A Memory-layout Oriented Run-time Technique for Locality Optimization[*]

Yong Yan
HAL Computer Systems, Inc.
1315 Dell Avenue
Campbell, CA 95008

Xiaodong Zhang Zhao Zhang
Computer Science Department
College of William & Mary
Williamsburg, VA 23187-8709

Abstract

Exploiting locality at run-time is a complementary approach to a compiler approach for those applications with dynamic memory access patterns. This paper proposes a memory-layout oriented approach to exploit cache locality for parallel loops at run-time on Symmetric Multi-Processor (SMP) systems. Guided by application-dependent hints and the targeted cache architecture, it reorganizes and partitions a parallel loop through shrinking and partitioning the memory-access space of the loop at run-time. In the generated task partitions, the data sharing among partitions is minimized and the data reuse in a partition is maximized. The execution of tasks in partitions is scheduled in an adaptive and locality-preserved way to achieve balanced execution, for minimizing the execution time of applications by trading off load balance and locality.

Based on simulation and measurement, we show our run-time approach can achieve comparable performance with the compiler optimizations for two applications, whose load balance and cache locality can be well optimized by the tiling and other program transformations. However, our experimental results also show that our approach is able to significantly improve the memory performance for the applications with dynamic memory access patterns. This type of programs are usually hard to be optimized by compilers.

1. Introduction

The increasing speed gap between the processor and the memory system makes techniques of latency hiding and reduction very important for both uniprocessor systems and multiprocessor systems. Recently, Symmetric Multi-Processor (SMP) systems have emerged as a major class of high-performance platforms, such as HP/Convex Exemplar S-class [1], Sun SPARCcenter 2000 [4], SGI Challenge [8], and DEC AlphaServer [18]. SMPs dominate the server market for commercial applications and are used as desktops for scientific computing. They are also important building blocks for large-scale systems. Because the access latency of a processor to the shared memory in a SMP is usually tens of times of that to a cache, improving the memory performance of applications on SMPs is crucial to the successful use of SMP systems.

The techniques for reducing the effect of long memory latency have been intensively investigated by researchers from application designers to hardware architects. The proposed techniques, so far, fall into two categories: *latency avoidance* and *latency tolerance* [9]. The latency tolerance techniques [7] are aimed at hiding the effect of memory-access latencies by overlapping computations with communications or by aggregating communications. Most of these techniques, while reducing the impact of contentionless access latencies, do so at the cost of increasing a program's bandwidth requirements [3]. The latency avoidance techniques, also called locality optimization techniques, are aimed at reducing low-level memory accesses using software and hardware approaches. In a SMP system, reducing the total number of accesses at low levels of the memory hierarchy is a substantial solution to reduce cache coherence overhead, memory contention, and network contention. Optimizing the locality of parallel computations is more demanding than tolerating their memory latency, which is the goal of this paper.

1.1. The problem

Locality optimization has been paid attention by many researchers for several years. The majority of the existing work focuses on compiler-based optimizations [6, 10, 11, 12, 14, 15]. Because a compiler can exploit detailed information of applications using comprehensive analysis techniques, compiler-based locality optimization tech-

[*]This research has been supported by the National Science Foundation under grants CCR-9400719, the Air Force Office of Scientific Research under grant AFOSR-95-1-0215 and by the Office of Naval Research under grant ONR-95-1-1239.

niques have been shown to be very effective in improving the performance of those applications to which they can be applied (see e.g. [6, 10, 11, 12, 14]). Unfortunately, many of the applications in the real world possess dynamic data-access patterns which cannot be analyzed at compiler time.

```
double A[X], B[Y], C[M][M];
int Arow[M+1], Acol[X], Bcol[M+1], Brow[Y];

sparse-mm()
{     int i, j, k, r, start, end;
      register double d;
      for (i=0; i<M; i++)
          for (j=0; j<M; j++){
              d = 0;
              start = Bcol[j]; end = Bcol[j+1];
              for (k=Arow[i]; k<Arow[i+1]; k++)
                  for (r=start; r<end; r++)
                      if (Acol[k] == Brow[r]){          task t(i, j)
                         d += A[k]*B[r];
                         start = r+1;
                         break;
                      }
              C[i][j] = d;
          }
}
```

Figure 1. A Sparse Matrix Multiplication (SMM) which has a dynamic data-access pattern and an irregular computation pattern.

