Scaling FMM with Data-Driven OpenMP Tasks on Multicore Architectures

Abdelhalim Amer[1], Satoshi Matsuoka[2], Miquel Pericàs[3], Naoya Maruyama[4], Kenjiro Taura[5], Rio Yokota[2], and Pavan Balaji[1]

[1] Argonne National Laboratory, Lemont, IL 60439, USA
[2] Tokyo Institute of Technology, Tokyo 152-8550, Japan
[3] Chalmers University of Technology, SE-412 96 Gothenburg, Sweden
[4] RIKEN Advanced Institute of Computational Science, Hyogo 650-0047, Japan
[5] University of Tokyo, Tokyo 113-0033, Japan

Abstract. Poor scalability on parallel architectures can be attributed to several factors, among which *idle times*, *data movement*, and *runtime overhead* are predominant. Conventional parallel loops and nested parallelism have proved successful for regular computational patterns. For more complex and irregular cases, however, these methods often perform poorly because they consider only a subset of these costs. Although data-driven methods are gaining popularity for efficiently utilizing computational cores, their data movement and runtime costs can be prohibitive for highly dynamic and irregular algorithms, such as fast multipole methods (FMMs). Furthermore, loop tiling, a technique that promotes data locality and has been successful for regular parallel methods, has received little attention in the context of dynamic and irregular parallelism.
We present a method to exploit loop tiling in data-driven parallel methods. Here, we specify a methodology to spawn work units characterized by a high data locality potential. Work units operate on tiled computational patterns and serve as building blocks in an OpenMP task-based data-driven execution. In particular, by the adjusting work unit granularity, idle times and runtime overheads are also taken into account. We apply this method to a popular FMM implementation and show that, with careful tuning, the new method outperforms existing parallel-loop and user-level thread-based implementations by up to fourfold on 48 cores.

1 Introduction

The technology trend of increasing core densities and deepening memory hierarchies in high-end processor packages is exacerbating the difficulty of harnessing their computational power. Higher core counts imply that applications have to expose more concurrent work in order to feed the computational units. Moreover, the performance of the memory subsystem is not keeping up with the core density. Consequently, pressure on the memory subsystem (e.g., caches, interconnects) and the distances that remote data has to traverse are increasing, adding to the existing CPU-memory performance gap.

A parallel execution can be formulated as a dynamic scheduling optimization problem. The goal is to minimize the makespan of a schedule of work units (*tasks, loop chunks*) on a set of computational cores. Unfortunately, finding an optimal schedule has proved to be an NP-complete problem even in the simple case of static scheduling, two processors, and one or two time units for task weights [11]. In practice, the model that dictates parallel execution (e.g., bulk-synchronous, data-driven) affects the resulting schedule and does not guarantee optimality.

The effectiveness of the resulting schedule is often quantified in terms of *parallel efficiency* relative to a sequential execution. Recent literature has shown that loss in parallel efficiency can be attributed to three primary factors: *idle times*, *data movement*, and *runtime overhead*[6] [7,9]. In order to mitigate these costs, parallel algorithms must expose sufficient parallelism (i.e., *parallel slackness*) and reduce data movement while keeping the runtime overhead low. Conventional parallel loops and nested parallelism have proved successful for regular computational patterns. For more complex and irregular cases, however, such methods often perform poorly because they take into account only a subset of these factors. For instance, bulk-synchronous approaches, exemplified by parallel loops, can suffer from underutilization of resources because of insufficient parallel slackness within or across computational steps. Data-driven methods can maximize resource utilization but often suffer from poor data locality (e.g., cache thrashing) and costly task management.

The difficulty of reducing data-movement costs in data-driven methods is often caused by high degrees of parallel slackness and poor data-locality incentives. Higher parallel slackness implies that the complexity of a proper work-unit-to-core mapping by the underlying runtime increases as well. Combined with poor data locality incentives, runtimes often operate with greedy heuristics (e.g. work-first and work-stealing policies) and thus execute work units in a data-locality-oblivious manner. Consequently, the resulting mapping can exhibit little data reuse and often causes substantial cache thrashing. Although some incentives have been proposed, such as hierarchical place trees [12], they remedy only part of the issue (e.g., reduction in remote memory accesses) and do not provide sufficient data reuse. On the other hand, loop tiling, a technique that promotes data locality and has been successful for regular parallelism, has seen little application in the context of data-driven and asynchronous tasking. In particular, it is arduous for these methods to exploit loop tiling within or across work units because of their fine-grained nature.

In this work, we investigate using OpenMP for a highly irregular fast multipole method (FMM) implementation. We choose FMM for its rich set of heterogeneous computational kernels and its complex dependencies that stress parallel efficiency. Furthermore, FMM input parameters allow us to control computation and synchronization requirements to help generalize our results. We propose a methodology to generate work units inherently suitable for data locality and amenable for granularity tuning to control the degree of parallel slackness and management overhead, thus taking into account all the primary factors that in-

[6] The time spent managing work units (e.g., creation, destruction, and scheduling)

fluence parallel efficiency. Specifically, work units operate on input-output data in tiled computational patterns inspired by cache-blocking techniques. Since tile sizes correlate with task granularity, they are exposed as tuning parameters. This method was applied in the context of an OpenMP task-based data-driven implementation that relies on the `task` construct to expose parallelism and the `depend` clause to express data dependence.

