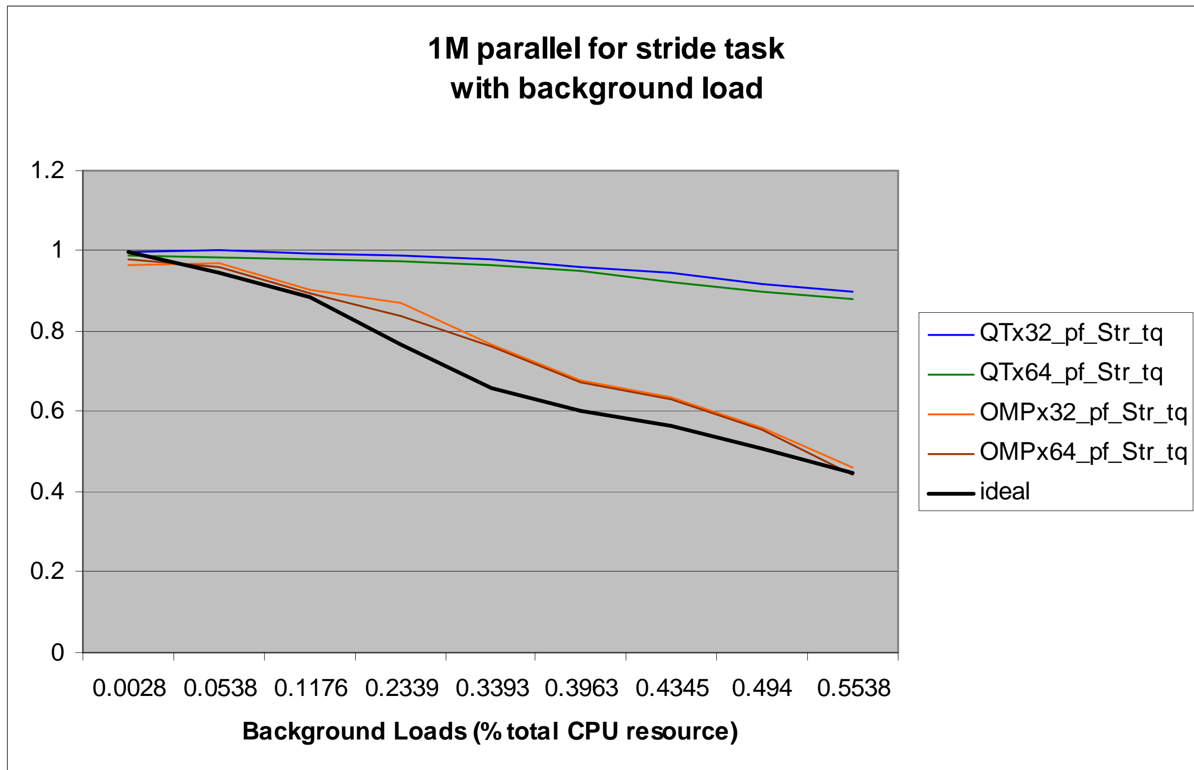
Comparison between QuickThread and OpenMP 3.0 under various system load conditions.

Copyright © 2009
QuickThread Programming, LLC
www.quickthreadprogramming.com

Copyright © 2009
QuickThread Programming, LLC



Test programs were written to produce a comparison between QuickThread and OpenMP running under various load conditions. The test program is derived from a single threaded program that simulates a processing load. This program was then used to derive two equivalent parallel programs, one using OpenMP 3.0 and a second using the QuickThread software development kit. All programs were written using Intel® C++ 11.1.068 (from Intel® Parallel Studio). Intel® is a Registered Trademark of Intel Corporation.

Each test program runs a series of tests ranging from single threaded through various degrees of parallelization in an attempt to seek maximum performance out of the test program. This test program is run under various synthetic background load conditions. All tests ran on Intel Q6600 quad core 2.4GHz system running Windows XP x64. Both 32-bit and 64-bit programs were tested.

The tests were not designed to test scalability of the application. Rather, the tests are designed to determine the effects on performance of the test application while running under different system background load conditions. Excepting for single threaded tests, all parallel tests use all cores for their thread pools.

The load conditions are broken into three groups that are of general interest to programmers:

How the program performs with no load.
How the program performs with varying loads induced by other single threaded applications.
How the program performs with varying loads induced by other instances of same program.

The varying load conditions are run with 0 to 8 instances of other single threaded applications or other instances of its self.

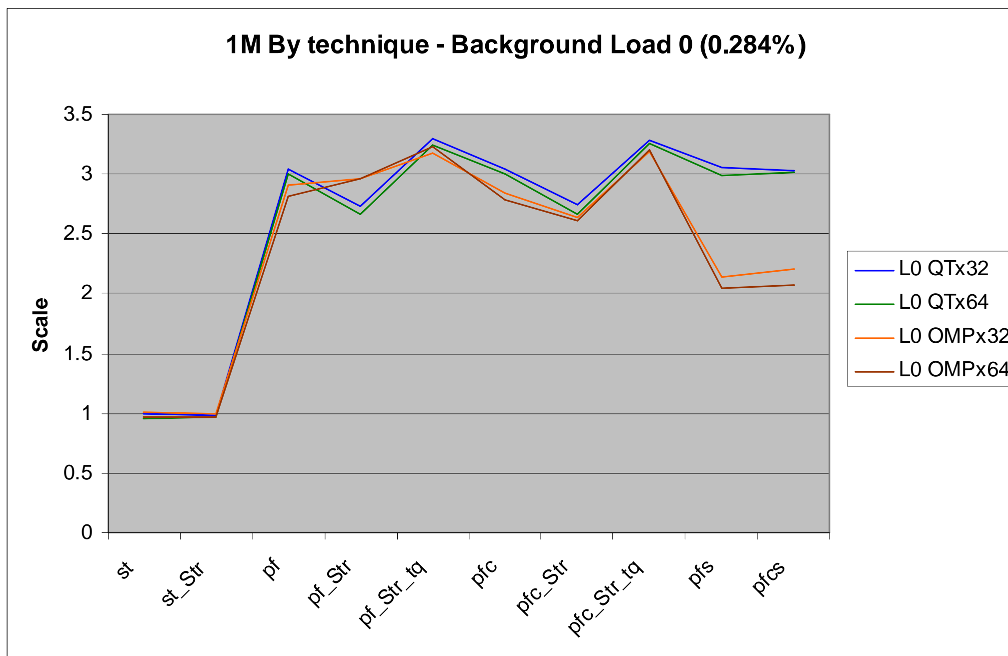
Copyright © 2009
QuickThread Programming, LLC
www.quickthreadprogramming.com

Eight different techniques were employed in each of the parallel programming language extension. Two of the techniques were serial versions of the problem to be used for reference data. While not all possible techniques were explored, the usual parallel programming techniques were explored in various permutations. The internal function suffixes are used as a key within this report.

st	Single thread
st_Str	Single thread with stride
pf	parallel for of loops
pf_Str	parallel for of inner loop slices, with slices of size stride by outer for loop
pf_Str_tq	parallel for of inner loop slices, with slices of size stride by outer for loop using tasks (equivalent statements within respective context)
pfc	parallel for of loops with dynamic scheduling and chunk
pfc_Str	parallel for of loop slices with dynamic scheduling and chunk, with slices of size stride by outer for loop
pfc_Str_tq	parallel for of loop slices with dynamic scheduling and chunk, with slices of size stride by outer for loop using tasks (equivalent statements within respective context)
pfs	parallel for combined with parallel sections
pfcs	parallel for with dynamic scheduling and chunk combined with parallel sections

Note, pfs and pfcs use sections supported by OpenMP 2.0 whereas the "…_tq" require the OpenMP 3.0 parallel task and taskq. Additional techniques and permutations were considered but rejected due to time considerations to run and assemble the permutations of data into an Excel spreadsheet.

The various parallelization strategies were run on an unloaded system. The programming technique that showed the best scaling factor on unloaded system was chosen for further analysis.
The scalability chart for No Load system follows:



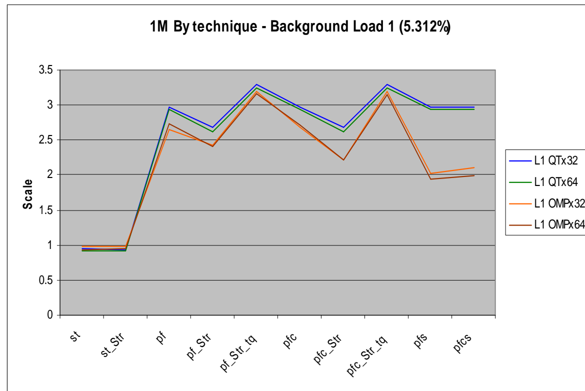
No background load

In the preceding "no load" chart the parallelization techniques pf_Str_tq and pfc_Str_tq produced best results for both QuickThread and OpenMP. When the test program array sizes were large (1024x1024 elements) the performance scaling favored QuickThread. As the array sizes were reduced, the

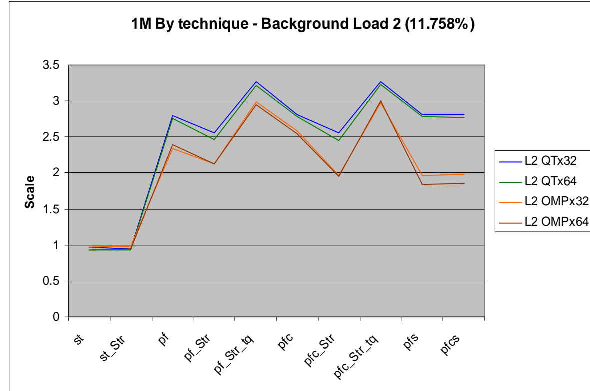
performance difference (in pf_Str_tq and pfc_Str_tq) was reduced. i.e. small array sizes may favor OpenMP in simple non-task based loop optimization sample programs. In Complex programs QuickThread will have the advantage.

For this test program, and under No Load condition, QuickThread has a small improvement over OpenMP.

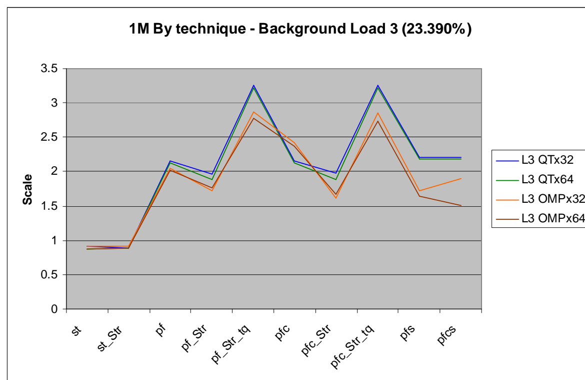
As we add background loads you will observe the performance difference grow between QuickThread and OpenMP. The tasking characteristic of QuickThread is one of the principal reasons for this benefit.



