

Optimizing Data Parallel Operations on Many-Core Platforms

Byoungro So, Anwar M. Ghuloum, and Youfeng Wu
Intel Corporation
Programming Systems Lab
2200 Mission College Blvd.
Santa Clara, CA 95054
{byoungro.so, anwar.ghuloum, youfeng.wu}@intel.com

Abstract

Data parallel operations are widely used in game, multimedia, physics and data-intensive and scientific applications. Unlike control parallelism, data parallelism comes from simultaneous operations across large sets of collection-oriented data such as vectors and matrices. A simple implementation can use OpenMP directives to execute operations on multiple data concurrently. However, this implementation introduces a lot of barriers across data parallel operations and even within a single data parallel operation to synchronize the concurrent threads. This synchronization cost may overwhelm the benefit of data parallelism. Moreover, barriers prohibit many optimization opportunities among parallel regions.

In this paper, we describe an approach to optimizing data parallel operations on many-core platforms, called sub-primitive fusion, which reduces expensive barriers by merging code regions of data parallel operations based on the data flow information. It also replaces remaining barriers with light-weight synchronization mechanisms. This approach enables other optimization opportunities such as data reuse across data parallel operations, dynamic partitioning of fused data parallel operations, and semi-asynchronous parallel execution among the threads.

We present preliminary experimental results for the sparse matrix kernels that demonstrate the benefits of this approach. We observe speedups up to 5 on an 8-way SMP machine compared against the serial execution time.

1. Introduction

Data parallelism, unlike control parallelism, comes from simultaneous operations across large sets of collection-oriented data such as arrays, vectors, matrices, sets, and trees. Data parallel operations include element-wise, prefix

sum, reduction, permutation, sparse matrix vector multiplication, etc. Careful inspection of these data parallel operations shows that they can be decomposed into a small number of phases, called *sub-primitives*; each performing local computation independently, or collecting the local computation result into a global data structure, or updating the local data structure from the aggregated global data, as illustrated in Figure 1. Decomposing into sub-primitives and factoring out the common phases enable modular compiler optimizations and allow users to compose their own data parallel operations from them.

During the global collection phase, threads exchange their local data with other threads, and introduce a lot of thread-to-thread communication and synchronization. Typically, a barrier is used to enforce an ordering among these phases inside a single data parallel operation as well as among different data parallel operations. However, barrier synchronization is too expensive in many cases where only a subset of threads need be synchronized at a specific time.

We have developed a highly optimized *Fusion* data parallel library on top of user-level fine-grained synchronization primitives. Basically, it implements language-independent data parallel primitives that are widely used in game, multimedia, physics, geometry, and HPC.

Many-core processors are well suited for fine-grained data parallelism because the core-to-core communication within the same silicon chip can be implemented more ef-

