Analytical Modeling and Optimization for Affinity Based Thread Scheduling on Multicore Systems

Fengguang Song [#1], Shirley Moore [#2], Jack Dongarra [#*3]

[#] *EECS Department, University of Tennessee*
Knoxville, TN, USA
{[1]song, [2]shirley, [3]*dongarra*}@eecs.utk.edu

[*] *Oak Ridge National Laboratory*
Oak Ridge, TN, USA

***Abstract*—This paper proposes an analytical model to estimate the cost of running an affinity-based thread schedule on multicore systems. The model consists of three submodels to evaluate the cost of executing a thread schedule: an affinity-graph submodel, a memory hierarchy submodel, and a cost submodel that characterize programs, machines, and costs respectively. We applied the analytical model to both synthetic and real-world applications. The estimated cost accurately predicts which schedule will provide better performance. Due to the NP-hardness of the scheduling problem, we designed an approximation algorithm to compute near-optimal solutions. We have extended the algorithm to support threads with data dependences. We conducted experiments with a computational fluid dynamics (CFD) kernel and Cholesky factorization on both UMA SMP and NUMA DSM machines. The results show that using the optimized thread schedule can improve the program performance by 25% to 400%, demonstrating that our method for determining an optimized thread schedule for multicore systems is efficient and practical.**

I. Introduction

With the emergence of chip multi-processors (CMP) [1], [2], [3], future DSM systems will have less powerful processor cores but will have tens of thousands of cores. Performance asymmetry in multicore platforms is another trend due to budget issues such as power consumption and area limitation as well as various degrees of parallelism in applications [4], [5], [6]. We call such a system "heterogeneous manycore DSM system" (see Fig. 1). Processor cores belonging to the same level (e.g., same chip or board) frequently share memory resources. For instance, cores on the same chip may share an L2 or L3 cache.

It is critical to improve user programs' memory access efficiency to speed up program performance. We run SGI's performance monitoring tool `pmshub` on the SGI Altix 3700 BX2 machine from NCSA. Figure 2 shows that a number of user programs are experiencing a large amount of remote memory accesses on the DSM machine.

Our goal is to search for an optimal thread schedule to improve the memory effectiveness on all levels in the multi-level memory hierarchy. Threads in our context refer to fine-grained user level threads that can be as small as a block of instructions for which a user program can create hundreds of thousands of such threads.

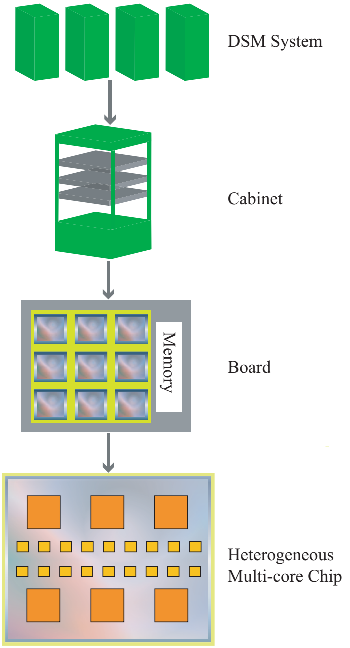

Fig. 1. Heterogeneous manycore DSM system under study.

To investigate the affinity-based thread scheduling problem, we first propose an analytical model to evaluate the cost of a thread schedule and then tackle it as an optimization problem. In particular, the analytical model consists of three submodels:

- *affinity graph submodel* for describing the affinity relationship between threads in a user program,
- *memory hierarchy submodel* for abstracting the memory hierarchy of a multicore system, and
- *cost submodel* for predicting the cost of a schedule to run the threads on the multicore system.

Our strategy is to let the affinity graph submodel characterize the user program and the memory hierarchy submodel characterize the system architecture. Then, in combination with the third cost submodel, we are able to answer the question "Given a multi-threaded program T and a machine M, what is the cost to use a thread schedule A to execute

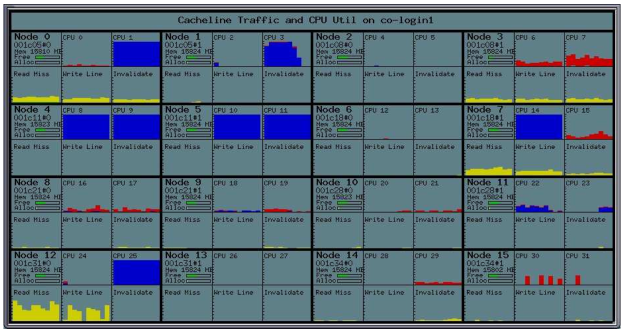

Fig. 2. A screen shot of the performance monitoring tool `pmshub` on SGI Altix. The light yellow area reflects how many remote memory accesses have occurred.

program T on machine M?" Since finding an optimal thread schedule is NP-hard, we propose a hierarchical graph partitioning algorithm to compute a near-optimal solution.

Our analytical model is supported experimentally. We have applied the model to two synthetic applications and a real application and showed that it can accurately measure the quality of a thread schedule. We have also extended our previous tool [7] to support DAG scheduling and deployed it on two real-world applications: a computational fluid dynamics (CFD) kernel and Cholesky factorization. The performance results on an Intel Quadcore Clovertown machine and a SGI Altix machine show that our new thread schedules are able to improve program performance greatly (by 25% to 42% for the CFD kernel and 30% to 400% for Cholesky factorization).

To save space, we sketch some of the proofs and provide the complete proofs in [8]. The paper is organized as follows. Section II introduces our previous work of a feedback directed performance tool to determine an optimized thread schedule as well as other related work. Section III describes the analytical model and its three submodels. Section IV describes the hierarchical graph partitioning algorithm for finding a near-optimal solution to the optimization problem. The extension to support threads with data dependences is described briefly in Section V. Section VI presents our experimental results. Finally, Section VII gives the conclusion.

II. Previous and Related Work

A. Feedback-Directed Optimization Tool

Our previous work developed a feedback-directed trace-based optimization tool to determine optimized thread schedules for multi-threaded programs [7]. The framework (see Fig. 3) relies upon a binary instrumentation tool to (i) obtain and analyze the memory trace of each thread and represent the nature of memory sharing between threads by an affinity graph. Next, (ii) we partition the affinity graph into a number of subgraphs. Based on the partition (one subgraph per processor), (iii) we use a breadth-first traversal method to compute an optimized schedule for each processor. Finally, (iv) users run the program again taking as input the feedback file.