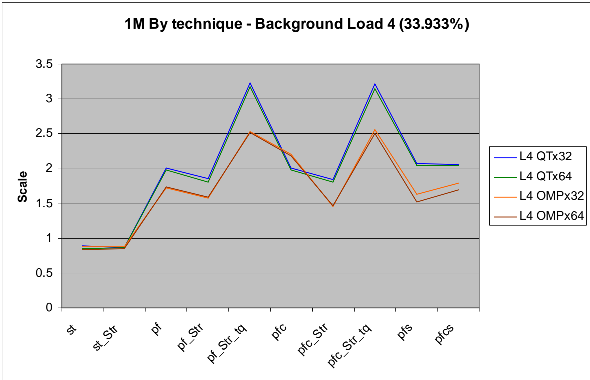
1 background application, 5.32% CPU load.



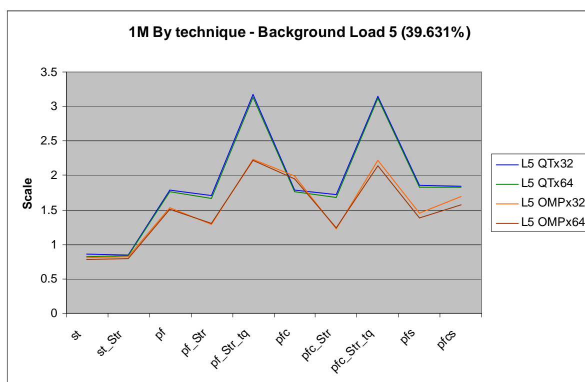
2 background applications, 11.76% CPU load.



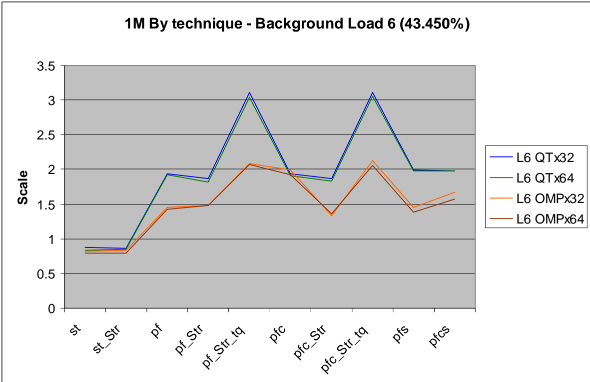
3 background applications, 23.39% CPU load.



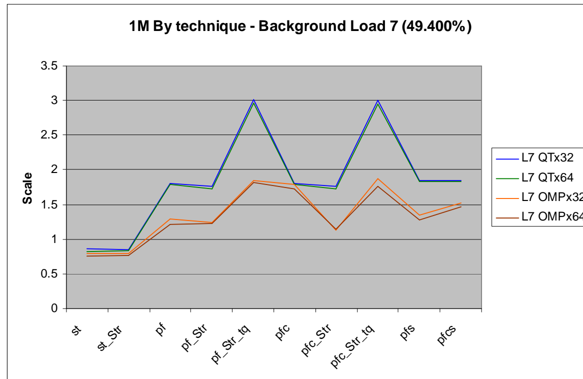
4 background applications, 33.93% CPU load.



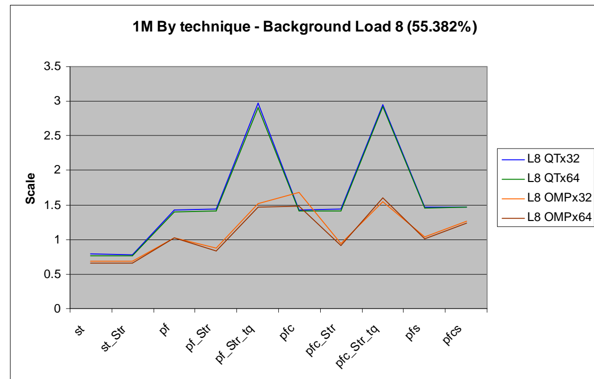
5 background applications, 39.63% CPU load.



6 background applications, 43.45% CPU load.



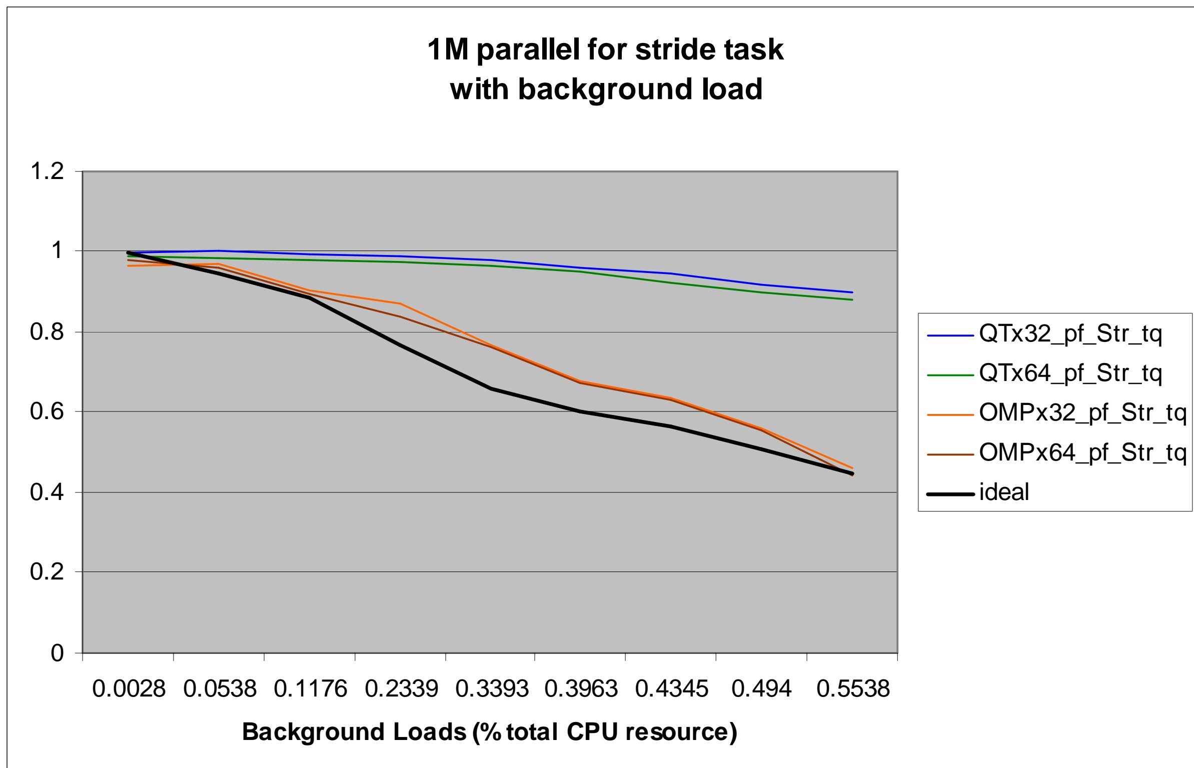
7 background applications, 49.40% CPU load.



8 background applications, 55.38% CPU load.

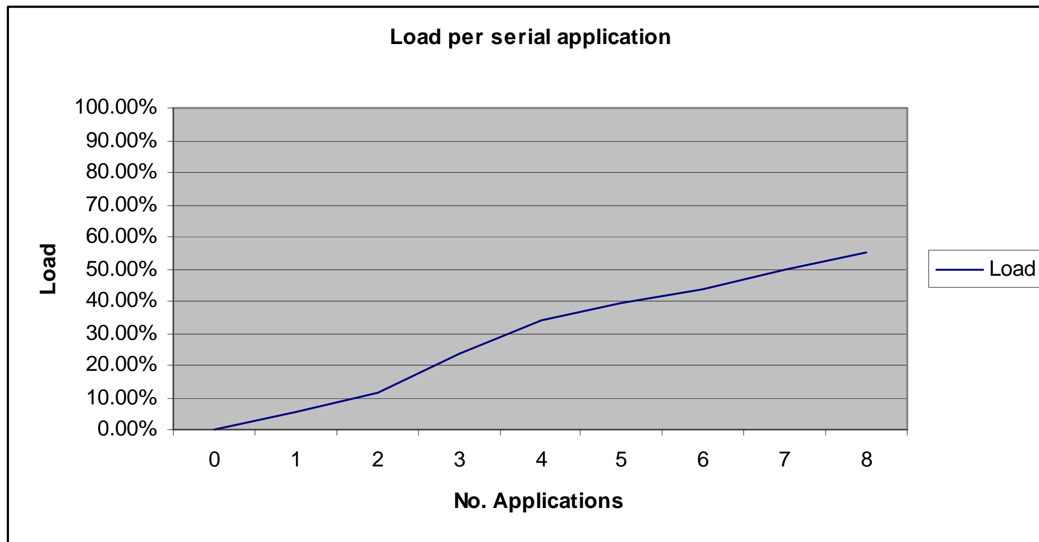
When measuring loads from a synthetic background application a “No Load” background load of 0.284% was measured. From the No Load chart (2nd preceeding page), together with all remaining load charts above, pf_Str_tq and pfc_Str_tq demonstrate to be the most favorable coding practice for both QuickThread and OpenMP. Although under the heaviest load condition the pfc (parallel for chunk) was best for OpenMP. Both parallel programming paradigms show almost the same scaling factors for each respective paradigm bit-ness (x32 vs x64). But with a slight edge towards x32.

Note, as the background load increases the scalability factor of QuickThread is significantly more favorable that that of OpenMP. The value of using QuickThread improves with system load. A performance verses load (normalized) follows:



QuickThread started with a 3.25x scaling factor under no load. As background load increased, QuickThread was able to recover some of the 0.75x difference. OpenMP did not do so well.

The other single threaded application load was intended to be a relatively low computational overhead application with no sleeping state. What was chosen for this was a batch file in a loop continually performing a DIR command. This produces a little disk I/O and a lot of display I/O, and no program sleeping states. The nine levels (0:8) of runs should be sufficient to estimate the effects on your system. Charts of the various background loads are listed in Appendix A.



