Analytical Modeling for Affinity-Based Thread Scheduling on Multicore Platforms [*]

Fengguang Song
University of Tennessee
EECS Department
Knoxville, TN
song@eecs.utk.edu

Shirley Moore
University of Tennessee
EECS Department
Knoxville, TN
shirley@eecs.utk.edu

Jack Dongarra
University of Tennessee
EECS Department
Knoxville, TN
dongarra@eecs.utk.edu

ABSTRACT
This paper proposes an analytical model to estimate the cost of running an affinity-based thread schedule on multicore shared-memory systems. This model considers a memory architecture as a generic tree structure and allows for a portable, architecture-aware optimization framework to find an optimized schedule for multi-threaded programs. It consists of three submodels in order to measure the cost of executing a thread schedule: affinity graph model, memory hierarchy model, and a cost model that characterize machines, programs, and costs respectively. With the aid of the model, we formalize the problem of finding the best thread schedule as an optimization problem. Due to the NP-hardness of the problem, we designed a hierarchical graph partitioning algorithm to compute an approximate solution. We then extended the algorithm to support threads with data dependencies (i.e., DAGs). The algorithm has been implemented in a feedback-directed optimization framework and applied to two real-world scientific applications: a Computational Fluid Dynamics (CFD) kernel and Cholesky factorization. We conducted our experiments on both SMP and DSM machines. The results show that our analytical model is accurate enough, and using the optimized thread schedule improves the program performance by 25% to 4 times, demonstrating that our method is efficient and practical.

1. INTRODUCTION
The shared-memory programming paradigm has been widely accepted and used for a long time. It surely makes the development work much easier. Modern large scale shared-memory machines often have a global address space that is distributed across hundreds of compute nodes. With the emergence of chip multi-processors (CMP) [3, 9, 15], the future DSM system will become smaller and has tens of thousands of processor cores. Performance asymmetry in multicore platforms is clearly another trend due to budget issues such as power consumption and area limitation as well as various degree of parallelism in applications [1, 4, 5, 6]. We call such a system "heterogeneous manycore DSM system" (in Figure 1). Processor cores belonging to the same level (e.g., same chip or board) share a certain amount of memory. For instance, a couple of cores on the same chip may share an L2 or L3 cache.

[*]This version includes the complete proofs of our theorems.

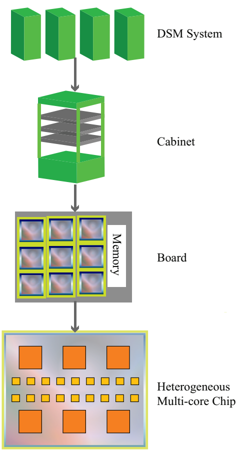


Figure 1: Heterogeneous manycore DSM system under study.

In this paper we investigate how to improve the memory effectiveness and maximize data reuse through affinity-based thread scheduling on the manycore shared memory platforms. While working on a large-scale DSM machine for a couple of years, we observed that it is critical for us to find a method to improve user programs' memory access efficiency. Figure 2 demonstrates that some applications are experiencing a large amount of remote memory accesses. The light yellow area indicates the number of remote memory accesses.

Therefore, we attempt to search for an optimal thread schedule to improve the memory effectiveness on all levels in the multi-level memory hierarchy. To investigate the affinity-based thread scheduling, we first propose an analytical model to estimate the cost of a thread schedule and then tackle it as an optimization problem. In particular, the analytical model consists of three submodels:

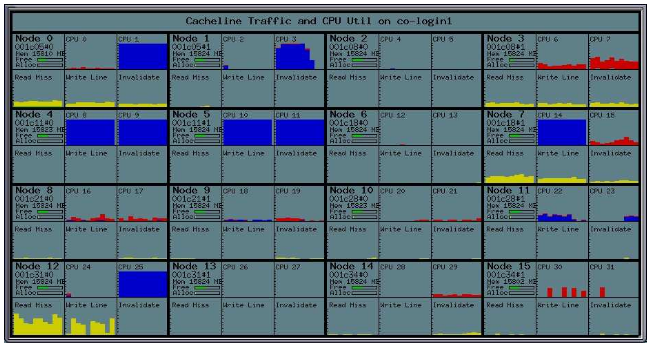

Cacheline Traffic and CPU Util on co-login1
Node 0 001c05#0 Mem 15810 MB Free Alloc CPU 0 CPU 1
Node 1 001c05#1 Mem 15824 MB Free Alloc CPU 2 CPU 3
Node 2 001c08#0 Mem 15824 MB Free Alloc CPU 4 CPU 5
Node 3 001c08#1 Mem 15824 MB Free Alloc CPU 6 CPU 7
Node 4 001c11#0 Mem 15823 MB Free Alloc CPU 8 CPU 9
Node 5 001c11#1 Mem 15824 MB Free Alloc CPU 10 CPU 11
Node 6 001c18#0 Mem 15824 MB Free Alloc CPU 12 CPU 13
Node 7 001c18#1 Mem 15824 MB Free Alloc CPU 14 CPU 15
Node 8 001c21#0 Mem 15824 MB Free Alloc CPU 16 CPU 17
Node 9 001c21#1 Mem 15824 MB Free Alloc CPU 18 CPU 19
Node 10 001c28#0 Mem 15823 MB Free Alloc CPU 20 CPU 21
Node 11 001c28#1 Mem 15824 MB Free Alloc CPU 22 CPU 23
Node 12 001c31#0 Mem 15824 MB Free Alloc CPU 24 CPU 25
Node 13 001c31#1 Mem 15824 MB Free Alloc CPU 26 CPU 27
Node 14 001c34#0 Mem 15824 MB Free Alloc CPU 28 CPU 29
Node 15 001c34#1 Mem 15802 MB Free Alloc CPU 30 CPU 31
Read Miss
Write Line
Invalidate

Figure 2: A screen shot of a performance monitoring tool on SGI Altix. The light yellow area reflects how many number of remote memory accesses have occurred.

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

Our strategy is to let the affinity graph model characterize the user program and the memory hierarchy model characterize the machine architecture. Finally in combination of the cost model, 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 program T on machine M?” Since finding an optimal thread schedule is NP-hard, we propose a hierarchical graph partitioning algorithm to compute an approximate solution. Furthermore, an extension to support threads with data dependencies (DAG scheduling) is also developed.

The analytical model is supported experimentally. We applied the model to two synthetic experiments and a real application and show that it can accurately measure the quality of a thread schedule. We also implemented a tool to compute an optimized thread schedule based on the hierarchical graph partitioning algorithm. We deployed the tool to 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 method is able to improve the program performance greatly (by 25% to 42% for the CFD kernel and 30% to 400% for Cholesky factorization).

The paper is organized as follows. Section 2 describes the analytical model and the three submodels in sequence. Section 3 proposes a hierarchical graph partitioning algorithm to find an approximate solution to the optimization problem. An extension to support threads with data dependencies and a framework to implement the algorithms are briefly described in Sections 4 and 5. Section 6 presents the experimental results. Section 7 introduces the related work. Finally, Section 8 concludes our work.

2. THE ANALYTICAL MODEL

2.1 Affinity-Based Thread Scheduling Problem

Given a set of single-application user-level threads $\{t_1, \ldots, t_m\}$ without data dependencies, 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.

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 each thread to have different workload and each processor to be heterogeneous with various computational capability.

2.2 Affinity Graph Model

In order to decide on which processors to place two threads, it is critical to know whether there exist data accessed in common by the two threads. If none of these data exist, we can place the threads freely regardless of data reuse. Therefore, we introduce the concept of ”affinity” to quantify how many data are accessed in common by each pair of threads.

Definition 1 (Affinity). *When two threads t_i, t_j access a number n of data $\{x_1, x_2, \ldots, x_n\}$ in common, we say there is an affinity relationship between t_i and t_j, and affinity$(t_i, t_j) = n$ is the strength of affinity.*

Since a user program has a set of threads, we introduce the concept of *affinity graph* to model the affinity relationship between the set of threads.

Definition 2 (Affinity Graph). *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 $t_j, \forall i \neq j\}$,*
- *$E = \{(t_i, t_j) \mid \exists$ data 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 $w_e(t_i, t_j) = 0$.*

Given an affinity graph $G = \langle T, E, w_t, w_e \rangle$, if $T_i \subset T$, we extend the definition of w_t and w_e to represent the weight of the subgraph for T_i. That is, $w_t(T_i) = \sum_{t \in T_i} w_t(t)$ and $w_e(T_i) = \sum_{t_i, t_j \in T_i} w_e(t_i, t_j)$.

Figure 3 shows an example of multiplying two 200×200 matrices. A and C are dense matrices. Matrix B has a special structure where the top right and bottom left blocks are all zeros. We run four threads T0-T3 to compute the matrix multiplication. T0-T3 compute the result for submatrices of C11, C12, C21, and C22 concurrently. The corresponding affinity graph is shown on the right hand side in Figure 3.

