Haverford College Computer Science Tech Report 2009-01
Archived 30 July 2009

$\mathbf{DSM + SL \stackrel{?}{=} SC}$

(Or, Can Adding Scalable Locality to Distributed Shared Memory Yield SuperComputer Power?)

Tim Douglas
tdouglas@haverford.edu

Sharon Warner
swarner@haverford.edu

David G. Wonnacott
davew@cs.haverford.edu

Haverford College
Haverford, PA 19041

Abstract

Distributed Shared Memory, such as that provided by Intel's Cluster OpenMP, lets programmers treat the combined memory systems of a cluster of workstations as a single large address space. This relieves the programmer of the burden of explicitly transferring data: a correct OpenMP program should still work with Cluster OpenMP. However, by hiding data transfers, such systems also hide a major performance factor: correct OpenMP programs with poor locality-of-reference become correct but intolerably slow Cluster OpenMP programs.

Scalable Locality describes the program property of locality that increases with problem size (just as Scalable Parallelism describes the property of parallelism that increases with problem size). In principle, the combination of an optimization that exposes scalable locality and a distributed shared memory system should yield a simple programming model with good performance on a cluster.

We have begun to explore a combination of Cluster OpenMP and the Pluto research compiler's implementation of *time tiling*, which can produce parallel programs with scalable locality from sequential loop-based dense matrix codes.

In this article, we review our approach, discuss our performance model and its implications for tile size selection, and present our most recent experimental tests of the viability of our approach and validity of our performance model. Our performance model captures only machine-independent issues that are critical to setting tile size. It deduces *lower bounds* on tile dimensions from a combination of purely hardware parameters (e.g. memory bandwidth) and parameters describing the software without reference to any particular hardware (e.g. number of live values produced by the loop nest). We also model load imbalance from OpenMP barriers, which is significant for smaller problems. Our results, while preliminary, are quite encouraging.

1 Introduction

The quest for good performance of parallel applications on clusters of workstations has traditionally employed software techniques that are quite different from those applied to the programming of parallel supercomputers. In particular, static automatic parallelization has been employed on supercomputers (especially for dense matrix codes on shared memory systems) but has not been successful on clusters — recent proceedings of IEEE's "Cluster" and "CCGRID" conferences include numerous papers about libraries and languages to allow manual control of parallelization, and about dynamic systems to adapt programs as they run, but very little work on either static parallelization or dense matrix codes.

We believe the lack of success with static parallelization is due in part to the inability of classic parallelization techniques to expose sufficient memory locality. When all nodes in a cluster are identical and reliable over the run-time of an application, a cluster of commodity workstations is equivalent to a supercomputer with extremely non-uniform memory access time. The existence of "distributed shared memory" software such as Intel's Cluster OpenMP ("ClOMP")[Hoe06] makes this equivalence clear: any legal OpenMP program can work on ClOMP (provided the proper directives about shared data have been added). However, programs with inadequate locality of memory reference, such as those produced by most automatic parallelizers, typically run extremely slowly.

Modern research compilers are capable of transforming imperfectly nested loops, allowing them to produce an extremely high degree of locality on many dense matrix codes. However, these compilers have primarily been applied to multi-core chips and other shared-memory architectures, rather than distributed memory systems or clusters.

We have begun to explore a combination of software *distributed shared memory* and static optimization to exploit *scalable parallelism* and *scalable locality* for shared-memory systems. Our results, while preliminary, are highly encouraging: we achieved over 95% of linear speedup for an eight-core system that consisted of 4 dual-core workstations connected by Ethernet. This result is at least competitive with the few dense matrix applications we have found in the cluster computing literature, and was achieved without the development of significant new algorithms or software. Equally importantly, our results demonstrate that effective optimization techniques can stem from the view of supercomputers and clusters as points on a continuum of machine balance parameters, rather than as distinct targets. This view encourages the development of general algorithms that scale over different architectures, rather than collections of specific techniques tuned to specific systems. We believe we are the first to apply this particular combination of tools to this problem; literature searches and personal communications (e.g. [Hoe09]) have so far yielded no directly related work (see Section 6 for details).

1.1 Scalable Parallelism

Amdahl's Law[Amd67] describes limits on the use of parallelism to speed up the application of a given algorithm to a given data set — as we add more processors, inherently sequential elements of the algorithm begin to dominate compute time, limiting the possible speedup.

In contrast, Gustafson's Law[Gus88] describes the opportunity to make effective use of an unlimited amount of parallelism by applying some algorithms for ever-larger data sets. If doubling the number of processors does not let us run a problem of size N in half the time, it may

still let us run a problem of size $2N$ in the same amount of time. Programs with this characteristic of allowing effective parallelization that grows with the problem size are said to exhibit *scalable parallelism*.

However, a program with scalable parallelism may not speed up on all parallel architectures. Consider a program in which processors often require values that were recently produced on some other processor. Such a program might, in principle, perform well on a system in which any processor has uniform fast access to any memory cell, but perform poorly when the cost of inter-processor communication is high (as it is on a cluster). This effect would occur whether the algorithm were expressed in a message-passing or a distributed shared memory framework.

1.2 Scalable Locality

Non-uniform memory access time is not unique to multiprocessor systems; a significant amount of work has gone into compiler optimizations such as tiling[IT88][WL91] to tune codes for better cache performance on uniprocessors. These optimizations improve the *locality of reference* of the code being optimized: they ensure that most memory references reach addresses (or cache lines) that have been used recently.

We have previously explored the limits of locality improvement[Won02], demonstrating that some codes could be made to exhibit *scalable locality*: a degree of locality that can be scaled up with some problem size parameter (typically the number of time-steps executed in an iterative calculation). Essentially, scalable locality indicates we may be able to exploit a processor that out-paces its main memory system to any arbitrary degree, just as scalable parallelism indicates we may be able to exploit an arbitrarily high degree of parallelism. For many codes, scalable locality is only exhibited after tiling of time-step loops, which typically requires skewing and tiling of imperfect loop nests, and was thus not generally exposed by the compilers of the 20^{th} century[Won02].

1.3 Combining Scalable Parallelism and Scalable Locality

Most locality optimizations produce chunks of code (or of iterations of a loop body) that perform many operations on a limited number of data elements. In principle, this can benefit parallelization by grouping together data and computations that should be placed on the same processor. Unfortunately, the scales on which computations must be "chunked" for cache performance are quite different from what is needed for effective parallelization, since network bandwidth may be hundreds or thousands of times less than main memory bandwidth. Most cache optimization techniques thus improve data re-use by too small a factor to effectively distribute data and computation around a parallel system.

Optimizations designed for scalable locality can be scaled up to handle the very low bandwidth of communication networks. In prior work, we argued that a scalable locality optimization, modified to take into account concurrent execution, could be used to simultaneously counter the effects of arbitrarily slow memory and inter-processor communication[Won00]. Although we lacked the necessary infrastructure to test this claim at the time, modern research compilers such as Pluto[KBB+07][BBK+08][BHRS08] can perform one version of this optimization. Thus, Pluto and ClOMP together let us try out this approach without major infrastructure development[DW09].

2 A Simple Example

Stencil codes are often used to illustrate optimization techniques that combine scalable parallelism with scalable locality in code that is not embarrassingly parallel. In particular, many papers [Won02][Won00][KBB+07] begin with code for explicit solution of the one-dimensional heat equation or its two-dimensional analog[SL99]. The one-dimensional stencil has the advantage of being very short and having an iteration space that can be graphed clearly on a (two-dimensional) page, while still illustrating some of the fundamental challenges of more complex codes: it is typically presented as a "imperfect loop nest", with two `i` loops inside the `t` loop. Even more importantly, there is a fixed upper bound on the potential improvement in data re-use within one time step, but no fixed upper bound on re-use when optimizing the entire nest.

Figure 1 gives one form of this code and, to its right, illustrates for $T = 6$ and $N = 9$ the pat-

```
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=0; i<N; i++) {
  A[i]=new[i];
 }
}
```

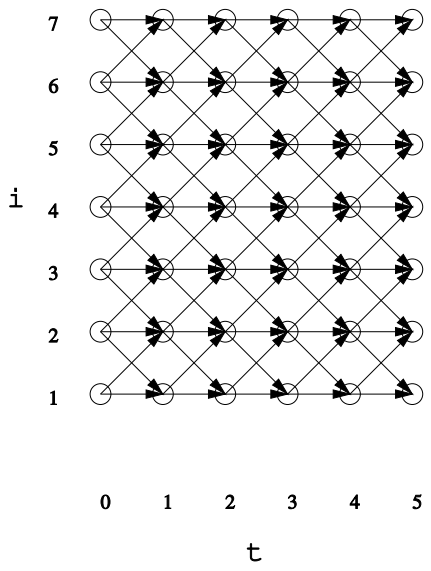

Figure 1. One-dimensional Jacobi Stencil Code (left) and Inter-Iteration Data Flow (right)

tern of data flow among the various executions of the statement that defines `new[i]`. For example, iteration [2, 4], uses values from [1, 3], [1, 4], and [1, 5], and thus there are arrows from these three iterations to [2, 4]. (Other forms of data dependence are discussed in [Won02].)

The flow of data fundamentally limits our ability to distribute and re-order the calculations: iteration [t, i] cannot be executed until the values from iterations with arrows pointing to [t, i] have been computed and (if necessary) sent to the processor performing this computation. We cannot take advantage of various approaches that would yield high performance for embarrassingly parallel code, for example, computing all iterations where $i = 1$ before doing any iterations where $i = 2$, or assigning one processor to compute iterations where $i = 1$ without communicating with the processor working on those where $i = 2$.

