Superscalar programming 101 (Matrix Multiply)

by

Jim Dempsey
QuickThread Programming, LLC
85 Cove Lane
Oshkosh, WI 54902

jim@quickthreadprogramming.com
www.quickthreadprogramming.com

Introduction

This document is a composition of a 5 part article presented on the Parallel Programming Community of the Intel Software Network (http://software.intel.com/en-us/parallel/).

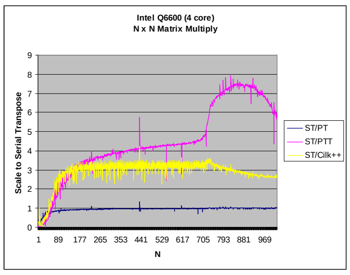
Some minor edits to this article have been made in this composition to aid in readability, in particular: a table of contends was added, the segment bylines have been removed, and an appendix was added to include code samples.

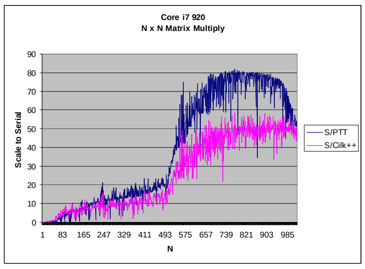
Part 1

The subject matter of this article is: How to optimally tune a well known algorithm. We will take this well known (small) algorithm, a common approach to parallelizing this algorithm, a better approach to parallelizing this algorithm, and then produce a fully cache sensitized approach to parallelizing this algorithm. The intention of this article is to teach you a methodology of how to interpret the statistics gathered during test runs and then use those interpretations at improving your parallel code.

This is a multi-part article, where the author believes the process in reaching a goal (learning how to assess results and improve upon your parallel programming technique) is as important as the goal (finished application). For without the process you will not know how to attain goal to the extent possible.

To titillate you into reading the complete set of articles, you will experience how to attain up to 80x parallel programming performance increase on a single processor (4 core with HT) system over a simple serial method on the same system. (don't zoom the charts)





Let us begin…

One of the posters on the Intel Software Network forums provided a link to an MSDN article titled "**How to: Write a parallel_for Loop**" (http://msdn.microsoft.com/en-us/library/dd728073.aspx). This article illustrates how to use the Visual Studio 2010 **Concurrency::parallel_for** to compute the product of two matrices. The example illustrates how to use the **parallel_for** in a nested loop.

The typical C++ serial method to compute the matrix multiplication of a square matrix might look as follows:

```
// Computes the product of two square matrices.
void matrix_multiply(
      double** m1, double** m2, double** result, size_t size)
{
   for (size_t i = 0; i < size; i++)
   {
      for (size_t j = 0; j < size; j++)
      {
         double temp = 0;
         for (int k = 0; k < size; k++)
         {
            temp += m1[i][k] * m2[k][j];
         }
         result[i][j] = temp;
      }
   }
}
```

Using the VS concurrency collection **parallel_for** the code looks like this:

```
// Compute the product of two square matrices in parallel.
void parallel_matrix_multiply(
      double** m1, double** m2, double** result, size_t size)
{
   parallel_for (size_t(0), size, [&](size_t i)
   {
      for (size_t j = 0; j < size; j++)
      {
         double temp = 0;
         for (int k = 0; k < size; k++)
         {
            temp += m1[i][k] * m2[k][j];
         }
         result[i][j] = temp;
      }
   });
}
```

This makes use of the C++0x Lambda functions and the Concurrency template for the **parallel_for**. This conversion on the surface ***appears*** to be elegant (unusually effective and simple) by essentially rewriting two lines of code: The first **for** statement and closing brace of that for statement.

The MSDN reported performance boost on a 4 processor system for 750 x 750 matrix multiplication as:

Serial: 3853 (ticks)
Parallel: 1311 (ticks)

Approximately a 2.94x speed-up.

This is not an ideal scaling situation in that it does not produce a 4x speed-up using 4 processors. But considering that there must be some setup overhead, 2.94x is not too bad on this small of matrix. And you might be inclined to think that there is only 25% room remaining for improvement.

The article did not chart the scaling as a function of N (one dimension of a square matrix) so it is difficult to tell the shape of the performance gain trend line as a function of N.

Although this article was written to illustrate the use of the **parallel_for** in the MS Concurrency, a parallel programmer might be miss-lead into assuming that this example illustrates how to write a parallel matrix multiplication function. After all, this article describes a parallel method for matrix multiplication and was written in an MSDN article – an authoritative source.

Let's see how we can do better at parallelization of the Matrix Multiplication.

First off, I do not have Visual Studio 2010, so I cannot use the sample program as-is.
I do have QuickThread (I am the author of this parallel programming toolkit www.quickthreadprogramming.com). Therefore, I adapted the code to use QuickThread.

Adaptation is relatively easy. Change the include files and minor differences in syntax.

```
// Computes the product of two square matrices in parallel.
```

```
void parallel_matrix_multiply(
      double** m1, double** m2, double** result, size_t size)
{
    parallel_for(
        0, size,
        [&](intptr_t iBegin, intptr_t iEnd)
        {
            for(intptr_t i = iBegin; i < iEnd; ++i)
            {
                for (intptr_t j = 0; j < size; j++)
                {
                    double temp = 0;
                    for (intptr_t k = 0; k < size; k++)
                    {
                        temp += m1[i][k] * m2[k][j];
                    }
                    result[i][j] = temp;
                }
            }
        }
    );
}
```

In the QuickThread dialect, the **parallel_for** passes the half open range as arguments to the function, as opposed to the **parallel_for** of the VS 2010 concurrency collection passing the single index into the body of the function. QuickThread chose to pass the half open range, as opposed to a single index, because knowing the range, the programmer and/or compiler can better optimize the code within the loop. N.B. this loop could have as easily been written using OpenMP or Cilk++, or TBB. The choice parallel tool language/dialect is not important at this point of the article.

Using the QuickThread modified code, and run on Windows XP x64 and using an Intel Q6600 4 Core processor without Hyper Threading we find the scaling as:

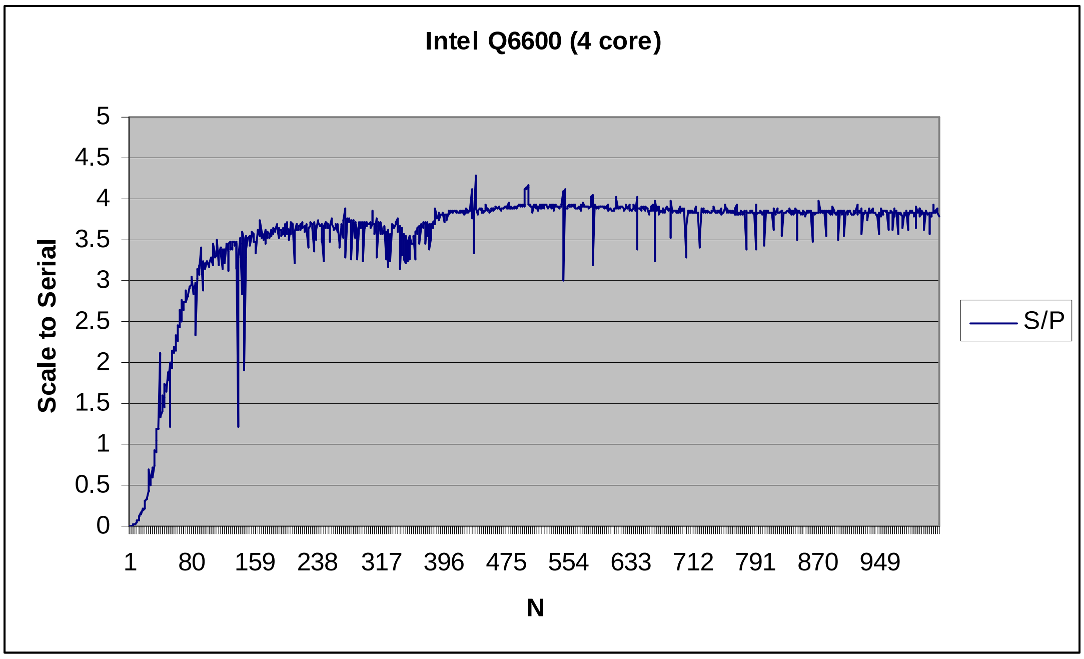


Fig 1 – Parallel Scaling

For an average scaling ratio of 3.77x at N values between 128 and 1024. And considering four threads are involved the above chart shows the parallel version of the serial code yielding a scaling factor of 0.94 (scale / number of cores). This is a reasonably good scaling factor.

The processor model was not listed for the Visual Studio test so it is hard to tell the reason why QuickThread yields 3.77x improvement on 4 cores, while VS yields 2.94x improvement on 4 processors (cores?). Setting aside the issue of which threading toolkit is better, let's concentrate on the parallelization of the matrix multiplication.

First thing to do is get an additional (different) set of sample data. Let's see what the performance is on a processor supporting L3 cache plus Hyper Threading (HT).

When running the program on an Intel Core i7 920, 4 cores with HT (8 hardware threads – but only 4 floating point instruction paths) we find a completely different trend line:

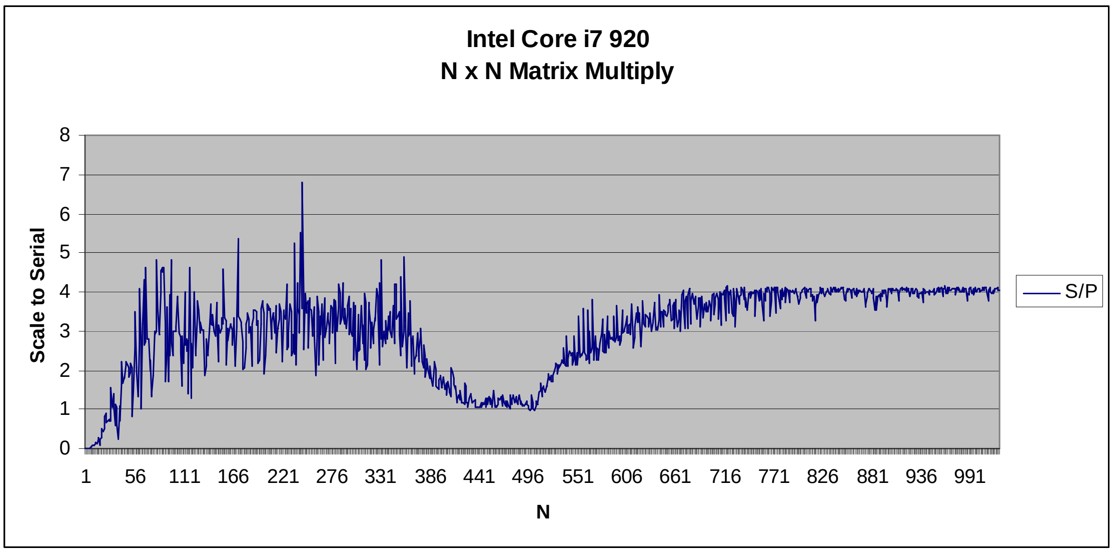


Fig 2

Why the dip when N ranges from 350 to 570?
When the performance recovers, why do we see 4x improvement instead of 8x improvement?
Why the jagged line (noise) towards lower N?

The dip is due to cache evictions caused by one thread adversely interacting with other threads. We get just over 4x performance because the Core i7 920 is a 4 core processor capable of 4 execution streams for floating point (although it has 8 hardware threads for integer execution streams). On the higher N values it would appear that we are maxing out the capabilities of processor, 4 cores == 4 x performance boost.

The trend line is jagged due to only one sample run being taken, and due to the short run times when N is low. Also, the variable overhead in starting the thread team (now 8 threads) is more noticeable on the left hand side of the curve. An average of a larger number of runs would smooth out this line, but this consumes unnecessary time from the developer. Additionally the large spike at about 250 indicates that the Serial version took a large tick count hit at that point due to out of application control circumstances (expansion of page file or other activity on the system). You do not need a precise chart for program analysis purposes.

Is there anything we can we do to improve this performance and/or correct for the dip in performance for N between 350 and 570?

The main computational section of code for both Serial and Parallel are the same:

```
      for (intptr_t j = 0; j < size; j++)
      {
            double temp = 0;
            for (intptr_t k = 0; k < size; k++)
            {
                  temp += m1[i][k] * m2[k][j];
            }
            result[i][j] = temp;
      }
```

Notice that the product accumulation statement

```
      temp += m1[i][k] * m2[k][j];
```

uses k to index the column of array `m1` and the row of array `m2`. This access pattern has two problems: First, it is not cache friendly, and second, it is not amenable to vectorization by the compiler. Let's look at fixing these problems.

In the Cilk++ sample programs, which is (or will be) available with the Intel Parallel Studio, we find a sample program showing a cache friendly implementation of matrix multiply:

```
// Multiply double precision square n x n matrices. A = B * C
// Matrices are stored in row major order.
void matrix_multiply(double* A, double* B, double* C, unsigned int n)
{
    if (n < 1) {
        return;
    }

    cilk_for(unsigned int i = 0; i < n; ++i) {
// This is the only Cilk++ keyword used in this program
// Note the order of the loops and the code motion of the i*n and k*n
// computation. This gives a 5-10 performance improvement over
// exchanging the j and k loops.
   int itn = i * n;
       for (unsigned int k = 0; k < n; ++k) {
            for (unsigned int j = 0; j < n; ++j) {
              int ktn = k * n;
                // Compute A[i,j] in the inner loop.
                A[itn + j] += B[itn + k] * C[ktn + j];
            }
        }
    }
    return;
}
```

The `itn` (`i` times n) and `ktn` (k times n) variables take large steps through the arrays. The array pointers point to single dimensioned arrays that are row packed. This is to say each row is appended to the prior row into one large single dimension array. Which is then indexed by way of `i*n` and/or `k*n`.

Serial and Parallel function signatures:

```
      void matrix_multiply(
            double** m1, double** m2, double** result, size_t size);
      void parallel_matrix_multiply(
            double** m1, double** m2, double** result, size_t size);
```

Note `double**` on the 2D array pointers.

Cilk++ function signature:

```
 void matrix_multiply(double* A, double* B, double* C, unsigned int n);
```

Note `double*` on the 2D array pointers.

Row packing principally does one beneficial thing to your program. Row packing eliminates a memory fetch of a pointer to the row (as done with array of row pointers). This uses fewer Virtual Memory Translation Look Aside Buffers. And this improves cache utilization.

```
                  temp += m1[i][k] * m2[k][j]; // serial/parallel
```

Uses: 1 TLB for code + 1 TLB for stack (could be 0) + 1TLB for m1 row table + 1 TLB for m1 data + 1 TLB for m2 row table + 1 TLB for m2 data = 6 (or 5) TLB's.

Whereas the row packing method employed by the Cilk++ sample program:

```
                A[itn + j] += B[itn + k] * C[ktn + j];
```

Uses: 1 TLB for code + 1 TLB for stack (could be 0) + 1 TLB for m1 data + 1 TLB for m2 data = 4 (or 3) TLB's.

The reduction in the numbers of TLB's required should produce an advantage.

And the corrisponding charts:

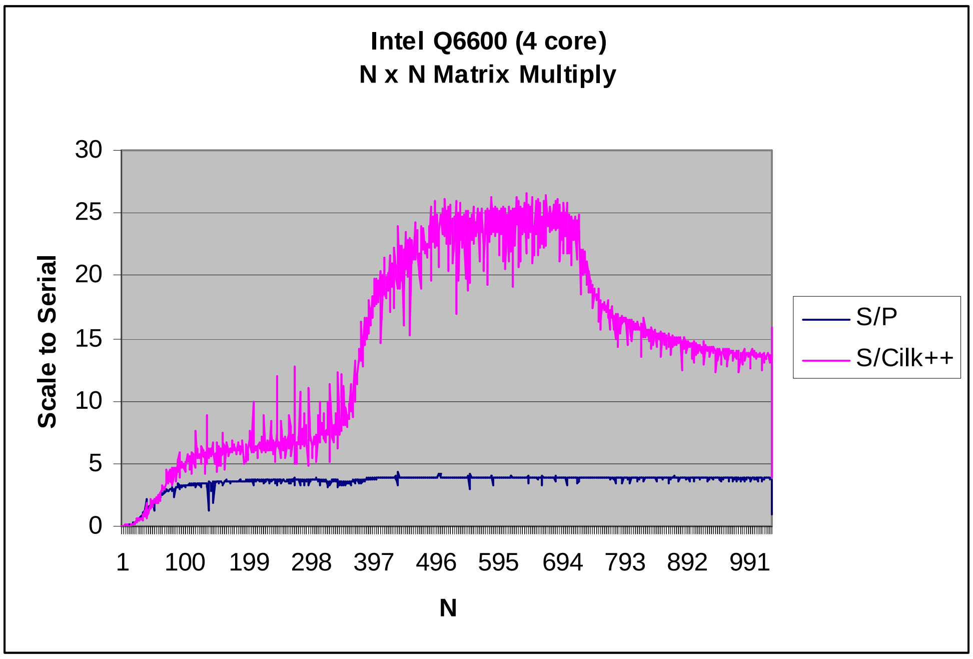


Fig 3 (above)

Wow!, the Cilk++ technique on the Q6600 has a nice peak between 400 and 700 where it attains close to 25x performance over Serial but dropps off to about 14x at the higher range. This represents a 3x to 6x improvement over the parallel version of the standard matrix multiply code. The elimination of the array of row pointers made a significant difference.

I should state that the technique used by the Cilk++ demo program is referred to as "Cilk++" on the charts and in the body of this text. The technique used can be used by other parallel programming dialects. Keep this in mind as you read this series of articles.

Now looking at the Core i7 920 processor we find:

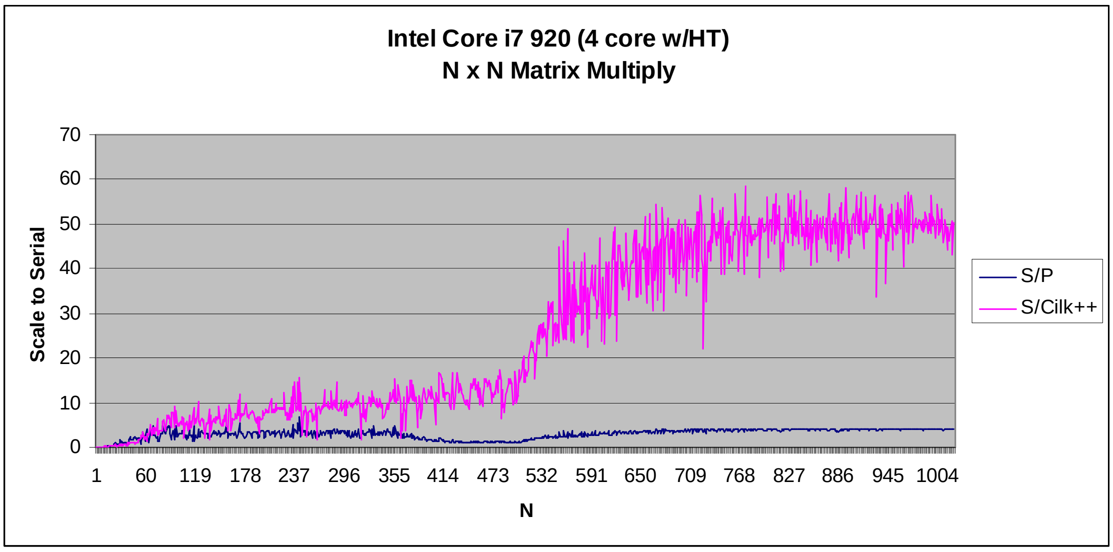


Fig 4

The Cilk++ method on the Core i7 920 attains almost 50x performance gain over serial method at about N=800 (12x over parallel), but appears as if it will be dropping off after 1024 (as it did earlier at 700 on the Q6600). Additional data points should be collected.

What this teaches us is: constructing 2D arrays as an array of pointers to 1D arrays looks good but can cost you dearly. Clearly the more efficient route is to use row packing to reduce the number of TLBs and corrisponding entries in the cache. But this means changing

```
  double** A; // referenced as A[iRow][iCol]
```

into some class

```
  Array2D<double>  A;
```

And then changing the allocations and references (or fancy template operators).

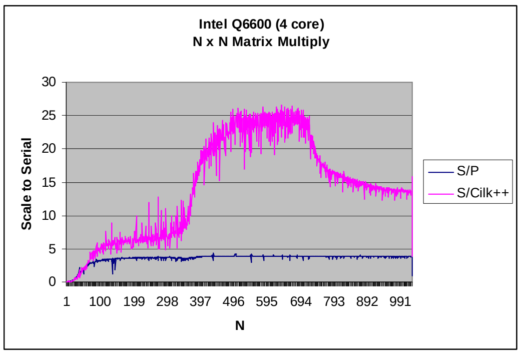
If you look at this from a different perspective the array class object(s) are effectively array descriptors. Array descriptors are a well used technique by FORTRAN.

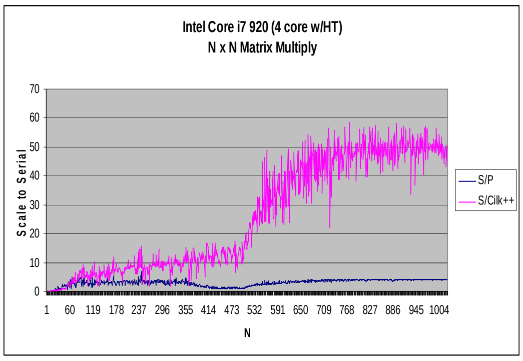
WIth an improvement of 12x over a parallel version of the serial implimentation this shows that paying attention to cache issues realy pays off. And that you may have to dispense with the familiar C/C++ programming practice of referencing a multi-dimensioned array as an array of pointers. A little extra programming effort pays off with significant returns in performance.

You would have to ask yourself: is this the best you can do?
And, is it worth your while to attempt at better performance?

Part 2

In the previous sectiion we had:





The above charts, impressive as they are, are an “apples and oranges” type of comparison. The chart is comparing a non-cache sensitive serial technique against a cache sensitive parallel technique. Good for promotional literature, certainly a good incentive to learn how to program in parallel, but this falls short for analysis purposes by the seasoned parallel programmer.

The first thing any parallel programmer should know is: improve the serial algorithm first, and then address the problem through parallelization (save intrinsic small vector functions for last).

If you look at the inner most loop of the Serial function we find:

```
for (int k = 0; k < size; k++)
{
   temp += m1[i][k] * m2[k][j];
}
```

We addressed the issue of the traditional C/C++ programming practice of using an array of pointers to rows, and saw that by dispensing with this practice that a 6x to 12x improvement in performance. What else can be done?

