Haverford College Computer Science Tech Report 2011-02
Archived 17 December 2011

An Experimental Investigation of Scalable Locality for Cluster Computing*

Ian Burnette
Haverford College
Haverford, Pa
iburnett@haverford.edu

Tim Douglas
Bump Technologies
Mountain View, Ca
timdoug@gmail.com

Micah Walter
Haverford College
Haverford, Pa
mwalter@haverford.edu

David Wonnacott
Computer Science Dept.
Haverford College
Haverford, Pa
davew@cs.haverford.edu

Abstract

Loop nest transformation has been used successfully to tune dense numerical codes for high performance on single- and multi-core shared-memory systems, but has not been widely applied to cluster computing. We have explored the use of these tools to produce the extremely high degree of memory locality needed to achieve high performance on a cluster with Intel's Cluster OpenMP software. Our experiments show high performance across our dedicated homogeneous 56-core/14-node research cluster with gigabit Ethernet. With proper tuning, performance drops by less than a factor of two, and sometimes only a few percent, when the network speed is reduced to 100Mb/sec. These results indicate that properly chosen compile-time optimizations can be used for cluster computing, and illustrate the importance of *scalable locality*, which may be of interest to programmers developing cluster codes manually.

∗. This work was supported by NSF grant CCF-0943455 and funds from Haverford College and a private donor.

1 Introduction

Loop nest transformation has been used successfully to tune dense numerical codes for high performance on shared-memory systems with both single and multiple processors or cores. This set of techniques has not been widely applied to cluster computing, perhaps because the machine parameters are very different from those used to develop and test the compiler algorithms.

Loop tiling [1] can be used to increase the execution speed of a multi-dimensional loop nest such as the first `i/j` nest in Figure 1. It was originally developed for uniprocessors, to increase cache hit rates by improving *locality of reference*. To illustrate this transformation with a simple example, consider what happens if the iterations of this loop are executed in the usual order for a large data set. In this case, element `A[1][1]`, which was defined when $i = 1$ and $j = 1$, may be gone from cache before the execution of iteration ($i = 2$, $j = 1$), when it is needed to play the role of `A[i-1][j]`. Thus, an execution of the loop body may need to read three values from main memory (`A[i][j]` and `A[i][j-1]` will likely be in cache) to execute its six floating-point operations, and performance may be limited by memory bandwidth rather than computation speed on modern CPUs.

Cache use is improved by executing groups of iterations in "tiles", for example as 10 by 10 blocks with all iterations where ($1 \leqslant i \leqslant 10$, $1 \leqslant j \leqslant 10$) occurring before those where ($1 \leqslant i \leqslant 10$, $11 \leqslant j \leqslant 20$). This limits the number of other values to be retrieved before a given value is reused (in this example, to about 20 values from `A`) regardless of the problem size. This increase in the locality of the reference that re-uses an array element means values can be retrieved from cache rather than main memory even for a large problem. In many cases, an iteration of this loop will need to retrieve only one element of `A` for each six-operation execution of the loop body. This may let the memory system keep the CPU busy on a system in which the rate of floating point operations does not exceed six times the speed at which floating point values can be delivered from cache, i.e., with a compute:bandwidth ratio (called *machine balance*[2][3]) of no more than 6:1.

Tiling also provides a convenient way to partition data for concurrent execution. Array elements near the edge of a tile may require communication, but those near the center of a tile typically will not. By increasing the tile size, we can increase the ratio of computation to communication.

Traditional algorithms for loop tiling can handle only perfectly nested loops, and thus cannot be used for the `t/i` nest in Figure 2. More recent techniques[4][5][6][7] based on the *polyhedral model* relax this restriction and allow tiling of Figure 2, or the entire three-dimensional space of iterations of Figure 1. This provides a higher degree of locality by taking advantage of re-use of array elements between the two different `i/j` nests and across different time steps, thus improving performance on high-performance hardware [5]. For a review of these techniques or the polyhedral approach to compilation, see the cited papers or the sources listed in the Wikipedia entries for the polyhedral model[8] and associated libraries[9].

