An empirical study of the performance of scalable locality on a distributed shared memory system

Tim Douglas
Haverford College

May 6, 2011

Abstract

We run an empirical study to determine the performance attainable using scalable locality on a distributed shared memory system. We utilize the PLUTO automatic parallelizer and locality optimizer on a one-dimensional Jacobi stencil with a distributed shared memory system provided by Intel's Cluster OpenMP on commodity hardware. Unlike earlier works which base performance concerns on tile size, we discover tile sizes are less crucial, and instead threading issues and cache interference need particular attention. On our 56 processor cluster of 14 quad-core machines, with preliminary single-level tiling we are able to realize 121.3 GFLOPS, or 63% of linear speedup. Our data suggest that adding further cores would increase performance linearly.

1 Introduction

As the presence of multi-core architectures has increased in the past handful of years, a serious amount of research has been put into parallelization, and automatic methods of doing so. This has been met a varied amount of success; there still does not exist a silver bullet that doubles application performance when given double the number of cores, but in specific instances automatic parallelization has been successful.

The PLUTO automatic parallelizer, in particular, has shown impressive results in modifying C source for parallelism in dense matrix computations [BBK+08]. Even though these kinds of computations are infrequently used in general purpose computing, the performance gains in the scientific computing realm are notable enough to drive continued research.

Intel and AMD's efforts for higher performance has greatly increased the wide-scale deployment of dual- and multi-core processors. Yet from a

theoretical standpoint, the dual-core processors found in today's commodity PCs are not different from the clusters of computers connected by various grades of interconnects that have been used in high-performance computing for many years. A four-core machine theoretically has the same compute capability as four single-core machines connected together by, e.g., Fast Ethernet.

Yet in practice, the multi-core machine will outperform the cluster in nearly every benchmark. This occurs because the connections between the processors in the shared memory system have much higher bandwidth and much lower latency than the cluster, even if the cores themselves have the same peak performance. Except for a certain class of algorithms, known as "embarrassingly parallel," each core needs information calculated by other cores for further computation. Hence, the interconnect performance affects total performance.

When used properly, the PLUTO parallelizer, in addition to turning single-threaded algorithms into multi-threaded ones, can also add a characteristic called "scalable locality." [Won02]. By modifying an algorithm's order of execution, one can greatly decrease the amount of inter-core communication necessary. When the bottleneck becomes the CPU instead of the interconnect, one should be able to treat a cluster of computers as if were just one shared memory system, achieving comparable performance.

1.1 Motivation

Shared memory systems with many (e.g., 64) cores exist and are able to run appropriately localized and parallelized codes quickly, but they are typically orders of magnitude more expensive than commodity hardware connected by commodity interconnects. The following table compares the cheapest 64-node x86 shared memory system sold by Oracle (formerly Sun) and the cluster we have created in Haverford's teaching laboratory.

Machine	Sun Fire X4800	Haverford "Springfield"
Processor	8 x 8-Core Intel Xeon X7560, 2.26 GHz	16 x 4-Core Intel Core i7 860, 2.80 GHz
Memory	512 GB (64 x 8 GB DDR3)	64 GB (32 x 2 GB DDR3)
Storage	2.4 TB (8 x 300 GB)	5.1 TB (16 x 320 GB)
Price	$114,196.00	≈$12,000

Granted, this comparison is rather contrived; one can surely expect greater stability with the X4800's server-grade components versus our consumer-

grade RAM and disks. After all, because of a hardware failure on one of our machines we were not able to get benchmarks using all 64 cores.

The algorithmic example we use is not entirely fabricated; the three-point one-dimensional stencil code we currently consider is a model similar to those frequently used in the scientific computing realm. Increased performance on these classes of algorithms is a clear benefit of this research.

As useful as PLUTO is with this stencil, it is currently rather limited with regard to the C subset it is able to analyze and modify. For example, in our testing it fails to process the TOMCATV mesh generation benchmark, a part of the SPEC92 floating point benchmark suite. Nevertheless, PLUTO's current capabilities do not represent the theoretical reach of the polyhedral model it uses. Increasingly sophisticated implementation are able to analyze more general programs.