Results after applying this method to the kernel-independent FMM (KIFMM) of Ying et al. [13] showed substantial scalability improvements over existing parallel loop and user-level thread-based implementations, where up to fourfold improvements have been observed on 48 cores. Furthermore, we show that the tuned parameter values can be portable for several other input problems except when the parallel slackness is severely hindered, such as with small problem sizes and large tasks. We also show the limits of our data-locality optimization on a heavy cache-coherent non-uniform memory access (ccNUMA) machine. These results indicate that tiling to improve temporal and spacial data locality needs to be combined with NUMA awareness in order to further reduce data movement costs.

2 Related Work

Although scheduling work on parallel machines has long been studied, its NP-completeness and the rapid growth in scale and complexity of the parallel computing landscape make it an important and open research topic. We discuss here recent work and the perspectives from which the researchers tackle the problem of parallel efficiency loss.

Tasirlar and Sarkar introduced the implementation of data-driven tasks as an extension to the *async-finish* model to allow arbitrary runtime task graphs execution [10]. The authors focused mostly on the syntax and semantics of the model, however, with little attention paid to data locality. Yan et al. abstracted the memory hierarchy using a hierarchical place trees (HPT) model [12]. However, HPT is not flexible enough to express arbitrary dependencies between tasks and hence may result in a lack in parallel slackness. Furthermore, their data locality incentive through the concept of *places* is weak, does not exploit spatial locality, and does not tackle cache-thrashing issues. Olivier et al. explored the concept of *locality domains*, similar to *places*, by extending OpenMP with runtime routines that allow users to implement locality-aware divide-and-conquer algorithms [7]. This approach, however, shares the same data-locality issues as do *places*. In addition, the study focused on loop parallelism and divide-and-conquer task-parallel algorithms that are prone to parallel slackness issues.

Data-driven methods have been used (e.g., [5,8]) to tackle the irregular nature of FMM. The FMM implementations used by these works did not exhibit data-locality issues, however, and work focused mostly on parallel slackness. In prior work we also characterized FMM implementations that exhibited opposing trade-offs: a bulk-synchronous that suffered mostly from idle times and a fine-grained data-driven implementation that was losing more on data locality [2]. Here, we

strive for a better balance between the different trade-offs by exploiting tiling in a data-driven execution. A preliminary description of this method was introduced in the doctoral dissertation of the lead author [1]. Here, we provide a more in-depth description, analysis, and evaluation of the method.

3 About the FMM Case Study

The fast multipole method is a technique developed to accelerate solving N-body problems. The challenge is to evaluate pairwise interactions between N bodies. A direct computation results in an $O(N^2)$ complexity, which makes it expensive for large problem sizes. FMM was proposed as a fast solution that uses a rapidly convergent method and has a $O(N)$ complexity [4].

In this work we use the kernel-independent FMM variant developed by Ying et al., which relies only on kernel evaluations and extends FMMs to a wider range of engineering and scientific problems [13]. KIFMM operates on three-dimensional domains containing the target simulation bodies. The domain is hierarchically decomposed into smaller boxes, or cells, with each box containing a maximum of q bodies, where q is an input parameter and the hierarchy of boxes forms an octree. Computation of the effect of bodies in a source box on other bodies in a target box depends on the proximity between the two boxes. For close boxes, direct computation is employed; for far boxes, a multistep far-field approximation is used instead. In KIFMM, proximity between boxes is represented by a set of interaction lists computed following Greengard notation [4]: U, V, X and, W. KIFMM implements the force evaluation through the following steps: U-list, Upward, V-list, W-list, X-list, and Downward. We distinguish two independent flows of computation: the near field *direct evaluation*, which is represented by the U-list computation, and the *far-field approximation*, which starts from the Upward step, proceeds with the V-list, W-list, and X-list computations, and finishes with the Downward step.

The performance of KIFMM is highly dependent on the balance between its kernels. The reason is the heterogeneity between them in terms of arithmetic intensity, data accesses, and synchronization.[7] This balance depends on the density and pattern of the body distribution and the maximum number of bodies allowed in each cells, q. Large values of q result in small trees and converge KIFMM toward a direct $O(N^2)$ complexity, which is expensive even on modern hardware. Smaller values lead to larger trees, where most of the computation is performed by the far-field approximation, a step dominated by the memory-intensive V-list kernel and heavy synchronizations. This balance property allows us to simulate application runs in different regimes and thus to generalize our findings. Consequently, the primary challenge here is to ensure scalable performance in all regimes. In the following, we present the existing thread-level parallelization strategies that will serve as baselines.

[7] For example, U-list kernels are compute intensive; V-list ones are memory intensive and incur sparse memory accesses.

```
void* V-list-step () {
  #pragma omp parallel for schedule(OMP_SCHED)
  for(trg=0; trg < trgNodeMax; trg++)   //Traverse all target nodes
     for(src in  Vlist(trg))            //Accumulate the contribution of all
        compute_V(trg,src);             //source nodes into the target
}
```

Fig. 1. OpenMP parallel loop of the V-list step. The `OMP_SCHED` macro controls the scheduling policy and takes the values `static` or `dynamic` (`chunk_size` defaults to 1).