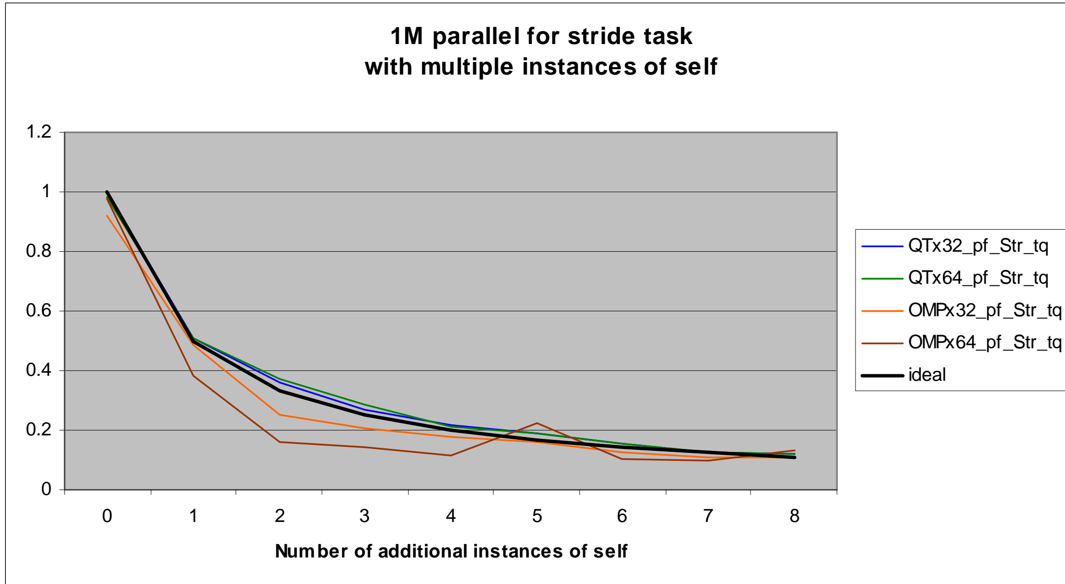
Numeric values of loads:

# Apps.	Load %
0	0.284%
1	5.382%
2	11.758%
3	23.390%
4	33.933%
5	39.631%
6	43.450%
7	49.400%
8	55.382%

Charts of the various background loads are listed in Appendix A.

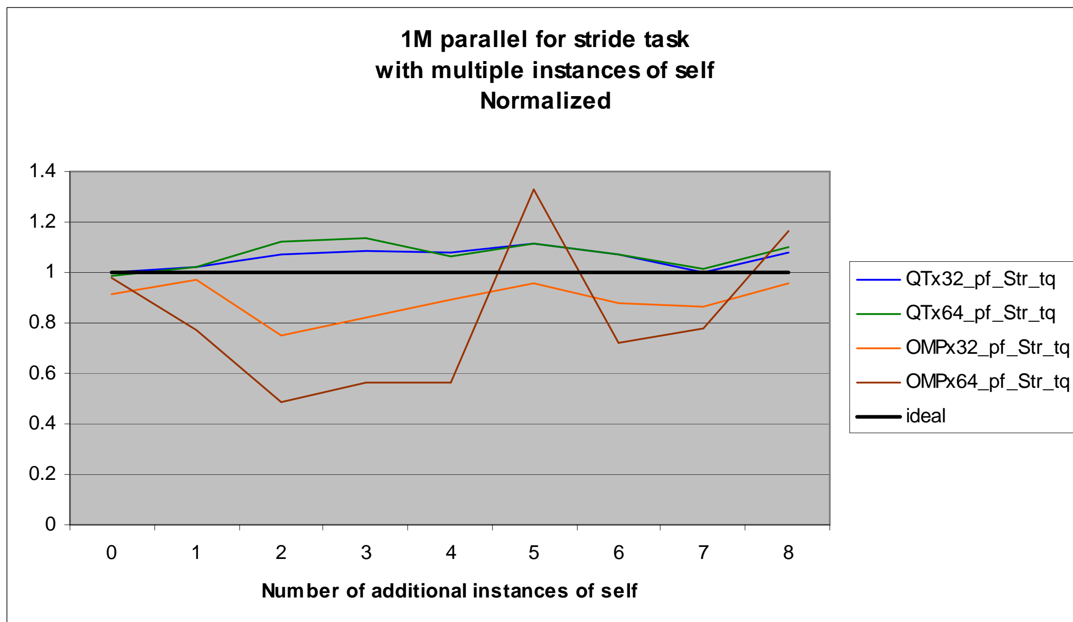
Copyright © 2009
QuickThread Programming, LLC

When comparing with multiple additional instances of the same program we find:



When no additional programs the left side scale reads 1 when the test application runs at unloaded speed. When 2 test applications run you expect 0.5, when 3 run you expect 0.333, etc… The ideal line maps the expected performance curve. QuickThread maintains an advantage over OpenMP except for a curious incident when 5 additional copies of the application is run on the 32-bit OpenMP version of the application. We haven't investigated the cause for this anomaly in the data. This is potentially a harmonic developing in the Windows scheduler.

When we straiten out the ideal line we see the chart as:



The synthetic test program has a functional diagram of:

```
loop:
        process 1
        process 2
        process 3
        process 4
end loop:
```

Processes 2 and 3 are dependent on process 1, but independent of each other. This is to say processes 2 and 3 can run in parallel. And process 4 is dependent on processes 2 and 3.

```
loop:
        process 1
  process 2, process 3
        process 4
end loop:
```

The data and procedure within all processes are non-Temporal. Meaning that each process can be sliced, and slices of the data sets can be run independently as long as the slices are properly sequenced amongst processes: 1,2,3,4 or 1, 2&3, 4.

Also, process 4 is non-conflicting with process 1. This means in the tasking model you can run process 1 of the next iteration concurrent with process 4 of the current iteration.

```
        process 1
  process 2, process 3
        process 4       process 1'
                  process 2', process 3'
                        process 4'      process 1''
                                process 2'', process 3''
                                        process 4''          ...
```

In this test program that data was collected but did not yield a benefit over the pf_Str_tq method. Since this tasking model is not available to OpenMP we though the additional run data would make the charts harder to read.

The data arrays used by the processes are aligned on cache line boundaries to avoid cache issues when compiled to run as QuickThread or OpenMP.

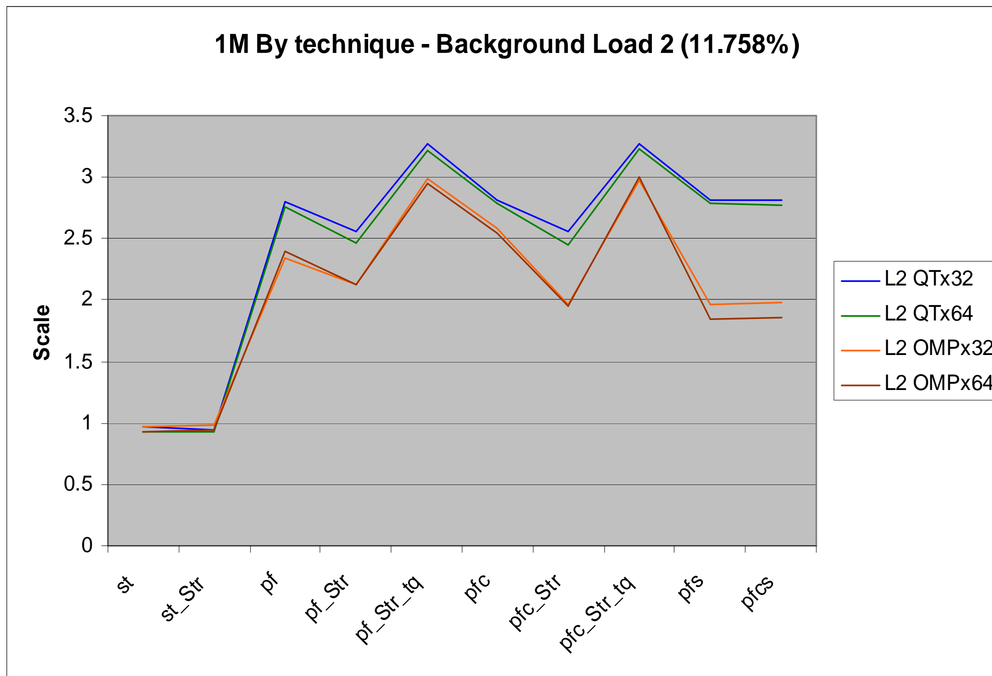
```
const int    arraySize = 1024*1024;  // power of 2

__declspec(align(64)) double A[arraySize];
__declspec(align(64)) double B[arraySize];
__declspec(align(64)) double C[arraySize];
__declspec(align(64)) double D[arraySize];
```

The size of the arrays are (1024*1024) which is 8MB, larger than the cache size on the system. The functions are meaningless other than they present simplified processes that might exist in an application.

While the data was cache aligned, no effort was made to align code. There generally is some variation in performance due to code alignment. However, code alignment should be explored as a final step in optimization when the code has firmed-up.

After the background load is established the test program runs and takes 100 sample times of each test run using various parallelization strategies. The average of the test runs is used (including MIN and MAX values). The strategies will be described following the commentary on the charts.

Copyright © 2009
QuickThread Programming, LLC



In the above chart we simulate a background system load of 11.758% consumed by two busy single threaded applications. With a moderate system load, QuickThread shows a distinct advantage over OpenMP. As the system background load increases, a greater advantage is obtained by using QuickThread.

st	Single thread
st_Str	Single thread with stride
pf	parallel for of loops
pf_Str	parallel for of inner loop slices, with slices of size stride by outer for loop
pf_Str_tq	parallel for of inner loop slices, with slices of size stride by outer for loop using tasks (equivalent statements within respective context)
pfc	parallel for of loops with dynamic scheduling and chunk
pfc_Str	parallel for of loop slices with dynamic scheduling and chunk, with slices of size stride by outer for loop
pfc_Str_tq	parallel for of loop slices with dynamic scheduling and chunk, with slices of size stride by outer for loop using tasks (equivalent statements within respective context)
pfs	parallel for combined with parallel sections
pfcs	parallel for with dynamic scheduling and chunk combined with parallel sections

Note: QuickThread does not have a parallel construct with the name taskq, however equivalent functionality is available using parallel_task (which has additional capabilities). The same name was used on the chart for tests with equivalent functionality.

When looking at the varying load conditions, QuickThread yields consistently better results than OpenMP. What this indicates is you will spend less time optimizing your code with QuickThread than you will with OpenMP.

Let's look at the programming differences. We won't list all the code, but we will list enough for you to be able to extrapolate what the code would look like.

For a complete listing comparison consult the side-by-side listing in appendix B and C

Single threaded test (st)

```
// single thread for loops
void test_st()
{
	Process1_st();
	Process2_st();
	Process3_st();
	Process4_st();
}
```

Where the process code was listed earler.

