2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing

Locality-Aware Dynamic Mapping for Multithreaded Applications

Betul Demiroz*, Haluk Rahmi Topcuoglu*, Mahmut Kandemir† and Oguz Tosun‡
*Computer Engineering Department, Marmara University, 34722, Istanbul, Turkey
Email: {betul.demiroz,haluk}@marmara.edu.tr
†Dept. of Computer Science and Engineering, Pennsylvania State University,
University Park, PA 16802, USA
Email: kandemir@cse.psu.edu
‡Computer Engineering Department, Bogazici University, 34342, Istanbul, Turkey
Email: tosuno@boun.edu.tr

***Abstract*—Locality analysis of an application helps us extract data access patterns and predict runtime cache behavior. In this paper, we propose a locality-aware dynamic mapping algorithm for multithreaded applications, which assigns computations with similar data access patterns to same cores. We collect the amounts of shared and distinct data used by all computations, called *chunks* and calculate sharing among those chunks. Then, chunks with the similar data access patterns are grouped into bins, which are subsequently assigned to threads for improving cache reuse and program performance. Our algorithm is illustrated with sparse matrix-vector multiply (SpMV), which is one of the most widely used kernel in engineering and scientific computing and suffers from irregular and indirect memory access patterns. Five inputs with different shapes and characteristics are considered for testing the performance of our algorithm. Based on the results of experimental study, our algorithm outperforms Linux scheduler with an average of 12.5% performance improvement for various scenarios considered.**

***Index Terms* - Chip Multiprocessors, dynamic mapping, cache behavior, multithreading.**

I. Introduction

Almost all major chip manufacturers are working on Chip Multiprocessors (CMP) design and optimization, and one can expect that CMPs become the standard and the primary building blocks for personal desktop computers as well as large scale parallel machines. Complex configurations with high number of cores have been delivered and prototyped including Suns' 8-core Niagara, IBM's Cell and Intel's 80-core TeraFlop.

In order to efficiently utilize the underlying systems, it is important to observe the runtime behaviors of the running threads and provide a dynamic thread mapping strategy. In [1] and [2], instead of application's execution trace, the similarity between program's parts are used for taking the thread assignment decision. This work contains two phases, the first phase uses a static analysis technique to group similar sections in a program and the second phase deals with scheduling. Another dynamic scheduling method is given in [3], which clusters threads that are sharing the same cache access pattern to same cores.

The management of the shared on-chip space is a critical issue on CMPs. A data element on the cache can be replaced by another, and when the displaced data element is requested again, a cache miss occurs. The overall performance is significantly affected by the conflicts on the shared cache. In order to obtain high performance from CMPs, the reuse of data elements on the cache should be increased. Shared cache conflicts can occur among threads of different applications, and these conflicts are more frequent when threads of the same application are mapped to different cores [4]. In order to predict the shared cache contention, a lightweight locality model is proposed in [5]. Various heuristics that attempt to increase cache performance of multiple multithreaded applications with large working sets is explored in [6]. Khanna et al. [7] proposed a hypergraph based locality aware dynamic scheduling algorithm that maps a stream of dynamically arriving jobs sharing input files to compute nodes and determines the order of the jobs in each node.

In this paper, a locality-aware mapping policy that observes the runtime characteristics of an application is proposed. This policy focuses on effective usage of caches by reusing data manipulated by computations of the same core. The application is divided into a number of computations, and computations performed on consecutive blocks of data are represented with *chunks*. Those chunks that have similar data access patterns are grouped into *bins*. Similarity calculations among chunks consider both the shared and distinct data, because using only shared data for similarity calculation would neglect the amount of distinct data, which would decrease the data reuse probability on the cache. The locality-aware mapping algorithm can be applied to any multithreaded application and would yield high performance improvements, especially for those with irregular data access patterns.

In order to validate the effectiveness of our proposed algorithm, we consider the sparse matrix-vector multiply (SpMV) by using the classical blocked version of the compressed sparse row (CSR) format. We perform a simulation study by considering various inputs with different shapes and characteristics. Our algorithm increases data reuse on the shared cache for all cases, and our proposed mapping strategy outperforms the Linux Scheduler for different distributions of nonzero data elements. Our proposed locality-aware mapping policy

978-0-7695-4633-9/12 $26.00 © 2012 IEEE
DOI 10.1109/PDP.2012.84

achieves a maximum speedup of 26.8% by decreasing the number of L2 cache misses by about 53% when compared against the baseline Linux scheduler.