Bulk-synchronous with OpenMP: This is a highly optimized implementation for multicore architectures [3]. All steps rely on OpenMP work-sharing constructs to parallelize the work on the target nodes of the octree. In particular, the Upward and Downward steps ensure parent-children dependencies through OpenMP barriers. To better expose the trade-off between parallel slackness and data-locality, we explored two opposing scheduling policies to implement all phases: (1) a static approach that divides the target node list equally among the threads without taking into account the workload variation and (2) a dynamic approach that distributes dynamically the workload. Figure 1 shows how this is applied to the V-list step, where `OMP_SCHED` controls the scheduling policy.

Fine-grained data-driven with lightweight threads: In this implementation, tasks operate at the tree node granularity and are spawned as lightweight threads using the MassiveThreads library [6]. That is, the flow of execution goes from source to the target boxes, where the far-field and direct evaluation computations are merged into a single flow by starting the Upward step and the direct evaluation at the same time. Here, fine-grained synchronization is required in order to respect data dependencies. Tasks are created only when their dependencies are satisfied; dependency tracking is achieved through fork-join control flows and atomic counters. Figure 2 gives an example of how a V-list task is executed for a source cell (`src`) after being called by an `Up` task (see [2] for more details about this implementation). Our analysis showed that the way the tasks are spawned in this data-driven implementation generates subtree working sets that have a positive data locality impact on the Upward and Downward steps, but it does not help with the communication-intensive V-list computation. Despite exposing massive parallel slackness, this implementation scales poorly for data-locality-sensitive scenarios.

4 Tasking through Temporal and Spatial Blocking

The difficulty of reconciling the factors affecting parallel efficiency lies in their orthogonal effect. Reducing idle times requires high degrees of parallel slackness and dynamic scheduling. The primary goal of dynamic scheduling is to balance work across computational units and is often data locality agnostic. This results

```
void* V (src) {
  for(trg in Vlist(src)) {           //Compute the contribution of src
     compute_V(trg,src);             //to all dependent target cells
     trg.down_counter++;             //Atomic increment of the sync counter
     if(trg.down_counter = in_depend(trg))  //all dependencies satisfied ?
           create_task(Down, trg); } //Create Down computation task
}
```

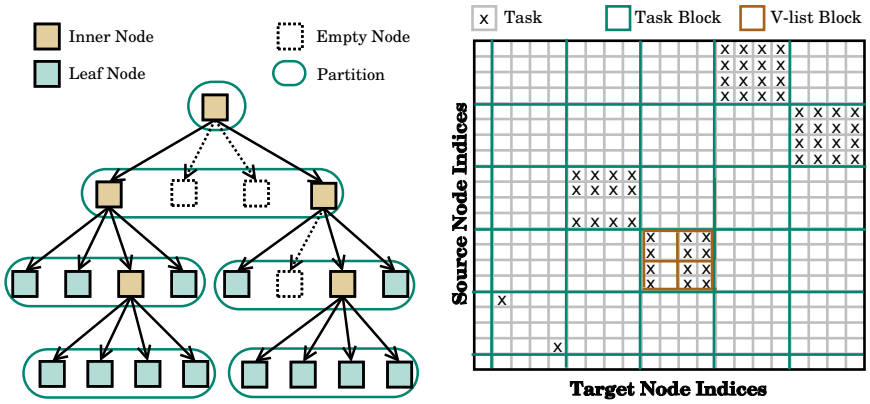
Fig. 2. V-list computation in a fine-grained data-driven implementation. The `create_task` function is a generic wrapper around lightweight thread creation.

in poor scheduling decisions that cause more cache thrashing from heterogeneous working sets than would a more homogeneous static scheduling approach. Furthermore, work unit management costs is higher at fine-grained levels. The previously described dynamic bulk-synchronous and fine-grained data-driven approaches suffer from this problem because they operate[8] without appropriate data locality incentives. In this section, we present a method that retains the advantages of dynamic scheduling while taking into account data locality and work unit granularity in order to lower data movement and work unit management costs. This method was primarily designed for data-driven implementations, but was also applied to parallel loops for later comparison.

In KIFMM, steps operate on objects that are arrays of basic elements. Each element is a data structure that encapsulates information required by a subset of the computational steps at the *tree node granularity*. The access pattern of each computational step can be modeled as a sparse matrix whose dimension depends on the number of objects manipulated. In the bulk-synchronous case, basic elements are written individually (dynamic scheduling) or in blocks (static scheduling), but reading is sparse and depends on the interaction lists. The lightweight-threads implementation has a similar pattern but is source centric; reads are individual, and writes are sparse. The sparsity of the memory access pattern is a major issue especially for the memory-intensive V-list step.