In prior work[10][6], Wonnacott argued that tiling a time-step loop differs fundamentally from most compile-time transformations (including tiling of spatial loops) in that it can often produce *scalable locality*, i.e. the locality (or ability of the system to re-use an element while still in cache) can be scaled to match machine balance. Tiling the `i/j` nest of Figure 1 can not produce an operation to new-value-reference ratio (called *compute balance* in [10]) above 6:1 for any tile size. However, when tiling of the full `t/i/j` loop nest, increases the tile size in the time dimension can produce we can arbitrarily increase the compute balance within a tile, and thus the potential for re-use. This tiling of time step loops (often called *time tiling* or *time skewing*) also makes it possible to improve the multiprocessor computation:communication ratio without requiring impractical data set sizes, though Wonnacott did not experiment with this in [10] or [6].

Factors such as cache size limitation or cache interference may prevent increased compute balance from yeilding increased locality. Wonnacott gave formulae for cache size requirement and discusses the prevention of iterference via *storage mappings* together with the iteration-space mappings used to reorder iterations, Wonnacott provided empirical evidence for this claim on uniprocessor systems by transforming several benchmark codes to produce in-core processing speeds for out-of-core data sets (in one case, running a data set residing in virtual memory faster than the untransformed code could execute a data set from level 2 cache, hundreds of times faster than the original code on the big data set). Wonnacott also predicted[6] that the tiled code could be used to provide good parallel performance on multiprocessors with relatively slow interconnects, but was unable to test this claim.

```
for (t = 0; t < T; t++) {
  for (i = 1; i < N-1; i++)
    for (j = 1; j < M-1; j++)
      new[i][j] = (A[i][j-1]+
         A[i-1][j]+A[i][j]*4+A[i+1][j]+
                   A[i][j+1]) * 0.125;

  for (i = 1; i < N-1; i++)
    for (j = 1; j < M-1; j++)
      A[i][j] = new[i][j];
}
```

Figure 1. A Two-Dimensional Jacobi Stencil

```
for (t = 0; t < T; t++) {
  for (i = 1; i < N-1; i++)
    new[i]=(A[i-1]+2*A[i]+A[i+1])*0.25;
  for (i = 1; i < N-1; i++)
    A[i] = new[i];
}
```

Figure 2. A One-Dimensional Jacobi Stencil

Below, we report on our experiments to exploit scalable locality to tune the benchmarks of [10] for cluster computers and describe remaining challenges. We have explored the use of the Pluto automatic parallelizer[11] and the AlphaZ program transformation system[12] to perform the time tiling transformation on these benchmarks. These tools can each produce OpenMP codes, which we executed on a 56-core/14-node homogeneous dedicated cluster using Intel's Cluster OpenMP [13].

2 Experimental Framework

The PLUTO automatic parallelizer[11] has shown impressive results in modifying C source for parallelism in dense matrix computations [7][14][15]. 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. Pluto accepts C source code with `#pragma` directives indicating which loop nests are to be tiled, and then performs tiling. Tile sizes can be selected manually with a configuration file, and Pluto can be set to produce uniprocessor C code or C code annotated with directives for OpenMP loop parallelization.

The AlphaZ system[12] is a tool that generates code for mathematical algorithms using the polyhedral model. AlphaZ generates this code not from existing C code, but rather from a series of equations written in a simple pure-functional programming language specially designed for the system, called Alphabets. For example, Figure 3 shows one way to express the one-dimensional stencil as AlphaZ equations. In addition to the equational Alphabets file, AlphaZ also takes a schedule as input; this schedule file contains specifications on how AlphaZ should handle statement ordering and memory mapping, as well as giving the option of enabling automatic tiling (or parallelization) of the code. Thus, the programmer can separately describe both the result to be computed and the manner in which the computation is being carried out. Performance tuning can be done with a great deal of manual control and with confidence that the correctness of the code will not be perturbed. AlphaZ, like Pluto, generates sequential or OpenMP-annotated code.

```
affine oneD {T,N | T>0 && N>4}

given
  double Initial {i | 0<=i<=N-1};
returns
  double Final {i | 0<=i<=N-1};
using
  double A {t,i | 0<=t<=T && 0<=i<=N-1};

through
  A[t,i] = case
    {| t==0} : Initial[i];
    {| t>0 && i==0} : (A[t-1,i]);
    {| t>0 && i==N-1} : (A[t-1,i]);
    {| t>0 && 0<i<N-1} :
      (  A[t-1,i-1] +
       2*A[t-1,i]   +
         A[t-1,i+1]  ) / 4.0;
  esac;

  Final[i] = A[T,i];
.
```