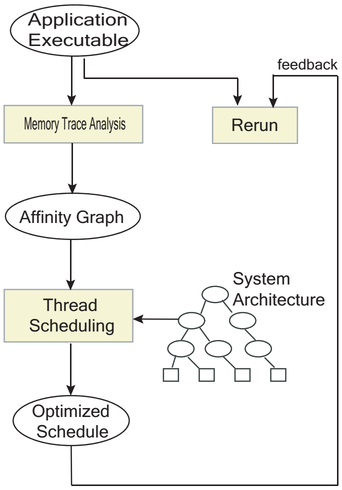

Fig. 3. Structure of the feedback-directed thread scheduling tool [7].

Although the schedules generated by our previous work were able to effectively speed up various applications, there were certain inputs for which the program performance was nearly unchanged. The analytical model presented in this paper is intended to provide insight into why some thread schedules improve performance and others do not, enabling us to explain the seemingly ad hoc speedups. This paper provides a theoretical foundation for the affinity-based thread scheduling problem and proves that the hierarchical partitioning algorithm is an approximation algorithm. Furthermore, we have extended our previous feedback-directed optimization method to support threads with data dependences. The new extension allows us to determine good schedules for DAG scheduling problems to optimize the memory access efficiency.

B. Other Related Work

The *augmented task graph* approach considers both computation and communication among tasks for distributed-memory systems [9]. We extend this approach to shared-memory systems by introducing the "affinity graph" to describe the data-reuse relationship between different threads (or tasks). We define the metric of data reuse as the number of words accessed in common by two threads (or tasks). Our memory hierarchy submodel is similar to the well-known hierarchical storage architecture but uses a simple abstract tree to combine both processors cores and various levels of memories.

Philbin et al. propose a user-level thread library to improve cache locality using fine-grained threads [10]. Pingali et al. create locality groups to restructure computations for a variety of applications but require hand-coded optimizations [11]. We address the problem of affinity thread scheduling based on a more generic and abstract shared-memory model and propose a hierarchical partitioning algorithm to solve the optimization problem.

Traff applies a hierarchical partitioning technique similar to ours to solve the MPI process mapping problem [12]. He implemented a framework to compute an optimal MPI process placement to minimize the message passing cost. Pichel et al. formulate sparse matrix-vector product as a graph problem where each row of the sparse matrix represents a vertex [13], [14]. Their method works effectively on both SMP and ccNUMA DSM systems, but is limited to the SpMV application.

III. The Analytical Model

Given a set of single-application user-level threads $\{t_1, \ldots, t_m\}$ without data dependences, and a number of heterogeneous processors $p_1, \ldots, p_n$ located in a shared-memory hierarchy, find a good schedule A to achieve:

(a) maximal data reuse within a processor,
(b) minimal remote memory accesses, and
(c) load balancing.

Note that here we use threads to refer to user-level threads that can be as small as a block of instructions (e.g., task) or as large as a kernel-level thread (e.g., pthread). In our experiments, we create a thread for each fine-grained task so that there are a large number of threads to schedule. Our previous work [7] (e.g., section 2.2) introduced several techniques to process large-size graphs efficiently.

Let schedule A be an onto function:

$$A : \{1, \ldots, m\} \longrightarrow \{1, \ldots, n\}, m \geq n.$$

$A(i) = j$ means put thread t_i on processor p_j. $A^{-1}(j)$ denotes the subset of threads running on processor p_j. We allow threads to have different workloads and processors to be heterogeneous with varying computational capabilities.

A. Affinity Graph Submodel

We use the concept of "affinity" to quantify how many data items are accessed in common by a pair of threads. Affinity graph builds on this concept and represents the affinity relationship among a set of threads. Affinity graph was first introduced in [7]. For completeness, we list it here briefly.

Definition 1 (Affinity Graph)**.** *The affinity graph is an undirected weighted graph $G = \langle T, E, w_t, w_e \rangle$, where*

- *$T = \{t_i$ is a user-level thread $\mid t_i$ is data independent of $t_j, \forall i \neq j\}$,*
- *E=$\{(t_i, t_j) \mid \exists$ datum x such that both t_i and t_j access $x\}$,*
- *$w_t : T \longrightarrow Z^+$ denotes the amount of computation of each thread,*
- *$w_e : E \longrightarrow Z^+$ denotes the affinity strength between two threads. If $(t_i, t_j) \notin E$, we define $w_e(t_i, t_j) = 0$.*

B. Memory-Hierarchy Submodel

We assume a shared memory system has a hierarchical memory architecture. For instance, a number of processor cores may share an L2 or L3 cache. We define such a hierarchical shared memory system as follows:

Definition 2 (Memory Hierarchy Submodel)**.** *A shared-memory system R is a tree of the form*

$$R = (r, T),$$

where r is a memory node, T is the children of r and

$$T = \{(r_i, T_{r_i}) \mid T_{r_i} \text{ is the children of } r_i\}.$$

We assume all the leaves are of the same height h (i.e., on the hth level), and all the edges on the same level l have identical weight w^l. Specifically, the leaf tree nodes $(r_i, \emptyset)$ denote processor cores and the interior tree nodes at levels $0 \ldots h-1$ denote memories. We also assume each memory contains a copy of the data in its children.

Figure 4 shows an example of a DSM system. For convenience, we define the ancestor memories of a node n by $ancestor(n) = \{m$: memory m is a node residing on the path from root to n$\}$.

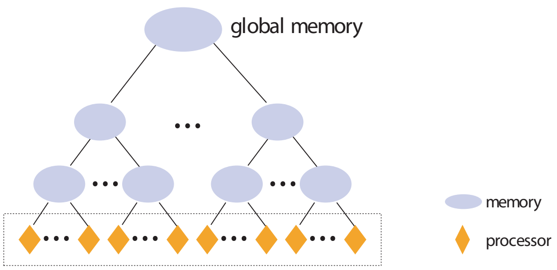

Fig. 4. A 3-level memory hierarchy on a multicore DSM system.