To improve the data locality of such sparse memory accesses, we present a partitioning scheme where tasks operate on every object in blocks of basic elements (both reads and writes). The resulting partitioning is a multidimensional tiling that clusters computational patterns that operate on contiguous data and exhibit high temporal and spatial locality. This is similar to existing cache-blocking techniques found in linear algebra optimizations. In KIFMM, however, we target the high levels of the memory hierarchy (e.g., last-level cache) because operations on tree nodes operate on larger data. For instance, an operation between two tree nodes can be composed of matrix multiplications or FFT transformations. These computations are carried out by external libraries and are often well optimized for lower-level caches. Figure 3a shows a quad-tree for a two dimensional domain where partitions are aligned to tree levels in order to avoid complex dependencies. The resulting partitions serve as the tile size for one data object. Considering a computation step that operates on two data

[8] Loop chunks or lightweight threads operate at the level of a single octree node.



(a) Box partitions following the quad-tree (b) Applying the partitions to the V-list step

Fig. 3. Partitioning example for a two dimensional domain (resulting in a quad-tree): (a) partition ranges within tree level boundaries to reduce unnecessary dependencies; (b) example of V-list interaction pattern between source and target boxes. The same partitioning scheme in (a) is applied to achieve two-dimensional tiles. A secondary tiling level is applied within work units for V-list.

```
void* V-list-step (){
  #pragma omp parallel for schedule(dynamic)
  for(i=0; i < trgNodeMax; i+=BS)       //Traverse all target blocks
    for(j=0; j < srcNodeMax; j+=BS)     //Traverse all source blocks
      for(trg=i; trg < BS; trg++)       //Traverse the targets in the block
        for(src=j; src < BS; src++)     //Traverse the sources in the block
          if(src in  Vlist(trg))        //Accumulate the contribution of all
            compute_V(trg,src);         //source nodes into the target
}
```

Fig. 4. Example of a bulk-synchronous tiled V-list computation with dynamic scheduling. `BS` denotes the block size.

objects, the resulting tiling is two dimensional. Figure 3b illustrates how the data is partitioned for the V-list step. In the following We use the same tile size for the tasks of all steps to simplify dependency tracking and tuning. Exploring different tile sizes for the various steps is left as a future work. We present below how the tiling method is applied for both to parallel loops and to data-driven tasks.

Tiled bulk-synchronous: We applied the previous tiling method to the bulk-synchronous method. Figure 4 illustrates how the V-list phase is implemented with OpenMP work-sharing constructs and a blocking factor (`BS`). Our implementation breaks the outer loop manually, but the same result could have been achieved by using the OpenMP `chunk` parameter.

```
#define DATA_OUT eff_val[beg_eval:trg_stride]
#define DATA_IN  eff_den[beg_eden:src_stride]
void* V-list-step (){
  for(i=0; i < trgNodeMax; i+=BS) {    //Traverse all target blocks
    int trg_stride = eff_trg_size*BS, beg_eval = trg_stride*i;
    for(j=0; j < srcNodeMax; j+=BS) {  //Traverse all source blocks
      int src_stride = eff_src_size*BS, beg_eden = src_stride*j;
      #pragma omp parallel task depend(out: DATA_OUT) depend(in: DATA_IN)
      for(n=i; n < i+BS; n+=VBS)           //Traverse the target V-list blocks
        for(m=j; m < j+BS; m+=VBS)         //Traverse the source V-list blocks
          for(trg=n; trg < n+VBS; trg++)   //Traverse targets in a V-list block
            for(src=m; src < m+VBS; src++) //Traverse sources in a V-list block
              if(src in  Vlist(trg))       //Accumulate the contribution of all
                compute_V(trg,src);     }} //source nodes into the target
}
```

Fig. 5. V-list computation in the tiled data-driven implementation using OpenMP tasks. Here, `eff_val` and `eff_den` are the input and output vectors, respectively. `BS` and `VBS` are the maximum task and V-list tile sizes, respectively.

Tiled data-driven with OpenMP: Task-dependency tracking was introduced in OpenMP 4.0 through the `depend` clause of the `task` construct. This clause takes as arguments the input-output storage locations, which can be scalar variables or arrays sections. In KIFMM, specifying individual array elements as dependencies is impractical because of the task management overhead. Passing the sparse storage locations directly (e.g., as a linked list) is not possible because the `depend` clause accepts only scalar variables and array sections. Thus, we express those dependencies conservatively by using array sections. This approach expresses more dependencies than necessary but incurs less dependency tracking overhead with large array sections. We then apply the tiling method by mapping tiles or blocks to array sections. In addition, we expose another blocking factor for the V-list step because communication-intensive kernels often perform worse in a data-driven execution as a result of cache thrashing from sharing cache with tasks that operate on different data [9]. Figure 3b illustrates how the data is partitioned in KIFMM with particular attention to the V-list blocks. Figure 5 shows how we implemented the V-list phase. We observe that this method is relatively simple to implement with OpenMP. The resulting manual tiling, however, makes the code less readable. We believe directive extensions to OpenMP to express tiled algorithms would be beneficial. For instance, the `taskloop` construct could be extended with tiling clauses. The rest of the stages are implemented similarly to allow a full data-driven execution.

Tuning method: One of the limits of tuning methods is the rapid growth of the design space with the number of parameters and the ranges of discrete values they can take. To reduce this complexity, we operate in several steps, starting from tuning single-threaded performance, then moving to tuning the

[9] Tiling with parallel loops achieves good data locality without this secondary tiling factor as will be shown in Section 5.1.

