A Paradigm for Parallel Matrix Algorithms:*
Scalable Cholesky

David S. Wise**, Craig Citro***, Joshua Hursey†, Fang Liu†, and Michael Rainey‡

Indiana University, Bloomington

Abstract. A style for programming problems from matrix algebra is developed with a familiar example and new tools, yielding high performance with a couple of surprising exceptions. The underlying philosophy is to use block recursion as the exclusive control structure, down to a $2^p \times 2^p$ base case anyway, where hardware favors iterative style to fill its pipe. Use of Morton-ordered matrices yields excellent locality within the memory hierarchy—including block sharing among distributed computers. The recursion generalizes nicely to an SPMD program where such sharing is the only communication.
Cholesky factorization of an $n \times n$ SPD matrix is used as a simple nontrivial example to expose the paradigm. The program amounts to four functions, two of which are finalizers for the other two. This insight allows final blocks to be shared with inter-node communication $\in \Theta(n^2)$ for this algorithm $\in \Theta(n^3)$ FLOPS.

CCS Categories and subject descriptors: C.1.2 [Processor Architectures]:Multiprocessors—Single-program, multiple-data-stream processors (SPMD); D.1.m [Programming Techniques]: Miscellaneous; E.1 [Data Structures]: Arrays; E.2 [Data Storage Representations]: contiguous representations; F.2.1 [Analysis of Algorithms and Problem Complexity]: Numerical algorithms and problems–computations on matrices.
General Term: Design, Languages, Performance
Additional Key Words and Phrases: quadtrees, Cholesky factorization, Morton order, finalizer.

1 Introduction

A methodology for portable, scalable algorithms for linear algebra is developed on divide-and-conquer paradigm. Performance for Cholesky factorization on a

* Supported, in part, by the National Science Foundation under grants numbered CCR-0073491, ACI–0219884, and EIA–0202048.
** Supported, in part, by NSF grants CCR–0073491 and ACI–0219884.
*** Supported, in part, by NSF grant CCR-0107395.
† Supported, in part, by NSF grant number ACI–0219884.
‡ Supported, in part, by NSF grant CCR03-34593.

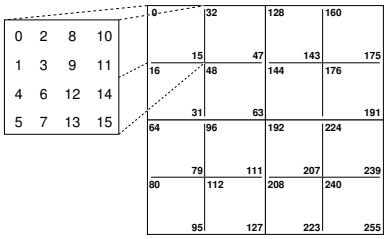

Fig. 1. Morton-order indexing of a 16× 16 matrix.

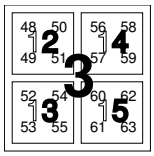

Fig. 2. Ahnentafel indexing of an order-4 matrix.

cluster of eight Xeon-powered nodes is presented; it is remarkable for a couple of reasons beyond the clean coding. First, the performance answers direct challenges to the performance of Morton-order representation [1, 2]. This paradigm brings C-coded performance on those matrices very close to that of Intel's hand-coded BLAS library [3]. Second, the recursive style does so well with locality of reference in MPI multiprocessing that the burden of interprocessor communication disappears—with cheap ol' Ethernet, that is. This eight-node multiprocessor also has Infiniband, Myrinet, and Quadrics interconnects, where the tests perform surprisingly irregularly.

Recursion is the essential tool for divide-and-conquer. The paradigm uses recursive data structures for locality, recursive programming to develop a partitioned algorithm, and SPMD recursion to balance runtime parallelism without any extra communication. The only communication for this $\Theta(n^3)$ algorithm is the $\Theta(n^2)$ sharing of blocks as they are finalized.

The underlying motivation for this paper is this straightforward paradigm of programming, yielding an understandable, portable code. We show how excellent performance arises from good locality on a uniprocessor, and demonstrate its support for distributed multiprocessing.

Three tools are developed as part of this paradigm. The first is Morton ordering for matrix representation [4, 2, 5, 1], with Ahnentafel indexing for the control of block-recursive programs [6, 7]. Figure 1 illustrates how it can represent a dense, two-dimensional array of any size in a contiguous block of address space. The properties of Morton ordering provide simple bounds checking, allow for several indexing schemes—including Cartesian indices via dilated integers [8, 9]—and guarantee that (for all p) any block of order 2^p at an address which is a multiple of 4^p resides in a block of contiguous addresses. Rectangular matrices are handled easily because Morton indices are monotonic across a row and down a column. Importantly, the matrix may occupy address space that is much bigger than the minimum to hold dense data. Just as importantly, however, blocks of unused addresses reside permanently in the most remote levels of the memory hierarchy—for the largest matrices as unallocated sectors on swapping disk.