The rest of this paper is structured as follows. The next section discusses the main issues regarding our dynamic locality-aware mapping policy. Section 3 presents the details of experimental setup and the results of our experimental evaluation. Finally, Section 4 summarizes our main conclusions.

II. Locality-aware Dynamic Mapping

Data reuse for multithreaded applications is a critical issue. Especially if the application's data distribution is known at runtime, data-to-thread distribution can be done dynamically. Therefore, locality-aware dynamic mapping can be considered to increase data reuse on the shared cache. Data-intensive, multithreaded applications operating on any type of data could be the target application domain for locality-aware dynamic mapping.

The main idea underlying this algorithm is to assign computations with similar data access patterns to same cores. An application program may access different data elements throughout execution, and the sequence of data elements retrieved by the application may change. In the ideal case, a data element is accessed consecutively; and therefore, it can be found on the cache. On the other hand, data element accesses are unordered in the worst case. In most of the applications, we do not observe the ideal case; therefore, the probability of data element reuse on the shared cache is low.

Many applications obtain their program data at runtime. In order to find the similarities between data elements, a dynamic strategy should be considered. As soon as the application acquires its input data, our approach begins with the similarity calculations and forms groups of data elements that have the maximum amount of sharing. These groups are then assigned to application threads, and these threads start their execution. Thus our proposed algorithm starts execution after the input data is read, and then it divides the program computations into chunks. By using the data elements accessed in each chunk, the locality-aware mapping algorithm groups chunks using similar data elements into bins, and it finishes its execution.

Figure 1 shows how similarity calculations are performed in our algorithm. While calculating similarities in our scheme, we are trying to find two chunks with the maximum number of shared data and minimum number of distinct data. Each chunk is represented with a bit vector to store the data that is accessed by the computation it is representing as given in Figure 1, lines 12-18. The bit vector of each chunk, according to the data accesses, is defined as follows:

$$chunk_{i,j} = \begin{cases} 1, & \text{if } data_j \text{ is accessed by } chunk_i \\ 0, & \text{otherwise.} \end{cases}$$

If two bit vectors are the same, this shows that data accesses of these two chunks are the same; if the resulting vector obtained by using AND operation on two bit vectors is 0, two chunks share no common data at all. In general, to maximize

```
Input: Number of bins (binNumber),
       Number of chunks per bin(chunkPerBin),
       Total data elements to be used by all threads (TDE)
Output: Pairwise similarity values of chunks
1. totalChunks = binNumber * chunkPerBin
2. for i=0 to totalChunks do
3.   chunksBin_i = 0
4.   for j=0 to TDE do
5.     bitVector_{i,j} = 0; //Initialize bit vector for each chunk
6.   end for
7.   for j=0 to totalChunks do
8.     totalShared_{i,j} = 0;
9.     totalDistinct_{i,j} = 0;
10.  end for
11. end for
12. for i=0 to totalChunks do
13.   for j=0 to TDE do
14.     if TDE_j is accessed by Chunk_i then
15.       bitVector_{i,j} = 1.
16.     end if
17.   end for
18. end for
19. for i=0 to totalChunks do
20.   for j=i+1 to totalChunks do
21.     shared = bitVector_i AND bitVector_j
22.     distinct = bitVector_i XOR bitVector_j
23.     mask=1
24.     while shared > 1 do
25.       totalShared_{i,j} = totalShared_{i,j} + shared AND mask
26.       shared = shared >> 1
27.     end while
28.     mask=1
29.     while distinct > 1 do
30.       totalDistinct_{i,j} = totalDistinct_{i,j} + distinct AND mask
31.       distinct = distinct >> 1
32.     end while
33.   end for
34. end for
35. for i=0 to totalChunks do
36.   for j=i+1 to totalChunks do
37.     similarity_{i,j} = totalShared_{i,j}/(totalDistinct_{i,j}+totalShared_{i,j})
38.   end for
39. end for
```

Fig. 1. Similarity calculations of chunks.

cache hit ratio and improve performance, we should group chunks that can reuse the highest amount of data.

