LLNL-PROC-610552

# Improving Application Performance using Hardware Performance Counters

S. H. Langer, W. Boyd

January 16, 2013

Improving Application Performance Using Hardware Performance Counters
Livermore, CA, United States
October 22, 2012 through October 26, 2012

# (U) Improving application performance using hardware performance counters

**[1]Steven Langer, [1,2]William Boyd**
[1]*Lawrence Livermore National Laboratory, Livermore, CA*
[2]*Massachusetts Institute of Technology, Cambridge, MA*

## Abstract

Moore's Law is an empirical observation that the number of transistors on a chip doubles every 18-24 months. The peak floating point performance per chip has been on a similar Moore's Law curve. An increase in performance proportional to the number of transistors is not automatic. Microprocessor designers have relied on a combination of faster transistors, larger caches, increased parallelism within a core, an increasing number of cores per chip, and a variety of other architectural changes to deliver these exponential improvements in performance per chip.

The performance of many applications has lagged behind the improvement in peak CPU performance in recent years. This paper discusses ways to recover some of that lost performance by modifying the source code based on performance measurements. We use pF3D , a code that simulates the interaction between a high intensity laser and a plasma, as a test case. pF3D uses a regular 3D Cartesian grid, so the geometry and memory access patterns are fairly simple to understand.

The first step in improving performance is to measure the wall clock time to see where the application spends most of its time. In the case of pF3D, it typically requires 25 functions to account for 95% of the run time. Once the important routines are identified, performance counter measurements are used to suggest promising code transformations.

The examples in this paper use Tau (http://www.cs.uoregon.edu/Research/tau/home.php) to gather performance data. Open Speed Shop (OSS, http://www.openspeedshop.org/wp/) has functionality similar to Tau. VTune from Intel and IBM's HPC Toolkit (http://researcher.watson.ibm.com/ and select "IBM High Performance Computing Toolkit" from the list of projects) are examples of performance monitoring tools from vendors.

Tau and OSS rely on PAPI to access hardware counters. PAPI (Performance API, http://icl.cs.utk.edu/papi/) provides a universal API to read the hardware registers that record events. Modern processors count the number of times that various hardware events occur. Events include the number of clock cycles, instructions

issued, cache misses, virtual memory address translation, SIMD instructions, and a large variety of other events.

## 1 Introduction

Many developers think of Moore's Law as stating that the performance of a chip will double every 18-24 months. What Moore actually postulated is that the number of transistors on a chip doubles every 18-24 months. An increase in performance proportional to the number of transistors is not automatic. Microprocessor designers have relied on a combination of faster transistors, larger caches, increased parallelism within a core, an increasing number of cores per chip, and a variety architectural changes to deliver these exponential improvements in performance per chip.

It has become increasingly difficult for a code with many physics packages to deliver a high fraction of the peak performance in recent years. Part of the problem is the decrease in the ratio of main memory performance relative to CPU performance. Another part of the problem is the difficulty in exploiting the architectural changes that chip manufacturers have made to allow performance per chip to continue to improve in an era where the clock speed increases (at best) very slowly.

This paper discusses ways to recover some of that lost performance by modifying the source code based on performance measurements. We use pF3D , a code that simulates the interaction between a high intensity laser and a plasma, as a test case. pF3D uses a regular 3D Cartesian grid, so the geometry and memory access patterns are fairly simple to understand.

### Optimizing memory access

One of the most important issues in achieving high application performance is efficient use of the memory system. An Intel Sandy Bridge socket has roughly 31 GB/s of main memory bandwidth (as measured by the Streams benchmark) and a peak double precision floating performance of roughly 333 GFLOP/s. To achieve peak performance, an application must perform roughly 80 floating point operations per 8 byte word fetched from memory. Modern processors have a limited amount of on chip memory (often operated as a cache) which has much higher bandwidth than main memory. Good application performance is possible if most data is read from the cache, not main memory.

Data is transferred between main memory and the cache in units of cache lines. A cache line is typically 64 bytes. The standard optimizations for dealing with memory bandwidth limitations are modifying the code to increase cache hit rates, modifying data layout so that the code uses all the bytes in each cache line that is fetched from main memory, and reducing the number of indirect memory references. The advect-sbs kernel in pF3D (see section 3) is an example of code whose performance depends mostly on memory access patterns.