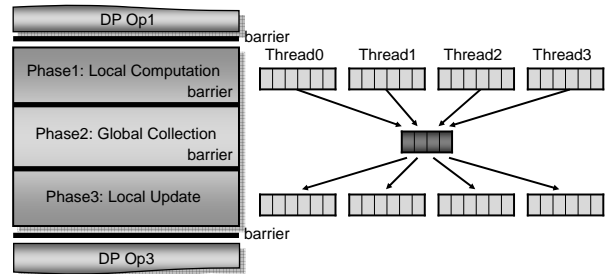


Figure 1. Sub-primitive decomposition

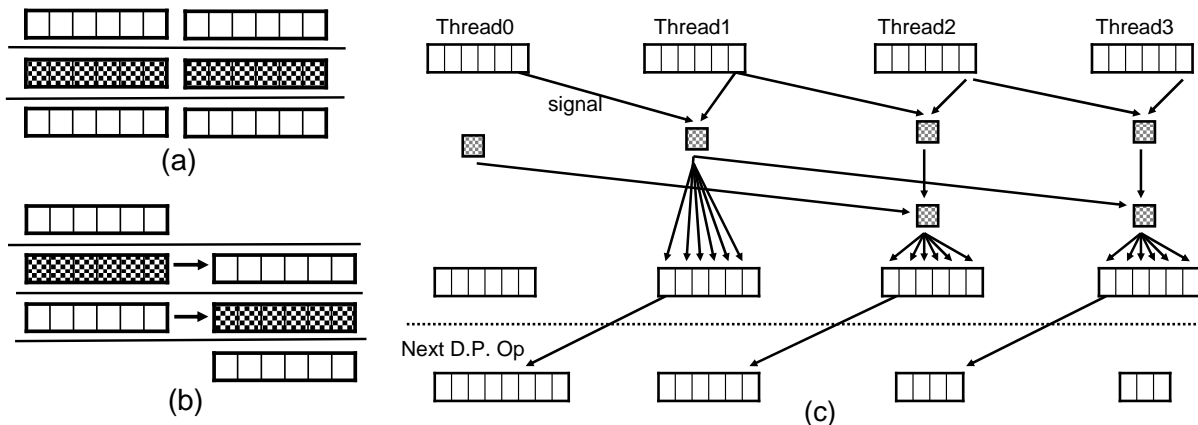


Figure 2. Sub-primitive decomposition

ficiently than on multiprocessor platforms. To exploit this benefit, we have developed several parallelization, data locality, data layout optimizations. Details are beyond the scope of this paper. In this paper, we introduce one of the key optimizations called *sub-primitive fusion*. It fuses parallel phases of data parallel operations both inside a single data parallel operation and across multiple operations by analyzing precise data flow information and synchronization requirements among threads at the data parallel operation level. We are building support for Fusion data parallel operations in a version of Intels production C/C++ compiler as a research vehicle for this work. The research compiler recognizes all Fusion data parallel operations as intrinsics, rather than treating them as function calls. The compiler can then lower each intrinsic into appropriate intermediate representation(s) for optimization.

We show our preliminary results from the optimization study on several multimedia kernels such as sparse matrix vector multiplication. Our experimental results show that the proposed approach can achieve speedups up to 5 on an 8-way SMP machine compared against the serial execution time.

The rest of the paper is organized as follows. Section 2 describes our new sub-primitive fusion optimizations. Section 3 presents the preliminary experimental results. Section 4 discusses some related work, and Section 5 concludes the paper and points out future directions.

2. Optimizations

A data parallel operation can be decomposed into a small number of *sub-primitives* which perform a common phase of many different data parallel operations. Different data parallel operations require a different set of sub-primitives. This decomposition of data parallel operations allows sev-

eral modular optimizations at a reasonable cost. In the following subsections, we briefly describe three of them.

2.1. Sub-Primitive Fusion

We have developed *sub-primitive fusion* which collects several sub-primitives into a fused phase, and executes them in serial or parallel, but eliminates redundant barriers between them. It is distinguished from loop fusion or code motion in that these traditional approaches will not work because of the barriers between them. Figure 2 compares three ways of fusing sub-primitives. *Static fusion* performs a horizontal fusion of sub-primitives. By analyzing the data dependence among sub-primitives, it can merge them if there is no data dependence or only intra-phase dependence exists. Some sub-primitives share some common functionality, and one can be replaced with a more expensive operation as long as it can be fused with other sub-primitives. For example, a reduction sum can be replaced with a prefix sum sub-primitive. Figure 2(a) shows an example of static fusion when there is no dependence among sub-primitives. Figure 2(b) shows another example with intra-phase dependence. If there is data dependence among sub-primitives, it delays the consumer sub-primitives until the producer primitives are finished. However, still different phases of different operations can execute in parallel.

Dynamic fusion performs a vertical fusion of sub-primitives. It essentially eliminates the barriers and orders sub-primitives through finer-grain synchronization mechanisms, such as synchronization variables or event signals. This scheme can be applicable to the cases where not all the threads need be involved in synchronization at a specific time. Figure 2(c) illustrates a fused prefix sum operation. In the first phase, each processor performs the prefix sum on its local data. In the second phase, each thread updates its local prefix sum result into a global data structure.

Each thread is waiting for a signal from a frontier thread with an increasing distance from itself, as illustrated in Figure 2(c). Thus, the expensive barrier can be replaced with a point-to-point synchronization. This light-weight synchronization can be one-to-one, one-to-many, or many-to-one depending on the sub-primitive type. In addition to reduced synchronization overhead, it also allows threads to work semi-asynchronously. Unlike barrier-based parallel execution model, each thread does not have to wait for irrelevant thread at a specific synchronization point, and can proceed autonomously. One of the future works is to vectorize the fused sub-primitives to fully exploit the capabilities of the multi-core system.

A potential overhead of sub-primitive fusion is increased register pressure and memory bandwidth requirement due to the increased amount of parallelism. Additionally, the overhead of dynamic fusion includes increased memory usage for fine-grain synchronization variables. But, this increase is only proportional to the number of processing cores, not to the problem size.