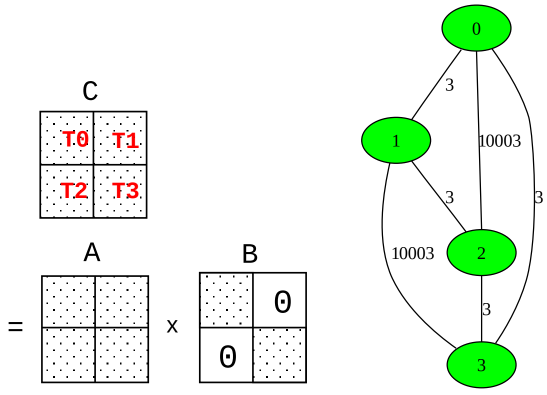


Figure 3: Parallel matrix multiplication using four threads and its corresponding affinity graph. Each thread of T0-T3 computes one block of matrix C.

We compare the performance of putting threads T0 and T2 on the same SMP node to that of putting T0 and T1 together (an intuitive way) for a system with two dual-CPU SMP nodes. Figure 4 shows the wallclock execution time of the two thread schedules. The optimized schedule is better than the original one by 20%. This example demonstrates that different thread schedules can result in great performance difference.

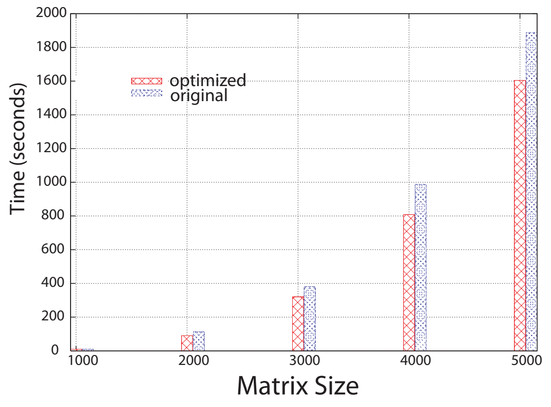


Figure 4: Comparison of two different thread schedules. The revealed affinity relationship (stronger affinity between T0 and T2) in Figure 3 helps improve the performance of the parallel matrix multiplication.

2.3 Memory-Hierarchy Model

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. The data stored in the L2 and L3 caches are duplicated in the local main memory. Also the global address space includes all the data stored in the local memories. We define such a hierarchical shared memory system as follows:

Definition 3 (Shared-Memory System). *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 5 shows an example of a DSM system.

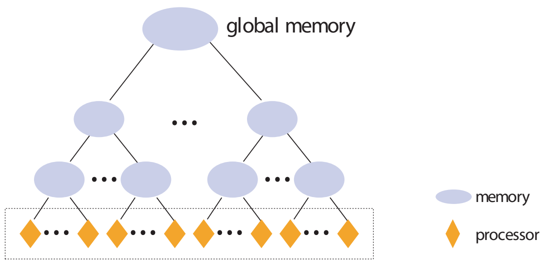


Figure 5: A 3-level memory hierarchy on a multicore DSM system.

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$\}$.

Definition 4 (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*

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

Lemma 1. *Let datum x reside in memory m and its 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 4,

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

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

And by $level(m2) < level(m)$, we get $lat(p', x) > lat(p, x)$. $\square$

Corollary 1. *If two threads t_i and t_j access the same datum x stored in memory m and its ancestor memories, 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. $\square$

Corollary 1 shows that the thread placement may affect a program's performance if the threads have an affinity relationship.

2.4 Cost Model

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

Definition 5 (Cost Model). *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*

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

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

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), $i = 1 \ldots k$. 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 of set

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

Proof. We want to show (1) $\bigcup_{m \in M} pair(m) = \mathcal{P}$, and (2) $pair(m_i) \cap pair(m_j)) = \emptyset$.

(1.1) To prove $\bigcup_{m \in M} pair(m) \subseteq \mathcal{P}$.

Let $\{p_x, p_y\} \in pair(m_i)$ for a certain $m_i \in M$. By definition of $pair(m_i)$, we know $p_x \in D(m_i)_{k_1}$ and $p_y \in D(m_i)_{k_2}$. Sine M is a tree, $p_x \neq p_y$. Therefore, $(p_x, p_y) \in \mathcal{P}$, such that $\bigcup_{m \in M} pair(m) \subseteq \mathcal{P}$.

(1.2) To prove $\mathcal{P} \subseteq \bigcup_{m \in M} pair(m)$.

Let $\{p_x, p_y\} \in \mathcal{P}$ and m_c be the lowest common ancestor of p_x and p_y. Assume m_c has k subsets $D(m_c)_i$ of processors that are derived from its k children (or branches), then p_x and p_y must belong to different branches of m_c (since otherwise m_c won't be the lowest common ancestor). WLOG, let $p_x \in D(m_c)_{k_1}$ and $p_y \in D(m_c)_{k_2}$, $k_1 \neq k_2$. Therefore, $(p_x, p_y) \in pair(m_c)$, such that $\mathcal{P} \subseteq \bigcup_{m \in M} pair(m)$.