2 Tiling and locality background

PLUTO achieves parallelization and scalable locality by a method of modifying loops called tiling. The manner in which loop tiling increases locality and allows parallelism is well-documented in a number of other publications, e.g., [Won00] and [DW09]. Here we give the reader a much briefer introduction.

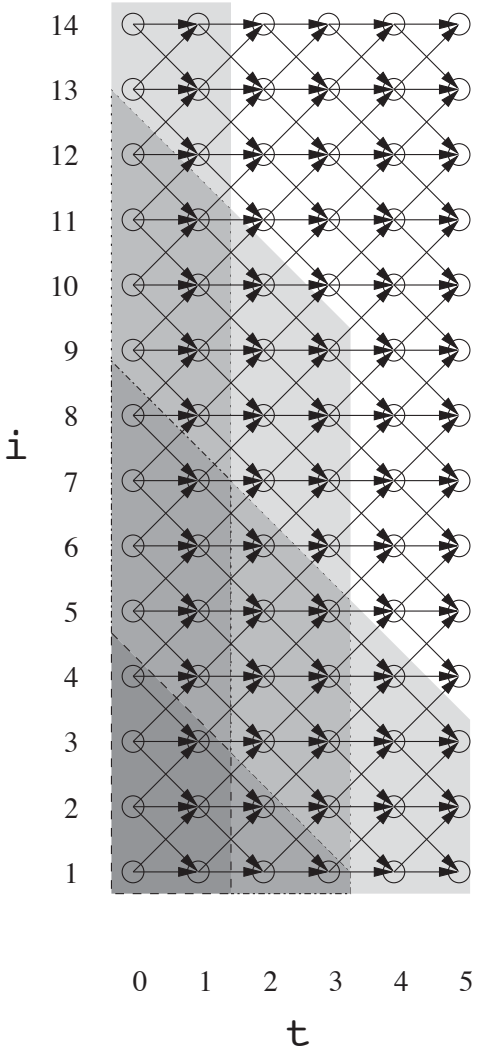
Consider the following C loop:

```
for (t = 0; t < T; t++) {
    for (i = 1; i < N-1; i++) {
        new[i] = (A[i-1] + A[i] + A[i+1]) * (1.0/3);
    }
    for (i = 1; i < N-1; i++) {
        A[i] = new[i];
    }
}
```

This code, known as an imperfectly nested Jacobi stencil [BHRS08], models, e.g., heat distribution on a wire. For every time step `t`, all locations on the wire `i` become the average of themselves and their neighbors in the previous time step.

This algorithm performs poorly because the manner in which it uses memory prevents efficient cache usage. Calculating each timestamp requires iterating over the entire array, so if the array does not fit into cache it continually needs to be read from RAM. This causes severe performance

Figure 1: Tiling a 1D Jacobi stencil



degradation, because RAM is at least an order of magnitude slower than cache [Dre07].

Consider Figure 1 (taken from [DW09]). In this diagram, each element of the array is represented by a circle. The "wire" consists of a column of circles, and the time steps are represented by the horizontal collection of columns. The individually shaded sections are "tiles" as constructed by PLUTO. Each tile encompasses a number of calculations (one set for

every circle), and shows data dependencies for that calculation (indicated by the arrows). Instead of doing the calculation bottom up, column by column, each tile can be done consecutively, compartmentalizing the data dependencies and increasing use of cache.

Each tile can also be independently computed; the only data transfers necessary are those on the edges of the tiles. In order to take advantage of this increased independence, PLUTO adds information for the compiler to parallelize it using the OpenMP framework. Information about OpenMP and the manner in which we run OpenMP programs on our cluster via Cluster OpenMP appears later in this paper.

Let us call the tile height σ (i.e., size of the tile in the space dimension), and the tile width τ (i.e., size of the tile in the time dimension). The border perimeter of these tiles grows as $\mathcal{O}(\sigma + \tau)$, or linearly with respect to their width and height. Unsurprisingly, tile volume grows quadratically with respect to the same values ($\mathcal{O}(\sigma\tau)$). Hence, when one makes the tiles large, volume (computation) becomes large compared to perimeter (communication); one can scale up tile size to increase the computation-to-communication ratio. For a system with a high computational ability and comparatively poor communication, a large ratio is necessary to achieve high performance.