Table 1. Target machine specifications

	Sandy Bridge	Magny-Cours
Processor	Xeon E5-2670	Opteron 6172
CPU Frequency (Ghz)	2.6	2.1
# Sockets	2	4
# NUMA-Nodes	2	8
#Cores/NUMA-Nodes	8	6
L3 Cache size (MB)	20	6-1
Compiler	ICC 15.0.0	GCC 4.9.2

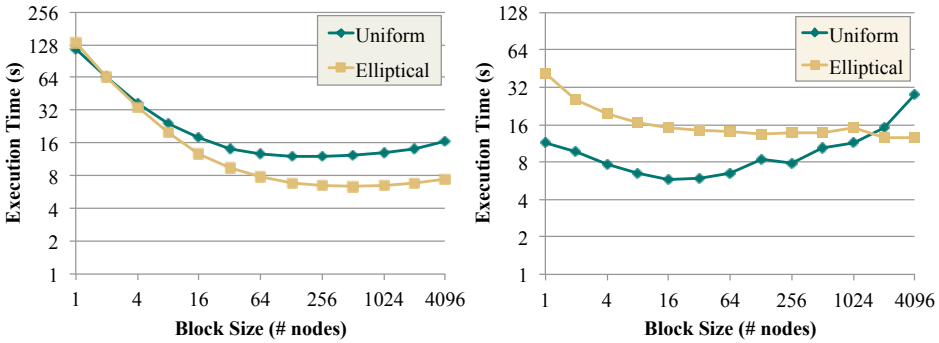
parallel data-driven implementation. Previously, with KIFMM, single-threaded performance was manually tuned [3]. We only tune q in single-threaded since it affects performance significantly across input problems and hardware specifications. The most important step here is to explore the design space of the task granularity and the V-list block size.

5 Characterization and Evaluation

We describe here a characterization study to identify the bottleneck sources of each implementation. We then present a performance scalability evaluation.

Experimental setup: We follow the same input problems as in [3]. That is, we simulate the evaluation of a single step where the bodies are spread following two distributions: a randomly uniform and an elliptical distribution. For the interaction that governs the physics, we use the Laplace kernel. We consider only double-precision computation because of its higher pressure on the memory subsystem, which we consider more insightful. For the target architectures we select representatives of ccNUMA multicore architectures, with a two-socket Intel Sandy Bridge and 8 four-socket NUMA nodes on an AMD Magny-Cours; detailed specifications are given in Table 1.

Tuning results: Tuning results for a uniform and an elliptical distribution on both target platforms are shown in Figure 6 for the bulk-synchronous approach and in Figure 7 for the data-driven approach. We observe that performance is highly variable depending on the task granularity and the V-list block size. In addition, we notice that the optimal parameters depend on the type of the input distribution and across hardware architectures. The portability of the tuned parameters with respect to the problem size and the parameter q is discussed further in a subsequent section. An important observation is the time explosion for small tasks when using an elliptical distribution. That inflation is due mostly to runtime overheads. Unfortunately, we do not have a quantitative measure of the runtime overhead, although we believe that this overhead is negligible for large enough tasks and that idle times and data movement dominate.



(a) Sandy Bridge (b) Magny-Cours

Fig. 6. Tuning the tiled bulk-synchronous implementation with 2^{22} bodies and $q = 128$ at full concurrency on each machine.

