An efficient task-based approach for solving the n-body problem on multicore architectures*

Martin Tillenius† and Elisabeth Larsson‡

Department of Information Technology, Uppsala University

Abstract With the aim of exploring programming models that can improve the efficiency in high performance scientific computing software development for multicore architectures, we have implemented a task-based library with dynamic scheduling and automatic handling of data-dependencies. The library has been evaluated both from a performance and a programmer productivity perspective. We find that the approach is a powerful way to express computational problems without burdening the programmer with the details of the parallel execution. The performance results are good when the individual tasks are large enough.

Keywords multicore, programming model, task-based programming, dynamic task scheduling

1 Introduction

All computer systems today, from laptops to supercomputers, are built from multicore processors. This means that all application software needs to be parallel in order to get good performance. In high performance computing this is not new, the software is already parallelized. However, software that used to perform well on pre-multicore systems does not in general yield acceptable performance when run on a multicore machine. It is becoming increasingly hard to produce efficient software with a reasonable programming effort. In this work, the aim is to develop a programming model that allows fast development while facilitating efficient use of multicore hardware resources.

Multicore systems typically have heterogeneous architectures, non-uniform memory access, and several running processes competing for the system resources, making it excessively hard to predict the time needed for a certain computation to finish. This makes it unrealistic to decide in advance which computations should be performed on which core and when, which has led us to consider a programming model where computations are scheduled onto cores dynamically at run-time.

Dynamic scheduling combined with representing tasks and their dependencies as a directed acyclic graph has shown to be a successful approach for dense linear algebra applications in for instance [6] and [4], and also shows promise for sparse linear algebra in [5].

There are several available environments for task-based programming, for instance Cilk[3], Wool[2], Intel's Threading Building Blocks[9], and tasking in OpenMP 3.0[7], which are all mainly focused on recursively created tasks, and also SMP Superscalar[8] and the task library described in [6], which are designed for more general task execution patterns driven by data dependencies. The task library we present here belongs to this second group.

2 Application

The application area for which we aim to develop parallel algorithms and software is radial basis function (RBF) approximation methods for solving partial differential equations, see [1] for an overview.

For a first evaluation of different programming models we consider the subproblem of generating adaptive point distributions for RBF methods. In an adaptive solver we want to distribute the center points of the RBFs more densely where we need accuracy and more sparsely elsewhere. Our approach for this is to see this as an n-body problem where we treat the center points as particles, introduce forces between these particles, and integrate Newton's equations of motion in time until we reach convergence.

The n-body problem often arises in scientific computing applications, for instance in studies of protein folding and material physics, and also in the study of galaxies in cosmology.

3 Implementation

This section describes the implementation details of our task library. The library is implemented in C++ and uses pthreads on Solaris and Linux and the Win32 API on Windows for handling threads. The main ideas of our implementation are described in the following sections.

3.1 Programmer Interface

In order to detect data dependencies, we demand that all shared data structures are wrapped in objects of a special class, which we will refer to as *datas*. The programmer creates a task by subclassing a certain task class and must

*The work was supported by the Linneus center of excellence Uppsala Programming for Multicore Architectures (UPMARC), funded by the Swedish Research Council

†Email: martin.tillenius@it.uu.se

‡Email: elisabeth.larsson@it.uu.se

register all the datas that the task accesses and what type of access it is. The programmer then adds tasks to the library in the order of the sequential algorithm and it is the responsibility of the task library to detect which tasks can be executed in parallel and to schedule the tasks so that no data-races occurs between registered data accesses.

3.2 Task Library Design and Data Locality

The task library uses a manager/worker scheme. The main thread is called the *manager* and is used to create and add tasks to the task library, but does not participate in executing the tasks. The task execution is performed by worker threads called *workers*, and as many workers as there are cores available on the system are created.

To reach good performance on multicore architectures, it is essential to reuse data that is already in the caches and to use the shared caches to communicate between threads. In order to keep track of which data is in which cache, each data is allocated to a certain worker when it is created, and each worker is pinned to a core of its own and not allowed to move around. In our application the datas are arrays that are sliced up in as many slices as there are available cores and distributed evenly among the workers.