### Vectorization

Recent microprocessors have wider SIMD (single instruction, multiple data) floating point units. Taking advantage of SIMD units requires either persuading the compiler to vectorize an existing code or rewriting loops using vendor specific vector intrinsics. We have had a fair amount of success obtaining SIMD vectorization using Intel icc if we apply the C99 restrict modifier to our pointer declarations. IBM's xlc will not vectorize the same code. Vectorization with xlc requires proprietary storage alignment declarations. The couple4 kernel in pF3D (see section 4) is an example of code that can obtain a significant speedup from SIMD vectorization.

### Dealing with small memory per core

TLCC systems at the tri-labs have 2 GB per core. Sequoia, an IBM Blue Gene/Q (BGQ) system at LLNL, has 1 GB per core. The BGQ cores execute instructions in order and are much more inclined to "stall" than Xeon or Opteron processors which can execute instructions out of order. IBM provides 4 hardware threads per core to help reduce the impact of these stalls. If a pure MPI code uses all the hardware threads on a BGQ system, there is only 256 MB per process. Some codes need more than 256 MB to fit the executable and its data arrays. Even if there is room for a process per hardware thread, the number of "guard zones" relative to real zones will go up given the small memory per hardware thread. The low memory per hardware thread on Sequoia is expected to be an issue on most future systems.

A hybrid version of pF3D that combines MPI and OpenMP parallelism is under development. Using several OpenMP threads per MPI domain will allow us to have large domains and keep the percentage of guard zones small. An even bigger advantage for pF3D will be that the percentage of memory that must be communicated during light advection will be significantly smaller in the hybrid code than in the pure MPI code.

This paper presents OpenMP scaling studies for three pF3D kernels. The paper focuses on single node performance. There is no explicit discussion of the benefits of reducing the percentage of guard zones. The results are still relevant to those whose codes will not fit on BGQ when using a pure MPI approach because they describe techniques that will be helpful to anyone trying to optimize OpenMP code. One of the key concerns with OpenMP is making sure that the time spent coordinating threads is less than the time spent computing. Our results give some indication of the impact of thread coordination overhead.

To assess OpenMP performance, we use weak scaling studies. In a weak scaling study, the amount of work per thread is held constant as threads are added.

A weak scaling study of this sort does not help to assess the performance tradeoff between threads and processes. When only one thread runs, it has access to the full memory bandwidth and other resources of the node. When there is a thread per core, there will be contention between threads for shared resources. We also present some hybrid scaling studies where we hold the work per core fixed while changing the mix of threads and processes.

## 2 Absorbdt Performance

The absorbdt kernel computes the absorption coefficient for the light propagating through the plasma. The absorption depends on the average nuclear charge of the ions. The original version of this kernel (absorbdt-old) called a function to compute zbar, the average nuclear charge, for each zone. Function call overhead was significant. The next version (absorbdt-loc) uses the C99 inline keyword to request that the zbar function call be inlined. absorbdt-new manually inlines the zbar function call. absorbdt-new-mat4 fully unrolls the loop over materials for the special case of 4 kinds of ions. This final version can be SIMD vectorized by version 13.0 of the Intel C compiler, but not by version 12.1 (which was used to generate this figure). If the compiler performs as requested, absorbdt-loc and absorbdt-new should have similar performance.