3 Related work

The benefits of the polyhedral model and tiling are well known, and freely available tools (e.g., PLUTO) exist to take advantage of such tiling techniques [KBB+07] [BBK+08] [BHRS08]. PLUTO performs tiling and requires tile size to be set by a compile-time parameter; in addition, it contains no heuristics to automatically set optimal values.

Tile size selection algorithms are not an unstudied topic. Yet most published algorithms primarily concern cache characteristics on a single threaded machine, such as [CK95], [hHK04], and [FCA05].

OpenMP for distributed shared memory systems is also well researched. [EHK+02], [HCK03], and [HCLK04] all discuss manners of running OpenMP on a DSM cluster, but none provide references to downloadable source or usable tools. A few of the researchers who have combined OpenMP and DSM empirically discuss performance, but note that "full applications need careful consideration of shared data access patterns" [SJBE03] and fail to mention tiling as a viable solution.

The combination we discuss is designed to test the approach of [Won02], which deliberately defines and discusses scalable locality, and [Won00], which

considers a "generalization for multiprocessors [that] lets us eliminate processor idle time caused by any combination of inadequate main memory bandwidth, limited network bandwidth, and high network latency, given a sufficiently large problem and sufficient cache." The paper discusses multiprocessing and takes into account memory bandwidth and latency, but does not provide benchmarks or empirical evidence to support its thesis.

[DW09] gives empirical results and is a direct precursor to this paper. Yet because of hardware availability at that time, it only lists benchmarks for eight cores, while we are able to get to 56.

4 Experimental setup

4.1 Hardware

For our experiments we began with 16 custom-built x86-64 machines. The shared nature of the lab required one machine, the main lab server, to be dedicated to other students' work, reducing our the number of available machines by one. Similarly, only four could be used full-time for experiments; use of the other computers required late-night coordination. A hardware failure in another one of the machines left us 14 nodes with which we could conduct research.

Each of the machines is powered by an Intel Core i7 860 microprocessor running at 2.8 GHz, with 4 GB of DDR3 memory. The processors are capable of hyper-threading, i.e., running multiple threads on one logical core, but that was disabled for our benchmarks. The cache layout is as follows (L1 & L2 are per-core, and L3 is shared between all cores):

Level	Size	Associativity	Line Size
L1 data	4 x 32 KB	8-way set associative	64 byte
L1 instruction	4 x 32 KB	4-way set associative	64 byte
L2	4 x 256 KB	8-way set associative	64 byte
L3	8 MB	16-way set associative	64 byte

The machines are connected via a 10/100 Ethernet switch administered by Haverford College.

4.2 Software

4.2.1 PLUTO

The benchmark we use begins as a simple C loop. In order to tile and transform the stencil, we utilize the PLUTO compiler [BBK+08], which takes these loops (properly delimited) and desired tile sizes, and outputs appropriately tiled C code, with OpenMP tags if parallelization is requested.

The automaticity of PLUTO's execution makes it notable and a good fit for our interests; our research is beneficial not only because we want to show clusters can be efficiently utilized, but that such can be done without much additional effort. Nevertheless, PLUTO does require some configuration (i.e., tile size), and determining what is optimal in that regard is of interest to us.

4.2.2 Cluster OpenMP

Standard OpenMP programs are only executable on traditional shared memory systems, consisting of all of processors and memory on one motherboard (or backplane or similar, in the case of more complicated setups, e.g., NUMA). In order to run OpenMP code on a cluster of machines connected by Ethernet, we must use other tools.

To run PLUTOized code on our cluster, we turn to a framework developed by Intel named Cluster OpenMP (hereafter ClOMP). According to its documentation, ClOMP "is based on a licensed derivative of the TreadMarks software from Rice University." [Hoe06] (For further information on TreadMarks, see [KCDZ94].) This framework allows our OpenMP code to execute on all of our machines even though they do not share a single address space and are connected by a commodity network interconnect.

Although other methods of executing OpenMP code over distributed shared memory systems exist, e.g., [HCK03], we found ClOMP to be the easiest and most turn-key method of doing so. OpenMP and ClOMP are both provided by the C compiler we use, `icc`, version 10.1.

4.2.3 Operating system

Because ClOMP requires a Linux kernel older than or equal to 2.6.24 and a `glibc` standard library older than or equal to 2.3.6, we installed an antiquated version 4.0 of Debian GNU/Linux on our machines.