Tasks are called *ready* if all their data dependencies are fulfilled and *waiting* otherwise, and each worker thread has a *ready queue* with ready tasks. A task that are created from a task is called a *subtask*, and is assigned to the worker it was created from, as it is expected to continue to work on the same data. A task created from the manager gets assigned to a worker by randomly selecting a data that the task accesses and assigning the task to the same worker as the selected data is assigned to.

The manager adds tasks to the ends of the workers' ready queues and the workers execute tasks from the front of their queue. Subtasks on the other hand are added to the front of the queue, as we expect them to access the same data that is already in the cache.

Load balancing is handled by *task stealing*, meaning that when a worker runs out of tasks to execute, it tries to steal tasks from other workers. Tasks are stolen from the end of the other worker's ready queue to avoid locking the front of the queue for the worker and because the tasks in the front of the ready queue are more probable to be working on the same data that is already in the cache.

3.3 Data Dependencies

Instead of representing dependencies between tasks, we focus on dependencies between a task and the datas it accesses. In the task library presented here, a task knows nothing about other tasks but only what datas it accesses.

The data dependencies are handled by attaching version numbers to all datas and storing a lowest required version number (called *required version*) for each access in the task when the task is added to the library.

When a task is finished executing, it increases the version number of each data it has accessed. Data dependencies are checked by comparing the required versions stored inside the task with the version numbers attached to the accessed data. By comparing these version numbers we can detect if a task is ready to run, and we can also be sure that for instance all tasks that wants to read a certain data have finished before allowing a task to overwrite this data. This does not add any artificial ordering of execution between the tasks, but allows the task library to schedule tasks freely.

The dependencies are examined when a task is added, and only unfulfilled data version requirements needs to be remembered. When a data is found whose version is too low, the task tells the data that is wishes to be notified when the desired version is available. When the task is later notified that the required version is available, it resumes checking the rest of the dependencies, and when all dependencies are solved it is moved to the ready queue.

3.4 Access Types

The user must register what type of access a task performs on each data, as the library needs this information to find the required version for that access. What access types to use depends on the problem, and for our application we have used the accesses Read, Write, and Add. The Read access are used to indicate that the task will not modify the data, the Write access is used for any task that might modify the data, and the Add access is used to indicate that the task will modify the data, but only by adding values, so that the order in which two Add accesses are executed does not matter.

The library needs to know which accesses can commute, and which accesses might modify the data. In our case the Read accesses can only commute with other Read accesses, the Write accesses cannot commute with any other accesses, and the Add accesses can only commute with other Add accesses. As the Add accesses commutes with accesses that modifies the data (that is, because Adds commutes with Adds), they need exclusive access to the shared data structure to avoid data races. For this, we have a lock on each data, and this lock is only used for mutual exclusion between accesses that commutes with accesses that modifies the data, while all other types of accesses ignores this flag. If a task wants to access a data when another task has already received exlusive access to it, the task requests the data object to signal when the lock is released. When a task is signaled that the lock was released, it is reassigned to the worker that held the lock and added to the front of its ready queue to be the next task to be executed, as the data it needs was just used there.

In other applications one can imagine introducing for instance a Sort and a Sum access, so that the Sort and Sum accesses commute, as the result from summing an array is independent of the order of the elements in that array. The Sum accesses would then commute with any other accesses that only read the data, while the Sort access would only commute with the Sum access.

3.5 Calculating required versions

The required versions are calculated by storing a *scheduler version* for each access type in the datas. When a task is added, the required version for each access is taken from this scheduler version for the access, and the scheduler versions in the data are then updated so that the scheduler version for access types that commutes with the access are left untouched, and the others are increased to the next unused version number. It is not necessary to store scheduler versions for access types that do not commute with any other access types as those are always the latest version, and access types that commutes with all access types need not be considered at all as they do not introduce any dependencies.

3.6 Renaming

If all the inputs to a task are available but the output is not, the task can still be executed if its output is redirected to a temporary location. This technique is called *renaming* and avoids locking the output for longer than necessary, allowing several tasks that want to update a single memory address to run in parallel. When a task whose output has been redirected finishes, a new task is spawned which copies the computed results from the temporary output buffer to the real output memory location as soon as access to this memory is granted.