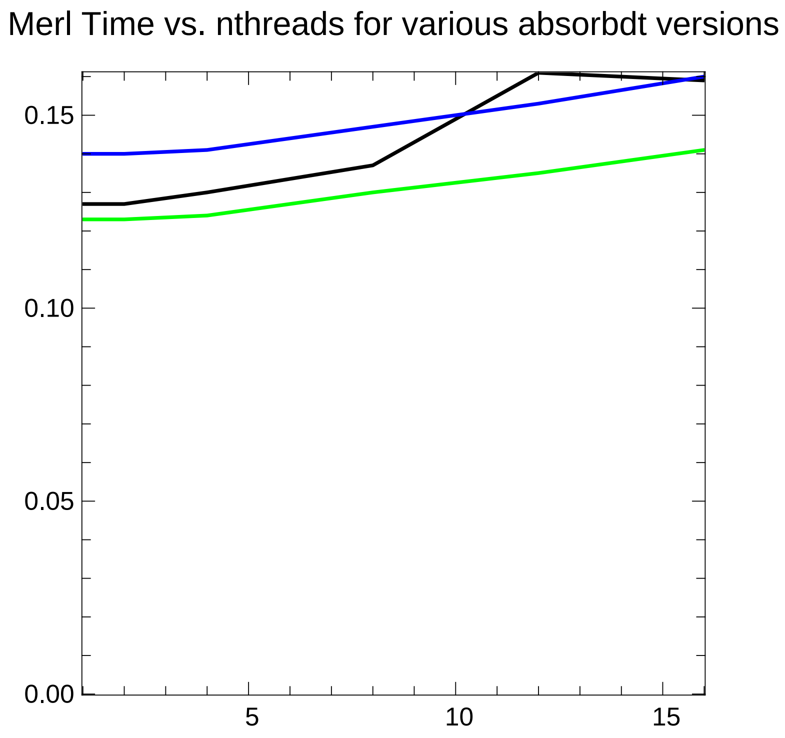

**Figure 1: The run time is shown as a function of the number of threads for three versions of the absorbdt kernel on an Intel Sandy Bridge processor. The black curve is for absorbdt-loc, the green curve is for absorbdt-new, and the blue curve is for absorbdt-new-mat4. The scaling is fairly good for all versions.**

Figure 1 shows the run time as a function of the number of threads for three versions of the absorbdt kernel on an Intel Sandy Bridge processor. The scaling is fairly good for all versions. The original version (absorbdt-loc) has poorer scaling than the other two versions. The difference in performance between absorbdt-loc and absorbdt-new shows that manual and compiler based inlining did not produce the same effect.

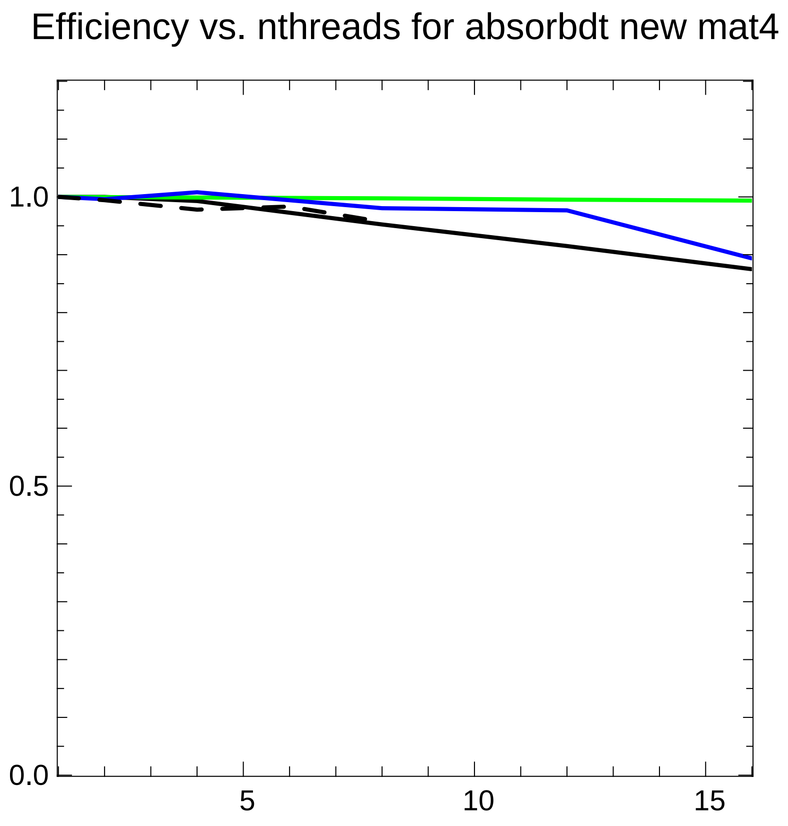

**Figure 2: The efficiency is plotted as a function of the number of threads for the absorbdt-new-mat4 kernel on several processors. The black curve is for an Intel Sandy Bridge, the green curve is for an IBM Blue Gene/Q, the blue curve is for an AMD Barcelona, and the dashed curve is for an Intel Nehalem. The scaling is good for all processors.**

Figure 2 shows the efficiency as a function of the number of threads for the absorbdt-new-mat4 kernel on several processors. The black curve is for an Intel Sandy Bridge, the green curve is for an IBM Blue Gene/Q, the blue curve is for an AMD Barcelona, and the dashed curve is for an Intel Nehalem. The scaling is good for all processors.

## 3 Advect SBS Performance

The SBS advection kernel computes the flux of material across the zone boundary in the x-, y-, or z-direction. It is an example of code whose performance is controlled by memory accesses.

pF3D advects ion acoustic waves between zones as part of the Stimulated Brillouin Scattering (hereafter SBS) package. The velocity is centered on zone faces and the quantities that are advected are located at zone centers. The kernel computes the flux on each face and uses that to update the variable in the neighboring zones. An if test determines if material is flowing into or out of the zone and then uses the variable in the upwind zone to compute the flux.

