Dtree: Dynamic Task Scheduling at Petascale

Kiran Pamnany[1(✉)], Sanchit Misra[1], Vasimuddin Md.[2], Xing Liu[3], Edmond Chow[4], and Srinivas Aluru[4]

[1] Parallel Computing Lab, Intel Corporation, Bangalore, India
kiran.pamnany@intel.com
[2] Department of Computer Science and Engineering, Indian Institute of Technology Bombay, Mumbai, India
[3] IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
[4] School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, USA

Abstract. Irregular applications are challenging to scale on supercomputers due to the difficulty of balancing load across large numbers of nodes. This challenge is exacerbated by the increasing heterogeneity of modern supercomputers in which nodes often contain multiple processors and coprocessors operating at different speeds, and with differing core and thread counts. We present Dtree, a dynamic task scheduler designed to address this challenge. Dtree shows close to optimal results for a class of HPC applications, improving time-to-solution by achieving near-perfect load balance while consuming negligible resources. We demonstrate Dtree's effectiveness on up to 77,824 heterogeneous cores of the TACC Stampede supercomputer with two different petascale HPC applications: ParaBLe, which performs large-scale Bayesian network structure learning, and GTFock, which implements Fock matrix construction, an essential and expensive step in quantum chemistry codes. For ParaBLe, we show improved performance while eliminating the complexity of managing heterogeneity. For GTFock, we match the most recently published performance without using any application-specific optimizations for data access patterns (such as the task distribution design for communication reduction) that enabled that performance. We also show that Dtree can distribute from tens of thousands to hundreds of millions of irregular tasks across up to 1024 nodes with minimal overhead, while balancing load to within 2 % of optimal.

Keywords: Petascale · Dynamic scheduling · Load balance

1 Introduction

The scheduling challenge on modern supercomputers arises from the need for fine-grained parallelism in applications. Scaling to large numbers of processors requires that there be a large number of parallel tasks that can be distributed.

X. Liu—During this research, Xing Liu was affiliated with Georgia Institute of Technology.

J.M. Kunkel and T. Ludwig (Eds.): ISC High Performance 2015, LNCS 9137, pp. 122–138, 2015.
DOI: 10.1007/978-3-319-20119-1_10

This brings forth the need for efficient scheduling – to distribute tasks among all available processors so as to minimize total run time.

Applications that can be decomposed into equal-sized tasks can be scheduled simply by statically distributing the tasks evenly among the number of processors. This additionally simplifies the distribution of input data for tasks as well as routing for any required inter-task communication. For these reasons, this class of applications can be easily scaled to large supercomputers.

However, many applications exhibit irregular characteristics and require more sophisticated scheduling to prevent unbalanced computing load and the resulting unnecessary increase in total run time. Such applications vary widely, and a number of techniques have been proposed to address the diverse problems presented. We introduce our approach by classifying irregular applications according to the following characteristics:

1. Irregular tasks – individual tasks may vary in length.
2. Dynamic task pool – the set of tasks may be dynamic, i.e. new tasks may be added during application execution.
3. Locality – input and/or output data for a task may be large, i.e. the cost of movement must be considered.
4. Dependencies – there may be inter-task dependencies or communication.

There are important applications that display various combinations of one or more of these characteristics. While it is possible to design dynamic scheduling algorithms that cope with all of them, the resulting overhead may be needlessly large for applications that exhibit only few of the characteristics. Hence, many solutions in the literature focus on particular characteristics, the most common being the data locality problem.

Devine *et al.* [11] address load balancing entirely in the context of data partitioning. Menon and Kalé [20] target iterative applications with a synchronous load balancer, measuring load information with a gossip protocol and running sender-initiated load transfer at application synchronization points. Zheng *et al.* [27] also balance load periodically, but use a hierarchy of load balancing domains. These solutions share a theme in being measurement-based – which nodes are overloaded and which underutilized must be determined before load balancing decisions can be taken.

Other solutions target applications using recursive parallelism, e.g. combinatorial search or divide-and-conquer, in which the task pool is dynamic. Work stealing [8] is the dominant technique in this space [12,19] but generalized approaches can struggle with scaling beyond 6K processors due to large increases in failed steals [12]. Lifflander *et al.* have scaled a specialized work stealing approach for iterative applications to 163K cores [16]. Guo *et al.* use locality hints to improve performance and explore scheduling policies in their work stealing scheduler, SLAW [13].

Min *et al.* have proposed a topology-aware hierarchical work stealing strategy [21]; their implementation is in UPC [2]. Saraswat *et al.* extend work stealing with work sharing over lifeline graphs [24]; their system is implemented in X10 [9].

All work stealing approaches share the problem of distributed termination detection – if a node repeatedly fails to steal work, it cannot conclude that there is no work left in the system; other means of detecting quiescence must be used.

Numerous scheduling algorithms exist for applications that model inter-task dependencies as static task graphs [15]. Dependencies in applications that have dynamic task pools may be expressed as dynamic task additions and scheduled via work stealing.

For applications that do not have a data locality problem, applying a technique designed to minimize that problem is inefficient or unfeasible. Similarly, applications with static task pools cannot benefit from a technique intended to facilitate the easy addition of dynamically created tasks.