5 Experimental results

GFLOPS were calculated by determining the number of floating point operations done (i.e., two additions and one multiply per cell per iteration) and dividing that by wall-clock time from start to end of for-loop execution. While this does not give us the highest numbers we could measure, we feel it most fairly represents the performance one might achieve with a real-world implementation of this framework for scientific computing purposes.

5.1 Pre-PLUTO runs

In order to establish a baseline for performance, we ran our 1D stencil with just one thread on one machine, untouched by PLUTO. For problem sizes that fit into RAM and were too big for cache ($\texttt{N} = 1{,}000{,}000$ or ≈ 15.2 MB) and $\texttt{T} = 10{,}000$ time steps, the fastest performance we could achieve with the best selection of `icc`'s optimizations was 1.03 GFLOPS. Even when `icc`'s own parallelization flags were used to (attempt to) create threaded code, we did not see any greater performance.

Care was taken to align the arrays such that cache interference did not lower performance. The specifics importance of this concern are mentioned later in this paper.

5.2 Single-threaded PLUTO runs

Next, we ran the code through PLUTO, using the same problem size, number of time steps, and array alignment as before. Tile size selection and its importance is mentioned later; for maximum performance, we used $\sigma = \tau = 1250$. After tiling, we were able to achieve 3.42 GFLOPS, a significant increase over the untiled code.

For these single-threaded experiments, there was no correlation between problem size and performance once the problem size was larger than L3 cache and the number of time steps was large enough for the execution to last more than a few seconds.

5.3 Multi-threaded, single-node PLUTO runs

To benchmark PLUTO's parallelization ability, we ran the same set of experiments using standard OpenMP (i.e., not ClOMP) with four threads on one machine. The parallelization done by PLUTO introduces non-parallelizable sections to the algorithm (i.e., the initial tiles on which the later ones depend), so Gustafson's law [Gus88] must be considered. Hence, to maximize

performance, we must increase the amount of work done. We used the same problem size of N = 1,000,000, but we increased the number of time steps by an order of magnitude to T = 100,000.

With this values and the same tile size of 1250 by 1250, we were able to record 11.5 GFLOPS, or 84% of linear speedup of the single-threaded PLUTO code. This number, itself an order of magnitude improvement over the use of just the C compiler without PLUTO, extols the power of tiling. Yet even though is not particularly notable in any research manner, it puts an important upper bound on the speeds we should expect when we use a distributed shared memory system.

5.4 Multi-threaded, multi-node PLUTO runs

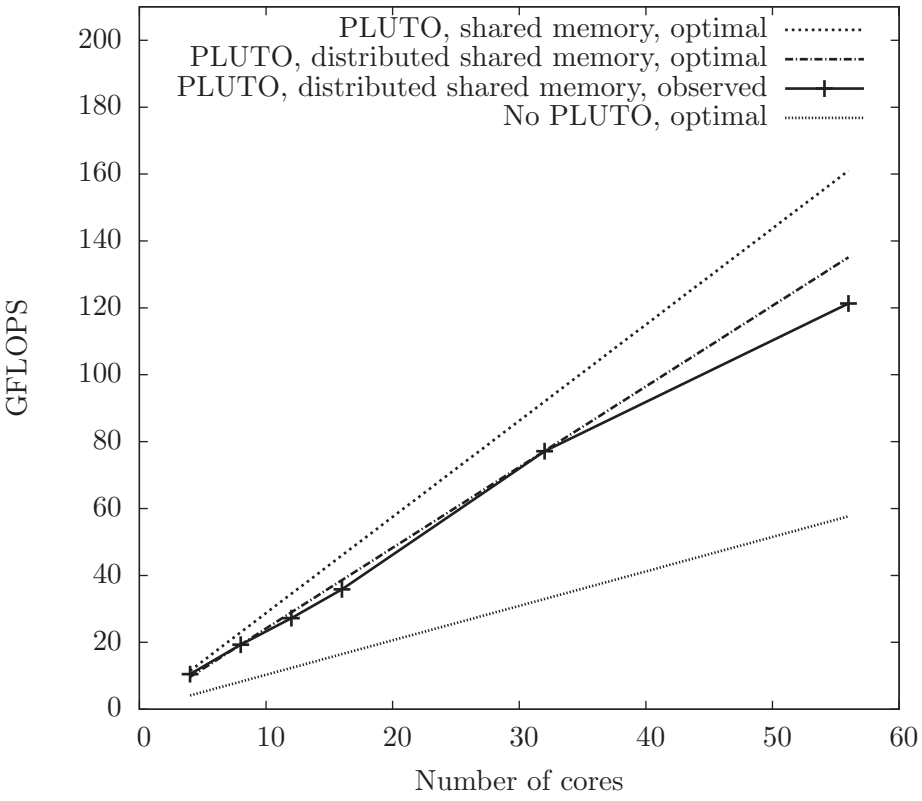
Our real research interest comes when we scale up to multiple machines. For maximum performance, we used known-good values for tile size (1250 by 1250), large time and space dimensions (8,000,000 for each), padding to prevent cache interference, and 50% thread oversubscription (as is mentioned later).

