Locality Enhancement for Large-Scale Shared-Memory Multiprocessors

Tarek Abdelrahman[1], Naraig Manjikian[2], Gary Liu[3] and S. Tandri[3]

[1] Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada M5S 3G4
[2] Department of Electrical and Computer Engineering
Queens University
Kingston, Ontario, Canada K7L 3N6
[3] IBM Canada, Ltd
Toronto, Ontario, Canada M3C 1V7

Abstract. This paper gives an overview of locality enhancement techniques used by the *Jasmine* compiler, currently under development at the University of Toronto. These techniques enhance memory locality, cache locality across loop nests (inter-loop-nest cache locality) and cache locality within a loop nest (intra-loop-nest cache locality) in dense-matrix scientific applications. The compiler also exploits machine-specific features to further enhance locality. Experimental evaluation of these techniques on different multiprocessor platforms indicates that they are effective in improving overall performance of benchmarks; some of the techniques improve parallel execution time by up to 6 times.

1 Introduction

Large-scale Shared-memory Multiprocessors (LSMs) are being increasingly used as platforms for high-performance scientific computing. A typical LSM consists of processors, caches, physically-distributed memory and an interconnection network, as shown in Figure 1. The memory is physically distributed to achieve scalability to reasonably large numbers of processors. Nonetheless, the hardware supports a shared address space that allows a processor to transparently access memory through the network. Caches are used to mitigate the latency of accessing both local and remote memory. The hardware also enforces cache coherence. Examples of LSMs include the Stanford Flash [8], the University of Toronto NUMAchine [17], the HP/Convex Exemplar [4], and the SGI Origin 2000 [6].

Today's parallelizing compilers are capable of detecting loop-level parallelism in scientific applications, but often favor greater parallelism over locality, resulting in parallel code with poor performance [3, 5]. Hence, a key challenge facing the parallelizing compilers research community today is how to restructure code and data to effectively exploit locality of reference while maintaining parallelism. This is particularly the case for LSMs, where the physically-distributed shared memory and the use of high-speed caches dictate careful attention to memory and cache locality, yet the large number of processors requires a high degree of parallelism. To address this issue, we have investigated new code and data transformations that enhance cache and memory locality in scientific programs while preserving compiler-detected parallelism. We have prototyped these transformations in the *Jasmine* compiler at the University of Toronto. The goal of this

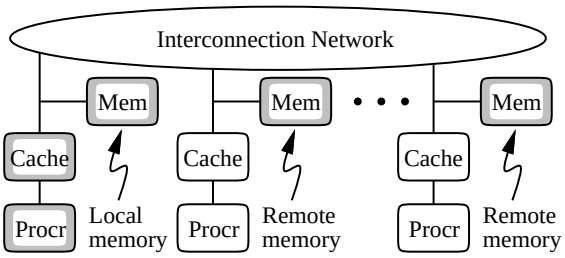


Fig. 1. A large-scale shared-memory multiprocessor.

paper is to give an overview of these transformations and to present recent experimental evaluation of their effectiveness in improving the parallel performance of applications on state-of-the-art LSMs.

There are 4 types of locality enhancement techniques employed by Jasmine. They are: memory locality enhancement (described in Section 2); inter-loop-nest locality enhancement (Section 3); intra-loop-nest locality enhancement (Section 4); and machine-specific locality enhancement (Section 5).

2 Memory Locality Enhancement

The placement of data in the physically-distributed shared memory of a LSM system has been traditionally delegated to the operating system. Page placement policies, such as "first-hit" and "round-robin" place pages in memory as these pages are initially accessed [9]. Unfortunately, operating system policies are oblivious to applications' data access patterns and manage data at too coarse of a granularity. It is too often the case that these policies fail to enhance memory locality, cause contention and hot-spots, and lead to poor performance [1, 9].

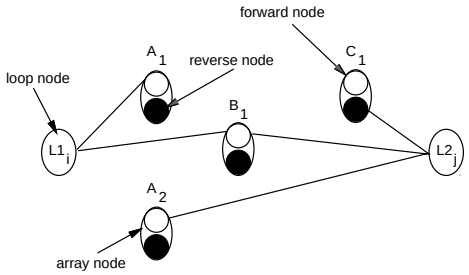
Data partitioning is an approach used by compilers for distributed memory multiprocessors, such as High-Performance Fortran (HPF), to map array data onto separate address spaces. Although such partitioning of an array is not necessary on a LSM because of the presence of a single coherent address space, data partitioning can be used to enhance memory locality—the compiler can place an array partition in the physical memory of the processor that uses this partition the most. Furthermore, data partitioning can eliminate false sharing, reduce memory contention and enhance cache locality across loop nests [15]. However, the task of selecting good data partitions requires the programmer to understand both the target machine architecture and data access patterns in the program [7]. Consequently, porting programs to different machines and tuning them for performance becomes a tedious and laborious process.

Hence, we designed and implemented an algorithm for automatically deriving computation and data partitions in dense-matrix scientific applications on LSMs [16]. The algorithm derives partitions taking into account not only the non-uniformity of memory accesses, but also shared memory effects, such as synchronization, memory contention and false sharing. It is used by Jasmine to place array data in the memory modules of a multiprocessor so as to enhance memory locality, reduce false-sharing, avoid contention and minimize the cost of synchronization. The algorithm consists of two main phases. In the first phase, *affinity* relationships are established between parallel loops and array data using array references in the program. These relationships are captured by the Array-Loop Affinity Graph, or the ALAG. Each node in this graph represents either

```
L1: for j = 1 to N−1
      forall i = j to N−1
        A(i,j) = (B(i)+B(i+1))*A(i,j+1)
      end for
    end for

L2: forall j = 1 to N
      for i = 2 to N
        A(i,j) = A(i−1,j) −B(j)+C(N−j+1,i)
      end for
    end for
```

(a) Example code.



(b) Associated ALAG.

Fig. 2. An example code and its ALAG.

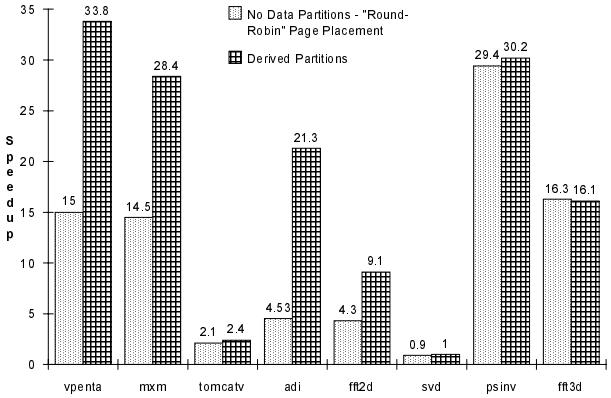


Fig. 3. Performance of benchmarks with and without CDP.

a parallel loop or an array dimension. Edges are introduced when the subscript expression in an array dimension contains the iterator of a parallel loop. Figure 2 shows a short program segment and its associated ALAG. The nodes in the ALAG are assigned initial distribution attributes and an iterative algorithm is used to derive *static* partitions. These partitions may not be optimal for some arrays that stand to be distributed differently in different loop nests. Hence, in the second phase of the algorithm, the static partitions for such arrays are re-evaluated to determine if *dynamic* partitions are more profitable. It is in this phase the machine-specific parameters and shared-memory effects are taken into account. The details of the algorithm are presented in [16].

The speedup (with respect to sequential execution) of 8 benchmarks applications on a 32-processor Convex SPP-1000 multiprocessor is shown in Figure 3. The parallel speedup of the applications using our derived computation and data partitions is superior to their speedup using operating system policies (about 2 times better on average for the applications). A more complete set of results on overall performance, on the importance of taking shared memory effects into account and on the computational efficiency of our approach is presented in [16].

3 Inter-Loop-Nest Locality Enhancement

Programs often contain sequences of parallel loop nests which reuse a common set of data. Hence, in addition to exploiting loop-level parallelism, a parallelizing compiler must restructure loops to effectively translate this reuse into cache locality. Existing techniques employ loop transformations which exploit data reuse within individual loop nests (e.g., tiling and loop permutations). Unfortunately, the benefit of these transformations is limited because there is often little reuse within a loop nest, or because today's large caches are capable of capturing what reuse exists [13]. In contrast, there exists considerable unexploited reuse *between* loop nests; the volume of data accessed in one loop nest is normally larger than the cache, causing data to be ejected from the cache before it can be reused in a subsequent loop nest.

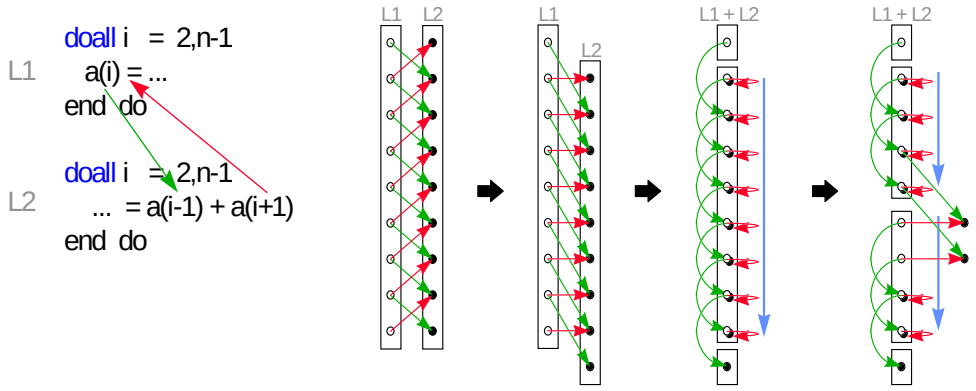
Loop fusion is used to combine a sequence of parallel loop nests into a single loop nest and enhance *inter-loop-nest locality*. In addition, fusion increases the granularity of parallelism when the resulting loop nest is parallel, and it also eliminates barrier synchronization between the original loop nests. However, data dependences between iterations from different loop nests may become loop-carried in the fused loop nest, and may make the resulting fusion illegal, or may prevent parallelization of the fused loop.

We present a transformation called *shift-and-peel* that enables the compiler to apply fusion and maintain parallelism despite the presence of dependences that would otherwise render fusion illegal or prevent parallelization [12]. The transformation consists of two operation, which respectively overcome fusion-preventing and serializing dependences. Figure 4 illustrates the application of *shifting* and *peeling* to a pair of one-dimensional parallel loops. Backward dependences (upward arrows) that prevent fusion are eliminated by alignment of iterations spaces. Cross-processor forward dependences (downward arrows) that prevent parallelization are eliminated by peeling their sink statements, as shown in the figure. Peeling enables the main blocks of fused iterations to be executed in parallel. The individual peeled statements are collected into smaller independent blocks that are executed after all main blocks have been executed. The details of the technique are in [12].

The speedup of 3 applications on the Convex SPP-1000 is shown in Figure 5 and it indicates that shift-and-peel improves parallel performance by 10%-30%. In one application, `spem`, there are 11 loop nest sequences which require our shift-and-peel transformation to enable legal fusion and subsequent parallelization. The longest sequence consisted of 10 loop nests, and most of the sequences contain 4 or more loop nests. The total data size for the application is 70 MBytes, which exceeds the total cache capacity of all processors and necessitates the use of inter-loop-nest locality enhancement.

4 Intra-Loop-Nest Locality Enhancement

Complex loop nest structures often consist of an outermost loop surrounding an inner sequence of loop nests. The shift-and-peel transformation enhances inter-loop-nest locality for the inner sequence of nest by fusing it into a single loop nest. This allows the reuse that may be carried by the outermost loop to further be exploited using *tiling*. Tiling re-orders and blocks iterations of the original loop into units of tiles to reduce the number of inner-loop iterations between uses



(a) Example loops. (b) Associated transformation.

Fig. 4. The shift-and-peel transformation.

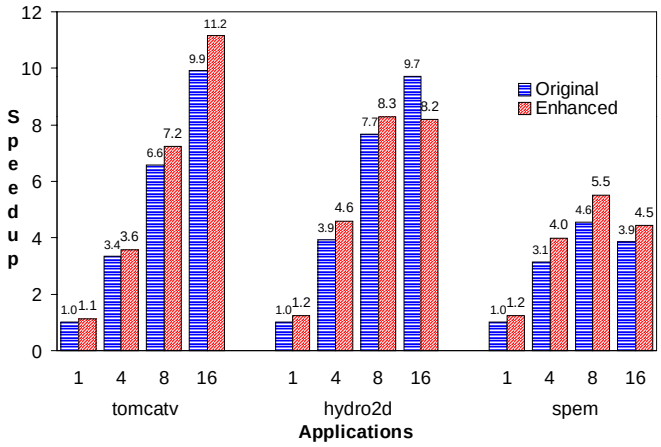
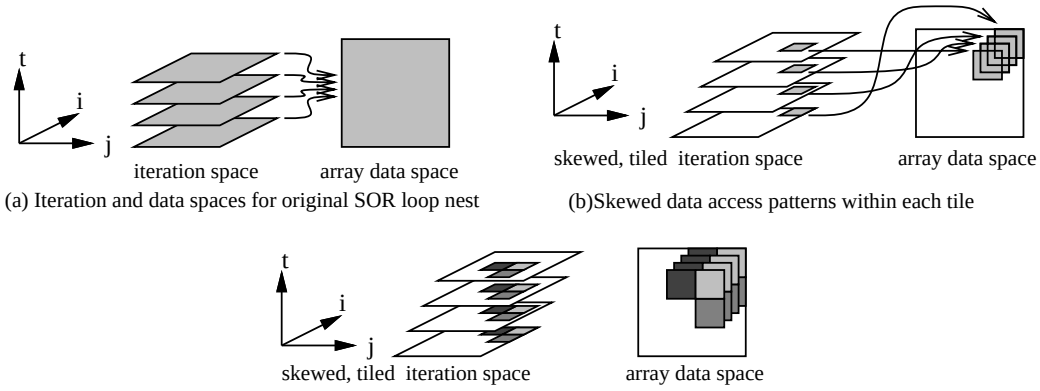


Fig. 5. The performance of applications with and without shift-and-peel.

of the same data. However, data dependences in a loop nest are often such that tiling violates the semantics of the original loop. It becomes necessary to use *loop skewing* to enable tiling, which introduces dependences that limit parallelism to *wavefronts* in the tiled iteration space. Skewing also modifies the data access patterns such that there is data reuse between adjacent tiles, as well as the data reuse that is captured within individual tiles, as shown in Figure 6 for the well-known `SOR` kernel.

Traditionally, dynamic scheduling has been used to execute wavefront parallelism [18]. However, dynamic scheduling does not exploit the reuse between adjacent tiles. We adapt block and cyclic static scheduling strategies in order to exploit this reuse in tiled loop nests on LSMs [11]. Our strategies also enable the use of small tiles to increase the degree of parallelism on each wavefront. The strategies are described in [11].

The use of our block scheduling strategy for the `Jacobi` PDE solver kernel on



(a) Iteration and data spaces for original SOR loop nest

(b)Skewed data access patterns within each tile

(c) Overlapping array regions swept by adjacent tiles

Fig. 6. Data access patterns that result after skewing and tiling of `SOR`.

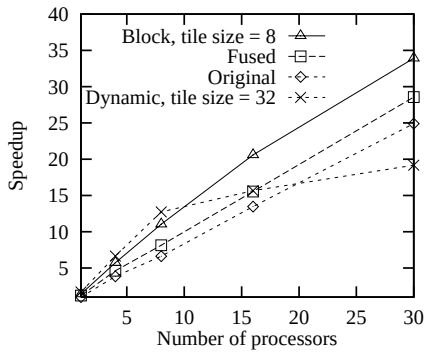


Fig. 7. The performance of 2048 × 2048 `Jacobi` with and without our wavefront scheduling.

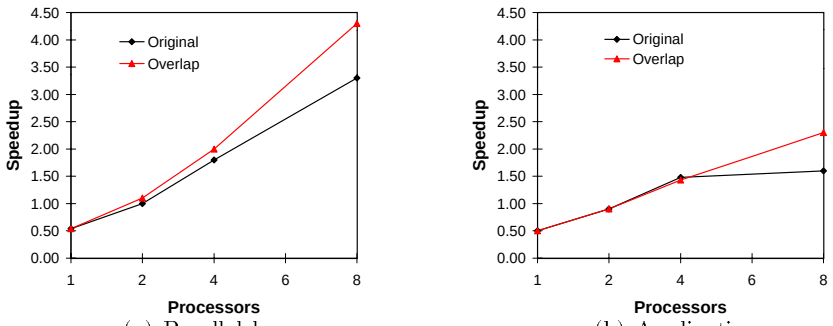
a 30-processor Convex SPP-1000 demonstrates its advantage over the dynamic strategy. The kernel consists of two inner loop nests and an outer loop which carries reuse. The shift-and-peel transformation is required in order to fuse the two inner loop nests. Loop skewing is required prior to tiling, which gives rise to wavefront parallelism. Figure 7 shows the speedup of the original `Jacobi`, the speedup after fusion but without tiling, the speedup after tiling with dynamic scheduling and the speedup after tiling with our adapted block scheduling. The best performance for a large number of processors is obtained by fusing and by using our adapted block scheduling strategy. Although not shown in the graph (for the sake of readability), dynamic scheduling with a small tile size performs far worse.

5 Machine-Specific Locality Enhancement

Several research and commercial LSMs incorporate architectural features that offer additional opportunities for locality enhancement. For example, the NUMAchine multiprocessor [17] supports block transfers from memory to a third level cache which operates at main memory speed, called the Network Cache (NC). Similarly, the POW network-of-workstation multiprocessor [14] supports

coherent memory-to-memory prefetching in a single address space. The Jasmine compiler exploits these features to reduce the latency of remote memory accesses in parallel loops. This is done by transferring to the local memory of a processor (i.e., the NC associated with a processor in NUMAchine, or the local workstation memory in POW) the remote data which is accessed by iterations assigned to the processor [10]. The data partitions automatically derived by our algorithm (or otherwise specified by the programmer) are utilized to determine local and remote data for each processor. In addition, computation partitions and array references are used to determine iterations of a parallel loop that access only local data (referred to as *local-computation*) and iterations that also access remote data. The iterations that access remote data are removed from the parallel loop and are inserted after the local-computation. An asynchronous transfer of the remote data is initiated before the local-computation to overlap remote access latency with the local-computations and make remote data local.

The speedup of `Jacobi` on POW (8 RS6000 workstations connected by Fast Ethernet and using the TreadMarks [2] software) is shown in Figure 8 with and without computation-communication overlap. The performance of the first parallel loop in `Jacobi` is shown in Figure 8(a) and overall performance is shown in Figure 8(b). The overlap transformation benefits both parallel loop performance and overall performance by about 30%.



(a) Parallel loop. (b) Application.

Fig. 8. Speedup of 512 × 512 `Jacobi` with 640 iterations.

6 Concluding Remarks

Compiler techniques that enhance locality are critical for obtaining high performance levels on LSMs. This paper gave an overview of code and data transformations used by the Jasmine compiler to enhance memory and cache locality. These techniques address not only the relevant architectural issues of LSMs (physically-distributed memory, cache, contention, etc.) but also characteristics of real applications (sequences of loop nest and complex dependences). Experimental evaluation of these transformations on state-of-the-art multiprocessors demonstrates their effectiveness in improving overall performance of applications.

References

1. T. Abdelrahman and T. Wong. Compiler support for array distribution on NUMA shared memory multiprocessors. *The Journal of Supercomputing*, to appear, 1998.
2. C. Amza, A. Cox, et al. Treadmarks: shared memory computing on networks of workstations. *IEEE Computer*, 29(12):78–82, 1996.
3. W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, 1996.
4. Convex Computer Corporation. *Convex Exemplar System Overview*. Richardson, TX, USA, 1994.
5. M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, 1996.
6. Silicon Graphics Inc. *The SGI Origin 20000*. Mountain View, CA, 1996.
7. K. Kennedy and U. Kremer. Automatic data layout for High Performance Fortran. In *Proc. of Supercomputing*, pages 2090–2114, 1995.
8. J. Kuskin, D. Ofelt, M. Heinrich, et al. The Stanford FLASH multiprocessor. In *Proc. of the 21st Annual Int'l Symposium on Computer Architecture*, pages 302–313, 1994.
9. R. LaRowe Jr., J. Wilkes, and C. Ellis. Exploiting operating system support for dynamic page placement on a NUMA shared memory multiprocessor. In *Proc. of the 3rd ACM Symposium on Principles and Practice of Parallel Programming*, pages 122–132, 1991.
10. G. Liu and T. Abdelrahman. Computation-communication overlap on network-of-workstation multiprocessors. In *Proc. of the Int'l Conference on Parallel and Distributed Processing Techniques and Applications*, to appear, 1998.
11. N. Manjikian and T. Abdelrahman. Scheduling of wavefront parallelism on scalable shared memory multiprocessors. In *Proc. of the Int'l Conference on Parallel Processing*, pages III–122–III–131, 1996.
12. N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Trans. on Parallel and Distributed Systems*, 8(2):193–209, 1997.
13. K. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proc. of the 7th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 94–104, 1996.
14. The POW multiprocessor project. University of Toronto. *http://www.eecg.toronto.edu/parallel/sigpow*, 1995.
15. S. Tandri and T. Abdelrahman. Computation and data partitioning on scalable shared memory multiprocessors. In *Proc. of the Int'l Conference on Parallel and Distributed Processing Techniques and Applications*, pages 41–50, 1995.
16. S. Tandri and T. Abdelrahman. Automatic partitioning of data and computation on scalable shared memory multiprocessors. In *Proc. of the Int'l Conference on Parallel Processing*, pages 64–73, 1997.
17. Z. Vranesic, S. Brown, et al. The NUMAchine multiprocessor. Technical Report CSRI-324, Computer Systems Research Institute, University of Toronto, 1995.
18. M. Wolf. *Improving locality and parallelism in nested loops*. PhD thesis, Department of Computer Science, Stanford University, 1992.

This article was processed using the LaTeX macro package with LLNCS style