Task Parallel Implementation of Belief Propagation in Factor Graphs

Nam Ma
Computer Science Department
University of Southern California
Los Angeles, CA 90089
Email: namma@usc.edu

Yinglong Xia
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
Email: yxia@us.ibm.com

Viktor K. Prasanna
Ming Hsieh Department of
Electrical Engineering
University of Southern California
Los Angeles, CA 90089
Email: prasanna@usc.edu

***Abstract*—Factor graphs have been increasingly used as probabilistic graphical models. Belief propagation is a prominent algorithm for inference in factor graphs. Due to the high complexity of inference, parallel techniques for belief propagation are needed. In this paper, we explore task parallelism for belief propagation in an acyclic factor graph. Our approach consists of building a task dependency graph based on the input factor graph and then using a dynamic task scheduler to exploit task parallelism. We conducted experiments on a state-of-the-art multi-core system using a variety of acyclic factor graphs. The experimental results show the efficiency and scalability of the proposed approach with a 37× speedup on a 40-core system.**

***Index Terms*—task parallelism; task scheduling; belief propagation; factor graph; multi-core;**

I. Introduction

Graphical models have been essential tools for probabilistic reasoning. *Factor graph*s [1] have emerged as a unified model of directed graphs (e.g. Bayesian networks) and undirected graphs (e.g. Markov networks). A factor graph naturally represents a joint probability distribution that is written as a product of factors, each involving a subset of random variables. Factor graphs have found applications in a variety of domains such as image processing, bioinformatics, and especially error-control decoding used in digital communications [2], [3], [4], [5], [6].

Inference is the problem of computing posterior probability distribution of certain variables given some value-observed variables as evidence. In factor graphs, inference proceeds with the well-known *belief propagation* algorithm [1]. Belief propagation is a process of passing messages along the edges of a graph. Processing each message requires a set of operations with respect to the probability distribution of the random variables in a graph. Such distribution is represented by *potential tables*. The complexity of belief propagation increases dramatically as the number of states of variables and node degrees of a graph increase. In many applications, such as digital communications, belief propagation must be performed in real time. Therefore, parallel techniques are needed to accelerate the inference.

Many parallel techniques have been proposed for belief propagation in factor graphs. However, most of these techniques are developed for loopy belief propagation in cyclic factor graphs with the employment of embarrassingly parallel algorithms [7], [8] or with the need of graph partitioning [9]. Parallelizing belief propagation in *acyclic* factor graphs still remains a challenging problem due to the precedence constraints among the nodes in the graphs. In the meanwhile, task scheduling has been extensively studied and used in parallel computing [10], [11], [12], [13]. In [14], task scheduling is shown to be an efficient tool for a class of linear algebra problems, known as regular applications, on general-purpose multi-core processors. Since belief propagation in acyclic factor graphs is an irregular application, task scheduling is even a more suitable tool for parallelizing it. Thus, we approach this problem by defining a task dependency graph for belief propagation and then using a dynamic task scheduler [15] to exploit task parallelism available in the task dependency graph.

Our contributions in this paper include:

- We define task dependency graphs for belief propagation in an acyclic factor graph based on the topology of the factor graph.
- We use a dynamic task scheduler to allocate tasks to cores. By using the dynamic task scheduling method with work-sharing approach, we maximize task parallelism and optimize load balancing across the cores.
- We implement our techniques on a state-of-the-art multi-core system consisting of 40 cores. The experimental results show the efficiency of our techniques in accelerating belief propagation.

The rest of the paper is organized as follows. In Section II, we review belief propagation in factor graphs and task scheduling problem. Section III discusses related work. Section IV presents our construction of a task dependency graph for belief propagation based on the factor graph topology. The use of task scheduling is discussed in Section V. Experimental setup and results are given in Section VI and VII, respectively. Section VIII concludes the paper.

II. Background

A. Factor Graphs

Given a set of random variables $X = \{x_1, x_2, \ldots, x_n\}$, a joint probability distribution of X can be written as a factorized function [16]:

This research was partially supported by the U.S. National Science Foundation under grant number CNS-1018801.

$$P(X) \propto \prod_{j=1}^{m} f_j(X_j) \tag{1}$$

where $\propto$ denotes *proportional to*; X_j is a subset of $\{x_1, \ldots, x_n\}$; factor $f_j(X_j)$ is called a local function of X_j, and m is the number of factors. A *local function* defines an unnormalized probability distribution of a subset of variables from X.

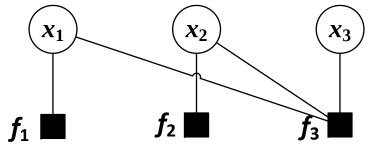


Fig. 1. A factor graph for the factorization $f_1(x_1)f_2(x_2)f_3(x_1, x_2, x_3)$.

A factor graph is a type of probabilistic graphical model that naturally represents such factorized distributions [1]. A factor graph is defined as $\mathbb{F} = (\mathbb{G}, \mathbb{P})$, where $\mathbb{G}$ is the graph structure and $\mathbb{P}$ is the parameter of the factor graph. $\mathbb{G}$ is a *bipartite graph* $\mathbb{G} = (\{X, F\}, E)$, where X and F are nodes representing *variables* and *factors*, respectively. E is a set of edges, each connecting a factor f_j and a variable $x \in X_j$ of f_j. Figure 1 shows a factor graph corresponding to the factorization $g(x_1, x_2, x_3) = f_1(x_1)f_2(x_2)f_3(x_1, x_2, x_3)$. Parameter $\mathbb{P}$ is a set of factor *potentials*, given by the definitions of local functions $f_i(X_j)$. For discrete random variables, a factor potential is represented by a table in which each entry corresponds to a state of the set of variables of the factor. We focus on discrete random variables in this paper.