The similarity between two chunks is computed by performing AND and XOR operations on two bit vectors as shown in Figure 1, lines 19-34. AND operation on two bit vectors calculates the total number of shared data between two chunks; whereas XOR operation calculates the total number of distinct data between two chunks. The similarity between two chunks is calculated by the following formula:

$$similarity_{i,j} = \frac{totalShared_{i,j}}{totalDistinct_{i,j} + totalShared_{i,j}}.$$

where $totalShared_{i,j}$ represents the total number of shared data points, and $totalDistinct_{i,j}$ is the total number of distinct data accessed by both chunks. It is obvious that two chunks with high data sharing and low data difference have higher similarity, so there is a high data reuse between them. We want to show that simply using data sharing or data difference would not capture the most similar chunk pair. If only the number of shared data points is used as the similarity metric, we would be neglecting the data differences, which could be high enough to destroy the sharing pattern. The number of

data points accessed by different chunks may vary; therefore, the ratio of data sharing and total accessed data would be more appropriate to capture both of the constraints.

Similarities between bit vectors are sorted, and chunks with highest similarity values are grouped and represented with *bins*. The number of chunks in each bin may vary according to the size of the data elements used in the application. Each bin contains equal number of chunks for a fair distribution of data elements. Our aim is to maximize data reuse on cache, so we group all chunks with high similarity into one bin. When assigning chunks to bins, we take into account both pairwise similarity between chunks and the similarity obtained from the chunks that are already assigned to bins to find the best bin for the given chunk pair.

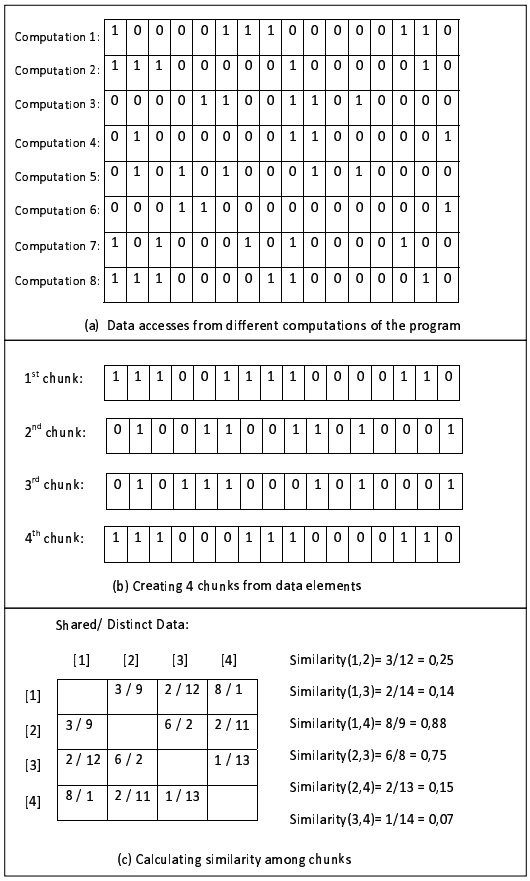

Fig. 2. Calculating similarity values for the case of 4 chunks with 2 bins.

An Example for Calculating Similarity Values of Chunks
To explain similarity calculations between chunks and our chunk assignment policy, an example application running 2 threads on 2 cores accessing different caches is given in Figure 2. In this example, we consider 2 bins. Assuming that the application consists of 8 computations, Figure 2.a shows the data elements accessed in each computation. Consider that 4 chunks are used, 2 contiguous computations are assigned to a chunk, and the bit vectors of the chunks will be formed as shown in Figure 2.b. If we take a closer look at the first chunk, all application data accessed in the first and second computations should be included in this chunk, so we can obtain the bit vector of the first chunk by using the OR operation on the data accessed in the first and the second computations of the application.

After generating all the other bit vectors in the same way, we calculate similarities between chunk pairs as shown in Figure 2.c and assign the most similar chunks to the same bin. In order to calculate the number of shared and distinct data points, AND and XOR operations are used, respectively. When the similarities between chunks are considered, chunk 1 and chunk 4 have the highest similarity value of 0.88, so these chunks are assigned to the same bin. As for chunk 2 and chunk 3, similarity value of 0.75 is obtained, and they are assigned to the second bin.