Our scheduler, Dtree, uses work sharing rather than work stealing, and further differs from other approaches in focusing on applications with independent irregular tasks in a static task pool. We decouple the task scheduling problem from the data distribution problem, focusing on balancing load by distributing tasks with minimal overhead. This allows us to demonstrate near-optimal load balance for an application that does not require data distribution. Furthermore, for an application that does require data distribution, the high efficiency of our scheduler allows us to discard application-specific optimizations designed to reduce data movement and still match the performance.

Many other approaches impose a particular programming model or framework on the application. We target the widely used and familiar hybrid MPI+X programming model (although the applications considered in this paper use OpenMP®, Dtree is agnostic to the shared memory parallelism model).

Finally, Dtree enables full utilization of heterogeneous nodes containing Intel® Xeon® processors as well as Intel® Xeon Phi™ coprocessors[1], balancing load across the different processors transparently and thereby eliminating the need for applications to manage such heterogeneity. The potential for this has been discussed in the literature, but to the best of our knowledge, ours is the first scalable solution showing experimental results.

We briefly introduced an early version of Dtree in the context of a specific application in previous work [22]. We have since significantly improved Dtree's performance and flexibility, extended it to support manycore processors and heterogeneous clusters, and evaluated it in depth, both with another large application, and with a micro-benchmark designed to find its limitations.

The remainder of this paper is organized as follows: we provide an in-depth description of Dtree in Sect. 2, followed by an analysis of Dtree's performance using a micro-benchmark to simulate a variety of workloads in Sect. 3. Sections 4 and 5 detail how we have used Dtree with a machine learning application and a quantum chemistry application, respectively, and present experimental results. We summarize and conclude in Sect. 6.

[1] Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

1.1 Experimental Setup

We performed all our experiments on the TACC Stampede supercomputer [3]. The Stampede nodes used in our experiments each contain two Intel Xeon E5-2680 at 2.7 Ghz, one Intel Xeon Phi SE10P coprocessor at 1.09 GHz, and 32 GB of DRAM. We compiled and ran our code using Intel Composer XE 2013 and either Intel MPI Library 4.1 or MVAPICH2 2.0b, depending on the application.

2 Dtree

A fundamental step in scaling an application is determining how work will be decomposed into a large number of independent tasks. Dtree addresses the challenge of scheduling these tasks across all available processors so as to minimize runtime.

2.1 Overview

Task Model. We follow Dinan *et al.* [12] in referring to the set of tasks as a *task pool* and requiring that all tasks in the pool are independent and able to execute till completion without blocking or waiting for results produced by any concurrently executing tasks. We further specify that a task pool may be dynamic and allow task addition during program execution, or static and contain the entire set of tasks to be completed at program start. Our approach targets static task pools.

It is possible, with a static task pool, to assign numbers to the tasks to arrange them in a total order. These *task IDs* allow a compute node to independently and uniformly identify the task, given the task ID. Dtree is agnostic to application task structure as it schedules tasks by distributing task IDs.

Task Characteristics. From the perspective of scheduling, the typical length of a task and the variance in task lengths are key characteristics of a task pool. A large number of short tasks and a smaller number of long tasks present quite different scheduling challenges. Similarly, tasks with high variance in length are difficult to balance whereas for tasks with low variance in length, the challenge is to outperform a static schedule.

Dtree's design, described in the following sub-section, addresses these challenges effectively. We validate these claims in Sect. 3.

2.2 Design

Distribution Tree. A centralized task distribution scheme offers excellent performance [7], but can face scalability problems due to the central node becoming a bottleneck. We address this with the well-known technique of arranging the nodes into a tree.

Figure 1 illustrates our approach, showing an example Dtree for 32 nodes. We arrange the tree to maximize the number of leaf nodes. The *fan-out* of the

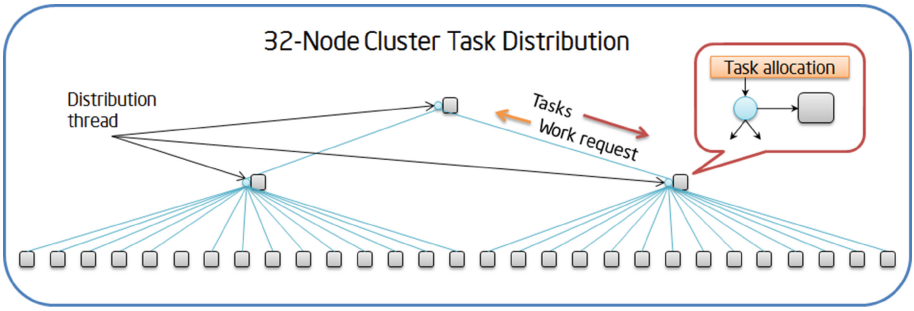


Fig. 1. Task distribution with an example Dtree for a 32 node cluster using a fan-out of 16. Observe that leaf nodes do not require a distribution thread.

tree, 16 in the example, is the maximum number of children for which a single node can be a parent, and is a function of the communication cost between a child node and a parent node. The protocol used by Dtree for task distribution is designed to minimize this communication cost, and thereby maximize the fan-out of the distribution tree.

Task Distribution. Dtree uses receiver-initiated task distribution – a node requests work from its parent in the tree and a parent node responds to requests from its children. As a parent node must listen for requests from its children, Dtree requires one thread in each parent node. All other threads in a parent node may be used for computation.