B. Belief Propagation

Evidence in factor graphs is a set of variables with observed values, for example, $E = \{x_{e_1} = a_{e_1}, \ldots, x_{e_k} = a_{e_k}\}$, $e_k \in \{1, 2, \ldots, n\}$. Given evidence, an updated marginal distribution of any other variable can be inquired. *Inference* is the process of computing the posterior marginals of variables, given evidence. *Belief propagation* is a well-known inference algorithm introduced in [17], [18] and later formalized for factor graphs in [1]. After evidence E is *absorbed* at the observed variables, belief propagation is performed. Belief propagation is based on message passing processes where messages are computed locally and sent from a node to its neighbors. Two types of messages are given in Eqs. (2) and (3), one sent from variable node x to factor node f, and the other sent from factor node f to variable node x ([1]):

$$\mu_{x \to f}(x) \propto \prod_{h \in \mathcal{N}_x \setminus \{f\}} \mu_{h \to x}(x) \tag{2}$$

$$\mu_{f \to x}(x) \propto \sum_{\mathcal{N}_f \setminus \{x\}} \left(f(X_f) \prod_{y \in \mathcal{N}_f \setminus \{x\}} \mu_{y \to f}(y) \right) \tag{3}$$

where x and y are variables; f and h are factors; $\mathcal{N}_f$ and $\mathcal{N}_x$ are the sets of neighbors of f and x respectively; $\sum$ denotes marginalization over a potential table; $\prod$ denotes products of the potential tables. Note that a message is a distribution of a random variable. Both message computations require the products of the incoming messages from all the neighbors excluding the one where the message will be sent. Computing a message from f to x requires a marginalization for x.

We assume that each variable has at most r states and each node has at most d neighbors. Thus, the size of the potential table of a factor f is at most r^d, and the size of a message for a variable x is r. The serial complexity of computing $\mu_{x \to f}(x)$ is $O(d \cdot r)$, and that of computing $\mu_{f \to x}(x)$ is $O(d \cdot r^d)$. It can be seen that the complexity of computing $\mu_{f \to x}(x)$ is dominant the complexity of computing $\mu_{x \to f}(x)$.

In *acyclic* factor graphs, message passing is initiated at the leaves. Each node computes and sends a message to one of its neighbors *after* receiving messages from *all* the other neighbors. The process terminates when two messages have been passed on every edge, one in each direction. A common approach for belief propagation is building a rooted tree with a node chosen as the root, then propagating messages from the leaves to the root and from the root to the leaves sequentially. There are $|E|$ messages in Eq. (2) and $|E|$ messages in Eq. (3) computed during the execution of belief propagation in cycle-free factor graphs. Thus, the overall serial complexity of belief propagation is $O(|E| \cdot d \cdot r + |E| \cdot d \cdot r^d) = O(|E| \cdot d \cdot r^d) = O(m \cdot d^2 \cdot r^d)$, where m is the number of non-leaf factor nodes and $|E| = m \cdot d$. Belief propagation in acyclic factor graphs leads to *exact inference*, since all the final results are guaranteed to be exact [1].

In *cyclic* factor graphs, belief propagation must be performed in an iterative message passing process that is called *loopy belief propagation*. In one iteration, all messages are computed simultaneously using the messages from the previous iteration [1]. The process is terminated when the changes of messages between two consecutive iterations are less than a threshold. The final approximate results imply *approximate inference* for cyclic factor graphs. The parallel version of belief propagation in cyclic factor graphs is known as an embarrassingly parallel algorithm.

Finally, once the belief propagation completes, the posterior marginals of variables are computed by Eqs. (4) and (5). The exact $P(x)$ and $P(X_f)$ is obtained by normalizing the right sides of the equations.

$$P(x) \propto \prod_{f \in \mathcal{N}_x} \mu_{f \to x}(x) \tag{4}$$

$$P(X_f) \propto f(X_f) \prod_{x \in \mathcal{N}_f} \mu_{x \to f}(x) \tag{5}$$

C. Task Scheduling

In our context, the input to task scheduling is a task dependency graph (a.k.a. directed acyclic graph - DAG), where each node represents a task and each edge corresponds to the precedence constraint between the tasks. Each task in the DAG is associated with a *weight*, which is the estimated execution time of the task. A task can start execution when *all* of its predecessors have been completed [19], [10]. The task scheduling problem is to map the tasks to the processing units (e.g. threads, cores, processors) in order to minimize the overall execution time on parallel computing systems. Task scheduling is in general an *NP-complete* problem [20], [21]. We consider scheduling an arbitrary DAG with given task weights and perform the mapping and scheduling of tasks on-the-fly. The goal of such dynamic scheduling includes not only the minimization of the overall execution time, but also the minimization of the scheduling overhead [11].

III. Related Work

Many parallel techniques for inference in general graphical models have been proposed. In [22], parallelism for inference is explored in Bayesian network. In [23], [24], map-reduce is used for parallelizing inference in Conditional Random Fields and in general undirected graphical models. In [25], the authors investigate data parallelism for exact inference with respect to Bayesian networks, where a Bayesian network must be first converted into a junction tree. Node level primitives are proposed for evidence propagation in junction trees. The primitives including table marginalization, table extension, and table multiplication/division are optimized for distributed memory platforms. In [26], data parallelism in exact inference in junction trees is explored on Cell BE processors. Although all synergistic processing elements (SPEs) in a Cell BE processor access a shared memory, data must be explicitly moved to the local store of a SPE in order to be processed.

There are a few earlier papers studying parallelism for belief propagation in factor graphs [7], [9]. However, while these techniques are proposed for general factor graphs which lead to approximate inference, we focus on acyclic factor graphs for exact inference. In digital communications, parallel belief propagation is implemented on FPGAs, GPUs, and multi-core processors [27], [8]. Similar to [7], these techniques apply to cyclic factor graphs with the employment of embarrassingly parallel algorithms. In [28], data parallelism is explored for belief propagation at node level computation. This approach is suitable only if the node level computation offers sufficient parallelism. Our approach explores structural parallelism offered by acyclic factor graphs based on the computational independencies among nodes that do not have any precedence relationship.

The scheduling problem has been extensively studied on parallel systems for several decades [13], [11], [12], [14]. Early algorithms optimized scheduling with respect to the specific structure of task dependency graphs [29], such as a tree or a fork-join graph. In general, however, programs come in a variety of structures [11]. Scheduling techniques have been proposed by several emerging programming systems such as Cilk [30], Intel Threading Building Blocks (TBB) [31], OpenMP [32], Charm++ [33] and MPI micro-tasking [34], etc. All these systems rely on a set of extensions to common imperative programming languages, and involve a compilation stage and runtime libraries. These systems are not optimized specifically for scheduling DAGs on manycore processors. For example, the authors in [35] show that Cilk is not efficient for scheduling workloads in dense linear algebra problems on multicore platforms. In [14], a dynamic task schedulers is proposed as the optimized method to exploit parallelism for a class of linear algebra problems on general-purpose multi-core processors. Because belief propagation in acyclic factor graph is an irregular application that has a DAG-structured computation, using a task scheduler is a suitable method for parallel execution. Thus, in this paper, we use a task scheduler to exploit task parallelism in the task dependency graph constructed for belief propagation in an acyclic factor graph.