Figure 2 gives simplified illustrations of two ways to collect groups of iterations for parallel execution for the case of $T = 6$ and $N = 16$, with order of execution proceeding from darker to lighter groups. These two groupings will be used below to illustrate some of the differences between intra-time-step and inter-time-step tilings.

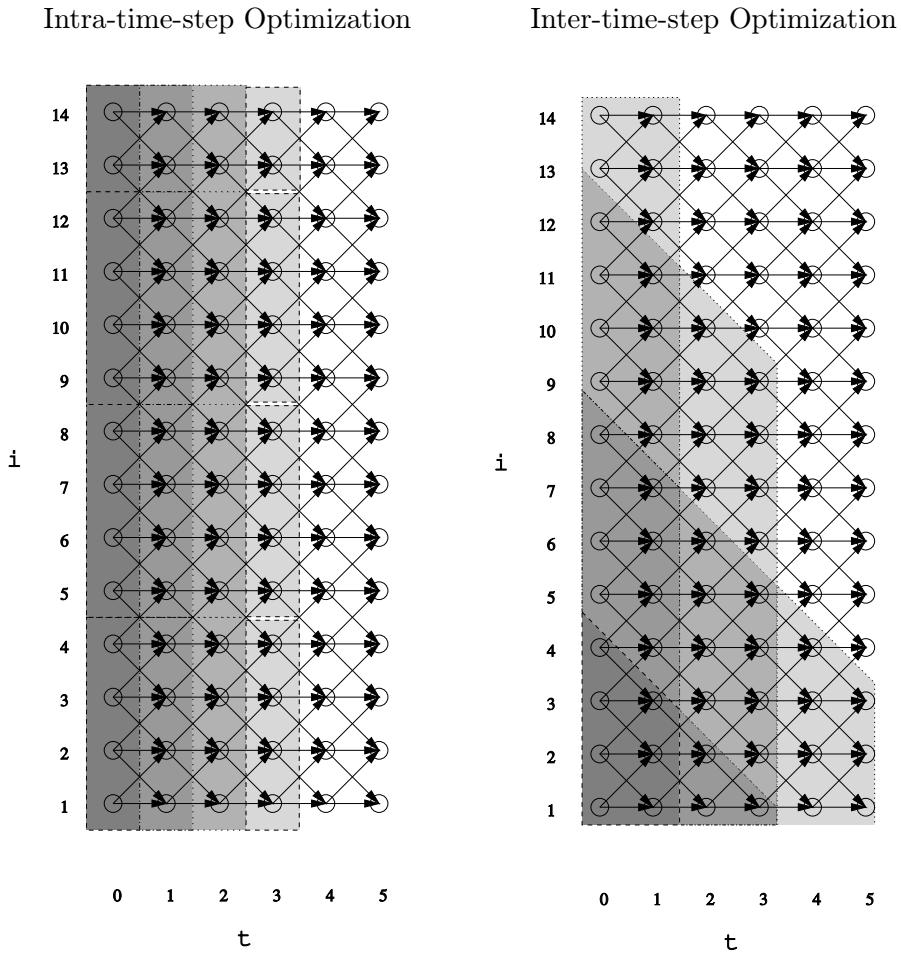

Figure 2. Tiling the Iteration Space of Figure 1 Without (left) and With (right) Time Tiling.

2.1 Intra-time-step Optimization

Prior to about 1997, loop optimizations were primarily applied within a time step, because inter-time-step optimization typically involves imperfectly nested loops that lie outside the domains of classic analysis and transformation techniques. To execute each time step of our example code on P processors (or processor cores), we could collect P groups of about $\frac{N}{P}$ iterations, each of which will be executed on one processor. The left side of Figure 2 shows contiguous groups for $P = 4$ — the four groups enclosed in dashed borders, on the darkest background, can be executed concurrently on four processors. For example, one processor executes iterations 5...8 for `t` $= 0$, while another meanwhile does iterations 9...12 for `t` $= 0$. Each processor can execute the same iterations of `i` for the next time step once it has completed its work and has access to the updated values from its neighbors. If the data for $\frac{N}{P}$ iterations fit in cache, we might hope that this tiling would achieve good cache use.

Unfortunately, inter-processor communication imposes costs of either message-passing (on a distributed memory system) or synchronization and motion of data to and from cache (for a shared memory system). Thus, this "intra-time-step" approach should work well only when these costs are very low or when they can be hidden — the latter might be achieved by first executing the border iterations (5 and 8, in our example) and then returning to execute the core iterations (6 and 7) while the border values are propagated. This approach requires that there be enough

core iterations to keep the processor busy, so $\frac{N}{P}$ must be large for systems with high communication cost. However, for large $\frac{N}{P}$, the data will not fit in cache. We may therefore be forced to choose between waiting for memory or waiting for communication — we cannot simultaneously avoid both both sources of delay with this tiling.

2.2 Inter-time-step Optimization ("Time Tiling")

Even on a uni-processor, inter-time-step optimization is necessary to achieve a high degree of locality when our example code is run on a large data set[Won02]. Tiles that cover σ iterations of the `i` loop and τ of the `t` loop involve $O(\sigma\tau)$ operations on $O(\sigma)$ memory locations. When τ is fixed at 1, re-use within a tile is limited, and if each tile's data must be fetched from a slow memory system, performance suffers. Only by increasing τ can we hope to improve intra-tile data re-use. Careful orchestration of the execution of iterations within the tiles lets us re-use addresses $O(\tau)$ times before displacing them from cache, producing dramatic speedups for large τ when memory bandwidth is very low[Won02]. Song and Li demonstrated the value of such inter-time-step locality optimization for a variety of benchmarks on hardware with better memory bandwidth[SL99].

The right side of Figure 2 illustrates an inter-time-step tiling for $\sigma = 4$ and $\tau = 2$; we use the term *time tiling* to refer to this general approach; variants have been explored by many authors, including [SL99][Won02][KBB+07]. As has been noted[Won00][KBB+07], these tiles can be executed concurrently if multiple processors are available. At first glance, this may not seem very appealing: the tile boundaries are more complicated, and only one processor can work initially. However, for large values of N and T, all processors will be busy for the majority of the computation. Furthermore, it is possible to have all processors start simultaneously by enclosing some iterations in multiple tiles (this corresponds to redundant work on several processors)[Won00][BHRS08], or by building a set of "molecular tiles" out of several atomic sub-tiles[Won00] using iteration-space slicing[PR99].

Time tiling can produce code that exhibits both scalable locality (as discussed above) and scalable parallelism: Asymptotically, if we double N, we double the degree of parallelism for tiles of height σ. If we double τ (which may require large T), we roughly double the level of intra-tile re-use (doubling the level of re-use is the limit as σ increases). Furthermore, each tile communicates $O(\tau)$ values to the neighbor above it and $O(\sigma)$ values to the neighbor to its right, and performs $O(\sigma\tau)$ operations. Thus, by increasing the tile size parameters, we can achieve an arbitrarily high balance of computation to communication cost. In principle, this can compensate for arbitrarily slow inter-processor communication *if* N and T are sufficiently large, as long as cache can hold all the values for 3τ iterations[Won00].

3 Tile Size Selection and Performance Models

Wonnacott initially focused on bounding tile sizes to ensure tile workload matches machine parameters[Won00]. Our initial experiments found that this is sufficient to hide memory access and inter-node communication time, for example giving performance from two single-core nodes connected over the network that matched performance for a single dual-core system. However, performance for both systems dropped off more quickly than we expected for smaller problem sizes. By taking into account the load imbalance created by synchronization at the end of OpenMP parallel loops, we are able to account for the observed drop in performance for smaller problems.

3.1 Lower Bounds on Tile Size

For stencil calculations, Wonnacott gave formulae to determine lower bounds on tile sizes for which the time needed to perform all computation within a tile at least equals the time needed for data transfers with memory and other processors[Won00]. This gives the minimum size for which we could hope to achieve full CPU utilization under the most optimistic assumptions about re-using cache (or registers) for temporary values, hiding memory latency, and overlapping computation with inter-processor communication. The formulae are all based on the assumption that floating-point operations and data traffic dominate all other factors.

Several parameters describing the hardware are required for the formulae:

C. The compute speed in MFLOPS

B. The main memory bandwidth, in MBytes/sec

$\boldsymbol{B_N}$**.** The network bandwitdth, in MBytes/sec

L. The network latency, in microseconds

Several parameters are also needed to describe the work done in each iteration of the loop:

O. The number of FP operations in one loop iteration

D. The number of bytes of FP data live-out from each iteration

Tile size parameters are labelled as above:

$\boldsymbol{\tau}$**.** The "width" of the tiles in the time dimension

$\boldsymbol{\sigma}$**.** The "height" of the tiles in the spatial dimension

The formulae of [Won00] are based on the use of "molecular tiles" that are not supported by Pluto (and have not been empirically shown to be valuable). However, we can apply the same insights to the tiling we do use: *for full CPU utilization in the absence of inter-tile locality, it is necessary that the total computation time for each tile exceed both the total memory acces time and total communication time for the tile.*

For the one-dimensional stencil of Figure 1, each tile that does not intersect the edge of the iteration space requires $\frac{O}{C}\tau\sigma$ microseconds of computation. Each such tile communicates $D\sigma$ bytes to the tile to its right (in Figure 1), and $2D\tau$ bytes to tile above it and to its upper-right. For large values of N and T, most tiles will not intersect the edge of the iteration space, so we optimize for these "middle tiles".