Figure 3. AlphaZ Equations for 1-d Stencil

Standard OpenMP programs are only executable on traditional shared memory systems. In order to run OpenMP code on a cluster of machines connected by Ethernet, we must use other tools. To run code on our cluster, we turn to a framework developed by Intel named Cluster OpenMP (hereafter ClOMP). According to its documentation[13], ClOMP "is based on a licensed derivative of the TreadMarks software from Rice University." 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.

ClOMP requires a number of configuration files in which many parameters can be set. The list of nodes in the cluster is provided in one file, and another describes the number of processes to be started (`--processes=`...), the number of independent threads within each process (`--process_threads=`...), and various other parameters. Except where stated otherwise, we used one process per node and four threads per process, with the parameters `--sharable_heap=100M` and `--divert-twins` (the latter was suggested by a message from the ClOMP system but we have not been able to find documentation for it).

OpenMP and ClOMP are both provided by the C compiler we use, Intel's `icc` version 10.1. We ran `icc` with optimization level `-O3`, and used the `-cluster-openmp` option to process OpenMP programs for execution on the cluster or the `-openmp` option for single-node OpenMP results. For the two-dimensional stencil, we used the option `-mcmodel=medium` to allow the use of our very large data set.

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, and purchased network cards compatible with this release.

Our codes were tested on a homogeneous cluster of 14 quad-core computers, which were dedicated to this research when experiments were being run, but available for teaching at other times. Each of the nodes 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 physical core, but that was disabled for our benchmarks to reduce the number of issues complicating our analysis.

The machines are connected via Ethernet switches administered by Haverford College. Early results were performed with ten nodes on a 10/100 switch and four on a gigabit switch, but later all were moved to the gigabit switch. Nodes connected at the gigabit switch could be configured to communicate at

100Mb/second with the `ethtool` command; a few quick experiments with copying large files over NFS confirmed that the network bandwidth had decreased as expected.

3 Experiments to be Run

We chose to experiment with two simple Jacobi stencil computations above, which have frequently been used to illustrate the benefits of time tiling on uniprocessors[5][10][7], and with a more significant kernel: a variant of the Tomcatv benchmark of SPEC95 with the early exit from the loop removed (to make it fit more easily into the polyhedral model) and with a much larger data set and number of time steps (to make it an appropriate test for a cluster of modern systems). The former were chosen to illustrate potential benefits, as they are well within the abilities of the Pluto and AlphaZ systems. The latter was used in [10] and was chosen to present a challenge to these tools due to the greater complexity of the code while remaining within the polyhedral model: the outer time-step loop includes a sequence of seven one- and two-dimensional loop nests containing a total of 30 computation statements with a complex dataflow pattern both within and across time steps.

We measured total run-time of the loop nests shown in Figures 1 and 2 via the linux `gettimeofday` function; our "start time" was marked after a non-parallel initialization of values for array `A`, and our "end time" was marked by exit from the computation loops.

Data set size was set to `N=8000000` for the one-dimensional stencil, and to `N=M=15500` for the two-dimensional stencil. The number of time steps was scaled up with the total number of cores, with 575000 time steps per quad-core node (producing a total of about 8000000 for the full cluster) for the one-dimensional stencil and 6250 per node for the two-dimensional stencil. These settings produce total run times between one and three kiloseconds for most of our experiments, enough to amortize the start-up costs and give an idea of the steady-state processing speed for a large problem.

Multi-node runs without Pluto were so much slower that we reduced the number of time steps to 25 for most runs (and conducted a few longer runs to confirm our expectation that this would give processing rates consistent with those for more time steps, due to the the lack of inter-time-step optimization).