A close relative of Morton-ordered arrays is Ahnentafel indexing, illustrated in Figure 2. Their difference amounts to two high-order bits that make all block indices unique, and still allow masking off the even or odd bits into dilated

integers that are cartesian indices. Thereby, bounds checking at all levels of the tree becomes straightforward. Instead of one pair of bounds, a vector of precomputed bounds is indexed by the level of the quadtree.

The second tool is recursive divide-and-conquer for the matrix operations [10, 1, 7, 11]. Section 2 illustrates the development of one for Cholesky factorization. An early lesson from such algorithms is to attenuate recursion above the 1×1 base cases because the last few levels are exponentially expensive, as Spieß observed long ago for Straßen's algorithm [12]. The base cases are expanded in Section 3.

The third tool, developed directly from the recursive algorithm, is the single-program–multiple-data (SPMD) recursion for multiprocessing. All processors execute exactly the same program—in this case, four recursive functions—that seem to synchronize their recursions even though most remain idle relative to work high in the process tree. The processors, themselves, are arranged as a binary tree (Figure 7) that restricts communication to parent-child links, and the control amounts to a parallel descent into this tree by each of the 2^p processors. One can envision each descending a different path p levels into the computation tree, to the level where each node has a singleton processor executing uniprocessor code.

Significantly, the algorithm is arranged so that the processor tree is an overlay of the quadtree that is the data structure, so that each of those singletons finds its work in a local chunk of memory. Interestingly, the bulk of the work (rank-k updates) can be performed in local memories without any communication at all. The aggregate $\Theta(n^3)$ processing on an order-n matrix can be carried out without any communication. Only after these results are finalized need they be shared, and so that sharing is deferred—but then that finalization and communication is only $\in \Theta(n^2)$. The sharing is explained in Section 4.

A few definitions are necessary for context, but lots of the details are available in citations.

Definition 1. [6] *The base of a matrix has* Morton-order *index* 0*. A submatrix (block) at Morton-order index i is either an element (scalar), or it is composed of* 4 *submatrices, with indices* $4i + 0, 4i + 1, 4i + 2, 4i + 3$*, labeled* northwest, southwest, northeast, *and* southeast.

Definition 2. [6] *An entire matrix has* Ahnentafel *index* 3*. A submatrix (block) at Ahnentafel index i is either an element (scalar), or it is composed of* 4 *submatrices, with Ahnentafel indices* $4i + 0, 4i + 1, 4i + 2, 4i + 3$*.*

Morton order is used to represent an entire matrix, whether it is rectangular or square, and regardless of its order. Ahnentafel indices are used for control; conversion to or from Morton order is easy, and simple bounds checking is available with either one [6]. In rectangular graphics that are wider than tall Morton order is often rendered in Z order. Because matrices tend to be taller than they are wide, we use И order. They are equivalent; both allow dilated integers to be used for cartesian indices and bounds checking.

```
#define evenBits (((unsigned int)-1) /3)
#define oddBits    (evenBits<<1)
#define diag(a) (3*((a) & evenBits))

#define nw(a)   ( (a)*4    )
#define sw(a)   ( (a)*4  +1)
#define ne(a)   ( (a)*4  +2)
#define se(a)   ( (a)*4  +3)

#define we(a)   (((a)<<2)+0)
#define ea(a)   (((a)<<2)+2)
#define no(a)   ( (a)    +0)
#define so(a)   ( (a)    +1)
```

```
#define e2w(a)   ((a) -2)
#define prnt(a)  ((a)>>2)

#define quadBd(soloBound)    (soloBound)
#define rectBd(soloBound) (4*(soloBound))
#define square    1
#define rectangle 2

#define amLeftChild(me,lgProcs)  \
  (((me)&(1<<(lgProcResource-(lgProcs)-1))) ==0)
    /*  Is "me" a left child at this level of the tree?
        Root 0 is understood leftChild by default. */
#define child(procRank,lgProcs)  \
  ( (procRank)+(1<<(lgProcResource-(lgProcs)  )) )
#define parent(procRank,lgProcs) \
  ( (procRank)-(1<<(lgProcResource-(lgProcs)-1)) )
```

Fig. 3. Helpful macros for Morton indexing.