If we assign each processor a band of tiles that runs from upper-left to lower-right, the $D\sigma$ bytes that are communicated to the right will stay in main memory, and the $2D\tau$ will be sent to a neighboring processor (after being sent to memory). This requires $\frac{2D(2\tau+\sigma)}{B}$ μS of memory traffic time (to read and write these values) and $\frac{2D\tau}{B_N} + L$ μS of network time. Thus, to achieve full CPU utilization, it is necessary that $BO\tau\sigma \geqslant 2CD(2\tau+\sigma)$ and $B_NO\sigma\tau \geqslant 2CD\tau + CB_NL$.

A two-dimensional analog of Figure 1 (shown in Figure 3) can be time-tiled for one level of parallelism or for two. The former is in some ways simpler, but the latter provides parallelism that scales linearly with the problem size (both scale with N^2). For one level of parallel loop, a simple insight will help us deduce the conditions under which each tile's computation time at least equals its memory access and communication times: we simply view it as a one-dimensional stencil on vectors of N real numbers. In other words, instead of viewing each circle in Figure 1's graph as the execution of a single update to `A[i]`, we view each circle as an update of all elements of `A[i, 0:N-1]`. In this view, we can re-use the formulae from the one-dimensional stencil with values of O and D that are functions of N.

```
for (t=0; t<T;   t++) {
 for (i=1; i<N-1; i++)
  for (j=1; j<N-1; j++)
   new[i][j]=(A[i-1][j]+A[i+1][j]+A[i][j]+A[i][j-1]+A[i][j+1])*0.2;
 for (i=1; i<N-1; i++)
  for (j=1; j<N-1; j++)
   A[i][j]=new[i][j];
}
```

Figure 3. Five-Point Two-Dimensional Stencil Code

To tile a two-dimensional stencil for two levels of parallelism, we tile the two spatial dimensions. We are currently using the same tile size (σ) for each spatial dimension. Each "middle tile" in this space requires $\frac{O}{C}\tau\sigma^2$ microseconds of computation. Each such tile communicates $D\sigma^2$ bytes to the tile to its right, and $2D\tau\sigma$ bytes along each of the 2 spatial dimensions. Thus, we requires $\frac{2D\sigma^2+8D\tau\sigma}{B}$ μS of memory traffic time and $\frac{4D\tau\sigma}{B_N} + L$ μS of network time, assuming once again that each processor is scheduled to take a diagonal band of tiles. To achieve full CPU utilization under these conditions, it is necessary that $BO\tau\sigma^2 \geqslant 2C\,D\sigma^2 + 8C\,D\tau\sigma$ and $B_N O\tau\sigma^2 \geqslant 4C\,D\tau\sigma + CB_NL$.

These formulae are all based on the full multiprocessor time-skewing algorithm[Won00], which includes several features not present in Pluto at this time. Pluto does not perform storage-remapping transformations to control memory traffic from temporaries, but we can compensate for this by treating `new[i]` as well as `A[i]` as live-out from an iteration of the stencil (since `new[i]` will also be stored to main memory, even though `new` is not live-out from the full loop nest). This corresponds to simply doubling D.

3.2 Cache Reqirements, Multi-level Tiling, and Overhead

The above lower bounds on tile size are necessary but not sufficient for full CPU utilization. Full performance can of course be inhibited by factors such as excessive memory traffic from temporary values, cache interference, cache size limits, and integer/control-flow instructions for things like subscript computation and loop overhead.

Our doubling of D accounts for one read one and write for each element of the tile's section of `new` as well as `A`. To keep from rising above this level of memory traffic for large values of σ, we must traverse the iterations within each tile with a diagonal "wavefront" as shown in [Won02] and elsewhere (for example, working from lower left toward upper right of each tile in Figure 1). If the cache can hold several wavefronts worth of values and there is no cache interference, each location will be read and written only once. Interference can be avoided with the storage-remapping technique of [Won02] and [Won00] or a cache of sufficient associativity.

The number of wavefronts that must be held in cache is controlled by the pattern of dependences; for our examples it is just over two. This means a cache size of just over $2D\tau$. In higher dimensions, the cache requirement for stencil computations grows linearly with τ and all but the largest of the tile dimensions, i.e., exponentially with the number of dimensions, but not with the total problem size (N or T).

Cache size thus places an upper bound on all but one of the tile dimensions. When the lower bounds found described in Section 3.1 are below the upper bounds imposed by cache size limits, we can make use of multi-level tiling. The lower bounds from memory bandwidth are typically far below those from the network, and we can satisfy memory bandwidth and cache size constraints for an inner level of tiles, and produce outer level (parallel) tiles that satisfy the network constraints.

We do not directly address issues of the overhead from non-floating point instructions in inner loops, as these are essentially the same for the parallel and uniprocessor codes. We can accomodate these effects to some degree by using a lower value for C in our calculations of minimum tile size, as long as C does not fall below the uniprocessor floating-point computation rate for the tiled code. In our experiments, we also use a value of B that is based on a run of the uniprocessor code — in this case, we run the original (un-tiled) code on a large data set, as this simple code will produce the most frequent stream of requests involving main memory. After making these adjustments, we attempt to minimize loop overhead by choosing a large value for the dimension corresponding to our innermost loop (but not a value so large as to overflow the L1 cache), and setting other parameters according to our constraints.

3.3 Load Balance Issues in OpenMP

The grouping of work into tiles, and execution of tiles along a diagonal wavefront, each interfere with our quest for full efficiency. The reasons are most easily seen in a diagram of a small example. Consider what happens when we use three cores to run the three-point stencil with an iteration space as tall as six tiles level two tiles. We might hope that, since each time step is six tiles high, we would end up with each of the three cores running two full tiles in each wavefront.

However, the actual situation is much worse, as we can see in as in Figure 4. Each core (identified by letter) starts working on a tile from a given wavefrount (identified by number) once no core is executing any tile of a previous wavefront. Initially there is significant imbalance because the first few wavefronts do not have three full tiles of data, so some cores remain idle.

Time Step 4 is the first time in which no core is idle at any time. Each of the tiles is the same size, and each core can execute one tile. However, Step 5 takes nearly twice the time as Step 4. Core B has to calculate two tiles each of which are close to full size. While Core B is working through its second tile, Cores A and C remain idle. One might expect three cores to calculate four tiles in $\frac{4}{3}$ as much time as Step 4. However, because the tiles each have to be calculated fully by only one core, the speedup is less.

As the diagonal strips increase in size to full height at time 6, the efficiency changes, but does not reach 100%: during the time that Core A takes to complete two tiles, Core C has to complete only one tile. Core C will be idle for half the time spent in time step 6. However, Step 6 is more efficient than Step 5, for Core B will be completing more than one tile, almost two full tiles, thus staying idle for only a short period of time. Continuing in this fashion, through Step 7 which takes place on a diagonal with four complete tiles and two smaller tiles, each core will complete more than one tile making Step 7 more efficient than Step 6.

For the one-dimensional stencil, the impact of load balancing on efficiency can be determined through a few simple calculations. Given N, T, σ, τ, and the second-level tile sizes σ' and τ', one can determine the number of tiles in a wavefront. The distance in the `i` dimension from the top of one tile to the top of the next tile in the wavefront (e.g. the top of A6 to the top of B6 in Figure 4) is $\sigma\sigma' + \tau\tau' \sin(\theta)$, where θ is the angle between the vertical and diagonal lines (in Figure 4). $\frac{N}{h}$ is therefore the number of tiles in a wavefront, ν. With c as the number of cores in

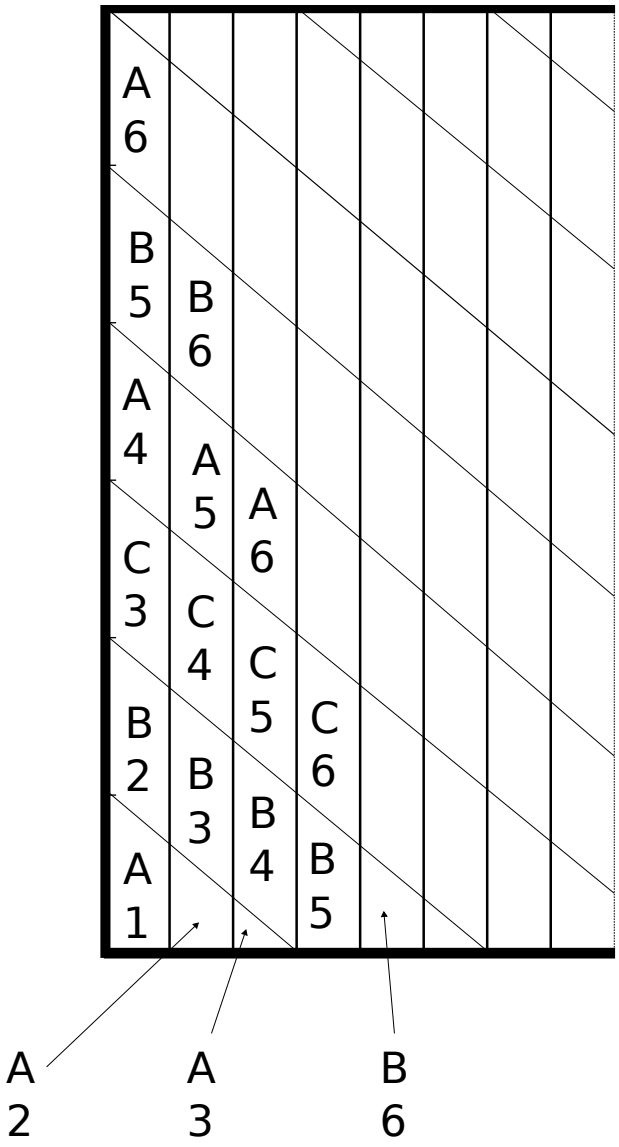

Figure 4. Load Imbalance and Execution of Second-Level Tiles

the system, "perfect" efficiency (citing only this specific balancing issue) occurs when $\nu \bmod c = 0$. As this modulo approaches 1 from above, efficiency gets worse because an increasing number cores sit idle. At the worst-case $\nu \bmod c = 1$ value, every core except for one will be idle for the maximum amount of time, i.e., one tile worth of computation. As $\nu \bmod c$ drops from 1 toward 0, efficiency increases because the amount of time spent with $c - 1$ processors idle is reduced.