The initial tiles are not the only non-parallelizable sections that exist when ClOMP is used. In addition to those, ClOMP has a "ramp-up" time on the order of tens of seconds to a minute before all nodes consistently do computation, which lowers the reported GFLOPS figure and further increases the need for large problem sizes. During this period the nodes show no CPU usage; the presence and purpose of this is unclear because of the black-box nature of ClOMP.

On 14 machines with 56 total cores, we were able to achieve 121.3 GFLOPS, or 63% of linear speedup with a 26 minute execution time. Figure 2 plots measured performance and three theoretical linear speedups. The lowest line represents the performance of the 1D stencil without PLUTO tiling. Of course, this is entirely unachievable, because the code is not parallelizable without other tools or transformations. Nevertheless, it shows what our code could achieve if we could somehow parallelize the algorithm without tiling.

The middle theoretical line represents linear speedup of the simplest eight-core, two-node configuration of our distributed shared memory system. The fact that our observed data with more cores reflects this theoretical optimum as well as it does is both surprising and promising. The highest line marks a linear speedup of the four-core, one-node shared memory runs; this is higher than the distributed shared memory system, but one must remember that this performance comes at a much greater fiscal cost.

Figure 2: Performance vs. Number of Cores



Let us consider the Oracle machine as quoted before. Both our cluster and the Sun Fire have Intel "Nehalem" processors. Our processors are faster, so if we scale our single-threaded PLUTO code to match its clock speed, we get 2.76 GFLOPS. Multiplying this by 64 cores and assuming 100% efficiency (as unrealistic as that is) of the shared memory system results in an estimated 176.6 GFLOPS. This puts the Oracle system at 1.55 MFLOPS/dollar, and our lab at a more favorable 10.1 MFLOPS/dollar.

6 Unexpected variables and thoughts

While we initially thought that the most important variable to modify for high performance was tile size, after experimentation, we discovered that

data padding to mitigate cache artifacts and number spawned threads were both paramount to high performance.

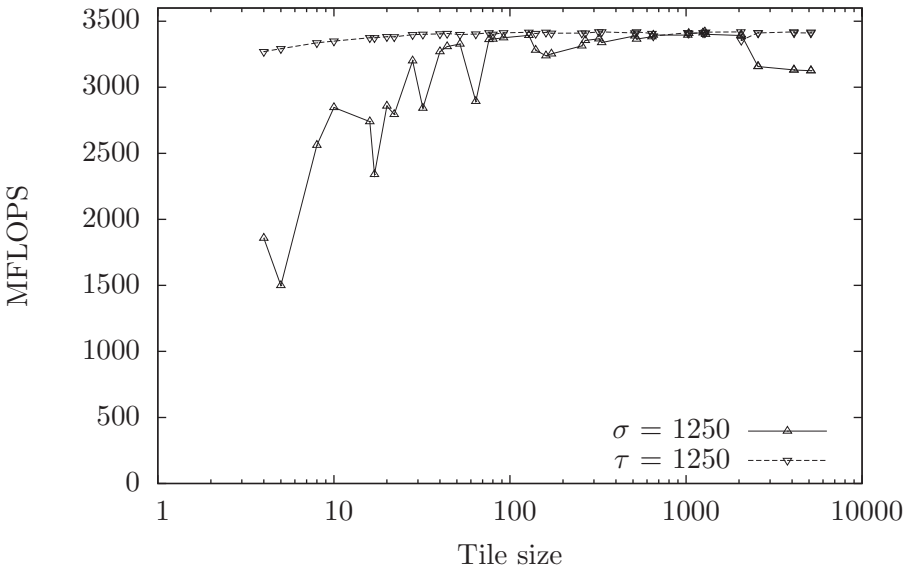
6.1 Tile size importance

Much to our surprise after reading many papers on the details of tile size selection, we were taken aback as to the comparative unimportance of tile sizes for this configuration.

