Feedback-Directed Thread Scheduling with Memory Considerations

Fengguang Song, Shirley Moore, and Jack Dongarra
Dept. of Computer Science, University of Tennessee
Knoxville, Tennessee, USA
song@cs.utk.edu, shirley@cs.utk.edu, dongarra@cs.utk.edu

ABSTRACT

This paper describes a novel approach to generate an optimized schedule to run threads on distributed shared memory (DSM) systems. The approach relies upon a binary instrumentation tool to automatically acquire the memory sharing relationship between user-level threads by analyzing their memory trace. We introduce the concept of Affinity Graph to model the relationship. Expensive I/O for large trace files is completely eliminated by using an online graph creation scheme. We apply the technique of hierarchical graph partitioning and thread reordering to the affinity graph to determine an optimal thread schedule. We have performed experiments on an SGI Altix system. The experimental results show that our approach is able to reduce the total execution time by 10% to 38% for a variety of applications through the maximization of the data reuse within a single processor, minimization of the data sharing between processors, and a good load balance.

Categories and Subject Descriptors

D.3.4 [**Software**]: Processors—*run-time environments, optimization*

General Terms

Performance, experimentation

Keywords

Distributed shared memory, shared-memory programming, affinity graph, scientific applications

1. INTRODUCTION

High performance computing platforms that support a shared-memory paradigm attain the benefits of large-scale parallel computing without surrendering much programmability [7]. On distributed shared memory (DSM) systems, a program can be written as if it were running on a symmetric multiprocessor (SMP) machine. A typical subclass of the DSM systems builds on the cache-coherent non-uniform memory architecture (ccNUMA). A contemporary ccNUMA system such as the SGI Altix consists of a large number of nodes, each of which has a couple of processors and a fixed amount of memory. With the emergence of chip multiprocessors (CMP), a compute node could have a number of multi-core chips each of which has many cores. Figure 1 depicts the hierarchy of such a system. Our previous research has shown that it is non-trivial to schedule threads on a single multi-core chip with a shared L2 cache due to resource contention and the nature of memory sharing between the threads [17].

In this paper, we focus on thread scheduling for highly scalable shared memory machines that have a multi-level memory hierarchy. The placement of threads onto the memory hierarchy often has impact on program performance if they have overlapping memory footprints. Compute-intensive scientific applications usually consist of a set of threads to conduct identical computation on either overlapping or disjoint subsets of the global data. When two threads are accessing the same data, the location to launch them will affect the program performance greatly. For instance, the worst situation would be to place them on different nodes which induces a great number of remote memory accesses, and the best one would be to place them on the same multi-core chip because the shared L2 cache can potentially eliminate the redundant loading of the same data [9, 17]. The intermediate situation is either placing them on the same node but different chips, or placing them on the same module but different chips (Power4 and Power5 machines have a module level [16]).

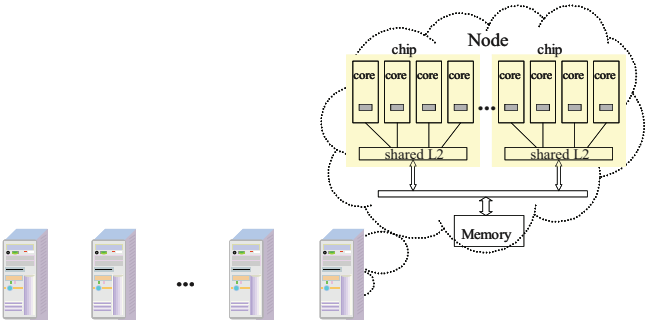

Figure 1: Memory hierarchy on a ccNUMA distributed shared memory system.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HPDC'07, June 25–29, 2007, Monterey, California, USA.
Copyright 2007 ACM 978-1-59593-673-8/07/0006 ...$5.00.

Our goal of thread scheduling is to improve the effectiveness of the memory hierarchy by identifying user-level threads and reorganizing them to enhance the program temporal and spatial locality, as well as placing tightly-coupled threads as close as possible. The user-level thread may be considered as a "logical task" whose granularity varies from as tiny as a single instruction to as big as an actual kernel-level thread. With the help of data dependence analysis, we can even reorder tiny instructions to maximize data reuse across the entire data set. However for simplicity and quick analysis, we identify fine-grained user-level threads as conceptual units of computation from the viewpoint of a programmer. Most of the time it is straightforward to identify the threads. For instance, an iteration of a loop nest or an update of an object may be created as a thread.

In order to improve the data locality, we must decide which user-level threads will be in the same group and in what order to execute them, as well as how to map the groups to different processors. Figure 2 illustrates the overall structure of our approach to realizing it. The approach relies upon a binary instrumentation tool to (i) obtain and analyze the memory trace of each thread and find out the nature of memory sharing between threads in an "affinity graph". An affinity graph is an undirected weighted graph where each vertex represents a thread and the weight for edge e_{ij} denotes the total number of distinct addresses accessed in common by threads i and j. To make the memory tracing method more practical, the instrumentation tool generates affinity graphs dynamically without storing the huge memory trace to disk. After the instrumented executable finishes, an affinity graph is built and written to a file. Next, (ii) we partition the graph into a number of subgraphs (in our experiments, the number is equal to the number of processors). Based on the partitions, (iii) we compute a "good" schedule to put threads on different processors correspondingly. The schedule is written in a file which will be later used as feedback to future executions. Finally, (iv) a user reruns the program taking as input the feedback file.