Theorem 1. [4, 13] *The Morton index into a matrix (2-dimensional array) is* $\sum_{\ell=0}^{w-1} q_\ell 4^\ell = 2\sum_{\ell=0}^{w-1} i_\ell 4^\ell + \sum_{\ell=0}^{w-1} j_\ell 4^\ell$ *corresponds to the cartesian index for row* $i = \sum_{\ell=0}^{w-1} i_\ell 2^\ell$ *and column* $j = \sum_{\ell=0}^{w-1} j_k 2^\ell$.

The three strengths of such indexing, as in real estate, is their inherent locality, locality, locality. Base cases, caches, RAM load, disk pages, and interprocessor communication all take advantage of the sequential storage of blocks of all sizes. And any block can be sent as an unbuffered stream.

The remainder of this paper is in five parts. The next section develops the outline of a recursive Cholesky factorization. Section 3 visits fast base cases for the recurrence. Then Section 4 expands that code toward the parallel implementation. Section 5 describes the times for the resulting algorithm with different MPI and interconnects. Finally, Section 6 offers conclusions.

2 Block-Recursive Cholesky

2.1 Some Macros

Macros that are used to orient the quadtrees and their processes appear in Figure 3. The reader is referred to the literature for basic operations on Morton indices and the important role of dilated integers [8, 9, 6]. Masking the even or odd bits from a Morton or Ahnentafel index yields an (even or odd) dilated integer to the row or column, respectively. Higher in the quadtree, those identify stripes of contiguous rows for bounds checking [6].

After cleaving a square block into quadrants, they are labeled by points of the compass (`nw`, `sw`, `ne`, `se`). The binary tree of processes follows this cleaving, as well. Results of processes are first split `we`st/`ea`st into rectangles and then `no`rth/`so`uth into quadrants again.

The last few macros answer questions controlling parallelism in the processor tree:

- Is this process (rank) a left child at this level?
- What is the right child of a process at this level?
- Which is the parent of a process at this level?

```
static void doCholeskyBlk(int quad)
{
  if ( quad >= soloBound ) doCholeskyUniproc(quad);
  else
  {
    doCholeskyBlk(                        nw(quad));
    triSolveBlk  (             sw(quad), nw(quad));
    schurTri     (/*se(quad),*/ sw(quad)          );
    doCholeskyBlk(  se(quad)                      );
  }
  return;
}


static void triSolveBlk(int sQuad, int nQuad)
{
  if ( sQuad>=soloBound ) triSolveUniproc(sQuad, nQuad);
  else
  {
    triSolveBlk(so(  we(sQuad)), no(we(nQuad)) );     /*D*/
    triSolveBlk(  no(we(sQuad)), no(we(nQuad)) );     /*A*/

    schurBlk   (so(  ea(sQuad)),     so(we(sQuad)) ); /*E*/
    triSolveBlk(so(  ea(sQuad)), so(ea(nQuad)) );     /*F*/
    schurBlk   (  no(ea(sQuad)),   no  (we(sQuad)) ); /*B*/
    triSolveBlk(  no(ea(sQuad)), so(ea(nQuad)) );     /*C*/
  }
  return;
}
```

```
static void schurBlk(int eQuad, int wQuad)
{
  if ( eQuad >= soloBound ) schurBlkUniproc(eQuad, wQuad);
  else
  {
    schurBlk( so(we(eQuad)), e2w(so(ea(wQuad))) );
    schurBlk( so(we(eQuad)),     so(ea(wQuad))  );

    schurBlk( no(we(eQuad)), e2w(no(ea(wQuad))) );
    schurBlk( no(we(eQuad)),     no(ea(wQuad))  );

    schurBlk( so(ea(eQuad)), e2w(so(ea(wQuad))) );
    schurBlk( so(ea(eQuad)),     so(ea(wQuad))  );

    schurBlk( no(ea(eQuad)), e2w(no(ea(wQuad))) );
    schurBlk( no(ea(eQuad)),     no(ea(wQuad))  );
  }
  return;
}
static void schurTri(           int wQuad)
{
  if ( wQuad >= soloBound ) schurTriUniproc(wQuad);
  else
  {
    schurBlk( sw(diag(prnt(ea(wQuad)))), e2w(so(ea(wQuad))) );
    schurBlk( sw(diag(prnt(ea(wQuad)))),     so(ea(wQuad))  );
    schurTri(                           e2w(no(ea(wQuad))) );
    schurTri(                               no(ea(wQuad))  );

    schurTri(                           e2w(so(ea(wQuad))) );
    schurTri(                               so(ea(wQuad))  );
  }
  return;
}
```

Fig. 4. C code for the rudimentary four functions.