4 Experimental Setup and Predictions

We have access to a small cluster with relatively modern multicore CPU's. (In early experiments with multicore systems based on the "Pentum D" architecture we had trouble getting good parallel results on even a single node.)

4.1 System Hardware and Software

We performed our experiments with four Dell Optiplex 745s, circa early 2007. The CPU in each machine is a dual-core x86-64 Intel Core 2 Duo E6700 microprocessor, running at 2.66GHz. The Core 2 micro-architecture contains a number of extensions to the x86 instruction set; the E6700 specifically supports MMX, SSE, SSE2, SSE3, and SSSE3 instructions. SSE is notable in that it adds single-precision floating point SIMD instructions, in addition to the SISD x87-compatible ones. SSE2 further extends SSE, by offering double-precision[Int09]. Each SSE register is 128 bits in length, so it can fit 2 double-precision floating point numbers. Hence, if vectorized appropriately, each core is capable of two floating point instructions per clock cycle. Therefore, each core is theoretically capable of 2.66GHz * 2 floating-point instructions/cycle.

Our tile size bounds, and the "linear speedup" calculations in our results, are based on the peak speed of the uniprocessor code that was fastest for large values of N. We compared the original code, the time-tiled code, and a version on which we performed manual alignment and fusion of the loops (see Figure 5). This last version is somewhat less cache intensive; it is our attempt to produce a code that minimizes memory traffic using "traditional" loop transformations (i.e., neither time tiling nor storage remapping).

As shown in Figure 6, the time-tiled code is by far the fastest for large problem sizes. It produces roughly 2380 MFLOPS/core for a wide range of N. The original code and that of Figure 8 were each slightly faster for small data sets but far slower for large N. Thus, we use $C = 2380$ in our calculations.

Each core contains two L1 caches, each 32KB in size and 8-way associative, one for instructions and one for data. The total L2 cache size for both cores is 4MB in size, and the L2 cache is 16-way associative. Our systems have no L3 cache. The motherboards are based on Intel's 965 chip set. Each contains 4GB of memory (as 4 interleaved 667MHz 1GB DDR2 modules), and is connected to the processor via a so-called "1066 MT/s" (megatransfers per second) front-side bus, which gives a peak theoretical bandwidth of 7.9 GB/s.

Note that we can predict the uniprocessor CPU utilization by setting $\tau = 1$ and using the formula from [Won00] to predict the performance of the original (untiled) code for large N. This gives $\frac{OB}{2DC} = \frac{3*7900}{2*16*2380} \approx 31\%$ efficiency, or 740MFLOPS if the system can maintain its theoretical peak memory bandwidth. The hardware actually produces about 400MFLOPS for the original code (Figure 6), which would correspond to $B \approx 4300$, or just over half of peak theoretical efficiency. Thus we use $B = 4300$ in our calculations.

In various experiments we connected the machines via a 10/100BaseT "fast Ethernet" switch or a "gigabit Ethernet" switch. Performance measurements from the NetPIPE tool[SMGH] show the former gives us approximately 88Mbps bandwidth (B_N) with 62 microsecond latency (L), and the latter 940Mbps bandwidth with 46 microsecond latency.

We used the Intel C++ Compiler ("icc"), version 10.1, build 20090203, because it provides the ClOMP framework and performs automatic vectorization; we believe it provides the highest single-core performance for the code and chip set combination used in our experiments. We used PLUTO version 0.4.1[Bon] to perform time tiling for parallelization and single-processor cache optimization. Because ClOMP requires a 2.3.x version of glibc, we used an older release of Debian GNU/Linux, version 4.0 "etch" (with a 64-bit userland) which provides a (heavily patched) glibc version 2.3.6.ds1-13etch9, and a 2.6.18 kernel.

```
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);
    A[i-1]=new[i-1];
  }
  A[(N-1)-1]=new[(N-1)-1];
  A[ N-1   ]=new[ N-1   ];
}
```

Figure 5. Figure 1 After Hand-application of Imperfectly Nested Loop Fusion.

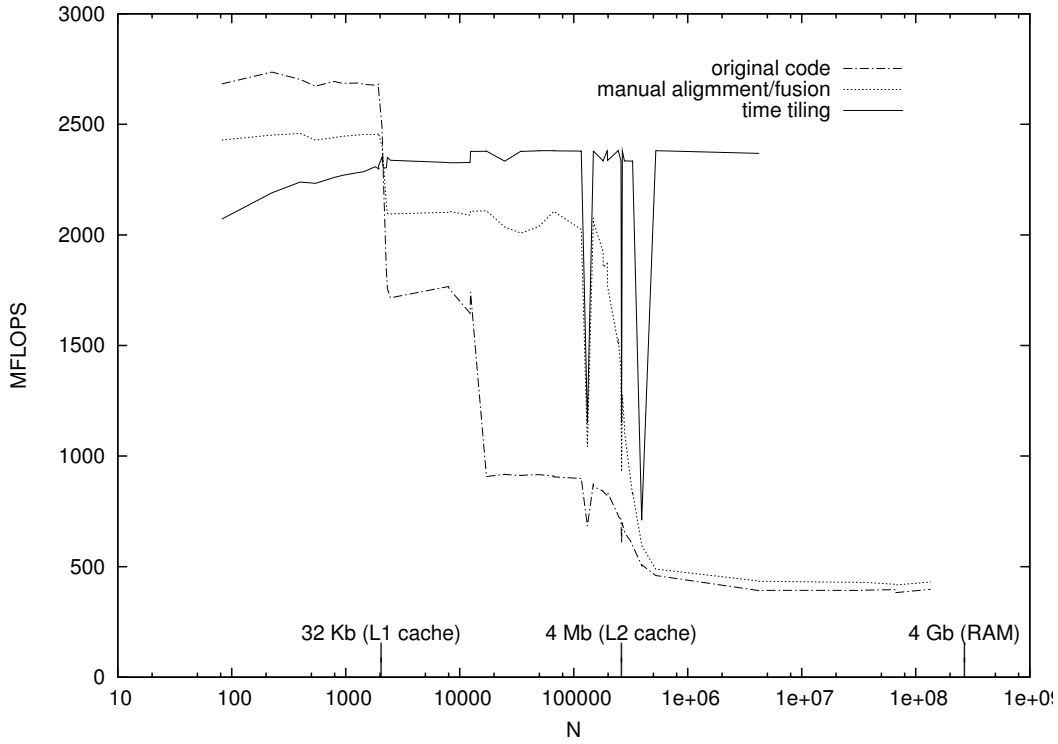

Figure 6. Single-core Processing Speeds of Three Uniprocessor Codes for Larger Data Sets

Our tile sizes for the time-tiled code in Figure 6 and our cluster computing experiments were based on executions of the original code with various data set sizes. Figure 7 shows the uniprocessor speed achieved for various values of N, using the original code compiled with icc only. Unsurprisingly, the speed is generally highest for $N < 2048$, which corresponds to data set sizes that fit in the 32KB L1 cache. We speculate that the reduced performance at the 8KB intervals corresponds to cache interference effects, despite Intel's claim of 8-way associativity. The peak speed of 2785 MFLOPS was achieved at $N = 96$, but speeds above 2700 MFLOPS occur almost all the way to the $N = 2048$ boundary. We used Figure 7 to select several promising tile sizes in the uniprocessor time-tiled code and found many that led to good performance. For example, the time-tiled code in Figure 6 was based on $\tau = 1024, \sigma = 1907$. Since the 2380 MFLOPS maintained by time-tiled code is over 85% of the peak 2785 MFLOPS of this low-loop-overhead code, we do not believe there is room for dramatic improvement in uniprocessor speed without extensive hand-tuning of the code.