III. Experimental Evaluation

A. Setup and Methodology

For performing our experiments, a multiprocessor simulation environment, Simics toolset [8], is used. Each configuration in the experiments runs Fedora 5 operating system. We modeled our target hardware with 16 cores, and each core has a private L1 cache and a private L2 cache. Also, the number of application threads used is equal to the number of cores in the target architecture, and each application thread is assigned to a single core. Our main configuration parameters and their default values are listed in Table I.

TABLE I
Simulated architecture parameters of experimental study.

$Parameter$	$Value$
Number of cores (n)	16
Number of threads (p)	16
System	PCI Based X86 System
Processor Type	$Pentium$ 4
CPU Frequency	3.5 GiB
Main Memory	1 GB
Private L1 Instruction Cache	8K, 2-way
Private L1 Data Cache	8K, 2-way
Private L2 Cache	32K, 4-way

In this work, we used SpMV code to evaluate the potential of our proposed approach, but it can be applied to any multithreaded application by representing the shared data elements with chunks. SpMV is a critical application that effects the performance of numerous applications in different domains, including economy, science, and engineering. Unlike other scientific kernels, such as dense linear algebra kernel, SpMV requires many indirect and irregular memory accesses; therefore efficient data distribution is crucial for SpMV. The properties of the sparse matrices can only be known at runtime, so code transformations as well as dynamic optimizations and tuning are needed for higher performance [9], [10], [11]. The performance of SpMV is also studied on multicore architectures [12] and GPUs [13], [14]. The data that SpMV application will use may not be known at compile time, and to obtain a sharing-aware data distribution to threads, data should

either be preprocessed or dynamically mapped to threads. So this application is suitable for measuring the performance of our algorithm. SpMV works on a sparse matrix A and a vector X and calculates $AX=B$. SpMV loads and stores the values of sparse matrix A and B only once; however, vector X is accessed indirectly many times. Our algorithm uses the access pattern of vector X and calculates the number of shared and distinct elements on this vector.

Five selected inputs of SpMV application from the University of Florida Sparse Matrix Collection [15] are considered for computational experiments. The properties and the shapes of those inputs are given in Table II and Figure 3, respectively.

TABLE II
APPLICATION INPUT MATRICES.

Input Name	# of rows	# of columns	# of nonzeros
Boddy4	17546	17546	69742
Boddy5	18589	18589	73935
Boddy6	19366	19366	77057
Illc1033	1033	320	4732
Ncvxqp9	16554	16554	31547

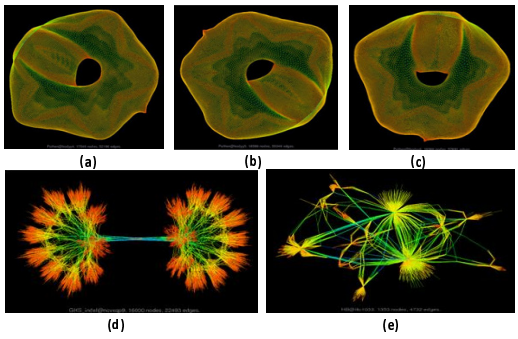

Fig. 3. The shapes of input matrices used by SpMV [15] (a) Boddy4, (b) Boddy5, (c) Boddy6, (d) Ncvxqp9, (e) Illc1033.

B. Performance of Dynamic Locality-aware Mapping

In our algorithm, the main thread obtains the input data and calculates similarities of the data elements accessed in different computations of the program. It groups chunks accessing similar data elements to bins and then begins computation as all other application threads do. Time spent on similarity calculation and bin assignment are not included in the following performance results. In all of the tests reported, the execution times are obtained from an average of five runs of the programs, the number of application threads is equal to the number of cores in the target architecture, and the number of chunks per bin are 2, 5, 10, 20, and 50.

The performance of our algorithm is compared with that of the standard Linux scheduler. The code running on standard Linux scheduler assigns consecutive blocks of chunks to bins, and bins are equally distributed to application threads. The standard Linux scheduler places threads in the least loaded processor and performs reactive and proactive dynamic load balancing. When a core becomes idle, a thread from a remote processor is migrated to the idle core in reactive load balancing; in addition, the processor time each thread uses is balanced in proactive load balancing. During thread scheduling and migration, data sharing is not considered by Linux scheduler [3].