Set aside the issue of the TLBs and row table for a moment. In examining the principal computational statement we find that while the k index in `m1[i][k]` sequentially accesses memory, the k index in `m2[k][j]` does not. What this means is the compiler is unable to vector this loop. And potentially worse, stepping down the rows at certain distance intervals are known to cause cache evictions (the dip in curve observed on Core i7 920). Using vectors on doubles might attain a 2x improvement – well worth going after. So let’s improve the vector-ability of the Serial version of this program. (and Parallel variant of the Serial version too)

Note, while we could use the approach done in the Cilk++ implementation (eliminate row table of pointers), I will choose an alternate means that will become useful later. First address the simple Serial and Simple Parallel to see what happens.

In order to assist the compiler in vectorizing the matrix multiplication (and minimize cache evictions) you can transpose the matrix `m2` (to `m2t`), and then you can transpose the indexes in the inner loop. At first this may sound absurd. To transpose the matrix m2 you will have to add overhead to:

Perform an allocation (actually size+1 allocations with 1 for the row pointers)
run transposition loops

reading and re-writing one of the arrays (array m2 to m2t)

You might assume that this must be slower than the simple matrix multiplication.
Well you would be wrong to assume this. Each cell in **m2** is used N times, each cell in **m1** is used N times. We have the potential of trading off the cache misses on 2N**2 reads plus N*N writes against N reads + N/2 writes + (2N**2)/2 reads + N/2 writes .The /2 is due to the inner loop now becomes a DOT function that operates on two sequential arrays and is an ideal candidate for vectorization by the compiler. The rewritten Serial loop (and inner loop transformed into an inline function) looks like:

```
// compute DOT product of two vectors, return result as double
inline double DOT(double* v1, double* v2, size_t size)
{
    double temp = 0.0;
    for(size_t i = 0; i < size; i++)
    {
       temp += v1[i] * v2[i];
    }
    return temp;
}
```

The matrix multiply function, with transposition now looks like:

```
void matrix_multiplyTranspose(
      double** m1, double** m2, double** result, size_t size)
{
    // create a matrix to hold the transposition of m2
    double** m2t = create_matrix(size);
    // perform transposition m2 into m2t
    for (size_t i = 0; i < size; i++)
    {
        for (size_t j = 0; j < size; j++)
        {
            m2t[j][i] = m2[i][j];
     }
    }
    // Now matrix multiply with the transposed matrix m2t
    for (size_t i = 0; i < size; i++)
    {
        for (size_t j = 0; j < size; j++)
        {
            result[i][j] = DOT(m1[i], m2t[j], size);
        }
    }
    // remove the matrix that holds the transposition of m2
    destroy_matrix(m2t, size);
}
```

Note, in a real implementation you might want to consider preserving the buffers allocate on the first call. Then on subsequent calls you check to see if the prior allocation was sufficient for the current call. If so, then bypass the allocation. If not then delete the buffers, and allocate new buffers. This is a further optimization detail left to the readers.

We will include the allocation/deallocation overhead in our charts. Also note, if the **m2** matrix is going to be used many times (e.g. in a filter system), then you can perform the transposition once, and then reused the transposed matrix many times. These are implementation strategies to keep in mind when taking these suggestions into your own code.

Let's see how the serial with transposition technique (ST) stacks up against the serial without transposition (S) and the parallel code (P) using the non-transposed matrix method:

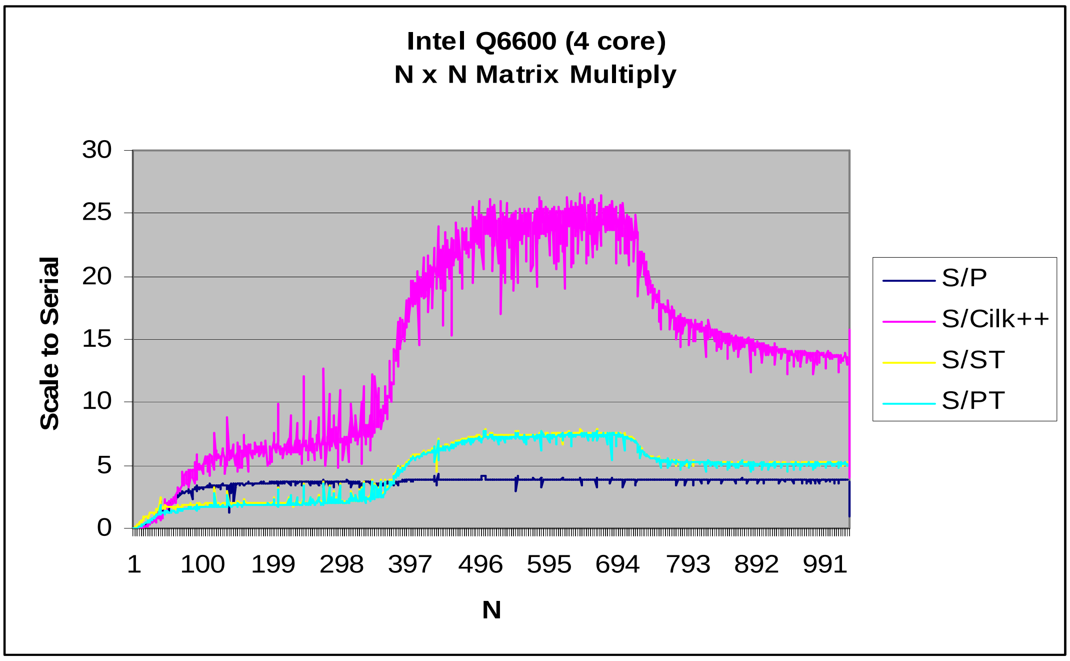


Fig 5 (above)

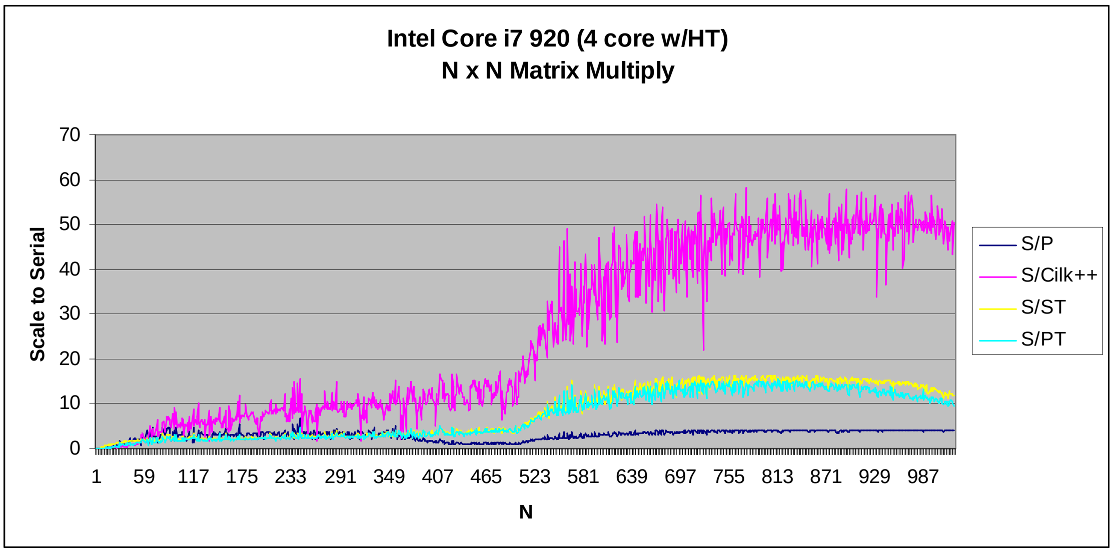


Fig 6

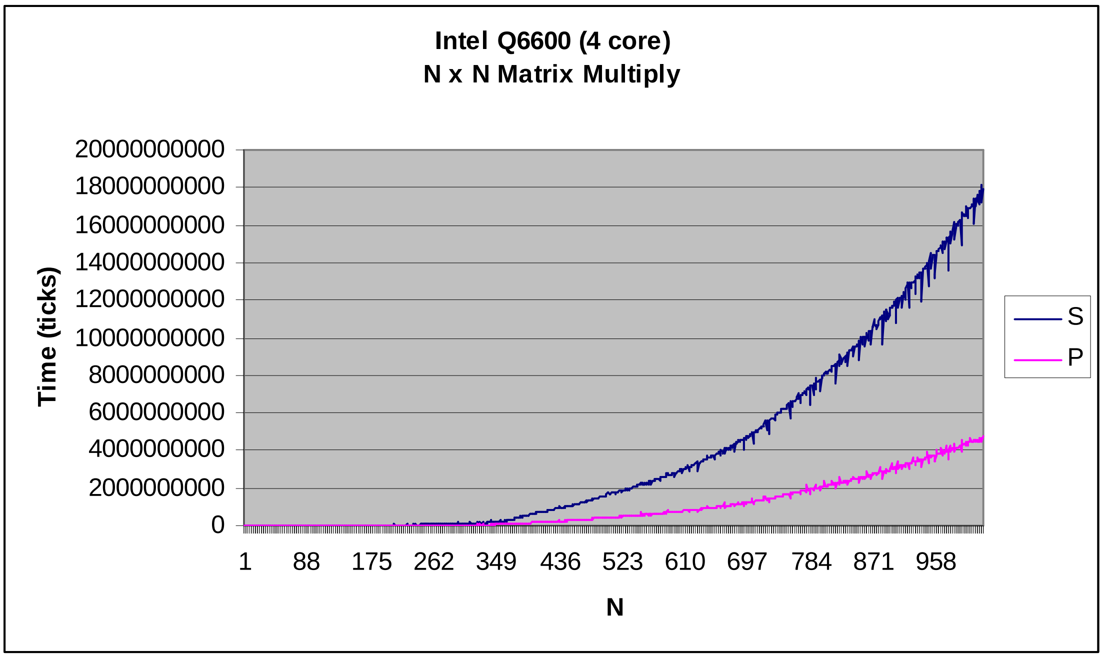
The revised technique, which still uses a row table is a significant improvement over the Serial and Parallel methods. The revised technique does not approach that of the row table elimination method of the Cilk++ demo program. Is there anything to learn from this?

The revised serial code, using one thread, overwhelms the parallel code using 4 or 8 threads on the two different processor at higher N values.

Although 4x better parallel performance than before, it is still less than the Cilk++ method, is this information useful in helping produce a better algorithm?

Why the "turbo charged" boost at 513 for the Serial Transpose method?
Although Parallel Transpose (PT) is 4x faster than Parallel (P) it is actually less than the performance of the Serial Transpose method! Why!?!?

The answer to this is the chart lines are relative to the original (non-transposed) Serial performance. Let's see the actual run time for the (non-transposed) Serial method to see if something shows up:



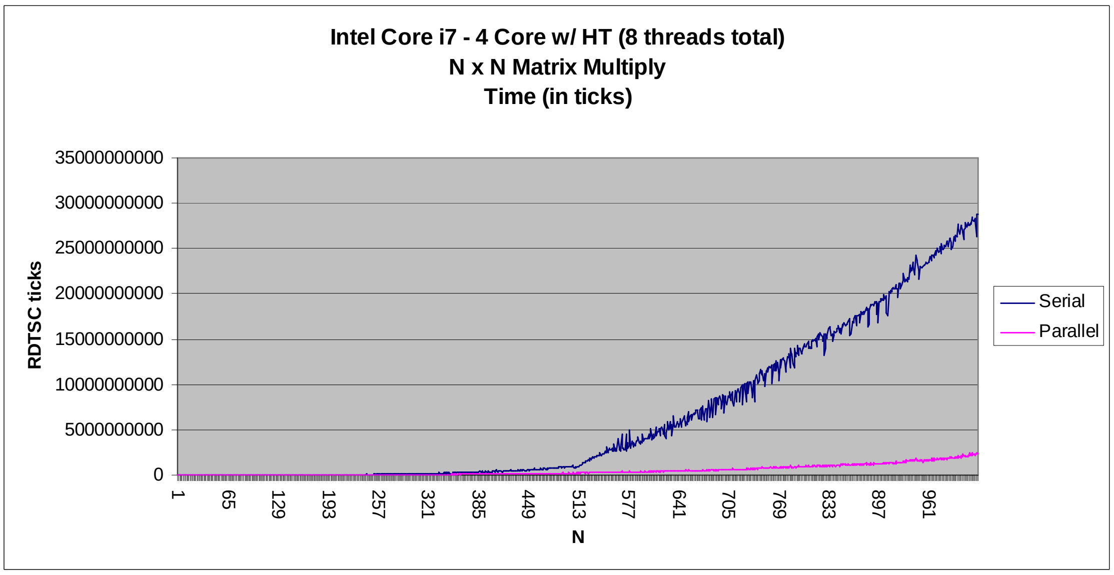


Fig 7

And there it is. At N = 513 we find the serial time experiencing a drastic change in slope. (Q6600 shows earlier slope change at about 350).

What is significant about N = 513 on Core i7 920?. We principally have access to two arrays of doubles dimension at 513 x 513. The number of bytes is 2 x 513 x 513 x 8 = 4,210,704. Hmm….

The Core i7 920 from specifications gleaned from the internet indicate that this processor has

L1 cache 32KB instruction + 32KB data (one for each core)
L2 cache 256KB (one for each core)
L3 cache 8MB (shared)

So why the performance drop at ~4MB instead of ~8MB? To be honest, I haven't performed an thorough analysis of this, my postulation is this is a TLB issue, so I will simply state that it appears that the cache system is being over taxed at this point. What is important, is the steep rise in the curve at this point accounts for the superscalar performance gain of a better serial technique over a non-optimal serial technique.

The important piece of information learned is the Serial Transpose method trashes the performance of the parallel technique using the non-transformed technique. (Although the Cilk++ technique is still 3.nnx faster that either)

Now when the Cilk++ row table elimination method is compared to Serial Transpose method we have:

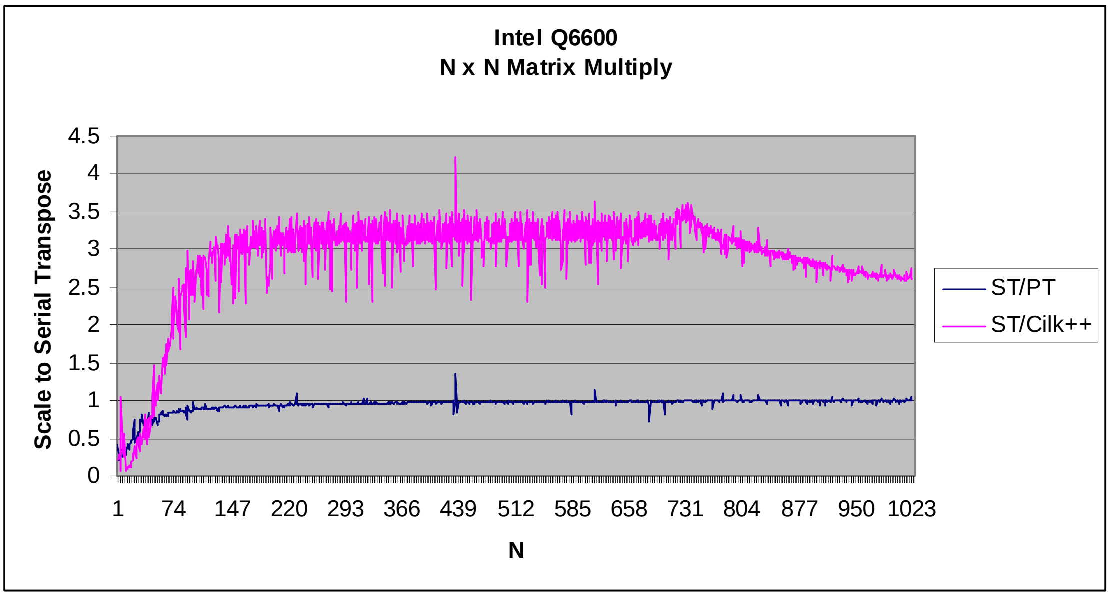


Fig 8

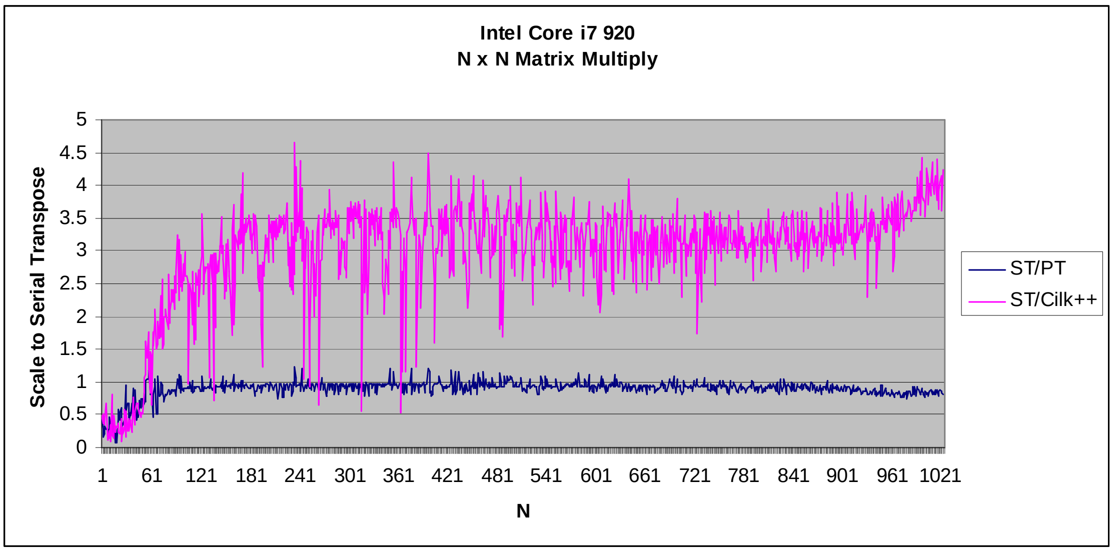


Fig 9

The Cilk++ technique shows approximately 3.25x performance over Serial Transpose method. More noteworthy, the Parallel Transpose is not faster than the Serial Transpose method. Why? And more importantly, does it matter why?

As a general rule, when you observe your parallel code **not** performing better than your serial code, this is a good indication that you have reached a memory bandwidth problem. The Cilk++ code performance clearly indicates the memory bandwidth problem is due to poor cache utilization by the Parallel Transpose technique.

Is the Cilk++ method the best we can expect to do?

The answer to this is: No.

Part 3

In the previous section we have seen that by reorganizing the loops and with use of temporary array we can observe a performance gain with SSE small vector optimizations (compiler does this) but a larger gain came from better cache utilization due to the layout change and array access order. The improvements pushed us into a memory bandwidth limitation whereby the Serial method now outperforms the Parallel method (of the Serial method).

The memory bandwidth limitation is an important factor to consider and warrants a change in programming strategy: In order to attain additional performance gains we are going to attack the problem from the perspective of trying to keep the data access patterns to the closest cache level.

But how are we going to do this?

On a system with Hyper Threading, such as the Core i7 920, the Hyper Threads within a single core share the 256KB L2 cache and, depending in internal architecture, share the 32KB L1 data cache. While all cores within the socket share the 8MB L3 cache (6MB or 12MB on other processor models).

The Cilk++ method uses the common practice of “divide and concur” to partition (or tile) the matrix into smaller working sets that nicely fit into the cache available to a thread. This is a good starting point. What we need to look at next is how to coordinate the activity *between* caches within the processor(s) of the system.

While you can tile using nested loops, consider this: As you reduce the tile size, you also increase the number of interactions with the thread scheduler (entering and exiting the inner most loop). Thread scheduling is not cheap. In the first chart, the overhead break even occurred at 50 x 50 tile size. Is there a different way to program such that you can use 2x2 or 2x1 or 1x2 tiles and attain superior performance?

The answer to this is: Yes

Targeting tasks to specific threads, or grouping of threads is difficult to do effectively using most threading tool kits (e.g. MS Parallel Collections, OpenMP, or Cilk++). Cache level task grouping is a built-in design feature of QuickThread. Example:

```
// divide the iteration space across each multi-core processor
// L3$ specifies by L3 cache
//    .OR.
// within processor (Socket) for processors without L3 cache
parallel_for( OneEach_L3$, intptr_t(0), size,
    [&](intptr_t iBeginL3, intptr_t iEndL3)
    {
        // divide our L3 iteration space by L2 within this threads L3
        parallel_for( OneEach_Within_L3$ + L2$, iBeginL3, iEndL3,
            [&](intptr_t iBeginL2, intptr_t iEndL2)
            {
                // Here we are running as the Master thread of a
                // 2 team member team (or 1 in the event of older
                // processor)
                //
                // Now bring in our other team member(s)
                // form team of threads sharing this threads L2 cache
                parallel_distribute( L2$,
                    [&](intptr_t iTMinL2, intptr_t nTMinL2)
                    {
                        // ... (Do Work)
                    } // [&](intptr_t iTMinL2, intptr_t nTMinL2)
                ); // parallel_distribute( L2$,
            } // [&](intptr_t iBeginL2, intptr_t iEndL2)
        ); // parallel_for( OneEach_Within_L3$ + L2$, iBeginL3, iEndL3,
    } // [&](intptr_t iBeginL3, intptr_t iEndL3)
); // parallel_for( OneEach_L3$, intptr_t(0), size,
```

The outer loop is a per socket division, the next deeper loop is a per L2 cache, and the inner layer is a n-Way split by the threads sharing L2 (2 threads on Core i7 920, 2 threads on Q6600)

The technique is to use the **parallel_distribute** as the inner most division, then use the loop partitioning of the outer loops within a state machine loop placed inside the **parallel_distribute** level. Say what?

The parallel_for in QuickThread performs the task to thread set selection and partitioning of the iteration space, but specifically does not drive the iteration. Due to this feature of QuickThread, the iteration domain can be passed deeper into the loop nest levels. The second loop does the same thing (pass iteration space to inner most parallel_distribute. Now as to why this is important.

This alternate technique creates the environment for an intelligent swarm of threads performing a coordinated attack of the problem. These threads know which threads share the nearest cache, which threads share each of the largest cache(s), and which threads are not sharing caches with the current thread. This identification is implicit by the sub range of the outer two loops and by the team member number (`iTMinL2`) within the **parallel_distribute**.