The Single thread with stride test (st_Str)

```
// single thread for loops using stride
void test_st_Stride()
{
   for(int iBegin = 0; iBegin < arraySize; iBegin+=Stride)
   {
      int iEnd = iBegin + Stride;
      if(iEnd > arraySize)
            iEnd = arraySize;
      Process1_st(iBegin, iEnd);
      Process2_st(iBegin, iEnd);
      Process3_st(iBegin, iEnd);
      Process4_st(iBegin, iEnd);
   }
}
```

The purpose of including this in the test was to provide some sense of the additional call overhead as well as any potential benefit of cache differences.

Copyright © 2009
QuickThread Programming, LLC

Parallel for test (pf)

The parallel for tests are the same algorithms as the single threaded, so we will only list the Process1_pf functions.

OpenMP

```
// process 4 - parallel for
void Process4_pf()
{
   #pragma omp parallel for
   for(int i=0; i<arraySize; ++i)
      D[i] = sqrt((B[i]*B[i])+(C[i]*C[i]));
}
```

QuickThread

```
// process 1 - parallel for
void Process1_pf()
{
   parallel_for(
      0, arraySize,
      [&](int iBg, int iEn)
      {
         for(int i=iBg; i<iEn; ++i)
            A[i] = ((double)i) / 1000.0;
         });
}
```

The QuickThread implementation is using the C++0x Lambda functions. If you prefer to not use Lambda functions then you simply take the Lambda function out and provide a name for the function and supply the half open range as arguments to the new task function.

```
// process 1 - parallel for
void Process1_pf_task(int iBg, int iEn)
{
   for(int i=iBg; i<iEn; ++i)
      A[i] = ((double)i) / 1000.0;
}

void Process1_pf()
{
   parallel_for(
      Process1_pf_task,
      0, arraySize); // as task args 1 and 2
}
```

The efficiency of the call overhead between using Lambda function or explicit function is dependent on the nature of and number of the variables passed between the caller and the called function. You may want to experiment with using both methods on a case by case basis to determine which is best.

Copyright © 2009
QuickThread Programming, LLC

Chunking version of the parallel for (pfc):

OpenMP

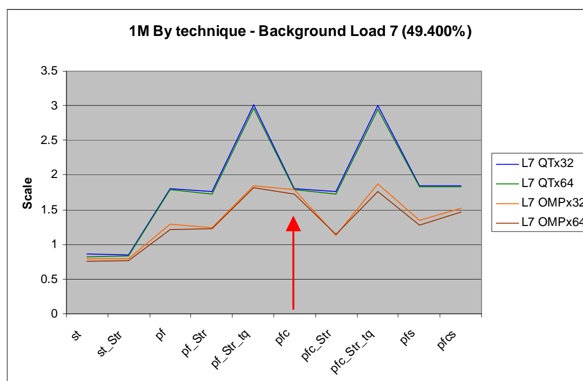
```
// process 4 - parallel for chunked
void Process4_pfc()
{
   #pragma omp parallel for schedule(dynamic, chunk)
   for(int i=0; i<arraySize; ++i)
      D[i] = sqrt((B[i]*B[i])+(C[i]*C[i]));
}
```

QuickThread

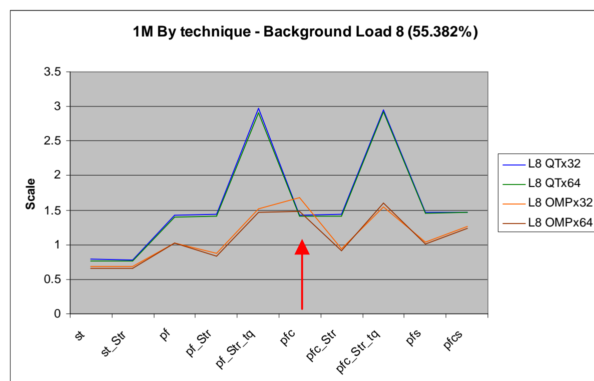
```
// process 1 - parallel for chunked
void Process1_pfc()
{
   parallel_for(
      chunk,
      0,arraySize,
      [&](int iBg, int iEn)
      {
            for(int i=iBg; i<iEn; ++i)
                  A[i] = ((double)i) / 1000.0;
      });
}
```

The use of a chunk factor is a load balancing technique. Chunking is enabled by using `schedule(dynamic, chunk)` for OpenMP, and inclusion of `chunk` argument for QuickThread.

When the system has additional processing loads, either internal to your application or external to your application, and when you have a large iteration space (or when execution time of iteration is relatively long), you can improve workload distribution by dividing the iteration space into more pieces than you have threads. When using dynamic chunking in OpenMP, and chunking in QuickThread, each chunk is acquired by threads as they become available. When the system is running other programs, the availability of cores (hardware threads) is not evenly distributed. When you use partitioning by number of threads and fixed distribution, then when other applications run, this can introduce a skew in the completion of each threads slice of the iteration space. Under background load conditions the completion skew time can be reduced by using a chunk size that divides the iteration space into more pieces than you have threads. Examine pfc data on then OpenMP (orange and brown) chart lines.



7 background applications, 49.40% CPU load.



8 background applications, 55.38% CPU load.

There is some additional overhead when dividing iteration space into finer pieces so there is a trade-off between reduction of skew and additional overhead. Evidence of this skew effect is shown in the charts for OpenMP when comparing pf and pfc as background load increases. This effect is not as apparent for QuickThread by this test program, but for other applications it will show an improvement.

Copyright © 2009
QuickThread Programming, LLC

Parallel for Stride Taskq (pf_Str_tq)

OpenMP

```
void test_pf_Stride_taskq()
{
   #pragma omp parallel
   {
      #pragma intel omp taskq
      {
         for(int iBegin = 0; iBegin < arraySize; iBegin+=Stride)
         {
            int iEnd = iBegin + Stride;
            if(iEnd > arraySize)
                  iEnd = arraySize;
            #pragma intel omp task captureprivate(iBegin,iEnd)
            {
                  Process1_pf(iBegin, iEnd);
                  Process2_pf(iBegin, iEnd);
                  Process3_pf(iBegin, iEnd);
                  Process4_pf(iBegin, iEnd);
            }
         }
      }
   }
}
```

QuickThread

```
// parallel for loops with stride and taskq
void test_pf_Stride_taskq()
{
      qtControl   StrideControl;
      for(int iBegin = 0; iBegin < arraySize; iBegin+=Stride)
      {
            int iEnd = iBegin + Stride;
            if(iEnd > arraySize)
                  iEnd = arraySize;

            // on parallel_task use [=] as opposed to [&]
            // as the values of iBegin2 and iEnd2 change
            // while building up the task list
            parallel_task(
                  &StrideControl,
                  [=]()
                  {
                        Process1_pf(iBegin, iEnd);
                        Process2_pf(iBegin, iEnd);
                        Process3_pf(iBegin, iEnd);
                        Process4_pf(iBegin, iEnd);
                  });
      }
}
```

The programming technique that exhibits the best performance under varying background loads is the pf_Str_tq - `test_pf_Stride_taskq(`. In this technique the large iteration space is divided up into

Stride sized slices, each slice becoming a task to sequentially run processes 1 through 4 on the slice while using parallel for within each process. Chunking was also experimented with the pfc_Str_tq test without appreciable difference in performance (see charts). Also tested, but not shown on the charts was to attempt further parallelization in pf_Str_tq by making process 2 and 3 run in parallel. For this test case, both QuickThread and OpenMP had negative returns on further attempts at parallelization.

Copyright © 2009
QuickThread Programming, LLC

Parallel for in sections (pfs)

OpenMP

```
// multi-threaded parallel for loop sections
void test_pfs()
{
	Process1_pf();
	#pragma omp parallel sections
	{
		#pragma omp section
		{
			Process2_pf();
		}
		#pragma omp section
		{
			Process3_pf();
		}
	}
	Process4_pf();
}
```

QuickThread

```
// multi-threaded parallel for loop sections
void test_pfs()
{
	Process1_pf();
	parallel_invoke(
		[&]() { Process2_pf(); },
		[&]() { Process3_pf(); });
	Process4_pf();
}
```

Parallel for sections was included as a reference for programmers using OpenMP 2.0 (prior to OpenMP 3.0 taskq and task).

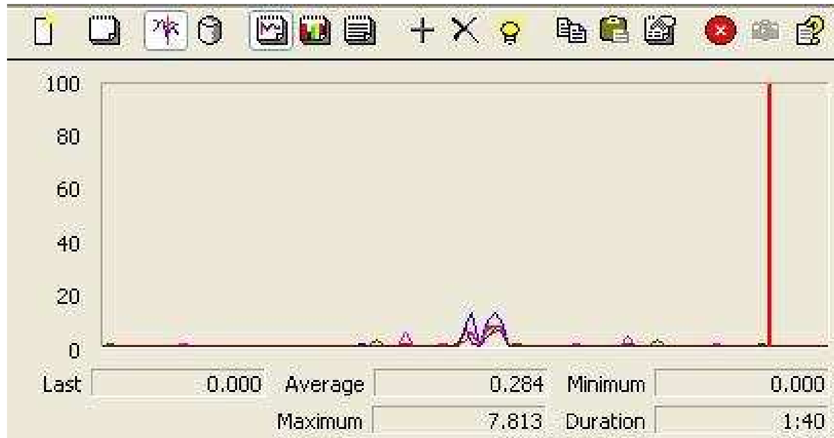
As you can see from the code snips there is some difference in programming style for the parallel constructs between OpenMP and QuickThread. Consistent performance under varying load conditions is a major plus when using QuickThread as well as slightly better performance under ideal (no load) conditions.

This test program is a relatively trivial problem. Real applications will have a higher level of complexity. The true benefit of using the tasking style scheduling system of QuickThread will improve as the complexity of the program increases.

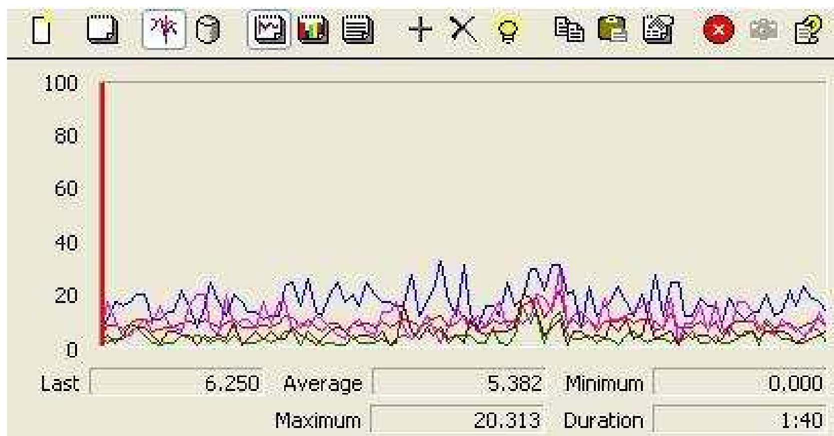
See Appendix B and C for complete coding differences.