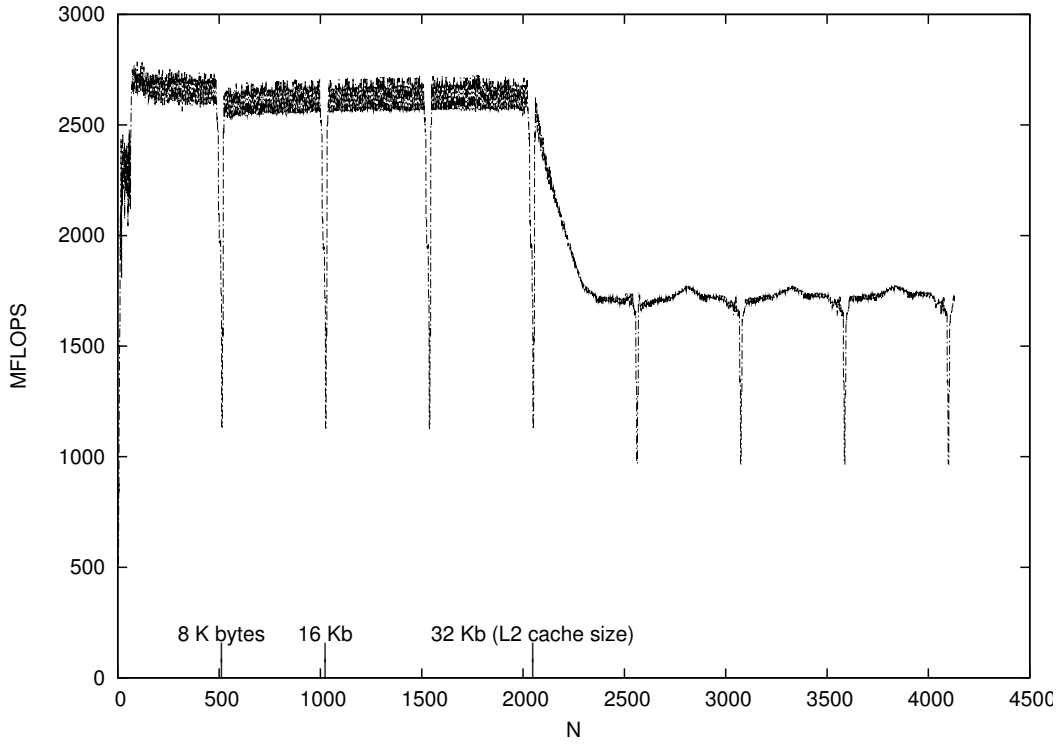

Figure 7. Single-core Processing Speed of Original Code as a Function of Data-Set Size (N)

4.2 Predictions of Minimum Tile Sizes

For the three-point stencil, $O = 3$ and $D = 8$ (one double-precision value), but for our analysis we use $D = 16$, as descussed in Section 3.1.

Plugging these numbers and our hardware parameters into the equation for the one-dimensional stencil, we find we need $BO\tau\sigma \geqslant 2DC(2\tau + \sigma) \Rightarrow 4300 * 3\tau\sigma \geqslant 2 * 8 * 2380 * (2\tau + \sigma) \Leftrightarrow \tau\sigma \geqslant 5.90\tau + 2.95\sigma$ to avoid the memory bottleneck. While this holds for fairly small tile sizes such as $\tau = 4, \sigma = 24$, or $\tau = 21, \sigma = 7$, tiles of this size may incur excessive loop overhead, and do not fit the network constraints. Setting $B_N O\tau\sigma \geqslant 2CD\tau + CB_N L \Rightarrow 88 * 3\tau\sigma \geqslant 2 * 2380 * 16\tau + 2380 * 88 * 62 \Rightarrow \tau\sigma \geqslant 288\tau + 49200$ will avoid the network bottleneck for fast Ethernet, and $\tau\sigma \geqslant 27\tau + 36500$ for gigabit Ethernet. These bounds and the memory constraint allows solutions for $\tau = 4$, $\sigma \geqslant 12600$, for $\tau = 10, \sigma \geqslant 5210$, and for $\tau = 32, \sigma \geqslant 1830$ on fast Ethernet, and solutions for $\tau = 4, \sigma \geqslant 9150$, for $\tau = 10, \sigma \geqslant 3680$, and for $\tau = 32, \sigma \geqslant 1170$ on gigabit Ethernet. As long as $\tau < 1024$, the cache requirement is below our system's L1 cache size.

For a two-dimensional five-point stencil that is tiled for one level of parallelism (as described in Section 3.1), we use $O = 5N$ and $D = 16N$ to predict the performance of Pluto's code on our cluster. The factors of N mostly "cancel out" in our calculations, leaving the difference of $\frac{5}{3}$ for O in most terms, requiring $\tau\sigma \geqslant 3.54\tau + 1.77\sigma$ for the memory balance inequality, and $\tau\sigma \geqslant 186\tau + \frac{57100}{N}$ for the fast Ethernet inequality. Without information about N, these constraints produce tiles of $\frac{3}{5}$ the size of the three-point stencil, but if we assume $N \geqslant 1024$, then $\tau = 4, \sigma \geqslant 181$ and $\tau = 32, \sigma \geqslant 174$ are allowed. For gigabit Ethernet and $N \geqslant 1024$, $\tau = 4, \sigma \geqslant 62$ and $\tau = 32, \sigma \geqslant 17$ are allowed.

For two levels of parallelism on the five-point stencil, our constraints allow solutions such as $\tau = 4, \sigma \geqslant 367$ and $\tau = 25, \sigma \geqslant 350$ for fast Ethernet, and $\tau = 4, \sigma \geqslant 75$ and $\tau = 25, \sigma \geqslant 50$ for gigabit Ethernet.

For either tiling of the two-dimensional stencil, the smaller spatial dimension is σ, so we need $\tau\sigma < 1024$ for our result to fit in cache. This is not the case for many of the sample solutions above, but conveniently for all these cases the memory bandwidth constraints are much looser, allowing $\tau = 4, \sigma \geqslant 62$ and $\tau = 32, \sigma \geqslant 8$ for one level of parallelism (if $N \geqslant 1024$) and $\tau = 4, \sigma \geqslant 124$ and $\tau = 10, \sigma \geqslant 22$ and $\tau = 25, \sigma \geqslant 17$ for two.

5 Experimental Results

We have performed experiments to test the feasibility of our goal: achieving good performance on a cluster of workstations by combining software distributed shared memory with automatic exposure of scalable parallelism and scalable locality. We also accidentally verified the load imbalance described in Section 3.3. The discussion below draws upon the preliminary work we presented at MASPLAS 2009[DW09], but shows significantly higher performance (for both uniprocessor and cluster codes) due to greater attention to tile size issues.

5.1 Experiments Performed

We compared a time-tiled version of the code of Figure 1 to two other parallel versions of this code, as well as three uniprocessor versions. One "comparison" parallel version was produced by simply adding a parallelization directive `#pragma omp parallel for shared(t) private(i)` in front of each of the `i` loops in Figure 1. The other, shown in Figure 8, involved hand-application of loop tiling for the `i` loops, which required prior alignment and fusion of loops with different bounds — this was the only possibly-useful intra-time-step loop optimization we could find that was not already being applied by icc. (For example, a quick experiment confirmed that icc is hoisting the computation of 1.0/3 outside of the loop nest.) All parallel versions of the code used ClOMP.

Our uniprocessor comparison codes included one example of time tiling, produced by invoking Pluto with the `--tile --l2tile` options but without the `--parallel` option. We also included the original code shown in Figure 1 and a manually-produced tiled parallel version of the code in Figure 5 (see Figure 8).

```
  for (t=0; t<T;   t++) {
#pragma omp parallel for shared(t) private(ib, i)
    for (ib=1; ib<N-1; ib+=SIGMA) {
      long int minn=(ib+SIGMA<N-1)? ib+SIGMA: N-1;
      for (i=ib; i<minn ; i++) {
        new[i]=(A[i-1]+A[i]+A[i+1])*(1.0/3);
        A[i-1]=new[i-1];
      }
    }
    A[(N-1)-1]=new[(N-1)-1];
    A[ N-1   ]=new[ N-1   ];
  }
```

Figure 8. Parallel Version of Figure 5 Tiled on Only the Spatial Loop.

For our five-point stencil, we once again compared three parallel versions of the code to the speed of the fastest uniprocessor code (the code that was time tiled by Pluto without parallel loops, which hit 2650 MFLOPS for large data sets). In addition to the time tiled code from Pluto and the original code with `omp parallel` pragmas on both spatial loops, we produced a "best intra-time-step-optimization" analog of Figure 8 by directing Pluto to process only the spatial loops. We have not yet produced a version of the two-dimensional stencil that is tiled in only one of the spatial dimensions.

We chose tile sizes that fit the constraints given in Section 4.2 and were consistent with the uniprocessor tile sizes selected in Section 4.1. For the three-point stencil, we used $\tau = 1024$, $\sigma = 1907$, with second-level tiling factors of 4 for `t` and 8 for `i`. We obtained similar results in experiments with $\sigma = 1811$ or 1804 or 1475, but varying the second level tiling factors affected uniprocessor and cluster performance significantly. For the five-point stencil, we used $\tau = \sigma = 32$ and second-level tiling factors of 2 at each level. For the non-Time-Tiled version of the code, the first- and second-level tile factors were combined to produce the value of `SIGMA` for Figure 8.

Having selected our tile sizes, we began our cluster computing experiments. For each interesting tile size, we ran each version of each code on both cores of 1, 2, 3, and 4 nodes. Since we are interested in exploring scalable parallelism, we focused primarily on large values of N and T and scaled up the total data set size with the total number of cores (i.e., looked for *weak* rather than *strong* parallel scaling). We ran the two-dimensional stencil experiments with both fast Ethernet and gigabit Ethernet despite the fact that our tiles were big enough to compensate for network speeds on either.

5.2 Results of Scalable Parallelism Experiments

Figure 9 shows the total system performance vs. degree of parallelism for large data sets ($N \approx 10^6$ per core for the three point stencil, and $N = 1000$ per core for the five-point stencil). Results for gigabit Ethernet experiments are shown only for the Time-Tiled code; for the other versions of this code the fast and gigabit Ethernet results were within 1% of each other.