The update of the array occurs in place. A zone cannot be updated until the flux on both faces has been computed. The different version of this kernel use 1D, 2D, or 3D temporaries to store fluxes until it is ready to do a zone update.

Advection of material in the y-and z-directions uses 2D temporary arrays and has stride one in the innermost loop.

advect-x-old advects material in the x-direction with the two innermost loops over y and z. It has stride nx access and only uses one float complex value (8 bytes) per 64 byte cache line fetched from main memory.

advect-x is a variant of advect-x-old that updates tiles that are 64 zones in y by 4 zones in z for all values of x, then moves to the next tile. As a result of this pattern, it uses entire cache lines. This version performed well in the Pentium 4 era, but is not optimal in the multi-core era.

advect-x-new uses a 3D temporary array and has stride one memory access. It performs better than the other versions, but requires significantly more memory.

advect-x-base calculates the flux at each zone boundary twice whereas all other kernels compute the flux once and store it in a temporary array. advect-x-base uses stride one for all memory accesses.

Figure 3 shows the performance of the different versions of the advect kernel on a Sandy Bridge processor. The time increases as the number of OpenMP threads increases. This means that there is contention for hardware resources or thread coordination overhead is significant.Tests show there is little thread coordination cost in advect-x-new for 12 or fewer threads on a Sandy Bridge. Contention for memory or L3 cache bandwidth is the most likely explanation for the drop in performance with more than 8 threads.

Figure 4 shows the performance of the different versions of the advect-x kernel on a Blue Gene/Q processor. advect-x and advect-x-new both have almost perfect scaling. It appears that prefetching works well and that there is enough memory bandwidth to prevent contention. The time increases as the number of OpenMP threads increases for advect-x-base and advect-x-old. The most likely explanation is inadequate memory bandwidth, perhaps due to inefficient prefetching.

Figure 5 shows the efficiency (the throughput per core relative to a one core run) of the advect-x-new kernel versus the number of threads. The scaling on the Blue Gene/Q is excellent. The loss in efficiency with increasing numbers of threads is similar for all the x86_64 processors.

Figure 6 shows the number of L3 cache misses for various versions of the advect-x kernel as a function of the number of threads on an Intel Sandy Bridge processor. The number of L3 misses for advect-x-new and

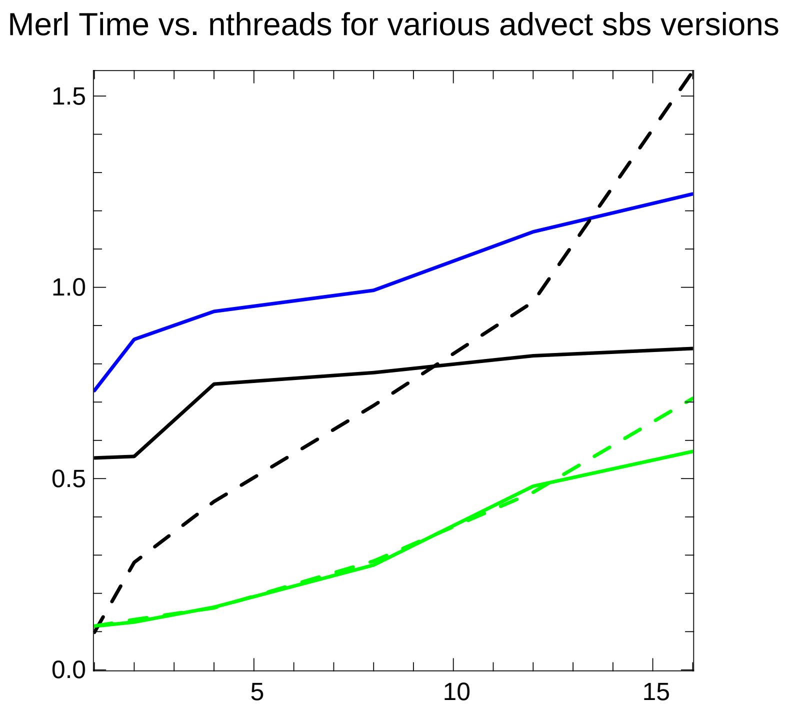

**Figure 3: The run time for various versions of the advect-x kernel is shown as a function of the number of threads on an Intel Sandy Bridge processor. The black curve is advect-x, the green curve is advect-x-new, the blue curve is advect-x-old, and the dashed curve is advect-x-base. In the absence of contention for hardware resources and thread coordination operations, the time would be the same for any number of threads.**