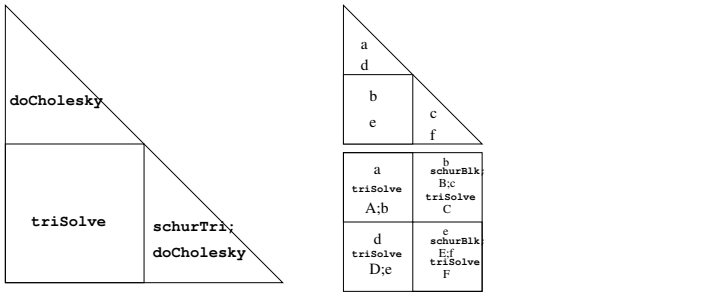

Fig. 5. Recursive decomposition of `doCholesky` and `triSolve`. In the latter, lower-case letters identify argument and upper-case indicate results of the six recursive calls.

2.2 The Algorithm

One finds iterative codes in textbooks covering Cholesky factorization. The simplicity of three nested loops is intoxicating, but it doesn't help much in balancing and scheduling processes. Good recursions can be found by abstracting the problem to be of size $2^p \times 2^p$. With that approach one arrives at the recursive algorithm in Figure 4.

Figure 5 sketches the recursion for two of the four functions presented in Figure 4. The other two are rank-k updates called `schurBlk` for a square result, and `schurTri` when the result lands on the diagonal of the symmetric matrix; it's half the work of `schurBlk`.

In most cases, the first parameter identifies the quadrant receiving an update (or side-effect) by the named code. Similarly, the left of Figure 5 identifies the

blocks of the top-level matrix by the functions that update them. The computationally intensive `schurBlk`, however, does not yet appear.

The right of Figure 5 illustrates the source of all its $\Theta(n^3)$ calls: `triSolve`. There, lowercase letters indicate arguments to the six calls identified by right-margin comments in Figure 4; upper case indicates their in-place effects. Not all three arguments to, say, `schurBlk` are required as its parameters—because two Ahnentafel indices suffice to compute the third. That is also true of `triSolve`, itself; the Ahnentafel index, `a`, of a block labeled A, D, C, or F is sufficient to determine the index of the diagonal block immediately above it (*e.g.*`(a&oddBits)/2*3`) —so the latter does not appear later in Figure 15. Similarly, `schurTri` in Figure 4 only has one parameter that does *not* index the side-effected block, whose index is calculated from that of the block A, D, C, or F in Figure 5 using `diag(a)` from Figure 3. We have found that deriving such dependent indices with dilated integers avoids a lot of incompatibility errors.

3 Base Cases

Excellence performance at the base cases is essential for high performance of recursive programs; anything less has an explosive cost. Three steps are necessary to achieve that performance: analysis, optimization, and tuning.

Theorem 2. *While factoring an $n \times n$ matrix (measured in base blocks—whether 1×1 or 32×32) the base case is invoked* exactly

- $\binom{n}{1}$ *times for* `doCholesky`*; flops* $\in \Theta(n)$.
- $\binom{n}{2}$ *times for* `triSolve`*; flops* $\in \Theta(n^2)$.
- $\binom{n}{2}$ *times for* `schurTri`*; flops* $\in \Theta(n^2)$.
- $\binom{n}{3}$ *times for* `schurBlk`*; flops* $\in \Theta(n^3)$.

The analysis of Theorem 2 focuses optimization on the base case of `schurBlk`. C and FORTRAN optimizers, however, are not set up for Morton order, so we wrote a macro for that Schur complement to generate code for RISC technology. With f and p as integer tuning parameters, it generates iterative C code for a $2^p \times 2^p$ base block with 2^f in-line flops; choice of p depends on data cache, and of f on instruction cache, decoding, and the pipeline. Experiments a range of values for a large problem select a 32×32 base block and 256 inline flops. All source code is in C; the only assemply code was used to gain SSE2 performance over the compiler's scalar code.

Figure 6 presents the uniprocessor timings that result from this tuning on the coding techniques described elsewhere [6, 7]. Consistently with presentations of other cache-oblivious algorithms [10], it plots time to perform an order-n Cholesky factorization divided by the $\frac{1}{3}n^3$ FLOPs necessary for the algorithm. In other words, it normalizes to a hypothetical leading coefficient for the cubic equation that would express the time as a function of n [14], which ought to be constant with good scaling.[1] Read from top-to-bottom, Figure 6 plots the quad-tree divide-and-conquer C-codes extended from Figure 5 running on the usual

[1] This plot is valuable also because it exposes relative performance on small tests.