Tile sizes were selected by an empirical search conducted by the undergraduate members of the team. We are currently investigating the question of whether various tile-size-selection heuristics are applicable to our context, and if so how accurate they are. For the one-dimensional stencil, we chose tile size 1250x1250; for the two-dimensional stencil we employed a two-level tiling, with the innermost tile size set to 8 iterations of the `j` loop, 64 of the `i` loop, and 1 of the `t` loop (with a total data footprint that fits in each core's L1 cache), and the level two tiles comprised of 16x2x128 level-one tiles, unless otherwise noted.

For our baseline of comparison on our speed-up graphs, we chose the processing speed of a single quad-core node using ClOMP. This was generally very close to the quad-core speed using standard OpenMP, though in all cases the quad-core node ran at well under four times the single-core speed (see Table 1).

Problem	1-core	OMP 4-core	ClOMP 4-core
1-d w/o Pluto	1.8	2.5	2.5
2-d w/o Pluto	1.3	2.1	2.0
1-d with Pluto	4.5	14.8	12.8
2-d with Pluto	1.8*	5.6	5.5

Table 1. Single-Node Speeds (in GFLOPS)

Note that the 1-core speed shown in Table 1 for the two-dimensional stencil processed by Pluto could probably be improved significantly by selecting tile size parameters for single-core performance; the speeds shown in the table all use the tile sizes given in Section 2. We believe the 1-core speed for the one-dimensional stencil is close to the maximum possible speed for any tile size[16]. We are currently investigating the limits to the quad-core speedup with time-tiled code; our current hypothesis is that in may result from the benefits of "Intel Turbo Boost" technology for the single-core run.

4 Results using Pluto

Pluto was able to successfully transform both Jacobi stencils, and the transformed code dramatically outperformed the result of icc without Pluto on both single-core, single-node, and cluster runs.

4.1 One-Dimensional Stencil

Figure 4 shows total compute speed for the one-dimensional stencil. Sets of dots indicate experimental results, and solid lines indicate a linear speed up of single quad-core node (i.e., if k nodes ran k times faster than one node executing the parallel code with ClOMP). The upper line and three sets of dots show the performance of time-tiled codes produced by Pluto, with the different dots corresponding to different network speeds.

The lower line and two sets of dots show results without time tiling, in which each of the `i` loops marked for parallel execution with the directive `#pragma omp parallel for shared(t) private(i)`. These codes ran so slowly that we had to reduce the number of time steps to 25. (A few experiments with slightly larger sizes confirmed our expectation that increasing time steps would not help this code, as each time step simply re-executes the same slow computation, without any advantage of inter-time-step locality).

The difference between the upper and lower lines shows the importance of time-tiling on a single node. Even on a single core, time tiling can produce significant speedups for large data sets due to improved memory locality[5][10].

The distance between each set of dots and the corresponding linear speedup shows the impact of inter-node communication. The time-tiled codes stay close to the (much higher) linear speedup line across a much wider range of cluster sizes, demonstrating that time tiling not only improves single node performance, but also helps hide the enormous communication cost between nodes: with gigabit Ethernet, the performance of the non-time-tiled code drop significantly away from linear speed-up by 20 total cores, but the time-tiled code retains a very high percentage across the entire cluster size, producing 163 GFLOPS, over 90% of linear speed up (178 GFLOPS) for 56 cores.