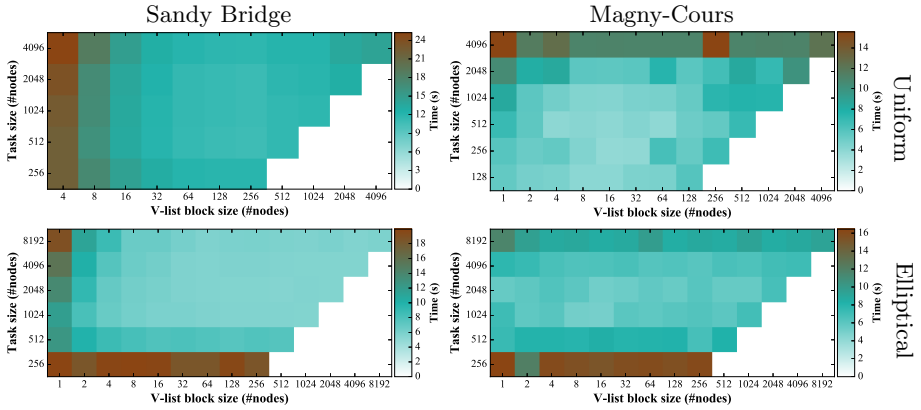


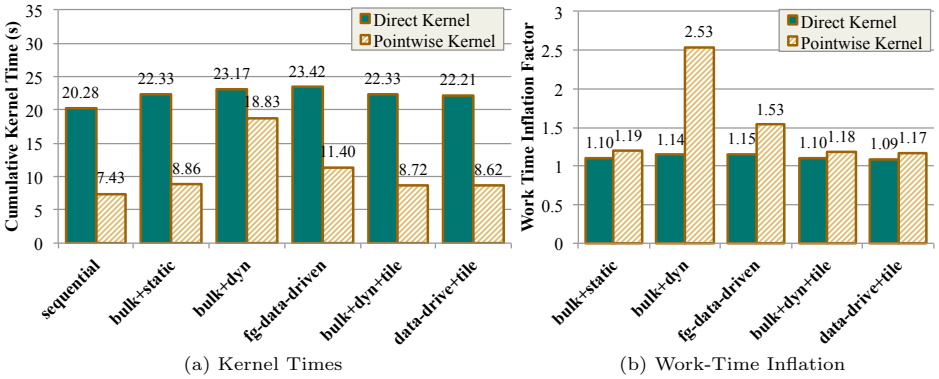
Fig. 7. Tuning the tiled data-driven implementation with 2^{22} bodies and $q = 128$ at fully concurrency on each machine.

5.1 Characterizing Data Locality and Idleness

Here we analyze all implementations under the same conditions and correlate the performance differences with idle times and data locality.

Data locality characterization: Assuming work units are atomic, that is, not susceptible to preemption,[10] we show in Figure 8a the cumulative execution time of all instances of two kernels that are atomic and contribute to most of the KIFMM work time. The *Direct* kernel is compute intensive and called by

[10] An atomic work unit executes to completion without interruption after being scheduled. In the context of parallel runtimes, such unit should not perform synchronization and scheduling operations, such as yielding execution.



(a) Kernel Times (b) Work-Time Inflation

Fig. 8. Data locality with an elliptical distribution, 2^{22} bodies, and $q = 128$ on Sandy Bridge.

all the higher-level kernels except V-list. This latter phase relies heavily on the memory-intensive *Pointwise* kernel. Compared with a sequential execution, kernel times with the parallel methods increase slightly for the compute-intensive kernel but can increase significantly for the communication-intensive kernel for some methods. Since the work units are atomic, runtime scheduling overheads and idle times should not affect these results. Thus, data movement is the primary factor that influences the variation of kernel times across the parallelization methods.

To better capture the inflation when running such atomic work units in parallel, we rely on the *work time inflation* metric, which measures the factor of the parallel execution time over a sequential execution for a given work unit [7,9]. We measured the work time inflation for the aforementioned kernels on the Sandy Bridge machine and show the results in Figure 8b. We notice that all methods have little inflation except the *Pointwise* kernel, which shows significant inflation in the case of the dynamic bulk-synchronous and the fine-grained data-driven implementations (2.53x and 1.53x inflation, respectively). Furthermore, we observe that the inflation was reduced substantially by the tiled implementations, incurring less than 1.20x inflation.

Idleness characterization: To characterize idleness, we manually instrumented the implementations to record the number of tasks running in parallel per interval of time. This metric indicates idle threads if the number of running tasks is less than the number of threads. The metric involves a sampling approach using the POSIX timer interface. Figure 9 shows the results with sample intervals of 5 milliseconds that ensure a low tracing overhead. We confirm that the data-driven approach exhibits the fewest idle threads among all approaches and that it offers a major advantage over the bulk-synchronous approach even after tuning the block sizes.

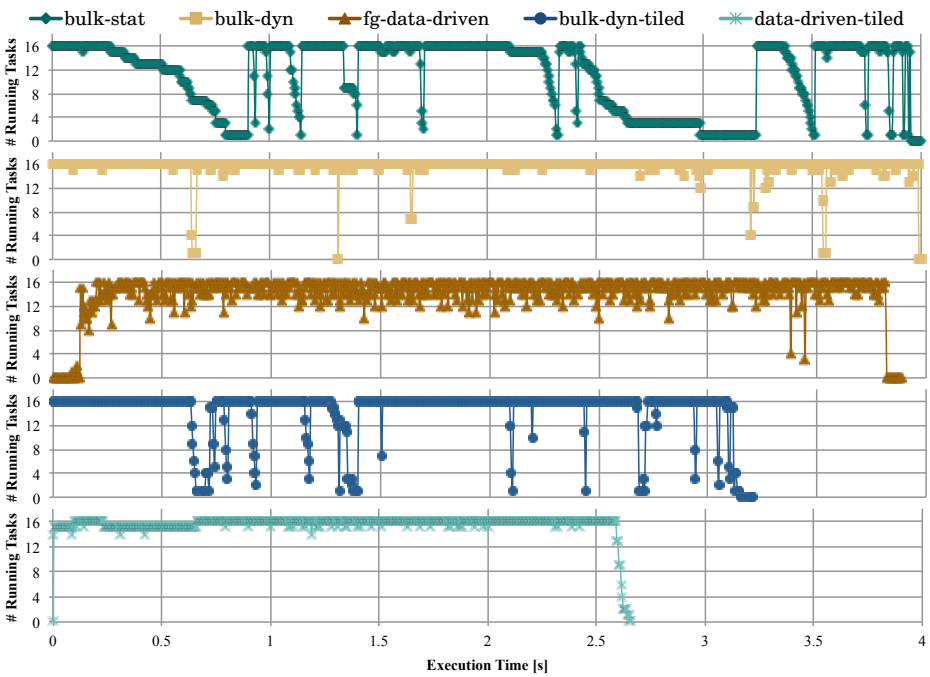


Fig. 9. Profiling idle time with an elliptical distribution and 2^{22} bodies and $q = 128$ on Sandy Bridge.