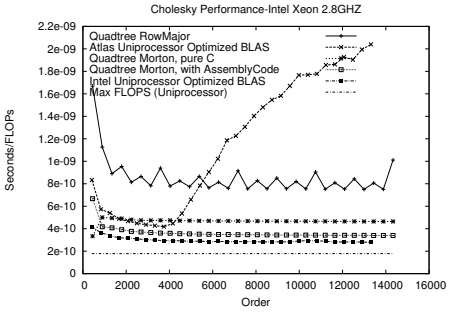

Fig. 6. Tuned uniprocessor performance.

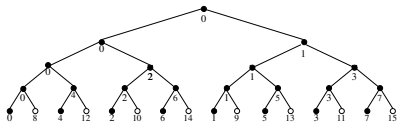

Fig. 7. A tree labeled for 16 processors, each communicating only with its parent and right child.

row-major implementation (almost flat), and ATLAS-optimized BLAS3 `dpotrf` [15] (not flat) whose performance degrades as it bleeds into outer caches. Below order 4000 it improves slightly on Morton-ordered recursions[2] (very flat) even though they do not compile to SSE2 instructions. Substituting eight SSE2/MMX packed-`double` instructions (instead of `double`) into `schurBlkBase`'s symbolic code gives the next plot. Only slightly better is Intel's hand-coded `dpotrf` [16, 3] (flat) and the idealized maximum FLOPs for our Xeon processor.[2]

4 Developing the Code

This section offers a terse description of the expansion of the recursive algorithm in Figure 5 to the SPMD version that offers perfectly balanced and scalable parallelism. This description is again written as if matrices were of order 2^p, which is an unnecessary constraint. Without it, however, another derivative of the code in Figure 5 with bounds checking is necessary to peel the unbalanced southeast perimeter from the matrix.

Three observations take us to the parallel code. First is the processor tree of Figure 7. The root is identified as the processor with MPI rank 0. Beneath it is a tree of processors indexed by a bit-reversal of level ordering: every processor is its own left child and its right child is easily computed from the level. This indexing expands to 2^p processors (for any p), and localizes communication within subtrees. Each processor communicates *only* with its parent and right child. With MPI ranks assigned cyclically, processors at the leaves (the open circles) can even share RAM on a single node.

Second is the recognition that the four functions are paired with respect to a locality operation. That is, the blocks that result from `schurTri` are not needed by any processes until they are finalized by an application of `doCholesky` there. These functions, however, are not applied very often (Theorem 2) and so are computed by the root (Rank 0) processor. Far more interesting are the results of `schurBlk` that are not shared by other processes until finalized by an application of `triSolve`. They are distributed out to RAM on remote processors

[2] These two plots appear as referents in later graphs.

in a pattern that will receive all the updates to these blocks, up to an including the finalization. As part of that `triSolve` finalization the results are assembled up and down the tree—so that all other processors immediately become aware of them. Theorem 2 shows that that communication is only time $\Theta(n^2)$ after the $\Theta(n^3)$ Schur complements.

The third observation is that *all* processors are always executing the same recursion, descending the processor tree and the quadtree level-by-level. The critical parameter `whereAmI` indicates which processor is designated to be "active," in the sense of actually carrying out computations and side-effects to blocks of the matrix. Base steps in the parallel code (uniprocessing calls), therefore, are always protected by a conditional comparison with it and the local processor's rank. Even non-base steps use it as if to "fork" work to two children, effected by locally computing that argument.

That is, in the SPMD recursion all processors are always executing the same code at the same level, with `whereAmI` identifying an ancestor and the local processor idle—aside from the synchronized recursion. Alternatively, it identifies the rank of the local processor, and it is actively computing on the blocks identified by the Ahnentafel indices. There is no need for interprocess communication to fork and join since all control is implicit in the recursion.

The necessary `MPI_send`s and `MPI_receive`s of data are embedded in the same recursion and similarly synchronized. Two functions are used locally to share information:

- `sendDownAll` sends a square or rectangle block down to all descendants of the local processor. It is invoked after receiving a finalized block from one's parent.
- `assembleBottomUpDown` is invoked immediately before a finalizing call to assemble distributed data onto all the local RAMs in the subtree. Typically, it happens just before `triSolve`.

This provides the only communication necessary to the SPMD parallelism.

Figure 4 then expands first into Figure 8 with provision to assemble distributed Schur complements. The rest of the parallel code appears as Figures 14 and 15, which are formatted to be read side-by-side. That way the control is seen to be identical, and the absence of any communication in Figure 14 is apparent. The rank-k updates are accumulated in distributed memory in sequential blocks, each of which exists on a unique machine in the cluster; the programmer can be oblivious as to just which one—recursion determines it. No sharing occurs until just before such blocks are finalized by the code in Figure 15, and after finalization the block is implicitly shared by all processors in the subtree of the finalizing one.