Lines with points correspond to experimental data, with triangles for the three-point stencil and squares for the five-point, and line style indicating the optimization technique. Lines without data points are computed linear speedups of the best uniprocessor speed for the given code (2650 MFLOPS for the five-point stencil and 2380 for the three-point).

The performance in the absence of time tiling clearly illustrates the peril of trying to parallelize this code without paying attention to communication cost. All such lines are not only far below a linear speedup, they are generally only a small fraction of the best one-core version of the code (though some exhibit near-perfect linear speedup from the abysmal two-core case).

The Time-Tiled codes (solid lines) also exhibit approximately linear scaling, but with better constant factors: Time-Tiling+ClOMP achieves over 95% of linear speedup at 8 cores for the three-point stencil, and about 70% of linear speedup at 8 cores for the five-point stencil.

5.3 Results of Load Balance Experiments

An unsuccessful attempt to produce our results quickly led us to confirm the load balance issues of Section 3.3. Figure 10 shows one example of the three-point stencil run with a variety of values of N and tile sizes as in Figure 9. The solid lines correpsond to the Time-Tiled code; for comparison, the non-Time-Tiled code was also run with the same tile sizes and N. The fall-off in performance for the middle two Time-Tiled lines is completely explained by load balance effects discussed in Section 3.3; the lowest (flat) solid line corresponds to a data set so small that ClOMP executes all tiles on one node.

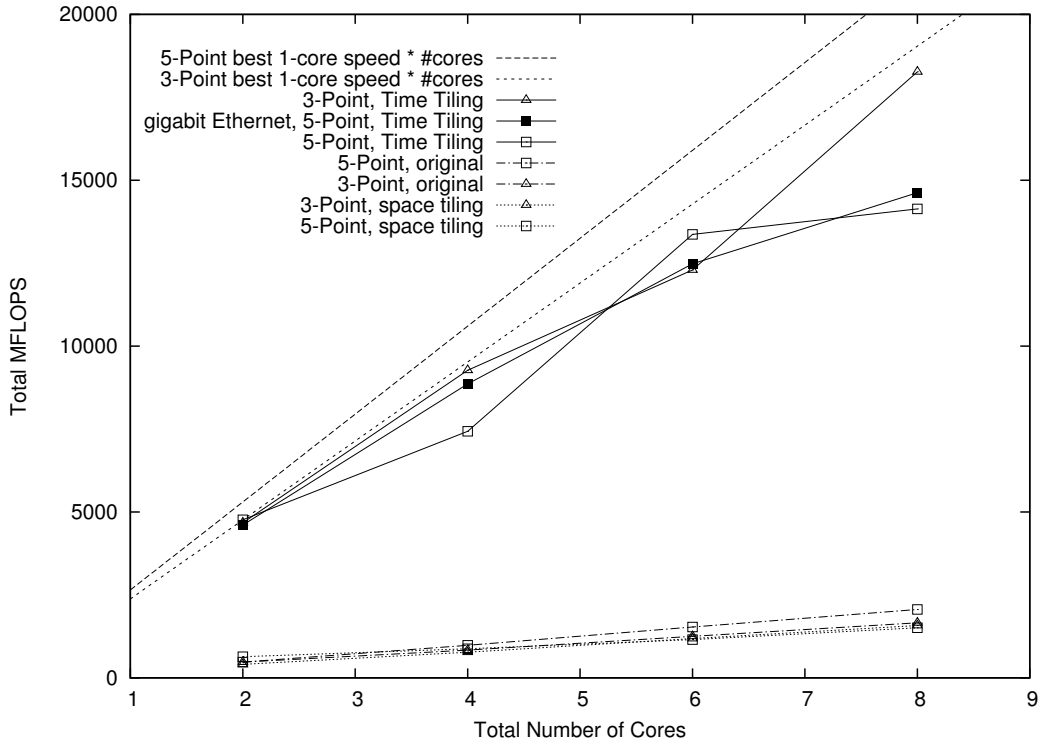

Figure 9. Performance of Various Parallel Codes vs. Linear Speedups.

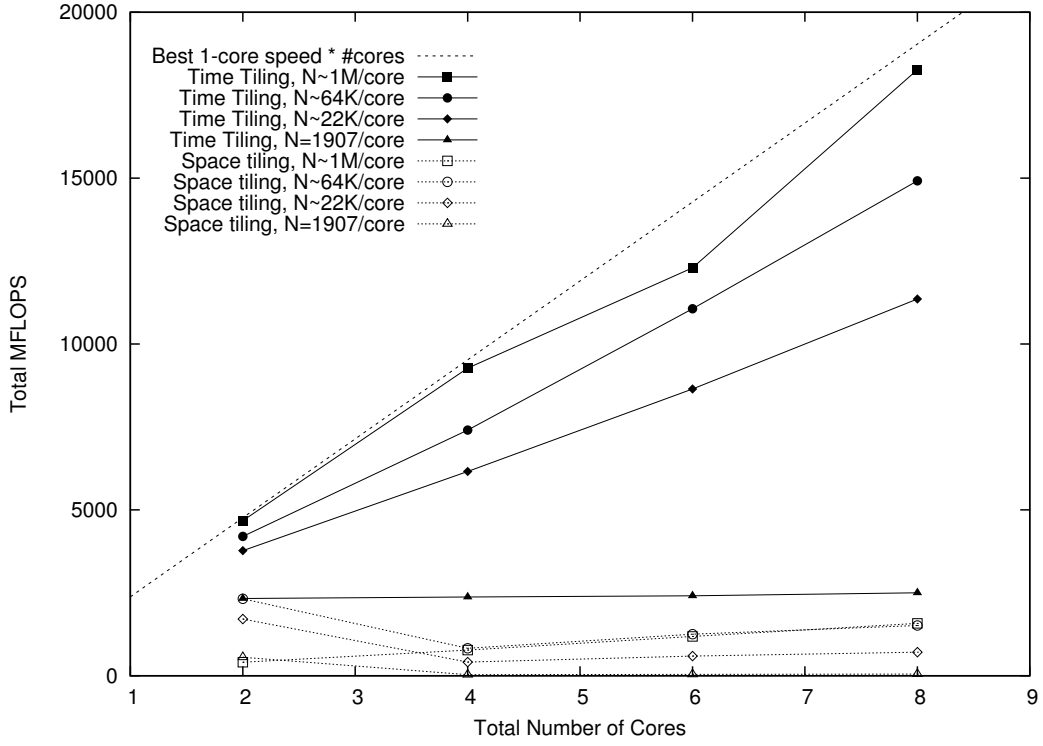

Figure 10. Total MFLOPS on Various Problem Sizes.

6 Related Work

Many of the great success stories for cluster computing involve applications that are embarrassingly parallel or novel algorithms designed for clusters, rather than acceleration of existing dependence-rich linear algebra codes. Success on a cluster typically involves coding a parallel algorithm in an explicitly parallel framework that allows programmer-control of communication, such as the explicit-message-passing paradigm used in MPI[Ope]. Some compiler work has been done to support MPI codes, such as that of Bronevetsky[Bro09], but this is not directly comparable to our work.

The SuperMatrix system[CZQO+07] 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[CZQO+07] 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[BME02][jMBE03][BE05]. Unlike our work, this approach presumes that the programmer has already identified 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 data set sizes, and thus obtained super-linear speedups in some cases[BE05].

For some problem domains, compilers have been developed to automatically generate the appropriate sends and receives from annotated sequential code (for example, Fortran D[HKT92]), 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[CG06]. 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 distributed memory code generator be integrated into each optimizing compiler.

Our focus on tile size selection is unusual in that we produce bounds on a range of acceptable sizes, whereas most approaches find a size that optimizes some cost function[RR08]. We are currently investigating the question of whether our tile size constraints could be integrated with an optimization approach through the general "posynomials" framework of [RR08].

7 Conclusions

For codes that can be made to exhibit scalable locality, Time Tiling can provide an optimiztion approach that can be tuned to give good performance on low-overhead shared memory machines or distributed memory systems with high communication costs. (This is analagous to the use of the uniprocessor Time Tiling as a tunable memory heirarchy optimization for L2 cache, RAM, and virtual memory[Won02].)

Cluster OpenMP is a valuable infrastructure for the exploration of these tunable techniques, as it allows the use of code generators for shared memory OpenMP systems on clusters. While we have not done an exact "head-to-head" comparison, Time Tiling plus ClOMP seem to outperform state-of-the-art distributed-memory code generators in some cases (though we believe that, with sufficient tuning, specialized distributed memory code generators should win out).

Synchronization barriers at the ends of OpenMP parallel loops can be a significant source of overhead for smaller problem sizes.

We look forward to expanding our results to include larger numbers of nodes, data sets that do not fit in RAM, and additional benchmarks.

Bibliography

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFPIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

[BBK+08] Uday Kumar Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J Ramanujam, Atanas Rountev, and P Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *CC 2008*, 2008.

[BE05] Ayon Basumallik and Rudolf Eigenmann. Towards automatic translation of openmp to mpi. In *Proc. of the International Conference on Supercomputing, ICS'05*, pages 189–198, 2005.

[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.

[BME02] Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann. Towards openmp execution on software distributed shared memory systems. In *ISHPC '02: Proceedings of the 4th International Symposium on High Performance Computing*, pages 457–468, London, UK, 2002. Springer-Verlag.