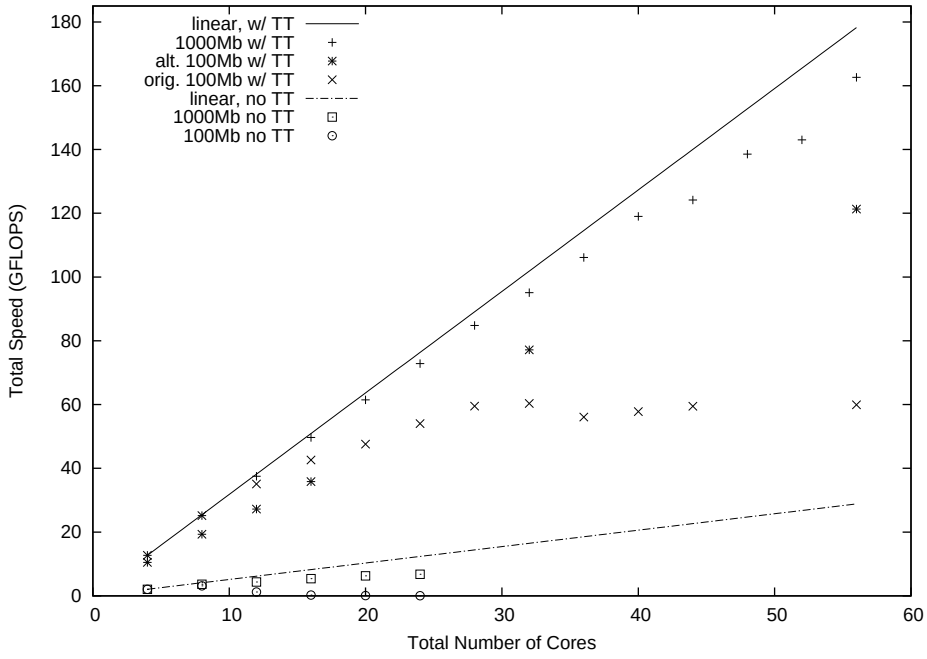

Figure 4. One–Dimensional Stencil Total Speed

With 100Mb ethernet, the time tiled code gives significant speed up out to 32 cores, while the performance of the non-time-tiled code quickly falls below single-node performance and drops toward zero as communication costs outweigh computation (by 24 cores, the total processing speed delivered by the cluster is down below 50 MFLOPS). Above 32 nodes, time tiled performance becomes more sensitive to details of tile size and ClOMP configuration; the "orig" data correspond to the original tiling, used in our experiments on the gigabit Ethernet experiments; the "alt." data correspond to the alternate configuration used by Douglas for a slightly different formulation of the one-dimensional Jacobi stencil in [16].

In Section 6, we discuss possible causes of, and remedies for, the lack of performance increase for time-tiled code on large clusters.

4.2 Two-Dimensional Stencil

Figure 5 shows corresponding results for the two-dimensional stencil. To generate non-time-tiled results here, both sets of `i` loops are once again marked for parallel execution, with the directive `#pragma omp parallel for shared(t) private(i, j)`.

For the two-dimensional stencil, classic tiling of each perfectly nested `i/j` loop pair can produce significant performance improvements over on tiled code, and the importance of time tiling on a single node is reduced. However, the tiling of spatial loops cannot be scaled up with any problem size parameter, and thus it cannot equal the time-tiled performance. Furthermore, performance suffers dramatically when the tiles are distributed around the cluster. The time-tiled code does not stay consistently close to the linear speed up line, even with gigabit ethernet, though it repeatedly approaches this line (jumping back to 87% at 24 cores after the fall to 73% at 20, and back to 92% at 44 cores after falling to 55% at 40). The causes of these drops are discussed in Section 6.

Note that the time-tiled code not only beats the non-time-tiled code in terms of raw performance, it also holds on to a larger fraction of linear speedup over a larger fraction of the cluster. Even with gigabit Ethernet, performance of the non-time-tiled code is below 50% of linear at 28 cores. The performance of the time-tiled code suffers less from the drop in network speed, holding on to 57% of its gigabit Ethernet performance, or 42% of linear speed-up, on the full 56-core cluster. At a mere 24 cores, the 100Mb run of the non-time tiled code retains only 13% linear speedup of its (lower) single-node performance.