Figure 14 presents the rank-k updates without any interprocess communication either for control (the SPMD recursion suffices) or for sharing. There again appear the guards on the base cases, performed only by the process selected by the parameter `whereAmI`, selecting the active processor. Here also is seen how the recursion forks within `schurBlk`, based on a bit masked from *every* processor's rank. Half the processors recur to the left subtree and half to the right,

```
static void doCholeskyBlk(int quad)
{
  if ( lgProcResource==0 || quad >= quadBd(soloBound) )
  {
    if (me != 0  /*whereAmI*/) {} else
      doCholeskyUniproc(quad);
    sendDownAll(square, quad,  0, lgProcResource);
       /* assuring that all processors have the factorization */
  }
  else
  {
      doCholeskyBlk(nw(quad));
       /* All  processors have the partial result. */

        /* Upon arriving here (except at topmost call) we must assume
        that rank-k updates (shurBlk calls) have scattered the current
        information in sw(quad) across remote memories, quad-by-quad.
        The first task is to assemble current information in all memories.
        */
      assembleBottomUpDown( sw(quad), 0, lgProcResource);
      triSolveBlk         ( sw(quad), 0, lgProcResource);
       /* All  processors have the partial result. */

      schurTri     (        sw(quad)                  );
       /* Not all processors have the rank-k updates.  They are local to... */
      doCholeskyBlk(se(quad)                          );
       /* All  processors have the partial result. */
  }
  return;
}
```

Fig. 8. The top level of parallelism.

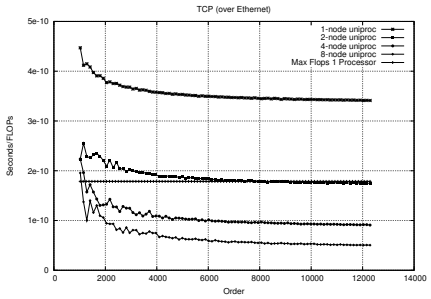

Fig. 9. Normalized time for Ethernet multiprocessing. Hardware-cost rating: **$**

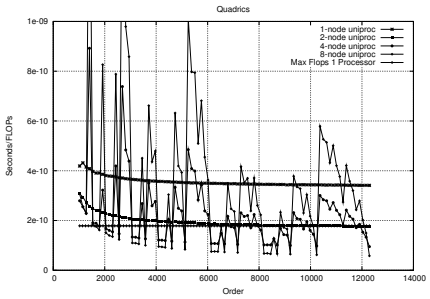

Fig. 10. Normalized time for Quadrics multiprocessing. Hardware cost: **$$$$**

one of which is newly invigorated as second argument for `whereAmI`; it allows computation when it arrives at the processor identified locally as `me`.

5 Experimental Results

The codes were tested on an Aspen Systems cluster of eight, dual-processor 2.8GHz Intel Xeons, each with 2GB of memory and 8KB L1, 512KB L2 cache. All the uniprocessing code was compiled using the native ICC compiler with `-O3` optimization for the Xeon. Section 3's hand conversion of scalar instructions introduced SSE2 packed-`double`s into the symbolic code for `schurBlkBase`.

The multiprocessing code was written using MPI [17] to distribute computation via four different interconnects. (The codes are identical, but there is room for only two plots here.) Since this work is motivated by programming style, we only sought confirmation of performance. The differences in performance among the interconnects were completely unexpected.

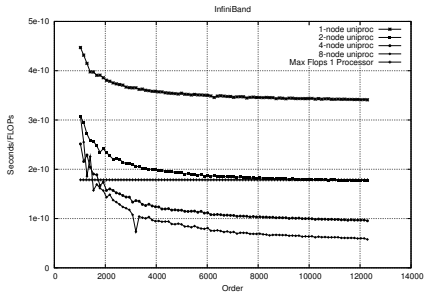

Fig. 11. Normalized time for Infiniband-connected uni-node multiprocessing. Hardware-cost rating: **$$**

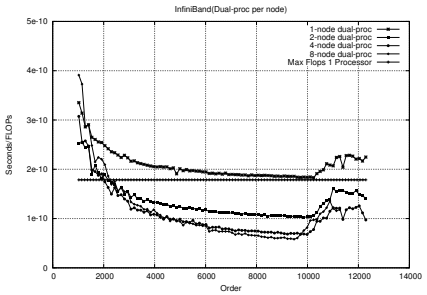

Fig. 12. Normalized time for Infiniband-connected dual-node multiprocessing. Hardware cost rating: **$$**