Adding an additional outer layer loop per NUMA node could easily be handled using:

```
    parallel_for( OneEach_M0$, intptr_t(0), size,
   ...
```

However, for this matrix multiplication, this would not provide any additional benefit unless your system has an L4 cache external to the processor and which is also shared amongst processors sharing the same NUMA node. NUMA aware issues are best handled in the memory allocation routines which can be placed within the L3 cache distribution layer of the above loop structure.

The interior state machine loop will partition (tile) the output array:

Socket (L3 cache)
Core Pair/HT Siblings (L2 cache/L1 cache)
Amongst Core Pair/HT Siblings (L2 cache/L1 cache)

To accomplish this, we segment the matrix into 2x2 tiles with each 2x2 tile being serviced by a 2 thread team. The 2 thread team I will call a "Tag Team" (as in Tag Team Wrestlers). On the Core i7 we will have 4 tag teams, one for each core, and the two threads for each team being Hyper Thread siblings within the same core. On Q6600 the tag team becomes the two threads sharing the same L2 (Q6600 has 2 L2 caches, each shared by 2 cores).

If you ever watch TV wrestling, there is a type of wrestling format where each team consists of two team members. Each team is *supposed to* have one member in the wrestling ring (square) at any one time, and they must tag their team mate when they wish to let them have a go at their opponent. I say *supposed to* in italics since more often than not, the tag exchange usually ends up with the team doing the tag cheating by having two wrestlers in the ring at one time. And in these situations they completely overwhelm their opponent. I am going to show you how to do the equivalent of this using cache directed thread scheduling as available with QuickThread.

The output matrix is divided into 2x2 tiles. The thread teams are derived from the thread pool into 2 thread teams, each thread sharing the closest cache level possible. On Core i7 920 these are Hyper Thread siblings within each core, on Intel Q6600 these are two cores sharing same L2 cache, on AMD Opteron 270 these are two cores within same processor on same NUMA node, etc… An optimization strategy will be employed whereby we will concurrently perform the transposition and DOT products of some of the cells. Then perform the DOT products on the remaining cells.

The Hyper Thread siblings within one core have two integer execution paths, but only one floating point execution path. We code to take advantage of this by having one thread of the tag team, called the Master thread, perform the transpositions using the integer execution path while the other Hyper Thread sibling, called the Slave thread, is performing the DOT product of the lower 1x2 portion of the 2x2 tile concurrently with the transposition (shortly following the transposition of each cache line). Upon completion of the transposition, the Master thread will then begin the DOT product of the upper left cell of 2x2 tile, and is joined shortly thereafter by the Slave thread performing the DOT product on the upper right cell. For the cells handled in this manner (those performing the transposition), cost of the DOT product of half these cells is essentially free, since it occurs during memory latency of the transposition. For the DOT product of the other half of this 2x2 cell, the row and transposed column of one of these output cells is fully cached (potentially all in L1 cache) with the other output cell having the transposed column residing in L1 cache and leaving the row of this output un-cached.

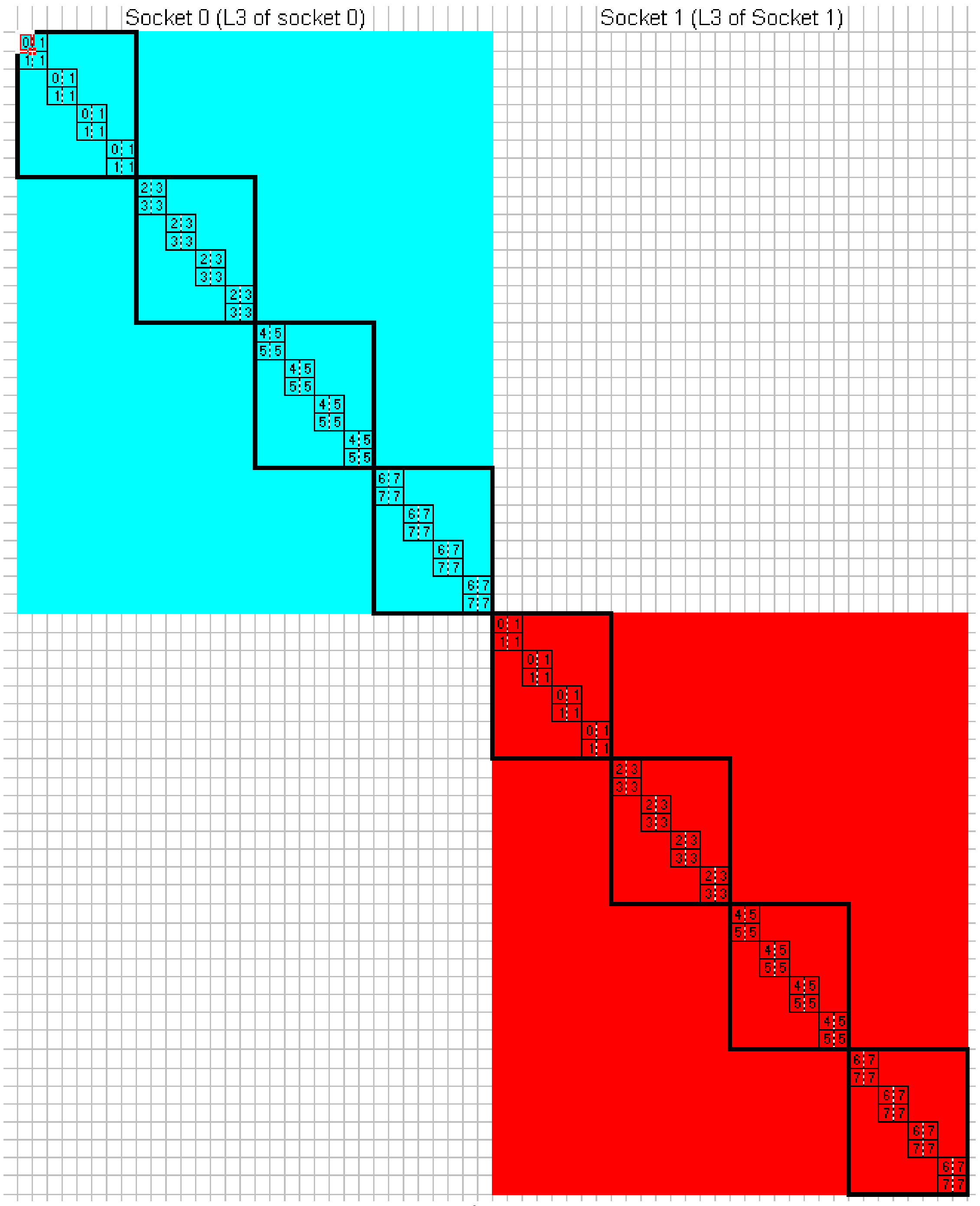
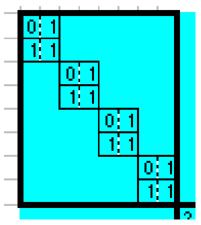


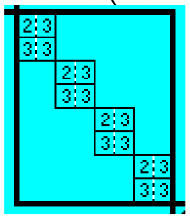
Fig 10

Figure 10 shows the initial Per Socket tile work assignment distribution (color backgrounds, an iterative per processor thread team (heavy outline), and 2x2 cell tile lightest outline.

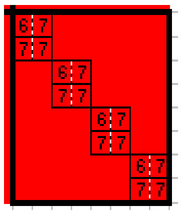
The 2x2 tiles that lie on the diagonal require additional work for the transposition of the m2 array into m2t array. The master thread (even number of 2 member team) performs the transposition (memory access intensive) using the integer execution unit of the HT core, while the other team member performs the DOT product of 3 of the cells in the 2x2 tile using the floating point execution unit of the HT core. These DOT products are performed concurrent with the transposition by means of snooping on the progress of the transposition made by the master thread of the tag team. The master thread of the 2 member team then performs the final DOT product. Some experimentation yet needs to be done to see if the Slave thread of the 2 member team ought to perform all 4 DOT products concurrent with the transposition. See below for the {transpose, 1x2},{1x1, 1x1} method:



The first two member thread team works on the upper left most 2x2 tile (and transposition) of its designated zone (shown above). Concurrent with this, the second team works on the upper left most tile (and transposition) of its designated zone:

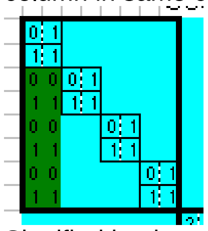


And so on to the last socket, last team working on the upper left most 2x2 tile (and transposition) of its designated zone:



The completion of these diagonal tiles are not signified by the end of a parallel construct. Instead, the completion is signified by writing a completion status into a mailbox. This completion will be asynchronous with the activities occurring by the other threads. And have the “overhead” induced by a non-Interlocked write to a shared memory mailbox.

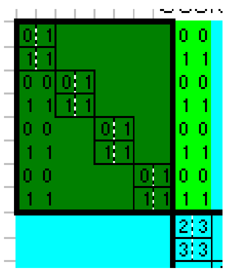
Once the upper left most tile of the diagonal is completed, the two member team consults a flag indicating if there is a work starved thread (early-on this flag will indicate no work starved threads. When there are no work starved threads, the two member team works on the 2x2 tiles in the column in same column of the diagonal tile just completed:



Signified by the green column above. However, each thread computes a 1x2 double DOT within the 2x2 tile. When that 2x2 tile is complete, the two member team consults a flag indicating if there is a work starved thread, if no work starved thread, the team continues down the column, if work starved, it progresses down the diagonal.

The goal of working down the column, just after transposition is the two columns just transposed are most likely to be residing hot in cache (L1, L2 or L3).

When a 2 member thread team completes its diagonal, and the column cells above/below its diagonals (within its two member team zone), The thread team then consults the transposition completed status of the other thread teams within its L3 cache (by way of mailbox)

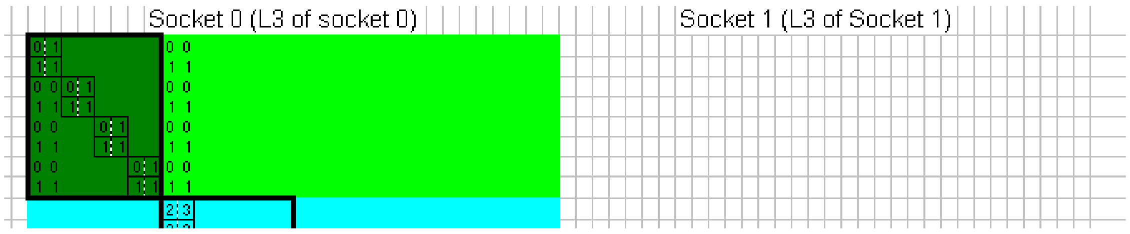


The light green column, is processed by the first team (0/1 in socket 0), after its work has complete, and after second team (2/3) has completed the upper left most diagonal. Which statistically will be done by the time the mailbox is consulted (after 4 transpose with DOTs, plus 12 double DOTs time delay).

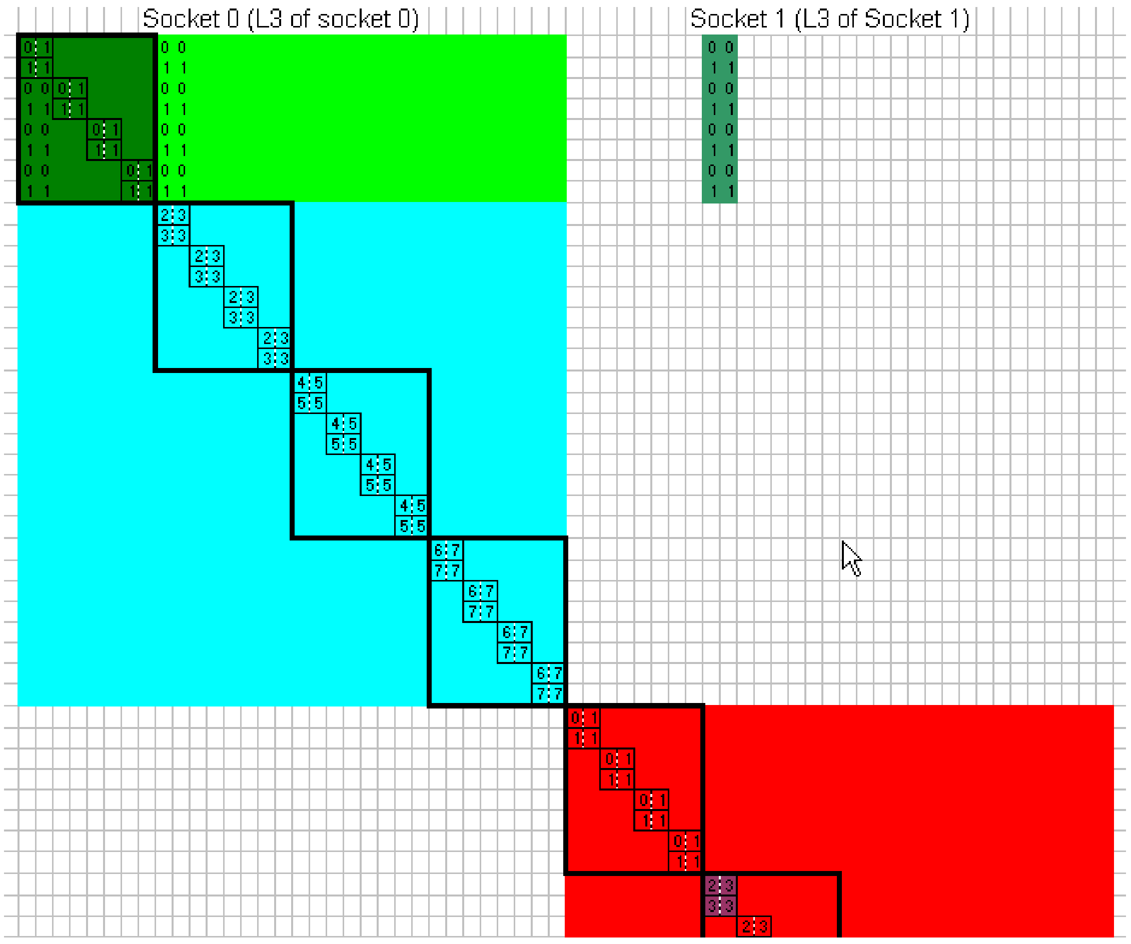
Each thread team does the same. As the thread team progresses over/under the columns of the diagonals of the teams sharing its L3 cache, if it finds an uncompleted designated column within its L3, it posts a “work starved” thread notification such that its L3 cache sharing teams can interrupt column processing and advance to next diagonal processing prior to completing its current column.

Each team, with exception for work starved thread indication, works independent of the other thread teams within its socket.

This iterative process continues until a thread team completes all the designated work within its socket tile:

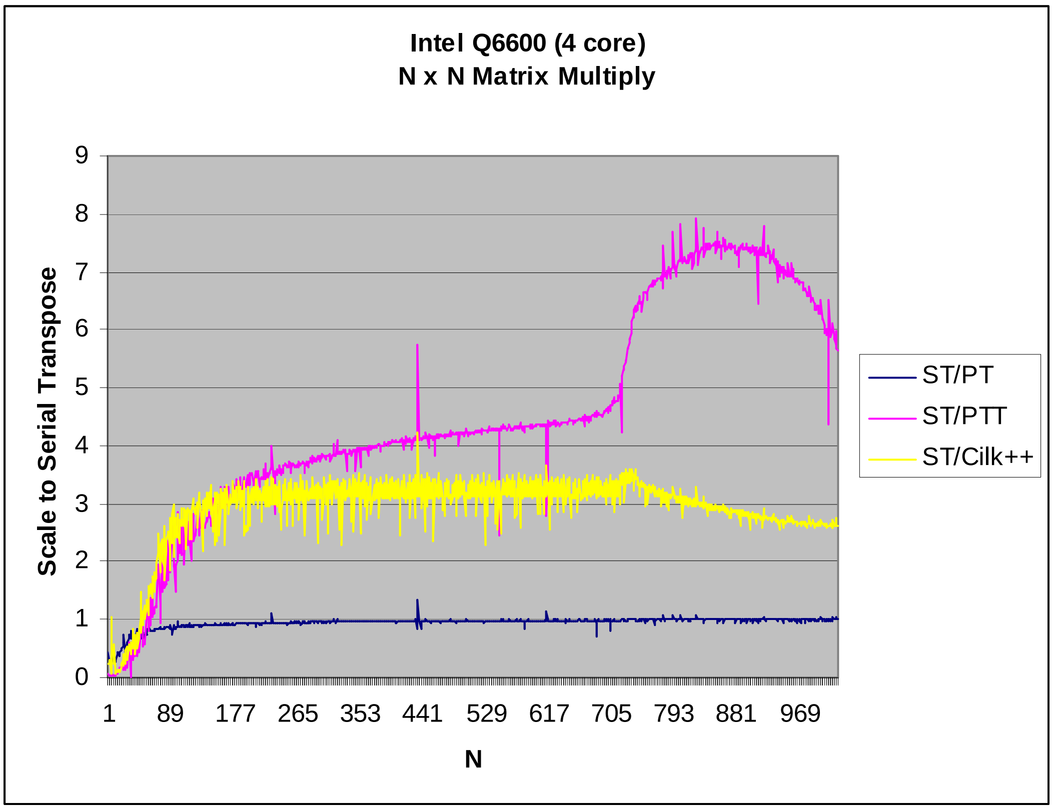


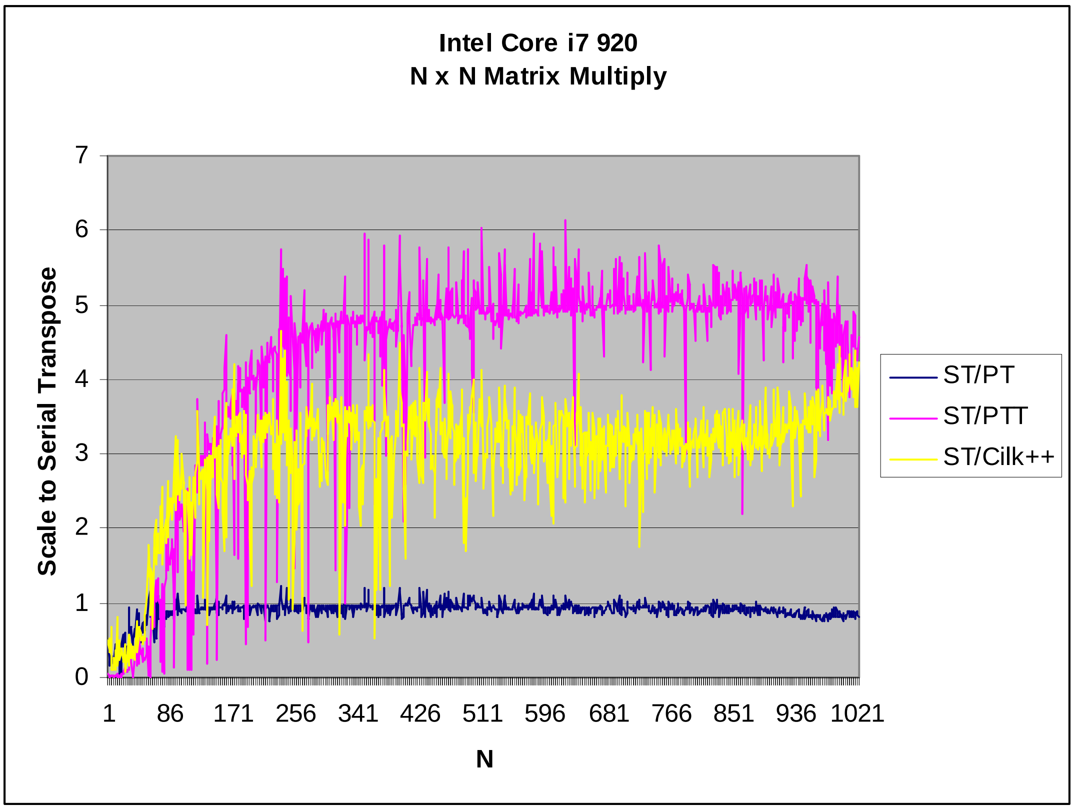
At this point, it now consults the diagonal completion status mailboxes for the diagonals of the other sockets. Being that this is a state machine instead of nested inner loops, the diagonal completion of the other sockets can occur in any order, but will tend to occur on diagonal 2x2 tiles in ascending order within each 2 member team, in each additional socket. As indicate by separated green bar, to right of Socket 0 team 0/1 zone in Socket 1 zone above completed diagonal for second team in red zone (socket 1) below.



As an additional optimization, on systems with NUMA capability, the rows of each array (m1, transpose m2 and output array) are segregated with NUMA considerations. In the 2 socket system, the upper half of the rows (turquoise/green) are in socket 0 locality, and the lower half (red) are socket 1 locality.