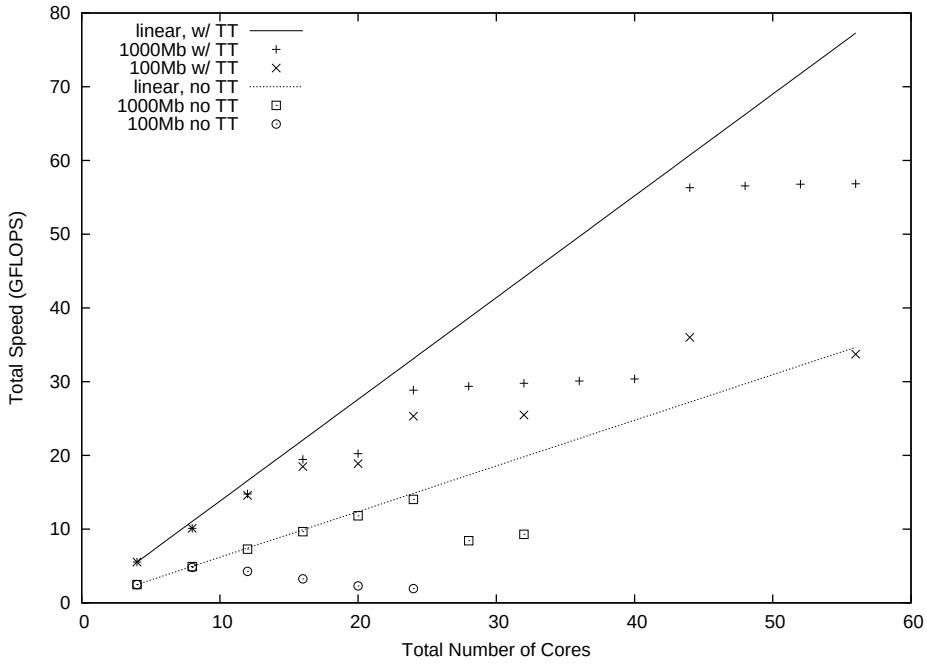

Figure 5. Two-Dimensional Stencil Total Speed

4.3 Simplified Tomcatv

We were not able to get Pluto to transform this benchmark due to the compiler failing during the transformation, with the most recent release we have (PLUTO 0.7.0-13-g25752c5) causing glibc to dectect a "double free or corruption (out)". We have contacted the developers, and hope to succeed with this code by "cleaning up" the program produced by `f2c`.

5 Results using AlphaZ

Our experiments with the AlphaZ system produced performance of about $\frac{1}{2}$ to $\frac{3}{4}$ of Pluto's codes for the one-dimensional stencil. Unfortunately, AlphaZ does not currently support two-level tiling, so it cannot express the transformation we wish to apply for the two-dimensional stencil, and we performed only limited experiments with this benchmark. On the other hand, AlphaZ was able to compile our simplified Tomcatv benchmark, though we have not yet had a chance to experiment with tiled or parallel execution of this code. For details on our experiments with AlphaZ beyond those given below, see [17].

5.1 One-Dimensional Stencil

When using AlphaZ to transform the one-dimensional stencil, we were able to achieve tiled single-core performance of 3.4 GFLOPS and single-node performance of 10.9 GFLOPS, both roughly 75% of the figure given for Pluto in Table 1. (Note, however, that the values are not precisely comparable, as they used different tile sizes.) We found it interesting that the performance of the most straightforward set of equations given in Figure 3 was sometimes slightly less than what we produced with a set of equations that mimicked the two arrays of the imperative code.

We used ClOMP to execute the OpenMP programs produced by AlphaZ on a 32-core subset of our cluster. In several experiments with different numbers of time steps, total performance did not exceed 40 GFLOPS, roughly half the performance of the Pluto-generated code.

5.2 Two-Dimensional Stencil

We were able to produce very quick uniprocessor codes for the two-dimensional stencil with AlphaZ. However, two-level tiling was critically important for good performance on the cluster, so we did not explore this benchmark in detail.

5.3 Simplified Tomcatv

We experimented with several AlphaZ formulations of the simplified Tomcatv benchmark. For the best of these, AlphaZ produced code that ran at about 75% of the speed of the code produced by `icc` for the C version we had tried to transform with Pluto. We look forward to having a chance to experiment with tiling and parallelization of this code, though it is not clear that AlphaZ will be able to tile it as it currently exists: one loop nest has intra-iteration dataflow along an entire dimension of an array, making tiling more complex than it would be for a two-dimensional stencil.

6 Discussion

The combination of Pluto and ClOMP has given us impressive cluster performance for the simple stencil computations, but we were unable to get Pluto to transform the more complex loop nest. We plan to modify this code without significantly simplifying the algorithm, in hopes of getting success with Pluto, and provide feedback to the developers.

AlphaZ was able to handle all algorithms and provide good performance on a single core. It can also express storage mapping transformations, unlike Pluto, and it will let us explore different formulations of the time skewing transformation. Its lack of direct support for multi-level tiling hinders some of our work for the two-dimensional stencil, though we hope to find a work-around, and we look forward to experimenting with it further.