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

[Bro09] Greg Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. *Code Generation and Optimization, IEEE/ACM International Symposium on*, 0:1–12, 2009.

[CG06] M. Classen and M. Griebl. Automatic code generation for distributed memory architectures in the polytope model. *Parallel and Distributed Processing Symposium, International*, 0:243, 2006.

[CZQO+07] Ernie Chan, Field G. Van Zee, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert van de Geijn. Satisfying your dependencies with supermatrix. *Cluster Computing, IEEE International Conference on*, 0:91–99, 2007.

[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.

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

[HKT92] Seema Hiranandani, Ken Kennedy, and Chau-wen Tseng. Compiling fortran d for mimd distributed-memory machines. *Communications of the ACM*, 35:66–80, 1992.

[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.

[Hoe09] Jay P. Hoeflinger. *Personal communication*, May 2009.

[Int09] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, March 2009. http://download.intel.com/design/processor/manuals/253665.pdf.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 319–329, 1988.

[jMBE03] Seung jai Min, Ayon Basumallik, and Rudolf Eigenmann. Optimizing openmp programs on software distributed shared memory systems. *International Journal of Parallel Programming*, 31:225–249, 2003.

[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.

[Ope] Open MPI Project. Open MPI: Open source high performance computing. http://www.open-mpi.org/.

[PR99] William Pugh and Evan Rosser. Iteration slicing for locality. In *12th International Workshop on Languages and Compilers for Parallel Computing*, August 1999.

[RR08] Lakshminarayanan Renganarayana and Sanjay Rajopadhye. Positivity, posynomials and tile size selection. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[SL99] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. In *PLDI '99: Proceedings of the 1999 ACM SIGPLAN conference on Programming language design and implementation*, pages 215–228, New York, NY, USA, 1999. ACM.

[SMGH] Quinn Snell, Armin Mikler, John Gustafson, and Guy Helmer. NetPIPE - a network protocol independent performance evaluator. http://www.scl.ameslab.gov/netpipe/.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 1991.

[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.

Appendix A Relevant Code

This appendix contains the code and experimental setup necessary to recreate our results.

Our four machines were named `ws1`, `ws5`, `ws6`, and `ws7`. We set up `ws1` as an NFS server for the three other nodes, so `/home` could be shared, fulfilling Cluster OpenMP's requirement that all nodes have local access to the binary being executed at the same path. Also, we established password-less SSH keys, so the authentication necessary for job creation and communication would run silently and unattended. As mentioned before, we used `icc` 11.0 and Debian 4.0 "etch", because it provides the antiquated glibc necessary for Cluster OpenMP.

The system works as follows: a Python script (named `runner.py`, run on `ws1`) creates a few run-dependent files, then tiles (if desired), compiles, and executes the appropriate code. Specifically, it creates a `tile.sizes` file that tells Pluto the sizes of the appropriate tiles, a `decls.h` file with the sizes of the arrays and timesteps in the run, and a `kmp_cluster.ini` which tells Cluster OpenMP on which machines to distribute the job, the number of threads per node, communication method, etc.

In order to communicate easily and notate the different variations of runs (no tiling, space tiling, and space & time tiling), we adopted a few naming conventions. The space & time tiling codes are called "detergent" ("Look at how well our detergent product cleans the dirty t-shirt..."), the time tiling codes are called "orange juice" ("...versus being cleaned in orange juice..."), and the untiled codes are called "dirt" ("...and just being washed in dirt! Huzzah!").

The following subsections are the contents of `runner.py`, the appropriate one- and two-dimentional detergent stencil codes in C, and patches to convert them into orange juice and dirt. Hopefully the Python script is either commented or self-explanitory enough. As with much research code, this system is not a great example of best coding practices, but the current `runner.py` sure beats our old system — a Python script that generated a bash script, complete with horrible stacks of differing nested quotes and double quotes.

A.1 `runner.py`

```
#!/usr/bin/python2.5
#
# runner.py -- June 2009, Tim Douglas (tdouglas@haverford.edu)
# plutoize our [1,2]dheat.c and run the tests

# icc 11.0 not working? make sure to export LANG=C

import os
import sys
import math
import time
import datetime
import subprocess

#POLYCC_EXE = "/home/davew/software/pluto-0.4.2/polycc"
POLYCC_EXE = "/home/davew/software/pluto-0.5.0/polycc"
#ICC_EXE = "/opt/intel/cce/10.1.022/bin/icc"
ICC_EXE = "/opt/intel/Compiler/11.0/083/bin/intel64/icc"
COMM_PROTOCOL = "ssh" # or rsh

# use None for stdout
PLUTO_OUTPUT = open("/dev/null", "w")
ICC_OUTPUT = open("/dev/null", "w")

PLUTO_OPTIONS = '--tile --parallel --l2tile'

clean_up = True
```

```
nodes_list = (
"ws1", "ws6", "ws5", "ws7",
)

# which programs do we want to run?
program_names = ("2dheat",)

# how many nodes are we using?
num_nodes_list = (2,)#1,2,3,4,)

# how many cores per each node?
cores_per_node_list = (2,)

# should the N in input_values be total N,
# or N per core?
n_per_core = True

input_values = (
#       N/core          T

# 1d heat numbers
# dirt
#       (1000,          8*1024,),



        (1000,          16*1024,),

# 2d heat numbers
# detergent
#       (500,           64*1024,),
#       (1000,          16*1024,),
# orange juice
#       (500,           1024,),
#       (1000,          4*1024,),
# dirt
#       (1000,          256,),

#       (     1907, 128*1024*1024), #####
#       (  16*1024,  64*1024*1024), ## nsf grant request values
#       (  64*1024,  32*1024*1024), ##
#       (1024*1024,     1024*1024), ####
)


tile_sizes_list = (
# 1d heat
#       (1024,          1908,           8,      4),

#       (1024,          1907,           8,      4),
#       (954,           1908,           8,      4),
#       (512,           1170,           8,      4),

#       (1024,          1907,),
#       (954,           1908,),
#       (512,           1170,),


# 2dheat
# detergent
#       (32,    32,     32,),
        (32,    32,     32,     2,              2,              2),


# orange juice
#       (32,    32,     14,     14,),

# dirt
#       (0,),
)


def do_run(program, tile_sizes, N, T, num_nodes, cores_per_node):
        # create decls.h
        decls_h = open("decls.h", "w")
        decls_h.write("#define N (%sL)\n" % N)
        decls_h.write("#define T (%sL)\n" % T)
        decls_h.close()

        # create tile.sizes
        ts_file = open("tile.sizes", "w")
        for tile_size in tile_sizes:
```

```
                ts_file.write("%s\n" % tile_size)
        ts_file.close()

        # create kmp_cluster.ini
        kmp_cluster = open("kmp_cluster.ini", "w")
        kmp_cluster.write("--process_threads=%s " % cores_per_node)
        kmp_cluster.write("--processes=%s " % num_nodes)
        kmp_cluster.write("--hostlist=%s " % ",".join(nodes_list[:num_nodes]))
        kmp_cluster.write("--launch=%s\n" % COMM_PROTOCOL)
        kmp_cluster.close()

        # run pluto
        subprocess.call(POLYCC_EXE + ' ' + program + '.c ' + PLUTO_OPTIONS, shell=True, stdout=PLUTO_OUTPUT, stderr=PLUTO_OUTPUT)

        # compile it!
        subprocess.call(ICC_EXE + ' -m64 -cluster-openmp -xhost -O3 ' + program + '.par.c -o par', shell=True, stdout=ICC_OUTPUT,
stderr=ICC_OUTPUT)

        # start printing out information
        print int(time.time()),"\t",
        print program,"\t",
        for tile_size in tile_sizes:
                print tile_size,"\t",
        print N,"\t",
        print T,"\t",
        print num_nodes,"\t",
        print cores_per_node,"\t",
        print num_nodes*cores_per_node,"\t",
        sys.stdout.flush()

        try:
                # run the test
                subprocess.call("./par " + str(num_nodes*cores_per_node), shell=True)
        except KeyboardInterrupt:
                print "\n# killing processes at time %s..." % time.time(),
                sys.stdout.flush()
                for node in nodes_list:
                        subprocess.call('ssh %s "killall -q par"' % node, shell=True)
                print "done."
                sys.exit(1)

        # clean up after ourselves
        if clean_up:
                for file in ("par", "1dheat.par.c", "decls.h", "tile.sizes", \
                                "kmp_cluster.ini", "1dheat.par.cloog", "1dheat.pluto.c", \
                                "2dheat.par.c", "2dheat.par.cloog", "2dheat.pluto.c",
                                "2dheat.par2d.c", "2dheat.par2d.cloog", "gmon.out",):
                        try: os.remove(file)
                        except OSError: pass # file doesn't exist


############


print "# time | program | tile sizes | N | T | num_nodes | cores_per_node | total_cores | start_time | run_time | mflops | mflops/core"

for program in program_names:

        for N_min, T_min in input_values:

                for tile_sizes in tile_sizes_list:
                        N = N_min
                        T = T_min


                        for num_nodes in num_nodes_list:
                                for cores_per_node in cores_per_node_list:
                                        N_cur = N
                                        if n_per_core:
                                                N_cur *= num_nodes*cores_per_node
                                        do_run(program, tile_sizes, N_cur, T, num_nodes, cores_per_node)

print "# done!"
```