We have experimented with the feedback-directed thread scheduling method using several application programs, including sparse matrix-vector multiplication (stored in compressed row storage or compressed column storage format), sparse matrix-matrix multiplication, parallel radix sorting, and a kernel from computational fluid dynamics codes. Our results show that the feedback-directed method can significantly reduce the execution time. In particular, it can successfully improve the performance of programs with dynamic memory access patterns and little compile-time information.

Our work makes the following contributions:

- We present a feedback-directed approach for thread scheduling for general-purpose programs. Since it is an offline method, compute-intensive optimizations are allowed to determine an optimal schedule. Also the overhead to execute (or follow) the predetermined schedule is less than that of dynamic scheduling methods.

- By identifying independent user-level threads, we are able to parallelize the original program while maximizing the program locality.

- We develop techniques for instrumenting the executable and analyzing the memory trace without triggering any disk I/O. These techniques are more practical than traditional trace-based methods.

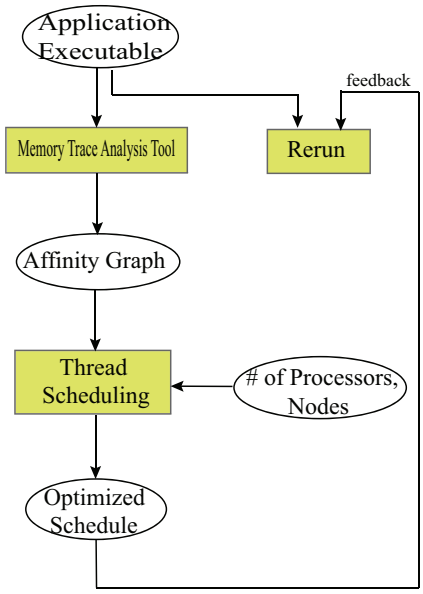

Figure 2: Overall structure of the thread scheduling method with memory considerations.

- We propose the concept of affinity graph to represent the nature of memory sharing between threads. We have applied it successfully to various applications.

- Our graph partitioning technique has a hierarchical structure that corresponds to the actual architecture of a real machine. For instance we can first partition threads to a set of nodes, then to a set of chips, and finally to a set of processor cores (we assume one kernel thread per processor core).

2. FEEDBACK-DIRECTED METHOD

A feedback-directed method strives to improve performance by using profiling information to exploit opportunities for optimizations. Our work is concerned with how to maximize data locality on each processor and minimize the data sharing between processors (each processor runs one kernel thread). Since we determine a thread schedule based on the referenced addresses, collecting a memory trace is typically necessary. There are four types of approaches to obtaining the memory trace: compiler-based, run-time system, online feedback-directed optimization (FDO), and offline feedback-directed optimization. We choose to use the offline FDO method due to the following reasons:

- The feedback-directed method can attain dynamic information about memory references, particularly for programs with irregular access patterns. The dynamic information cannot be obtained at compile time.

- The offline method has much less overhead than online methods since the optimal schedule is determined during the very beginning profiling program run.

- Our method collects a more accurate and detailed memory trace for each thread to determine an optimal schedule. Current run-time systems usually use a couple of heuristics to schedule threads.

We now present the overall structure of our feedback-directed method. We first need to collect the memory trace of applications by using a memory trace analysis tool. The analysis tool supports binary instrumentation and is built upon Pin [8].

2.1 Memory Trace Analysis

Pin follows the model of ATOM and allows tool writers to analyze applications at the instruction level [8]. It uses a dynamic just-in-time (JIT) compiler to instrument binary codes while they are running. The set of Pin APIs provide support for observing a process's architectural states (e.g., register contents and memory references).

We wrote a memory trace analysis tool in C++ and used the Pin API to implement two types of routines: *instrumentation routine* and *analysis routine*. The instrumentation routine tells Pin to insert instrumentation to every instruction that reads or writes data. The virtual address of the referenced datum is passed as an argument to the analysis routine. In order to differentiate addresses from distinct threads, the `thread ID` is passed as another argument to the analysis routine. Instrumenting the binary executable also enables us to keep the original memory access pattern while reflecting various compiler optimizations.

Two analysis routines `RecordMemRead` and `RecordMemWrite` are implemented for read and write operations, respectively. The analysis routine works as an event handler. For each memory reference, the routine identifies the thread ID for that reference and stores it into a buffer. Each thread owns a buffer for keeping distinct addresses (i.e., two references to the same address will be stored once). We tried writing the memory trace to a file, but the size of the trace file and the I/O cost increased so fast that it soon became impractical to use.

The space requirement of our memory trace analysis tool is equal to the sum of the distinct addresses referenced by each thread. It is possible for us to impose a limit on the number of distinct addresses for each thread so that all the tracing operations can be performed in memory (i.e., without disk). This diskless tracing method is much less expensive and more practical than writing the memory trace to disk.

2.2 Techniques to Process Large Graphs

While a modern computer system has no problem keeping the distinct addresses in memory for most applications, the size of the generated affinity graph can explode very quickly. For instance, 10^5 fine-grained threads (10^5 vertices) may require 40GB memory if the graph is totally connected (10^{10} edges). The size of the graph is limited by the capacity of the memory. Given a fixed amount of memory, it is not trivial to build an appropriately sized affinity graph without exceeding the available memory.