Definition 3 (Memory Latency)**.** *If processor p accesses datum x that is stored in memory m and its ancestor memories, we define memory latency $lat(p, x) = w(p, m)$, where*

$$w(p, m) = \sum_{\text{edge } e \in \text{ path } \sigma \text{ from p to m}} w^{level(e)}.$$

Lemma 1. *Let datum x reside in memory m together with m's ancestor memories, if processor p is a descendant of m but processor p' is not, then $lat(p, x) < lat(p', x)$.*

Proof: Let the lowest common ancestor of p and p' be $m2 = lca(p, p')$. Since p' is not under the subtree of m, $level(m2) < level(m)$. By Definition 3,

$$lat(p, x) = w(p, m) = w^{h-1} + w^{h-2} + \ldots + w^{level(m)},$$

$$lat(p', x) = w(p, m2) = w^{h-1} + w^{h-2} + \ldots + w^{level(m2)}.$$

Since $level(m2) < level(m)$, we get $lat(p', x) > lat(p, x)$. ■

Corollary 1. *If two threads t_i and t_j access the same datum x stored in memory m, placing them on two processors located in the subtree of m minimizes $lat(t_i, x) + lat(t_j, x)$.*

Proof: By Lemma 1, placing thread t_i on a descendant processor p_i of m will minimize $lat(t_i, x)$. Similarly, $lat(p_j, x) = \min_{\forall p} lat(p, x)$ if p_j is another descendant processor of m. Therefore, placing the two threads on on two processors in the subtree of m minimizes $lat(t_i, x) + lat(t_j, x)$ since both latencies are minimal. ■

Corollary 1 implies that the thread placement may affect the program performance if two threads have an affinity relationship.

C. Cost Submodel

After knowing the affinity relationship between threads and the characteristics of the underlying architecture, we are now ready to estimate the cost of running a thread schedule.

Definition 4 (Cost Submodel)**.** *Given an affinity graph G, a shared-memory system M, and a thread schedule A, we define the cost to execute schedule A on system M as*

$$cost(G, M, A) = \sum_{\forall p_i, p_j} cost(A^{-1}(p_i), A^{-1}(p_j), M, G),$$

where

$$cost(T_i, T_j, M, G) = \sum_{t_i \in T_i, t_j \in T_j} w_e(t_i, t_j) lat(p_i, m_c),$$

m_c is the lowest common ancestor of processors p_i and p_j.

Lemma 2. *Assume a shared-memory system M has a set P of processors, and each memory m has k children such that m has k subsets $D(m)_i$ of processors (each child has a subtree and leads to a subset of processors), $i = 1 \dots k$. Suppose*

$$pair(m) = \{(p_x, p_y) \mid p_x \in D(m)_i, p_y \in D(m)_j, i, j \in [1, k]\},$$

then $\{pair(m) \mid m \in M\}$ is a partition for set

$$\mathcal{P} = P \times P \setminus \{(p_i, p_i) \mid p_i \in P\}.$$

Proof Sketch: It is easy to show that (1) $\bigcup_{m \in M} pair(m) \subseteq \mathcal{P}$, (2) $\mathcal{P} \subseteq \bigcup_{m \in M} pair(m)$, and (3) $pair(m_i) \cap pair(m_j)) = \emptyset$. ■

Theorem 1. *Suppose a schedule A runs a set of threads in affinity graph G on a system M. Let $time_l$ denote $lat(p, p\text{'s ancestor memory at level } l)$ and M_l denote the set of memories at level l, then*

$cost(G, M, A)$ can also be expressed as:

$$\sum_{l=0}^{h-1} \sum_{m \in M_l} \sum_{\substack{(p_i, p_j) \\ \in pair(m)}} \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y) \times time_l.$$

In other words,

$$cost(G, M, A) = \sum_{l=0}^{h-1} time_l \times SharingOnLevel_l,$$

where $SharingOnLevel_l$ denotes the amount of affinity between threads that access those memories located on level l. That is,

$$SharingOnLevel_l =$$

$$\sum_{m \in M_l} \sum_{\substack{(p_i, p_j) \\ \in pair(m)}} \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y).$$

Proof Sketch: Suppose processors p_i and p_j have the lowest common ancestor memory $lca(p_i, p_j)$. By definition,

$$cost(G, M, A) =$$

$$\sum_{p_i \neq p_j} \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y) lat(p_i, lca(p_i, p_j))$$

$$= \sum_{(p_i, p_j) \in \mathcal{P}} \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y) time_{level(lca(p_i, p_j))}.$$

By Lemma 2,

$$cost(G, M, A) =$$

$$= \Big(\sum_{m \in M} \sum_{(p_i, p_j) \in pair(m)}\Big) \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y) time_{level(m)}$$

Since every memory $m \in M_l$ for a certain l,

$$cost(G, M, A) =$$

$$\Big(\Big(\sum_{l=0}^{h-1} \sum_{m \in M_l}\Big) \sum_{\substack{(p_i, p_j) \\ \in pair(m)}}\Big) \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y) time_{level(m)}.$$

By $time_{level(m)} \in \{time_0, \dots, time_{h-1}\}$, and all memories $\in M_l$ have the same $time_l$,

$$cost(G, M, A) =$$

$$time_0 \sum_{m \in M_0} \sum_{(p_i, p_j) \in pair(m)} \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y)$$

$$+ time_1 \sum_{m \in M_1} \sum_{(p_i, p_j) \in pair(m)} \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y)$$

$$\dots$$

$$+ time_{h-1} \sum_{m \in M_{h-1}} \sum_{(p_i, p_j) \in pair(m)} \sum_{\substack{t_x \in A^{-1}(p_i) \\ t_y \in A^{-1}(p_j)}} w_e(t_x, t_y).$$

That is,

$$cost(G, M, A) = \sum_{i=0}^{h-1} time_i \times SharingOnLevel_i.$$

■

IV. Solving the Optimization Problem

Given an affinity graph $G = \langle T, E, w_t, w_e \rangle$ and a shared-memory system M, the problem of finding an optimal schedule A^* such that $cost(G, M, A^*) = \min_{\forall A} cost(G, M, A)$ can be considered as an integer linear programming problem.

A. An Integer Linear Programming Problem

Suppose $time_i > time_{i+1} > 0$ and M is of height h. Let

$$x_i = SharingOnLevel_i$$

denote the sum of affinity strength on level i ($0 \leq i \leq h-1$), and $x_h = \sum_{p_i} \sum_{t_x,t_y \in A^{-1}(p_i)} w_e(t_x, t_y)$ denote the sum of affinity strength within each processor. The ILP problem is formulated as follows:

1) Minimize $\sum_{i=0}^{h-1} x_i \times time_i$
2) Subject to
   $x_0 + x_1 + \ldots + x_{h-1} + x_h = w(E)$,
   $x_i \in Z^+$ for $i \in [0, h-1]$, and
   $\{x_0, x_1, \ldots, x_{h-1}\}$ is derived from a load balanced thread schedule that distributes the set of threads T across n processors evenly.