A.2 1dheat-detergent.c

```
/*
1dheat.c -- a simple 1D finite difference heat equation stencil calculation
designed to benchmark Cluster OpenMP and pluto
by Tim Douglas (tdouglas@haverford.edu) -- Summer 2008

compile with:
[i|g]cc -Wall -O2 1dheat.c -o 1dheat
*/

#include <math.h>   /* for sqrt */
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

/* get N (size of array) and T (num of timesteps)
 * from decls.h, generated by par_runner.py */
#include "decls.h"

#define OPS_PER_ITER 3

double A[N], new[N];

double cur_time(void)
{
        struct timeval tv;
        gettimeofday(&tv, 0);
        return tv.tv_sec + tv.tv_usec*1.0e-6;
}

int main(int argc, char *argv[])
{
        double run_time, mflops, total = 0, sum_err_sqr = 0;
        char chtotal = 0;
        int cores;
        long i, t;
        /* long long n_fp_ops = 0; */

        if (argc != 2) {
                fprintf(stderr, "usage: %s numcores\n", argv[0]);
                exit(-1);
        }

        cores = atoi(argv[1]);

        /* seed the PRNG with a constant so we can check/compare the results */
        srand(42);
        for (i = 0; i < N; i++)
                A[i] = rand() % 700;


        printf("%f\t", run_time = cur_time());
        fflush(stdout);

#pragma scop
        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);
                        /* n_fp_ops = n_fp_ops + OPS_PER_ITER; */ /*  just a check */
                }
                for (i = 1; i < N-1; i++) {
                        A[i] = new[i];
                }
        }
#pragma endscop

        run_time = cur_time() - run_time;
        mflops = (((double)OPS_PER_ITER*(N-2)*T) / run_time) / 1000000L;

        printf("%7.2f\t%7.2f\t%7.2f  \t", run_time, mflops, mflops / cores);

        /* printf("counted %lld FP ops, expected %lld\t", n_fp_ops,
                ((long long)OPS_PER_ITER)*(N-2)*T); */

        printf("\n# ");
        for (i = 0; i < N; i++)
```

```
                total += A[i];
        printf("sum(A) = %7.2f\t", total);
        for (i = 0; i < N; i++)
                sum_err_sqr += (A[i] - (total/N))*(A[i] - (total/N));
        printf("rms(A) = %7.2f\t", sqrt(sum_err_sqr));
        for (i = 0; i < N*sizeof(A[0]); i++)
                chtotal += ((char *)A)[i];
        printf("sum(rep(A)) = %d\n", chtotal);

        return 0;
}
```

A.2.1 1dheat-detergent-orange_juice.patch

```
--- 1dheat-detergent.c  2009-06-12 13:39:44.000000000 -0400
+++ 1dheat-orange_juice.c       2009-07-28 15:11:47.000000000 -0400
@@ -51,8 +51,8 @@
        printf("%f\t", run_time = cur_time());
        fflush(stdout);

-#pragma scop
        for (t = 0; t < T; t++) {
+#pragma scop
                for (i = 1; i < N-1; i++) {
                        new[i] = (A[i-1] + A[i] + A[i+1]) * (1.0/3);
                        /* n_fp_ops = n_fp_ops + OPS_PER_ITER; */ /*  just a check */
@@ -60,8 +60,8 @@
                for (i = 1; i < N-1; i++) {
                        A[i] = new[i];
                }
-       }
 #pragma endscop
+       }

        run_time = cur_time() - run_time;
        mflops = (((double)OPS_PER_ITER*(N-2)*T) / run_time) / 1000000L;
```

A.2.2 1dheat-detergent-dirt.patch

```
--- 1dheat-detergent.c  2009-06-12 13:39:44.000000000 -0400
+++ 1dheat-dirt.c       2009-07-22 14:10:07.000000000 -0400
@@ -51,17 +51,17 @@
        printf("%f\t", run_time = cur_time());
        fflush(stdout);

-#pragma scop
        for (t = 0; t < T; t++) {
+               #pragma omp parallel for
                for (i = 1; i < N-1; i++) {
                        new[i] = (A[i-1] + A[i] + A[i+1]) * (1.0/3);
                        /* n_fp_ops = n_fp_ops + OPS_PER_ITER; */ /*  just a check */
                }
+               #pragma omp parallel for
                for (i = 1; i < N-1; i++) {
                        A[i] = new[i];
                }
        }
-#pragma endscop

        run_time = cur_time() - run_time;
        mflops = (((double)OPS_PER_ITER*(N-2)*T) / run_time) / 1000000L;
```

A.3 2dheat-detergent.c

```
/*
2dheat.c -- a simple 2D finite difference heat equation stencil calculation
designed to benchmark Cluster OpenMP and pluto
by Tim Douglas (tdouglas@haverford.edu) -- Summer 2009

compile with:
[i|g]cc -Wall -O2 1dheat.c -o 1dheat
*/

#include <math.h>   /* for sqrt */
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

/* get N (size of array) and T (num of timesteps)
 * from decls.h, generated by par_runner.py */
#include "decls.h"

#define OPS_PER_ITER 5

double A[N][N], new[N][N];

double cur_time(void)
{
        struct timeval tv;
        gettimeofday(&tv, 0);
        return tv.tv_sec + tv.tv_usec*1.0e-6;
}

int main(int argc, char *argv[])
{
        double run_time, mflops, total = 0, sum_err_sqr = 0;
        char chtotal = 0;
        int cores;
        long t, i, j;
        /* long long n_fp_ops = 0; */

        if (argc != 2) {
                fprintf(stderr, "usage: %s numcores\n", argv[0]);
                exit(-1);
        }

        cores = atoi(argv[1]);

        /* seed the PRNG with a constant so we can check/compare the results */
        srand(42);
        for (i = 0; i < N; i++)
                for (j = 0; j < N; j++)
                        A[i][j] = rand() % 700;

        printf("%f\t", run_time = cur_time());
        fflush(stdout);

#pragma scop
        for (t = 0; t < T; t++) {
                for (i = 1; i < N-1; i++) {
                        for (j = 1; j < N-1; j++) {
                                new[i][j] = (A[i-1][j] + A[i+1][j] + A[i][j] + A[i][j-1] + A[i][j+1]) * 0.2;
                                /* n_fp_ops = n_fp_ops + OPS_PER_ITER; */ /*  just a check */
                        }
                }
                for (i = 1; i < N-1; i++) {
                        for (j = 1; j < N-1; j++) {
                                A[i][j] = new[i][j];
                        }
                }
        }
#pragma endscop

        run_time = cur_time() - run_time;
        mflops = (((double)OPS_PER_ITER*(N-2)*(N-2)*T) / run_time) / 1000000L;

        printf("%7.2f\t%7.2f\t%7.2f  \t", run_time, mflops, mflops / cores);

        /* printf("counted %lld FP ops, expected %lld\t", n_fp_ops,
```

```
                ((long long)OPS_PER_ITER)*(N-2)*(N-2)*T); */

        printf("\n# ");

        for (i = 0; i < N; i++)
                for (j = 0; j < N; j++)
                        total += A[i][j];
        printf("sum(A) = %7.2f\t", total);

        for (i = 0; i < N; i++)
                for (j = 0; j < N; j++)
                        sum_err_sqr += (A[i][j] - (total/N))*(A[i][j] - (total/N));
        printf("rms(A) = %7.2f\t\n", sqrt(sum_err_sqr));

        return 0;
}
```

A.3.1 2dheat-detergent-orange_juice.patch

```
--- 2dheat-detergent.c  2009-07-28 15:01:30.000000000 -0400
+++ 2dheat-orange_juice.c       2009-07-28 15:06:33.000000000 -0400
@@ -51,8 +51,8 @@
        printf("%f\t", run_time = cur_time());
        fflush(stdout);

-#pragma scop
        for (t = 0; t < T; t++) {
+#pragma scop
                for (i = 1; i < N-1; i++) {
                        for (j = 1; j < N-1; j++) {
                                new[i][j] = (A[i-1][j] + A[i+1][j] + A[i][j] + A[i][j-1] + A[i][j+1]) * 0.2;
@@ -64,8 +64,8 @@
                                A[i][j] = new[i][j];
                        }
                }
-       }
 #pragma endscop
+       }

        run_time = cur_time() - run_time;
        mflops = (((double)OPS_PER_ITER*(N-2)*(N-2)*T) / run_time) / 1000000L;
```

A.3.2 2dheat-detergent-dirt.patch

```
--- 2dheat-detergent.c  2009-07-28 15:01:30.000000000 -0400
+++ 2dheat-dirt.c       2009-07-28 15:08:38.000000000 -0400
@@ -51,21 +51,23 @@
        printf("%f\t", run_time = cur_time());
        fflush(stdout);

-#pragma scop
        for (t = 0; t < T; t++) {
+               #pragma omp parallel for
                for (i = 1; i < N-1; i++) {
+                       #pragma omp parallel for
                        for (j = 1; j < N-1; j++) {
                                new[i][j] = (A[i-1][j] + A[i+1][j] + A[i][j] + A[i][j-1] + A[i][j+1]) * 0.2;
                                /* n_fp_ops = n_fp_ops + OPS_PER_ITER; */ /*  just a check */
                        }
                }
+               #pragma omp parallel for
                for (i = 1; i < N-1; i++) {
+                       #pragma omp parallel for
                        for (j = 1; j < N-1; j++) {
                                A[i][j] = new[i][j];
                        }
                }
        }
-#pragma endscop

        run_time = cur_time() - run_time;
        mflops = (((double)OPS_PER_ITER*(N-2)*(N-2)*T) / run_time) / 1000000L;
```