Now let's produce the charts and check the results data:


Intel Q6600 (4 core)
N x N Matrix Multiply
Scale to Serial Transpose
0
1
2
3
4
5
6
7
8
9
1
89
177
265
353
441
529
617
705
793
881
969
N
ST/PT
ST/PTT
ST/Cilk++


Intel Core i7 920
N x N Matrix Multiply
Scale to Serial Transpose
0
1
2
3
4
5
6
7
1
86
171
256
341
426
511
596
681
766
851
936
1021
N
ST/PT
ST/PTT
ST/Cilk++

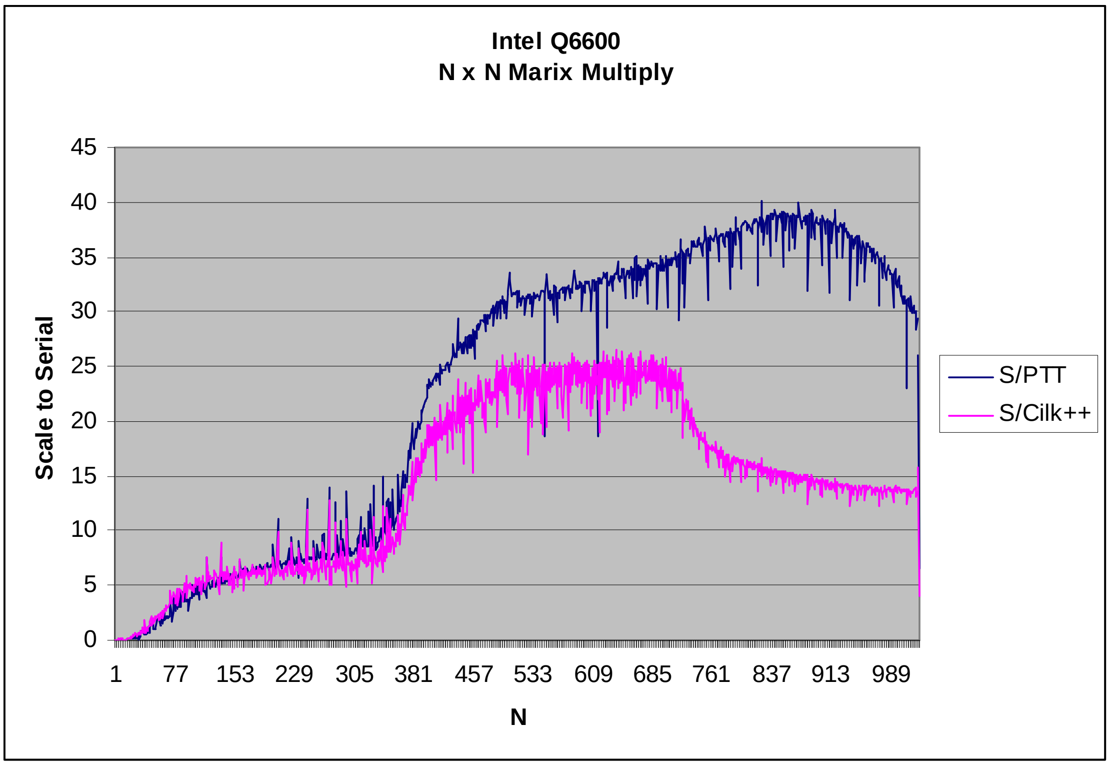
The average scale factors (compared to Serial Transpose) for N = 128 : 1024 are:

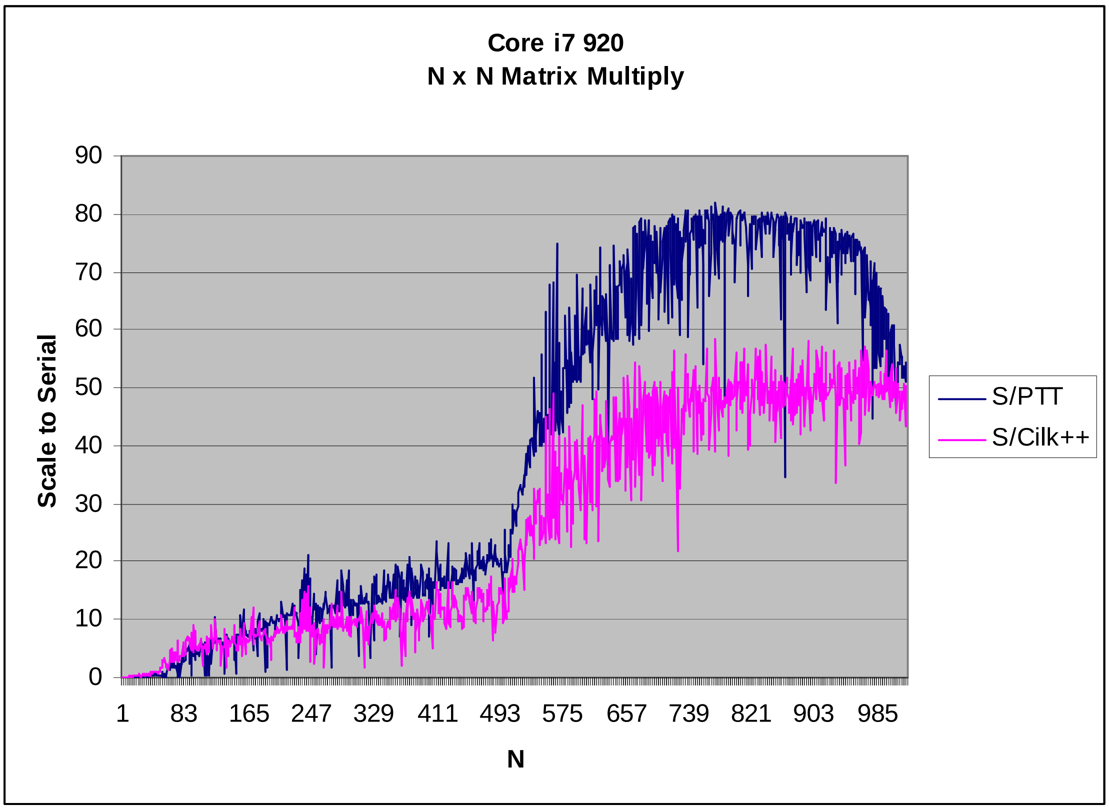
Q6600 4.96x for Parallel Transpose Tag Team and 3.06x for Cilk++
Core i7 920 4.69x for Parallel Transpose Tag Team and 3.20x for Cilk++

The peak values for selected section, ~880 of Q6600 shows ~ 7.5x for Parallel Tag Team vs 3x for Cilk++ method.

The Parallel Tag Team definitely has the advantage over the Cilk++ method (at least in this N range).

To inflate the figures, let's compare back to original Serial method without Transpose





We can now observe that at selected points in the chart we have attained a 38x improvement over Serial on Q6600 and near 80x improvement in scaling over the Serial on Core i7 920.

The complete description of Parallel Transposition Tag Team can be found in the sample code for MatrixMultiply.cpp included in the QuickThread demo kit. www.quickthreadprogramming.com

Can this be improved upon…

I will have to say, yes it can.

Early on in this blog post I mentioned: ”save intrinsic small vector functions for last”.

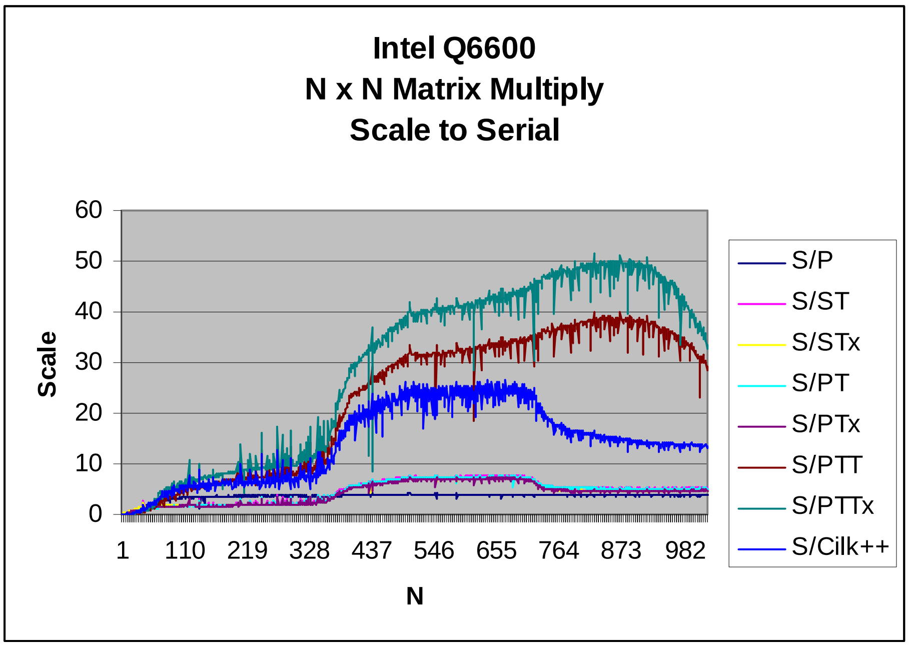
Internal to the MatrixMultiply, the inner most loop performs a DOT product. All the programming variations are using an inline function to perform the DOT product. The QuickThread Parallel Tag Team method (PTT) uses an additional variation on this DOT function to produce two DOT products at the same time. Each thread of the 2 thread team working on a 2x2 tile, produce results for half of this tile (1x2). Performing the two DOT products concurrently within each thread improves cache hit probability on the larger matrix sizes.

Below are the two DOT functions, the two functions modified to use xmm intrinsics (I am not very good at using xmm intrinsics, you may be able to do better), and two selector functions that call the appropriate DOT within the test program.

```
// compute DOT product of two vectors, returns result as double
// used in QuickThread variations
double DOT(double v1[], double v2[], intptr_t size)
{
    double temp = 0.0;
    for(intptr_t i = 0; i < size; i++)
    {
       temp += v1[i] * v2[i];
    }
   return temp;
}

double xmmDOT(double v1[], double v2[], intptr_t size)
{
   // __declspec(align(16)) not working reliably for me
   double temp[4];
   intptr_t alignedTemp = (((intptr_t)&temp[0]) & 8) >> 3;

   __m128d  _temp = _mm_set_pd(0.0, 0.0);
   __m128d *_v1 = (__m128d *)v1;
   __m128d *_v2 = (__m128d *)v2;

   intptr_t halfSize = size / 2;

   for(intptr_t i = 0; i < halfSize; i++)
   {
      _temp = _mm_add_pd(_temp, _mm_mul_pd(_v1[i], _v2[i]));
   }
   // fix code to remove temp[4] array
   _mm_store_pd( &temp[alignedTemp], _temp);
   if(size & 1)
      temp[alignedTemp] += v1[size-1] * v2[size-1];

   return temp[alignedTemp] + temp[alignedTemp+1];
}

// compute two DOT products at once
// effectively
// r[0] = DOT(v1, v2, size);
// r[1] = DOT(v1, v3, size);
// except running both results at the same time
void DOTDOT(double v1[], double v2[], double v3[], double r[2],
intptr_t size)
{
   double   temp[2];
   temp[0] = 0.0;
   temp[1] = 0.0;
   for(int i=0; i < size; ++i)
   {
      temp[0] += v1[i] * v2[i];
      temp[1] += v1[i] * v3[i];
   } // for(int i=0; i < size; ++i)
   r[0] = temp[0];
   r[1] = temp[1];
}

// compute two DOT products at once
// effectively
// r[0] = DOT(v1, v2, size);
// r[1] = DOT(v1, v3, size);
// except running both results at the same time
```

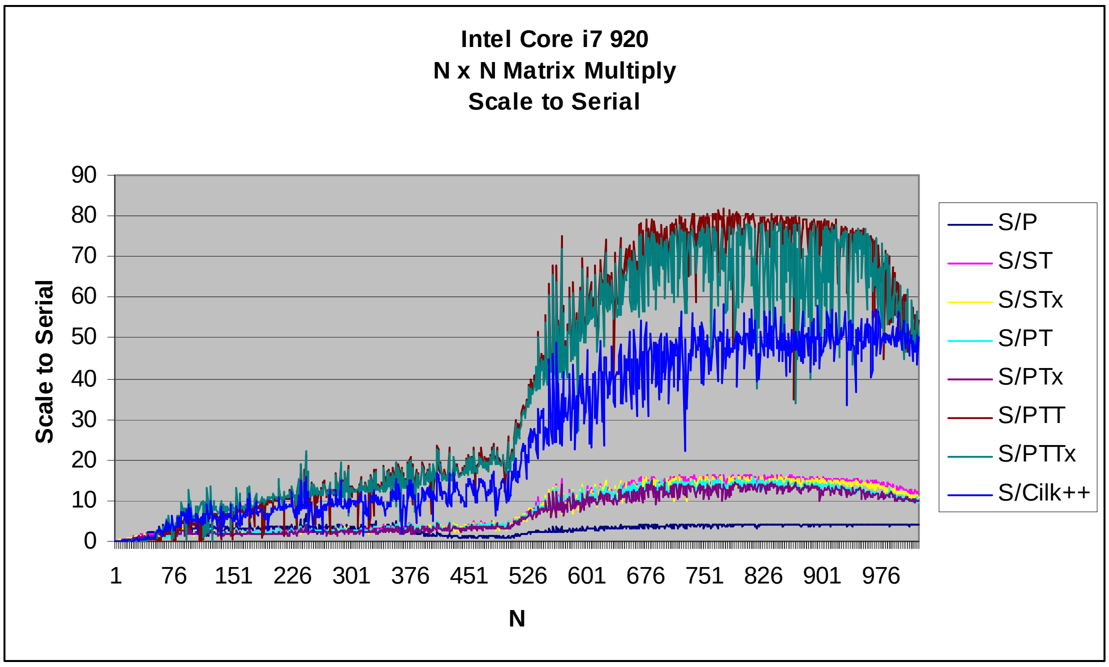
```
void xmmDOTDOT(double v1[], double v2[], double v3[], double r[2],
intptr_t size)
{
   // __declspec(align(16)) not working reliably for me
   double   temp[6];
   intptr_t alignedTemp = (((intptr_t)&temp[0]) & 8) >> 3;
   __m128d  _temp0 = _mm_set_pd(0.0, 0.0);
   __m128d  _temp1 = _mm_set_pd(0.0, 0.0);
   __m128d *_v1 = (__m128d *)v1;
   __m128d *_v2 = (__m128d *)v2;
   __m128d *_v3 = (__m128d *)v3;

   intptr_t halfSize = size / 2;

   for(intptr_t i = 0; i < halfSize; i++)
   {
      _temp0 = _mm_add_pd(_temp0, _mm_mul_pd(_v1[i], _v2[i]));
      _temp1 = _mm_add_pd(_temp1, _mm_mul_pd(_v1[i], _v3[i]));
   }
   _mm_store_pd( &temp[alignedTemp], _temp0);
   _mm_store_pd( &temp[alignedTemp+2], _temp1);
   if(size & 1)
   {
      temp[alignedTemp] += v1[size-1] * v2[size-1];
      temp[alignedTemp+2] += v1[size-1] * v3[size-1];
   }

   r[0] =  temp[alignedTemp] + temp[alignedTemp+1];
   r[1] =  temp[alignedTemp+2] + temp[alignedTemp+3];
}

bool UseXMM = false;

double doDOT(double v1[], double v2[], intptr_t size)
{
   if(UseXMM)
      return xmmDOT(v1, v2, size);
   return DOT(v1, v2, size);
}

void doDOTDOT(double v1[], double v2[], double v3[], double r[2],
intptr_t size)
{
   if(UseXMM)
      xmmDOTDOT(v1, v2, v3, r, size);
   else
      DOTDOT(v1, v2, v3, r, size);
}
```

As stated in the comments, I've experience an alignment issue with the compiler directive and had to resolve this with a little tweak of the code as a work around. The incorporation of the xmm intrinsic functions into these two routines were relatively easy (for an inexperienced xmm programmer like myself). Let's look at the results:



On the Q6600 (4 core no HT) the S/PTTx with the xmm intrinsic functions clearly exhibited a benefit, approximately 30% improvement. The QuickThread Parallel Tag Team "x" method preponderantly uses the xmmDOTDOT (double DOT function). However, the S/STx, using the xmmDOT (single DOT) function performs slightly worse than the C++ code without the intrinsic functions. This would indicate the compiler optimizations were better than the hand optimizations of an inexperienced intrinsic programmer (not unusual).

Looking at the Core i7 920 we see a different picture:



This processor, with HT, experienced a detriment in performance (-12%) in using the hand written xmm helper functions. Note, the executable was the same for both systems.

For you as a vendor of a program for use on various systems, this is an important piece of information. Knowing the platform specific behavior means you can query the system at program startup and then change the selection of which variant of the code to use. Typically you would involve selecting a functor (function pointer) in the code as opposed to using a selection function.

An alternate approach for unknown behavior is to use a heuristics approach. The application would select each variant of your code on the first few calls and measure the time of execution for each method, then after enough samples are taken, select the best performing variation of the functions and insert the appropriate functor into the dispatch pointer.

Can this be improved upon…

I will have to say, yes it can.

Note the chart lines are rather “noisy”. Additional tuning can improve the harmony and thus move the trend line upwards. This amounts to an estimated additional 15% over the current Parallel Transpose Tag Team method. (xmmDOTDOT on non-HT systems, DOTDOT on HT systems)

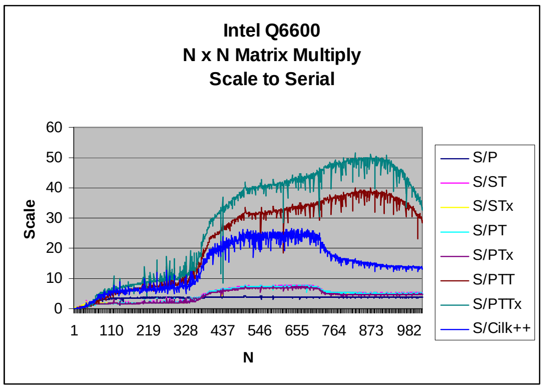
15% is usually not worth going after, however, note that both Parallel Tag Team method and the Cilk++ method appear to be dropping off at 1024. Additional tests should be run with larger matrixes, and more importantly on multi-socket systems. When I have an opportunity to collect such data, I will be in the position to publish updated information regarding this performance test.

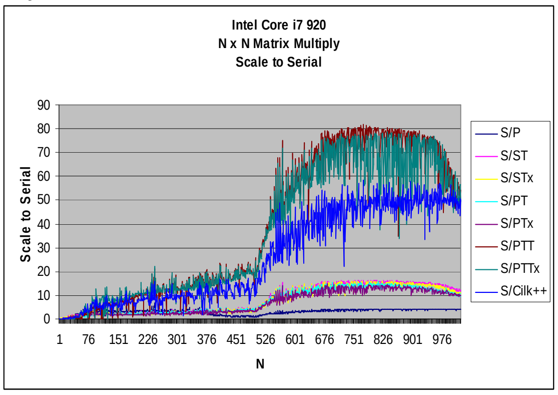
Is there a superior technique that can improve upon this?
History shows, the answer to this is yes.

In Part-4 we will examine issues relating to multi-socket systems.

Part 4

In the last installment we saw the effects of the QuickThread Parallel Tag Team method of Matrix Multiplication performed on two single processor systems:





Where the Intel Q6600 (4 core – no HT) with two cores (two threads) sharing L1 and L2 caches attained a 40x to 50x improvement over serial method, and in Intel Core i7 920 (4 core – with HT) and with four cores (eight threads) sharing one L3 cache and one core (two threads) sharing L1 and L2 caches attained 70x to 80x improvement. Let's see how this performs using two processors, each similar to Core i7 920.

When run on a Dual Xeon 5570 systems with 2 sockets and two L3 caches, each shared by four cores (8 threads). and each processor with four L2 and four L1 caches each shared by one core and 2 threads, we find:

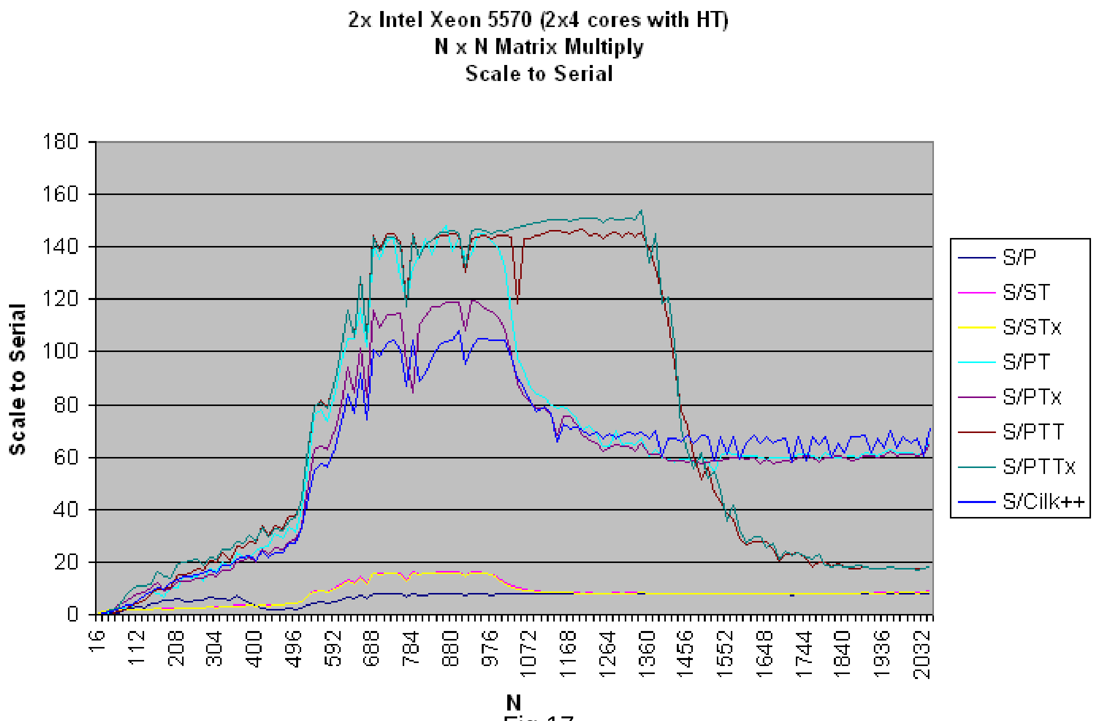


Fig 17

for a scale to serial of 140x to 150x in the N = 700 to 1344 range. The performance is almost twice that of the Core i7 920. This was somewhat expected.

There are some interesting observations to be made about this performance profile. While the 2x speed-up was expected, the Parallel Transpose method performed as well as the Parallel Tag Team method with N = 700 to 1024, then drops off precipitously. This is about half of the performance peak range of the Parallel Tag Team method (700 to 1344).

Why are the plateaus the same height?
What is the interpretation of the reason for the drop-off difference?

The plateaus are the same height for the same reason we saw in Fig 5 and Fig 6 where the Serial Transpose and the Parallel Transpose performance were essentially the same (yellow and red lines in Fig 17 above). The reason being: a resource bandwidth limitation. In Fig 5 and Fig 6 the limiting resource appeared to be memory bandwidth (due to Parallel Tag Team method having ample room to out perform Parallel Transpose). Due to the relative equalities of the plateaus (in N = 700 to 1024) some other resource than memory band width appears to be the limiting factor. This leaves cache access overhead or SSE Floating Point bottleneck.

Both of these bottlenecks will tend to clip the height of the performance curve but not the width. You can observe in the chart above that the two Parallel Tag Team methods managed to double the breadth of the peak performance curve thus permitting larger matrices to be handled effectively by the program. The reason for the increase in the breadth (larger matrixes handled) is principally due to more effective reuse of cached data due to the solution path through the problem (sequence in which computations are made).

The insight learned from Fig 17 is: When your problem working data set exceeds that of the cache system, you may find some paths to the solution more efficient than a simple nested loop.

In the 5th article we will explore how we can extend the performance curve to handle larger matrixes. Will this involve more cores/CPUs and/or different solution path?