(2) To prove $pair(m_i) \cap pair(m_j) = \emptyset$ if $m_i \neq m_j$.

Suppose $\exists (p_x, p_y) \in pair(m_i) \cap pair(m_j)$ and $m_i \neq m_j$. By definition of $pair$, m_i is the lowest common ancestor of p_x and p_y. Similarly, m_j is also the lowest common ancestor of p_x and p_y. Since p_x and p_y have a unique lowest common ancestor, thus $m_i = m_j$, contradicting the assumption of $m_i \neq m_j$. $\square$

Theorem 1. *Suppose a thread schedule A places the set of threads of affinity graph G to a system M. Let $time_l$ denote $lat(p, p$'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 another word,

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

$SharingOnLevel_l$ denotes the amount of affinity between threads that access memories 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. Suppose processors p_i and p_j have the lowest common ancestor memory $lca(p_i, p_j)$. By definition,

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

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

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

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

By Lemma 2, $\{pair(m)\}$ partitions $\mathcal{P}$, then

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

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

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

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

We know $time_{level(m)} \in \{time_0, time_1, \ldots, time_{h-1}\}$ and all memories $\in M_l$ have the same $time_l$,

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

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

$$\ldots$$

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

That is,

$$cost(G, M, A) = time_0 \times SharingOnLevel_0 +$$

$$time_1 \times SharingOnLevel_1 + \ldots$$

$$+time_{h-1} \times SharingOnLevel_{h-1}.$$

□

3. SOLVING THE 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.

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 distribute the set of threads T across n processors evenly.

Note that the values of x_i's are also constrained by certain thread schedules.

By the following Lemma 3, the classic graph partitioning problem can be reducible to the problem of minimizing $cost(G, M, A)$ if M's edges have the same weight that is also reducible to the problem of minimizing $cost(G, M, A)$ for arbitrary M's. 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 system M with n processors. If $time_0 = time_1 = \ldots = time_{h-1}$ on M, an optimal n-way classic graph partitioning P^* of G also minimizes cost(G,M,A) if schedule A uses the same partition as P^*.*

PROOF. Let $P^* = \{T_1, T_2, \ldots, T_n\}$ be an optimal n-way partition to graph G, then

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

Suppose $time_0 = time_1 = \ldots = time_{h-1} = c$,

$$\sum_{T_i, T_j \in P^*} \sum_{u \in T_i, v \in T_j} w_e(u, v)c = \min_{\forall P} \sum_{T_i, T_j \in P} \sum_{u \in T_i, v \in T_j} w_e(u, v)c \quad (1)$$

Since

$$cost(G, M, A) = \sum_{p_i, p_j} \sum_{u \in A^{-1}(p_i), v \in A^{-1}(p_j)} w_e(u, v)c$$

and for partition $P = \{T_1, T_2, \ldots, T_n\}$, we can construct a schedule A that has the same partition P such that $A(T_i) = p_j$, thus

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

and

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

By Equation 1,

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

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

That is,

$$cost(G, M, A^*) = \min_{\forall A} cost(G, M, A), \text{ where}$$

${A^*}^{-1}$ has the same partition as P^*. □

3.1 A Hierarchical Partitioning 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 .

Processors on a system may have different computational powers, hence 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 $\sum_i d_i = 1$.

We propose a greedy hierarchical partitioning algorithm to divide the affinity graph according to a partition distribution vector. The optimization goal is to minimize the sharing between partitions in an order from level 0 to level $h-1$. Assume a parallel system n compute nodes each of which has p processors. (a) We divide the threads into n sets $N_1, N_2, \ldots, N_n$ by minimizing

$$\sum_{1 \le i,j \le n} share(N_i, N_j), i \ne j.$$

Now each N_i has been assigned a set of threads. Since each compute node also has p processors, (b) we further partition N_i to p sets $P_1, P_2, \ldots, P_p$. Similarly, this is achieved by minimizing the sharing between two processors:

$$\sum_{1 \le i,j \le p} share(P_i, P_j), i \ne j.$$

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*

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

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

PROOF. The conclusion can be drawn directly from Theorem 5.2 presented in paper [14]. $\square$

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*

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

PROOF. By definition of $cost(G, M, A)$,

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

$$\le \frac{time_0 \sum_{i=0}^{h-1} SharingOnLevel_i}{time_{h-1} \sum_{i=0}^{h-1} SharingOnLevel_i^*}$$

Since $cost(P) = \sum_{i=0}^{h-1} SharingOnLevel_i$ if A^{-1} has the same partition as P, by Lemma 4,

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

$\square$

4. EXTENSION FOR DAG SCHEDULING

The previous hierarchical graph partitioning algorithm is able to find a good schedule for threads without data dependencies. However when threads are dependent on each other and hence form a DAG, we must extend the algorithm to deal with DAG scheduling. In our method, given a 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 itself. The total number of levels is equal to the length of the critical path of G. Within each level, we use the the hierarchical graph partitioning algorithm to determine a good schedule to run the threads on that level. Due to data dependencies, threads in level $i+1$ cannot start until threads in level i complete. We call this simple approach "greedy multi-level thread scheduling". Figure 6 depicts how to divide a DAG into four levels.

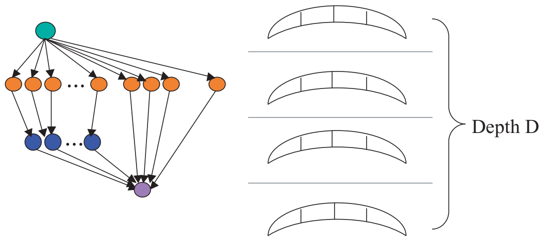


Figure 6: An example of greedy multi-level thread scheduling. The DAG is divided into four levels. The level index of each node is equal to the length of the longest path from root to itself.

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 $i \in [1, D]$.

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

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

$\square$

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 $\max(W/p, T_\infty)$.

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

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

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

$\square$

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 6.3.2 shows the performance result of our experiment on Cholesky factorization. Since the new schedule improves both load balance and data locality, its efficiency is high.

5. A FEEDBACK-DIRECTED OPTIMIZATION FRAMEWORK

We have designed a feedback-directed optimization tool based upon the analytical model and the hierarchical graph partitioning algorithm to schedule multi-threaded programs [16]. The tool uses a trace-based method and is able to determine a good schedule without writing to any file. The trace analysis cost is cheap (less than 100 seconds for our largest experiments).

The framework 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. This step is performed in memory and there is no disk IO involved. Next, (ii) we partition the affinity graph into a number of subgraphs at different levels corresponding to the system architecture. Based on the partitions (one subset of threads per processor), (iii) we use a breadth-first traversal method to compute a "good" schedule for each processor. The schedules are written in a file which will be later used as a feedback to the future executions. Finally, (iv) users run the program again taking as input the feedback file. For more details, interested readers can refer to [16]. Figure 7 illustrates the overall structure of the above process.

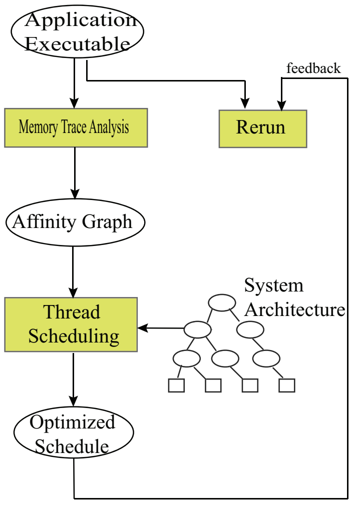


Figure 7: Overall structure of the feedback-directed affinity thread scheduling tool.

6. EXPERIMENTAL EVALUATION

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

6.1 Evaluate 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 are ranging from simple, synthetic to real-world applications.

For the first two synthetic experiments, we allocate a contiguous memory of size 128M bytes. Each thread only accesses 1/4 portion 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 8 illustrates how four threads could access a block of 128MB memory.

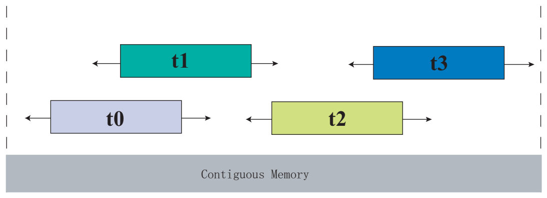


Figure 8: Four threads are accessing a contiguous memory of size 128MB. Each thread occupies 1/4 of it and their locations are arbitrary.

The first experiment is the simplest one where the memory segment of the four threads are disjoint (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 between the four threads except for the pair of $t0$ and $t1$, we only compare two thread placements: placing $t0$ and $t1$ together on the same node (i.e., `(t0,t1)(t2,t3)`), and placing them separately (i.e., `(t0,t2)(t1,t3)`). From Figure 9, we can see that the performance of the placement `(t0,t2)(t1,t3)` becomes worse and worse with the increment of the overlapping footprint. On the other hand, 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, but used to measure the quality of a thread schedule and a ranking is sufficient to find the best thread schedule.

In the second experiment, we performed ten program runs each of which has a different footprint pattern. All the footprint patterns were generated randomly. To generate a pattern, we use a random generator to create a starting position $addr_i$ for each thread t_i so that t_i accesses addresses in the range of $[addr_i, addr_i + 32MB)$. Given 4 threads and 2 SMP nodes each with 2 processors, there are totally $\binom{4}{2}/2! = 3$ thread schedules. We denote them as `(t0, t1)`, `(t0, t2)`,

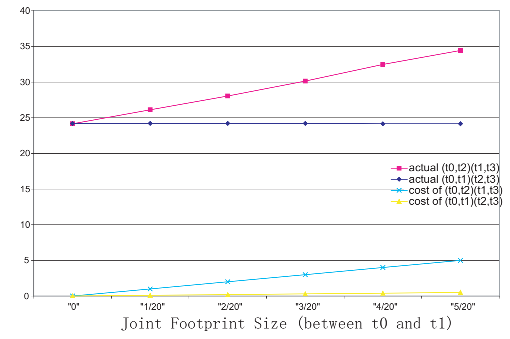


Figure 9: Compare the estimated cost to the execution time for two thread schedules: `(t0,t1)(t2,t3)` and `(t0,t2)(t1,t3)`.

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 compute the cost for every run and compare it to the actual performance. From Figure 10, we find that the cost of the three schedules consistently reflects the ranking of their actual program performance. In another word, given any footprint pattern, if schedule A has a cost bigger than schedule B, the actual performance of the program using schedule A will be slower than that using schedule B.

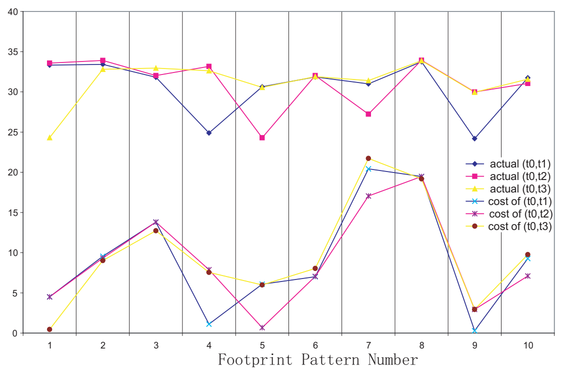


Figure 10: Compare the estimated cost to the actual performance of three thread schedules on ten randomly generated memory footprint patterns.

Finally, we applied the analytical model to a real-world application: sparse matrix vector product (`SpMV`). The sparse matrices were downloaded from the UF Sparse Matrix Collection [2]. 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 results can be found in our previous work [16]. Now we want to use the analytical model to analyze this phenomenon and investigate what is going on with the old and the new thread schedules.

We picked two cases from our experiments. One experiment multiplies the sparse matrix of `msc01440` and improves the performance by 23%. The other one multiplies the sparse matrix of `circuit_1` but is slower than the original program by 1%. Tables 1 and 2 list the values of $SharingOnLevel_i$ (see Theorem 1) for the original and new schedules, respectively. Assume 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 1, the new schedule reduces the remote memory access cost greatly by 68.5% and thus improves the program performance. However, based on Table 2, 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 1: 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%

Table 2: 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%

6.2 The Applications

We applied the hierarchical graph partitioning algorithm to two applications to find improved thread schedules.

6.2.1 Computational Fluid Dynamics (CFD) Kernel

The CFD kernel implements an iterative irregular-mesh partial differential equation (PDE) solver abstracted from computational fluid dynamics applications [13]. 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. The parallel version of the kernel has the following structure as shown in Figure 11.

Each edge is a user-level thread. We partition the threads (or edges) into p sets (for p processors) to maximize data reuse by grouping together the threads accessing the same vertices. In addition, for each of the p sets, we reorder its threads by means of the breadth-first traversal on the set's corresponding subgraph.

6.2.2 Cholesky Factorization

Given an $n \times n$ symmetric positive definite matrix A, Cholesky factorization computes $A = LL^T$ and L is an $n \times n$ lower

```
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
```

Figure 11: Parallel version of the CFD kernel.

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:

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

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

1) level-3 BLAS `POTRF` to solve $L_{1:b,1:b}$,

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

2) level-3 BLAS `TRSM` to solve a linear equation system to get $L_{b+1:n,1:b}$,

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

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

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

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:

$$A'_{b+1,n:b+1:n} = L_{b+1:n,b+1:n} L^T_{b+1:n,b+1:n}.$$

The code is shown in Figure 12. 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` $= n/b$.

```
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
```

Figure 12: Parallel Cholesky factorization.

In order to use the greedy multilevel algorithm described in Section 4, we need to know what its task graph looks like. Each task in the DAG corresponds to a Level-3 BLAS operation. Figure 13 shows an example of the DAG for a 4 block by 4 block matrix and its level division. The figure displays only one iteration of the outer loop. The other iterations have a similar structure and are not shown here.

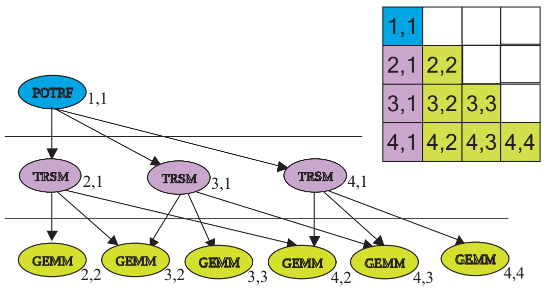


Figure 13: DAG for Cholesky factorization (one iteration).

6.3 Application Measurements

We conducted all the experiments on two platforms, respectively. 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 one is an SGI Altix 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.

6.3.1 CFD Kernel

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 number of edges than the number of vertices. Figure 14 shows that using the optimized thread schedule reduces the execution time by 25% to 35% on the Intel Clovertown SMP machine.

On the SGI DSM machine, the program always takes as input a mesh of $40,000$ vertices and $400,000$ edges. For various number of processors (i.e., 4, 8, 16, and 32), our method reduces the execution time by 32% to 42% depicted in Figure 15.

6.3.2 Cholesky Factorization

Unlike the CFD kernel program with independent threads, Cholesky factorization owns threads with data dependencies. The greedy multilevel thread scheduling method is

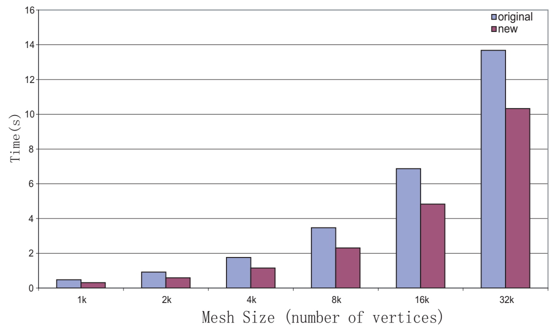


Figure 14: Performance of the CFD kernel on Intel Clovertown with various meshes.

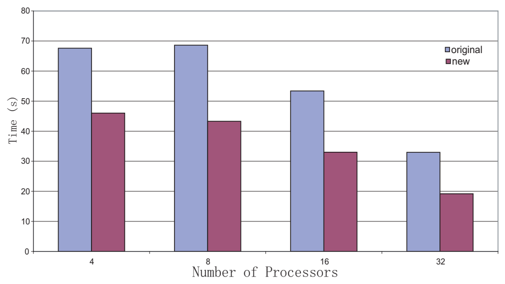


Figure 15: Performance of the CFD kernel on SGI Altix with an input mesh of size <40000, 400000>.

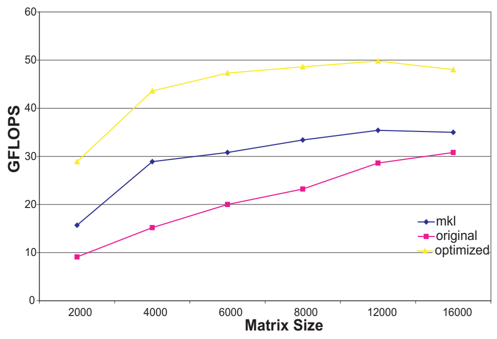


Figure 16: Performance of Cholesky factorization on Intel Clovertown given various matrix sizes.

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 greatly.

For comparison, Figure 16 also lists the performance of Intel MKL 9.1 library. 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 a better performance than the original one. On the SGI machine, we conducted experiments using different number of processors (4, 8, and 16) and compared them with Intel MKL 7.2. Each experiment takes as input matrices with different sizes. Figure 17 demonstrates that the new program is faster than the original program by 30% to 4 times.

7. RELATED WORK

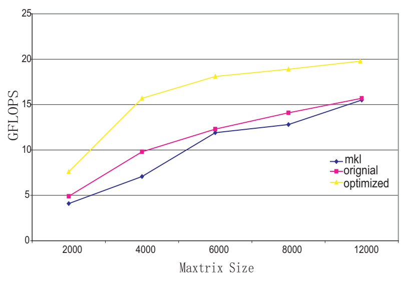
Over the past decade, a lot of research work has proposed ways of reorganizing data structures and altering programs to improve the memory access efficiency. Philbin et al. [10] describe a user-level thread library to improve cache locality using fine-grained threads. When a thread is created, a hint of the starting addresses of the accessed arrays is provided. Yan et al. [18] also developed a runtime system to maximize data reuse. Yan's approach is more generic and can be applied to parallel programs on SMP machines. Pingali et al. [13] create locality groups to restructure computations for a variety of applications but require hand-coded optimizations. In contrast, we investigate 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 applied a hierarchical partitioning technique similar to ours to solve the MPI process mapping problem [17]. He implemented a framework to compute an optimal MPI process placement to minimize the message passing cost. Pichel et al. [11, 12] formulate sparse matrix-vector product as a graph problem where each row of the sparse matrix represents a vertex. It works effectively on both SMP and ccNUMA DSM systems, but is not so generic as our affinity graph model and limited to the SpMV application. The widely used affinity loop scheduling method minimizes cache miss rate by allocating loop iterations to the processor whose cache already contains the necessary data [8]. Unlike the method, we attempt to reorder the inner loop iterations before assigning them to processors. Furthermore, we use threads as the scheduling unit (instead of loop iterations) and is not restricted to the particular two-level loop nest.