Performance within a percent of the maximum speed of 3.42 GFLOPS was achieved with time step tile sizes τ from 256 to 5132, and space tile sizes σ from 1024 to 1292. These bounds held true once multiple threads were used. Considering this newly discovered band of acceptable sizes, we decided on a known-good value of 1250 by 1250, and continued all of the tests with that tile size. A graph keeping the space dimension constant at this known-good value and varying the time dimension and doing the reverse is shown in Figure 3.

This paper was initially intended solely to explore tile sizes necessary for maximum performance. Yet the aforementioned figure shows that "reasonably large" is an uncannily appropriate choice. As long as tile sizes are sized

Figure 3: Performance vs. Varying Tile Dimension, Single Thread



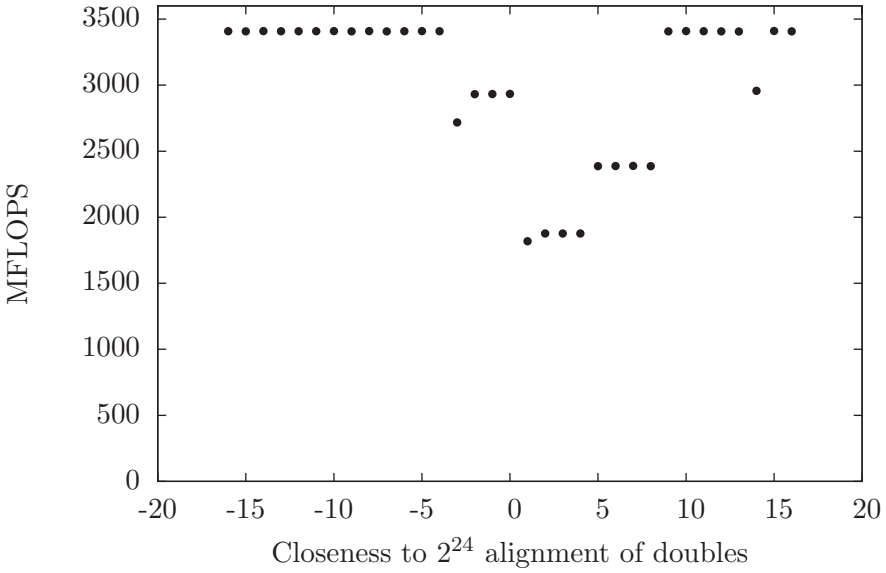
such that the pertinent data fits comfortably in cache, good performance is realized. A certain amount of care needs to be taken to decide what counts as "reasonable," but random selection still performs admirably.

6.2 Cache issues

Because of competition and the nature of the industry, specifics about the internals of the cache system on the Intel processors we use are secret. Nevertheless, we are aware of its associativity, and can imagine that with an eight-way set associative cache, artifacts of cache design should not hinder our performance.

Modern CPU caches have a certain "associativity" — a metric indicating the number of places data from main memory may exist in cache. Increased associativity increases cache hit rates (because there are more places where it can be located) but at the expense of increased latency (because more cache lines need to be searched for the desired data). At one end of the spectrum is the "direct-mapped cache" in which data in memory can only exist in one place in cache. The other end is the "fully-associative cache" in which every line in memory can be anywhere in cache.

Figure 4: Performance vs. Array Alignment for PLUTO-Tiled code, 1250 by 1250 Tile Sizes, Single Thread



If data in memory is aligned in a particular manner, lines may overlap forcing underutilization of cache; if, e.g., two values needed for computation can only fit in one cache line because of their location in main memory, then then benefits of cache cannot be realized, causing decreased performance.