Part 5

In the part 4 we saw the effects of the QuickThread Parallel Tag Team Transpose method of Matrix Multiplication performed on a Dual Xeon 5570 systems with 2 sockets and two L3 caches, each shared by four cores (8 threads). and each processor with four L2 and four L1 caches each shared by one core and 2 threads, we find:

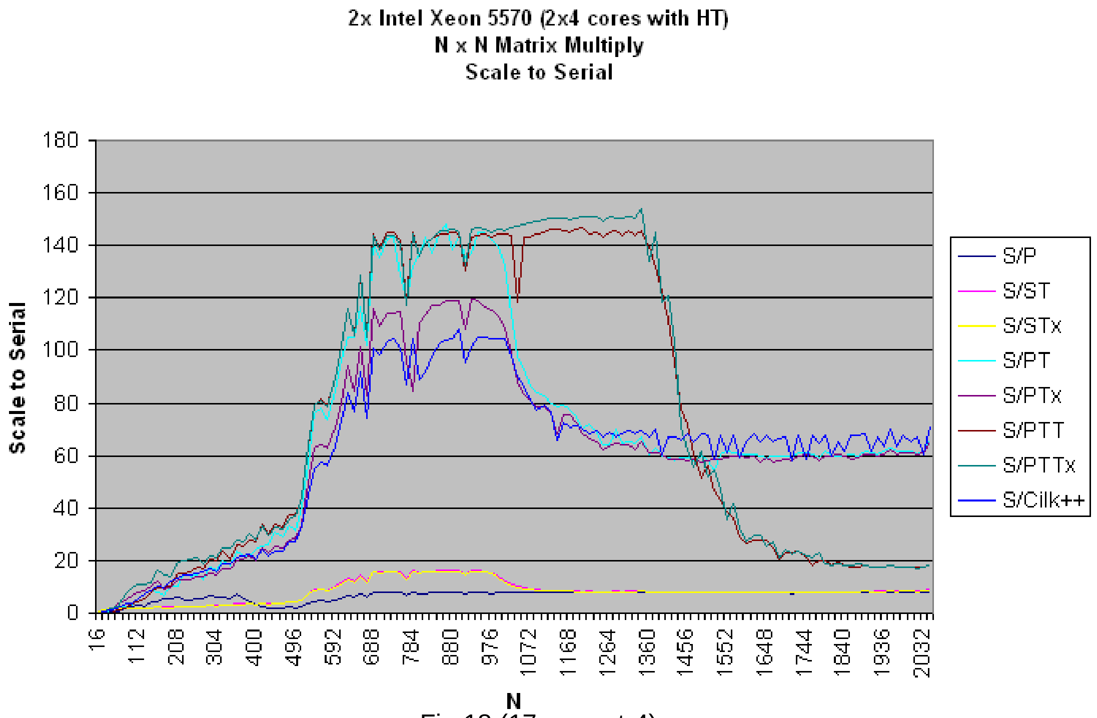


Fig 18 (17 on part-4)

The Intel Many Core Testing Laboratory was kind enough to provide me some time using their systems. http://software.intel.com/en-us/articles/intel-many-core-testing-lab/

Running the same method (sans Cilk++) on a 4 processor Intel Xeon 7560, each processor with 8 cores plus Hyper Threading (total of 32 cores, 64 threads) we observe:

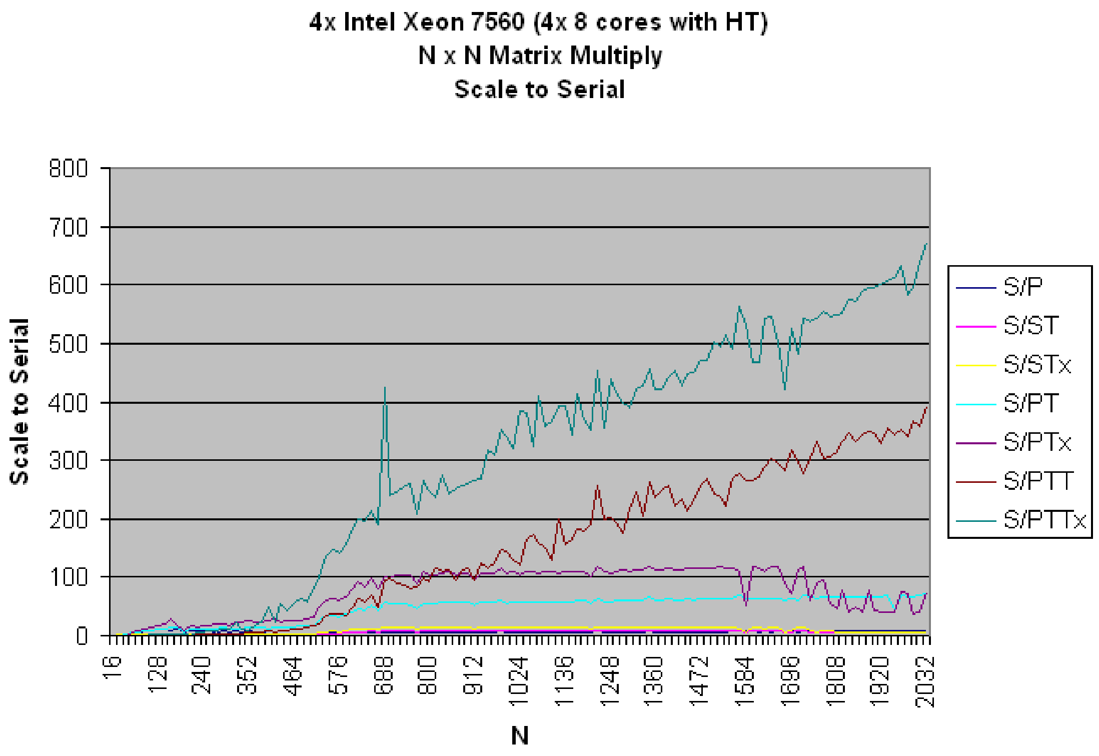


Fig. 19

In this chart we do not see a plateau in the scaling. This is due to the problem size at N=2048 being fully contained within the system cache. **Caution** - keep in mind that the above chart represents the scaling to the cache insensitive Serial method.

When comparing this to the cache sensitive Serial Transpose method we find a different set of results:

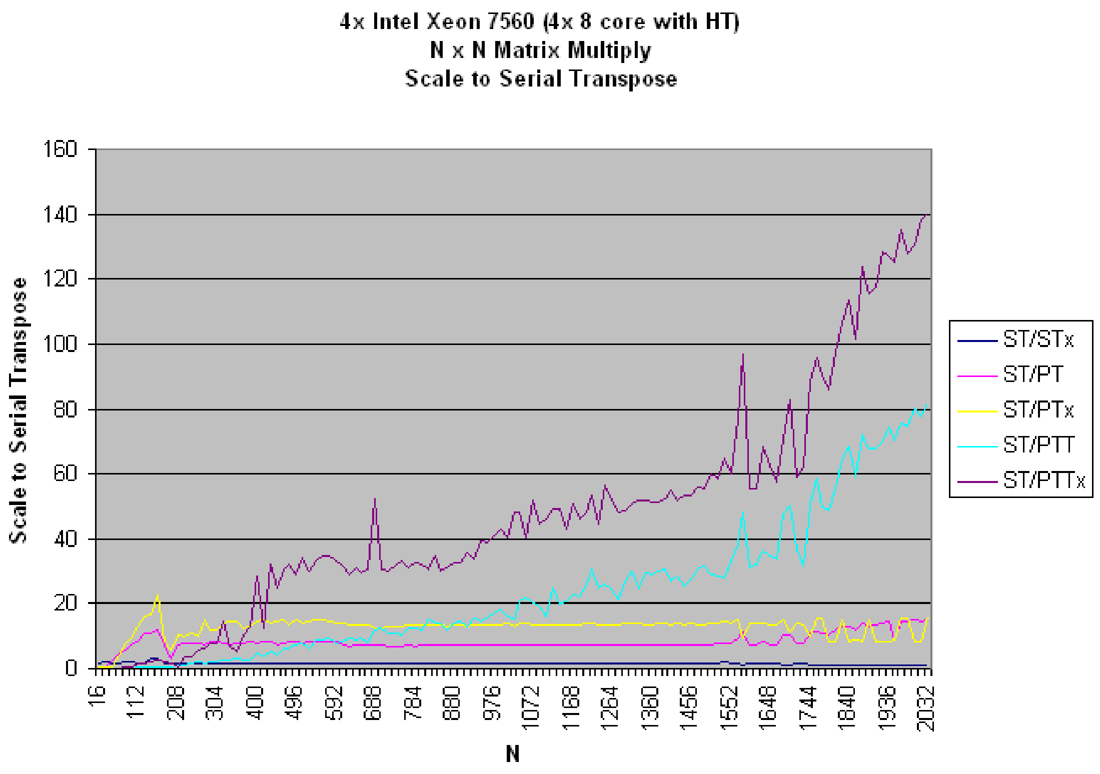


Fig 20

The sharp change in slope at N=1500-1700 is mainly due to the drop in performance of the reference data of the Serial Transpose method, rather than due to improvement in PTTx.

Looking at scaling factors (parallel performance / number of hardware threads) is often used as a decision factor in making a purchase. Let's look at the scaling factor charts:

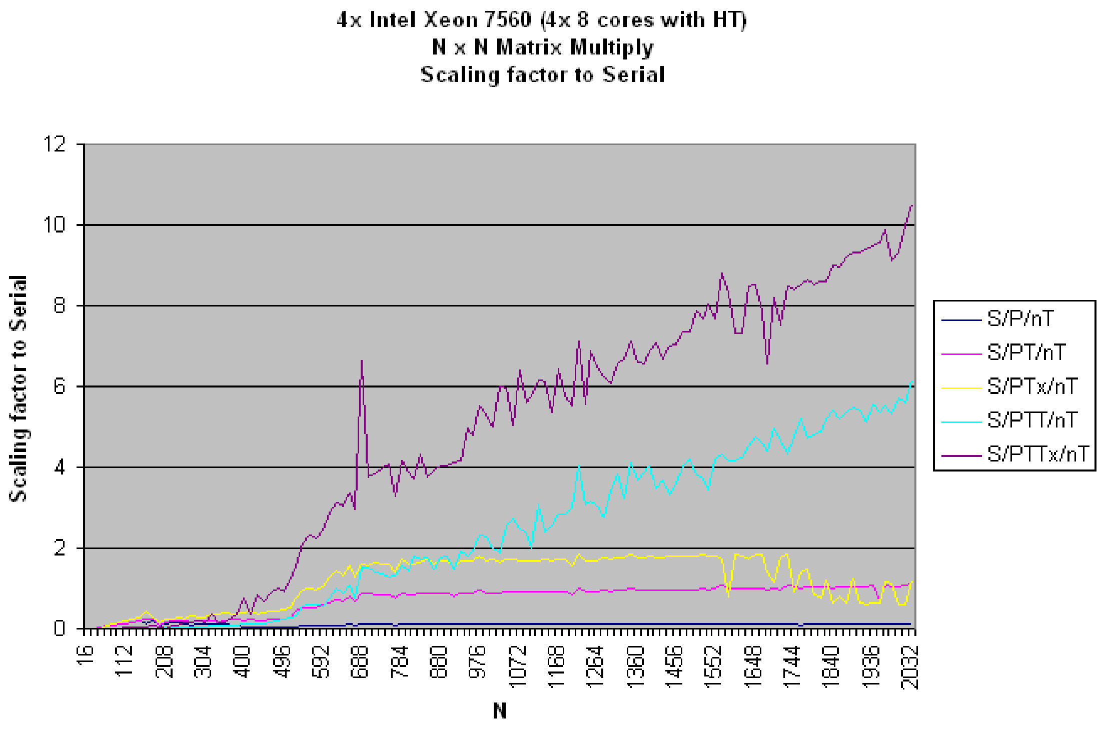


Fig 21

We find that as the problem size increases we observe a nice positive slope on the scaling factor. This looks exceptionally good. Too good to be believed. It is important to remember that the Serial method is not cache sensitive and is not a valid base line for comparison.

When we produce the scaling factor as compared to the cache friendly Serial Transpose method we find a completely different picture:

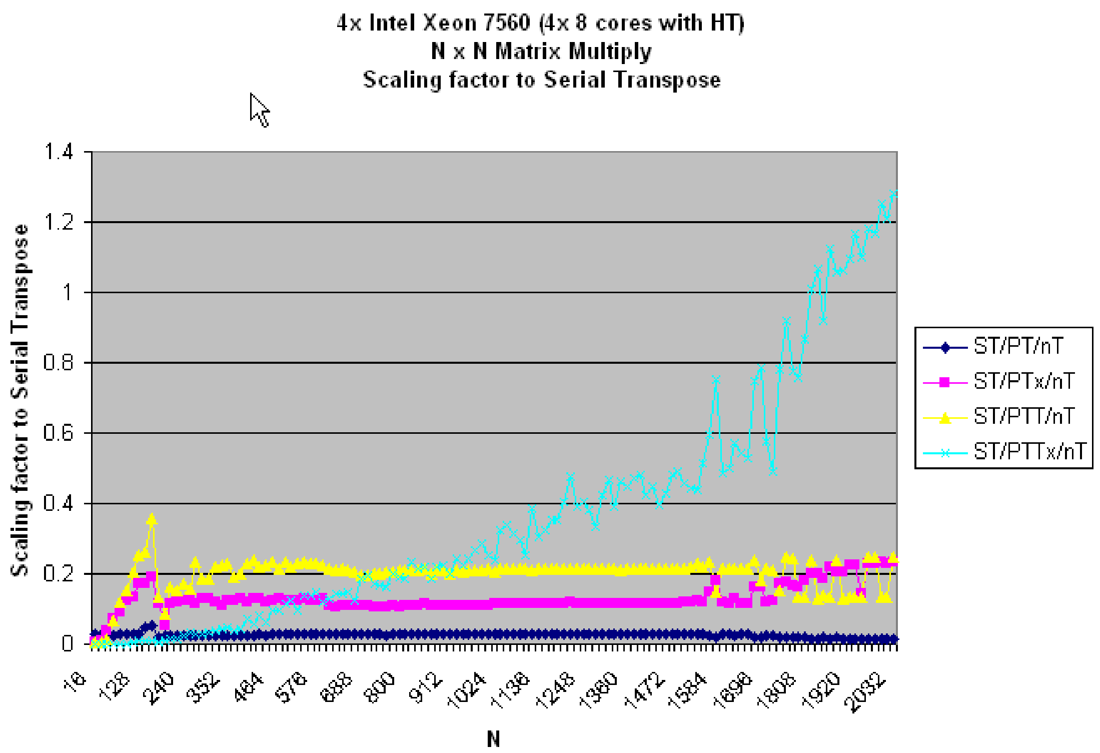


Fig 22

This chart will really deflate your programmer's ego. After all this hard work, we find that the scaling factor to a cache sensitive Serial Transpose method does not pay off (factor crosses 1) until N = 1824.

Comparing the factors of the 2x Intel Xeon 5570, as factored against the Serial Transpose we find:

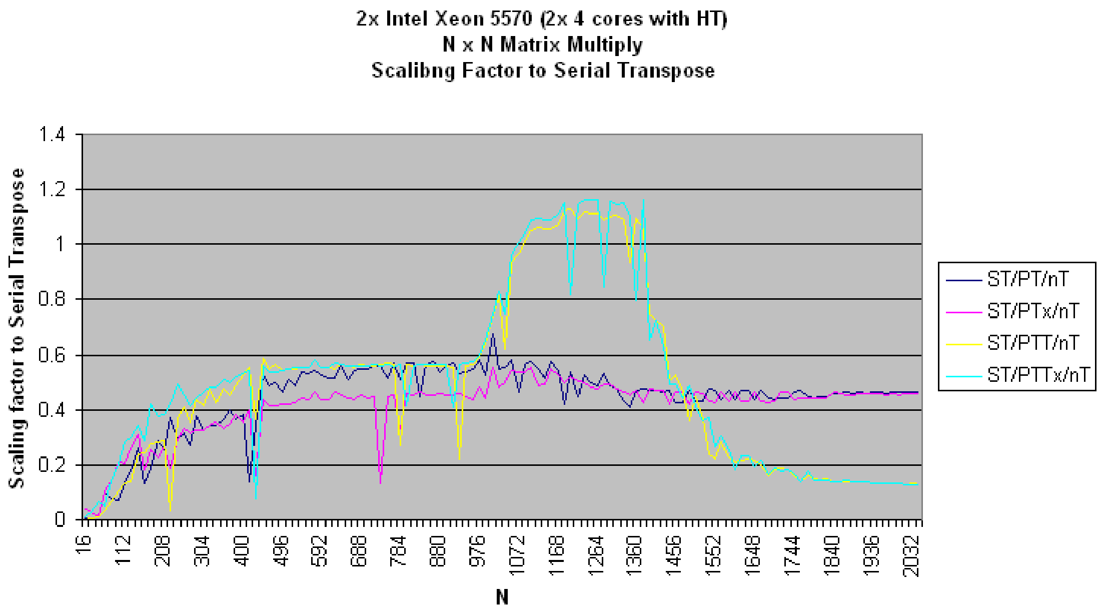


Fig 23

As expected, both systems can attain super scaling at different problem sizes. This is due to the different amount of cache memory on each system.

Although scaling factor provides a good perspective as to return on investment with respect to purchase of more processors, the scaling factor of one processor architecture is not meaningful when compared to a different processor architecture. A company ought to be interested in total return on investment, and this includes a time element.

When looking at the time element, we get a completely different picture. When comparing the fastest method (QuickThread Parallel Tag Team Transpose with SSE intrinsic functions) we find:

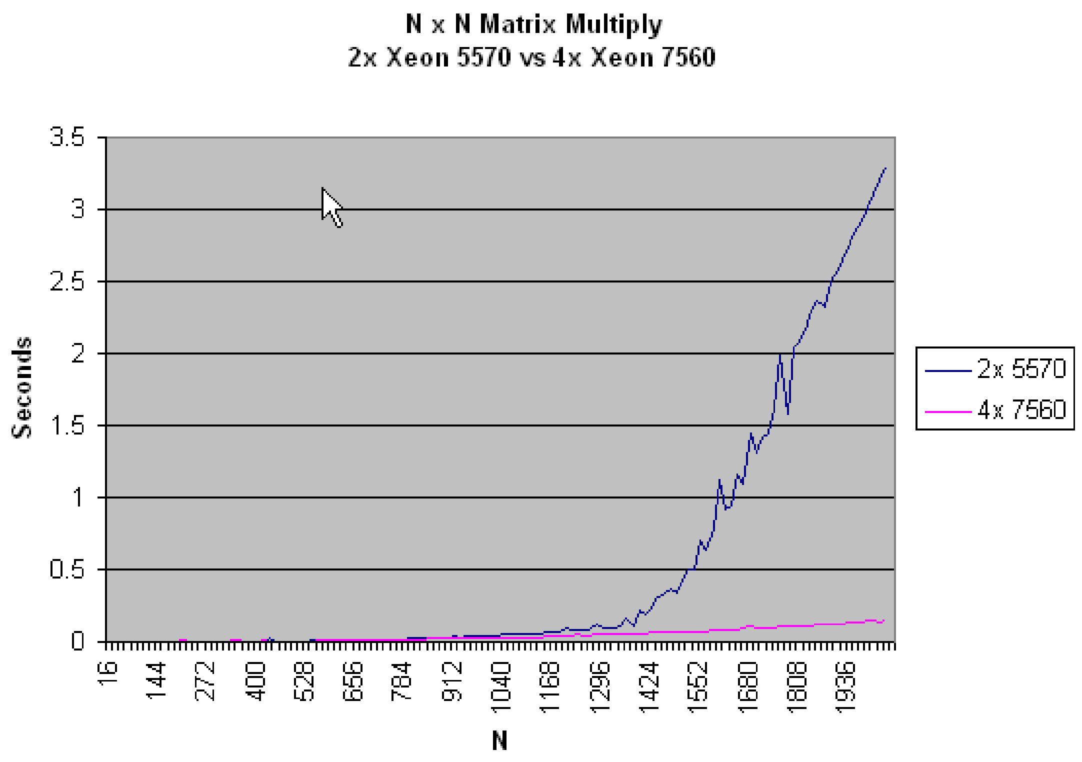


Fig 24

When including time, as a determination for cost benefit, we find that there is a rather drastic transition in the cost benefit ratio as you cross a particular threshold in the problem size (N=1400). The point being made here is to use appropriately sized test cases when making evaluations for purchase decisions. The cost/benefit and performance curves will not always be suitable for extrapolation.

When we run the fastest method (QuickThread Parallel Tag Team Transpose with SSE intrinsic functions) to larger matrix sizes we find

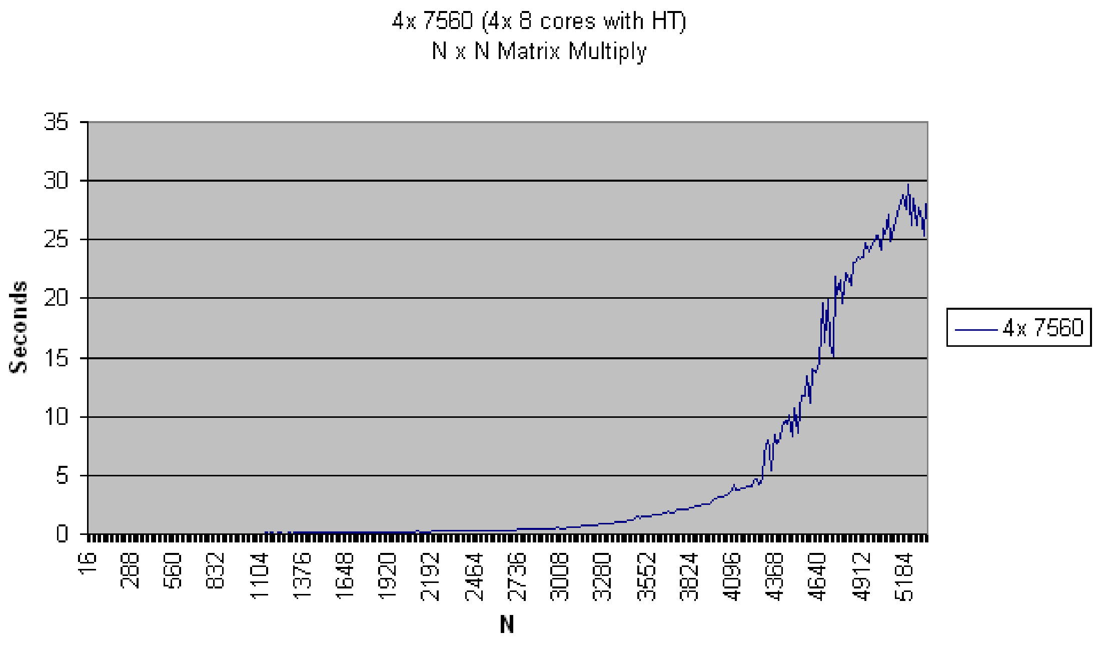


Fig 25

Matrixes up to N = 3000 to 4096 can be handled with 4 processors (32 cores / 64 threads), larger matrixes may require additional processors and/or a revised method.