advect-x are nearly independent of the number of threads. The number of L3 misses for advect-x-old and advect-x-base increases as the number of threads increases. advect-x has roughly 3 times as many misses as advect-x-new. A naive analysis suggests that advect-x will need to fetch each cache line 8 times while advect-x-new will only need to fetch it once. Intel processors have hardware prefetching. If prefetching works well, many cache lines will be fetched from main memory without generating an L3 miss. The data suggests, but does not prove, that the 3X difference in cache misses is due to the need to fetch each line many times in advect-x.

## 4 Couple4 Performance

pF3D solves for the complex amplitudes of the laser light wave, the SRS wave and the SBS wave. There are two independent polarization for each of these waves. pF3D also solves for the complex amplitude of the ion acoustic wave and the electron plasma wave. The couple4 kernel computes the coupling (energy

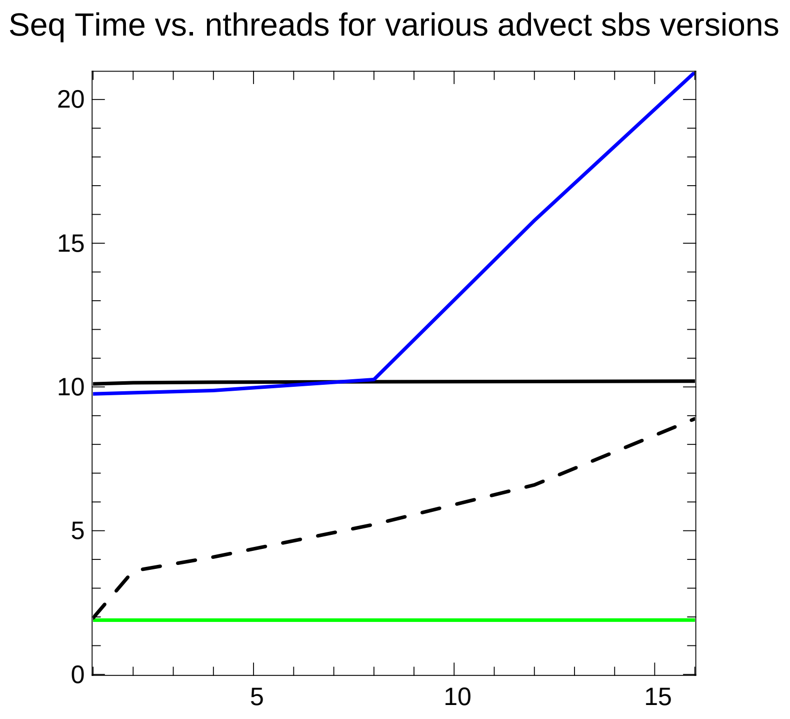

**Figure 4: The run time for various versions of the advect-x kernel is shown as a function of the number of threads on an IBM Blue Gene/Q processor. The black curve is advect-x, the green curve is advect-x-new, the blue curve is advect-x-old, and the dashed curve is advect-x-base. In the absence of contention for hardware resources and thread coordination operations, the time would be the same for any number of threads.**

transfer) between these waves.

The couple4 kernel has fairly high computational intensity (operations per word of memory fetched), making it an interesting target for SIMD vectorization.

The waves have complex amplitudes, so there is a lot of complex arithmetic in couple4. Version 12.1 and before of the Intel C compiler was not able to vectorize couple4.

A new version of this kernel, couple4-nocomp, was written where all complex arithmetic was turned into real arithmetic by hand. couple4 has a three-dimensional loop. The real and imaginary parts of the amplitudes are gathered into temporary vectors before entering the innermost loop. All operations then occur on stride one real arrays and the compiler vectorizes the loop.