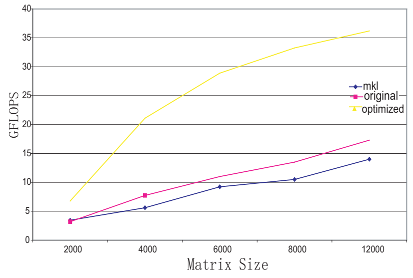
Marathe et al. investigated how to place pages on a ccNUMA DSM system using a hardware profile-guided method [7]. They run a truncated version of a user application to decide a good page placement through a hardware monitor. Differently, we use a binary instrumentation tool to analyze the memory trace to determine how to place threads and we do not rely on hardware facilities.

8. CONCLUSIONS

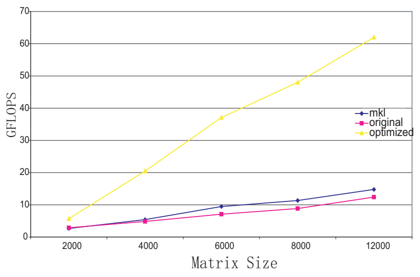
With more cores on a single chip and deeper levels in memory hierarchies, it is more difficult to run shared-memory multi-threaded programs efficiently. We present an analytical model to evaluate the performance of a thread schedule. The model has three components: affinity graph model to describe the affinity relationship between threads, memory hierarchy model to characterize the underlying shared-memory architecture, and cost model to estimate the cost for a certain thread schedule. We also propose a hierarchical graph partitioning algorithm to find an approximate



(1) 2 Nodes, 4 CPUs



(2) 4 Nodes, 8 CPUs



(3) 8 Nodes, 16 CPUs

Figure 17: Performance of Cholesky factorization on SGI Altix with 4, 8, 16 processors, respectively. We compare the performance of the original multi-threaded program to that of the program using the optimized thread schedule. The performance of Intel MKL 7.2 library is also displayed here.

solution. The experimental results show that the analytical model can accurately estimate the cost of a thread schedule for two synthetic programs and a real-world application. Based on the model and the hierarchial partitioning algorithm, we developed a tool to determine an optimized thread schedule. We applied the tool to two applications. The experiments on both SMP and DSM machines show that using the optimized schedule is able to improve the CFD kernel and Cholesky factorization greatly.

9. REFERENCES

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

[2] T. Davis. University of Florida sparse matrix collection. In *http://www.cise.ufl.edu/research/sparse*, 1997.

[3] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[4] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO*, pages 81–92. ACM/IEEE, 2003.

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

[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*, pages 64–75. IEEE Computer Society, 2004.

[7] J. Marathe and F. Mueller. Hardware profile-guided automatic page placement for ccNUMA systems. In J. Torrellas and S. Chatterjee, editors, *PPOPP*, pages 90–99. ACM, 2006.

[8] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 5(4):379–400, 1994.

[9] C. McNairy and R. Bhatia. Montecito: A dual-core, dual-thread itanium processor. *IEEE Micro*, 25(2):10–20, 2005.

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

[11] 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*, pages 66–71. IEEE Computer Society, 2004.

[12] 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 *International Symposium on Parallel Architectures,Algorithms and Networks, 2005 (ISPAN 2005).*, 2005.

[13] 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*, 31(4):305–338, 2003.

[14] H. D. Simon and S.-H. Teng. How good is recursive bisection? *SIAM J. Sci. Comput.*, 18(5):1436–1445, 1997.

[15] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, and J. Joyner. Power5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, 2005.

[16] 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*, pages 97–106, 2007.

[17] J. L. Träff. Implementing the mpi process topology mechanism. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[18] Y. Yan, X. Zhang, and Z. Zhang. Cacheminer: A runtime approach to exploit cache locality on SMP. *IEEE Trans. Parallel Distrib. Syst.*, 11(4):357–374, 2000.