In Figure 1, we present a sparse matrix multiplication algorithm where two sparse source matrices have dense representations. In the innermost loop, the two elements to be multiplied, `A[k]` and `B[r]`, are indirectly determined by the data in arrays `Arow`, `Acol`, `Bcol` and `Brow`. At compiler-time, because a compiler does not know what kind of data the program is going to process, it cannot determine how the program accesses its data. The data-access pattern of this program can only be determined at run-time when input data has been obtained. However, on a SMP system, the design of a run-time locality optimization technique is challenged by low overhead requirement and the complexities of minimizing data sharing among caches, maximizing data reuses in the cache, and trading off locality and the other performance factors.

1.2. Our solution and contributions

Because most data reuses of an application occur in loop structures [15] and the parallel loop is a major program structure in scientific applications [13, 14, 16, 22], we propose a run-time technique to improve the memory performance of parallel loops with dynamic data-access patterns.

In our run-time technique, the memory-access patterns of parallel tasks in a program are captured at run-time using a multi-dimensional memory-access space based on simple application-dependent hints. Based on the abstracted memory-access space and the cache architecture, the locality of a program is optimized through two types of space-based transformations: space shrinking and space partitioning. Then, tasks are adaptively scheduled to trade off locality and load imbalance, aiming at minimizing the parallel computing time. The proposed information abstraction and transformations can be efficiently implemented at acceptable overhead. Finally, with respect to three applications with different data-access patterns, the effectiveness of the proposed technique is evaluated in detail on an event-driven simulator and two commercial SMP systems,

1.3. Comparisons with related work

Exploiting cache locality at run-time has been paid attention by some previous work. References [13, 22] present dynamic loop scheduling algorithms that consider the affinity of loop iterations to processors. Although significant performance improvement can be acquired for some applications, the type of affinity exploited by this approach is not very popular and the relations between memory references of different iterations are not considered. The proposed technique in this paper not only takes into consideration the affinity of parallel tasks to processors, it also uses information on the underlying cache architecture and memory reference patterns of tasks to minimize cache misses and false sharing.

In the design of the COOL language [5], the locality exploitation issue is addressed by using language mechanisms and a run-time system. Both task affinity and data affinity are specified by users and then are implemented by the run-time system. A major limit with this approach is that the quality of locality optimizations totally depends on a programmer. For complicated applications, such as the example in Figure 1, it is difficult for a user to specify affinity. Our proposed technique uses a simple programming interface for a user or compiler to specify simple information about data, not about complicated affinity relations. Regarding the run-time locality optimization of sequential programs, reference [16] proposes a memory-layout oriented method. It reorganizes the computation of a loop based on some simple hints about the memory reference patterns of loops and cache architectural information. Compared with a uniprocessor system, a cache coherent shared memory system has more complicated factors that should be considered for locality exploitation, such as data sharing and load imbalance.

More recently, reference [2] uses a run-time system to color the virtual pages of a program based on both machine-specific parameters and a summary of the array access patterns generated by the high-level compiler. This approach still depends on the compiler-time static analysis on data-access patterns

1.4. Organization of this paper

The remaining of this paper contains four sections. The next section describes our run-time optimization technique in detail. Section 3 presents our performance evaluation method and performance results. In Section 4, we conclude our work.

2. A memory-layout oriented optimization technique

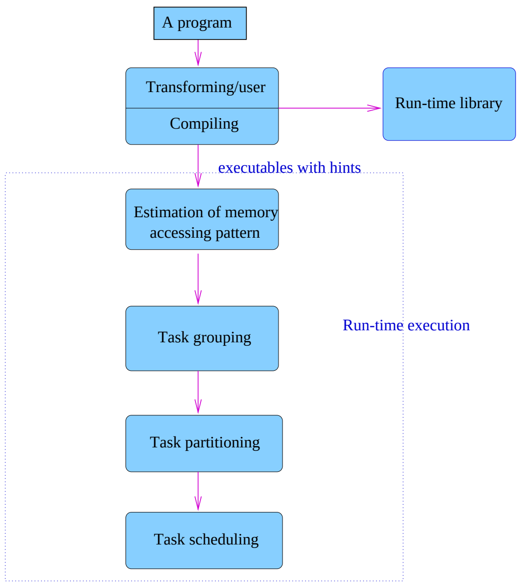


Figure 2. Framework of the run-time technique.

Our run-time technique has been implemented as a set of library functions. Figure 2 presents a framework for our run-time optimization. A given sequential application program is first transformed by a compiler or rewritten by a user to insert run-time functions. The generated executable file is encoded with application-dependent hints. At run-time, the encoded run-time functions are executed to fulfill the following functionalities: estimating the memory-access pattern of a program, reorganizing tasks into cache affinity groups where the tasks in a group are expected to heavily reuse their data in the cache, partitioning task-affinity groups onto multiple processors so that data sharing among multiple processors are is minimized, and then adaptively scheduling the execution of tasks.