The execution times of locality-aware mapping and Linux scheduler for 5 inputs are given in Figure 4. When the number of chunks per bin is equal to 2, data is divided into 32 parts and all chunks include large amount of program data. Even if 2 chunks have some sharing, the distinct data between them is higher so the similarity between chunks are close to each other. Therefore, we are not able to obtain high similarity between chunks. As a result, both our algorithm and the Linux scheduler have the same performance for all inputs used in this case.

Based on the characteristics of the input data used, one can observe different trends between performance and chunk per bin number. For inputs Boddy4 and Ncvxqp9, we observe the best performance when the chunk number is 50; whereas Boddy5, Boddy6, and Illc1033 inputs have the lowest execution time when 10 chunks are used. When chunk per bin number is changed, the data included in the chunks change, and the similarity between chunks is revised. For inputs Boddy4 and Ncvxqp9, we can obtain the highest similarity among chunks when 50 chunks are considered, but for other inputs 10 chunks are necessary to obtain the highest similarity. As a result, both the performance of our algorithm and Linux scheduler is affected by the varying number of chunks in use.

If we examine Figure 4, we see that both the Linux scheduler and our algorithm show similar performance trends for varying number of chunks. For all of the inputs considered, our algorithm outperforms the Linux scheduler, and this is due to increasing data reuse in the cache.

Figure 5 shows the speedup values of our algorithm in comparison to the Linux scheduler with respect to different number of chunks for different inputs. The speedup values obtained when Boddy4 input used vary between 3.3% and 6.6%, whereas the average speedup value is 5.6%. For Boddy5 input, the speedup values range between 6.8% and 9.9%, and the average speedup value is 8.3%. The minimum, maximum, and average speedup values for Boddy6 input are 1.4%, 7.9%, and 5.3%. The data elements of Ncvxqp9 and Illc1033 inputs have irregular access pattern and as a result, these inputs show the best performance with average speedup values of 18.3% and 24.5%, respectively. The minimum and maximum speedup values for input Ncvxqp9 are 16.4% and 22.4%, meanwhile the values when Illc1033 input is used are 19.8% and 26.8%. According to these values, our algorithm shows an average speedup of 12.44% for inputs with different characteristics.

IV. CONCLUSION

In this paper, we present a locality-aware dynamic mapping algorithm which targets to assign computations with similar data access patterns to same cores. Our algorithm divides

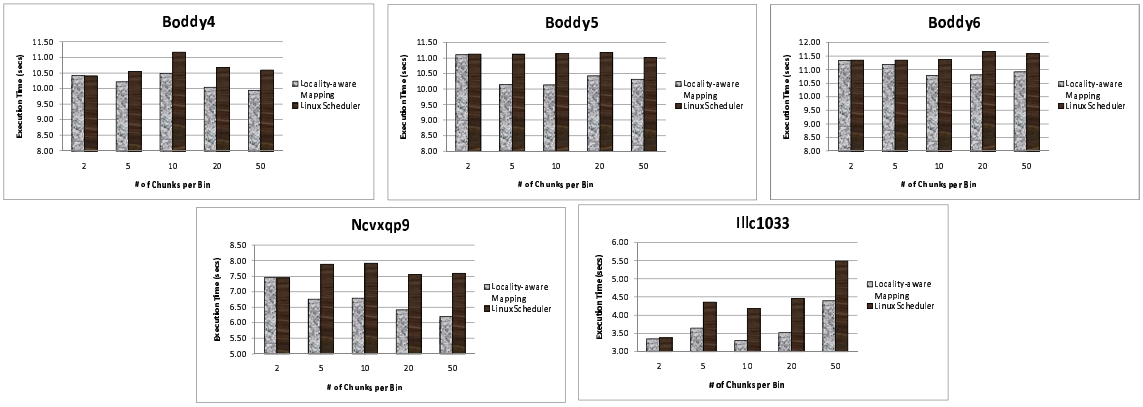

Fig. 4. Performance of Locality-aware dynamic mapping and Linux scheduler for various input files.