Note that the values of x_i's are also constrained by load-balanced thread schedules.

By the following Lemma 3, the classic graph partitioning problem can be reduced to the problem of minimizing $cost(G, M, A)$ if M's edges have the same weight that is in turn reducible to the problem of minimizing $cost(G, M, A)$ for arbitrary M. Since the classic graph partitioning problem is NP-hard, finding an optimal schedule to minimize $cost(G, M, A)$ is also NP-hard.

Lemma 3. *Given a graph G and a shared-memory machine M with n processors. If $time_0 = time_1 = \ldots = time_{h-1}$ on M, P^* is an optimal n-way classic graph partitioning for G iff A^* is an optimal thread schedule for M, where P^* and A^* are of the same family of subsets.*

Proof Sketch: First we need to show: if P^* is an optimal n-way classic graph partitioning, then A^* is an optimal thread schedule. Let $P^* = \{T_1, \ldots, T_n\}$ be an optimal n-way partition of graph G, if $time_0 = \ldots = time_{h-1} = c$, then

$$\sum_{T_i,T_j \in P^*} \sum_{\substack{u \in T_i \\ v \in T_j}} w_e(u,v)c = \min_{\forall P} \sum_{T_i,T_j \in P} \sum_{\substack{u \in T_i \\ v \in T_j}} w_e(u,v)c.$$

Suppose schedule A^* has the same partition as P^* such that $A^*(T_i) = p_j$, then

$$\sum_{T_i,T_j \in P^*} \sum_{\substack{u \in T_i \\ v \in T_j}} w_e(u,v)c = \sum_{p_i,p_j} \sum_{\substack{u \in {A^*}^{-1}(p_i) \\ v \in {A^*}^{-1}(p_j)}} w_e(u,v)c.$$

$\forall$ partition P,

$$\sum_{T_i,T_j \in P} \sum_{\substack{u \in T_i \\ v \in T_j}} w_e(u,v)c = \sum_{p_i,p_j} \sum_{\substack{u \in A^{-1}(p_i) \\ v \in A^{-1}(p_j)}} w_e(u,v)c,$$

$$\sum_{p_i,p_j} \sum_{\substack{u \in {A^*}^{-1}(p_i) \\ v \in {A^*}^{-1}(p_j)}} w_e(u,v)c = \min_{\forall A} \sum_{p_i,p_j} \sum_{\substack{u \in A^{-1}(p_i) \\ v \in A^{-1}(p_j)}} w_e(u,v)c$$

The converse can be proved in a similar manner. ■

B. Hierarchical Partitioning Approximation Algorithm

Similar to *cut* in the classic graph partitioning problem, we use *share* to express the affinity strength between two partitions:

$$share(T_x, T_y) = \sum_{\forall u \in T_x, \forall v \in T_y} w_e(u,v),$$

where T_x and T_y are two disjoint thread sets.

If processor cores on a system have different computational powers, we use a *partition distribution vector* to define Unbalanced Graph Partitioning. Given affinity graph $G = \langle T, E, w_t, w_e \rangle$ and $W = w_t(T)$, the partition distribution vector $\langle d_1, d_2, \ldots, d_n \rangle$ defines a partition $\{P_i\}$ whose weight $w_t(P_i) = d_i \times W$ and $\sum_i d_i = 1$. A more powerful processor core will be assigned a larger portion of the computational tasks accordingly. The graph partitioning algorithm uses the partition distribution vector to guarantee that the workload on each core is load balanced.

There are two optimization goals in our partitioning process: conforming to the partition distribution vector and minimizing the sharing between partitions. We propose a hierarchical partitioning algorithm to divide the affinity graph according to the partition distribution vector. The goal is to minimize the sharing between partitions in the order of level 0 to level $h-1$ in a top-down fashion.

Lemma 4. *Let n be the number of partitions, G be a graph, and the system M has a height h. Assume P^* is an optimal n-way classic graph partitioning. The hierarchical graph partitioning algorithm can find a $(2,n)$-way graph partition P such that*

$$\frac{cost(P)}{cost(P^*)} \leq h, \text{ where}$$

$$cost(P) = \sum_{T_i,T_j \in P} share(T_i, T_j).$$

Proof: The conclusion can be drawn directly from Theorem 5.2 presented in paper [15]. ■

Theorem 2. *Suppose an optimal thread schedule A^* has $cost(G, M, A^*)$. Then the hierarchical graph partitioning algorithm can find a schedule A such that*

$$\frac{cost(G,M,A)}{cost(G,M,A^*)} \leq h \frac{time_0}{time_{h-1}}.$$

Proof Sketch: Firstly we prove

$$\frac{cost(G,M,A)}{cos(G,M,A^*)} \leq \frac{time_0 \sum_{i=0}^{h-1} SharingOnLevel_i}{time_{h-1} \sum_{i=0}^{h-1} SharingOnLevel_i^*}.$$

By Lemma 4,

$$\frac{cost(G,M,A)}{cos(G,M,A^*)} \leq h \frac{time_0}{time_{h-1}}.$$

■

V. Extension to Support DAG Scheduling

Our previous hierarchical partitioning algorithm described in [7] assumed threads have no data dependence. When threads are dependent on each other and hence form a DAG, we need to extend the algorithm to deal with DAG scheduling. In the extension, given DAG G, we divide G into a number of levels (horizontally), each of which consists of a subset of independent threads. This step can be achieved by analyzing G and determining the longest path from the root to each node. The total number of levels is equal to the length of the critical path. Within each level, we use the the hierarchical graph partitioning algorithm to determine a good schedule to run the threads. Due to data dependences, no thread in level $i+1$ can start until all threads in level i complete. We call this simple approach ”greedy multi-level thread scheduling”. Figure 5 depicts how to divide a DAG into four levels.

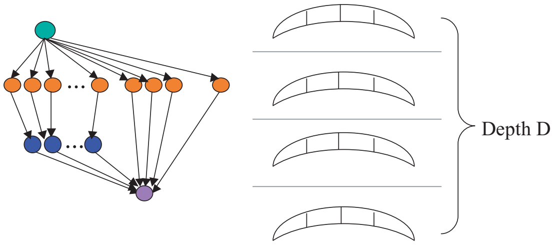

Fig. 5. An example of greedy multi-level thread scheduling. The DAG is divided into four levels. The level index of each node is equal to the length of the longest path from the root to that node.

Lemma 5. *Suppose a DAG has D levels and the total amount of computation is W. If each thread t_i computes an amount $w(t_i)$ of work, where $w(t_i) \in [0,1]$, then the greedy multi-level thread scheduler for p processors takes at most $\frac{W-D}{p}+D$ time.*

Proof: Let s_i denote the amount of work on level i, where $i \in [1, D]$.

$$Time = \sum_{i=1}^{D} \lceil \frac{s_i}{p} \rceil \leq \sum_{i=1}^{D} (\frac{s_i}{p} + (1 - \frac{1}{p}))$$

$$= \frac{W}{p} + D - \frac{D}{p} = \frac{W-D}{p} + D$$

■

Theorem 3. *The greedy multi-level thread scheduling method has an approximation ratio of $1 + \frac{D}{(\frac{W}{p})}$.*

Proof: Let C and C^* represent the actual execution time and the optimal execution time, respectively. It is easy to show that all the execution time is at least $\max(W/p, T_\infty)$.

$$\text{By } C \leq \frac{W-D}{p} + D \text{ and } C^* \geq \frac{W}{p},$$

$$\frac{C}{C^*} \leq \frac{(W-D)/p + D}{C^*} \leq \frac{(W-D)/p + D}{(\frac{W}{p})}$$

$$= 1 + \frac{(1-1/p)D}{(\frac{W}{p})} < 1 + \frac{D}{(\frac{W}{p})}$$

■

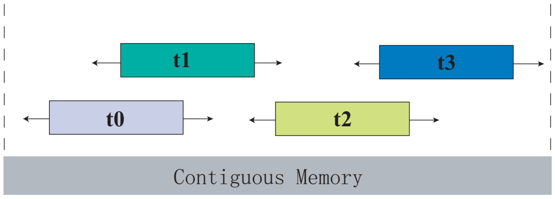

Fig. 6. Four threads access a contiguous memory of size 128MB. Each thread occupies 32MB and their locations are arbitrary.

Based on Theorem 3, if $W/D > p$, the greedy multi-level scheduling method takes time at most twice the optimal time. Programs with fine-grain threads often satisfy $W/D > p$ and are commonly found in scientific applications such as *Cholesky*, *LU*, and *QR* factorizations. Section VI-C2 shows the experimental results for Cholesky factorization. Since the new thread schedule improves both load balance and data locality, its efficiency is high.

VI. Experimental Evaluation

This section describes how we evaluate the analytical model, and reports the performance of two scientific applications to which we apply optimized thread schedules. The optimized schedules are computed by the hierarchical partitioning algorithm.

A. Evaluating the Analytical Model

To evaluate whether or not the analytical model could correctly estimate the cost of a thread schedule, we conducted three experiments on a DSM machine (SGI Altix) that has two compute nodes, each of which has two processors. In the experiments, we ran four threads on four processors. In terms of complexity, the three experiments range from simple to synthetic to real-world applications.

For the first two synthetic experiments, we allocate a contiguous memory block of size 128M bytes. Each thread only accesses $1/4$ of the 128MB memory. The thread first initializes the memory with some values and then computes the sum of the square of each element. The location of the memory segment could be anywhere as long as it is within the range of the 128MB memory. The affinity strength between two threads is equal to the size of the overlapping area between their footprints. Figure 6 illustrates how four threads could access a block of 128MB memory.

The first experiment is the simplest one where the memory segments of the four threads are disjoint initially (i.e., evenly distributed and affinity=0). Then we gradually move thread $t1$ towards thread $t0$ so that the overlapping area of $t0$ and $t1$ becomes bigger and bigger. Since there is no affinity change among the four threads except for the pair of $t0$ and $t1$, we only compare two thread placements: placing $t0$ and $t1$

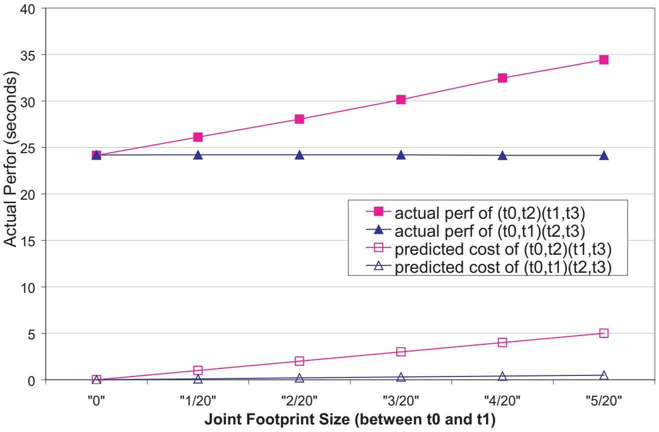

Fig. 7. Compare the predicted cost to the actual execution time for two schedules: `(t0,t1)(t2,t3)` and `(t0,t2)(t1,t3)`.

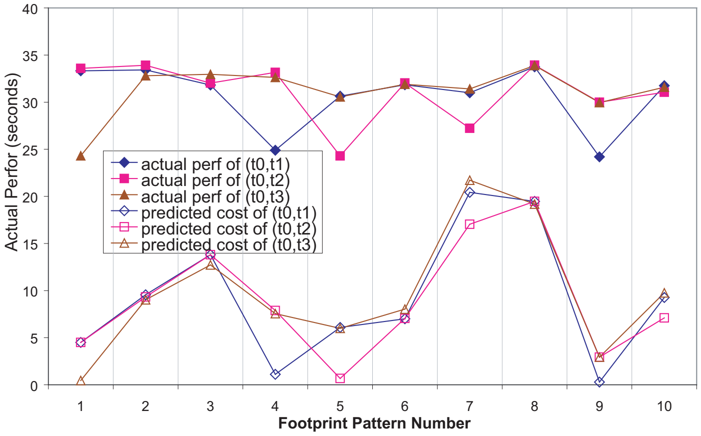

Fig. 8. Compare the predicted cost to the actual execution time for three thread schedules on ten randomly generated memory footprint patterns.

together on the same node (i.e., `(t0,t1)(t2,t3)`), and placing them separately (i.e., `(t0,t2)(t1,t3)`).

Figure 7 shows that the actual performance (i.e., execution time in seconds) of the placement `(t0,t2)(t1,t3)` becomes worse and worse with the increment of the overlapping footprint. The cost estimated by the analytical model has the same trend as the actual performance. Note that the cost model is not intended to predict the execution time. Rather, it is used to measure the quality of a thread schedule, and a ranking is sufficient to find the best thread schedule. The estimated cost is able to reflect how much data reuse a thread schedule has. But it is not the total execution time because estimating execution time also requires we predict the cache miss rates on the machine and measure the parameter of $time_i$ on each memory level i.

In the second experiment as shown in Fig. 8, we performed ten program runs each of which had a different footprint pattern. All the footprint patterns were generated randomly. To generate a pattern, we used a random number generator to create a starting position $addr_i$ for each thread t_i such that t_i accesses addresses in the range of $[addr_i, addr_i + 32MB)$. Given 4 threads on two SMP nodes each with 2 processors, there are totally $\binom{4}{2}/2! = 3$ thread schedules. We denote them as `(t0, t1)`, `(t0, t2)`, and `(t0,t3)`, respectively. For each of the ten footprint patterns, we ran the same program three times each with a different thread schedule. We also estimated the cost of every run and compared it to the actual performance.

From Fig. 8, we can see that the cost of the three schedules consistently reflects the ranking of their actual program performance. In other words, given any footprint pattern, if schedule A has a cost higher than schedule B, the actual performance of the program using schedule A is worse than that using schedule B.

As another example, we applied the analytical model to an important kernel in many scientific applications: sparse matrix vector product (`SpMV`). The sparse matrices were downloaded from the UF Sparse Matrix Collection [16]. For many matrices, we can improve the program performance by 15% to 25%. For a few other matrices, the program using the new thread schedule instead runs 1% slower than the original program. The complete experimental result can be found in our previous work [7].

We used the analytical model to analyze this phenomenon and investigate the differences between the old and the new thread schedules. We picked two cases from our experiments. One experiment multiplies the sparse matrix `msc01440` and improves the performance by 23%. The other one multiplies the sparse matrix `circuit_1` but is slower than the original program by 1%. Tables I and II list the values of $SharingOnLevel_i$ (see Theorem 1) for the original and new schedules, respectively. Assuming the remote memory access time is $time_0$ and the local memory access time is $time_1$, we can compute $cost(G, M, A)$ by adding $time_0 \times SharingOnLevel_0$ and $time_1 \times ShareingOnLevel_1$. As shown in Table I, the new schedule reduces the remote memory access cost by 68.5% and thus improves program performance. However, based on Table II, there is only a small reduction by using the new schedule (4.4% in remote memory accesses). Due to the overhead of executing the new schedule, the actual performance shows a 1% slowdown instead of a small speedup.

TABLE I
APPLYING THE ANALYTICAL MODEL TO STUDY WHY `SpMV` WAS IMPROVED WITH INPUT `msc01440`.

Affinity on diff. levels	Original schedule	New schedule	Reduction
Remote memory	54,175	17,043	68.5%
Local memory	107,765	13,964	87.0%

B. Applications

We applied the hierarchical graph partitioning algorithm to two applications to find optimized thread schedules to improve the program performance.

TABLE II
APPLYING THE ANALYTICAL MODEL TO STUDY WHY SpMV WAS NOT IMPROVED WITH INPUT circuit_1.

Affinity on diff. levels	Original schedule	New schedule	Reduction
Remote memory	127,620	121,973	4.4%
Local memory	230,759	206,076	10.7%

1) Computational Fluid Dynamics (CFD) Kernel: The CFD kernel implements an iterative irregular-mesh partial differential equation (PDE) solver abstracted from computational fluid dynamics applications [11]. The irregular meshes are used to model physical structures and consist of n_v vertices and n_e edges, denoted by $\langle n_v, n_e \rangle$. The kernel iterates over the edges of the mesh, computing the forces between both end points of each edge. It then modifies the values on the vertices. Our previous work also did experiments on the CFD kernel but used only four processors on a single platform [7]. The parallel version of the kernel has the structure shown in Fig. 9. Each edge corresponds to a user-level thread during the execution.

```
for iter = 1, NUM_ITER
  #pragma omp parallel for
  for i = 1, num_edges
    v1 = left[i];
    v2 = right[i];
    force = f(x[v1],x[v2]);
    y[v1] += force;
    y[v2] -= force;
  end for
end for
```

Fig. 9. Parallel version of the CFD kernel.

2) Cholesky Factorization: Given an $n \times n$ symmetric positive definite matrix A, Cholesky factorization computes $A = LL^T$ where L is an $n \times n$ lower triangular matrix. For efficiency, we implemented a right-looking blocked algorithm so that we can apply Level-3 BLAS directly to a block of matrix A. The blocked Cholesky factorization algorithm works as follows:

$$\text{Given } A = \begin{pmatrix} A_{1:b,1:b} & A_{1:b,b+1,n} \\ A_{b+1:n,1:b} & A_{b+1:n,b+1:n} \end{pmatrix},$$

$$\text{We compute } L = \begin{pmatrix} L_{1:b,1:b} & 0 \\ L_{b+1:n,1:b} & L_{b+1:n,b+1:n} \end{pmatrix}$$

by calling:
(i) level-3 BLAS POTRF to solve $L_{1:b,1:b}$,

$$A_{1:b,1:b} = L_{1:b,1:b} L^T_{1:b,1:b}$$

(ii) level-3 BLAS TRSM to solve a linear equation system for $L_{b+1:n,1:b}$ in parallel,

$$L_{b+1:n,1:b} L^T_{1:b,1:b} = A_{b+1:n,1:b}$$

(iii) level-3 BLAS GEMM to compute a rank-r update on the trailing matrix $A_{b+1:n,b+1:n}$ in parallel,

$$A'_{b+1:n,b+1:n} \leftarrow A_{b+1:n,b+1:n} - = L_{b+1:n,1:b} L^T_{b+1:n,1:b}$$

We apply the above 3 steps repeatedly to $A'_{b+1:n,b+1:n}$ until A' consists of a single $b \times b$ block. The code is shown in Fig. 10. Variable A_ij refers to a block which is located in the ith row and jth column in terms of blocks. Given an $n \times n$ matrix and a block of size b, nblocks = n/b. To use the multilevel algorithm described in Section V, we must know the program's task graph. Each task in the DAG corresponds to a Level-3 BLAS operation. Figure 11 shows the corresponding task graph for a 4 block by 4 block matrix and its level division. The figure displays only one iteration of the outer loop.

```
for k = 1, nblocks
  dpotf2(A_kk);
  #pragma omp parallel for
  for j = k+1, nblocks
    dtrsm(A_kk, A_jk);
  end for
  for i = k+1, nblocks
    #pragma omp parallel for
    for j = k+1, i
      dgemm(A_ik, A_jk, A_ij);
    end for
  end for
end for
```

Fig. 10. Parallel tiled Cholesky factorization.

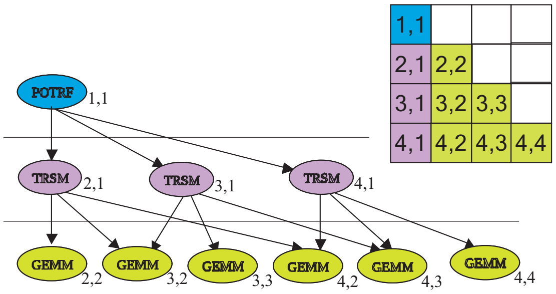

Fig. 11. DAG for Cholesky factorization (one iteration).

C. Experimental Results

We conducted all the experiments on two platforms. One platform is a single SMP machine consisting of two sockets, each of which has a quad-core 2.66 GHZ Intel Clovertown chip. Since the set of two cores on each chip share an L2 cache, the corresponding memory hierarchy has two levels: the main memory on the machine and the L2 caches on each chip. The other platform is an SGI Altix 3700 BX2 system with 256 compute nodes. Each node has two 1.6 GHZ Intel Itanium processors. The system has a ccNUMA Distributed Shared Memory (DSM) that is physically distributed across different nodes. Every processor can access any memory location through the SGI NUMAlink 4 interconnect. The memory access time depends on the distance between the processor and the node where the physical memory is located. The corresponding memory hierarchy also has two levels: the virtual global memory and memories on each compute node.

For each program of the CFD kernel and Cholesky factorization, we always compare the performance of the program using the optimized thread schedule to that of the program built by compiler optimizations. Since the SMP machine has a fixed number of eight cores, we vary the input size to run experiments. On the DSM machine, we chose to vary the number of processors to compare the program performance.

1) CFD Kernel Performance: Over a number of irregular meshes, we compare the total execution time of the new program using the new thread schedule to that of the original program. For our examples, a mesh always has 10 times more edges than vertices. Figure 12 shows that using the optimized thread schedule reduces the execution time by 25% to 35% on the Intel Clovertown SMP machine.

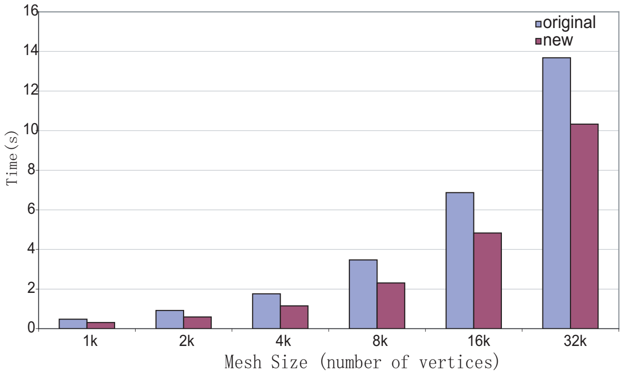

Fig. 12. CFD kernel on Intel Clovertown.

On the SGI DSM machine, the program always takes as input a mesh of $40,000$ vertices and $400,000$ edges. For various numbers of processors (i.e., 4, 8, 16, and 32), our method reduces the execution time by 32% to 42%, as depicted in Fig. 13.

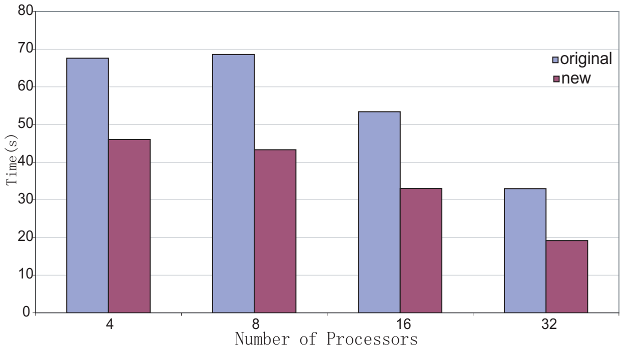

Fig. 13. CFD kernel on SGI Altix.

2) Cholesky Factorization Performance: Unlike the CFD kernel program with independent threads, Cholesky factorization has threads with data dependences. The greedy multilevel thread scheduling method is used to determine an optimized schedule for the corresponding DAG level by level. Compared to the original schedule that allocates threads to processors in a block distribution way, the new schedule improves not only data locality but also load balance. The related numerical results have been verified.

For comparison, Fig. 14 also displays the Intel MKL 9.1 performance. On the Intel Clovertown machine, we can see that the new program is 60% to 200% faster than the original one, while the MKL library always provides better performance than the original one. On the SGI machine, we conducted experiments using different numbers of processors (4, 8, 16) and compared the performance with that of MKL 7.2. Each experiment takes as input matrices with different sizes. Figures 15, 16, and 17 demonstrate that the new program is faster than the original program by 30% to 400% for different number of processors. The speedup on SGI Altix is greater than that on Intel Clovertown is because the optimized schedule on Altix not only reduces the number of cache misses, but also reduces the number of expensive remote memory accesses due to the NUMA architecture.

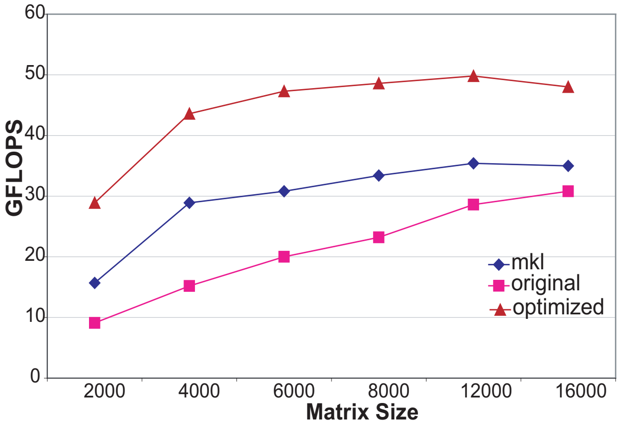

Fig. 14. Cholesky factorization on Intel Clovertown.