In order to minimize run-time overhead, a multi-dimensional hash table is internally built to manage a set of task-affinity groups. Meanwhile, a set of hash functions are given to map into an appropriate task-affinity group in the hash table. Locality oriented task reorganization and partitioning are integratedly finished in the task mapping. This section describes our information estimation method, the design of the hash table and hash functions along with task reorganization and partitioning, and our task scheduling algorithm.

2.1. Memory-access pattern estimation

In a loop structure, referenced data are usually structured as arrays. Let $A_1, A_2, \cdots, A_n$ be the n arrays accessed in the loop body of a nested data-independent loop. Each array is usually laid out in a contiguous memory region, independent of the other arrays. In rare cases, an array may be laid out across several uncontiguous memory pages. Although our run-time system may not handle these rare cases efficiently, the system works well for most memory layout cases in practice. Visualizing an array in an independent dimension, the memory regions of the n arrays can be integratedly abstracted as an n-dimensional memory-access space, expressed as $(A_1, A_2, \cdots, A_n)$ where arrays are arranged in any selected order by a user. This n-dimensional memory-access space actually contains all the memory addresses that are accessed by a loop. This abstract is similar to that used in [16].

In order for the run-time system to precisely capture this memory space information, the following three hints must be provided by the interface.

Hint 1. *n, the number of arrays accessed by tasks.*

Hint 2. The *size* in bytes of each array. Based on this, the run-time system maintains a Memory-access Space Size vector $(s_1, s_2, \cdots, s_n)$, denoted the MSS vector, where s_i is the size of i-th array $(i = 1, 2, \cdots, n)$.

Hint 3. The *starting memory address* of each array. From this, the underlying run-time system constructs a starting address vector $(b_1, b_2, \cdots, b_n)$, denoted the SA vector, where $b_i (i = 1, 2, \cdots, n)$ is the starting memory address of i-th array.

Here **Hint 1** is a static information. The array size may be static if the size is known at compiler-time or dynamic if the size is determined by run-time data and the hint should tell how to calculate the size at run-time. The starting addresses are dynamic because memory addresses can only be determined at run-time. **Hint 3** tells how to determine the starting addresses at run-time.

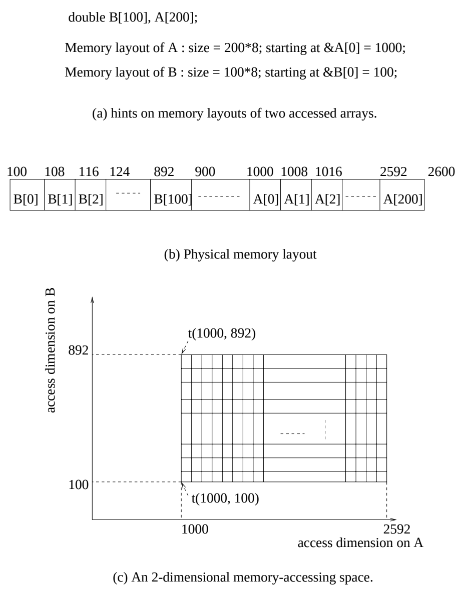
After determining the global memory-access space of a loop, we need to determine how each parallel iteration accesses the global memory-access space so that we can reorganize them to improve memory performance. Here, we abstract each instance of a loop body of a parallel loop as a parallel task. The access region of a task in an array is simply represented by the starting address of its access region. So, the following hint should be provided by interface functions.

Hint 4: *A memory-access vector of task t_j*:

$$(a_{j1}, a_{j2}, \cdots, a_{jn})$$

where a_{ji} is the starting address of the referenced region on i-th array by t_j $(i = 1, 2, \cdots, n)$. In some loop structures, a parallel iteration may not contiguously access an array so that the access region may not be precisely abstracted by the starting address. In this case, the loop iteration should be further split into smaller iterations so that each iteration accesses a contiguous region on each array. In addition, the following hint also should be provided to assist task partitioning.

Hint 5: *The number of processors, p.*



```
double B[100], A[200];

Memory layout of A : size = 200*8; starting at &A[0] = 1000;
Memory layout of B : size = 100*8; starting at &B[0] = 100;
```

(a) hints on memory layouts of two accessed arrays.
100 108 116 124 892 900 1000 1008 1016 2592 2600
B[0] B[1] B[2] B[100] A[0] A[1] A[2] A[200]
(b) Physical memory layout
access dimension on B
892
100
t(1000, 892)
t(1000, 100)
1000
2592
access dimension on A
(c) An 2-dimensional memory-accessing space.

Figure 3. Memory-access space representation.