TABLE III
L1 AND L2 CACHE COUNTS (IN MILLIONS) WHEN USING THE MAXIMUM NUMBER OF CHUNKS.

Method	*Input*	L1 Cache			L2 Cache		
		Miss #	Access #	Hit %	Miss #	Access #	Hit %
Locality-aware Mapping	Boddy4	78.9	2431	96.8	67.0	486	86.2
	Boddy5	83.6	2574	96.8	132.4	514	74.3
	Boddy6	86.6	2679	96.8	111.4	487	79.2
	Ncvxqp9	63.5	1395	95.5	69.0	487	77.6
	Illc1033	50.9	1040	95.1	28.1	242	88.4
Linux Scheduler	Boddy4	66.5	2433	97.3	96.2	474	79.7
	Boddy5	70.5	2577	97.3	163.4	502	67.5
	Boddy6	73.9	2684	97.2	143.3	524	72.6
	Ncvxqp9	56.6	1404	96.0	98.2	304	67.7
	Illc1033	19.2	1288	98.5	46.8	257	81.8

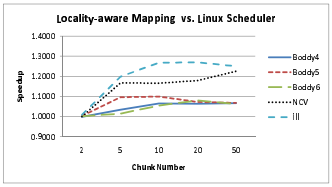

Fig. 5. Speedup values with respect to different number of chunks.

computations of a given application into chunks in order to provide load balancing; and a set of chunks are grouped into bins to provide data locality. The main goal behind this algorithm is to decrease cache contention by increasing data reuse on L2 cache. We test our algorithm by applying it to SpMV when considering five inputs with different shapes and characteristics. Our experimental evaluation indicates that our algorithm outperforms the standard Linux Scheduler on an average of 12.5% with respect to running time.

ACKNOWLEDGMENT

This research was supported by The Scientific and Technological Research Council of Turkey (TUBITAK) with a research grant (Project Number: 108E035).

REFERENCES

[1] J. H. Anderson et al., *Predictive Thread to Core Assignment on a Heterogeneous Multi-core Processor*, In Proc. SIGOPS, 2007.

[2] T. Sondag et al., *Phase-Guided Thread-to-core Assignment for Improved Utilization of Performance-Asymmetric Multi-Core Processors*, In Proc. Iwmse, pp.73-80, 2009.

[3] D. Tam et al., *Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors*, In Proc. ACM SIGOPS, pp. 47-58, 2007.

[4] S. Srikantaiah et al., *Adaptive Set Pinning: Managing Shared Caches in CMPs*, In Proc. ASPLOS, 2008.

[5] Y. Jiang et al., *Combining Locality Analysis with Online Proactive Job Co-Scheduling in Chip Multiprocessors*, In Proc. HIPEAC, 2010.

[6] J. Calandirino et al., *Cache-Aware Real-Time Scheduling on Multicore Platforms: Heuristics and a Case Study*, In Proc. ECRTS, pp. 299-308, 2008.

[7] Gaurav Khanna et al., *A Data Locality Aware Online Scheduling Approach for I/O-Intensive Jobs with File Sharing* In Proc. JSSPP, pp. 141-160, 2006.

[8] Virtutech Simics 4.0, http://www.virtutech.com.

[9] S. Haque et al., *Cache Friendly Sparse Matrix-vector Multiplication*, In Proc. PASCO, pp. 175-176, 2010.

[10] J. Demmel et al., *Self-adapting linear algebra algorithms and software*, In Proc. IEEE, 93(2):293-312, 2005.

[11] E. Im et al., *SPARSITY: Optimization framework for sparse matrix kernels*, In Proc. IJHPCA, 18(1):135-158, 2004.

[12] Sam Williams et al., *Optimizing sparse matrix-vector multiply on emerging multicore platforms*, Journal of Parallel Computing, 35(3):178-194, 2009.

[13] J. Choi et al., *Model-driven Autotuning of Sparse Matrix-Vector Multiply on GPUs*, In Proc. PPoPP, pp. 115-126, 2010.

[14] J. Bolz et al., *Sparse matrix solvers on the GPU: Conjugate Gradients and Multigrid*, In ACM Transactions on Graphics, Vol. 22, pp. 917-924, 2003.

[15] The University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices.