technical Report UT-CS-09-636, Innovative Computing Laboratory, University of Tennessee, 2009.
 - [16] M. S. Lam and M. C. Rinard. Coarse-grain parallel programming in Jade. *SIGPLAN Not.*, 26(7):94–105, April 1991.
 - [17] C. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51:244–257, 2010. 10.1007/s11227-010-0405-3.
 - [18] C. Niethammer, C. W. Glass, and J. Gracia. Avoiding serialization effects in data / dependency aware task parallel algorithms for spatial decomposition. In *Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, ISPA '12*, pages 743–748, Washington, DC, USA, 2012. IEEE Computer Society.
 - [19] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, July 2013.
 - [20] J. M. Pérez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *2008 IEEE International Conference on Cluster Computing*, pages 142–151, 2008.
 - [21] M. Tillenius. *Leveraging Multicore Processors for Scientific Computing*. Licentiate thesis, Department of Information Technology, Uppsala University, September 2012.
 - [22] M. Tillenius and E. Larsson. An efficient task-based approach for solving the n -body problem on multicore architectures. In *PARA 2010: State of the Art in Scientific and Parallel Computing*. University of Iceland, Reykjavík, 2010, 4 pp.
 - [23] M. Tillenius, E. Larsson, R. M. Badia, and X. Martorell. Resource-aware task scheduling. Technical Report 2014-001, Department of Information Technology, Uppsala University, January 2014.
 - [24] M. Tillenius, E. Larsson, E. Lehto, and N. Flyer. A task parallel implementation of an rbf-generated finite difference method for the shallow water equations on the sphere. Technical Report 2014-011, Department of Information Technology, Uppsala University, April 2014.
 - [25] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. A unified scheduler for recursive and task dataflow parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–11, October 2011.
 - [26] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK users' guide: queueing and runtime for kernels. Technical Report ICL-UT-11-02, Innovative Computing Laboratory, University of Tennessee, 2011.