4 Results

4.1 Programming Model Design

The result of the programming model design is presented as a pseudo code example shown in Listing 1. The program in the example calculates the forces between all pairs of particles and moves the particles accordingly. An array `p` stores the n particles and an array `f` stores the forces acting upon the particles. These arrays are wrapped in objects of a class called `Data` that slices the array up info `numSlices` slices and keeps track of accesses to the slices.

Listing 1: Pseudo code example of an application that calculates the interaction between n particles and then moves the particles accordingly

```
Data<Particle> p(n, numSlices)
Data<Force> f(n, numSlices)

for (i=0; i<numSlices; ++i)
  addTask(zero(write(f[i])))
for (i=0; i<numSlices; ++i)
  for (j=i+1; j<numSlices; ++j)
    addTask(eval(read(p[i], p[j]), ...
                 add(f[i], f[j])))
for (i=0; i<numSlices; ++i)
  addTask(move(read(f[i]), write(p[i])))
```

The problem is divided up into three different task types. The first task type, `zero`, sets the forces acting upon the particles to zero. These tasks take no input but writes to a slice of the forces array. A write access to a slice of `f` is registered as it is passed as an argument to the task.

The next class of tasks is `eval` which evaluates the force between two slices of particles and updates the forces in `f` for both slices of particles. As it does not matter in which order the force array is updated, the write access is registered as Add. Before an `eval` task is run, the task library checks that both slices of particles in `p` are available, that the corresponding `zero` task is finished writing to the desired slice of `f`, and that no two `eval` tasks tries to update the same slice of `f` at the same time.

Finally, the `move` task reads the accumulated force acting on a slice of particles and moves the particles accordingly. It is run first when all `eval` tasks updating the force acting on the slice of particles are finished.

As can be seen in Listing 1, the extra notation that the user needs to add in order to run this application in parallel is quite unobtrusive. Note that the dependencies between the tasks are not expressed explicitly, but only implicitly indicated by the order in which the tasks are added. The programmer does not need to make sure that a task is ready to be executed before adding it to the task library, but the task library will make sure all the previously added tasks it depends on will finish first before executing the added task.

4.2 Performance

To test if the task library successfully takes advantage of the architecture, we have performed experiments on a small application that takes a 10 time steps in an n-body simulation. A time step is represented by a force evaluation, and an update of the particles positions and velocities. The application was executed and timed 10 times, and the mean time of these runs was taken. We also implemented a serial version that does not use tasks at all, and compared the two versions for different number of particles and different number of cores, as shown in Figure 1.

All tests were run on a dedicated server with two Quad-core Intel®Xeon 5520 (Nehalem 2.26 GHz, 8MB cache) processors giving a total of 8 cores.

As can be seen in Figure 1, the computations must not be too small in order to achieve good performance. With only 100 particles, the parallel version is slower than the serial version when running on more than one core. In order to have good speed-up, at least 1000 particles are needed.

We have also plotted the scheduling of the tasks over the processor cores, as is shown in Figure 2. In this figure, five time steps are executed with 1000 particles. The green boxes represents force evaluations between particles in the same slice of the particle array, and the blue boxes are force evaluations between different slices. The blue boxes are about 500 µs each which is about the granularity required in order for the parallelization overhead to be comparatively small. The vertical lines in the left of the figure are tasks that zeros the force array and are only about 2 µs each. The reason there is a delay after the first zeroing

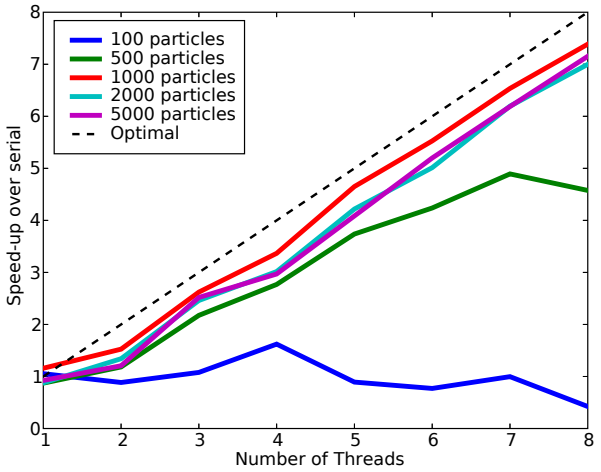

Figure 1: Speed-up as compared to the serial implementation.