Based on the above hints, the memory-access space of the loop is abstracted as a n-dimensional memory-access space:

$$(b_1 : b_1 + s_1 - 1, b_2 : b_2 + s_2 - 1, \cdots, b_n : b_n + s_n - 1).$$

Task t_j is abstracted as point $(a_{j1}, a_{j2}, \cdots, a_{jn})$ in the memory-access space based on the estimation on its memory-access pattern. Figure 3 presents an example of the abstract representation of the memory accesses based on the physical memory layout of arrays `A` and `B` in the SMM given in Figure 1. Figure 3(a) gives the hints on the memory-access space. Figure 3(b) illustrates the memory layout of two arrays where `B` and `A` are laid out at starting address 100 and 1000 respectively. Each array element has size of 8 bytes. The memory space of arrays A and B is the whole memory space accessed by tasks. Then, the memory-access space is represented as a 2-dimensional space as shown in Figure 3(c) where each point gives a pair of possible starting memory-access addresses on `A` and `B` respectively by a task. For example, `t(1000, 100)` means task `t` will access array `A` at starting memory address 1000, and access array `B` at starting physical address 100.

2.2. Task reorganization

In the memory-access space, nearby task points access the same or nearby memory addresses in memory. So, grouping nearby tasks in the memory access space has a good change to enhance temporal locality and spatial locality when they execute together. This is achieved by shrinking the memory-access space based on the underlying cache size.

Let $\{t_i(a_{i1}, a_{i2}, \cdots, a_{in}) | i = 1, 2, \cdots, m\}$ be a set of m data-independent tasks of a parallel loop, and $(b_1 : b_1 + s_1 - 1, b_2 : b_2 + s_2 - 1, \cdots, b_n : b_n + s_n - 1)$ be the memory-access space of the parallel loop. Conceptually, task t_i (i=1, $\cdots$, n) is mapped onto point $(a_{i1}, a_{i2}, \cdots, a_{in})$ in the memory-access space based on the starting memory addresses of their memory-access regions. In addition, let p be the number of processors and C be the capacity of the underlying secondary cache in bytes.

Task reorganization consists of two steps. In the first step, the memory-access space $(b_1 : b_1 + s_1 - 1, b_2 : b_2 + s_2 - 1, \cdots, b_n : b_n + s_n - 1)$ is shifted into origin point (0, $\cdots$, 0) by subtracting $(b_1, b_2, \cdots, b_n)$ from the coordinates of all task points. In the second step, we use *equal-shrinking* method to shrink each dimension of the shifted memory by fC/n. The n-dimensional space resulted from shrinking is called a n-dimensional bin space. Here, f is a weight constant in (0, 1]. In the bin space, each point is associated with a task bin to hold all the tasks that are mapped into the task bin.

In Figure 4, the shrinking procedure of the memory-access space is exemplified by the 2-dimensional memory-access space given in Figure 3. Before shrinking, the original memory-access space is shifted to origin point (0,0) (see Figure 4(b)). The shifting function is shown in Figure 4(b). Then each dimension of the shifted memory-access space is shrunk by C/2 into a new 2-dimensional bin space in Figure 4(c). The tasks in the shadow square in Figure 4(b) would not access more space than the cache size, and are mapped onto one point in the bin space so that they can be grouped together to execute.

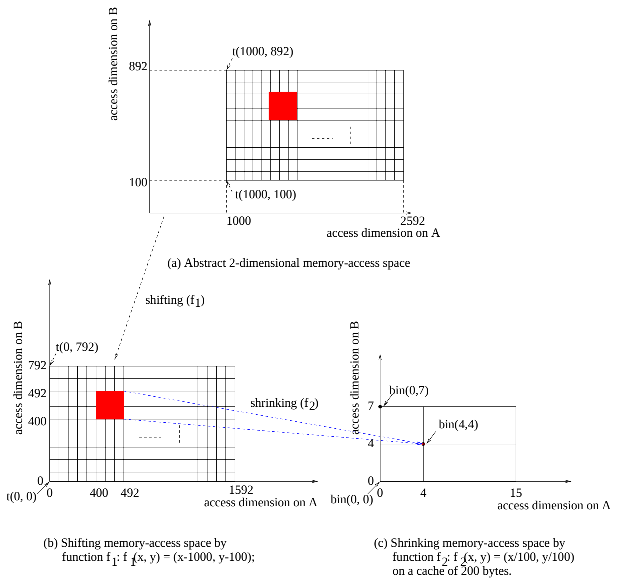

access dimension on B
t(1000, 892)
892
100
t(1000, 100)
1000
2592
access dimension on A
(a) Abstract 2-dimensional memory-access space
shifting (f1)
t(0, 792)
792
492
400
0
t(0, 0)
0
400
492
1592
access dimension on A
shrinking (f2)
bin(0,7)
7
bin(4,4)
4
0
bin(0, 0)
0
4
15
access dimension on A
(b) Shifting memory-access space by function f1: f1(x, y) = (x-1000, y-100);
(c) Shrinking memory-access space by function f2: f2(x, y) = (x/100, y/100) on a cache of 200 bytes.