Conclusions up to this point:

The fastest method: Parallel Tag Team Transpose with SSE intrinsic functions, relies on the QuickThread ability to schedule affinity pinned threads by cache level proximity. The ability to coordinate work using cache sensitivity can pay off big in your optimization strategies.

Larger matrix sizes could be handled in an improved manner with the same number of processors (4x 8 core with HT) when combined with an additional tiling strategy which will include additional overhead. This is typically called the *divide and concur* method, often used by parallel programmers.

Taking the matrix at N = 5200, and splitting it in two (both axis) yields a tile of N = 2600 and four such tiles. This requires 4 x 2 = 8 iterations using the smaller tiles. The matrix at N = 2600 took approximately 0.33 seconds to compute, therefore estimated computation time would be at 0.33 x 8 = 2.64. An estimated 10x improvement over the un-tiled method, but which may not be fast enough for your purposes.

Would divide and concur be the best strategy to use?

This depends on the interpretation of best.

In terms of relative performance return for effort in programming, this may be so. However, in looking at Fig 18 (17 on part-4), and comparing the Cilk++ to QuickThread Parallel Tag Team XMM method, we have demonstrated that by paying particular attention to cache locality, specifically, what’s in L1, L2 and L3 caches, and when it is in those caches, that you can attain an additional 1.4x to 2.5x performance boost in performance.

I will attempt to lay out the strategy which I believe will make effective use of the system caches. While the sketch below won't show the specific method, it will demonstrate the general plan of attack.

The current Parallel Tag Team (transpose) method divides the work by L3, then subsequently L2 regions and then takes an L1 friendly path in producing the results. This strategy works exceptionally well up until the size of the matrix reaches a point where the execution path begins to evict data from the L3 cache. It is my postulation that by employing a method where you follow the same path of the Parallel Tag Team (transpose) method, but impose a clipping technique on the distance from the diagonal, that you can minimize L3 cache evictions. The chart below illustrates a clipped L2 path through the current L3 workspace.

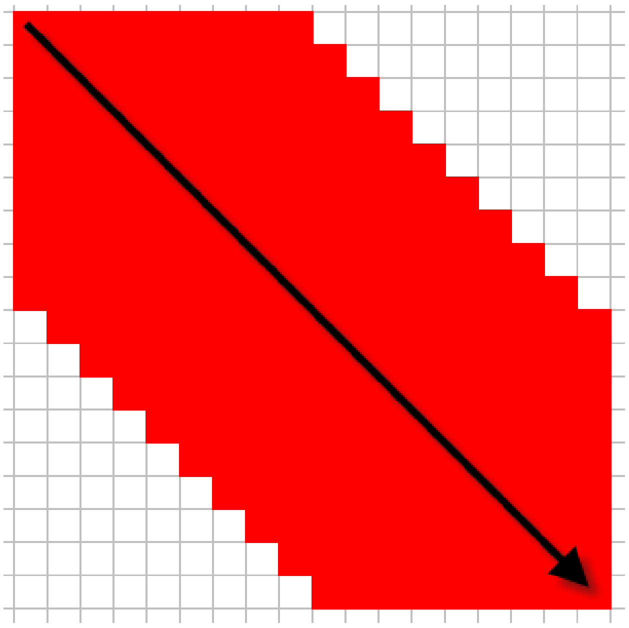


Fig 26

In the above chart, the general execution path follows the arrow. The colored (red) cells indicate the output cells who's results have been computed. The white cells indicate those output cells that have yet to be computed.

In the current Parallel Tag Team (transpose) method all of the above cells would have been colored, in the proposed method for large matrixes, a clipping technique limits the distance from the diagonal of the output cells to be computed while processing the diagonal. N.B. The above is a simplification of a 1P system.

In processing of a large matrix, had the computations included the empty cells, the computation would first suffer L2 cache evictions to L3, then at some size, eviction of L3. This is (postulation) possibly confirmed by Fig 25.

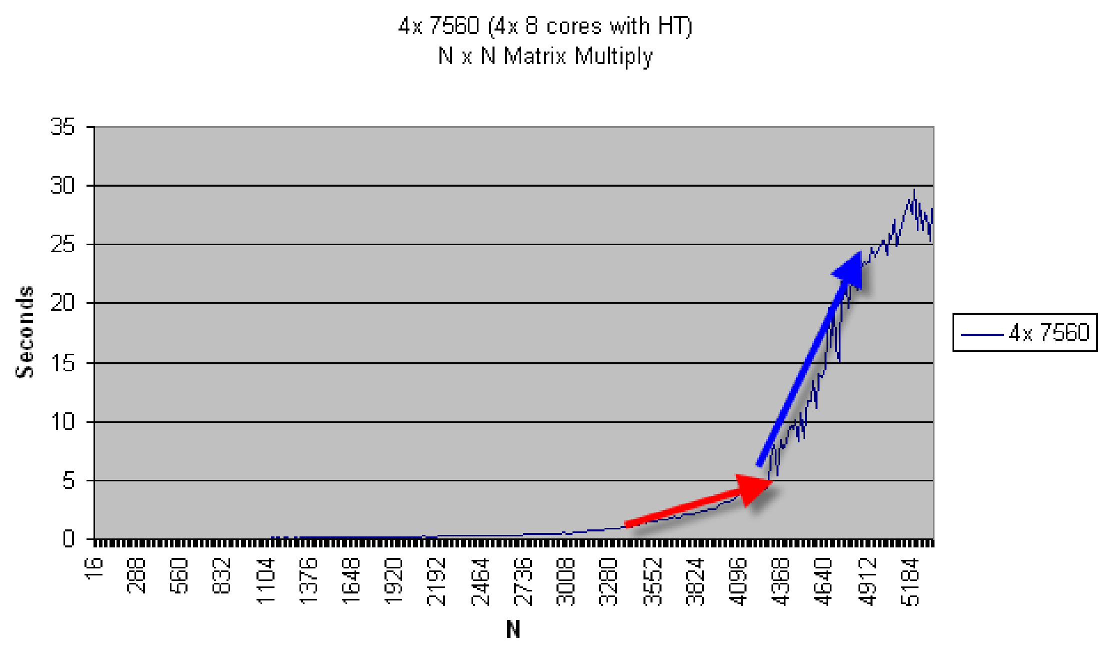


Fig 27

In above Fig 27 (Fig 25 with arrows added), the red arrow depicts L2 evictions and the blue arrow depicts L3 evictions.

Back to Fig 26. Upon completion of output cells in the Fig 26 we will find:

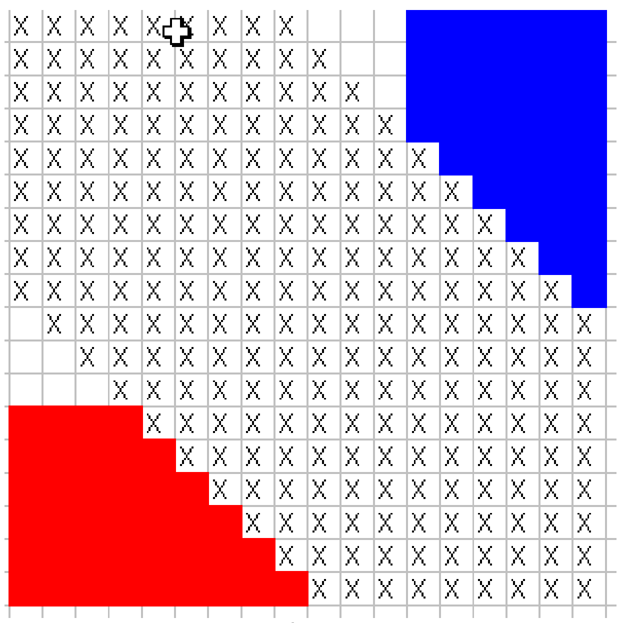


Fig 28

Where the X's mark the cells in the output L2 zone who's results have been completed. The blue cells represent columns (stored as rows) in the m2t array that are still residing in L2 cache, and the red cells represent row cells in the m1 array that are in the L2 cache. Additionally, (not depicted by colorization) some portion of the bottom row(s) and right most column(s) are still residing in the L1 cache. The remaining un-X'd white may, or may not, be residing in the L3 cache.

The next computation sequence (subject to verification) ought to follow the sequence as depicted by the arrows in Fig 29:

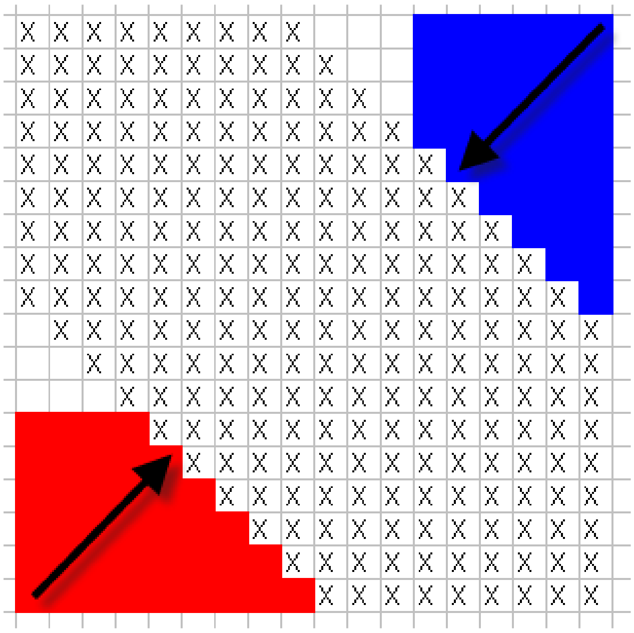


Fig 29

The red and blue ends of the output matrix should be processed in an alternating sequence as you progress along the arrows towards the first diagonal.

In the earlier mentioned divide and concur method (tiling), you would process 4 smaller tiles twice each or 8x the time of a smaller tile, presumably of a size found optimal for L2 cache size. The tiling method might benefit from L2 residual data resulting in a 6x to 8x run time of the smaller matrix as opposed to 8x the run time.

In the proposed method (call it cross diagonal), and for the size range depicted above in Figs 28 and 29, and based upon my prior experience with the Parallel Tag Team Transpose technique, it is estimated that it may be possible to produce the result in 1.5x to 2x the time of the smaller matrix. Potentially besting the divide and concur method (tiling) by a factor of 4x. It should be stressed that the actual differences may vary from this estimate. Extrapolation, as mentioned earlier, often does not follow the curve established by present data.

I hope you have found my series of articles insightful. This article cannot convey the detail of the QuickThread Parallel Tag Team Transpose XMM method whereas the code can convey this detail. For those interested in obtaining a copy of the code and a demo license for QuickThread feel free to contact me at my email address below. QuickThread runs on Windows and Linux systems. Both x32 and x64 for Windows but only x64 for Linux (Ubuntu and Red Hat).

Jim Dempsey
jim@quickthreadprogramming.com
www.quickthreadprogramming.com

Appendix (program)