We adopt several techniques to improve the time complexity and space complexity of the process of graph creation. It rarely happens in practice that every thread has a memory sharing relationship to all other threads if we don't consider the very small number of global variables. Thus affinity graphs are often sparse and symmetric. We represent affinity graphs by an adjacency list and only store edges e_{ij} where i < j to reduce the memory requirement. This is analogous to storing an upper-triangular adjacency matrix.

A simple algorithm for building a graph from the recorded memory trace is shown in Figure 3. The memory trace analysis tool instrumented the executable and stored addresses in `thrd_addrs` for each thread. Next, the algorithm compares the memory trace of every two threads i and j where i < j. `create_edge()` creates an edge between thread i and thread j and assigns a proper weight to it.

```
1 map<tid, addr> thrd_addrs;
2 for i = 1, num_thrds-1
3   for j = i+1, num_thrds
      //Compare traces of threads i, j.
4     create_edge(thrd_addrs, i, j);
5   end for
6 end for
```

Figure 3: A simple algorithm to build affinity graphs.

Suppose there are T threads and each thread accesses N addresses. Even though the function `create_edge()` has a linear time complexity $O(N)$ (by merging two ordered lists), the overall time complexity is equal to $O(T^2N)$. In practice N is often bounded but T could be very large (e.g., 10^6 to 10^8). Furthermore, no matter how sparse a graph is, this algorithm always takes time $O(T^2N)$ to build the graph, which could lead to many hours of computation.

To be more efficient in processing large graphs, we change to a different data structure and develop a new algorithm. The new algorithm employs an adjustable parameter of $DenseRatio \in [0 \ldots 1]$ to control the density of the graph. If an address is accessed by all threads, the graph is fully connected. However this address will not help graph partitioning in the next step. Therefore we adjust *DenseRatio* to eliminate those edges that can form a clique of size greater than $NumberThreads \times DenseRatio$. $DenseRatio = 0.0$ will yield a graph with no edges and $DenseRatio = 1.0$ will yield a graph with all possible edges.

Figure 4 lists the pseudo-code for the more efficient algorithm. The new data structure `addr_thrds` stores `addr` as keys instead of prior `t_id` as keys. The algorithm takes as input the memory trace stored in `addr_thrds` and builds an affinity graph. For each address, we determine the number of threads that have accessed it. If the number is greater than *DenseRatio* times the total number of threads, we skip the analysis for this address and the creation of the relevant edges. Otherwise, we create edges among the set of threads.

Again, let N be the total number of addresses referenced by the application and T be the total number of threads. The time complexity of the algorithm varies between $O(N)$ and $O(NT^2)$ depending on how sparse the graph is. The worst-case time complexity occurs when the graph is fully connected and $DenseRatio = 1.0$.

Certainly we can use *DenseRatio* to reduce the graph density. Note that the previous algorithm in Figure 3 always has a time complexity of $O(NT^2)$. The new algorithm not only has a better average-case time complexity but also adopts an adjustable parameter *DenseRatio* to eliminate particular edges. In our experiments, we use $DenseRatio = 0.9$ to reduce edges and are able to create sparse graphs. In addition, most of the eliminated edges are due to a couple of global

```
1 T = total number of threads;
2 map<addr, tid> addr_thrds;
3 for each addr in addr_thrds do
4   thrd_set = threads accessing addr;
5   m = size of thrd_set;
6   if (m > DenseRatio*T) continue;
7   create edges between any pair
    of threads within thrd_set;
8 end for
```

Figure 4: A more efficient algorithm to build affinity graphs.

variables. Note that a single global variable can lead to a fully connected graph.

The generated affinity graphs can be written either in Graphviz DOT format for the purpose of displaying, or in Chaco/METIS format for the next stage of graph partitioning.

3. AFFINITY GRAPH MODEL

We use a weighted undirected graph $G = \langle V, E \rangle$ called "affinity graph" to model the nature of memory sharing between threads. In the graph, each vertex $v \in V$ denotes a thread and each edge $e_{ij} \in E$ indicates that there exists a sharing relationship between threads v_i and v_j. The weight w_{ij} associated with edge e_{ij} denotes that thread i and thread j have accesses to a number w_{ij} of virtual addresses in common. Note that w_{ij} does not measure the frequency of references to the addresses. We assume that the greater the weight w_{ij}, the higher probability it indicates of improved data locality if we place threads i and j on the same processor. We also assume all threads are independent.

There are two types of threads: user-level threads and kernel threads. The user-level thread may be either as small as a single instruction or as large as a parallel task. When the thread has a single instruction, users can derive an optimal schedule satisfying the data dependence constraint to minimize the cache miss rate. For medium to large applications, we regard a certain number of loop iterations as a user-level thread. On the other hand, if a vertex denotes a kernel thread, we can use the corresponding graph to determine how to map the kernel threads onto different processors.

Figure 5 shows an example of multiplying 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 use four threads T0-T4 to compute matrix multiplication. Correspondingly, T0-T4 compute the result for sub-matrices C11, C12, C21 and C22 in parallel. After instrumentation and execution of the program, an affinity graph is created as shown on the right side in Figure 5. This example demonstrates what an affinity graph is and its ability to reveal the memory sharing relationship between threads.