Figure 4. Equally shrinking a memory-access space.

2.3. Task partitioning

After shrinking an n-dimensional memory-access space, tasks have been grouped based on locality affinity information in an n-dimensional bin space. Task partitioning is aimed at partitioning the n-dimensional bin space into p partitions (p is the number of processors and each partition is an n-dimensional polyhedron) so that

1. *the data sharing degree among partitions is minimized*, which is measured by the volume of boundary spaces among partitions.

2. *p partitions are balanced*, where the balance refers to partitions with the same volume.

The major function of partitioning an n-dimensional bin space B^n(0:L_1, 0:L_2, $\cdots$, 0:L_n) is to find a partitioning vector $\vec{k}(k_1, k_2, \cdots, k_n)$ so that the above conditions are satisfied. Because finding an optimal partition vector is a NP-complete problem, we propose a heuristic algorithm based on the following partitioning rules. Detailed proofs can be found in [21].

Theorem 1 Ordering Rule
For a given partitioning vector $\vec{k}(k_1, k_2, \cdots, k_n)$ not in decreasing order, the partitioning vector resulting by sorting $\vec{k}$ in decreasing order is at least as good as $\vec{k}$ in terms of the sharing degree.

Theorem 2 Increment Rule 1
For an n-dimensional bin space B^n, and partitioning vectors $\vec{k}$(k_1, k_2, $\cdots$, k_i, k_{i+1} $\times$ q, 1, $\cdots$, 1) and $\vec{k}'$(k_1, k_2, $\cdots$, k_i $\times$ q, k_{i+1}, 1, 1, $\cdots$, 1), where $q > 1$, $\vec{k}$ is better than $\vec{k}'$ in terms of the sharing degree if and only if

$$k_i \times L_{i+1} > k_{i+1} \times L_i.$$

Corollary 1 Increment Rule 2
For an n-dimensional bin space B^n, and partitioning vectors $\vec{k}(k_1, k_2, \cdots, k_i, k_{i+1}, 1, \cdots, 1)$ and $\vec{k}'(k_1, k_2, \cdots, k_i \times k_{i+1}, 1, 1, \cdots, 1)$, where $k_{i+1} > 1$, $\vec{k}$ is better than $\vec{k}'$ in terms of the sharing degree if and only if

$$k_i \times L_{i+1} > \times L_i.$$

Based on the above three rules, we design an efficient heuristic algorithm as follows.

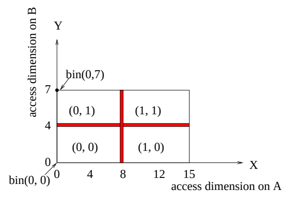
1. Factor p, the number of processors, to generate all the prime factors of p in decreasing order. Assume that there are q prime factors: $r_1 \geq r_2 \geq \cdots \geq r_q$. Initially, the n-dimensional partitioning vector $\vec{k}$, stored in $k[1:n]$, is (1, 1, $\cdots$, 1) for the bin space $B^n(0:L_1, 0:L_2, \cdots, 0:L_n)$.

2. Let $last$ index the position in $k[1:n]$ where $k[i] > 1$ for $i < last$ and $k[i] = 1$ for $i \geq last$. Initially, $last$ = 1. For each prime factor r_j where j increases from 1 to q, do the following:

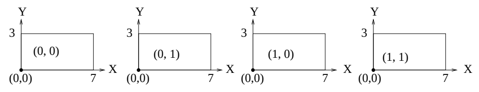
   (a) When ($last \leq n$), use the increment rule 2 to determine whether r_j should be put in $k[last]$. Based on the ordering rule, the best place to put r_j must be in $k[1:last]$. So, we use increment rules to find a better place in $k[1:last]$. If so, $last$ is increased by 1 and go back; otherwise, use the increment rule 1 to put r_j together with $k[last-1]$ or $k[last-2]$, then reorder $k[1:last-1]$ in decreasing order and go back.

   (b) Otherwise: use the increment rule 1 to put r_j together with $k[last-1]$ or $k[last-2]$, then reorder $k[1:last-1]$ in decreasing order and go back.