While running our experiments, we noticed a number of factors that seem to contribute to the execution speed of our time-tiled codes running on a cluster via ClOMP. Where performance suffered for the two dimensional stencil despite the use of gigabit Ethernet, the `htop` tool indicated that some nodes were being fully utilized, while others remained completely idle. The drops from 92% efficiency at 44 cores to 73% at 56, and from 87% at 24 cores to 55% at 40, seem to be due to additional nodes being added to our system without being given any work. We believe that more consistent efficiency could probably be achieved for this code was a different infrastructure, or possibly with different options to ClOMP.

In contrast, the lower performance with 100Mb Ethernet for our one-dimensional stencil involved many nodes with relatively low processor utilization. Preliminary experiments with larger tiles, which might provide better balance, gave even worse results. The lower time-tiled results of Figure 4 occurred with the settings described in Sections 2 and 3 of this article when all nodes were connected with a gigabit switch and `ethtool` was used to set each node to communicate at 100Mb/sec; for our first set of experiments[16], which produced better results, most nodes were connected via a 100Mb/sec switch and `ethtool` was used on only four nodes, different level two tile sizes or ClOMP parameters may have been used, and experiments were done with a one-dimensional Jacobi stencil with the same dataflow pattern and data set size, but for which the computation in the loop nest did not double the `A[i]` term, and the sum was multiplied by $\frac{1}{3}$ rather than $\frac{1}{4}$. We plan to investigate issues of benchmark variation, network hardware, ClOMP parameters (including number of processes per node and number of threads per process), tile size, etc., in January 2012, and hope to have an explanation before the CCGrid meeting. One challenge arising from the use of a collection of different software components is a wide number of places where a performance bug could hide.

7 Related Work

The SuperMatrix system[18] supports the execution of dense matrix codes on clusters, but code must be specially written for it. While most of the supermatrix results presented at the Cluster 2007 conference[18] are not directly comparable to ours, the one graph of efficiency (for Cholesky factorization) shows efficiencies consistently below 90% for eight threads (on a sixteen-processor system; lower efficiencies were obtained for more threads).

Min, Basumallik, and Eigenmann have explored compile-time optimization to tune parallel codes written with OpenMP for distributed shared memory systems[19][20][21]. Unlike our work, this approach presumes that the programmer has already identified and expressed parallelism. More significantly, this work did not target scalable locality, focusing on data privatization, "selective data touch", "overlap of computation and communication through dynamic scheduling and barrier elimination", "recognition of transformed reduction idioms", and "optimization of sends and receives for shared data". Results in these papers are hard to compare with our preliminary results, as they did not scale up problem sizes and obtained super-linear speedups in some cases[21].

For some problem domains, compilers have been developed to automatically generate the appropriate sends and receives from annotated sequential code (for example, Fortran D[22]), but we do not know of recent work to compile this sort of code for clusters.

The work that is most closely related to ours is that of Classen and Griebl[23]. This work, like ours, uses a modern compiler infrastructure to achieve effective automatic parallelization on a distributed memory system. Classen and Griebl also measured speedups of a one-dimensional stencil, though their code was an in-place two-point stencil rather than our three-point stencil with a temporary array (we believe either system could be applied to either stencil). They demonstrated roughly 70% efficiency for 4 dual-core 1GHz Pentium III nodes. In principle, we believe this sort of special-purpose code generator for distributed memory could at least equal any combination of shared-memory optimization and distributed shared memory software — the main drawback of the "code generation for distributed memory" approach is that it requires that a specialized distributed memory code generator be integrated into each optimizing compiler.

8 Conclusions

We have demonstrated that static code optimization tools can produce good performance on a cluster of non-trivial size, through a simple coupling of techniques designed for shared memory but scaled up for higher locality, together with software distributed shared memory. These results do not require exotic network interconnects, and can even be used with less than state-of-the-art interconnect speeds. This makes clusters an appealing alternative to the traditional expensive supercomputers that are usually targeted by optimizing compilers.

While the compile-time transformations we have used are designed for dense numerical codes, the ideas behind our work may be more broadly applicable in the cluster computing context. Specifically, we see this as an illustration of the general principle that optimization infrastructure designed to allow tuning of locality as well as parallelism can be used for supercomputers and homogeneous dedicated clusters.