As Dtree distributes task IDs, the application need only specify T, the total number of tasks to be distributed. An important component of Dtree's efficiency is that tasks are distributed in groups identified by a range of task IDs, i.e. a start and end task ID. Initially, the root node alone holds a single task group comprised of tasks IDs 0 to $T - 1$.

Distribution Cost. A child node's request for work from its parent node takes the form of an 8-byte message. A parent node's response is a task group, sent in a 16-byte message. Thus, a request/response pair consists of two small messages.

Dtree is built on top of MPI. Running over a typical Infiniband interconnect, point-to-point MPI small messages see a one-way latency of approximately 1.31 ms, which equates to over 3 million messages per second [6]. In practice, this allows us to use a very large fan-out for the distribution tree, which minimizes the number of parent nodes required and consequently, the number of threads required by the scheduler.

Task Allocation. At program start, a node initializes its Dtree with an *init-work()* call. This returns a task group which is allocated as follows: the Dtree root node distributes $d_f \times T$ tasks to its children. All parent nodes in the tree apply the same strategy which leaves a buffer of tasks with each parent.

The number of tasks allocated to a child node is computed on the basis of the number of nodes in the child's sub-tree. In the example Dtree in Fig. 1, the left side sub-tree has 15 leaf nodes while the right side sub-tree has 14 leaf nodes. Parent nodes also work, and must be counted as part of their sub-trees for allocation purposes. Thus the left side intermediate parent node apportions 1/16 of its task allocation to each of its children and to itself. The root node must consider itself and its 2 children, which are parents themselves. Thus it computes *distribution fractions* of 1/32, 16/32, and 15/32 respectively.

When a node completes executing its initial allocation of tasks, it requests additional work from its Dtree using a *getwork()* call. This request is satisfied by applying the child's distribution fraction to the tasks remaining in the buffer. The resulting number is scaled by d_r, which controls the task drain rate, and subjected to a minimum threshold, t_{min}. When a node's task buffer is exhausted, Dtree transparently requests more work from the node's parent.

This approach results in reducing amounts of work being issued to requesting children, effectively balancing load.

Heterogeneity. A node may specify a multiplier, n_m, to its Dtree in order to indicate its performance relative to other nodes. The default multiplier is 1.0. As an example, if a cluster has two types of nodes, A and B, and a node of type B executes a task from the task pool in roughly twice the time a node of type A would take to execute that task, then nodes of type A would specify a multiplier of 2.0 while nodes of type B would specify a multiplier of 1.0.

Dtree parent nodes use node multipliers to compute distribution fractions for their children, effectively scaling task allocations. Load can be balanced effectively even if these estimates cannot be provided, however Dtree can be more efficient if they are available.

Mapping Task IDs. On receiving a task group to execute from Dtree, the application may schedule a task per thread, or use multiple (or all) threads to process the task. In either case, the task ID needs to be mapped to the input data for that task. The application must establish a swift means of translating the task IDs in the task group to the requisite task data. If this data is non-local, it may potentially be pre-fetched at this point, enabling a significant performance boost from computation and communication overlap. Global Arrays [4] may be

Table 1. Dtree parameters

F	Tree fan-out. Up to 1024, depending on network traffic
T	Total number of tasks
d_f	Size of the initial (static) distribution. 0.2 is a reasonable default
d_r	Task drain rate. 0.5 is a reasonable default
t_{min}	Minimum task allocation. Function of mean task length
n_m	Node multiplier. 1.0 is default

used effectively for data storage as task IDs (0 to $T-1$) can be used to index them.

3 Micro-benchmark

We describe the use of Dtree with full applications in the following two sections. However, we have additionally written a micro-benchmark in order to evaluate Dtree performance and efficiency under a range of conditions. The micro-benchmark is an MPI application that uses OpenMP® for intra-node parallelism. It does not perform any actual computation – a task is simply a timed delay.

As discussed in Sect. 2.1, the key considerations for scheduling the tasks in a pool are the mean and standard deviation of the task lengths. We instrumented the execution of the ParaBLe and GTFock applications on two real datasets each, and recorded the actual lengths of the tasks. From this data, we have computed the statistics shown in Table 2.

Table 2. Measured task pool statistics in seconds

Application	Dataset	Mean	Std. Dev.
ParaBLe	<leaf,development>	0.696	0.314
	<seed,development>	0.414	0.114
GTFock	15mer	0.733	0.197
	graphene.336	0.644	0.153

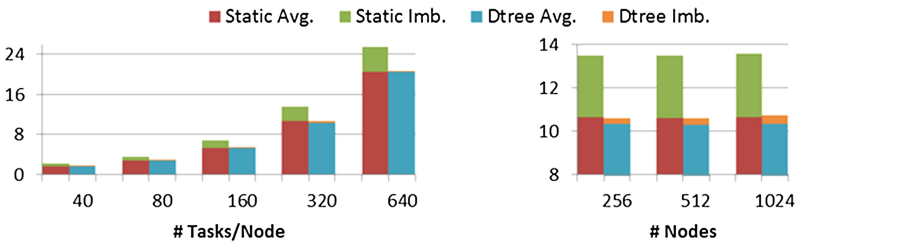
We observe from the histograms of the measured task lengths that they could roughly be approximated by a Gaussian probability distribution. We apply this observation to generate task lengths for the micro-benchmark as follows: given a mean and a standard deviation, we use the Intel Math Kernel Library [1] to generate pseudo-random numbers in a Gaussian distribution. Each node uses the same seed for the generator and generates R sets of n random numbers each, where R is the number of participating nodes, and $n \times R$ is the total number of tasks specified. For each of the R sets, we vary the mean for the n numbers randomly within one standard deviation. This approach produces an artificial task pool that approximates the characteristics seen in the real applications.