IV. Task Dependency Graph for Belief Propagation

Belief propagation is a process of passing messages along the edges. As given in Section II, in an acyclic factor graph, a node computes and sends a message to its neighbor after receiving messages from all the other neighbors. The process proceeds with first converting the factor graph into a rooted tree with a node chosen to be the root. Then, in this rooted tree, messages are propagated from the leaves to the root and from the root to the leaves, sequentially. Note that a leaf node in factor graph only needs to send its message to its single neighbor without any computation. Thus, only non-leaf nodes in the factor graph need to compute messages. Figure 2(a) shows a factor graph where non-leaf nodes form the *skeleton* of the factor graph.

It is natural to construct a task dependency graph for belief propagation in which the computation at each node in the skeleton becomes a task. We denote this as the First method. Tasks are defined as follows. On the way up from the leaves to the root of the skeleton, a task at each node is the computation of a message sent to the parent. The computation of a message at a variable node and at a factor node are based on Eq. 2 and 3, respectively. On the way down from the root to the leaves of the skeleton, a task at each node is the computation of all messages sent to the children. These tasks have different task sizes depending on the complexity of the computations. Let d denote the node degree and let r denote the number of states of each variable. On the way up, the task complexity is $w_{uf} = d \cdot r^d$ at a factor node, and is $w_{uv} = d \cdot r \cdot d$ and at a variable node. On the way down, the task complexity at a factor node is $w_{df} = d^2 \cdot r^d$, and at a variable node it is $w_{dv} = d^2 \cdot r \cdot d$. Detail implementation of message computation can be found in [28] with the assumption that each message is computed by only one thread.

The task dependency graph is constructed from the skeleton of a factor graph as illustrated in Figure 2. The top half of the task dependency graph corresponds to the process of propagating messages from the leaves to the root, the bottom half of it corresponds to the process of propagating messages from the root to the leaves of the skeleton. It can be seen that task parallelism is affected by the number of children of each node in the skeleton and by the total number of nodes.

In the task dependency graph given by Figure 2(b), each task in the bottom half is the computation of all the messages

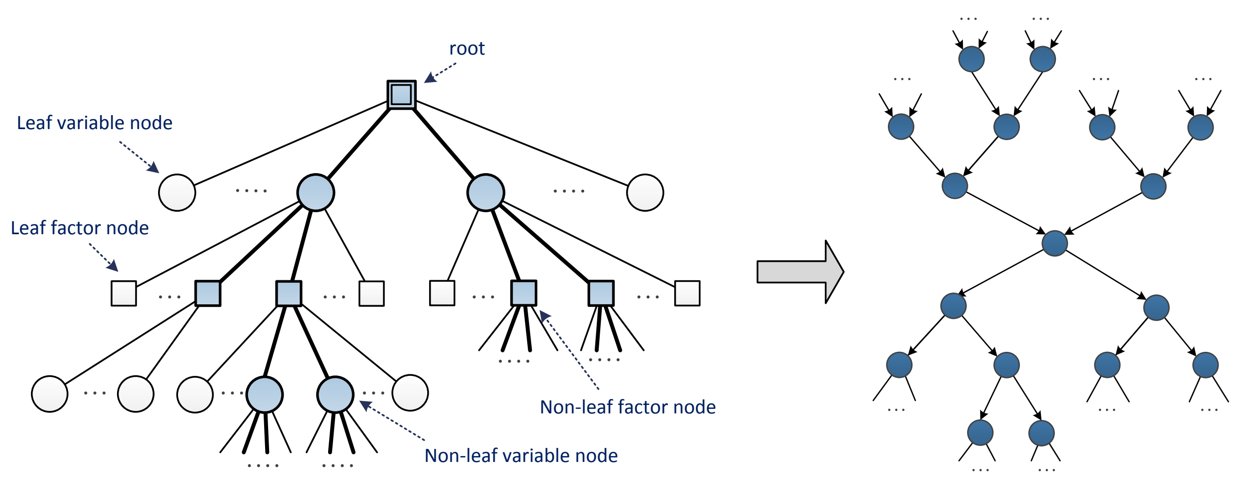


Fig. 2. (a) A factor graph with its skeleton formed by the non-leaf nodes, and (b) the corresponding task dependency graph.

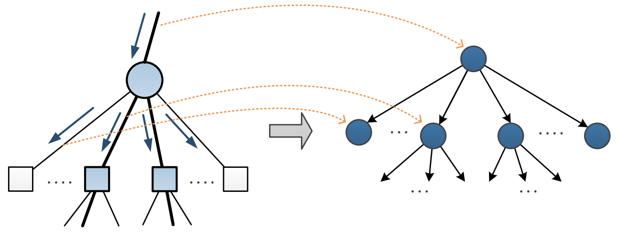


Fig. 3. An alternative way to construct a task dependency graph from a factor graph.

sent to the children of the corresponding node in the factor graph. Alternatively, we can define a task as the computation of only one message. Thus, the new tasks correspond to the edges in the factor graph. A task in the bottom half of this task dependency graph (Figure 2(b)) becomes smaller independent tasks in the resulting task dependency graph. Hence, this task dependency graph has more parallelism and has smaller tasks. This alternative construction of task graph is illustrated in Figure 3. In this paper, we use the First method to construct the task dependency graph, as given in Figure 2.

V. Task Scheduling for Belief Propagation in Factor Graphs

The task dependency graph constructed above ensures the precedence order of propagating messages in the input factor graph. It also exhibits the task parallelism available in belief propagation: ready tasks that have no precedence constraints can be executed in parallel. So as to exploit the task parallelism, we use a task scheduling method that dynamically distributes ready tasks to threads based on the current workload of the threads [15]. The scheduling method is briefly described as follows.