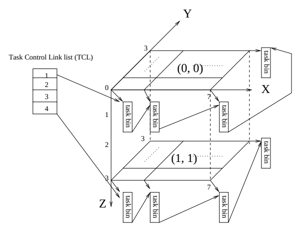
The above algorithm has a computational complexity $O(n+\sqrt{p})$. After the determination of a partitioning vector, the bin space is partitioned into multiple independent spaces that are further reconstructed in a $(n+1)$-dimensional space. This procedure is shown in Figure 5 where the bin space produced in Figure 4 is partitioned by vector (2, 2). The partitions in Figure 5(a) are first transformed into four independent spaces in Figure 5(b), which are further transformed into a 3-dimensional space shown in in Figure 5(c). The 3-dimensional space in Figure 5(c) is implemented as



(a) Indexing of partitions.



(b) independent address space of each partition.



(c) 3-dimensional internal representation of the memory access space.

Figure 5. Partitioning: the bin space is evenly divided into 4 partitions from X and Y dimensions.

a 3-dimensional hash table where task bins in each partition are chained together to be pointed by a record in a Task Control Linked (TCL) list. The hashing of tasks into the hash table is finished by the space transformation functions (details are presented in [21]).

2.4. Task scheduling

In order to minimize the parallel computing time of partitioned tasks, we present a Locality-preserved Adaptive Scheduling (LAS) algorithm by extending our linearly adaptive algorithm proposed in [22].

Initially, the i-th task group chain in the TCL list is considered to be the local task chain of processor i, for $i = 1, 2, \cdots, p$ (p is the total number of processors). Each task chain has a head and a tail. The initial allocation maintains the minimized data sharing achieved in the task reorganization step among processors. The number of current tasks in the local chain of processor i is recorded by a TCL counter, denoted C_i, which is used in the LAS algorithm to estimate load imbalance. In addition, each processor has a chunking control variable of initial value of p, denoted K_i for processor i, to determine how many tasks to be executed at each scheduling step.

The scheduling algorithm still works in two phases: the local scheduling phase and the global scheduling phase. All the processors start at the local scheduling phase. In the local scheduling phase, processor i calculates its load status relative to the other processors as follows:

$$\text{heavy} \quad \text{if } C_i > \textstyle\sum_{j=1}^{p} C_j/p + \alpha \tag{1}$$

$$\text{light} \quad \text{if } C_i < \textstyle\sum_{j=1}^{p} C_j/p - \alpha \tag{2}$$

$$\text{normal} \quad \text{otherwise} \tag{3}$$

Here, α is $\lceil \sum_{j=1}^{p} C_j/p)/(2p) \rceil$, which decreases with the execution to control the load distribution more closely. Then, it adjusts its chunking control variable, K_i, as following:

$$K_i = \begin{cases} \max\{p/2, K_i - 1\} & \text{if its load is light} \\ \min\{2p, K_i + 1\} & \text{if its load is heavy} \\ K_i & \text{otherwise} \end{cases} \tag{4}$$

Finally, processor i gets $1/K_i$ of remaining tasks from the head of its local task chain to execute. The varying range $[p/2, 2p]$ for the chunking control variables has been shown to be safe for balancing load [13, 22].

When processor i finishes its local tasks, it sets its chunking control variable, K_i, to p, then enters the global scheduling phase where it will get $1/K_i$ of remaining tasks on the most heavily load processor from the tail of the task chain.

3. Performance evaluation

3.1. Evaluation method

We implemented our locality optimization technique as three simple run-time library functions. Performance evaluation is based on simulation and measurement. Simulation was conducted on an event-driven simulator for bus-based shared memory systems, which was built on the MINT, a MIPS interpreter [19]. Measurements were conducted on two commercial systems: HP/Convex S-class which is a crossbar-based cache coherent SMP system with 16 processors, and Sun Ultr-SPRACstation-20 which is a bus-based cache coherent SMP system with 4 processors.

The selected applications are: (1) dense matrix multiplication, denoted as DMM, that has a regular computation pattern and a static data-access pattern, adjoint convolution, denoted as AC, that has a irregular computation pattern and a static data-access pattern, and sparse matrix multiplication with 30% non-zero elements, denoted as SMM, that has a irregular computation pattern and a dynamic data-access pattern. Their optimized versions by exploiting locality using our runtime library are denoted as DMM_LO, AC_LO, and SMM_LO respectively. For comparison, the three benchmarks are parallelized respectively using the best existing techniques as follows. (1) For the DMM application, we parallelized the blocked matrix multiplication algorithm given by Wolf and Lam [20]. This program is denoted as DMM_WL. (2) For the AC application, we first used loop split, loop reverse, and loop fusion transformations to get a balanced outer loop, which is then equally partitioned across processors. This program is denoted as AC_BF. (3) For the SMM application, we used the linearly adaptive scheduling technique proposed in [22]to schedule the executions of parallel iterations in SMM. This program is denoted as SMM_A. The detailed description about these programs are given in [21].