In each of the following experiments, we compare the runtime and the average load imbalance of a micro-benchmark run that dynamically schedules tasks using Dtree, against the runtime and average load imbalance of a run that is statically scheduled, i.e. an equal number of tasks are distributed to each node. We run one MPI rank per node, with each rank using all 16 threads available on the two Intel Xeon processors. For the experiments involving Intel Xeon Phi coprocessors, we run one MPI rank per coprocessor, using all 240 threads available.

3.1 Scaling Experiments

The total number of tasks to be distributed is significant in evaluating scheduling efficiency. Clearly, as this number reduces, it becomes harder to balance load – at 1 task per thread, it becomes impossible for any scheduler. Our first experiment evaluates Dtree's effectiveness in this regard. We ran these tests using Intel Xeon processors only on 256 nodes with a mean task length of 0.5 s and a standard deviation of 0.125 s. In Fig. 2a, we see that even with as few as 2.5 tasks per thread (40 tasks per node with each node running 16 threads) and a total runtime of 2 s, Dtree does better than a static schedule, although we do observe a load imbalance of 13 %. For 40 tasks per thread, the load imbalance is 1.6 %.

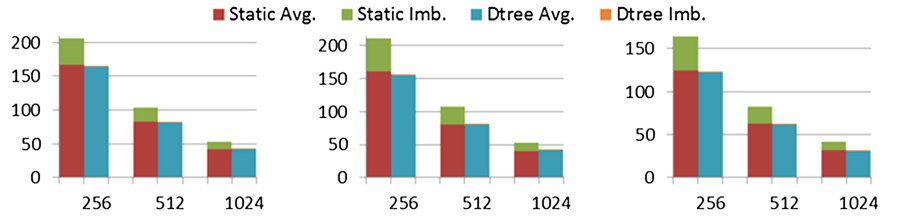
We then perform a weak scaling experiment using only Intel Xeon processors, the results of which are shown in Fig. 2b. We ran these tests using a mean task length of 0.5 s and a standard deviation of 0.125 s with 320 tasks per node. Dtree exhibits a load imbalance of no more than 5 %.



(a) CPU-only scaling w.r.t. # tasks/nodes at 256 nodes (b) Weak scaling CPU-only using 320 tasks/node

Fig. 2. Scaling number of tasks and weak scaling with the micro-benchmark

We then performed a strong scaling experiment using only Intel Xeon processors, running 1,310,720 tasks with a mean task length of 0.5 s and a standard deviation of 0.125 s. The results are shown in Fig. 3a. Given a larger number of tasks, Dtree reduces load imbalance to no more than 1.4 % of runtime, even at 1024 nodes.



(a) Strong scaling CPU-only (b) Strong scaling coprocessor-only (c) Strong scaling heterogeneous

Fig. 3. Strong scaling experiments with the micro-benchmark. The X-axis is # nodes and Y-axis is time-to-solution in seconds.

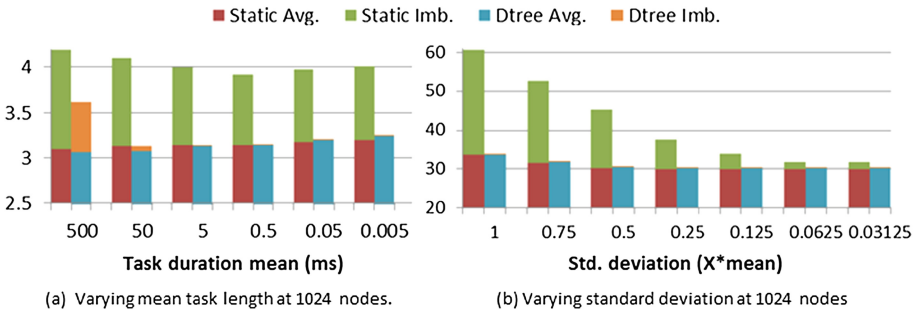


Fig. 4. Effect of task length and standard deviation. Y-axis is time-to-solution in seconds. Standard deviation for (a) is 0.125 s. Mean for (b) is 0.05 s.

We repeated the strong scaling experiment using only Intel Xeon Phi coprocessors in order to evaluate the performance of Dtree on a manycore cluster. We execute 19,660,800 tasks with a mean task length of 0.5 s and a standard deviation of 0.165 s. The results are shown in Fig. 3b. Again, we see that Dtree reduces load imbalance to no more than 1.9 % of runtime.

Our final scaling experiment with the micro-benchmark evaluates performance on a heterogeneous cluster of both Intel Xeon and Intel Xeon Phi processors. We execute 10,000,000 tasks with mean task lengths of 0.35 and 0.87 s, and standard deviations of 0.12 and 0.0.29 s for the host processor and coprocessor respectively. We see in Fig. 3c that despite the differing speeds and core counts of the processors, Dtree reduces load imbalance to no more than 3.7 % of runtime.

3.2 Task Length Experiments

Given the importance of task lengths in dynamic scheduling, we used the micro-benchmark to analyze Dtree performance for task pools with differing characteristics. In particular, we assessed the impact of changing the mean task length, and also of changing the standard deviation. We ran all these experiments on 1024 nodes, using only Intel Xeon processors.

We begin by experimenting with the mean task length. In order to maintain a roughly consistent runtime across experiments, we increase the number of tasks when we reduce the mean length. Thus, for a mean task length of 0.5 s, we run 6,400 tasks, whereas for a mean task length of 0.000005 s, we run 640,000,000 tasks. Note that as the task length reduces, thread scheduling overhead increases – this issue is unrelated to Dtree but can affect runtime considerably, to the point of obscuring the performance we are evaluating. For this reason, for this experiment alone, we use a single thread per MPI rank. Figure 4a shows the results of this experiment, clearly demonstrating that Dtree can effectively schedule tasks as short as 5 ms (13,500 processor cycles).

We then explore the effect of varying the standard deviation of the tasks in the task pool. We see in Fig. 4b that even with very small variance in task lengths, Dtree outperforms a static schedule.

4 ParaBLe

Bayesian network structure learning is an important machine learning problem. ParaBLe (Parallel Bayesian Learning) implements a parallel heuristic algorithm to solve this problem at scale [22,23]. We have modified this application to use Dtree. In this section, we provide a brief introduction to the algorithm, describe how we use Dtree, and evaluate the performance gains from doing so.

4.1 Bayesian Network Structure Learning Algorithm

A Bayesian network is represented as a directed acyclic graph in which nodes represent the set of variables of interest in the domain. Data regarding the observations of these variables are taken to estimate the joint probability distribution of the variables – a Bayesian network is the graphical representation of a factorization of the joint probability distribution. Automatically learning the structure of a Bayesian network from data is an NP-hard problem [10].

Let n denote the number of variables, X, Y, etc. denote individual variables, $\mathcal{X}$ denotes the set of all variables, and A, B, etc. denote subsets of $\mathcal{X}$. The structure learning problem can be defined as follows. Let the function $s(X, A)$ model the fitness of choosing elements of set A as the parents of X. Let $CP(X) \subseteq \mathcal{X}$ denote the set of candidate parents of a variable X. For each variable X, we need to identify $Pa(X) \subseteq CP(X)$ that maximizes $s(X, Pa(X))$. This requires computing $s(X, A)$ for all $A \subseteq CP(X)$. ParaBLe represents this computation using a hypercube of dimension $|CP(X)|$, in which each node represents a subset of $CP(X)$. Therefore, for each variable X, all the $2^{|CP(X)|}$ nodes of the corresponding hypercube have to be computed. Note that choosing $CP(X) = \mathcal{X} \setminus \{X\}$ results in exact structure learning, which has exponential complexity. ParaBLe heuristically determines much smaller $CP(X)$ sets to stem the computational complexity.

4.2 Work Decomposition

Consider the n candidate parents sets, exploring the subsets of which constitutes the total amount of work to be done. The CP sets vary significantly in size and the corresponding work varies exponentially in the size of the CP sets, therefore the required work for each variable is vastly different. The number of variables and the exponential variation in the corresponding workloads does not permit balancing the load to thousands of nodes. To more effectively distribute work, ParaBLe chooses an r-dimensional hypercube as the largest allowed unit of work, for a specific threshold r. For any variable X with $|CP(X)| > r$, the corresponding hypercube is divided into $2^{|CP(X)|-r}$ sub-hypercubes each creating a work item[2]. Any variable X with $CP(X) \leq r$ creates a work item with hypercube dimension $|CP(X)|$. This creates a sufficiently large list of work items.

[2] We refer to Dtree tasks as work items in this application.

4.3 Using Dtree

The work items are arranged in a global order so that using a small metadata and a bijective mapping, the ID of a work item can be mapped to the corresponding (variable, hypercube) pair. The data for learning the network is small enough to be replicated on every processor. Hence, no communication is needed to provide the data required for a work item. ParaBLe executes one work item with one thread. The results of the work items assigned to a node are accumulated on the node and reduced across all nodes on completion.

ParaBLe has optimized implementations for executing work items on both the Intel Xeon and Intel Xeon Phi processors. It uses the offload model [5] to achieve heterogeneity, running MPI only on the host processor and using a dynamic scheduling algorithm to balance load between the processors.

We modified the application to use the symmetric model, eliminating the complex offloading code and creating a simpler, truly heterogeneous version, running MPI on both the host processor and on the coprocessor, and using Dtree to balance load across all the processors and coprocessors in the cluster.

4.4 Datasets Used

ParaBLe was created to learn genome-scale networks from microarray data, a grand challenge in systems biology, and we test Dtree's performance for the same application. In this application, nodes in the Bayesian network represent genes of the underlying organism. The data comes from a large number of experiments, each of which measures quantitatively the expression level of each gene in the organism under different experimental conditions. We ran our experiments on two datasets for the plant *Arabidopsis thaliana* – one for the seedling tissue type and the other for the leaf tissue type, both classified under developmental conditions. More details on these datasets may be found in [22].

4.5 Experimental Results

We ran strong scaling experiments with ParaBLe on the two datasets over 256, 512, and 1024 heterogeneous nodes on Stampede. We measured the time-to-solution of the implementation using the offload model with static and dynamic task distribution across nodes using an earlier version of Dtree, against the implementation using the symmetric model with the improved Dtree. The graphs in Fig. 5 show the results of these experiments. We see that performance improves in all cases, up to 18 % with respect to dynamic on 1024 nodes, while load imbalance is almost completely eliminated.

5 GTFock

Fock matrix construction is an important algorithm in quantum chemistry. It is the most time consuming step in the widely used Hartree Fock (HF) algorithm

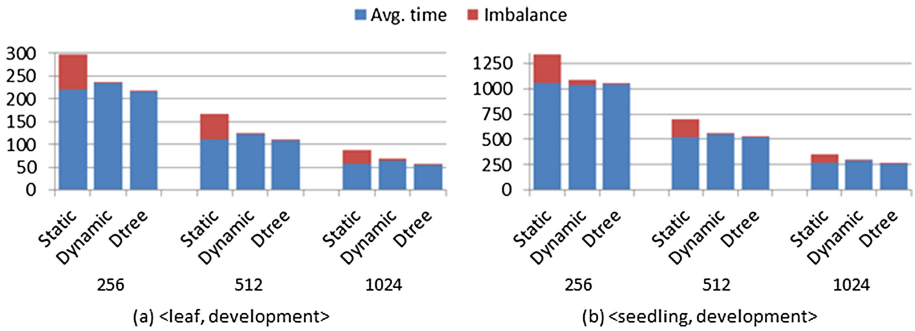


Fig. 5. ParaBLe performance (static and dynamic schedule) vs. using Dtree for 256, 512 and 1024 nodes. Y-axis is time-to-solution in seconds.

and optimizations of Fock matrix construction have been studied for decades. GTFock [17], NWChem [26], GAMESS [25], ACESIII [18] and MPQC [14] are some of the many existing computational chemistry packages that implement parallel Fock matrix construction and HF.

In this section, we provide a brief introduction to GTFock and to Fock matrix construction, describe how we use Dtree, and study the resulting performance.

5.1 Fock Matrix Construction

The Fock matrix is defined as

$$F_{ij} = H_{ij}^{core} + \sum_{kl} D_{kl}(2(kl|ij) - (ik|jl)) \tag{1}$$

where i, j, k and l are indices such that $0 \leq i, j, k, l < n_f$, where n_f is the number of basis functions. The two dimensional matrices H^{core} and D are fixed for this computation. Each $(ij|kl)$ is an element of a 4-dimensional array called an electron repulsion integral (ERI) tensor. Along each dimension, the indices are grouped into "shells" of 1 to 10 (or more) indices, which is necessary to facilitate efficient computation of ERIs in quantum chemistry programs.

GTFock defines a task[3] as computing the set of ERIs,

$$(M, : |N, :) = \{(ij|kl)\ s.t.\ i \in shell\ M,\ k \in shell\ N, 0 \leq j, l < n_f\} \tag{2}$$

and then computing contributions to the Fock matrix due to these ERIs. Here, M and N are shell indices. Some ERIs can be pre-determined to be very small, and their computation can be neglected. The ERI tensor also has 8-way symmetry, so only unique elements of $(ij|kl)$ need to be computed. Thus the number of ERIs in $(M, : |N, :)$ to be computed in each task can vary, making it hard to balance the load.

[3] We refer to Dtree tasks as work items, to prevent confusion with GTFock tasks.

All the tasks can be represented as a 2-dimensional task array indexed by M and N. The way the tasks are designed, there is a higher overlap between the corresponding blocks of D and F of batches $(M_1, : |N_1, :)$ and $(M_2, : |N_2, :)$ if $|M_2 - M_1|$ and $|N_2 - N_1|$ are small. Based on this, GTFock statically allocates subarrays of the task array across nodes. Therefore, each node can prefetch the required submatrices of D for all the tasks assigned to it. As a result of this partition, there is a significant overlap between the submatrices of D for tasks assigned to a node, thus greatly reducing the volume of data to be communicated. Similarly, submatrices of F can be accumulated locally first and the global copy can be updated once the entire parition is finished.

The blocks of tasks assigned to a node are divided into sub-blocks and computation is performed one sub-block at a time. When a node is done with its own quota, it uses work stealing to steal one of the sub-blocks from some other node that still has some left. It will also need to copy the corresponding blocks of D and update the corresponding blocks of F. Since this happens only towards the end, it results in only a small increase in communication while significantly balancing the load.

5.2 Using Dtree

For Dtree, the work items are defined as equal sized subarrays of the task array. When a node receives a work item, it uses the size of the blocks (S) and n_{shells}, to uniquely map the ID to the corresponding block of the task array. It then loads the corresponding blocks of D from the global copy, processes the work item, and updates the corresponding blocks of F in the global copy. Larger work items allow more overlap between the blocks of D and F within the work item and thereby reduce communication, but also make it harder to balance load.

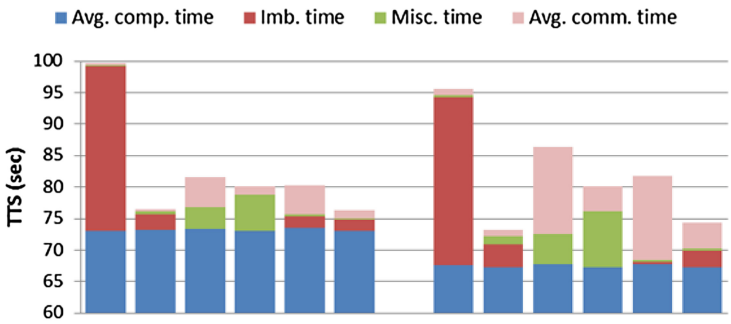
5.3 Datasets Used

We have used two datasets to study the performance of Dtree for Fock matrix construction. The system 15mer is a 15 nucleotide strand of DNA containing 981 atoms, 4826 shells and 10498 functions. The system 1hsg is a human immunodeficiency virus (HIV) II protease complexed with a 92-atom ligand which is an HIV inhibitor. We have modified the system to include only residues 8 Angstroms from any atom in the ligand, and call this system 1hsg_80. It has 1035 atoms, 4576 shells and 9584 functions.

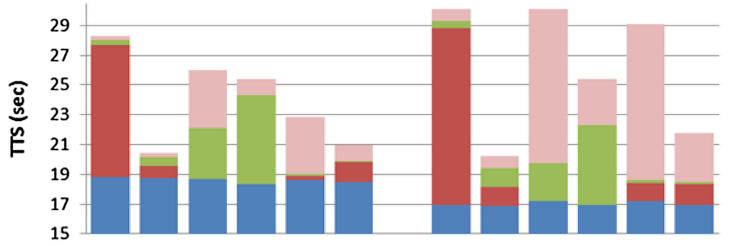
5.4 Experimental Results

Figure 6 shows the results of strong scaling experiments for Fock matrix construction by GTFock, using four different scheduling schemes – static, static with work stealing, using ADLB [19], and using Dtree – on the two datasets.

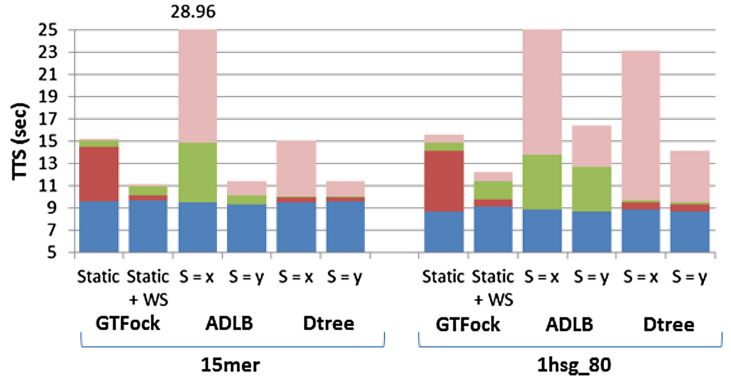
As expected, the static schedule has the highest amount of imbalance, but negligible communication. Adding work stealing significantly reduces the imbalance while increasing the communication only slightly. Dtree reduces imbalance



(a) 256 nodes. x= 30×30, y= 70×70.



(b) 1024 nodes. x= 15×15, y= 35×35.



(c) 2025 nodes. x= 10×10, y= 20×20.

Fig. 6. GTFock strong scaling experiment. Avg. comp. time and Avg. comm. time are averages of time spent by each node in computation and communication, respectively. Misc. time is obtained by subtracting the average computation and communication time from the average total time, and includes scheduling overhead. Imb. time is time-to-solution minus average total time and quantifies the load imbalance across nodes. The bars for ADLB do not have any imbalance since every node in effect waits for every other node before exiting the scheduler. So, for ADLB, the imbalance is included in the Misc. time. Parameters used: GTFock(# of sub-blocks = 5×5), ADLB(# servers = $\frac{1}{64}(\#nodes), task\_size = 1$), Dtree($F = 256$)

even further, but as it must fetch data for each task separately, the communication time is much higher.

We observe that as Dtree sends a range of tasks at a time, it is possible for a node to prefetch the data for most of these tasks, overlapping this communication with the computation for the first few tasks. This would significantly reduce the communication cost, and coupled with the lower imbalance exhibited with Dtree, would result in a smaller time-to-solution, even relative to work stealing.

The use of ADLB suffers the same communication overheads as the use of Dtree, but the time-to-solution with ADLB is much higher. There are three possible reasons for this. ADLB probably has higher load imbalance, as well as more overhead in task distribution. Moreover, ADLB uses dedicated servers for work distribution, as a result of which it loses some performance. We have experimentally determined that 1 of every 64 nodes needs to be a server. Thus, of 1024 nodes, 16 must be servers, resulting in a loss of $\frac{1}{64} = 1.6\,\%$ of the performance.

Effect of Work Item Size. We tested two work item sizes for both Dtree and ADLB. Our results confirm that smaller work items achieve better load balancing while requiring more communication. Therefore, if the communication and computation is overlapped, using smaller work items might be better.

6 Conclusion

We have presented Dtree, a dynamic task scheduler for applications with irregular computations. Dtree can effectively balance load for widely varying task profiles at petascale on heterogeneous supercomputers. We have demonstrated these capabilities with a micro-benchmark, and with two important HPC applications: ParaBLe, for Bayesian network structure learning, and GTFock, for Fock matrix construction. We have further shown near-perfect scaling up to 2K MPI ranks, and find no obstacle to scaling well beyond that. Dtree can easily be deployed in hybrid MPI+X applications, has negligible overhead, and enables balanced heterogeneous computation.[4]

Acknowledgements. The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. URL: http://www.tacc.utexas.edu.