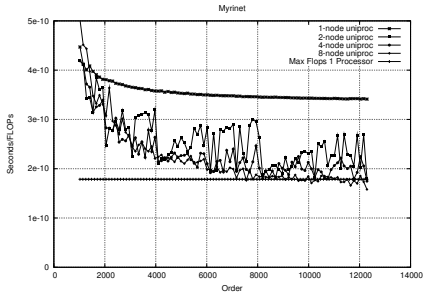

Fig. 13. Normalized time for Myrinet-connected uni-node multiprocessing. Hardware cost rating: **$$**

– TCP via on-board GIGABIT ETHERNET.
– Infiniband [18] through InfiniCon INFINIIO 2000 [19].
– Myrinet M3-E64 [20].
– Quadrics ASNET II [21].

Multiprocessing code for the last was compiled with Quadrics's release of MPI [22]. The other three were compiled with LAM/MPI, 7.0.1 here [17]. All tests were run twice, most of them thrice. There were minor variances, therefore, that are not shown here; the shapes of these plots are all faithful and reproducible.

Times are presented here in units of seconds-per-FLOP as above [14].

Each of the plots offers a surprise. Figure 9 indicates excellent performance. Although Ethernet is the cheapest interconnect and presumably the slowest, its plots land at $\frac{1}{2}, \frac{1}{4}$, and $\frac{1}{8}$ of Figure 6's baseline: perfect scaling for $2, 4, 8$ processors. With the timings obtained, this figure alone demonstrates the success of the paradigm and its resulting code.

Infiniband in Figure 11 does almost as well, but its eight-processor performance is poor on smaller matrices. Most notably both it and Myrinet have memory pegged in RAM before startup for resident buffers. That consumes *address* space that Morton order requires for any test above order 8192. That is,

the test for 8224 requires a `malloc` of just over 1.5GB, even though only 1/6 of that migrates through RAM to cache. So there are no tests above that order.

The dual-processor Infiniband tests in Figure 12 are more puzzling. One can see results for 16 processors, but they are little better than for 8 uniprocessor nodes. Performance degrades for dual processors on large matrices, above order 9000, as RAM prematurely fills. Because this MPI cannot share memory on a single node, pegged buffers and individual copies of the matrix are necessary for each of the dual processors. As memory fills, they both revert to virtual memory, although their locality under paging is also very good.

Quadrics, the most expensive, scales as perfectly for 2 uniprocessor nodes in Figure 10. Normalized times for 4 and 8 nodes tend toward a nicely scaled asymptote, but leap raggedly above it. The strange leaps indicate MPI/system troubles. The 8-processor leaps consistently double those for 4 processors, and this pattern has been confirmed in repeated runs.

The most surprising is Myrinet in Figure 13. Not only does it also suffer from pegging, but all multinode tests are ragged. Although the 8-node test seems more stable, all of them trend towards times that we would expect from just two processors. One suspects difficulties between the network controllers and the operating system.

6 Conclusions

It was already known how to obtain scalable parallelism from algorithms for Cholesky factorization. The point of this paper is the paradigm to obtain both excellent performance from hierarchical memory and excellent parallel scaling.

We illustrate an algorithm that yields cache-oblivious behavior in hierarchical memory, and similarly delivers scalable and balanced parallelism on distributed multiprocessors. We obtain both locality in hierarchical memory and excellent parallel scaling from the same style, using quadtree recursion on Morton-ordered matrices. We delivered excellent performance in moving from shared-memory parallel `dpotrf` to a SPMD a distributed-memory parallel implementation. Along the way we expose some strange behavior from current Linux/MPI implementations. The best behavior is achieved on the cheapest interconnection.

The perfect scaling of Figure 9 demonstrates success for our recursive paradigm for high-performance code. Single-recursion multiple-data (here dubbed SRMD), Morton-ordered matrices, and good style are winners.

Acknowledgement

The first author thanks his hosts at the University of Washington where Section 3 was thrashed out.