The following is the source code use as the test program. Portions of this source code requires the QuickThread parallel programming toolkit. Formatting edits were perform for the purpose of this document (shorter tab spacing, long comments broken to multiple comment lines and long statements were reformatted with appropriate line breaks).

```
/*
 * MatrixMultiply.cpp
 *
 *  Created on: Jun 11, 2010
 *      Author: jim
 */
// derived from: http://msdn.microsoft.com/en-us/library/dd728073.aspx

#if defined(__linux)
// Linux dependent headers
// numa.h used by the
#include "numa.h"
#else
// compile with: /EHsc
#include <windows.h>
#endif

// #define USE_matrix_compare
#include <emmintrin.h>

// Headers for QuickThread
#include <QuickThread.h>
#include <parallel_task.h>
#include <parallel_for.h>
#include <parallel_distribute.h>
using namespace qt;  // QuickThread uses namespace qt

#include <iostream>
using namespace std;

#if defined(__linux)
#define USE_random
#endif

#if defined(USE_random)
#include <random>
#else
#include <boost\random.hpp>
using namespace boost;
#endif
```

```
long  PageSize = 0;        // Virtual Memory page size in bytes
int_Native nThreadsPerL2 = 0; // set at init time

double** m1 = NULL;
double** m2 = NULL;
double** resultSerial = NULL;
double** resultSerialt = NULL;
double** resultSerialtXMM = NULL;
double** resultParallel = NULL;
double** resultParallelt = NULL;
double** resultParalleltXMM = NULL;
double** resultParalleltt = NULL;
double** resultParallelttXMM = NULL;

qt_allocator<double> cacheAligned_doubles(64);

// On initializaton set to L1 cache line size 64, 128, 256, ...
int_Native  CacheLineSize_L1 = 0;   // (bytes)

// may be CacheLineSize_L1 or 2*CacheLineSize_L1
int_Native  CacheFlushLineSize = 0; // (bytes)

// Size of L1 cache (determined at init time)
int_Native  CacheSize_L1 = 0;    // (bytes)

// Size of L2 or L3 cache, whichever larger/present
int_Native  CacheSize_Larger = 0;   // (bytes)

// qtControl object used to access internal functions
qtControl   qtCtrl;

// Calls the provided work function
// and returns the number of milliseconds
// that it takes to call that function.
template <class Function>
uint64_t time_call(Function&& f)
{
   uint64_t begin = __rdtsc();;
   f();
   uint64_t end = __rdtsc();;
   return end - begin;
}

// Creates a square matrix with the given number of rows and columns.
double** create_matrix(intptr_t size);

// Frees the memory that was allocated for the given square matrix.
void destroy_matrix(double** m, intptr_t size);

// Initializes the given square matrix with values that are generated
// by the given generator function.
template <class Generator>
double** initialize_matrix(double** m, intptr_t size, Generator& gen);

// Computes the product of two square matrices.
// this the a text book standard matrix multiplication of square matrix
```

```
void matrix_multiply(double** m1, double** m2, double** result,
intptr_t size)
{
   for (intptr_t i = 0; i < size; i++)
   {
      for (intptr_t j = 0; j < size; j++)
      {
         double temp = 0;
         for (int k = 0; k < size; k++)
         {
            temp += m1[i][k] * m2[k][j];
         }
         result[i][j] = temp;
      }
   }
}

// compute DOT product of two vectors, returns result as double
// used in QuickThread variations

double DOT(double v1[], double v2[], intptr_t size)
{
    double temp = 0.0;
    for(intptr_t i = 0; i < size; i++)
    {
       temp += v1[i] * v2[i];
    }
   return temp;
}

double xmmDOT(double v1[], double v2[], intptr_t size)
{
   double temp[4];
   intptr_t alignedTemp = (((intptr_t)&temp[0]) & 8) >> 3;

   __m128d  _temp = _mm_set_pd(0.0, 0.0);
   __m128d *_v1 = (__m128d *)v1;
   __m128d *_v2 = (__m128d *)v2;

   intptr_t halfSize = size / 2;

   for(intptr_t i = 0; i < halfSize; i++)
   {
      _temp = _mm_add_pd(_temp, _mm_mul_pd(_v1[i], _v2[i]));
   }
   _mm_store_pd( &temp[alignedTemp], _temp);
   if(size & 1)
      temp[alignedTemp] += v1[size-1] * v2[size-1];

   return temp[alignedTemp] + temp[alignedTemp+1];
}


// compute two DOT products at once
// effectively
// r[0] = DOT(v1, v2, size);
// r[1] = DOT(v1, v3, size);
// except running both results at the same time
void DOTDOT(double v1[], double v2[], double v3[], double r[2],
intptr_t size)
{
   double   temp[2];
   temp[0] = 0.0;
   temp[1] = 0.0;
   for(intptr_t i=0; i < size; ++i)
   {
      temp[0] += v1[i] * v2[i];
      temp[1] += v1[i] * v3[i];
   } // for(intptr_t i=0; i < size; ++i)
   r[0] = temp[0];
   r[1] = temp[1];
}

// compute two DOT products at once
// effectively
// r[0] = DOT(v1, v2, size);
// r[1] = DOT(v1, v3, size);
// except running both results at the same time
void xmmDOTDOT(double v1[], double v2[], double v3[], double r[2],
intptr_t size)
{
   double   temp[6];
   intptr_t alignedTemp = (((intptr_t)&temp[0]) & 8) >> 3;
   __m128d  _temp0 = _mm_set_pd(0.0, 0.0);
   __m128d  _temp1 = _mm_set_pd(0.0, 0.0);
   __m128d *_v1 = (__m128d *)v1;
   __m128d *_v2 = (__m128d *)v2;
   __m128d *_v3 = (__m128d *)v3;

   intptr_t halfSize = size / 2;

   for(intptr_t i = 0; i < halfSize; i++)
   {
      _temp0 = _mm_add_pd(_temp0, _mm_mul_pd(_v1[i], _v2[i]));
      _temp1 = _mm_add_pd(_temp1, _mm_mul_pd(_v1[i], _v3[i]));
   }
   _mm_store_pd( &temp[alignedTemp], _temp0);
   _mm_store_pd( &temp[alignedTemp+2], _temp1);
   if(size & 1)
   {
      temp[alignedTemp] += v1[size-1] * v2[size-1];
      temp[alignedTemp+2] += v1[size-1] * v3[size-1];
   }

   r[0] =  temp[alignedTemp] + temp[alignedTemp+1];
   r[1] =  temp[alignedTemp+2] + temp[alignedTemp+3];
}

bool UseXMM = false;

double doDOT(double v1[], double v2[], intptr_t size)
{
   if(UseXMM)
      return xmmDOT(v1, v2, size);
   return DOT(v1, v2, size);
}

void doDOTDOT(double v1[], double v2[], double v3[], double r[2],
intptr_t size)
{
   if(UseXMM)
      xmmDOTDOT(v1, v2, v3, r, size);
   else
      DOTDOT(v1, v2, v3, r, size);
}


// Computes the product of two square matrices.
void matrix_multiplyTranspose(double** m1, double** m2, double**
result, intptr_t size)
{
   // create a matrix to hold the transposition of m2
   // N.B.
   // When m2 will be used multiple times it would be more efficient
      // to move the transposition outside the multiplication function
      // and perform the transposition once.
   // Also, when matrix_multiplyTranspose is called multiple times
   // the empty array m2t could be saved across calls, thus saving
   // reallocation. However, this would also prohibit recursive
      // calls of this function.
   double** m2t = create_matrix(size);

   for (intptr_t i = 0; i < size; i++)
   {
     for (intptr_t j = 0; j < size; j++)
     {
         m2t[j][i] = m2[i][j];
     }
   }

   for (intptr_t i = 0; i < size; i++)
   {
     for (intptr_t j = 0; j < size; j++)
     {
       result[i][j] = doDOT(m1[i], m2t[j], size);
     }
   }
   destroy_matrix(m2t, size);
}

#if defined(USE_matrix_compare)
void matrix_compare(double** m1, double** m2,intptr_t size)
{
   for (intptr_t i = 0; i < size; i++)
   {
      for (intptr_t j = 0; j < size; j++)
      {
        double delta = abs(m1[i][j] - m2[i][j]);
        double epsilon = max(abs(m1[i][j]), abs(m2[i][j]))
                                    / pow(10.0,12);
        if(delta > epsilon)
        {
           wcout << L"m1[" << i << L"][" << j << L"] != m2["
                        << i << L"][" << j << L"]" << endl;
        }
      }
   }
}
#endif

// Computes the product of two square matrices in parallel.
void parallel_matrix_multiply(
      double** m1, double** m2, double** result, intptr_t size)
{
   parallel_for(
         intptr_t(0), size,
         [&](intptr_t iBegin, intptr_t iEnd)
         {
            for(intptr_t i = iBegin; i < iEnd; ++i)
            {
              for (intptr_t j = 0; j < size; j++)
              {
                double temp = 0;
                for (intptr_t k = 0; k < size; k++)
                {
                  temp += m1[i][k] * m2[k][j];
                }
                result[i][j] = temp;
              }
            }
         });
}

// Computes the product of two square matrices in parallel.
// using transposition of matrix prior to multiplication
void parallel_matrix_multiplyTranspose(double** m1, double** m2,
double** result, intptr_t size)
{
   // create a matrix to hold the transposition of m2
   // N.B.
   // When m2 will be used multiple times
   // it would be more efficient to move the transposition
   // outside the multiplication function and perform the
   // transposition once.
   // Also, when matrix_multiplyTranspose is called multiple times
   // the empty array m2t could be saved across calls, thus saving
   // reallocation. However, this would also prohibit recursive
   // calls of this function.
   double** m2t = create_matrix(size);
   // QuickThread
   parallel_for(
         AllThreads$,
         intptr_t(0), size,
         [&](intptr_t iBegin, intptr_t iEnd)
         {
            for(intptr_t i = iBegin; i < iEnd; ++i)
            {
              for (intptr_t j = 0; j < size; j++)
              {
                m2t[j][i] = m2[i][j];
              }
            }
         });
   parallel_for(
      AllThreads$,
      intptr_t(0), size,
      [&](intptr_t iBegin, intptr_t iEnd)
      {
         for(intptr_t i = iBegin; i < iEnd; ++i)
         {
           for (intptr_t j = 0; j < size; j++)
           {
             result[i][j] = doDOT(m1[i], m2t[j], size);
           }
         }
      });
   destroy_matrix(m2t, size);
}


struct TagTeamsSystem
{
   // ctor captured args
   double** m1;
   double** m2;
   double** result;
   intptr_t size;

   volatile bool  TreadWaitingForTransposition;
   // allocated transposition array for m2
   double** m2t;

   // create array indicateing number of cells in column pair
   // transposed
   // 0 = no data transposed, size = all data transposed
   // Transposition performed 2 columns to 2 rows at a time
   // (potential for last transposition being 1 column to one row)
   // m2t_transposed holds transpostion counts for the pair of rows
   volatile intptr_t* m2t_transposed;

   // *** m2t_transposed currently has junk

   bool  HaveAllocationError;

   bool  allocationError() { return HaveAllocationError; }
   TagTeamsSystem(
          double** _m1, double** _m2, double** _result, intptr_t _size)
   {
      m1 = _m1;
      m2 = _m2;
      result = _result;
      size = _size;
      TreadWaitingForTransposition = false;
      m2t = new double*[size];
      m2t_transposed = new intptr_t[(size + 1) / 2];
      if(!m2t || !m2t_transposed)
      {
         HaveAllocationError = true;
         return;
      }
      // *** m2t currently has junk for row pointers
      // *** m2t_transposed currently has junk for counters

      // tentatively indicate valid allocation
      // other threads can refute this assertion
      HaveAllocationError = false;
   }
   ~TagTeamsSystem()
   {
      if(m2t_transposed)
         delete [] m2t_transposed;
      if(m2t)
         delete [] m2t;
   }
};

struct TagTeamsProcessor
{
   TagTeamsSystem* tts;
   intptr_t iBeginL3;
   intptr_t iEndL3;
   bool  HaveAllocationError;
   double** m1;
   double** m2;
   double** m2t;
   double** result;
   intptr_t size;
   volatile intptr_t* m2t_transposed;

   bool  allocationError() { return HaveAllocationError; }
   TagTeamsProcessor(
      TagTeamsSystem* _tts, intptr_t _iBeginL3, intptr_t _iEndL3)
   {
      tts = _tts;
      iBeginL3 = _iBeginL3;
      iEndL3 = _iEndL3;
      HaveAllocationError = false;
      // make local copies of often used variables
      m1 = tts->m1;
      m2 = tts->m2;
      m2t = tts->m2t;
      result = tts->result;
      size = tts->size;
      m2t_transposed = tts->m2t_transposed;

      // first null out the entire slice of our rows in m2t
      // (in the event other thread has allocation error)
      for(intptr_t iRow = iBeginL3; iRow < iEndL3; ++iRow)
         m2t[iRow] = NULL;
      // clear counts held in m2t_transposed
      for(intptr_t i = iBeginL3; i<iEndL3; i += 2)
         m2t_transposed[i / 2] = 0;
      // now allocate
      for(intptr_t iRow = iBeginL3; iRow < iEndL3; ++iRow)
      {
         // NUMA aware allocation
         m2t[iRow] = cacheAligned_doubles.allocate(size);
         if(m2t[iRow] == NULL)
         {
            // indicate allocation error
            HaveAllocationError = true;
            return;  // let dtor clean up
         }
      } // for(intptr_t iRow = iBeginL3; iRow < iEndL3; ++iRow)
   }
   ~TagTeamsProcessor()
   {
      // deallocate our rows of m2t
      for(intptr_t iRow = iBeginL3; iRow < iEndL3; ++iRow)
      {
         if(m2t[iRow] == NULL)
            break;

         // NUMA aware deallocation
         cacheAligned_doubles.deallocate( m2t[iRow], size);
      }
   }

};

struct TagTeamSameCache
{
   TagTeamsProcessor*   ttp;
   intptr_t iBeginL2;
   intptr_t iEndL2;
   intptr_t iBeginL3;
   intptr_t iEndL3;
   bool  HaveAllocationError;

   // Row and Col of a 2 x 2 window
   // *** may exceed bounds of array ***
   volatile intptr_t iRow2x2;
   volatile intptr_t iCol2x2;
   volatile intptr_t MasterTileSequenceNumber;
   volatile intptr_t SlaveTileSequenceNumber;
   intptr_t nTeamMembersInL2;
   double** m1;
   double** m2;
   double** m2t;
   double** result;
   intptr_t size;
   volatile intptr_t* m2t_transposed;


   // DoDiagonalsState table
   // 0 = no processing
   // 1 = transposition started or ended for our diagonal
   // 2 = column process started or ended for our diagonal (L2)
   // 3 = column process started or ended for our processor (L3)
   // 4 = column process started or ended for any processor
   char*    DoDiagonalsState; // = new char[(size + 1) / 2];
   intptr_t iDoDiagonalL2index;  // iBeginL2 : iEndL2
                        // (use DoDiagonals[iDoDiagonalL2index/2])

   intptr_t iDoDiagonalL3index;  // iBeginL3 : iEndL3
                        // (use DoDiagonals[iDoDiagonalL3index/2])

   intptr_t iDoDiagonalSystem;   // 0 : size
                        // (use DoDiagonals[iDoDiagonalSystem/2])

   intptr_t iDoColumnIndex;   // iBeginL2 : iEndL2
   intptr_t iDoColumnColumn;  // iBeginL2 : iEndL2
   intptr_t iLastDiagonalDone;
   inline bool IsOnDiagonal()
   {
      return (iRow2x2 == iCol2x2);
   }
   inline bool Is2x2()
   {
      return((iRow2x2 + 2 <= size) && (iCol2x2 + 2 <= size));
   }
   inline bool Is2x1()
   {
      return((iRow2x2 + 2 <= size) && (iCol2x2 + 1 == size));
   }
   inline bool Is1x2()
   {
      return((iRow2x2 + 1 == size) && (iCol2x2 + 2 <= size));
   }
   inline bool Is1x1()
   {
      return((iRow2x2 + 1 == size) && (iCol2x2 + 1 == size));
   }

   bool  allocationError() { return HaveAllocationError; }
   TagTeamSameCache(
            TagTeamsProcessor* _ttp,
            intptr_t _iBeginL2, intptr_t _iEndL2)
   {
      ttp = _ttp;
      iBeginL2 = _iBeginL2;
      iEndL2 = _iEndL2;
      iBeginL3 = ttp->iBeginL3;
      iEndL3 = ttp->iEndL3;

      HaveAllocationError = false;
      m1 = ttp->m1;
      m2 = ttp->m2;
      m2t = ttp->m2t;
      result = ttp->result;
      size = ttp->size;
      m2t_transposed = ttp->m2t_transposed;
      MasterTileSequenceNumber = -1;
      SlaveTileSequenceNumber = -1;
      DoDiagonalsState = new char[(size + 1) / 2];
      memset(DoDiagonalsState, 0, (size + 1) / 2);
```

```
      iLastDiagonalDone = size;  // initialize out of bounds
   }
   ~TagTeamSameCache()
   {
      if(DoDiagonalsState)
         delete [] DoDiagonalsState;
   }
   inline bool TileSeenBySlave() { return (
         (MasterTileSequenceNumber == SlaveTileSequenceNumber)
         || ((iEndL2 - iBeginL2) == size)
         || (nTeamMembersInL2 == 1)); }
   inline bool TileAdvancedByMaster() { return (
         (MasterTileSequenceNumber > SlaveTileSequenceNumber)
         || (nTeamMembersInL2 == 1)); }


   inline bool PickNextDiagonal()
   {
      // any remaining tiles on our diagonal
      if(iDoDiagonalL2index >= iEndL2)
         return false;     // no

      // set tile focus to next tile on diagonal
      iRow2x2 = iDoDiagonalL2index;
      iCol2x2 = iRow2x2;

      // Notify Slave thread that tile selection is ready
      ++MasterTileSequenceNumber;

      // record last diagonal done (likely in L1/L2 cache)
      iLastDiagonalDone = iRow2x2;

      // indicate begun transposition
      DoDiagonalsState[iLastDiagonalDone / 2] = 1;

      // and bump iDoDiagonalL2index of 2x2 tile for next time
      iDoDiagonalL2index += 2;

      // indicate valid tile chosen
      return true;
   }

   inline bool PickingFirstTile()
   {
      // see if first time call
      if(MasterTileSequenceNumber >= 0)
         return false;     // no

      // yes, initialize state machine
      iDoDiagonalL2index = iBeginL2;   // iBeginL2 : iEndL2
                        // (use DoDiagonals[iDoDiagonalL2index/2])

      iDoDiagonalL3index = iBeginL3;   // iBeginL3 : iEndL3
                        // (use DoDiagonals[iDoDiagonalL3index/2])
      iDoDiagonalSystem = 0;        // 0 : size
                        // (use DoDiagonals[iDoDiagonalSystem/2])
      iDoColumnIndex = iEndL2;      // iBeginL2 : iEndL2
      iDoColumnColumn = iEndL2;     // iBeginL2 : iEndL2
      // return next (first) tile on our diagonal
      return   PickNextDiagonal();
   }

   // while we are traversing down a column of the last
   // diagonal picked by our thread, and near the end
   // of completing the matrix multiplication, some other
   // thread may have finished up all its diagonals, and
   // columns on diagonal, and has found no additional
   // work to do. When this occures .AND. when this thread
   // has additional diagonals to perform, then we want to
   // process or next diagonal prior to following down the
   // column of the last tile we picked.
   inline bool PickingForDiagonalTileForStarvedThread()
   {
      if(ttp->tts->TreadWaitingForTransposition)
      {
         // attempt to pick our next diagonal
         if(PickNextDiagonal())
         {
            // got tile, clear starvation flag
            ttp->tts->TreadWaitingForTransposition = false;
            return true;
         }
      } // if(ttp->tts->TreadWaitingForTransposition)
      return false;
   }

   inline bool PickingTileDownColumnOfOurDiagonal()
   {
      // skip over our diagonal if/when reached
      if(iDoColumnIndex == iDoColumnColumn)
         iDoColumnIndex += 2;

      // are we done with this column?
      if(iDoColumnIndex >= iEndL2)
         return false;  // yes, indicate pick fail

      // the row becomes the iDoColumnIndex
      iRow2x2 = iDoColumnIndex;
      iCol2x2 = iDoColumnColumn;
      ++MasterTileSequenceNumber;   // Notify Slave thread
      // and bump iDoColumnIndex for next pick
      iDoColumnIndex += 2;
      return true;
   }

   inline bool PickingFirstTileDownColumnOfOurLastDiagonal()
   {
      // assure not at end of matrix
      if(iLastDiagonalDone >= size)
         return false;

      if(DoDiagonalsState[iLastDiagonalDone / 2] == 1)
         DoDiagonalsState[iLastDiagonalDone / 2] = 2;
      iDoColumnColumn = iLastDiagonalDone;
      iDoColumnIndex = iBeginL2;
      iLastDiagonalDone = size;  // invalidate for test above
      // the following may fail on very small matrix
      // do not assume it will always succeed
      // now walk down this column
      return PickingTileDownColumnOfOurDiagonal();
   }

   inline bool PickingFirstTileOfColumnOfOurL3()
   {
      for(  iLastDiagonalDone = iDoDiagonalL3index;
            iLastDiagonalDone < iEndL3;
            iLastDiagonalDone += 2)
      {
         // convert to 2x2 index number
         intptr_t i = iLastDiagonalDone / 2;
         // see if we have not already processed this column
         // we have not performed the diagonal transposition
         // for tiles of other cores of this processor.
         // Therefore:
         // DoDiagonalsState[i] == 0 for unprocessed tiles
         //                            (by our thread)
         // DoDiagonalsState[i] == 1 we did diagonal
         //                   (but not entire column of that diagonal)
         // DoDiagonalsState[i] == 2 we did diagonal
         //                      (and entire column of that diagonal)
         // DoDiagonalsState[i] == 3 other thread in our L3
         // did diagonal and we did column of that diagonal
         // find ((DoDiagonalsState[i] == 0)
         //    && (m2t_transposed[i] == size))
         if(DoDiagonalsState[i] == 0)
         {
            // we haven't done this column pair
            // see if other thread has completed transposion
            if(m2t_transposed[i] == size)
            {
               // transposition complete
               // indicate we selected this column
               DoDiagonalsState[i] = 3;
               // see if we can advance index for next time
               if(iLastDiagonalDone == iDoDiagonalL3index)
                  iDoDiagonalL3index += 2;
                  if(PickingFirstTileDownColumnOfOurLastDiagonal())
                     return true;
            } // if(m2t_transposed[i] == size)
         } // if(DoDiagonals[i])
         else
         {
            if(iLastDiagonalDone == iDoDiagonalL3index)
               iDoDiagonalL3index += 2;
         }
      } // for(iLastDiagonalDone = iDoDiagonalL3index;
      // iLastDiagonalDone < iEndL3;
      // iLastDiagonalDone += 2)
      return false;
   }
```

```
bool PickingFirstTileOfColumnOfAnyProcessor()
{
   for(  iLastDiagonalDone = iDoDiagonalSystem;
         iLastDiagonalDone < size;
         iLastDiagonalDone += 2)
   {
      // convert to 2x2 index number
      intptr_t i = iLastDiagonalDone / 2;
      // see if we have not already processed this column
      if(DoDiagonalsState[i] <= 1)
      {
         // we haven't done this column pair
         // see if other thread has completed
         // transposion
         if(m2t_transposed[i] == size)
         {
            // transposition complete
            // indicate we selected this diagonal
            DoDiagonalsState[i] = 4;
            // see if we can advance index for next time
            if(iLastDiagonalDone == iDoDiagonalSystem)
               iDoDiagonalSystem += 2;
            if(PickingFirstTileDownColumnOfOurLastDiagonal())
               return true;
         } // if(m2t_transposed[i] == size)
      } // if(DoDiagonalsState[i])
      else
      {
         if(iLastDiagonalDone == iDoDiagonalSystem)
            iDoDiagonalSystem += 2;
      }
   } // for(   iLastDiagonalDone = iDoDiagonalSystem;
   //        iLastDiagonalDone < size;
   //        iLastDiagonalDone += 2)
   ttp->tts->TreadWaitingForTransposition = true;
   return false;
}

inline bool MoreTilesToPick()
{
   return ((iDoDiagonalSystem < size)
            || (iDoDiagonalL3index < iEndL3)
            || (iDoDiagonalL2index < iEndL2));
}

// select a tile, return true if found, false when done
// only master thread of team calls this function
// Also, this function selects the next tile along the diagonal
bool SelectTile()
{
   // wait until slave thread has observed prior
   // tile selection (passes on 1st time call)
   while(!TileSeenBySlave())
      _mm_pause();

   // quick test for first time call
```

```
      if(PickingFirstTile())
      {
         return true;
      }

      // test for thread starvation
      if(PickingForDiagonalTileForStarvedThread())
      {
         return true;
      }

      if(PickingTileDownColumnOfOurDiagonal())
      {
         return true;
      }

      if(PickingFirstTileDownColumnOfOurLastDiagonal())
      {
         return true;
      }

      if(PickingFirstTileOfColumnOfOurL3())
      {
         return true;
      }

      if(PickNextDiagonal())
      {
         return true;
      }

      if(PickingFirstTileOfColumnOfAnyProcessor())
      {
         return true;
      }

      while(MoreTilesToPick())
      {
         if(PickingFirstTileOfColumnOfOurL3())
      {
         return true;
      }
         if(PickingFirstTileOfColumnOfAnyProcessor())
      {
         return true;
      }
         _mm_pause();   // or qtYield()
      }
      // set focus out of range such that
      // Slave thread knows of termination condition
      iRow2x2 = size;
      iCol2x2 = size;
      // notify slave of termination condition
      ++MasterTileSequenceNumber;
      return false;
   } // bool SelectTile()
};
```

```
struct TagTeamMember
{
   TagTeamSameCache* ttc;
   intptr_t iTeamMemberInL2;
   intptr_t nTeamMembersInL2;
   bool  HaveAllocationError;
   double** m1;
   double** m2;
   double** m2t;
   double** result;
   intptr_t size;
   volatile intptr_t* m2t_transposed;

   intptr_t iRow2x2;
   intptr_t iCol2x2;

   size_t   sizeMinus1; // = size_local - 1;
   size_t   sizeTrunc;  // = size_local & ~1;
                        // (even number of size)
   // Redefine pointers as int64_t
   int64_t** m2_as_int64;  // = (int64_t**)m2;
   int64_t** m2t_as_int64; // = (int64_t**)m2t;

   bool  allocationError() { return HaveAllocationError; }
   TagTeamMember( TagTeamSameCache* _ttc,
                  intptr_t _iTeamMemberInL2,
                  intptr_t _nTeamMembersInL2)
   {
      ttc = _ttc;
      iTeamMemberInL2 = _iTeamMemberInL2;
      nTeamMembersInL2 = _nTeamMembersInL2;
      ttc->nTeamMembersInL2 = nTeamMembersInL2;
      HaveAllocationError = false;
      m1 = ttc->m1;
      m2 = ttc->m2;
      m2t = ttc->m2t;
      result = ttc->result;
      size = ttc->size;
      sizeMinus1 = size - 1;  // often used evaluation
      sizeTrunc = size & ~1;  // even number of size
      m2t_transposed = ttc->m2t_transposed;
      // Redefine pointers as int64_t
      m2_as_int64 = (int64_t**)m2;
      m2t_as_int64 = (int64_t**)m2t;
   }
   ~TagTeamMember()
   {
      if((size == 1) && (iTeamMemberInL2 > 0))
      {
         // advance SlaveTileSequenceNumber
         ++ttc->SlaveTileSequenceNumber;
      }
   }

   bool SelectTile()
   {
      if(IsMaster())
      {
         ttc->SelectTile();
         // copy to local values (may be invalid)
         iRow2x2 = ttc->iRow2x2;
         iCol2x2 = ttc->iCol2x2;
         // return true if we have valid tile
         return (iRow2x2 < size);
      }

      // Slave thread
      // loop until master advances tile
      while(!ttc->TileAdvancedByMaster())
         _mm_pause();

      // ** save prior to advancing SlaveTileSequenceNumber
      iRow2x2 = ttc->iRow2x2;
      iCol2x2 = ttc->iCol2x2;
      // advance SlaveTileSequenceNumber
      ++ttc->SlaveTileSequenceNumber;
      // return true if not void (have work to do)
      return (iRow2x2 < size);
   } // bool SelectTile()

   inline bool IsMaster() { return (iTeamMemberInL2 == 0); }
   inline bool IsSlave() { return (iTeamMemberInL2 != 0); }
   inline bool IsOnlyThread() { return (nTeamMembersInL2 == 1); }
   inline bool IsOnDiagonal()
   {
      return (iRow2x2 == iCol2x2);
   }
   inline bool Is2x2()
   {
      return((iRow2x2 + 2 <= size) && (iCol2x2 + 2 <= size));
   }
   inline bool Is2x1()
   {
      return((iRow2x2 + 2 <= size) && (iCol2x2 + 1 == size));
   }
   inline bool Is1x2()
   {