VII. Conclusion

Improving memory effectiveness is an important technique to achieve high program performance. While there exist tools and runtime systems to schedule threads efficiently, little is known about what would be an optimal affinity thread schedule to maximize the memory effectiveness and why it is optimal. We present an analytical model to evaluate the performance of a thread schedule. The model has three submodels: an affinity graph submodel to describe the affinity relationship between threads, a memory hierarchy submodel to characterize the underlying shared-memory architecture, and a cost submodel to estimate the cost of a certain thread schedule. The experimental results show that the analytical model can accurately estimate the cost of a thread schedule. We also propose a hierarchical graph partitioning algorithm to find near-optimal solutions. We applied the hierarchial partitioning algorithm that has been extended to support DAGs to two applications: computational fluid dynamics (CFD) kernel and Cholesky factorization. Experiments on both SMP and DSM machines show that using the optimized schedule is able to improve program performance by 25% to 400%, demonstrating that our model and algorithms are practical and efficient.

Acknowledgment

This work has been partially supported by the Department of Energy Office of Science under grant No. DE-FC02-06ER25761, and by Microsoft Research.

References

[1] R. Golla, “Niagara2: A highly threaded server-on-a-chip,” 2007.

[2] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, “Larrabee: a many-core X86 architecture for visual computing,” *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, 2008.

[3] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O’Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden, “IBM Power6 microarchitecture,” *IBM J. Res. Dev.*, vol. 51, no. 6, pp. 639–662, 2007.

[4] S. Balakrishnan, R. Rajwar, M. Upton, and K. K. Lai, “The impact of performance asymmetry in emerging multicore architectures.” in *ISCA*. IEEE Computer Society, 2005, pp. 506–517.

[5] R. Kumar, D. M. Tullsen, and N. P. Jouppi, “Core architecture optimization for heterogeneous chip multiprocessors.” in *PACT*, E. R. Altman, K. Skadron, and B. G. Zorn, Eds. ACM, 2006, pp. 23–32.

[6] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, “Single-ISA heterogeneous multi-core architectures for multithreaded workload performance.” in *ISCA*. IEEE Computer Society, 2004, pp. 64–75.

[7] F. Song, S. Moore, and J. Dongarra, “Feedback-directed thread scheduling with memory considerations,” in *HPDC ’07: Proceedings of the 16th international symposium on High performance distributed computing*, 2007, pp. 97–106.

[8] F. Song, S. Moore, and J. Dongarra, “Analytical modeling for affinity-based thread scheduling on multicore platforms,” University of Tennessee, Computer Science Tech. Rep. UT-CS-08-626, 2008.

[9] H. El-Rewini, T. G. Lewis, and H. H. Ali, *Task scheduling in parallel and distributed systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.

[10] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li, “Thread scheduling for cache locality.” in *ASPLOS*, 1996, pp. 60–71.

[11] V. K. Pingali, S. A. McKee, W. C. Hsieh, and J. B. Carter, “Restructuring computations for temporal data cache locality.” *International Journal of Parallel Programming*, vol. 31, no. 4, pp. 305–338, 2003.

[12] J. L. Träff, “Implementing the MPI process topology mechanism,” in *Supercomputing ’02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–14.

[13] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera, “Improving the locality of the sparse matrix-vector product on shared memory multiprocessors.” in *PDP*. IEEE Computer Society, 2004, pp. 66–71.

[14] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera, “A new technique to reduce false sharing in parallel irregular codes based on distance functions,” in *ISPAN*. IEEE Computer Society, 2005, pp. 306–311.

[15] H. D. Simon and S.-H. Teng, “How good is recursive bisection?” *SIAM J. Sci. Comput.*, vol. 18, no. 5, pp. 1436–1445, 1997.

[16] T. Davis, “University of Florida sparse matrix collection.” [Online]. Available: http://www.cise.ufl.edu/research/sparse

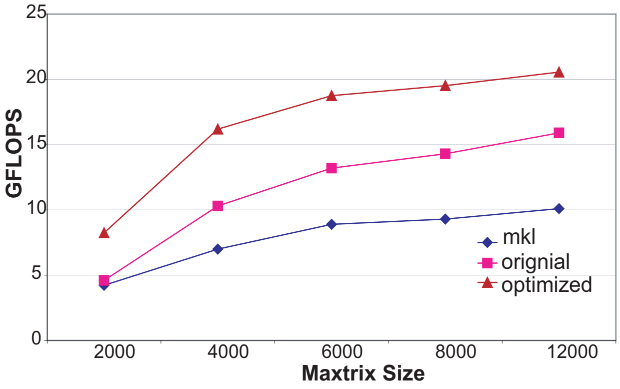

Fig. 15. Cholesky factorization on SGI Altix with 4 processors.

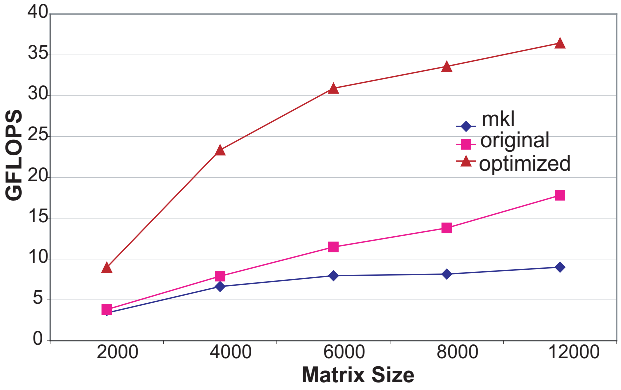

Fig. 16. Cholesky factorization on SGI Altix with 8 processors.

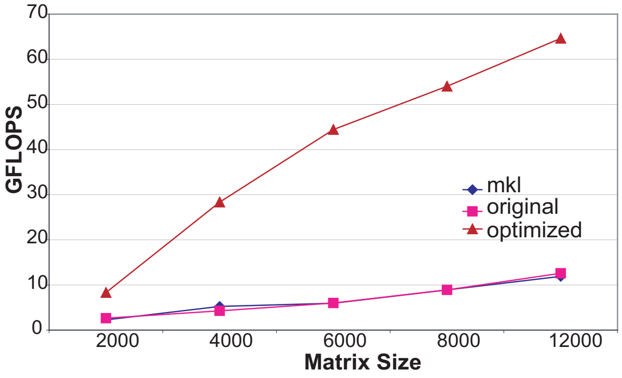

Fig. 17. Cholesky factorization on SGI Altix with 16 processors.