Appendix A

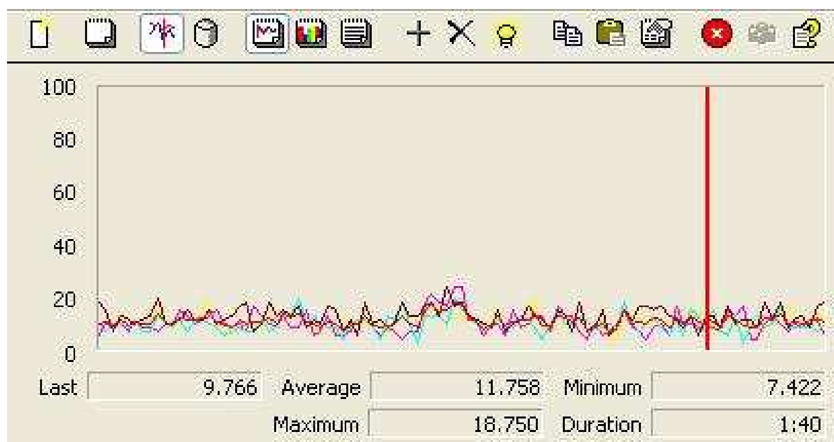
Charts of background loads induced as single threaded applications



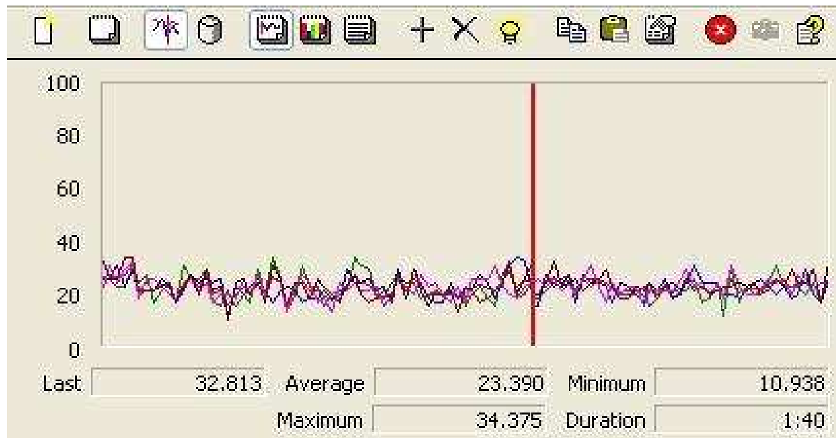
The No Load state has 0.284% overhead



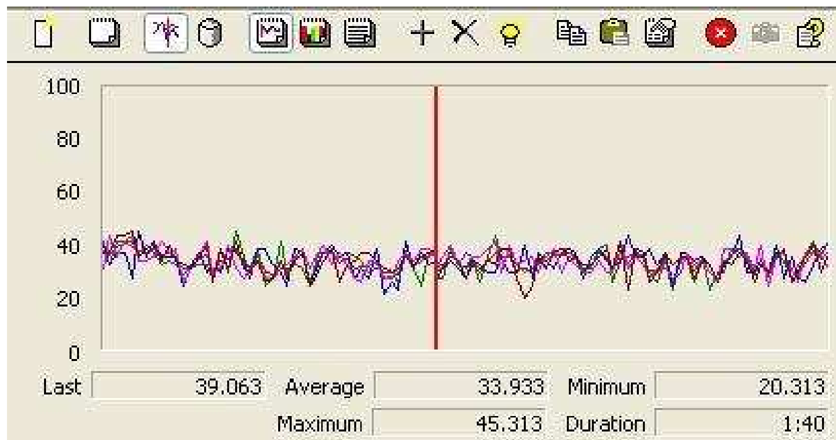
The 1 application load has 5.382% overhead.



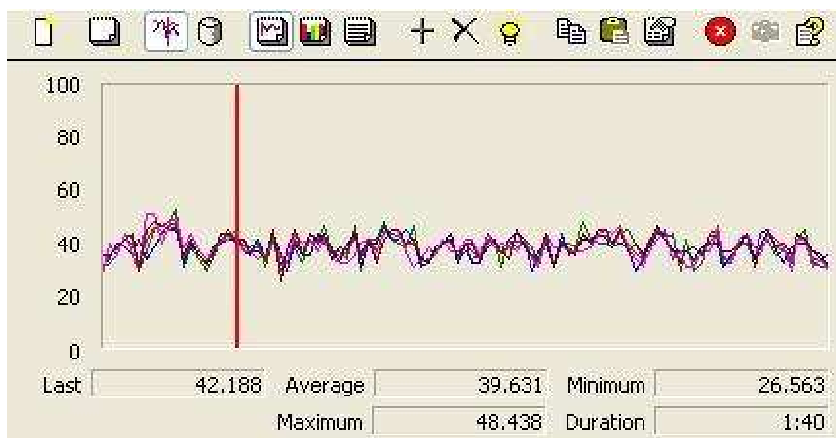
The 2 application load has 11.758% overhead.



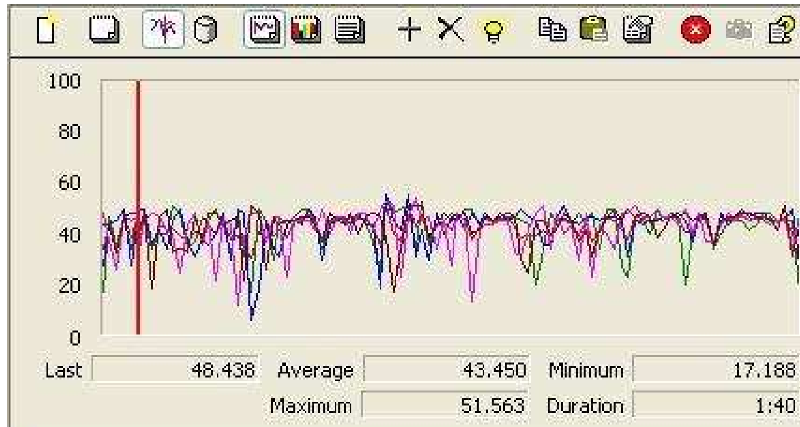
The 3 application load has 23.390% overhead.



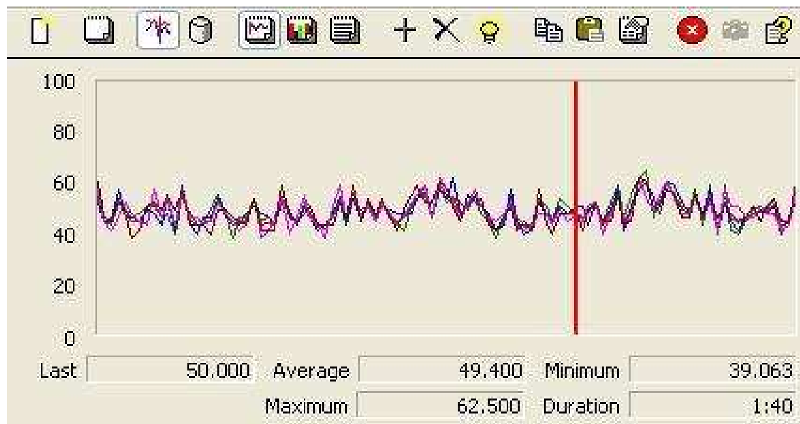
The 4 application load has 33.933% overhead.



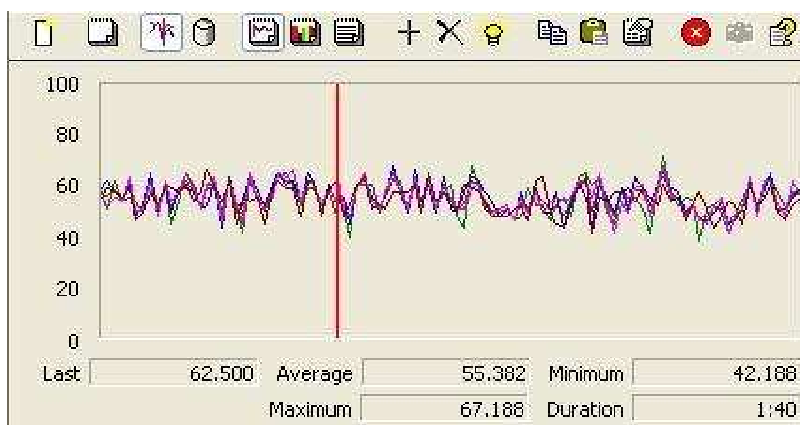
The 5 application load has 39.631% overhead.



The 6 application load has 43.450% overhead.



The 7 application load has 49.400% overhead.



The 8 application load has 55.382% overhead.