      return((iRow2x2 + 1 == size) && (iCol2x2 + 2 <= size));
   }
   inline bool Is1x1()
   {
      return((iRow2x2 + 1 == size) && (iCol2x2 + 1 == size));
   }

   void DoDiagonalAsOnlyThread()
   {
      // must be on processor with diminished capacity
      // only 1 thread in this tag team
      // Equivilent to DoDiagonalAsMasterThread() without
      // notification
      // And performing double DOT on what would have been done
      // by Slave
      intptr_t iRow = iRow2x2;   // same coord as 2x2
```

```
                                    // (when master thread)
intptr_t iCol = iCol2x2;   // same coord as 2x2
                                    // (when master thread)
if(iCol == sizeMinus1)
{
   // last column is single column
   for(intptr_t i=0; i < sizeTrunc; i += 2)
   {
      int64_t  temp[2]; // compiler should SSE register temp
      // collect 2x2 tile from next row pair
      // in tile column pair
      temp[0] =  m2_as_int64[i][iCol];
      temp[1] =  m2_as_int64[i+1][iCol];
      // store the transposed 1x2 tile into the m2t array
      m2t_as_int64[iCol][i] = temp[0];
      m2t_as_int64[iCol][i+1] = temp[1];
   }
   // check for odd size
   if(size & 1)
   {
      m2t_as_int64[iCol][sizeMinus1]
         = m2_as_int64[sizeMinus1][iCol];
   }
}
else
{
   // not last column is single column

   // N.B. keep sizeTrunc as local variable instead of
   // outer scope reference
   // the following for loop will run faster
   // walk down the column pair of m2 transposing to
   // row pair in m2t
   for(intptr_t i=0; i < sizeTrunc; i += 2)
   {
      int64_t  temp[4]; // compiler should SSE register temp
      // collect 2x2 tile from next row pair
      // in tile column pair
      temp[0] =  m2_as_int64[i][iCol];
      temp[2] =  m2_as_int64[i][iCol+1];
      temp[1] =  m2_as_int64[i+1][iCol];
      temp[3] =  m2_as_int64[i+1][iCol+1];
      // store the transposed 2x2 tile into the m2t array
      m2t_as_int64[iCol][i] = temp[0];
      m2t_as_int64[iCol][i+1] = temp[1];
      m2t_as_int64[iCol+1][i] = temp[2];
      m2t_as_int64[iCol+1][i+1] = temp[3];
   }
   // check for odd size
   if(size & 1)
   {
      m2t_as_int64[iCol][sizeMinus1]
                        = m2_as_int64[sizeMinus1][iCol];
      m2t_as_int64[iCol+1][sizeMinus1]
                        = m2_as_int64[sizeMinus1][iCol+1];
   }
}
```

```
      // indicate to other threads that we completed transposition
      m2t_transposed[iCol / 2] = size;
      if(Is2x2())
      {
         // master thread finishes up with double DOT
         // for upper row of 2x2
         doDOTDOT(   m1[iRow],
                     m2t[iCol],
                     m2t[iCol+1],
                     &result[iRow][iCol], size);
         // then double DOT for lower row of 2x2
         doDOTDOT(   m1[iRow+1],
                     m2t[iCol],
                     m2t[iCol+1],
                     &result[iRow+1][iCol], size);
      }
      else
      {
         // 1x1
         result[iRow][iCol] = doDOT(m1[iRow], m2t[iCol], size);
      }
   } // void DoDiagonalAsOnlyThread()

   void DoDiagonalAsMasterThread()
   {
      // Master thread
      intptr_t iRow = iRow2x2;   // same coord as 2x2
                                 // (when master thread)
      intptr_t iCol = iCol2x2;   // same coord as 2x2
                                 // (when master thread)

      // master of tag team performs transposition
      // and DOT of its row in the tag team tile
      // Notify the other thread(s) of of my team
      // after the cache line(s) is(are) flushed
      intptr_t notificationInterval
                         = CacheFlushLineSize / sizeof(double);
      if(iCol == sizeMinus1)
      {
         // last column is single column
         for(intptr_t i=0; i < sizeTrunc; i += 2)
         {
            int64_t  temp[2]; // compiler should SSE register temp
            // collect 2x2 tile from next row pair in tile column pair
            temp[0] =  m2_as_int64[i][iCol];
            temp[1] =  m2_as_int64[i+1][iCol];
            // store the transposed 1x2 tile into the m2t array
            m2t_as_int64[iCol][i] = temp[0];
            m2t_as_int64[iCol][i+1] = temp[1];
            // see if we wrote the last pair in cache flush line
            if(((i+2)%notificationInterval) == 0)
            {
               // inform other thread(s) of this team that
               // transposition has occured up through and including
               // this column
               m2t_transposed[iCol / 2] = i+2;
            }
         }
         // check for odd size
         if(size & 1)
         {
            m2t_as_int64[iCol][sizeMinus1]
                              = m2_as_int64[sizeMinus1][iCol];
            // TileColTransposed updated below
         }
      }
      else
      {
         // not last column is single column

         // N.B. keep sizeTrunc as local variable instead of
         // outer scope reference, the following for loop will run
         // faster
         // Walk down the column pair of m2 transposing to
         // row pair in m2t
         for(intptr_t i=0; i < sizeTrunc; i += 2)
         {
            int64_t  temp[4]; // compiler should SSE register temp
            // collect 2x2 tile from next row pair in tile column pair
            temp[0] =  m2_as_int64[i][iCol];
            temp[2] =  m2_as_int64[i][iCol+1];
            temp[1] =  m2_as_int64[i+1][iCol];
            temp[3] =  m2_as_int64[i+1][iCol+1];
            // store the transposed 2x2 tile into the m2t array
            m2t_as_int64[iCol][i] = temp[0];
            m2t_as_int64[iCol][i+1] = temp[1];
            m2t_as_int64[iCol+1][i] = temp[2];
            m2t_as_int64[iCol+1][i+1] = temp[3];
            // see if we wrote the last pair in cache flush line
            if(((i+2)%notificationInterval) == 0)
            {
               // inform other thread(s) of this team that
               // transposition has occured up through and including
               // this column
               m2t_transposed[iCol / 2] = i+2;
            }
         }
         // check for odd size
         if(size & 1)
         {
            m2t_as_int64[iCol][sizeMinus1]
                              = m2_as_int64[sizeMinus1][iCol];
            m2t_as_int64[iCol+1][sizeMinus1]
                              = m2_as_int64[sizeMinus1][iCol+1];
         }
      }
      // Inform other teams that transposition of this column
      // is complete
      m2t_transposed[iCol / 2] = size;
      if(Is2x2())
      {
         // master thread finishes up with DOT on
         // upper left corner of 2x2 tile
         result[iRow][iCol]
```

```
                        = doDOT(m1[iRow], m2t[iCol], size);
      }
      else
      {
         // Slave thread did this for us
      }
   } // void DoDiagonalAsMasterThread()

   void DoDiagonalAsSlave()
   {

      intptr_t iRow = iRow2x2;   // same coord as 2x2
                                 // (when master thread)
      intptr_t iCol = iCol2x2;   // same coord as 2x2
                                 // (when master thread)

      if(Is1x1())
      {
         // special case for 1x1 tile
         // slave produces the one and DOT
         double temp = 0.0;
         double* v1 = m1[iRow];
         double* v2 = m2t[iRow];

         for(intptr_t i = 0; i < size; i++)
         {
            // The slave thread(s) tail along after the master thread.
            // Wait until master completes transposition of cache line.
            while(i >= m2t_transposed[iRow / 2])
            {
               _mm_pause();
#if defined(USE_mm_pause_count)
               // when gathering tuning statistics
               ++_mm_pause_count_2;
#endif
            }
            temp += v1[i] * v2[i];
         }
         result[iRow][iRow] = temp;

         return;
      } // if(Is1x1())
      // 2x2 tile
      // produce a double DOT trailing along the master thread
      // performing transposition
      double temp0 = 0.0;
      double temp1 = 0.0;
      double* v1 = m1[iRow+1];
      double* v2 = m2t[iCol];
      double* v3 = m2t[iCol+1];
      // see how the compiler optimizaton vectorizes the following
      for(intptr_t i=0; i < size; ++i)
      {
         // The slave thread(s) tail along after the master thread.
         // Wait until master completes transposition of cache line.
         while(i >= m2t_transposed[iCol / 2])
         {
            _mm_pause();
#if defined(USE_mm_pause_count)
            // when gathering tuning statistics
            ++_mm_pause_count_3;
#endif
         }
         temp0 += v1[i] * v2[i];
         temp1 += v1[i] * v3[i];
      } // for(intptr_t i=0; i < sizeTrunc; ++i)

      result[iRow+1][iCol] = temp0;
      result[iRow+1][iCol+1] = temp1;

      v1 = m1[iRow];
      // perform DOT on (0,1) of 2x2 cell
      result[iRow][iCol+1] = doDOT(v1, v3, size);
   } // void DoDiagonalAsSlave()

   void DoDiagonal()
   {
      if(IsMaster())
      {
         if(IsOnlyThread())
            DoDiagonalAsOnlyThread();
         else
            DoDiagonalAsMasterThread();
      }
      else
      {
         DoDiagonalAsSlave();
      }
   } // void DoDiagonal()

   void DoOffDiagonal()
   {
      intptr_t iRow = iRow2x2;
      intptr_t iCol = iCol2x2;
      if(Is2x2())
      {
         // get our threads row of 1x2 part of 2x2
         iRow += iTeamMemberInL2;
         // perform double DOT
         doDOTDOT(   m1[iRow],
                     m2t[iCol],
                     m2t[iCol+1],
                     &result[iRow][iCol], size);
         if(IsOnlyThread())
         {
            iRow += 1;
            // perform double DOT
            doDOTDOT(   m1[iRow],
                        m2t[iCol],
                        m2t[iCol+1],
                        &result[iRow][iCol], size);
         }
         return;
      }
      // not 2x2
      if(IsOnlyThread())
      {
         if(Is2x1())
         {
            result[iRow][iCol] = doDOT(m1[iRow], m2t[iCol], size);
            iRow += 1;
            result[iRow][iCol] = doDOT(m1[iRow], m2t[iCol], size);
            return;
         }
         if(Is1x2())
         {
            doDOTDOT(   m1[iRow],
                        m2t[iCol],
                        m2t[iCol+1],
                        &result[iRow][iCol], size);
            return;
         }
         if(Is1x1())
         {
            result[iRow][iCol] = doDOT( m1[iRow], m2t[iCol], size);
            return;
         }
         return;  // ??
      }
      // either Master or Slave
      if(Is2x1())
      {
         iRow += iTeamMemberInL2;
      }
      else
      if(Is1x2())
      {
         iCol += iTeamMemberInL2;
      }
      else
      if(Is1x1())
      {
         if(IsSlave())
            return;
      }
      // single DOT
      result[iRow][iCol] = doDOT(m1[iRow], m2t[iCol], size);
   }


   void DoWork()
   {
      while(SelectTile())
      {
         if(IsOnDiagonal())
            DoDiagonal();
         else
            DoOffDiagonal();
      }
   }
};
```

```
void parallel_matrix_multiplyTransposeTagTeam(double** m1, double** m2,
double** result, intptr_t size)
{
    // create whole system tag team object for this
    // matrix multiplication
    TagTeamsSystem System(m1, m2, result, size);
    if(System.allocationError())
    {
      cout << "Allocation error for System" << endl;
      return;
    }

    // divide the iteration space accross each multi-core processor
    // L3$ specifies by L3 cache
    //    .OR.
    // within processor for processors without L3 cache
    parallel_for(
      OneEach_L3$,
      intptr_t(0), size,
      [&](intptr_t iBeginL3, intptr_t iEndL3)
      {
         // adjust partitioning such that partitions
         // fall on 2x2 cell boundaries
         iBeginL3 += (iBeginL3 & 1);
         if(iBeginL3 >= size)
            return;
         if(iEndL3 < size)
            iEndL3 += (iEndL3 & 1);

         TagTeamsProcessor Processor(
                        &System, iBeginL3, iEndL3);
         if(Processor.allocationError())
         {
            cout << "Allocation error for Processor" << endl;
            return;
         }

         if(nThreadsPerL2 > 0)
         {
            // divide our L3 iteration space by L2 within this L3
            parallel_for(
               OneEach_Within_L3$ + L2$,
               iBeginL3, iEndL3,
               [&](intptr_t iBeginL2, intptr_t iEndL2)
               {
                  // adjust partitioning such that
                  // partitions fall on 2x2 cell boundaries
                  iBeginL2 += (iBeginL2 & 1);
                  if(iBeginL2 >= size)
                     return;
                  if(iEndL2 < size)
                     iEndL2 += (iEndL2 & 1);
                  // Here we are running as the Master thread of a
                  // 2 team member team (or 1 in the event of older
                  // processor)
                  //
                  TagTeamSameCache  SameCache(  &Processor,
                                                iBeginL2,
                                                iEndL2);
                  if(SameCache.allocationError())
                  {
                     cout << "Allocation error for SameCache" << endl;
                     return;
                  }
                  // Now bring in our other team member(s)
                  parallel_distribute(
                     L2$,
                     [&](  intptr_t iTeamMemberInL2,
                           intptr_t nTeamMembersInL2)
                     {
                        TagTeamMember  TeamMember(
                                          &SameCache,
                                          iTeamMemberInL2,
                                          nTeamMembersInL2);
                        if(TeamMember.allocationError())
                        {
                           cout << "Allocation error for TeamMember"
                                 << endl;
                           return;
                        }
                        TeamMember.DoWork();
                     } // [&](   intptr_t iTeamMemberInL2,
                       //        intptr_t nTeamMembersInL2)
                  ); // parallel_distribute( L2$,
               } // [&](intptr_t iBeginL2, intptr_t iEndL2)
            ); // parallel_for(
               //    OneEach_Within_L3$ + L2$, iBeginL3, iEndL3,
         }
         else
         {
            // older style processor (e.g. AMD Opteron 270)
            // two cores/socket, no sharing of L2
            // divide our L3 iteration space by L2 within this L3
            intptr_t iBeginL2 = iBeginL3;
            intptr_t iEndL2 = iEndL3;
            // adjust partitioning such that partitions
            // fall on 2x2 cell boundaries
            iBeginL2 += (iBeginL2 & 1);
            if(iBeginL2 >= size)
               return;
            if(iEndL2 < size)
               iEndL2 += (iEndL2 & 1);
            // Here we are running as the Master thread of a 2 team
            // member team (or 1 in the event of older processor)
            //
            TagTeamSameCache  SameCache(&Processor, iBeginL2, iEndL2);
            if(SameCache.allocationError())
            {
               cout << "Allocation error for SameCache" << endl;
               return;
            }
            // Now bring in our other team member(s)
            // Note, we use L3$ distribution but reference as L2
```

```
            parallel_distribute(
               L3$,
               [&](intptr_t iTeamMemberInL2, intptr_t nTeamMembersInL2)
               {
                  TagTeamMember  TeamMember( &SameCache,
                                             iTeamMemberInL2,
                                             nTeamMembersInL2);
                  if(TeamMember.allocationError())
                  {
                     cout << "Allocation error for TeamMember" << endl;
                     return;
                  }
                  TeamMember.DoWork();
               } //[&](intptr_t iTeamMemberInL2,
                  //   intptr_t nTeamMembersInL2)
            ); // parallel_distribute( L2$,
         }
      } // [&](intptr_t iBeginL3, intptr_t iEndL3)
    );
}

#if defined(_WIN32)
long  get_PageSize()
{
   SYSTEM_INFO info;
   GetSystemInfo(&info);
   return (long)info.dwPageSize;
}
#endif
#if defined(__linux)
long  get_PageSize()
{
}
#endif


bool verbose = false;
intptr_t size;
uint64_t timeSerial = 0;
uint64_t timeSerialTranspose = 0;
uint64_t timeSerialTransposeXMM = 0;
uint64_t timeParallel = 0;
uint64_t timeParallelTranspose = 0;
uint64_t timeParallelTransposeXMM = 0;
uint64_t timeParallelTransposeTagTeam = 0;
uint64_t timeParallelTransposeTagTeamXMM = 0;

void report()
{
   cout << size
      << "," << timeSerial
      << "," << timeParallel
      << "," << timeSerialTranspose
      << "," << timeSerialTransposeXMM
      << "," << timeParallelTranspose
      << "," << timeParallelTransposeXMM
      << "," << timeParallelTransposeTagTeam
      << "," << timeParallelTransposeTagTeamXMM
      << endl;
}

void reportTopology()
{
   qtInit   qtInit(-1,0);
   qtControl   ctrl;
   int_Native nThreads = 0;
   nThreads = qt_get_num_threads();
   cout << "nThreads=" << nThreads << endl;
   cout << "nL3=" << ctrl.SelectAffinities(OneEach_L3$) << endl;
   cout << "nThreadsPerL3=" << ctrl.SelectAffinities(L3$) << endl;
   cout << "CacheSize_L3=" << CacheLevelSize(3) << endl;
   cout << "CacheLineSize_L3=" << CacheLevelLineSize(3) << endl;
   cout << "nL2=" << ctrl.SelectAffinities(OneEach_L2$) << endl;
   cout << "nThreadsPerL2=" << ctrl.SelectAffinities(L2$) << endl;
   cout << "CacheSize_L2=" << CacheLevelSize(2) << endl;
   cout << "CacheLineSize_L2=" << CacheLevelLineSize(2) << endl;
   cout << "nL1=" << ctrl.SelectAffinities(OneEach_L1$) << endl;
   cout << "nThreadsPerL1=" << ctrl.SelectAffinities(L1$) << endl;
   cout << "CacheSize_L1=" << CacheLevelSize(1) << endl;
   cout << "CacheLineSize_L1=" << CacheLevelLineSize(1) << endl;

   volatile intptr_t coutL3 = 0;
   parallel_distribute(OneEach_L3$,
         [&](intptr_t iTeamMemberL3, intptr_t nTeamMembersL3)
         {
            while(iTeamMemberL3 > coutL3)
               _mm_pause();
            volatile intptr_t coutInL3 = 0;
            parallel_distribute(L3$,
               [&](intptr_t iTeamMemberInL3, intptr_t nTeamMembersInL3)
               {
                  while(iTeamMemberInL3 > coutInL3)
                     _mm_pause();
                  if(iTeamMemberInL3 == 0)
                     cout << "L3(" << iTeamMemberL3 << ") = {";
                  cout << qt_get_thread_num();
                  if(iTeamMemberInL3 + 1 == nTeamMembersInL3)
                     cout << "} NUMA(" << get_NUMA_NodeNumber() << ")"
                           << endl;
                  else
                     cout << ",";
                  cout << flush;
                  ++coutInL3;
               });
               ++coutL3;
         });
   volatile intptr_t coutL2 = 0;
   parallel_distribute(OneEach_L2$,
         [&](intptr_t iTeamMemberL2, intptr_t nTeamMembersL2)
         {
            while(iTeamMemberL2 > coutL2)
               _mm_pause();
            volatile intptr_t coutInL2 = 0;
            parallel_distribute(L2$,
               [&](intptr_t iTeamMemberInL2, intptr_t nTeamMembersInL2)
               {
                  while(iTeamMemberInL2 > coutInL2)
                     _mm_pause();
                  if(iTeamMemberInL2 == 0)
                     cout << "L2(" << iTeamMemberL2 << ") = {";
                  cout << qt_get_thread_num();
                  if(iTeamMemberInL2 + 1 == nTeamMembersInL2)
                     cout << "}" << endl;
                  else
                     cout << ",";
                  cout << flush;
                  ++coutInL2;
               });
               ++coutL2;
         });

   if(ctrl.SelectAffinities(L1$) > 1)
   {
      volatile intptr_t coutL1 = 0;
      parallel_distribute(OneEach_L1$,
            [&](intptr_t iTeamMemberL1, intptr_t nTeamMembersL1)
            {
               while(iTeamMemberL1 > coutL1)
                  _mm_pause();
               volatile intptr_t coutInL1 = 0;
               parallel_distribute(L1$,
                  [&](intptr_t iTeamMemberInL1,
                      intptr_t nTeamMembersInL1)
                  {
                     while(iTeamMemberInL1 > coutInL1)
                        _mm_pause();
                     if(iTeamMemberInL1 == 0)
                        cout << "L1(" << iTeamMemberL1 << ") = {";
                     cout << qt_get_thread_num();
                     if(iTeamMemberInL1 + 1 == nTeamMembersInL1)
                        cout << "}" << endl;
                     else
                        cout << ",";
                     cout << flush;
                     ++coutInL1;
                  });
                  ++coutL1;
            });

   }
}

void reportHeader()
{
   cout << "N"
      << "," << "S"
      << "," << "P"
      << "," << "ST"
      << "," << "STx"
      << "," << "PT"
      << "," << "PTx"
      << "," << "PTT"
      << "," << "PTTx"
      << endl;
}


int main(int argc, char *argv[])
{
   if(argc == 1)
   {
      reportTopology();
      reportHeader();
      return 0;
   }
    intptr_t sizeBegin = 750; // ~10 seconds in Debug build on
                              // 2x Opteron 270 (4 cores)
   if(argc >= 2)
        sizeBegin = atoi(argv[1]);
   intptr_t sizeEnd = sizeBegin;
   if(argc >= 3)
        sizeEnd = atoi(argv[2]);
   intptr_t sizeIncrement = 2;
   if(argc >= 4)
        sizeIncrement = atoi(argv[3]);

   for(size = sizeBegin; size <= sizeEnd; size += sizeIncrement)
   {

   if(verbose)
   {
      wcout << L"matrix size = " << size << L" x " << size << L" ("
         << size * size << L")" << endl;
   }

   PageSize = get_PageSize();

   // Create a random number generator.
   mt19937 gen(42);

   // Create and initialize the input matrices and the matrix that
   // holds the result.
   m1 = initialize_matrix(create_matrix(size), size, gen);
   m2 = initialize_matrix(create_matrix(size), size, gen);
   resultSerial = create_matrix(size);
   resultSerialt = create_matrix(size);
   resultSerialtXMM = create_matrix(size);
   resultParallel = create_matrix(size);
   resultParallelt = create_matrix(size);
   resultParalleltXMM = create_matrix(size);
   resultParalleltt = create_matrix(size);
   resultParallelttXMM = create_matrix(size);
   bool doSerial = false;
   bool doSerialTranspose = false;
   bool doSerialTransposeXMM = false;
   bool doParallel = false;
   bool doParallelTranspose = false;
   bool doParallelTransposeXMM = false;
   bool doParallelTransposeTagTeam = true;

  if(doSerial)
   {
      timeSerial = time_call([&] {
         matrix_multiply(m1, m2, resultSerial, size);
        });

   }

   if(doSerialTranspose)
   {
   timeSerialTranspose = time_call([&] {
      matrix_multiplyTranspose(m1, m2, resultSerialt, size);
   });
#if defined(USE_matrix_compare)
   matrix_compare(resultSerial, resultSerialt, size);
#endif
   }

   if(doSerialTransposeXMM)
   {
   timeSerialTransposeXMM = time_call([&] {
      UseXMM = true;
      matrix_multiplyTranspose(m1, m2, resultSerialtXMM, size);
      UseXMM = false;
   });
#if defined(USE_matrix_compare)
   matrix_compare(resultSerial, resultSerialtXMM, size);
#endif
   }
   {
      // scope the qtInit object _after_ running serial test
      // as we do not wish for the establishment of the thread
      // pool to detract from the serial run time
      // initialize QuickThread thread pool
      // (-1 = all available hardware threads for compute class,
      //   0 = no I/O threads)
      qtInit   qtInit(-1,0);
      int_Native nThreads = 0;
      nThreads = qt_get_num_threads();
      {
         qtControl   ctrl;
         nThreadsPerL2 = ctrl.SelectAffinities(L2$);
      }

   CacheLineSize_L1 = CacheLevelLineSize(1);
   CacheFlushLineSize = CLFLUSHCacheLineSize;
   CacheSize_L1 = CacheLevelSize(1);
   CacheSize_Larger = CacheLevelSize(3);
   if(!CacheSize_Larger)
      CacheSize_Larger = CacheLevelSize(2);
   if(!CacheSize_Larger)
      CacheSize_Larger = CacheLevelSize(1);

   if(doParallel)
   {
```

```
   timeParallel = time_call([&] {
      parallel_matrix_multiply(m1, m2, resultParallel, size);
   });

#if defined(USE_matrix_compare)
   matrix_compare(resultSerial, resultParallel, size);
#endif
   }

   if(doParallelTranspose)
   {
   timeParallelTranspose = time_call([&] {
      parallel_matrix_multiplyTranspose(m1, m2, resultParallelt, size);
   });
#if defined(USE_matrix_compare)
   matrix_compare(resultSerial, resultParallelt, size);
#endif
   }

   if(doParallelTransposeXMM)
   {
   timeParallelTransposeXMM = time_call([&] {
      UseXMM = true;
      parallel_matrix_multiplyTranspose(m1, m2, resultParalleltXMM,
size);
      UseXMM = false;
   });
#if defined(USE_matrix_compare)
    matrix_compare(resultSerial, resultParalleltXMM, size);
#endif
   }

   if(doParallelTransposeTagTeam)
   {
   timeParallelTransposeTagTeam = time_call([&] {
      parallel_matrix_multiplyTransposeTagTeam(
         m1, m2, resultParalleltt, size);
   });
#if defined(USE_matrix_compare)
   matrix_compare(resultSerial, resultParalleltt, size);
#endif
   }
   timeParallelTransposeTagTeamXMM = time_call([&] {
      UseXMM = true;
      parallel_matrix_multiplyTransposeTagTeam(
         m1, m2, resultParallelttXMM, size);
      UseXMM = false;
   });
#if defined(USE_matrix_compare)
   matrix_compare(resultSerial, resultParallelttXMM, size);
#endif

   report();

   // Free the memory that was allocated for the matrices.
   destroy_matrix(m1, size);
   destroy_matrix(m2, size);
   destroy_matrix(resultSerial, size);
   destroy_matrix(resultSerialt, size);
   destroy_matrix(resultSerialtXMM, size);
   destroy_matrix(resultParallel, size);
   destroy_matrix(resultParallelt, size);
   destroy_matrix(resultParalleltXMM, size);
   destroy_matrix(resultParalleltt, size);
   destroy_matrix(resultParallelttXMM, size);
   } // end scope the qtInit object _after_ running serial test

   } // for(intptr_t size = sizeBegin;
      //    size <= sizeEnd;
      //    size += sizeIncrement)
  return 0;
}

// Creates a square matrix with the given number of rows and columns.
double** create_matrix(intptr_t size)
{
   // allocate an array of pointers for rows
   double** m = new double*[size];
   for (intptr_t i = 0; i < size; ++i)
   {
      //m[i] = new double[size];
      m[i] = cacheAligned_doubles.allocate(size);
   }
   return m;
}

// Frees the memory that was allocated for the given square matrix.
void destroy_matrix(double** m, intptr_t size)
{
   for (intptr_t i = 0; i < size; ++i)
   {
      // delete[] m[i];
      cacheAligned_doubles.deallocate( m[i], size);
   }
   delete [] m;
}

// Initializes the given square matrix with values that are generated
// by the given generator function.
template <class Generator>
double** initialize_matrix(double** m, intptr_t size, Generator& gen)
{
   for (intptr_t i = 0; i < size; ++i)
   {
      for (intptr_t j = 0; j < size; ++j)
      {
         m[i][j] = static_cast<double>(gen());
      }
   }
   return m;
}
```