Bibliography

[1] F. Irigoin and R. Triolet, "Supernode partitioning," in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, 1988, pp. 319–329.

[2] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Technical Committee on Computer Architecture Newsletter*, Dec 1995.

[3] D. Callahan, J. Cocke, and K. Kennedy, "Estimating interlock and improving balance for pipelined machines," *Journal of Parallel and Distributed Computing*, vol. 5, no. 4, pp. 334–358, Aug. 1988.

[4] J. McCalpin and D. Wonnacott, "Time Skewing: A value-based approach to optimizing for memory locality," Dept. of Computer Science, Rutgers U., Tech. Rep. DCS-TR-379, Feb. 1999. [Online]. Available: `ftp://www.cs.rutgers.edu/pub/technical-reports/dcs-tr-379.ps.Z`

[5] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," in *ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, May 1999, pp. 215–228.

[6] D. 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.

[7] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. 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*. New York, NY, USA: ACM, 2007, pp. 235–244.

[8] Wikipedia, "Polytope model — Wikipedia, the free encyclopedia," 2010, [Online; accessed 03 Dec 2011]. [Online]. Available: `http://en.wikipedia.org/wiki/Polytope_model`

[9] ——, "Frameworks supporting the polyhedral model — Wikipedia, the free encyclopedia," 2010, [Online; accessed 03 Dec 2011]. [Online]. Available: `http://en.wikipedia.org/wiki/Frameworks_supporting_the_polyhedral_model`

[10] D. Wonnacott, "Achieving scalable locality with Time Skewing," *Internation Journal of Parallel Programming*, vol. 30, no. 3, pp. 181–221, Jun. 2002.

[11] U. Bondhugula, "PLUTO - an automatic parallelizer and locality optimizer for multicores," http://www.cse.ohio-state.edu/ bondhugu/pluto/.

[12] Melange-Research-Group, "AlphaZ," http://www.cs.colostate.edu/AlphaZ/wiki/doku.php.

[13] J. P. Hoeflinger, "Extending OpenMP to Clusters," *Intel white paper*, 2006. [Online]. Available: `http://cache-www.intel.com/cd/00/00/28/58/285865\_285865.pdf`

[14] U. Bondhugula, A. 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*. New York, NY, USA: ACM, 2008, pp. 101–113.

[15] U. K. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model," in *CC 2008*, 2008.

[16] T. Douglas, "An empirical study of the performance of scalable locality on a distributed shared memory system [Haverford College undergraduate (senior) thesis]," May 2011.

[17] M. J. Walter and D. G. Wonnacott, "Experiences with expressing and optimizing dense numerical algorithms in AlphaZ," Haverford College Computer Science Department, Tech. Rep. HC-CS-TR-2011-01, Oct. 2011. [Online]. Available: `http://cs.haverford.edu/TechReports/HC-CS-TR-2011-01.pdf`

[18] E. Chan, F. G. V. Zee, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, "Satisfying your dependencies with supermatrix," *Cluster Computing, IEEE International Conference on*, vol. 0, pp. 91–99, 2007.

[19] A. Basumallik, S.-J. Min, and R. Eigenmann, "Towards openmp execution on software distributed shared memory systems," in *ISHPC '02: Proceedings of the 4th International Symposium on High Performance Computing*. London, UK: Springer-Verlag, 2002, pp. 457–468.

[20] S. jai Min, A. Basumallik, and R. Eigenmann, "Optimizing openmp programs on software distributed shared memory systems," *International Journal of Parallel Programming*, vol. 31, pp. 225–249, 2003.

[21] A. Basumallik and R. Eigenmann, "Towards automatic translation of openmp to mpi," in *Proc. of the International Conference on Supercomputing, ICS'05*, 2005, pp. 189–198. [Online]. Available: `http://www.ecn.purdue.edu/ParaMount/publications/BasEi05.pdf`

[22] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiling fortran d for mimd distributed-memory machines," *Communications of the ACM*, vol. 35, pp. 66–80, 1992.

[23] M. Classen and M. Griebl, "Automatic code generation for distributed memory architectures in the polytope model," *Parallel and Distributed Processing Symposium, International*, vol. 0, p. 243, 2006.