Version 13.0 of the Intel C compiler can vectorize the complex arithmetic. couple4 then runs faster than couple4-nocomp, presumably because the copies between the complex arrays and the real temporaries

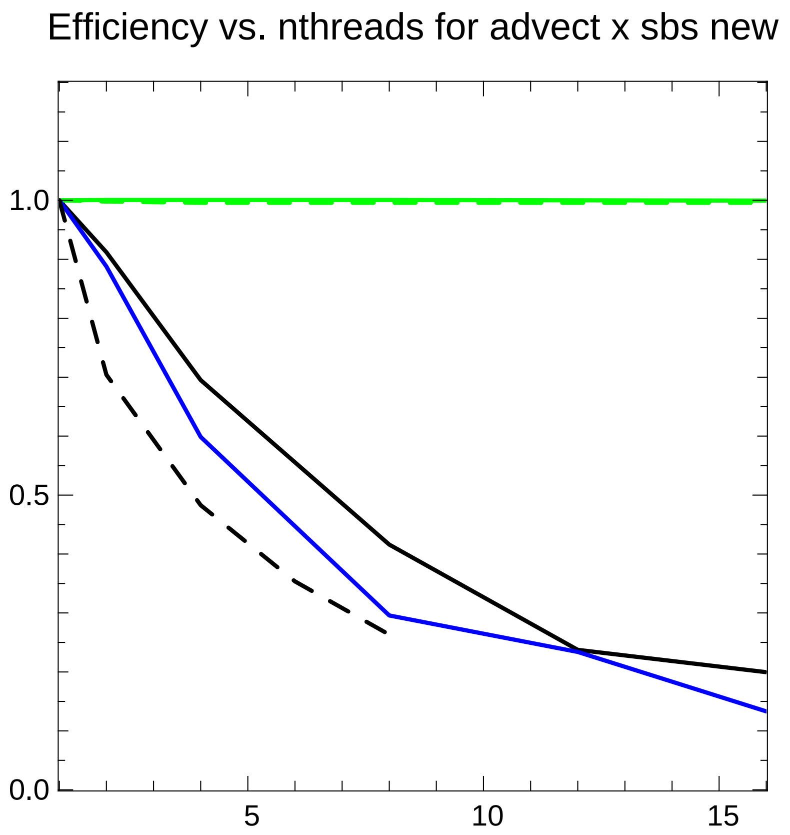

**Figure 5: The efficiency (the throughput per core relative to a one core run) of the advect-x-new kernel is plotted versus the number of threads. Black is an Intel Sandy Bridge processor, green is an IBM Blue Gene/Q processor, blue is an Intel Nehalem processor, and the dashed line is an AMD Barcelona processor. The scaling on the Blue Gene/Q is excellent. The loss in efficiency with increasing numbers of threads is similar for all the x86_64 processors.**

consumes significant amounts of time.

TAU can record the number and type of floating point instructions issued during the execution of couple4. These counts are helpful in understanding the speedup due to vectorization.

Figure 7 shows the GFLOP/s/core as a function of the number of threads for the couple4 kernel. The number of floating point operations was measured using TAU. The event counts showed that 80% of the operations were performed with float precision and 20% were performed with double precision.

The dashed black curve is the only one where the compiler performed SIMD vectorization. The Sandy Bridge processor has 256 bit registers so the maximum speedup that can be achieved through SIMD vectorization is 8X in float precision. The observed SIMD speedup is 2.8X for one thread.