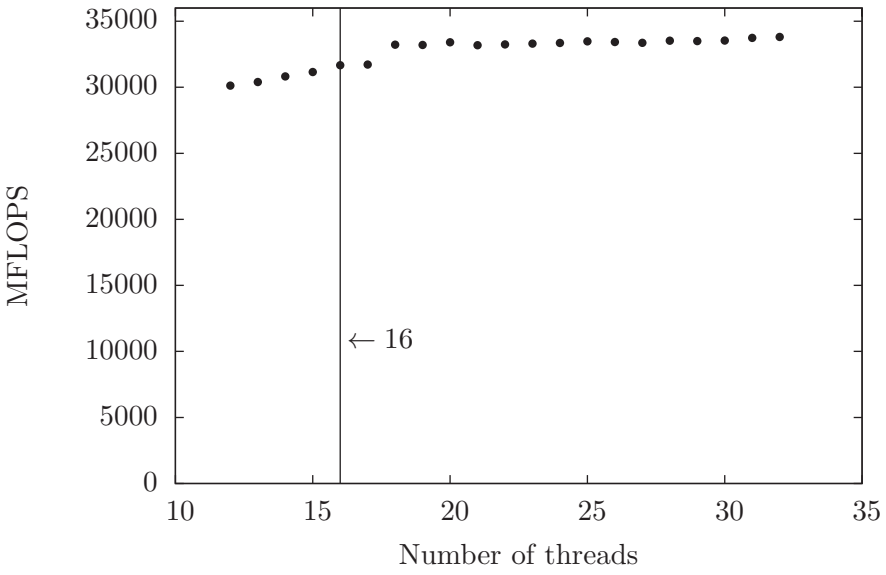
One expects slowdown and cache artifacts with a direct-mapped cache and low-associativity caches. High enough associativities, like the 8-way set associative L1 cache in the Core i7 860, should mitigate these cache artifacts. Yet our research disproves this claim; as Figure 4 shows, when the two data arrays are aligned in a certain manner, we see greatly depressed performance. Experimental results show that as long as the two arrays are do not exist 4 kilobytes $\pm$ 80 bytes apart, maximal performance is achieved.

Most notably, this is not a product of tiling; when both tiled and untiled, and when the arrays fit into L1 cache and did not, we noticed these artifacts.

6.3 Thread scheduling issues

The `htop` command (an enhanced version of `top`) shows system performance statistics. We used it to make sure that tiles were properly being spread across the cluster and cores used accordingly. Yet when we conducted runs

Figure 5: Performance vs. Number of Threads Launched, 16-Core Cluster



with the number of threads equal to the number of cores (which theoretically should cause complete CPU usage), we consistently noticed utilization significantly lower than 100%; typically all four cores would hover around 80%.

Having noticed the lower utilization, we posited that an increased number of threads would give higher performance. As shown in Figure 5, this is the case. This occurred solely during Cluster OpenMP runs; OpenMP runs gave us 100% usage in `htop`, and did not show increased performance with thread oversubscription.

Why this occurs is currently unclear. We are not experts on the internals of Linux scheduling, and the black-box closed-source nature of ClOMP inhibits further investigation into its internals.

7 Future work

The impressive performance we measured will hopefully drive interest in continued research. There are many unexplored benchmarks and characteristics of our research framework that can be investigated.

- It is possible to construct sub-tiles within each tile using a technique called multi-level tiling. While this paper focuses solely on single-level tiling, choosing appropriate sub-tile sizes may enhance cache and network use and further increase performance.

- The 1D stencil is the most "trivial" of imperfectly nested loops; future research can cover, e.g., 2D Jacobi stencils or even more complex algorithms like the TOMCATV part of the SPEC'92 benchmarks.

- The issue of underutilized cores and the need for oversubscription for maximal performance runs contrary to generally accepted benchmarking guidelines. Our research machines run a 2.6.24 version of the Linux kernel, with the Completely Fair Scheduler, introduced in the previous point release (2.6.23) and written by Ingo Molnár.

  Whether a different scheduler would change these performance figures is unknown. Booting our system into an older 2.6.18 kernel with the $\mathcal{O}(1)$ scheduler would allow us to test a different arrangement. If it is indeed a kernel issue and not a ClOMP problem, further investigation might cover even more schedulers, e.g., the "BFS" scheduler by Con Kolivas.