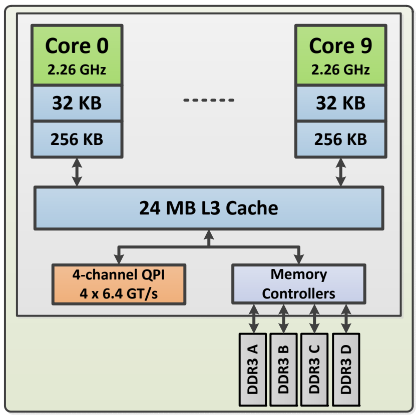
The input task dependency graph is represented by a list called *Global task list* (GL), which all the threads have access to. Each element of the GL stores a task and the related information such as the dependency degree, task weight, the successors, and the task meta data (e.g. application specific parameters). The *dependency degree* of a task is initially set as the number of incoming edges of the task. We keep the *successors* that are the pointers to the elements of succeeding tasks along with each task. Each element of the GL also stores *task meta data*, such as the task type and pointers to the data buffer of the task. A shared *Completed task buffer* is used to store the IDs of tasks that have completed execution.

The *Allocate module* in each thread allocates ready tasks from the GL to the threads for execution. A task in the GL is ready when its dependency degree becomes zero. The Allocate module pops out each task from the Completed task buffer and decrements the dependency degree of each successor of the task. If the dependency degree of a successor reaches zero, it is allocated by the Allocate module to the thread that has the smallest workload.

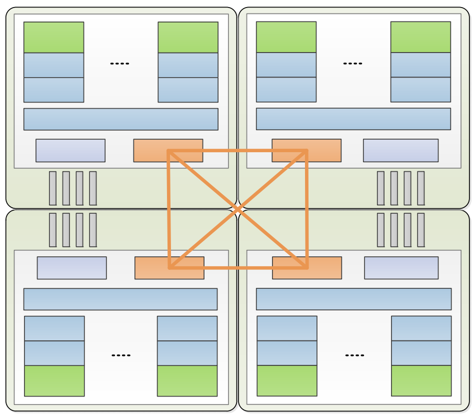
Each thread has a *Local task list* (LL) to store the ready tasks that will be executed in this thread. Any thread can add tasks to an LL, but only the owner thread of that LL can remove its tasks. This organization reduces the access conflicts among the threads, hence reducing the overhead of scheduling. Each LL has a *weight counter* to keep track of the current workload of the thread. When a task is added to an LL, the LL's weight counter is updated by the task weight.

Each thread has an *Execute module* that picks the ready tasks in its own LL for execution. This module performs the task based on the meta data associated with the task. When completing the execution of a task, this module stores the ID of the task to the Completed task buffer and send a notice the Allocate module in its thread. The weight counter of its LL is also updated.

Hence, the task scheduler helps maximize the task parallelism of the task graph since all ready tasks will be allocated to a thread for parallel execution. Because the scheduler allocates tasks to the thread that has the smallest workload,



(a)



(b)

Fig. 4. Architectures of (a) the 10-core Intel Westmere-EX processor and (b) the 4-socket Westmere-EX system with 4 processors fully connected via QPI links.

load balancing is supported during execution.

VI. Experimental Setup

A. Platforms and Implementation Setup

We conducted experiments on a 40-core Intel Westmere-EX system as a representative state-of-the-art multi-core system. The system consists of four Xeon E7-4860 processors fully connected through 6.4 GT/s QPI links. Each processor has 10 cores running at 2.26 GHz, 24 MB L3 cache, and 16 GB DDR3 memory. Thus, the system has total 64 GB shared memory. Figure 4, adapted from [36], shows the detailed architecture of Xeon E7-4860 and the whole system.

We used Pthreads for the task scheduler. Hyperthreading was disabled. In our initial experiments, we noted that running multiple threads on each core showed no improvement. Thus, we initiated as many threads as the number of hardware cores, and bound each thread to a separate core. These threads persisted over the entire program execution and communicated with each other using the shared memory. We evaluated the scalability of our proposed method using various numbers of threads up to 40.

B. Datasets

So as to examine the impact of task size and amount of task parallelism on the execution time, we generated synthetic factor graphs with various skeleton structures. A skeleton was generated as a rooted tree in which the root is a factor node. Three types of skeletons used in our experiments are described below.

- A *Balanced-tree* skeleton was generated as a tree in which every node, except the leaf nodes, had an identical number of children c. A balanced-tree skeleton offers sufficient parallelism to achieve high speedup.
- A *Slim-tree* skeleton, as shown in Figure 6, was formed by first creating a balanced binary tree with b, $b > 1$ leaf nodes. Then each of the leaf nodes was connected to a chain consisting of $\lceil \frac{n-(2b-1)}{b} \rceil$ nodes. Increasing b increases the amount of structural parallelism.
- A *Random-tree* skeleton was generated based on a random graph generator described in [12]. Parameters used to specify a random-tree include: (1) number of nodes n, (2) tree height h, (3) maximum number of children of each node c_m. In the experiments, we kept $n = 2000$ and varied h and c_m to get various shapes of tree. For each pair of h and c_m, we generated 20 random trees from which we computed the average m, average c, and measured the average execution time.

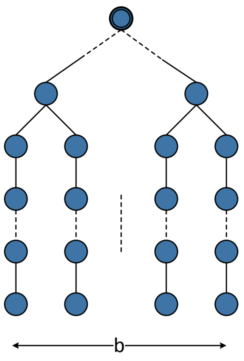
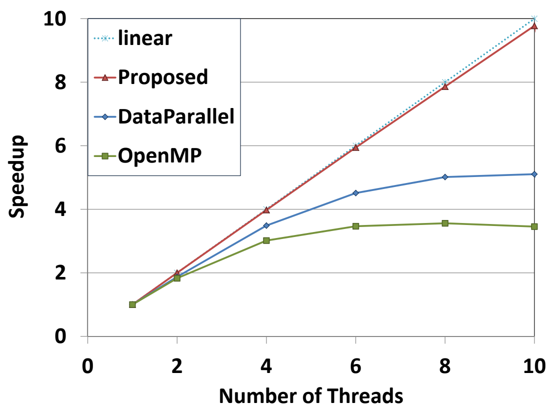


Fig. 6. The slim-tree skeleton of factor graphs.