5.2 Performance Evaluation

The goals of this section are threefold: (1) evaluate all implementations in terms of scalability; (2) discuss the portability of the tuning parameters; and (3) correlate scalability and time to solution.

Scalability evaluation: Figure 10 shows scalability results with an input problem of 2^{22} bodies. We observe that the Sandy Bridge results reflect the previous characterization, where the tiled data-driven implementation is the most scalable, followed by the tiled bulk-synchronous method, because it suffers little work time inflation and idleness. This method also performs the best on Magny-Cours, although the parallel efficiency at full concurrency is not perfect. The reason is the data movement costs since the platform is a heavy ccNUMA machine and our data locality optimizations are not NUMA aware.

Tuning values portability: Here we fix the task block and V-list block sizes after tuning them for 2^{22} bodies in a uniform distribution and $q = 128$ on Sandy Bridge. We then vary the input problem while monitoring the speedup (Figure 11). We observe that the tiled implementations perform the best in most cases except for small problem sizes and large values of q. In this latter case, the

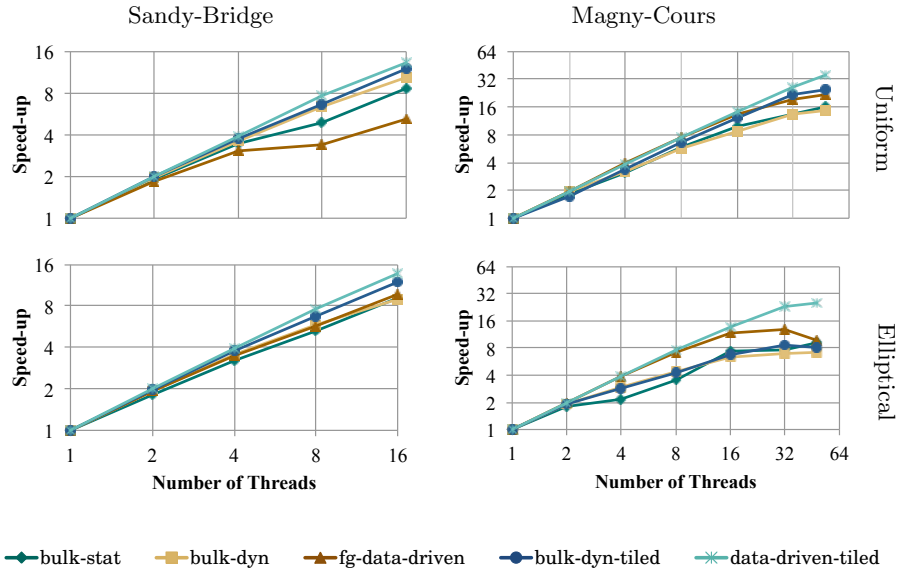


Fig. 10. Scalability evaluation with 2^{22} bodies and $q = 128$.

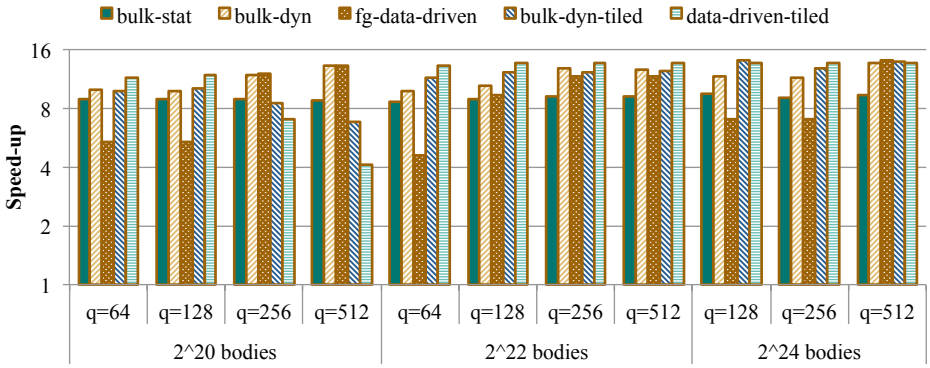


Fig. 11. Portability of the tuning parameters with an elliptical distribution on Sandy Bridge with respect to the input problem size and q.

resulting octree is small; and since our decomposition is performed at the tree node level, tasks are too coarse grained, and thus parallel slackness is severely hindered. Furthermore, large values of q imply a compute-intensive regime that does not benefit from our data locality optimizations.