[4] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance.

References

1. Intel® math kernel library MKL. http://software.intel.com/en-us/intel-mkl
2. Upc consortium. upc language specifications, v1.2. Technical report LBNL-59208, Lawrence Berkeley National Lab (2005)
3. TACC Stampede supercomputer (2014). http://top500.org/system/177931
4. Global arrays webpage (2015). http://hpc.pnl.gov/globalarrays/
5. Intel mpi on intel xeon phi coprocessor systems (2015). https://software.intel.com/en-us/articles/using-the-intel-mpi-library-on-intel-xeon-phi-coprocessor-systems
6. Mvapich: Performance (2015). http://mvapich.cse.ohio-state.edu/performance/pt_to_pt/
7. Bhatele, A., Kumar, S., Mei, C., Phillips, J., Zheng, G., Kale, L.: Overcoming scaling challenges in biomolecular simulations across multiple platforms. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, pp. 1–12, April 2008
8. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5), 720–748 (1999). http://doi.acm.org/10.1145/324133.324234
9. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, pp. 519–538. ACM, New York (2005). http://doi.acm.org/10.1145/1094811.1094852
10. Chickering, D.M., Heckerman, D., Geiger, D.: Learning Bayesian networks is NP-hard. Technical report MSR-TR-94-17, Microsoft Research (1994)
11. Devine, K.D., Boman, E.G., Heaphy, R.T., Hendrickson, B.A., Teresco, J.D., Faik, J., Flaherty, J.E., Gervasio, L.G.: New challenges in dynamic load balancing. Appl. Numer. Math. **52**(2–3), 133–152 (2005). http://dx.doi.org/10.1016/j.apnum.2004.08.028
12. Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable work stealing. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009, pp. 53:1–53:11. ACM, New York (2009). http://doi.acm.org/10.1145/1654059.1654113
13. Guo, Y., Zhao, J., Cave, V., Sarkar, V.: Slaw: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2010, pp. 341–342. ACM, New York (2010). http://doi.acm.org/10.1145/1693453.1693504
14. Janssen, C.L., Nielsen, I.M.: Parallel Computing in Quantum Chemistry. CRC Press, Boca Raton (2008)
15. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput. Surv. **31**(4), 406–471 (1999). http://doi.acm.org/10.1145/344588.344618
16. Lifflander, J., Krishnamoorthy, S., Kale, L.V.: Work stealing and persistence-based load balancers for iterative overdecomposed applications. In: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2012, pp. 137–148. ACM, New York (2012). http://doi.acm.org/10.1145/2287076.2287103

17. Liu, X., Patel, A., Chow, E.: A new scalable parallel algorithm for Fock matrix construction. In: 2014 IEEE International Parallel & Distributed Processing Symposium (IPDPS), Phoenix, AZ (2014)
18. Lotrich, V., Flocke, N., Ponton, M., Yau, A., Perera, A., Deumens, E., Bartlett, R.: Parallel implementation of electronic structure energy, gradient, and hessian calculations. J. Chem. Phys. **128**, 194104 (2008)
19. Lusk, E.L., Pieper, S.C., Butler, R.M., et al.: More scalability, less pain: a simple programming model and its implementation for extreme computing. SciDAC Rev. **17**(1), 30–37 (2010)
20. Menon, H., Kalé, L.: A distributed dynamic load balancer for iterative applications. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2013, pp. 15:1–15:11. ACM, New York (2013). http://doi.acm.org/10.1145/2503210.2503284
21. Min, S.J., Iancu, C., Yelick, K.: Hierarchical work stealing on manycore clusters. In: 5th Conference on Partitioned Global Address Space Programming Models (2011)
22. Misra, S., Vasimuddin, M., Pamnany, K., Chockalingam, S., Dong, Y., Xie, M., Aluru, M., Aluru, S.: Parallel Bayesian network structure learning for genome-scale gene networks. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC14, pp. 461–472, November 2014
23. Nikolova, O., Aluru, S.: Parallel Bayesian network structure learning with application to gene networks. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 63:1–63:9 (2012)
24. Saraswat, V.A., Kambadur, P., Kodali, S., Grove, D., Krishnamoorthy, S.: Lifeline-based global load balancing. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP 2011, pp. 201–212. ACM, New York (2011). http://doi.acm.org/10.1145/1941553.1941582
25. Schmidt, M.W., Baldridge, K.K., Boatz, J.A., Elbert, S.T., Gordon, M.S., Jensen, J.H., Koseki, S., Matsunaga, N., Nguyen, K.A., Su, S., et al.: General atomic and molecular electronic structure system. J. Comput. Chem. **14**(11), 1347–1363 (1993)
26. Valiev, M., Bylaska, E.J., Govind, N., Kowalski, K., Straatsma, T.P., Van Dam, H.J., Wang, D., Nieplocha, J., Apra, E., Windus, T.L., et al.: NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations. Comput. Phys. Commun. **181**(9), 1477–1489 (2010)
27. Zheng, G., Bhatelé, A., Meneses, E., Kalé, L.V.: Periodic hierarchical load balancing for large supercomputers. Int. J. High Perform. Comput. Appl. **25**(4), 371–385 (2011). http://dx.doi.org/10.1177/1094342010394383