- Intel's Hyper-Threading allows each core of our processors to run two threads concurrently; a hyper-threaded quad-core processor appears as eight cores to the operating system. This may provide further performance benefits, especially in light of the thread scheduling issues.
- We have compared FLOPS per dollar to the Oracle system; one might want to do a comparison based on FLOPS per Watt, similar to the Green500 list.
- The machines were tested using a commodity 10/100 Fast Ethernet interconnect. We posit that tiling allows us to achieve high performance even with slower interconnects; running the same tests with 10Mbit Ethernet could validate or disprove this claim. Similarly, the effect of higher and lower latency has not been covered here and merits investigation.
- As always, scaling to even larger clusters with more nodes and cores could be enlightening.

8 Conclusion

The implications of scalable locality on distributed shared memory systems are vast. On dense stencils, we have shown that appropriate loop tiling can make a cluster of machines perform nearly as well as a shared memory system that costs an order of magnitude more. In principle this performance is not limited to simple stencils; clusters with properly tiled code should perform similarly with more complicated algorithms that permit scalable locality.

With such a striking increase in real-world MFLOPS/dollar values, those interested in a certain level of performance might want to begin to consider a cluster instead of a shared memory system. More legwork and research must occur before serious deployment, but this paper shows that the capabilities of clusters are not limited to embarrassingly parallel and other low-communication programs.

References

[BBK+08] Uday Kumar Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J Ramanujam, Atanas Rountev, and P Sadayappan. Automatic transformations for communication-minimized paral-

lelization and locality optimization in the polyhedral model. In *CC 2008*, 2008.

[BHRS08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 101–113, New York, NY, USA, 2008. ACM.

[CK95] Stephanie Coleman and Kathryn S. M Kinley. Tile size selection using cache organization and data layout. pages 279–290. ACM Press, 1995.

[Dre07] Ulrich Drepper. What every programmer should know about memory. *Red Hat white paper*, 2007. http://people.redhat.com/drepper/cpumemory.pdf.

[DW09] Tim Douglas and David Wonnacott. DSM + SL =? SC (or, can adding scalable locality to distributed shared memory yield supercomputer power?). In *MASPLAS 2009*, April 2009.

[EHK+02] Rudolf Eigenmann, Jay Hoeflinger, Robert H. Kuhn, David Padua, Ayon Basumallik, Seung-Jai Min, and Jiajing Zhu. Is OpenMP for grids? In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 02), Fort Lauderdale*, 2002.

[FCA05] Basilio B. Fraguela, Martn G. Carmueja, and Diego Andrade. Optimal tile size selection guided by analytical models, 2005.

[Gus88] John L. Gustfson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[HCK03] Lei Huang, Barbara Chapman, and Ricky Kendall. OpenMP for clusters. In *The Fifth European Workshop on OpenMP, EWOMP 2003*, pages 22–26, 2003.

[HCLK04] L. Huang, B. Chapman, Z. Liu, and R. Kendall. Efficient translation of OpenMP to distributed memory. In *Proceedings of the 4th International Conference on Computational Science (ICCS)*. Springer-Verlag, 2004.

[hHK04] Chung hsing Hsu and Ulrich Kremer. A quantitative analysis of tile size selection algorithms. *Journal of Supercomputing*, 27, 2004.

[Hoe06] Jay P. Hoeflinger. Extending OpenMP to Clusters. *Intel white paper*, 2006. http://cache-www.intel.com/cd/00/00/28/58/285865_285865.pdf.

[KBB+07] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 235–244, New York, NY, USA, 2007. ACM.

[KCDZ94] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *IN PROCEEDINGS OF THE 1994 WINTER USENIX CONFERENCE*, pages 115–131, 1994.

[SJBE03] Min Seung-Jai, Ayon Basumallik, and Rudolf Eigenmann. Optimizing OpenMP programs on software distributed shared memory systems. *International Journal of Parallel Programming*, 31:225–249, 2003.

[Won00] David Wonnacott. Using Time Skewing to eliminate idle time due to memory bandwidth and network limitations. In *International Parallel and Distributed Processing Symposium*. IEEE, May 2000.

[Won02] David Wonnacott. Achieving scalable locality with Time Skewing. *Internation Journal of Parallel Programming*, 30(3):181–221, June 2002.