We conducted a few experiments on the SGI Altix 3700 machine which is composed of dual-processor nodes. From the affinity graph generated for the four threads shown in Figure 5, we conclude that threads 0 and 2 should run on the same node while threads 1 and 3 should run on another node. Otherwise the program will incur a lot of remote memory accesses. We compare the performance of putting threads 0 and 2 together to that of putting 0 and 1 together (an intuitive way). Figure 6 shows the wallclock execution

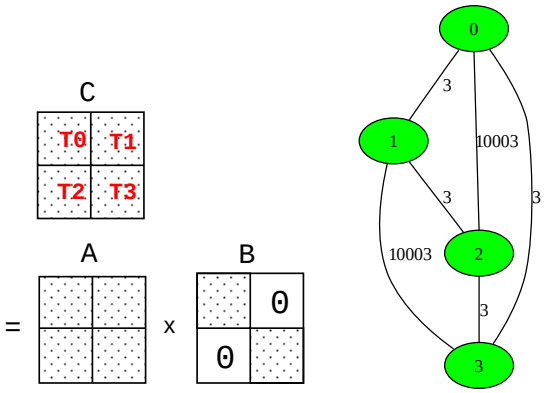

Figure 5: Matrix multiplication using four threads and the corresponding affinity graph. Each thread of T0-T3 computes for one block of matrix C. Matrix B is a sparse matrix.

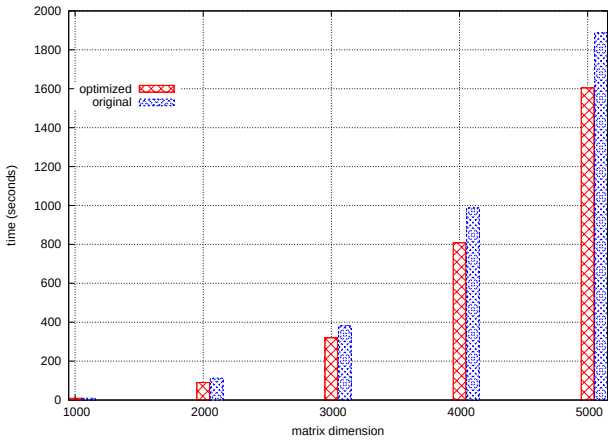

Figure 6: Comparison of two different thread schedules. The revealed affinity relationship helps improve the multi-threaded program performance.

time of these two placements. The optimized placement is better than the original one by 20%.

3.1 Affinity Graph Partitioning

For most of the programs, it is not obvious to decide how to map threads onto different processors. In this section we give a formal definition of affinity graph and describe our approach to using graph partitioning to derive thread groups. Figure 5 shows an example of affinity graph.

Definition 1. An 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 independent of $t_j, i \neq j\}$,
- $E = \{(t_i, t_j) \mid i \neq j, \exists$ address x such that both t_i and t_j access x $\}$,
- $w_t : T \longrightarrow Z^+$,
- $w_e : E \longrightarrow Z^+$, and $w_e(t_i, t_j) = 0$ if $(t_i, t_j) \notin E$.

The purpose of graph partitioning is to identify tightly-coupled threads and divide a graph into subgraphs as disjoint as possible. We also hope to group together threads that have large overlapping footprints. The partitioning process is hierarchical since it corresponds to the actual memory hierarchy on a real computer. For instance, on a ccNUMA DSM system, we first divide the threads into a number of groups which is equal to the number of nodes. Next we divide each group further into subgroups that correspond to the processors in a node.

Definition 2. Given an affinity graph G and a partition $P = \{T_1, \ldots, T_n\}$,

$$Sharing(T_i, T_j) = \begin{cases} \sum_{u \in T_i} \sum_{v \in T_j} w_e(u,v) & : \quad i \neq j \\ 0 & : \quad i = j \end{cases}$$

The optimization criterion aims to minimize the sharing between groups. Assume there are n compute nodes each of which has p processors. (a) We divide the threads in graph G into n parties $N_1, N_2, \ldots, N_n$ by minimizing

$$\sum_{1 \leq i,j \leq n} Sharing(N_i, N_j).$$

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

$$\sum_{1 \leq i,j \leq p} Sharing(P_i, P_j).$$

We use the `Chaco` software package to partition affinity graphs. Chaco provides several methods for finding small edge separators: inertial, spectral, Kernighan-Lin and multilevel methods [4]. In our experiments, we use the multilevel Kernighan-Lin method [5] with recursive bipartitions to determine optimal subgraphs.

4. APPLICATIONS

We performed experiments with a variety of applications to evaluate our feedback-directed method of thread scheduling. These applications cover a range of domains and have been widely used by users.

4.1 Sparse Matrix-Vector Multiplication

Sparse matrix-vector multiplication (SpMV) is an important subroutine in many iterative methods. We use two formats of Compressed Column Storage (CCS) and Compressed Row Storage (CRS) to implement iterative methods and try to improve their performance. The program using the CRS format is presented in Figure 7. The inner `for` loop is distributed to a number of processors and executed in parallel. Arrays `val, col, row` store the sparse matrix `A` while `x, y` store the column vectors for computing $y = Ax$. The code using the CCS format is similar to that in Figure 7 and not shown here.

For both formats of CRS and CCS, we try to use the technique of affinity graph partitioning to improve the program performance. Suppose matrix A is stored in CCS format, the vector $y = Ax$ is computed as follows:

$$y = \sum_{j=1}^{n} a_j x_j,$$

```
1 for iter = 1, NUM_ITER
2   #pragma omp parallel for
3   for i = 1, num_rows
4     y = 0;
5     for j = row[i], row[i+1]-1
6       y += val[j]*x[col[j]];
7     end for
8     y[i] = y;
9   end for
10end for
```

Figure 7: Parallel iterative method calling SpMV $y = Ax$. Sparse matrix A is stored in CRS format by arrays `val, col, row`.

where a_j is the jth column of A. For the parallel version with p threads, each thread computes a partial sum and updates the vector y. But an update of element y_i may invalidate a set of neighboring y_i' elements in other processors due to false sharing. By grouping the columns that access the same elements in vector y, we are able to improve the temporal locality and reduce the chances of false sharing on vector y.

As for sparse matrix A stored in CRS format, each thread computes a subset of vector y, where

$$y_i = \sum_{j=1}^{n} a_{ij} x_j.$$

Note that the only chance of data reuse lies in vector x. If two rows read the same set of elements $x_{s1}, x_{s2}, \ldots, x_{s_d}$ in vector x, running them continuously will reuse the d elements and improve the temporal locality.

4.2 Sparse Matrix-Matrix Multiplication

To compute sparse matrix-matrix multiplication (SpMM), we store matrix A row by row in CRS format and store matrix B column by column in CCS format. The parallel SpMM program is described in Figure 8. Sparse matrices are distributed by rows across the p processors. Each processor computes for a number of $\frac{N}{p}$ consecutive rows in matrix C.

The outer two loops `i` and `j` in Figure 8 are data independent and can be executed in parallel. A user-level thread may either compute a dot product of the ith row of matrix A and the jth column of matrix B (i.e., the outer two loops are parallelized), or multiply the ith row of A and the whole matrix of B (i.e., only the outermost loop `i` is parallelized). The former definition is more fine-grained and can identify a greater number of threads to reorder to maximize the program locality. But it is too costly to compute since the number of threads N^2 may result in a very large graph. Instead, we use the latter coarse-grained definition to do our experiments. The experimental results demonstrate that we can still speedup the program by 10-20% even using the coarse-grained threads.

4.3 Parallel Radix Sorting

Sorting is also an important kernel for high-performance multiprocessing. Parallel radix sorting has been shown to be a simple and efficient parallel method that outperforms other parallel sorting algorithms [6, 15]. It is one of the important kernels in the NAS Parallel Benchmark and SPLASH-2 benchmark suites.

The radix sorting algorithm performs one round of sorting for every r bits of the keys. For a set of 32-bit integer keys

```
1 struct CRS A;
2 struct CCS B;
3 double    *C;
4 #pragma omp parallel for
5 for i = 1, N
6   for j = 1, N
7     c = C[i*N+j];
8     for idx_a = A.row[i],A.row[i+1]-1
9     for idx_b = B.col[j],B.col[j+1]-1
10      if(A.col[idx_a]==B.row[idx_b])
11        c+=A.val[idx_a]*B.val[idx_b];
12    end for
13    end for
14    C[i*N+j] = c;
15  end for
16end for
```

Figure 8: Parallel version of SpMM.

and r=8, four rounds of iterations are needed. Suppose N keys are stored in an array and distributed to p threads t_0, t_1, ..., t_{p-1}, and that thread t_i has to perform a local radix sort on the segment of $[\frac{N}{p} \times t_i, \frac{N}{p} \times t_{i+1})$ keys. During each round, a thread builds a local histogram of the occurrences of its local keys. Next it computes a global histogram by combining other threads' histograms. Finally, each thread writes its keys back to corresponding positions in the array. Likewise, the next round of sorting starts for another r bits of keys.

Regarding the memory access pattern, each thread first reads keys from its assigned segment, sorts them, and then writes keys to remote positions in other segments. Therefore if we place two threads onto the same node that write many keys to each other, the remote memory accesses will become local memory accesses, leading to better performance.

In our experiments, we schedule threads based on the first round of r bits. We apply the optimization to the partitioned parallel radix sorting algorithm which sorts keys in a left-to-right fashion instead of the traditional right-to-left [6]. The algorithm uses the most significant r bits at the beginning. After the first round, the rest of sorting is always confined locally within each thread. Hence the number of remote memory accesses is greatly reduced during the whole process of radix sorting.

4.4 IRREG

The IRREG kernel is an iterative irregular-mesh partial differential equation (PDE) solver abstracted from computational fluid dynamics (CFD) applications [14]. The irregular meshes are used to model physical structures and consist of nodes and edges. The computational kernel iterates over the edges of the mesh, computing the forces between both end points of each edge. It then modifies the values of all nodes. Figure 9 shows the parallel version of IRREG.

Each edge is a user-level thread in our experiment. We partition the threads (or edges) into p sets (for p processors) to maximize data reuse by grouping together the threads accessing the same nodes. In addition, for each group, we reorder the threads by means of the breadth-first traversal of the group's corresponding subgraph. Running the optimized schedule reduces the execution time by around 35%.

5. EXPERIMENTAL RESULTS

All of the experiments are conducted on an SGI Altix

```
1 for iter = 1, NUM_ITER
2   #pragma omp parallel for
3   for i = 1, edges
4     n1 = left[i];
5     n2 = right[i];
6     force = f(x[n1],x[n2]);
7     y[n1] += force;
8     y[n2] -= force;
9   end for
10end for
```

Figure 9: Parallel version of IRREG.

3700 system with 256 nodes, each of which has two 1.6 GHz Itanium processors. The system has a ccNUMA Distributed Shared Memory (DSM) architecture and the memory is physically distributed across nodes. Each processor can access any memory location through the SGI NUMAlink 4 interconnect. Memory access time depends on the distance between the processors and the nodes where the physical memory is located. Our approach is a generic approach that works for various types of applications, therefore we compare our programs using the feedback-directed method to the programs built by compiler optimizations.

5.1 Implementation Issues

All the parallel applications are implemented in C using Pthreads. We obtain the optimized schedule automatically through our memory trace analysis tool described in Section 2. Since we can only set thread and memory affinity on the login node of the system, to verify our prototype, we run a small number of four kernel threads on two nodes with four processors to conduct the experiments. In all of the experiments, the fine-grained user level threads are totally independent and can be executed in parallel.

In order to execute the program with the new schedule, we need to make a minor change to the original program manually. This step could be done easily by extending a compiler. For instance, instead of running `for i = 0 to n`, the compiler can wrap the fundamental computation unit by `for i = mytasks[0] to mytasks[n]`, where `mytasks` is specified in the optimal schedule.

After finding the affinity relationship between threads, we use the *cpuset* library provided by SGI Linux [18] to bind threads to different processors. Note that the binding of threads to processors happens only once when the threads are created. The cpuset APIs `cpuset_pin()` and `cpuset_membind()` allow our C programs to place both processor and physical memory within a cpuset.

5.2 SpMV

Sparse matrix-vector multiplication is called by a synchronous iterative method. For each sparse matrix, we run a fixed number of iterations. There are two programs written for SpMV. One program uses the CCS storage format and the other one uses the CRS format. Users often choose one of the two formats to implement their programs. Table 1 lists the sparse matrices used in our SpMV experiments. They were downloaded from the UF Sparse Matrix Collection [1]. A matrix was selected if the amount of computation for each row is approximately equal.

Figure 10 shows the effect of the optimization. Values less than 1 indicate performance speedup. Our scheduling

Table 1: Sparse matrices used in the SpMV experiment.

	Name	Dimension	NNZ
1	msc01440	1,440	44,998
2	circuit_1	2,624	35,823
3	bp_1000	822	4,661
4	coater1	1,348	19,457
5	msc23052	23,052	1,142,686
6	mark3jac040	18,289	106,803
7	utm3060	3,060	42,211

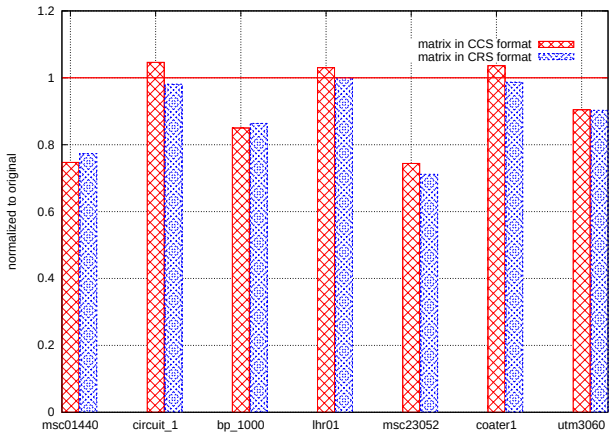

Figure 10: Performance of SpMV. Both formats of CCS and CRS are used for each sparse matrix.

method reduces execution times by 10% to 25% for four out of seven matrices in both CCS and CRS formats. The improvement for the CRS program comes from maximized data reuse, and the improvement for the CCS program comes from reduced false sharing. Only three matrices (circuit_1, lhr01, and coater1 in CCS format) show a little slowdown (less than than 5%). There are three possible reasons: 1) each row has too few elements so that the scheduling overhead exceeds the gain of the improved data reuse; 2) the amount of computation for each processor becomes imbalanced; 3) there is no opportunity to enhance the data reuse for certain sparse matrices (i.e., the original order is near-optimal).

5.3 SpMM

We conducted sparse matrix-matrix multiplication $C = AB$ with four test inputs which are shown in Table 2. Instead of attempting various combinations of pairing sparse matrices, we simply let B equal the transpose of A. In the experiment, matrix A is stored row by row in the CRS format and matrix B is stored by columns in the CCS format. We let a user-level thread compute the product of the ith row of A and the whole matrix of B. Figure 11 indicates that the reduction of the total execution time is around 20%. Unlike the above SpMV experiment, this time the thread scheduling method is effective for all the test inputs.

Table 2: Sparse matrices used in the SpMM experiment.

	Name	Dimension	NNZ
1	msc01440	1,440	44,998
2	msc01050	1,050	26,198
3	utm3060	3,060	42,211
4	bcsstk13	2,003	83,883

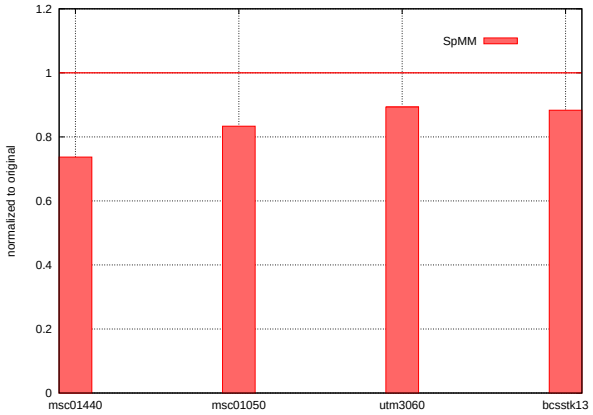

Figure 11: Performance of SpMM.