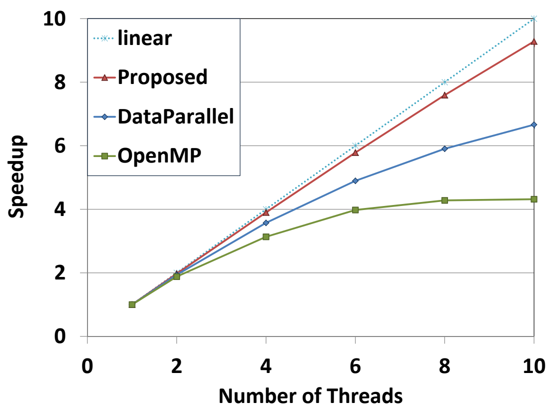
C. Performance Metrics and Baselines

We used execution time and speedup to to evaluate the performance of our method. The speedup on P cores is calculated as the ratio of the execution time on a single core to the execution time on P cores.

We compared our proposed method with two parallel baselines. The first baseline used OpenMP directives to parallelize the message computation at node level. The second baseline used data parallel technique for the message computation [28]. The second baseline has shown very good results when po-



(a)



(b)

Fig. 5. Speedups of belief propagation using various methods on a 10-core Westmere-EX processor for (a) the balanced-tree factor graph and (b) the random-tree factor graph.

tential tables are large. The comparison of performance of our method with these two baselines is given in the next section.

VII. Experimental Results

A. Notation Summary

We summarize the notations used in our experimental results as follows:

- n: number of nodes in the skeleton of a factor graph (i.e. number of non-leaf nodes in the factor graph).
- m: number of factor nodes in the skeleton.
- c: number of children of each node in the skeleton.
- d: degree of a node (i.e. number of neighbors of a node) in the factor graph.
- r: number of states of a variable.
- b: number of chains of the slim-tree skeleton.
- h: height of skeleton.
- P: number of threads (cores).

B. Comparison with Baselines

We compare our method with the two baselines using a balanced-tree dataset and a random-tree dataset. The balanced-tree dataset has $r = 2, d = 12, n = 2000, c = 4$, leading to $m = 908$. The random-tree dataset has $r = 2, d = 14, n = 1024, h = 15$; the generated factor graph has average number of children $c = 4.4$ and $m = 723$. Figure 5 shows the experimental results for the three methods using these two datasets on a 10-core Intel Westmere-EX processor. In both datasets, the proposed method scales much better than the two baselines. Especially for the balanced-tree dataset, the proposed method achieves almost linear speedup with the number of threads, while the two baselines have very limited speedup. From the two datasets, it can be seen that the balanced-tree dataset has smaller task size yet more task parallelism compared to the random-tree dataset.

TABLE I
Overall complexity and data size for slim-tree factor graphs with $n = 2047$, $r = 2$, $d = 14$, and various values of b.

b	m	overall complexity ($m \cdot d^2 \cdot r^d$)	data size (MB) ($8 \cdot m \cdot r^d$)
16	1029	3.3E+09	128
32	1013	3.3E+09	126
64	1045	3.4E+09	130
128	981	3.2E+09	122

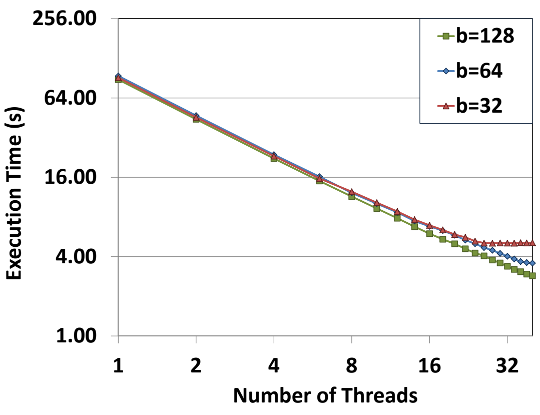
TABLE II
Overall complexity and data size for balanced-tree factor graphs with $n = 2000$, $r = 2$, $d = 14$, and various values of c.

c	m	overall complexity ($m \cdot d^2 \cdot r^d$)	data size (MB) ($8 \cdot m \cdot r^d$)
2	1318	4.2E+09	165
4	908	2.9E+09	114
6	1334	4.3E+09	167

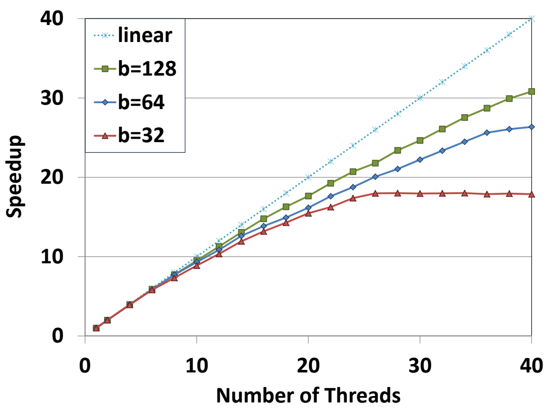
C. Impact of Task Parallelism

In factor graphs with slim-tree skeleton, the amount of task parallelism varies as b. At any time, at most b tasks can be executed in parallel. In this experiment, slim-tree factor graphs with $n = 2047, r = 2, d = 14$, and $b = 32, 64, 128$ were used. Task size is fixed by r and d. Size of the potential table at each factor node is $r^d = 2^{14}$. The number m of factor nodes in the skeleton is varied by various values of b. The overall serial complexity and the data size are given in Table I. The table shows that the datasets have similar overall complexities. The total size of the data is the size of all the potential tables multiplied by 8 (the size of a double-precision data for each potential value). The data size is larger than the total cache size to guarantee that there was no cache advantage in the experiments.