Appendix B – OpenMP version of test program

```
// WhiteRabbitOMP.cpp
//

#include "stdafx.h"
#include "omp.h"

// power of 2 for test
const intarraySize = 1024*1024;

__declspec(align(64)) double A[arraySize];
__declspec(align(64)) double B[arraySize];
__declspec(align(64)) double C[arraySize];
__declspec(align(64)) double D[arraySize];

// number of test runss should be enough
// to observe good trend
const int numRuns = 100;

int Stride = 8192;
// chunking factor for loop
int chunk =1024;
```

Appendix C – QuickThread version of test program

```
// WhiteRabbit1QT.cpp
//

#include "stdafx.h"


// power of 2 for test power of 2
const intarraySize = 1024*1024;

__declspec(align(64)) double A[arraySize];
__declspec(align(64)) double B[arraySize];
__declspec(align(64)) double C[arraySize];
__declspec(align(64)) double D[arraySize];

// number of test runss should be enough
// to observe good trend
const int numRuns = 100;

int Stride = 8192;
// chunking factor for loop
int chunk = 1024;
```

Copyright © 2009
QuickThread Programming, LLC


```
// process 1 - single threaded
void Process1_st()
{
   for(int i=0; i<arraySize; ++i)
      A[i] = ((double)i) / 1000.0;
}

// process 1 - single threaded slice
void Process1_st(int iBegin, int iEnd)
{
   for(int i=iBegin; i<iEnd; ++i)
      A[i] = ((double)i) / 1000.0;
}

// process 1 - parallel for
void Process1_pf()
{
   #pragma omp parallel for
   for(int i=0; i<arraySize; ++i)
      A[i] = ((double)i) / 1000.0;
}

// process 1 - parallel for slice
void Process1_pf(int iBegin, int iEnd)
{
   #pragma omp parallel for
   for(int i=iBegin; i<iEnd; ++i)
      A[i] = ((double)i) / 1000.0;
}

// process 1 - parallel for chunked
void Process1_pfc()
{
#pragma omp parallel for schedule(dynamic,
chunk)
   for(int i=0; i<arraySize; ++i)
      A[i] = ((double)i) / 1000.0;
}
```

```
// process 1 - single threaded
void Process1_st()
{
   for(int i=0; i<arraySize; ++i)
      A[i] = ((double)i) / 1000.0;
}

// process 1 - single threaded slice
void Process1_st(int iBegin, int iEnd)
{
   for(int i=iBegin; i<iEnd; ++i)
      A[i] = ((double)i) / 1000.0;
}

// process 1 - parallel for
void Process1_pf()
{
   parallel_for(
      0,arraySize,
      [&](int iBg, int iEn)
     {
      for(int i=iBg; i<iEn; ++i)
         A[i] = ((double)i) / 1000.0;
      });
}

// process 1 - parallel for slice
void Process1_pf(int iBegin, int iEnd)
{
   parallel_for(
      iBegin,iEnd,
      [&](int iBg, int iEn)
     {
      for(int i=iBg; i<iEn; ++i)
         A[i] = ((double)i) / 1000.0;
      });
}

// process 1 - parallel for chunked
void Process1_pfc()
{
   parallel_for(
      chunk,0,arraySize,
      [&](int iBg, int iEn)
     {
      for(int i=iBg; i<iEn; ++i)
         A[i] = ((double)i) / 1000.0;
   });
}
```

Copyright © 2009
QuickThread Programming, LLC
www.quickthreadprogramming.com

```
// process 1 - parallel for chunked slice
void Process1_pfc(int iBegin, int iEnd)
{
   #pragma omp parallel for schedule(dynamic,
chunk)
   for(int i=iBegin; i<iEnd; ++i)
      A[i] = ((double)i) / 1000.0;
}
```

```
// process 1 - parallel for chunked slice
void Process1_pfc(int iBegin, int iEnd)
{
   parallel_for(
      chunk,iBegin,iEnd,
      [&](int iBg, int iEn)
     {
      for(int i=iBg; i<iEn; ++i)
         A[i] = ((double)i) / 1000.0;
      });
}
```

```
// process 2 - single thread
void Process2_st()
{
   for(int i=0; i<arraySize; ++i)
      B[i] = sin(A[i]);
}
```

```
// process 2 - single thread
void Process2_st()
{
   for(int i=0; i<arraySize; ++i)
      B[i] = sin(A[i]);
}
```

```
// process 2 - single thread slice
void Process2_st(int iBegin, int iEnd)
{
   for(int i=iBegin; i<iEnd; ++i)
      B[i] = sin(A[i]);
}
```

```
// process 2 - single thread slice
void Process2_st(int iBegin, int iEnd)
{
   for(int i=iBegin; i<iEnd; ++i)
      B[i] = sin(A[i]);
}
```

```
// process 2 - parallel for
void Process2_pf()
{
   #pragma omp parallel for
   for(int i=0; i<arraySize; ++i)
      B[i] = sin(A[i]);
}
```

```
// process 2 - parallel for
void Process2_pf()
{
   parallel_for(
      0,arraySize,
      [&](int iBg, int iEn)
     {
      for(int i=iBg; i<iEn; ++i)
         B[i] = sin(A[i]);
      });
}
```

```
// process 2 - parallel for slice
void Process2_pf(int iBegin, int iEnd)
{
   #pragma omp parallel for
   for(int i=iBegin; i<iEnd; ++i)
      B[i] = sin(A[i]);
}
```

```
// process 2 - parallel for slice
void Process2_pf(int iBegin, int iEnd)
{
   parallel_for(
      iBegin,iEnd,
      [&](int iBg, int iEn)
     {
      for(int i=iBg; i<iEn; ++i)
         B[i] = sin(A[i]);
      });
}
```

Copyright © 2009
QuickThread Programming, LLC


```
// process 2 - parallel for chunked
void Process2_pfc()
{
   #pragma omp parallel for schedule(dynamic,
chunk)
   for(int i=0; i<arraySize; ++i)
      B[i] = sin(A[i]);
}
```

```
// process 2 - parallel for chunked
void Process2_pfc()
{
   parallel_for( chunk,0,arraySize,
      [&](int iBg, int iEn)
     {
      for(int i=iBg; i<iEn; ++i)
         B[i] = sin(A[i]);
      });
}
```

```
// process 2 - parallel for chunked slice
void Process2_pfc(int iBegin, int iEnd)
{
   #pragma omp parallel for schedule(dynamic,
chunk)
   for(int i=iBegin; i<iEnd; ++i)
      B[i] = sin(A[i]);
}
```

```
// process 2 - parallel for chunked slice
void Process2_pfc(int iBegin, int iEnd)
{
   parallel_for( chunk,iBegin,iEnd,
      [&](int iBg, int iEn)
     {
      for(int i=iBg; i<iEn; ++i)
         B[i] = sin(A[i]);
      });
}
```

```
// process 3 - single thread
void Process3_st()
{
   for(int i=0; i<arraySize; ++i)
      C[i] = cos(A[i]);
}
```

```
// process 3 - single thread
void Process3_st()
{
   for(int i=0; i<arraySize; ++i)
      C[i] = cos(A[i]);
}
```

```
// process 3 - single thread slice
void Process3_st(int iBegin, int iEnd)
{
   for(int i=iBegin; i<iEnd; ++i)
      C[i] = cos(A[i]);
}
```

```
// process 3 - single thread slice
void Process3_st(int iBegin, int iEnd)
{
   for(int i=iBegin; i<iEnd; ++i)
      C[i] = cos(A[i]);
}
```

```
// process 3 - parallel for
void Process3_pf()
{
   #pragma omp parallel for
   for(int i=0; i<arraySize; ++i)
      C[i] = cos(A[i]);
}
```

```
// process 3 - parallel for
void Process3_pf()
{
   parallel_for( 0,arraySize,
      [&](int iBg, int iEn)
     {
      for(int i=iBg; i<iEn; ++i)
         C[i] = cos(A[i]);
      });
}
```

```
// process 3 - parallel for slice
void Process3_pf(int iBegin, int iEnd)
{
   #pragma omp parallel for
   for(int i=iBegin; i<iEnd; ++i)
      C[i] = cos(A[i]);
}
```

```
// process 3 - parallel for slice
void Process3_pf(int iBegin, int iEnd)
{
parallel_for( iBegin,iEnd,
   [&](int iBg, int iEn)
     {
      for(int i=iBg; i<iEn; ++i)
         C[i] = cos(A[i]);
   });
}
```

V1.0.1
Copyright © 2009
QuickThread Programming, LLC
www.quickthreadprogramming.com

```
// process 3 - parallel for chunked
void Process3_pfc()
{
   #pragma omp parallel for schedule(dynamic,
chunk)
   for(int i=0; i<arraySize; ++i)
      C[i] = cos(A[i]);
}


// process 3 - parallel for chunked slice
void Process3_pfc(int iBegin, int iEnd)
{
   #pragma omp parallel for schedule(dynamic,
chunk)
   for(int i=iBegin; i<iEnd; ++i)
      C[i] = cos(A[i]);
}


// process 4 - single thread
void Process4_st()
{
   for(int i=0; i<arraySize; ++i)
      D[i] = sqrt((B[i]*B[i])+(C[i]*C[i]));
}

// process 4 - single thread slice
void Process4_st(int iBegin, int iEnd)
{
   for(int i=iBegin; i<iEnd; ++i)
      D[i] = sqrt((B[i]*B[i])+(C[i]*C[i]));
}

// process 4 - parallel for
void Process4_pf()
{
   #pragma omp parallel for
   for(int i=0; i<arraySize; ++i)
      D[i]=sqrt((B[i]*B[i])+(C[i]*C[i]));
}


// process 4 - parallel for slice
void Process4_pf(int iBegin, int iEnd)
{
   #pragma omp parallel for
   for(int i=iBegin; i<iEnd; ++i)
      D[i] = sqrt((B[i]*B[i])+(C[i]*C[i]));
}
```

```
// process 3 - parallel for chunked
void Process3_pfc()
{
   parallel_for( chunk, 0,arraySize,
      [&](int iBg, int iEn)
     {
      for(int i=iBg; i<iEn; ++i)
         C[i] = cos(A[i]);
      });
}

// process 3 - parallel for chunked slice
void Process3_pfc(int iBegin, int iEnd)
{
   parallel_for( chunk,iBegin,iEnd,
   [&](int iBg, int iEn)
     {
      for(int i=iBg; i<iEn; ++i)
         C[i] = cos(A[i]);
   });
}

// process 4 - single thread
void Process4_st()
{
   for(int i=0; i<arraySize; ++i)
      D[i] = sqrt((B[i]*B[i])+(C[i]*C[i]));
}

// process 4 - single thread slice
void Process4_st(int iBegin, int iEnd)
{
   for(int i=iBegin; i<iEnd; ++i)
      D[i] = sqrt((B[i]*B[i])+(C[i]*C[i]));
}

// process 4 - parallel for
void Process4_pf()
{
   parallel_for( 0,arraySize,
      [&](int iBg, int iEn)
     {
      for(int i=iBg; i<iEn; ++i)
         D[i]=sqrt((B[i]*B[i])+(C[i]*C[i]));
   });
}

// process 4 - parallel for slice
void Process4_pf(int iBegin, int iEnd)
{
  parallel_for( iBegin,iEnd,
    [&](int iBg, int iEn)
    {
      for(int i=iBg; i<iEn; ++i)
         D[i]=sqrt((B[i]*B[i])+(C[i]*C[i]));
     });
}
```