Scalability vs. efficiency: Figure 11 also shows that the fine-grained dynamic implementations (bulk-synchronous and fine-grained data-driven) perform similar to or outperform the tiled implementations in compute-intensive regimes (i.e., large q). A misleading decision by application developers is to aim at generat-

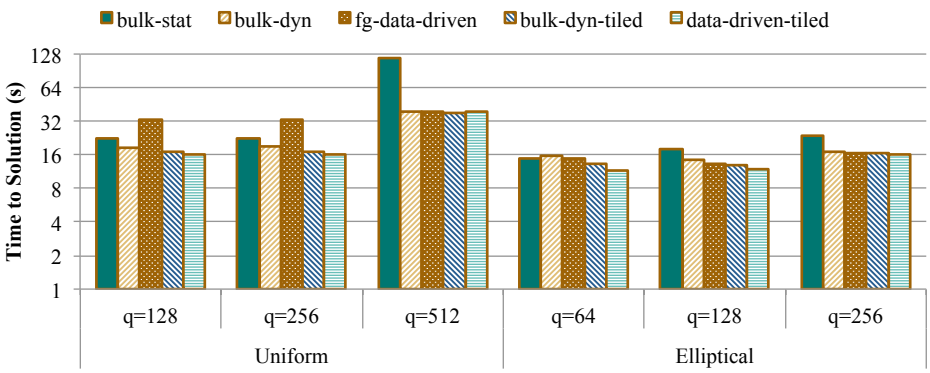


Fig. 12. Time to solution at full scale (16 cores) for an elliptical distribution with respect to the implementation method and the values of q for a large problem size (2^{24} bodies).

ing a large number of compute-intensive tasks and schedule them dynamically; the belief is that by having few memory-intensive tasks, the dynamic execution should be able to scale almost linearly. Scalability, however, does not necessarily mean optimal time to solution. Figure 12 shows the execution time necessary for one KIFMM iteration to solve a large problem. We note that using large values of q translates into worse performance, although Figure 11 exhibits almost linear scalability with all dynamic executions. The issue here is that a large ratio of compute-intensive tasks moves the complexity of the whole application toward $O(N^2)$, which is heavy and slow even on modern hardware. As a result, managing data locality in the presence of memory-intensive tasks can be more efficient. In addition, other applications might not have such balance between heterogeneous kernels and might have a more homogeneous communication-intensive nature and thus will require careful data locality optimizations.

6 Conclusion and Future Work

We presented in this paper a methodology to express parallelism while taking into account data locality through tiling computation patterns when using work-sharing and task constructs. The resulting tiled bulk-synchronous and data-driven implementations showed substantial improvement, with the data-driven method being the most scalable. In addition, the tuning parameters proved to be fairly portable except when parallel slackness was severely reduced.

From a programming model perspective, the tiled implementations required more extensive changes. One of the desirable feature that was missing was the ability to write tiled algorithms readily. From a performance perspective, we showed that optimizing for data locality not only requires spatial and temporal locality but also necessitates NUMA awareness in order to reduce expensive

remote data-movement. We plan to investigate methods of combining tiling abstractions with NUMA awareness, by using OpenMP places for instance.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357, and by JST, CREST (Research Areas: Advanced Core Technologies for Big Data Integration; Development of System Software Technologies for post-Peta Scale High Performance Computing).

References

1. Amer, A.: Parallelism, data movement, and synchronization in threading models on massively parallel systems. Tech. rep., Tokyo Institute of Technology, Department of Mathematical and Computing Sciences (2015)
2. Amer, A., Maruyama, N., Pericàs, M., Taura, K., Yokota, R., Matsuoka, S.: Fork-join and data-driven execution models on multi-core architectures: Case study of the FMM. In: Proceedings of the 2013 ACM/IEEE conference on Supercomputing. pp. 255–266. Springer (2013)
3. Chandramowlishwaran, A., Williams, S., Oliker, L., Lashuk, I., Biros, G., Vuduc, R.: Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS). pp. 1–12 (2010)
4. Greengard, L.: The Rapid Evaluation of Potential Fields in Particle Systems, vol. 52. MIT Press (1988)
5. Ltaief, H., Yokota, R.: Data-driven execution of fast multipole methods (2012)
6. Nakashima, J., Taura, K.: MassiveThreads: A thread library for high productivity languages. In: Concurrent Objects and Beyond, pp. 222–238. Springer (2014)
7. Olivier, S.L., De Supinski, B.R., Schulz, M., Prins, J.F.: Characterizing and mitigating work time inflation in task parallel programs. In: Proceedings of the 2012 ACM/IEEE conference on Supercomputing. pp. 1–12. IEEE (2012)
8. Pericas, M., Amer, A., Fukuda, K., Maruyama, N., Yokota, R., Matsuoka, S.: Towards a dataflow FMM using the OmpSs programming model. 136th IPSJ Conference on High Performance Computing (2012)
9. Pericàs, M., Amer, A., Taura, K., Matsuoka, S.: Analysis of data reuse in task-parallel runtimes. In: High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation, pp. 73–87. Springer (2014)
10. Tasirlar, S., Sarkar, V.: Data-driven tasks and their implementation. In: 2011 International Conference on Parallel Processing (ICPP). pp. 652–661 (2011)
11. Ullman, J.D.: *NP*-complete scheduling problems. Journal of Computer and System Sciences 10(3), 384–393 (1975)
12. Yan, Y., Zhao, J., Guo, Y., Sarkar, V.: Hierarchical place trees: A portable abstraction for task parallelism and data movement. In: Languages and Compilers for Parallel Computing, pp. 172–187. Springer (2010)
13. Ying, L., Biros, G., Zorin, D., Langston, H.: A new parallel kernel-independent fast multipole method. In: ACM/IEEE conference on Supercomputing. p. 14 (2003)