Figure 7 shows the execution times and speedups of the proposed method for the slim-tree factor graphs on the 40-core

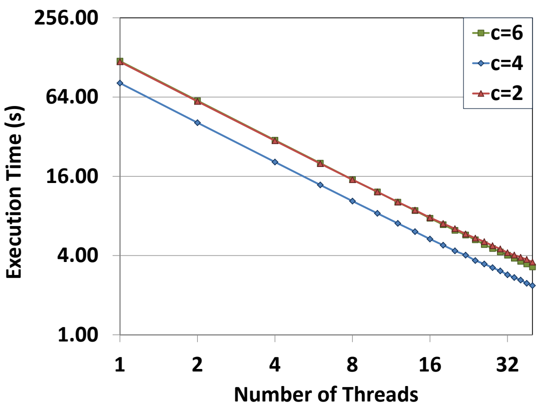


(a) Execution Time

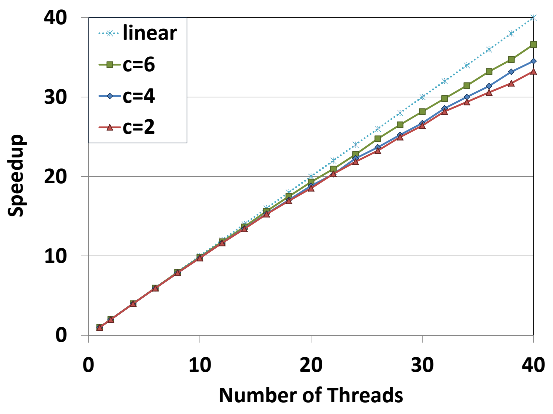


(b) Speedup

Fig. 7. Performance of belief propagation in slim-tree factor grahps, $n = 2047$, $r = 2$, $d = 14$, and various values of b on the 40-core Intel Westmere-EX system.



(a) Execution Time



(b) Speedup

Fig. 8. Performance of belief propagation in balanced-tree factor graphs with $n = 2000$, $r = 2$, $d = 14$, and various values of c on the 40-core Intel Westmere-EX system.

Westmere-EX system. The execution times are approximately identical when using 1 thread. When the number of threads increases, it can be seen that belief propagation in factor graphs with larger b achieves higher speedup. When b is smaller than P, the speedup is limited by b no matter how many threads are used. The achieved speedup is actually smaller than b due to the overhead of the scheduler and limited task parallelism at the balanced-tree part of the graph. For $b = 128$, the proposed method achieves a speedup of 31 using 40 threads.

For balanced-tree datasets, the greater the number c of children of each node, the larger the amount of task parallelism is. The datasets have $n = 2000$, $r = 2$, $d = 14$, and various values of c which derive various values of m. Task sizes and potential table sizes were also fix by r and d. The overall complexity and data size are given in Table II. With $c = 4$, the generated factor graph has higher number m of non-leaf factors than that with $c = 4$ or $c = 6$ does. Thus, as can be in Figure 8, the dataset with $c = 4$ has smaller execution time than the datasets with $c = 2$ and $c = 6$. Regarding scalability, the greater the value of c, the higher speedup is achieved. For $c = 6$, the proposed method scales almost linearly with the number of threads, achieving a speedup of 37 using 40 threads. The speedups for $c = 2$ and $c = 4$ are also very good, much higher than the speedups achieved for slim-tree factor graphs. The reason is that the scheduler kept all cores busy because of the sufficient parallel activities provided by the balanced-tree factor graphs.

D. Impact of Task Size

In this experiment, we used balanced-tree datasets with a fixed amount of task parallelism. Various values of r and d were used to examine the impact of task sizes on the performance of our method. The datasets have $n = 2000$, $c = 4$, resulting $m = 908$. The four task types, given in Section IV, have different task sizes (i.e. task weights). Table III provides the task sizes, the overall complexity, and total size for the datasets with various values of r and d. As can be seen in this table, tasks at factor nodes are far larger

TABLE III
TASK SIZE, OVERALL COMPLEXITY, AND DATA SIZE FOR BALANCED-TREE FACTOR GRAPHS WITH $n = 2000$, $c = 4$, AND VARIOUS VALUES OF r AND d.

r	d	m	w_{uf} $(d \cdot r^d)$	w_{df} $(d^2 \cdot r^d)$	w_{uv} $(d \cdot r \cdot d)$	w_{dv} $(d^2 \cdot r \cdot d)$	overall complexity $(f \cdot d^2 \cdot r^d)$	data size (MB) $(8 \cdot f \cdot r^d)$
2	16	908	1.0E+06	1.7E+07	5.1E+02	8.2E+03	1.5E+10	454
2	14	908	2.3E+05	3.2E+06	3.9E+02	5.5E+03	2.9E+09	113
2	12	908	4.9E+04	5.9E+05	2.9E+02	3.5E+03	5.4E+08	28
2	10	908	1.0E+04	1.0E+05	2.0E+02	2.0E+03	9.3E+07	7
12	5	908	1.2E+06	6.2E+06	3.0E+02	1.5E+03	5.6E+09	1723
8	5	908	1.6E+05	8.2E+05	2.0E+02	1.0E+03	7.4E+08	227
4	5	908	5.1E+03	2.6E+04	1.0E+02	5.0E+02	2.3E+07	7

than tasks at variable nodes ($d \cdot r^d$ vs. $d \cdot r \cdot d$). Also tasks performed during the downward propagation process are larger than tasks performed during the upward propagation process by a factor of d. When fixing d and varying r, the task sizes and the potential table sizes are highly various.

Impact of task size on the performance of the proposed method is illustrated in Figures 9 and 10. The results show that the larger the tasks, the better the speedup is achieved. For the datasets with $r = 2$ and various values of d, the achieved speedups vary slightly. Using 40 threads, the proposed method achieves a speedup of 33 for $d = 10$ and a speedup of 35 for $d = 16$. For the datasets with $d = 5$ and various values of r, the achieved speedups vary more greatly. Using 40 threads, the proposed method achieves a speedup of 28 for $r = 8$ and a speedup of 35 for $r = 12$. The results are expected, since larger tasks help reduce the the ratio of overhead of the scheduler to the overall execution time. The results also show that our method can achieve good scalability even for the factor graphs with a small amount of computation at node level.

E. Results on Random Trees

In the random-tree datasets, the number c of children of each node in the skeleton was generated randomly from a uniform distribution over $[0, c_m]$. We fixed the number n of nodes and varied tree height h and c_m to get different shapes for the skeleton. In particular, the datasets have $n = 2000$ and h chosen from $5, 15, 25$. Value of c_m was set to be 20 for $h = 5$, 15 for $h = 15$, and 10 for $h = 25$. For each value of h, we generated a set of 20 random trees. The smaller the tree height h, the more "bushy" the tree is, and hence more task parallelism is available. In this experiment, we kept $r = 2$ and $d = 14$. Table IV shows average values of m and c, the overall complexity and data size for each value of h. As expected, the average value of c decreases as h increases. The average value of m also slightly decreases as h increases.