Copyright © 2009
QuickThread Programming, LLC


```
// process 4 - parallel for chunked
void Process4_pfc()
{
   #pragma omp parallel for schedule(dynamic,
chunk)
   for(int i=0; i<arraySize; ++i)
      D[i]=sqrt((B[i]*B[i])+(C[i]*C[i]));
}


// process 4 - parallel for chunked slice
void Process4_pfc(int iBegin, int iEnd)
{
   #pragma omp parallel for schedule(dynamic,
chunk)
   for(int i=iBegin; i<iEnd; ++i)
      D[i] = sqrt((B[i]*B[i])+(C[i]*C[i]));
}


// single thread for loops
void test_st()
{
   Process1_st();
   Process2_st();
   Process3_st();
   Process4_st();
}
__int64test_st_results[numRuns+1];

// single thread for loops using stride
void test_st_Stride()
{
   for(int iBegin = 0;
       iBegin < arraySize;
       iBegin+=Stride)
  {
   int iEnd = iBegin + Stride;
   if(iEnd > arraySize)
      iEnd = arraySize;
   Process1_st(iBegin, iEnd);
   Process2_st(iBegin, iEnd);
   Process3_st(iBegin, iEnd);
   Process4_st(iBegin, iEnd);
  }
}
__int64test_st_Stride_results[numRuns+1];
```

```
// process 4 - parallel for chunked
void Process4_pfc()
{
   parallel_for( chunk,0,arraySize,
      [&](int iBg, int iEn)
     {
      for(int i=iBg; i<iEn; ++i)
         D[i]=sqrt((B[i]*B[i])+(C[i]*C[i]));
      });
}

// process 4 - parallel for chunked slice
void Process4_pfc(int iBegin, int iEnd)
{
   parallel_for( chunk,iBegin,iEnd,
      [&](int iBg, int iEn)
     {
      for(int i=iBg; i<iEn; ++i)
         D[i]=sqrt((B[i]*B[i])+(C[i]*C[i]));
   });
}

// single thread for loops
void test_st()
{
   Process1_st();
   Process2_st();
   Process3_st();
   Process4_st();
}
__int64test_st_results[numRuns+1];

// single thread for loops using stride
void test_st_Stride()
{
   for(int iBegin = 0;
       iBegin < arraySize;
       iBegin+=Stride)
  {
   int iEnd = iBegin + Stride;
   if(iEnd > arraySize)
      iEnd = arraySize;
   Process1_st(iBegin, iEnd);
   Process2_st(iBegin, iEnd);
   Process3_st(iBegin, iEnd);
   Process4_st(iBegin, iEnd);
  }
}
__int64test_st_Stride_results[numRuns+1];
```

```
// parallel for loops
void test_pf()
{
   Process1_pf();
   Process2_pf();
   Process3_pf();
   Process4_pf();
}
__int64test_pf_results[numRuns+1];

// parallel for loops with stride
void test_pf_Stride()
{
   for(int iBegin = 0;
       iBegin < arraySize;
       iBegin+=Stride)
  {
   int iEnd = iBegin + Stride;
   if(iEnd > arraySize)
      iEnd = arraySize;
   Process1_pf(iBegin, iEnd);
   Process2_pf(iBegin, iEnd);
   Process3_pf(iBegin, iEnd);
   Process4_pf(iBegin, iEnd);
  }
}
__int64test_pf_Stride_results[numRuns+1];
```

```
// parallel for loops
void test_pf()
{
   Process1_pf();
   Process2_pf();
   Process3_pf();
   Process4_pf();
}
__int64test_pf_results[numRuns+1];

// parallel for loops with stride
void test_pf_Stride()
{
   for(int iBegin = 0;
       iBegin < arraySize;
       iBegin+=Stride)
  {
   int iEnd = iBegin + Stride;
   if(iEnd > arraySize)
      iEnd = arraySize;
   Process1_pf(iBegin, iEnd);
   Process2_pf(iBegin, iEnd);
   Process3_pf(iBegin, iEnd);
   Process4_pf(iBegin, iEnd);
  }
}
__int64test_pf_Stride_results[numRuns+1];
```

```
// parallel for loops with stride and taskq
void test_pf_Stride_taskq()
{
   #pragma omp parallel
  {
      #pragma intel omp taskq
     {
      for(int iBegin = 0;
          iBegin < arraySize;
          iBegin+=Stride)
        {
         int iEnd = iBegin + Stride;
         if(iEnd > arraySize)
            iEnd = arraySize;
         #pragma intel omp task
captureprivate(iBegin,iEnd)
         {
            Process1_pf(iBegin, iEnd);
            Process2_pf(iBegin, iEnd);
            Process3_pf(iBegin, iEnd);
            Process4_pf(iBegin, iEnd);
         }
        }
     }
  }
}
__int64test_pf_Stride_taskq_results[numRuns+1];

// parallel for loops chunked
void test_pfc()
{
   Process1_pfc();
   Process2_pfc();
   Process3_pfc();
   Process4_pfc();
}
__int64test_pfc_results[numRuns+1];

// parallel for loops chunked with stride
void test_pfc_Stride()
{
   for(int iBegin = 0;
       iBegin < arraySize;
       iBegin+=Stride)
  {
   int iEnd = iBegin + Stride;
   if(iEnd > arraySize)
      iEnd = arraySize;
   Process1_pfc(iBegin, iEnd);
   Process2_pfc(iBegin, iEnd);
   Process3_pfc(iBegin, iEnd);
   Process4_pfc(iBegin, iEnd);
  }
}
__int64test_pfc_Stride_results[numRuns+1];
```

```
// parallel for loops with stride and taskq
void test_pf_Stride_taskq()
{
   qtControlStrideControl;
   for(int iBegin = 0;
       iBegin < arraySize;
       iBegin+=Stride)
  {
   int iEnd = iBegin + Stride;
   if(iEnd > arraySize)
      iEnd = arraySize;
   // on parallel_task use [=]
   // as opposed to [&]
   // values of iBegin2 and iEnd2 change
   // while building up the task list
   parallel_task(
      &StrideControl,
      [=]()
        {
         Process1_pf(iBegin, iEnd);
         Process2_pf(iBegin, iEnd);
         Process3_pf(iBegin, iEnd);
         Process4_pf(iBegin, iEnd);
      });
  }
}
__int64test_pf_Stride_taskq_results[numRuns+1];

// parallel for loops chunked
void test_pfc()
{
   Process1_pfc();
   Process2_pfc();
   Process3_pfc();
   Process4_pfc();
}
__int64test_pfc_results[numRuns+1];

// parallel for loops chunked with stride
void test_pfc_Stride()
{
   for(int iBegin = 0;
       iBegin < arraySize;
       iBegin+=Stride)
  {
   int iEnd = iBegin + Stride;
   if(iEnd > arraySize)
      iEnd = arraySize;
   Process1_pfc(iBegin, iEnd);
   Process2_pfc(iBegin, iEnd);
   Process3_pfc(iBegin, iEnd);
   Process4_pfc(iBegin, iEnd);
  }
}
__int64test_pfc_Stride_results[numRuns+1];
```

Copyright © 2009
QuickThread Programming, LLC
www.quickthreadprogramming.com

```
// parallel for loops chunked
// with stride and taskq
void test_pfc_Stride_taskq()
{
   #pragma omp parallel
  {
      #pragma intel omp taskq
     {
      for(int iBegin = 0;
          iBegin < arraySize;
          iBegin+=Stride)
        {
         int iEnd = iBegin + Stride;
         if(iEnd > arraySize)
            iEnd = arraySize;
         #pragma intel omp task
captureprivate(iBegin,iEnd)
           {
            Process1_pfc(iBegin, iEnd);
            Process2_pfc(iBegin, iEnd);
            Process3_pfc(iBegin, iEnd);
            Process4_pfc(iBegin, iEnd);
           }
        }
     }
  }
}
__int64
test_pfc_Stride_taskq_results[numRuns+1];

// multi-threaded parallel for loop sections
void test_pfs()
{
   Process1_pf();
   #pragma omp parallel sections
  {
   #pragma omp section
     {
      Process2_pf();
     }
   #pragma omp section
     {
      Process3_pf();
     }
  }
   Process4_pf();
}
__int64test_pfs_results[numRuns+1];
```

```
// parallel for loops chunked
// with stride and taskq
void test_pfc_Stride_taskq()
{
   qtControlStrideControl;
   for(int iBegin = 0;
       iBegin < arraySize;
       iBegin+=Stride)
  {
   int iEnd = iBegin + Stride;
   if(iEnd > arraySize)
      iEnd = arraySize;
   // on parallel_task use [=]
   // as opposed to [&]
   // values of iBegin2 and iEnd2 change
   // while building up the task list
   parallel_task(
      &StrideControl,
      [=]()
        {
         Process1_pfc(iBegin, iEnd);
         Process2_pfc(iBegin, iEnd);
         Process3_pfc(iBegin, iEnd);
         Process4_pfc(iBegin, iEnd);
      });
  }
}
__int64
test_pfc_Stride_taskq_results[numRuns+1];

// multi-threaded parallel for loop sections
void test_pfs()
{
   Process1_pf();
   parallel_invoke(
      [&]() { Process2_pf(); },
      [&]() { Process3_pf(); });
   Process4_pf();
}
__int64test_pfs_results[numRuns+1];
```

```
// multi-threaded parallel for loop sections
void test_pfcs()
{
   Process1_pfc();
   #pragma omp parallel sections
  {
      #pragma omp section
     {
      Process2_pfc();
     }
      #pragma omp section
     {
      Process3_pfc();
     }
  }
   Process4_pfc();
}
__int64test_pfcs_results[numRuns+1];

int numberOfItrations = 1;

void test(__int64 t[], void (*fn)())
{
   t[0] = __rdtsc();     // get start time
   for(int iRun = 1;
       iRun <= numRuns;
       ++ iRun)
  {
      t[iRun] = __rdtsc();// get start time
      for(int i=0;
          i<numberOfItrations;
          ++i)
         fn(); // run process function
      // elapse time
      t[iRun] = (__rdtsc() - t[iRun]);
  }
   t[0] = (__rdtsc() - t[0]); // elapse time
}

int testNumber = 0;// 0 == all
```

```
// multi-threaded parallel for loop sections
void test_pfcs()
{
   Process1_pfc();
   parallel_invoke(
      [&]() { Process2_pfc(); },
      [&]() { Process3_pfc(); });
   Process4_pfc();
}
__int64test_pfcs_results[numRuns+1];

int numberOfIterations = 1;

void test(__int64 t[], void (*fn)())
{
   t[0] = __rdtsc();     // get start time
   for(int iRun = 1;
       iRun <= numRuns;
       ++ iRun)
  {
      t[iRun] = __rdtsc();// get start time
      for(int i=0;
          i<numberOfIterations;
          ++i)
         fn(); // run process function
      // elapse time
      t[iRun] = (__rdtsc() - t[iRun]);
  }
   t[0] = (__rdtsc() - t[0]); // elapse time
}

int testNumber = 0;// 0 == all
```