2.2. Semi-Asynchronous Execution Model

We have developed a work queue model of parallel threads. The traditional fork-join style of parallel execution model is too expensive in many cases, especially if the data parallel operation is executed repeatedly in a loop for a relatively small data set. The main idea is to place work queues between the worker threads and the master thread, and have worker threads keep running in an infinite loop until the end of the entire program execution. The master thread executes the serial portion of the program and puts data parallel work items in queues. A work item is a combination of fused sub-primitives, either from a single data parallel operation or across fused operations. Each worker thread actually executes only a single task during the entire execution of the program, which goes into an infinite loop and waits for a "work-to-do" event from the master thread. When the event is notified, each thread picks up a work item from the queue, and performs the cooperative data parallel operation assigned to itself. Once it is done, it then waits for the next event again. This parallel execution model is still a kind of SPMD model, but it has several advantages. First, it eliminates the overhead of repeated creation/termination of threads and data parallel task. Secondly, this model allows each thread to execute the given sub-primitives at its own pace, minimizing the idle time of faster threads. In result, we get better overall load balancing; some worker threads may complete their work items faster than the others and it can essentially lead to the situation where different threads work on different data parallel operations simultaneously.

2.3. Dynamic Work Partitioning

We have been designing a dynamic partitioning algorithm for overall load balance among the worker threads. A traditional way of partitioning a parallel work is to distribute work load across the threads as evenly as possible. In general, this approach works well if we don't know how each worker threads cooperate at runtime. However, due to the semi-asynchronous characteristics of our parallel execution model as well as the collection-oriented nature of data parallel algorithms, each thread in reality finishes its work load at quite different times. To make up for this load imbalance, the dynamic partitioning algorithm employs separate work queues for each thread, and partitions the work load unevenly; i.e., it assigns more work load to faster threads and less work for slower ones. For example, consider the unbalanced algorithm given in Figure 2(c). Thread0 finishes its work share first, and Thread1 second, and Thread2 and Thread3 last. Our runtime system knows exactly how each sub-primitive is partitioned and scheduled at different stages across work threads. Thus, it assigns more work load to Thread0 than to Thread1, and subsequently much more than Thread2 and Thread3. Another future work is work load migration, which moves a work load assigned to slower threads to faster threads at run time.

3. Experimental results

We have evaluated our sub-primitive fusion for a selected collection of data parallel operations written in C on an 8-way SMP machine. This includes prefixsum [4], reduction, multi-reduce, multi-prefixsum, scatter, gather, and four different types of sparse matrix vector multiplications (csr, csc, csr symmetric, csc symmetric).

When we perform the experiments using our data parallel library, a minor change to the source code is done manually to control the data parallel library. Our data parallel library by default does not insert a barrier at the end of the data parallel operation, but sometimes barrier is more convenient when all the threads have to synchronize at some points. An extern variable is used to indicate whether all the threads have to synchronize at a barrier. Another extern variable is to inform the worker threads to terminate in the infinite loop described in Section 2.2. This manual change will be done by the compiler in the future by analyzing the data flow information at each data parallel operation. The problem size is 10 million vector elements of double precision type for sparse matrix vector multiplications, while integer type vector elements were used for the other operations. We used Intel C compiler version 9 at the default optimization level and vectorization was not enabled in this experiment.

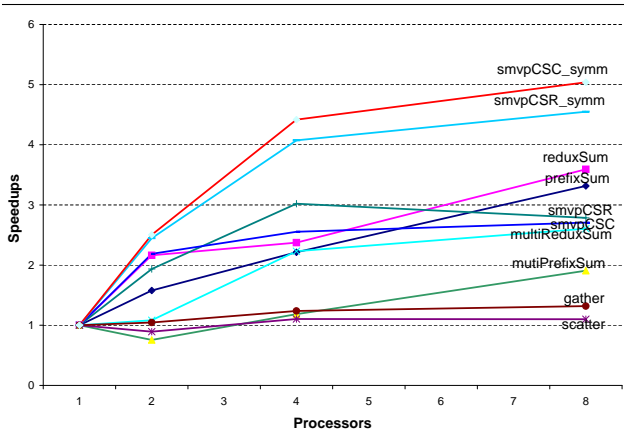


Figure 3. Speedups over serial

3.1. Results

To show the scalability of our approach, we performed experiment on 1, 2, 4, and 8 processors as shown in Figure 3. The X-axis represents a different number of processors used, and the Y-axis the speedup number normalized to the serial execution time.