Figure 11 shows the performance of our method on the generated random-tree factor graphs. We measured the average execution time for each set of factor graphs corresponding to h. The execution times reflect the complexity and scalability of the proposed method for each set of factor graphs. Using a single thread, when h increases, the execution time slightly decreases corresponding to the average value of m. As expected, the method shows best scalability for factor graphs with $h = 5$, achieving a speedup of 36.6 using 40 threads. The larger the tree height, the better the speedup is. The speedups for random-tree factor graphs are almost as good as the speedups for balanced-tree factor graphs.

TABLE IV
OVERALL COMPLEXITY AND DATA SIZE FOR RANDOM-TREE FACTOR GRAPHS WITH $n = 2000$, $r = 2$, $d = 14$, AND VARIOUS VALUES OF h.

h	average m	average c	overall complexity	data size (MB)
5	1701	11.4	5.46E+09	213
15	1634	8.7	5.25E+09	204
25	1505	6.3	4.83e+09	188

VIII. CONCLUSION

In this paper, we explored task parallelism for belief propagation in acyclic factor graphs. Our approach consists of constructing a task dependency graph for the input factor graph and then using a task scheduler to allocate tasks to the cores for parallel execution. Our experimental results show that it is an efficient method for parallelizing belief propagation in factor graphs. For factor graphs with sufficient task parallelism, the proposed method achieved 37× speedup on a 40-core system. As part of our future work, we plan to integrate data parallel technique to this method so that a large task can also be parallelized by the scheduler. In addition, the scheduler can group small tasks to reduce communication overhead and increase data locality.

IX. ACKNOWLEDGEMENTS

Thanks to the management, staff, and facilities of the Intel Manycore Testing Lab ([37], [38]).

REFERENCES

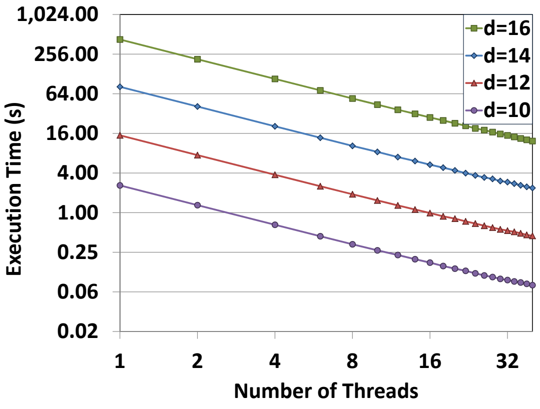
[1] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 498–519, 2001.

[2] E. Sudderth and W. T. Freeman, "Signal and Image Processing with Belief Propagation," *IEEE Signal Processing Magazine*, vol. 25, Mar. 2008.

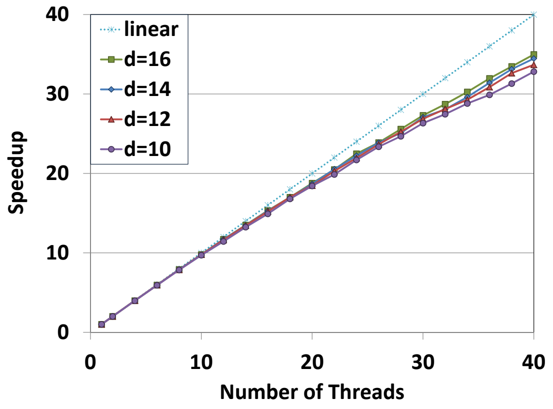
[3] A. Mitrofanova, V. Pavlovic, and B. Mishra, "Integrative protein function transfer using factor graphs and heterogeneous data sources," in *Proceedings of the 2008 IEEE International Conference on Bioinformatics and Biomedicine*. IEEE Computer Society, 2008, pp. 314–318.

[4] T. Richardson and R. Urbanke, "The Capacity of Low-Density Parity Check Codes under Message-Passing Decoding," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 599–618, 2001.

[5] C. Schlegel and L. Perez, *Trellis and Turbo Coding*. John Wiley & Sons, 2003.

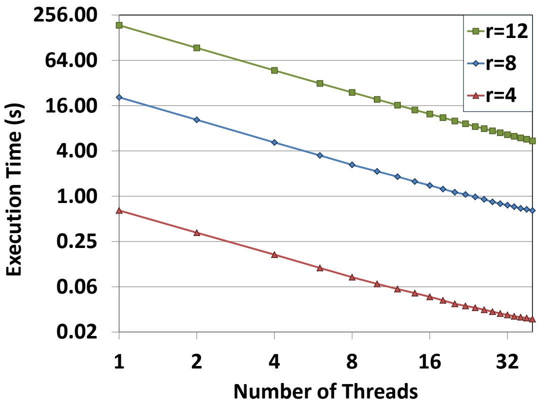


(a) Execution Time

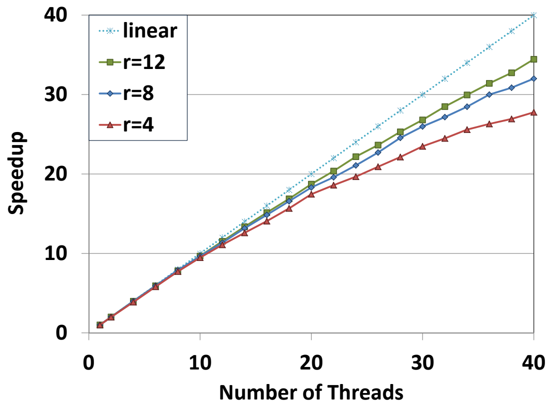


(b) Speedup

Fig. 9. Performance of belief propagation in balanced-tree factor graphs with $n = 2000$, $c = 4$, $r = 2$, and various values of d on the 40-core Intel Westmere-EX system.

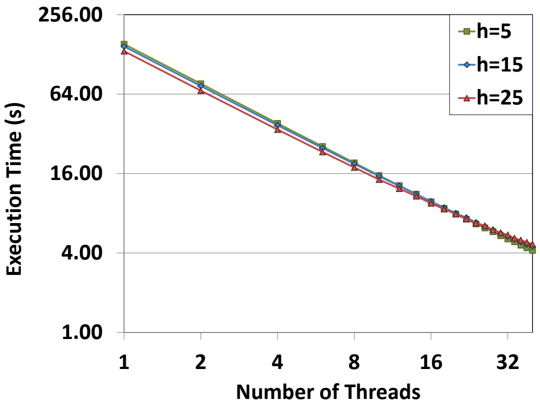


(a) Execution Time

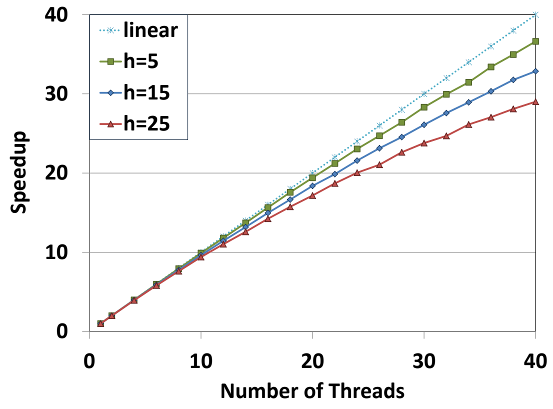


(b) Speedup

Fig. 10. Performance of belief propagation in balanced-tree factor graphs with $n = 2000$, $c = 4$, $d = 5$, and various values of r on the 40-core Intel Westmere-EX system.



(a) Execution Time



(b) Speedup

Fig. 11. Performance of belief propagation in random-tree factor graphs with $n = 2000$, $r = 2$, $d = 14$ and various values of h on the 40-core Intel Westmere-EX system.

[6] H. Loeliger, "An Introduction to factor graphs," *IEEE Signal Processing Magazine*, vol. 21, no. 1, pp. 28–41, 2004.

[7] A. Mendiburu, R. Santana, J. A. Lozano, and E. Bengoetxea, "A parallel framework for loopy belief propagation," in *GECCO (Companion)*, 2007, pp. 2843–2850.

[8] G. Falcão, L. Sousa, and V. Silva, "Massively LDPC decoding on Multicore Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 2, pp. 309–322, February 2011.

[9] J. Gonzalez, Y. Low, C. Guestrin, and D. O'Hallaron, "Distributed parallel inference on large factor graphs," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, Montreal, Canada, July 2009.

[10] O. Sinnen, *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2007.

[11] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406–471, 1999.

[12] H. Topcuouglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, pp. 260–274, March 2002.

[13] I. Ahmad, S. Ranka, and S. Khan, "Using game theory for scheduling tasks on multi-core processors for simultaneous optimization of performance and energy," in *Intl. Sym. on Parallel Dist. Proc.*, 2008, pp. 1–6.

[14] F. Song, A. YarKhan, and J. Dongarra, "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems," in *International Conference for Hight Performance Computing, Networking Storage and Analysis*, 2009.

[15] Y. Xia and V. K. Prasanna, "Collaborative scheduling of dag structured computations on multicore processors," in *Conf. Computing Frontiers*, 2010, pp. 63–72.

[16] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009.

[17] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988.

[18] S. L. Lauritzen and D. J. Spiegelhalter, "Local computation with probabilities and graphical structures and their application to expert systems," *Royal Statistical Society B*, vol. 50, pp. 157–224, 1988.

[19] I. Ahmad, Y.-K. Kwok, and M.-Y. Wu, "Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors," in *Proceedings of the 1996 International Symposium on Parallel Architectures, Algorithms and Networks*, 1996, pp. 207–213.

[20] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[21] C. Papadimitriou and M. Yannakakis, "Towards an architecture-independent analysis of parallel algorithms," in *Proceedings of the twentieth annual ACM symposium on Theory of computing*, 1988, pp. 510–513.

[22] A. V. Kozlov and J. P. Singh, "A parallel Lauritzen-Spiegelhalter algorithm for probabilistic inference," in *Proceedings of Supercomputing*, 1994, pp. 320–329.

[23] N. Piatkowski and K. Morik, "Parallel inference on structured data with CRFs on GPUs," in *International Workshop at ECML PKDD on Collective Learning and Inference on Structured Data (COLISD2011)*, 2011.

[24] S. Singh, A. Subramanya, F. Pereira, and A. McCallum, "Distributed map inference for undirected graphical models," in *Neural Information Processing Systems (NIPS), Workshop on Learning on Cores, Clusters and Clouds*, 2010.

[25] Y. Xia and V. K. Prasanna, "Scalable Node Level Computation Kernels for Parallel Exact Inference," *IEEE Transactions on Computers*, vol. 59, no. 1, pp. 103–115, 2009.

[26] ——, "Parallel Evidence Propagation on Multicore Processors," *The Journal of Supercomputing*, 2010.

[27] M. Karkooti and J. Cavallaro, "Semi-parallel reconfigurable architectures for real-time LDPC decoding," in *Proceedings of International Conference on Information Technology: Coding and Computing*, 2004.

[28] N. Ma, Y. Xia, and V. K. Prasanna, "Data parallelism for belief propagation in factor graphs," in *Proceedings of the 23th International Symposium on Computer Architecture and High Performance Computing*, October 2011, pp. 56–63.

[29] E. G. Coffman, *Computer and Job-Shop Scheduling Theory*. New York, NY: John Wiley and Sons, 1976.

[30] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," Cambridge, Tech. Rep., 1996.

[31] Intel Threading Building Blocks, "http://www.threadingbuldingblocks.org/."

[32] OpenMP Application Programming Interface, "http://www.openmp.org/."

[33] Charm++ programming system, "http://charm.cs.uiuc.edu/research/charm/."

[34] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani, "Mpi microtask for programming the cell broadband enginetm processor," *IBM Systems Journal*, vol. 45, no. 1, pp. 85–102, 2006.

[35] J. Kurzak and J. Dongarra, "Fully dynamic scheduler for numerical computing on multicore processors," Tech. Rep., 2009.

[36] D. P. V. Agarwal, F. Petrini and D. Bader, "Scalable graph exploration on multicore processors," in *Proceedings of the 22nd IEEE and ACM Supercomputing Conference (SC10)*, 2010.

[37] Intel® Manycore Testing Lab. [Online]. Available: http://www.intel.com/software/manycoretestinglab

[38] Intel® Software Network. [Online]. Available: http://www.intel.com/software