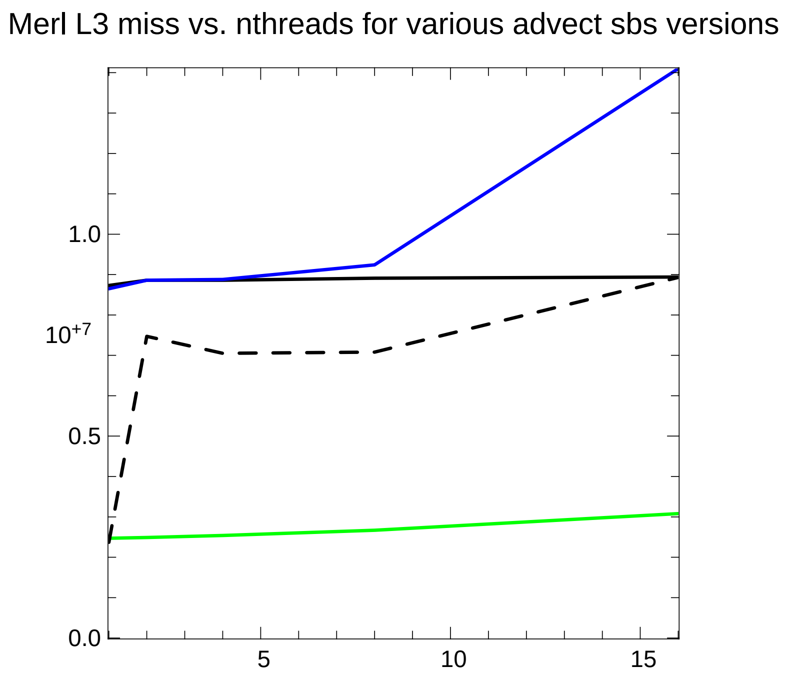

**Figure 6: The number of L3 cache misses for various versions of the advect-x kernel is shown as a function of the number of threads on an Intel Sandy Bridge processor. The black curve is advect-x, the green curve is advect-x-new, the blue curve is advect-x-old, and the dashed curve is advect-x-base. Only advect-x-new has low miss rates for 1-16 threads.**

The TAU operation counts revealed an interesting result. The Intel compiler performs most complex arithmetic using SIMD registers in unvectorized code. In the case of the solid black curve, 85% of the floating point operations were performed using ”packed SIMD” operations. The unvectorized code had roughly a 2X speedup from the use of SIMD registers. That means that the vectorization in the dashed black curve delivered a 2.8X speedup out of a theoretically possible speedup of 4X.

couple4 gets a strong speedup from SIMD vectorization with 8 or fewer threads. There is enough cache bandwidth to support effective use of the AVX SIMD unit for this kernel.

The performance of the vectorized couple4 drops considerably when using more than 8 threads. The OpenMP version of couple4 allocates all arrays on a single socket, so the resulting code can only use half the memory bandwidth of the node. A version of pF3D combining MPI and OpenMP parallelism is under development. When pF3D runs on a Sandy Bridge node, it will always have at least one MPI process per socket. That means only the OpenMP performance up to 8 threads is of interest and SIMD vectorization still gives a 2.5X speedup at that point.

The IBM compiler used on the Blue Gene/Q is currently unable to perform SIMD vectorization of any of the pf3d kernels. The BGQ performance does not reflect any benefit from the 4-wide SIMD unit unless the IBM compiler uses it for complex arithmetic. TAU is now available for the BGQ and we plan to investigate this question.

Figure 8 shows the efficiency of the couple4-nocomp kernel as a function of the number of threads for several processors. The scaling is excellent for the BGQ processor. The x86_64 processors all show a noticeable drop off in efficiency. Efficiency is not the only consideration - the Sandy Bridge is faster for all thread counts than the Barcelona, even though its drop in efficiency is greater.

## 5 Conclusions and Future Work

We have examined the performance of three different kernels extracted from pF3D. The couple4 kernel is compute intensive and benefits from SIMD vectorization. TAU gathered hardware event data for floating point operations executed by couple4. The counts showed that a non-SIMD version of the kernel gained significant speedup from the use of SSE2 registers for complex arithmetic. This helped explain why the kernel did not deliver an 8X speedup when it was vectorized for AVX registers.

The advect-x kernel has low computational intensity. Event counts gathered by TAU demonstrate that advect-x-new has lower L3 cache miss rates than the other versions of this kernel on an Intel Sandy Bridge. All versions of this kernel are memory bandwidth limited, but advect-x-new puts less pressure on the memory system than the other versions and, as a result, performs better.

All versions of the absorbdt kernel have similar performance and TAU event counts are similar for all versions.

Hardware event counts have helped us gain insight into the performance of the pF3D kernels and allowed us to perform simple optimizations that resulted in significant speedups. Similar techniques can be used in attempts to improve the performance of other codes.

## Acknowledgements

This paper has been released as LLNL-PROC-610552. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC.