To parallelize scatter and gather operations, the current implementation requires expensive setup before actual parallel operations. Although, we are currently in the process of reducing these extra overhead, we expect overall performance impact is small when we fuse multiple data parallel operations because it should amortize the extra overhead by combining multiple operations and synchronization.

Prefixsum, reduction, multi-prefixsum, and multi-reduction operations scales reasonably well.

Sparse matrix vector multiplication is a combination of several data parallel sub-primitives, which are fused both statically and dynamically. Their synergistic effects lead to a superlinear speedup on 2 processors for smvpCSC and reduction sum operations and on 4 processors for symmetric versions of sparse matrix vector multiplications. A part of this speedup attributes to the flattening of vector and loop structure which makes the loop more prefetcher friendly by transforming the accesses more regular.

Overall, this set of results supports our claim that sub-primitive fusion scales well on SMP machines, and we expect better performance on many-core platforms.

4. Related Work

C Vector Library (CVL) [2] at CMU is a library of low-level vector operations such as element-wise, scan, reduction, and permutation. It provides a vector abstraction that is used as a backend of nested data parallel language NESL [1]. It supports segmented vectors which is useful

to represent nested data parallelism [6]. Our data parallel library is motivated by this library, but supports more varied data parallel operations and automatic scratch space management in the library.

Chakravarty et. al introduced NEPAL [3], a nested data parallel extension to Haskell functional language, which attempts to maximize both flexibility of control parallelism and static knowledge of data parallelism by combining the flattening of nested parallelism and calculations fusion techniques. They implement similar vector write-once semantics as we do because Haskell is a pure functional language. Moreover, they perform similar optimizations such as flattening segmented operations, unfolding library routines into smaller operations, and merging them using calculational fusion. What distinguishes our work from theirs is that we do not require looking into the parallel function definition (e.g. for user defined functions) to successfully fuse. However, the approaches are very similar. We believe that decomposing the primitives into sub-primitives expose deeper fusion and other optimization opportunities. Finally, we are designing a language independent library that will work in non-functional contexts, but still allows inter-primitive optimizations.

Streaming languages, such as StreamIt [7] and Brook [5], provide support for data parallel computation and a more explicit notion of dataflow dependences between data parallel primitives. However, we have generally found that these languages are too restrictive to support irregular applications such as, say, sparse matrix based algorithms. Moreover, performing simple, flat reductions can be awkward and incongruous with respect to the base programming models. However, it is worth noting that the many architectures researched for streaming and data flow languages are reasonable targets for a more general form of data parallelism.

5. Conclusion

This paper introduced a novel data parallel optimization technique *sub-primitive* fusion. Three different schemes are presented in this paper. The preliminary experiments on selective data parallel operations and sparse matrix vector multiplication reveal that a combination of all three schemes can significantly improve the performance of data parallel operations.

We are currently working on vectorizing fused sub-primitives to take better advantage of the architecture. We expect the combining effect on the performance would be significant for multimedia, game, and physics applications.

6. Acknowledgement

We would like to thank Mauricio Breternitz, Jr, Tin-fook Ngai, Jason H. Lin, Yongjian Chen, Leaf Petersen, James Stichnoth, and Timothy Mattson for their invaluable advice and effort on how to improve this research.

References

- [1] G. E. Blelloch. Nesl: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, Carnegie Mellon University, 1993.
- [2] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, M. Reid-Miller, J. Sipelstein, and M. Zagha. Cvl: A c vector library. Technical Report CMU-CS-93-114, Carnegie Mellon University, 1993.
- [3] M. M. T. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel. Nepal — nested data parallelism in Haskell. *Lecture Notes in Computer Science*, 2150:524+, 2001.
- [4] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Supercomputing*, pages 666–675, 1990.
- [5] M. Erez, J. H. Ahn, N. Jayasena, T. J. Knight, A. Das, F. Labonte, J. Gummaraju, W. J. Dally, P. Hanrahan, and M. Rosenblum. Merrimac: Supercomputing with streams. In *Proceedings of the 2004 SIGGRAPH GP² Workshop on General Purpose Computing on Graphics Processors*, 2004.
- [6] G. W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. The MIT Press, Cambridge, Massachusetts, 1989.
- [7] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Computational Complexity*, pages 179–196, 2002.