5.4 Parallel Radix Sorting

The input keys array is distributed across four processors (i.e., four kernel threads) to do the parallel radix sorting. Suppose we create the four threads $t_0, \ldots, t_3$ on processors $p_0, \ldots, p_3$. Thread t_i keeps the corresponding ith block of the array. Processors p_0 and p_1 are co-located in one compute node while p_2 and p_3 are in another one.

We use a synthetic input to compare the effect of different placements of threads on processors. Taking as input the synthetic input, t_0 and t_3 have to swap keys while t_1 and t_2 have to swap as well. A straightforward placement would be placing thread t_i on processor p_i correspondingly (we call this experiment "original"). In this experiment each swapping involves a pair of remote memory writes. A better way would be to put p_0 and p_3 on the same node so that data swapping only involves local memory accesses (we call this experiment "optimized").

To find out what the best performance could be, we modified the program and removed those operations performing remote memory write (we call it "original without memory write"). The modified program provides further insight into the lower bound of the execution time, which is the best we could achieve. Figure 12 depicts the wallclock execution time of the three programs with different input sizes. Although the execution time is improved by just 10%, the reduction in remote memory accesses is equal to 30% - 50% of the total memory access time (i.e., 30% - 50% of the total communication time).

5.5 IRREG

The IRREG program takes an irregular mesh as the input and iterates through all the edges and updates the end

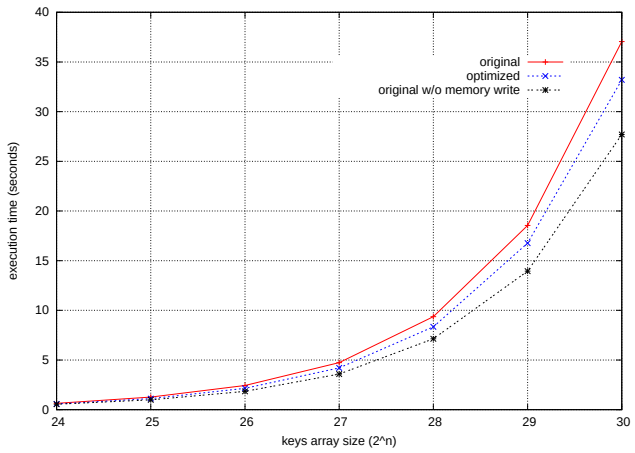

Figure 12: Performance of one-round parallel radix sorting.

points of each edge. Suppose a mesh has N nodes and E edges. We use a random generator to assign nodes to the end points of every edge which are uniformly distributed between 0 and N-1. We analyze the memory trace of the program to determine an optimal schedule. The new schedule defines four sets of edges (for four processors) and each set has an ordered list of edges. We compare the performance of the original program to the optimized program with the new schedule. As shown in Table 3, our feedback-directed optimization method reduces the program execution time by 33% to 38%. The reason for the increased performance improvement with the larger mesh size is because a larger working set often results in a greater number of cache misses which is later reduced by using an optimized schedule.

Table 3: Performance of IRREG.

Irregular Meshes		Time	Time	Reduced
Nodes	Edges	(original)	(optimized)	Time
400	4,000	1.11	0.74	33.3%
4,000	40,000	10.88	6.87	36.9%
40,000	400,000	102.88	63.57	38.2%

6. RELATED WORK

A lot of previous research work has proposed ways of reorganizing data structures and altering programs to maximize the data reuse. Philbin et al. [11] describe a user-level thread library to improve cache locality using fine-grained threads. All data-independent units of computation implied in the sequential program are created as fine-grained threads all at once. When a thread is created, a hint of the starting addresses of the accessed arrays must be provided as an argument. After thread creation, the thread library reschedules the threads at runtime to reduce the number of L2 cache misses. This method only works well for sequential programs.

Yan et al. [19] developed a runtime library to maximize data reuse. Yan's approach is more generic and can be applied to parallel programs on SMP machines. In addition, they adopt an adaptive scheduler to achieve load balance between processors. Like Philbin's approach, they also use the starting addresses of arrays as hints to determine an optimal schedule. Pingali et al. [14] use locality groups to restructure computations for a variety of applications but require hand-coded optimizations. In contrast, we use a binary instrumentation tool to analyze memory trace offline and automatically acquire more precise relationship information between threads. Such precise information is critical for thread scheduling on a ccNUMA distributed shared memory system. Our approach also has less overhead than the runtime approaches by employing a feedback-directed method.

Ding [3] improves the program locality through trace-driven computation regrouping. He developed a trace-driven tool to measure the exact control dependence between instructions and applied techniques of memory renaming and reallocation. Due to the expense of scheduling individual instructions, this method is limited to small fragments of a few kernels. It is also only applicable to sequential programs. Ding and Kennedy [2] introduced a compiler technique to minimize program bandwidth consumption. They propose a two-step strategy to fuse loops based on the reuse distance and regroup (intermix) several arrays based on their reference affinity.

Similar to our approach, Pichel et al. [12, 13] formulate sparse matrix-vector product as a graph problem. In the graph, each row of the sparse matrix represents a vertex. The distance between vertices are determined by their locality model. The proposed technique tries to improve the program locality by reorganizing rows of the original matrix through graph partitioning and subgraph reordering. It works effectively on ccNUMA DSM systems, but is limited to the SpMV application.