3.2. Performance results

Processors	Miss rate					
	DMM application		AC applic ation		SMM application	
	DMM_WL	DMM_LO	AC_BF	AC_LO	SMM_A	SMM_LO
2	0.006	0.008	0.051	0.043	0.025	0.011
4	0.006	0.008	0.051	0.044	0.025	0.011
8	0.005	0.007	0.052	0.044	0.025	0.012

Table 1. Cache miss-rate based comparison where experiments were conducted under shrinking factor f = 1.

Table 1 presents the miss rates of the six benchmark programs on 2 processors to 8 processors. Regarding regular application DMM, the locality-optimized version (DMM_LO) using the run-time technique is 9% to 14% higher than the well-tuned version (DMM_WL) in the number of cache misses (Table 1). AC_LO, a locality optimized program of AC using the run-time technique, is shown to achieve slightly better cache performance than AC_BF, a well-tuned program. Regarding the application SMM, the run-time locality technique is shown to be very effective in reducing cache misses. The cache miss rate was reduced for more than 50% as shown in Table 1.

program	On HP S-class					On SUN SPARC		
	size	processors				size	processors	
		2	4	8	16		2	4
DMM_WL	1024	11	5.7	3.0	1.8	1024	108	57
DMM_LO	1024	13	6.6	3.9	2.2	1024	115	63
AC_BF	400	180	102	65	39	256	763	390
AC_LO	400	144	91	60	38	256	698	349
SMM_A	1024	4.1	2.5	1.4	0.8	1024	37	20
SMM_LO	1024	2.2	1.3	0.5	0.5	1024	23	12

Table 2. Measured time (in seconds) based comparison. (f = 1).

Table 2 presents execution comparisons on two SMP systems. Measured load balance is presented in Table 3. Regarding the DMM program, DMM_WL consistently performed a little bit better than DMM_LO, not larger than 20% on both SMP systems. The better load balance in DMM_WL is a reason for this. This shows that the run-time optimization can also achieve a comparable performance with the compiler-based optimization for regular applications. For program AC, AC_LO performed better than AC_BF on two processors on both SMP systems. When more processors were applied, the execution times were close. But, AC_BF always balanced load better due to its perfect initial partition. But, the load imbalance occurred in the AL_LO was no larger than 1%. This shows that the run-time optimization has chance to outperform compiler-based optimization for applications with irregular computation pattern. For SMM, SMM_LO had achieved a much better performance improvement over the SMM_A. About 50% reduction in execution time was observed for all test cases on both SMP systems. This confirms the effectiveness of the run-time technique in improving the performance of applications with dynamic memory-access patterns

program	On HP S-class					On SUN SPARC		
	size	processors				size	processors	
		2	4	8	16		2	4
DMM_WL	1024	0.0026	0.0052	0.0095	0.010	1024	0.01	0.02
DMM_LO	1024	0.024	0.021	0.038	0.040	1024	0.06	0.03
AC_BF	400	0.0007	0.001	0.0018	0.0031	256	0.002	0.003
AC_LO	400	0.003	0.004	0.006	0.010	256	0.003	0.005
SMM_A	1024	0.02	0.03	0.04	0.06	1024	0.012	0.022
SMM_LO	1024	0.03	0.05	0.06	0.06	1024	0.035	0.038

Table 3. Measured load imbalance in terms of the rate of the time deviation to the mean time. (f = 1).

Table 4 presents run-time overhead measurements. On both SMP systems, the run-time overhead is not larger than 10% of the execution time in all cases except one. This shows the effectiveness of using hash table and hash functions to integrate locality optimizations at run-time.

program	On HP S-class					On SUN SPARC		
	size	processors				size	processors	
		2	4	8	16		2	4
DMM_LO	1024	6	8	9	10	1024	9	10
AC_LO	400	0.3	0.25	0.3	0.3	256	0.1	0.2
SMM_LO	1024	5	8	16	2	1024	9	10

Table 4. Run-time overhead in percentage of total time. (f = 1).

In Table 5, we show the effects of selecting different shrinking factors for f. The change in f does affect the execution times of benchmark programs. But, this effect is not very big. So, we recommend to use f=1 for programming ease.

4. Conclusion

In this paper, we have presented a run-time locality optimization technique and have shown its effectiveness for optimizing the memory performance of applications with dynamic memory-access pattern. For those applications

Application	Machine	value of f			
		1	0.5	0.25	0.125
DMM_LO (N=1024)	S-class	6.6	6.1	5.8	5.8
DMM_LO (N=1024)	HyperSPARC	63	64	58	59
AC_LO (N=400)	S-class	91	90	91	90
AC_LO (N=256)	HyperSPARC	349	347	352	373
SMM_LO (N=1024)	S-class	1.3	1.3	1.4	1.5
SMM_LO (N=1024)	HyperSPARC	12	13	14.6	14.2

Table 5. The effects of different values of f on execution time (in seconds) using 4 processors.

whose memory performance can be optimized well by current compiler-based optimizations, the presented run-time technique can achieve comparable performance. Using the hash table and hash functions has been shown to be an effective way to reduce run-time overhead.