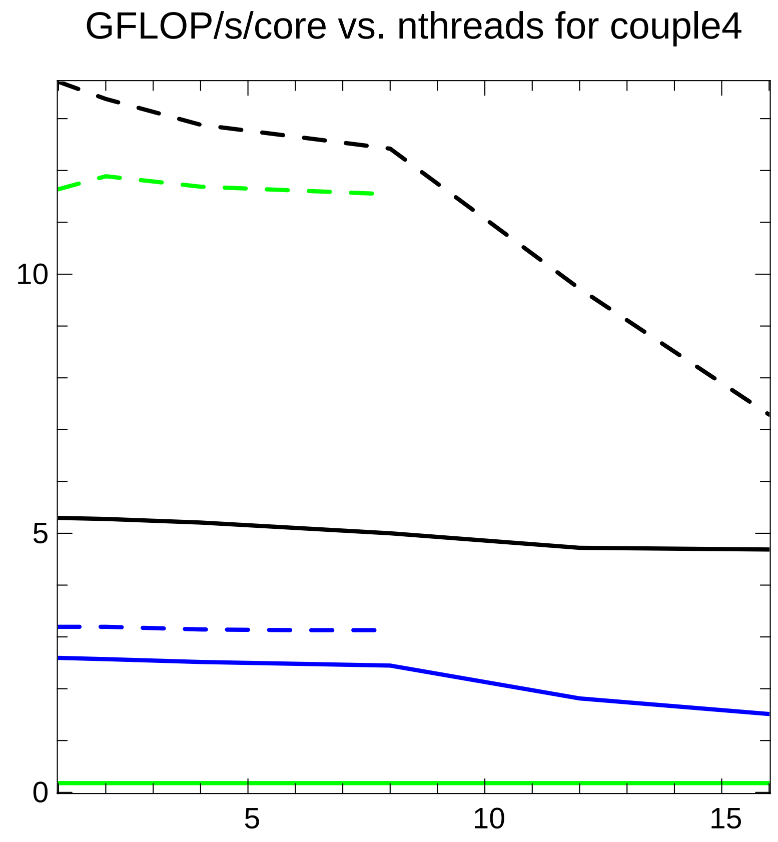

**Figure 7: The GFLOP/s/core is shown as a function of the number of threads for the couple4 kernel on several different processors. The solid black curve is an Intel Sandy Bridge using icc 12.1. The dashed black curve is a Sandy Bridge using icc 13.0. The dashed green curve is a Sandy Bridge using icc 13.0 running multiple OpenMP processes under MPI so that all cores are used in all cases. The solid green curve is an IBM Blue Gene/Q with one thread per core. The solid blue curve is an AMD Barcelona. The dashed blue curve is an Intel Nehalem.**

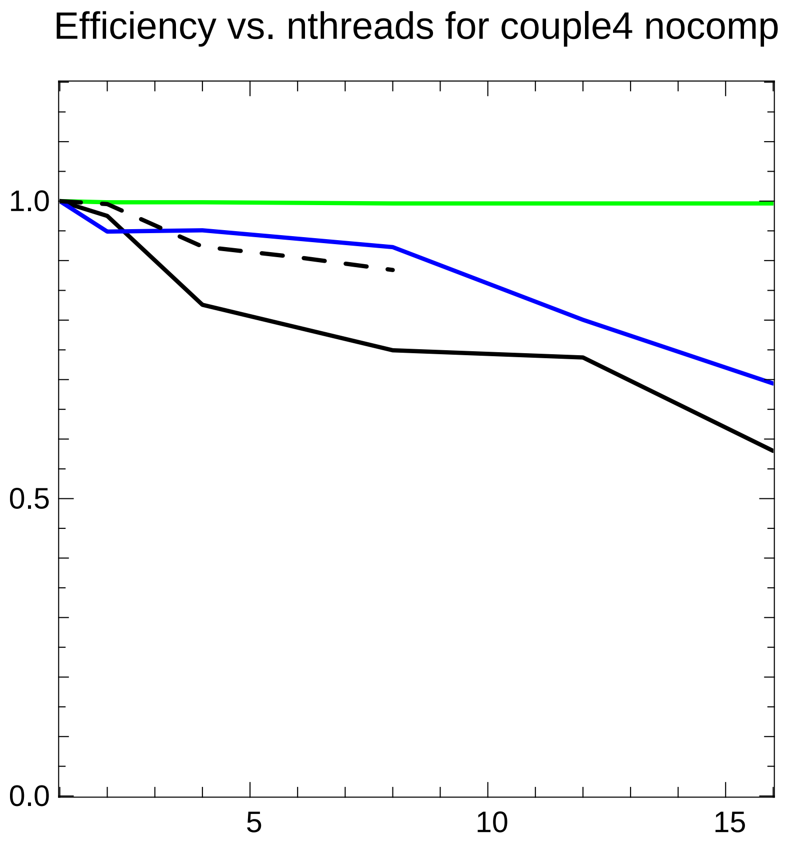

**Figure 8: The normalized efficiency of the couple4-nocomp kernel is shown as a function of the number of threads for several processors. Black is an Intel Sandy Bridge, green is an IBM Blue Gene/Q, blue is an AMD Barcelona, and the dashed curve is an Intel Nehalem. The scaling is excellent for the BGQ processor. The x86_64 processors all show a noticeable drop off in efficiency.**