Affinity loop scheduling minimizes the cache miss rate by allocating loop iterations to the processor whose cache already contains the necessary data [10]. The affinity scheduling emphasizes loop iteration assignment and assumes that the loop iterations have an affinity to particular processors. A typical use of this method is to schedule a parallel loop that is nested within a sequential loop. Unlike affinity loop scheduling, our method tries to reorder and parallelize the inner loop iterations before assigning them to particular processors. Furthermore, we use more generic fine-grained threads as the scheduling unit rather than loop iterations.

Marathe et al. investigated how to place pages on a ccNUMA DSM system using a hardware profile-guided method [9]. They run a truncated version of a user application to decide a good page placement through a hardware monitor. Then they leverage the "first-touch" technique to allocate pages to assigned processors. Differently, we use a binary instrumentation tool to analyze the memory trace to determine the memory sharing relationship between threads and we don't rely on any hardware facilities.

7. DISCUSSION

In our model, we assume each thread has the same amount of computation. Unfortunately it is not true for many applications. If two threads have heavier workload and larger footprints than the others, they are more likely to be grouped together because the number of shared addresses might be large too. This kind of grouping may result in a load imbal-

ance problem. To solve this problem, we are extending the current graph model by assigning weights to vertices.

The graph model doesn't distinguish read and write operations. On a ccNUMA DSM system, a memory write will invalidate a number of cache lines in other processors. It is also an expensive operation. Therefore when performing graph partitioning, we hope to give a higher priority to writes than to reads. Our model represents the affinity relationship by the number of addresses accessed in common. A more accurate way would be using virtual cache-line numbers to reflect the real data movement (load/store of cache lines). It can also reduce the space complexity of our method.

We adopt *DenseRatio* to control the number of edges (or sparsity) in the graph. The current prototype cannot handle too huge a graph with hundreds of millions of threads. Such a huge graph without edges still requires a lot of memory. We could build the graph and write it to disk to solve this issue. Since we adopt a diskless approach to analyzing the memory trace, the ability to record traces is limited by the amount of memory. To reduce the memory requirement, we hope to represent contiguous memory addresses by regions instead of individual points. Other alternative solutions might be designing an online algorithm, or collecting a partial memory trace for each thread.

Furthermore, executing the complete instrumented executable could be much slower (e.g., 10 to 100 times) than the original executable due to the cost of calling the analysis subroutine, storing the memory trace in associated arrays, and creating graphs in the end. However, a partial execution of the truncated program can overcome the problem. For instance, a single outer-loop iteration of an iterative method is sufficient to build an affinity graph. Our ongoing work is implementing a mechanism to support the partial memory tracing method.

8. CONCLUSION

We present a feedback-directed framework to maximize program locality on distributed shared-memory systems. We first run a binary instrumentation tool to automatically identify the nature of memory sharing between threads which is represented by an affinity graph. The second step is to perform graph partitioning to determine an optimized schedule for assigning threads to particular processors. The optimized schedule improves the data locality of the original program and reduces TLB misses as well as the number of remote memory references on DSM systems.

Experiments on four different types of applications have shown that our method is significantly effective and can reduce the program execution time by up to 38%. Although we are using a memory tracing method, the online analysis completely eliminates expensive disk I/O operations. Our approach demonstrates that the feedback-directed method is especially good for applications with irregular computation and dynamic memory access patterns. The overhead to execute the applications using the optimal schedule (i.e., feedback) is also cheap.

With the affinity graph model, we are able to use a hierarchical mechanism to partition the graph corresponding to the actual memory hierarchy on a real system (e.g., node, chips, and processor cores). Our experiments so far involved either the node level or the processor level. After incorporating the hierarchy structure into our framework, we shall conduct more experiments on a large number of processors. It would also be possible to implement the feedback-directed strategy in commercial compilers so that the programmer can automatically achieve high-performance on leading-edge shared memory systems.

9. ACKNOWLEDGMENTS

The authors would like to thank HPDC'07 reviewers for their valuable comments and suggestions on the initial draft of the paper. This research is supported by the National Science Foundation under grant No. 0444363.

10. REFERENCES

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

[2] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *J. Parallel Distrib. Comput.*, 64(1):108–134, 2004.

[3] C. Ding and M. Orlovich. The potential of computation regrouping for improving locality. In *SC*, page 13. IEEE Computer Society, 2004.

[4] B. Hendrickson and R. Leland. The Chaco user's guide: Version 2.0. In *Sandia Tech Report SAND94-2692*, 1994.

[5] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 28, New York, NY, USA, 1995. ACM Press.

[6] S.-J. Lee, M. Jeon, D. Kim, and A. Sohn. Partitioned parallel radix sort. *J. Parallel Distrib. Comput.*, 62(4):656–668, 2002.

[7] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 37. ACM Press, 1995.

[8] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In V. Sarkar and M. W. Hall, editors, *PLDI*, pages 190–200. ACM, 2005.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] H. Shan and J. P. Singh. Parallel sorting on cache-coherent dsm multiprocessors. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, page 40, New York, NY, USA, 1999. ACM Press.

[16] 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.

[17] F. Song, S. Moore, and J. Dongarra. Modeling of L2 cache behavior for thread-parallel scientific programs on Chip Multi-Processors. In *UT Computer Science Technical Report UT-CS-06-583*, 2006.

[18] SGI. Linux resource administration guide. In *SGI Techpubs Library 007-4413-011*, 2006.

[19] 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.