Our work does have some limits that need to be studied further. (1) The access-pattern of a task on an array is estimated only by a starting address. When a task accesses several non-contiguous regions on an array, multiple starting addresses should be used or the task should be split into several small tasks. Further investigation is needed for this case. (2) In space shrinking, we only use an equal-shrinking approach. When tasks accesses different arrays in different ways, a non-equal shrinking approach should be used. But, the difficulty is how to estimate array-access patterns at low cost. (3) The program structure should be extended to consider data-dependence. By combining our run-time locality optimization technique with the run-time parallelization technique given in [17], more general program structures can be handled.

ACKNOWLEDGEMENT: We thank Dr. C. C. Douglas for sending us their thread library. We are grateful to Dr. Greg Astfalk for his constructive suggestions and help in using the HP/Convex S-class. Finally, we appreciate Neal Wagner's careful reading of the manuscript and constructive comments.

References

[1] G. Astfalk and T. Brewer. An overview of the HP/convex exemplar hardware. Technical report, Hewlett-Packard Inc., System Technology Division, 1997.

[2] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. Compiler-directed page coloring for multiprocessors. *Proceedings of ASPLOS'96*, pages 244–255, Oct. 1996.

[3] D. Burger, J. R. Goodman, and A. Kagi. Limited bandwidth to affect processor design. *IEEE Micro*, pages 55–62, November/December 1997.

[4] M. Cekleov and et al. SPARCcenter 2000: Multiprocessing for the 90's. *IEEE COMPCON*, pages 345–353, February 1993.

[5] R. Chandra, A. Gupta, and J. L. Hennessy. Data locality and load balancing in cool. *Proceedings of PPOPP'93*, pages 249–259, May 1993.

[6] S. Coleman and K. S. Mckinley. Tile size selection using cache organization and data layout. *Proceedings of PLDI'95*, pages 279–289, June 1995.

[7] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., U. S. A., 1997.

[8] M. Galles and E. Williams. Performance optimizations, implementation, and verification of the sgi challenge multiprocessor. *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, pages 134–143, Jan. 1994.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., U. S. A., 1996.

[10] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. *Proceedings of PPOPP'95*, pages 179–188, July 1995.

[11] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multilevel blocking. *Proceedings of PLDI'97*, pages 346–357, May 1997.

[12] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. *Proceedings of ASPLOS'91*, pages 63–74, April 1991.

[13] E. P. Markatos and T. J. Leblanc. Using processor affinity in loop scheduling scheme on shared-memory multiprocessors. *IEEE Trans. Para. & Dist. Syst.*, 5(4):379–400, April 1994.

[14] K. S. McKinley, S. Carr, and C. W. Tseng. Improving data locality with loop transformations. *ACM Trans. Prog. Lang. Syst.*, 18(4):424–453, July 1996.

[15] K. S. Mckinley and O. Teman. A quantitative analysis of loop nest locality. *The Proceedings of ASPLOS'96*, pages 94–104, Oct. 1996.

[16] J. E. Philbin, O. J. Anshus, C. C. Douglas, and K. Li. Thread scheduling for cache locality. *Proceedings of ASPLOS'96*, pages 60–71, Oct. 1996.

[17] J. H. Saltz and R. Mirchandaney. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.*, pages 603–612, May 1991.

[18] M. B. Steinman, G. J. Harris, A. Kocev, V. C. Lamere, and R. D. Pannell. The alphaserver 4100 cached processor module architecture and design. *Digital Technical Journal*, 8(4):21–37, 1996.

[19] J. E. Veenstra and R. J. Fowler. Mint: A front end for efficient simulation of shared-memory multiprocessors. *Proceedings of MASCOTS'94*, pages 201–207, Jan. 1994.

[20] M. E. Wolf and M. Lam. A data locality optimizing algorithm. *Proceedings of PLDI'91*, pages 30–44, June 1991.

[21] Y. Yan. *Exploiting Cache Locality at Run-time*. PhD thesis, Computer Science Department, College of William & Mary, May 1998.

[22] Y. Yan, C. M. Jin, and X. Zhang. Adaptively scheduling parallel loops in distributed shared-memory systems. *IEEE Trans. Para. & Dist. Syst.*, 8(1):70–81, Jan. 1997.