References

1. Chatterjee, S., Lebeck, A.R., Patnala, P.K., Thottenthodi, M.: Recursive array layouts and fast parallel matrix multiplication. IEEE Trans. Parallel Distrib. Syst. **13** (2002) 1105–1123 http://dx.doi.org/10.1109/TPDS.2002.1058095

2. Thiyagalingam, J., Beckmann, O., Kelly, P.H.J.: Is Morton layout competitive for large two-dimensional arrays, yet? Concur. Comput. Prac. Exper. (2004) To appear in special issue on Compilers for Parallel Computing. http://www.doc.ic.ac.uk/~phjk/Publications/IsMortonYetCCPandE2004.pdf
3. Goto, K., van de Geijn, R.: On reducing TLB misses in matrix multiplication. FLAME Working Note 9, Univ. of Texas, Austin (2002) http://www.cs.utexas.edu/users/flame/pubs/GOTO.ps.gz
4. Morton, G.M.: A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Ontario (1966)
5. Drakenberg, P., Lundevall, F., Lisper, B.: An efficient semi-hierarchical array layout. In Lee, G., Yew, P.C., eds.: Interaction between Compilers and Computer Architectures. Volume 613 of Kluwer Intl. Series in Engineering and Computer Science. Kluwer, Deventer, Netherlands (2001) http://www.mrtc.mdh.se/publications/0313.pdf
6. Wise, D.S.: Ahnentafel indexing into Morton-ordered arrays, or matrix locality for free. In Bode, A., Ludwig, T., Karl, W., Wismüller, R., eds.: Euro-Par 2000 – Parallel Processing. Volume 1900 of Lecture Notes in Comput. Sci. Springer, Heidelberg (2000) 774–883 http://www.springerlink.com/link.asp?id=0pc0e9gfk4x9j5fa
7. Wise, D.S., Frens, J.D., Gu, Y., Alexander, G.A.: Language support for Morton-order matrices. Proc. 8th ACM SIGPLAN Symp. on Principles and Practice of Parallel Program., SIGPLAN Not. **36** (2001) 24–33 http://doi.acm.org/10.1145/379539.379559
8. Schrack, G.: Finding neighbors of equal size in linear quadtrees and octrees in constant time. CVGIP: Image Underst. **55** (1992) 221–230
9. Raman, R., Wise, D.S.: Converting to and from dilated integers. Submitted for publication (2004) http://www.cs.indiana.edu/~dswise/Arcee/castingDilated-comb.pdf
10. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache–oblivious algorithms. In: Proc. 40th Ann. Symp. Foundations of Computer Science. IEEE Computer Soc. Press, Washington, DC (1999) 285–298 http://dx.doi.org/10.1109/SFFCS.1999.814600
11. Frens, J.D.: Matrix Factorization Using a Block-Recursive Structure and Block-Recursive Algorithms. PhD thesis, Indiana Univ., Bloomington (2002) http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR568
12. Spieß, J.: Untersuchungen des Zeitgewinns durch neue Algorithmen zur Matrix-Multiplication. Computing **17** (1976) 23–36
13. Tocher, K.D.: The application of automatic computers to sampling experiments. J. Roy. Statist. Soc. Ser. B **16** (1954) 39–61 See pp. 53–55.
14. Johnson, D.S.: A theoretician's guide to the experimental analysis of algorithms. In Goldwasser, M.H., Johnson, D.S., McGeoch, C.C., eds.: Data Structures, Near Neighbor Searches, and Methodology: 5th & 6th DIMACS Implementation Challenges. Volume 59 of DIMACS Ser. Discrete Math. Theoret. Comput. Sci. Amer. Math. Soc., Providence (2002) 215–250 http://www.research.att.com/~dsj/papers.html
15. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: Proc. Supercomputing '98. IEEE Computer Soc. Press, Washington, DC (1998) 38 http://dx.doi.org/10.1109/SC.1998.10004
16. Intel Corp. Santa Clara, CA: Intel Math Kernel Library. (2003) http://www.intel.com/software/products/mkl/
17. LAM/MPI Bloomington, IN: `www.lam-mpi.org`. (2004)
18. InfiniBand Trade Assn. Portland, OR: `www.infinibandta.org`. (2004)
19. InfiniCon Systems King of Prussia, PA: `www.infinicon.com`. (2004)
20. Myricom Inc. Arcadia, CA: `www.myri.com`. (2004)
21. Quadrics Ltd. Bristol, UK: `www.quadrics.com`. (2004)
22. Quadrics Ltd. Bristol, UK: Quadrics Release of MPICH 1.24. (2004) www.quadrics.com

Fig. 14. Aligned C code for `schurBlk` and `schurTri`. No communication at all occurs among the highly parallel `schurBlk` recursions.

Fig. 15. Aligned C code for `triSolve`. The subtree of memories must be synched for finalization after distributed `schurBlk`s.