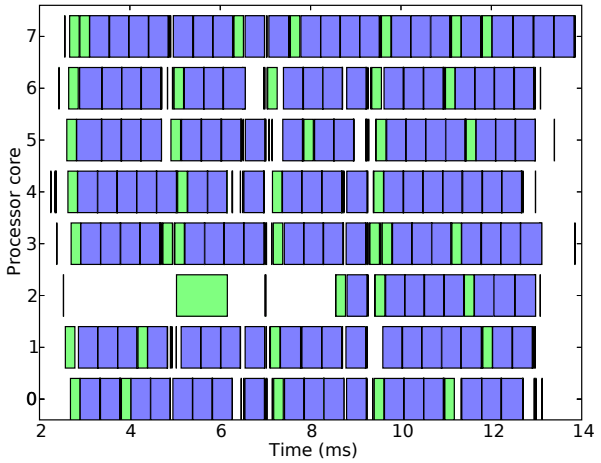

Figure 2: The scheduling of the tasks over the processor cores. Shown in this figure is one time step with 1000 particles.

tasks and that they appear to run sequentially rather than in parallel is that the manager has not yet had time to create more tasks. The vertical lines visible to the right are tasks that move the particles according to the calculated forces. There are also tasks that write renamed output from force evaluation tasks back to the real output location, which are only barely visible as thicker boundary edges on the larger tasks.

In the execution shown in Figure 2, core number 2 is delayed and most of its work is distributed among the other threads. This is because the manager happens to be scheduled to this core by the operating system, so the core is initially busy creating and adding the tasks.

5 Conclusions and Future Work

A task-based programming model is a good base for building higher level abstractions and allows the flexibility needed for attaining good performance. Letting the task library handle the dependencies and scheduling of tasks makes the programmers job much easier than if this has to be coded explicitly. As shown i Listing 1, the extra programming needed for task management is not too obtrusive.

The performance of the task library scales reasonable well as long as the task are not too fine grained, as seen in Figure 1. In the case of the small problem with only 100 particles, the parallelization overhead dominates and the parallel version is slower than the serial when its run on more than one core. For reasonable scaling, 1000 or more particles are needed, which corresponds to task sizes of about 500 µs.

When a core is unexpectedly delayed, as core number 2 in Figure 2, the task library successfully adapts and the work is shared among the other cores. This adaptiveness is valuable both when the software is executed on a system that is shared with other running processes, and also when the amount of work in a task cannot easily be estimated.

The next step from here is to explore how well recursive algorithms such as building tree structures can be expressed in this programming model. A possible case study is to organize the particles in a tree structure and use this to group distant particles together so that they can be approximated as one single large particle.

We will also optimize the task library to lower the parallelization overhead and achieve better performance for fine-grained tasks.

References

[1] G. F. Fasshauer. *Meshfree Approximation Methods with MATLAB*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2007.

[2] K.-F. Faxén. Wool–a work stealing library. *SIGARCH Comput. Archit. News*, 36(5):93–100, 2008.

[3] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[4] J. D. Hogg. A DAG-based parallel Cholesky factorization for multicore systems. Technical report, Computational Science and Engineering Department, Rutherford Appleton Laboratory, 2008.

[5] J. D. Hogg, J. K. Reid, and J. A. Scott. A DAG-based sparse Cholesky solver for multicore architectures. Technical report, Computational Science and Engineering Department, Rutherford Appleton Laboratory, 2009.

[6] J. Kurzak and J. Dongarra. Fully dynamic scheduler for numerical computing on multicore processors – LAPACK working note 220, 2009.

[7] OpenMP 3.0. `http://www.openmp.org`.

[8] J. M. Perez, R. M. Badia, and J. Labarta. A flexible and portable programming model for SMP and multi-cores. Technical report, BSC-UPC, 2007.

[9] Intel® Threading Building Blocks 2.2 for open source. `http://www.threadingbuildingblocks.org/`.