V1.0.1
Copyright © 2009
QuickThread Programming, LLC
www.quickthreadprogramming.com

```
int main(int argc, char* argv[])
{
  if(argc > 1)
  {
   std::cout << "argv[1]=" << argv[1];
   numberOfItrations = atol(argv[1]);
   if(numberOfItrations <= 0)
       numberOfItrations = 1;
  }
  if(argc > 2)
  {
   std::cout << "argv[2]=" << argv[2];
   testNumber = atol(argv[2]);
  }
  if(argc > 1)
    std::cout << std::endl;




  // pick up parameters for use inside loop
  // thread count
  intmaxThreads = omp_get_max_threads( );

  std::cout << "HW Threads = "
  << maxThreads << " Number Of Itrations = "
  << numberOfItrations << std::endl;

  if((testNumber == 0) || (testNumber == 1))
    test(test_st_results, test_st);
  if((testNumber == 0) || (testNumber == 2))
    test(test_st_Stride_results,
         test_st_Stride);
  if((testNumber == 0) || (testNumber == 3))
    test(test_pf_results, test_pf);
  if((testNumber == 0) || (testNumber == 4))
    test(test_pf_Stride_results,
         test_pf_Stride);
  if((testNumber == 0) || (testNumber == 5))
    test(test_pf_Stride_taskq_results,
         test_pf_Stride_taskq);
  if((testNumber == 0) || (testNumber == 6))
    test(test_pfc_results, test_pfc);
  if((testNumber == 0) || (testNumber == 7))
    test(test_pfc_Stride_results,
         test_pfc_Stride);
  if((testNumber == 0) || (testNumber == 8))
    test(test_pfc_Stride_taskq_results,
         test_pfc_Stride_taskq);
  if((testNumber == 0) || (testNumber == 9))
    test(test_pfs_results, test_pfs);
  if((testNumber == 0) || (testNumber ==10))
    test(test_pfcs_results, test_pfcs);
```

```
int main(int argc, char* argv[])
{
  if(argc > 1)
  {
   std::cout << "argv[1]=" << argv[1];
   numberOfIterations = atol(argv[1]);
   if(numberOfIterations <= 0)
       numberOfIterations = 1;
  }
  if(argc > 2)
  {
   std::cout << "argv[2]=" << argv[2];
   testNumber = atol(argv[2]);
  }
  if(argc > 1)
    std::cout << std::endl;

   // initialize QuickThread thread pool
   // all compute threads, no I/O threads
   qtInitqtInit(-1,0);

  // pick up parameters for use inside loop
  // thread count
  int maxThreads=(int)qt_get_num_threads();

  std::cout << "HW Threads = "
  << maxThreads << " Number Of Itrations = "
  << numberOfIterations  << std::endl;

  if((testNumber == 0) || (testNumber == 1))
    test(test_st_results, test_st);
  if((testNumber == 0) || (testNumber == 2))
    test(test_st_Stride_results,
         test_st_Stride);
  if((testNumber == 0) || (testNumber == 3))
    test(test_pf_results, test_pf);
  if((testNumber == 0) || (testNumber == 4))
    test(test_pf_Stride_results,
         test_pf_Stride);
  if((testNumber == 0) || (testNumber == 5))
    test(test_pf_Stride_taskq_results,
         test_pf_Stride_taskq);
  if((testNumber == 0) || (testNumber == 6))
    test(test_pfc_results, test_pfc);
  if((testNumber == 0) || (testNumber == 7))
    test(test_pfc_Stride_results,
         test_pfc_Stride);
  if((testNumber == 0) || (testNumber == 8))
    test(test_pfc_Stride_taskq_results,
         test_pfc_Stride_taskq);
  if((testNumber == 0) || (testNumber == 9))
    test(test_pfs_results, test_pfs);
  if((testNumber == 0) || (testNumber ==10))
   test(test_pfcs_results, test_pfcs);
```

Copyright © 2009
QuickThread Programming, LLC
www.quickthreadprogramming.com

```
std::cout << "iRun";
if((testNumber == 0) || (testNumber == 1))
  std::cout << ", st";
if((testNumber == 0) || (testNumber == 2))
  std::cout << ", st_Str";
if((testNumber == 0) || (testNumber == 3))
  std::cout << ", pf";
if((testNumber == 0) || (testNumber == 4))
  std::cout << ", pf_Str";
if((testNumber == 0) || (testNumber == 5))
  std::cout << ", pf_Str_tq";
if((testNumber == 0) || (testNumber == 6))
  std::cout << ", pfc";
if((testNumber == 0) || (testNumber == 7))
  std::cout << ", pfc_Str";
if((testNumber == 0) || (testNumber == 8))
  std::cout << ", pfc_Str_tq";
if((testNumber == 0) || (testNumber == 9))
  std::cout << ", pfs";
if((testNumber == 0) || (testNumber ==10))
  std::cout << ", pfcs";
std::cout << std::endl;
for(int iRun=1; iRun<=numRuns; ++iRun)
{
  std::cout << iRun;
  if((testNumber==0) || (testNumber==1))
    std::cout << ", "
    << test_st_results[iRun];
  if((testNumber==0) || (testNumber==2))
    std::cout << ", "
    << test_st_Stride_results[iRun];
  if((testNumber==0) || (testNumber==3))
    std::cout << ", "
    << test_pf_results[iRun];
  if((testNumber==0) || (testNumber==4))
    std::cout << ", "
    << test_pf_Stride_results[iRun];
  if((testNumber==0) || (testNumber==5))
    std::cout << ", "
    << test_pf_Stride_taskq_results[iRun];
  if((testNumber==0) || (testNumber==6))
    std::cout << ", "
    << test_pfc_results[iRun];
  if((testNumber==0) || (testNumber==7))
    std::cout << ", "
    << test_pfc_Stride_results[iRun];
  if((testNumber==0) || (testNumber==8))
    std::cout << ", "
   << test_pfc_Stride_taskq_results[iRun];
  if((testNumber==0) || (testNumber==9))
    std::cout << ", "
    << test_pfs_results[iRun];
  if((testNumber==0) || (testNumber==10))
    std::cout << ", "
    << test_pfcs_results[iRun];
  std::cout << std::endl;
}
```

```
std::cout << "iRun";
if((testNumber == 0) || (testNumber == 1))
  std::cout << ", st";
if((testNumber == 0) || (testNumber == 2))
  std::cout << ", st_Str";
if((testNumber == 0) || (testNumber == 3))
  std::cout << ", pf";
if((testNumber == 0) || (testNumber == 4))
  std::cout << ", pf_Str";
if((testNumber == 0) || (testNumber == 5))
  std::cout << ", pf_Str_tq";
if((testNumber == 0) || (testNumber == 6))
  std::cout << ", pfc";
if((testNumber == 0) || (testNumber == 7))
  std::cout << ", pfc_Str";
if((testNumber == 0) || (testNumber == 8))
  std::cout << ", pfc_Str_tq";
if((testNumber == 0) || (testNumber == 9))
  std::cout << ", pfs";
if((testNumber == 0) || (testNumber ==10))
  std::cout << ", pfcs";
std::cout << std::endl;
for(int iRun=1; iRun<=numRuns; ++iRun)
{
  std::cout << iRun;
  if((testNumber==0) || (testNumber==1))
    std::cout << ", "
    << test_st_results[iRun];
  if((testNumber==0) || (testNumber==2))
    std::cout << ", "
    << test_st_Stride_results[iRun];
  if((testNumber==0) || (testNumber==3))
    std::cout << ", "
    << test_pf_results[iRun];
  if((testNumber==0) || (testNumber==4))
    std::cout << ", "
    << test_pf_Stride_results[iRun];
  if((testNumbe ==0) || (testNumber==5))
    std::cout << ", "
    << test_pf_Stride_taskq_results[iRun];
  if((testNumber==0) || (testNumber==6))
    std::cout << ", "
    << test_pfc_results[iRun];
  if((testNumber==0) || (testNumber==7))
    std::cout << ", "
    << test_pfc_Stride_results[iRun];
  if((testNumber==0) || (testNumber==8))
    std::cout << ", "
    <<test_pfc_Stride_taskq_results[iRun];
  if((testNumber==0) || (testNumber==9))
    std::cout << ", "
    << test_pfs_results[iRun];
  if((testNumber==0) || (testNumber==10))
    std::cout << ", "
    << test_pfcs_results[iRun];
  std::cout << std::endl;
}
```

```
  std::cout << "Elapse";
  if((testNumber == 0) || (testNumber == 1))
    std::cout << ", " << test_st_results[0];
  if((testNumber == 0) || (testNumber == 2))
    std::cout << ", " <<
    test_st_Stride_results[0];
  if((testNumber == 0) || (testNumber == 3))
    std::cout << ", " << test_pf_results[0];
  if((testNumber == 0) || (testNumber == 4))
    std::cout << ", "
    << test_pf_Stride_results[0];
  if((testNumber == 0) || (testNumber == 5))
    std::cout << ", "
    << test_pf_Stride_taskq_results[0];
  if((testNumber == 0) || (testNumber == 6))
    std::cout << ", "<< test_pfc_results[0];
  if((testNumber == 0) || (testNumber == 7))
    std::cout << ", "
    << test_pfc_Stride_results[0];
  if((testNumber == 0) || (testNumber == 8))
    std::cout << ", "
    << test_pfc_Stride_taskq_results[0];
  if((testNumber == 0) || (testNumber == 9))
    std::cout << ", " <<test_pfs_results[0];
  if((testNumber == 0) || (testNumber ==10))
    std::cout << ", "
    << test_pfcs_results[0];
  std::cout << std::endl;
  std::cout << std::endl;
  return 0;
}
```

```
  std::cout << "Elapse";
  if((testNumber == 0) || (testNumber == 1))
    std::cout << ", " << test_st_results[0];
  if((testNumber == 0) || (testNumber == 2))
    std::cout << ", "
    << test_st_Stride_results[0];
  if((testNumber == 0) || (testNumber == 3))
    std::cout << ", " << test_pf_results[0];
  if((testNumber == 0) || (testNumber == 4))
    std::cout << ", "
    << test_pf_Stride_results[0];
  if((testNumber == 0) || (testNumber == 5))
    std::cout << ", "
    << test_pf_Stride_taskq_results[0];
  if((testNumber == 0) || (testNumber == 6))
    std::cout << ", "<< test_pfc_results[0];
  if((testNumber == 0) || (testNumber == 7))
    std::cout << ", "
    << test_pfc_Stride_results[0];
  if((testNumber == 0) || (testNumber == 8))
    std::cout << ", "
    << test_pfc_Stride_taskq_results[0];
  if((testNumber == 0) || (testNumber == 9))
    std::cout << ", " <<test_pfs_results[0];
  if((testNumber == 0) || (testNumber ==10))
    std::cout << ", "
    <<  test_pfcs_results[0];
  std::cout << std::endl;
  std::cout << std::endl;
  return 0;
}
```