Dare to be first.

UNIVERSITY OF DELAWARE

Mills Performance Testing: Threading Efficiency

Dr. Jeffrey Frey, IT-NSS / University of Delaware

frey@udel.edu

Background

The AMD "Interlagos" processors used in the Mills cluster are a relatively new design and have been neither widely tested nor widely integrated into compiler optimization repertoires. This implies that many numerical libraries are also not ready to run most efficiently on the modular "Bulldozer" architecture that AMD has adopted in these processors. Each "Bulldozer" module consists of two cores[1] which share a 256–bit floating–point execution unit (FPU); according to AMD, the FPU also functions as two distinct 128–bit FPUs. The instruction set is augmented with instructions that target the 256–bit mode and additional "fused multiply–add" functionality designed to leverage the shared FPU. In theory, the presence of the AVX and FMA4 instructions offsets the penalties associated with two cores' sharing an FPU.

Over the last several months many Mills users have reported less than optimal scaling of their code across the 24 (or 48) cores available in a node. This agrees with reports made by other HPC entities that stock[2] floating–point intensive code scales poorly on "Interlagos". In this document we will present the results of our own tests on Mills and use them to inform some basic recommendations with respect to optimizing the execution of your numerical workload on the cluster.

Methods

To test the FPU we have adopted the simple methodology used by NAG in their baseline tests of "Interlagos" systems:

- OpenMP parallelism
- dgemm(): double–precision general matrix multiplication
- Intel 2011 compiler suite (64 bit), MKL 10.3.8[3]

In addition, the Portland 12.4 compiler with the AMD ACML 5.0.0 library (OpenMP–parallelized, FMA4–enabled variant) was tested to establish (what should be) best performance relative to a compiler/library that is not "Interlagos" aware.

[1] Some have debated the accuracy of the term "core" due to the resource sharing involved in a module. Integer resources are duplicated but the FPU is shared, so it may be more accurate to label the design as a hybrid of classical *multi-core* and *multi-thread* processor architectures.

[2] Stock meaning that the code has not been recompiled by a system that optimizes for the "Bulldozer" architecture.

[3] Matlab makes use of the Intel MKL for the majority of its BLAS/LAPACK workload, so the tests should be indicative of what one can expect from that commercial product.

Since the "Bulldozer" module is designed with dedicated integer resources per core, we also tested an OpenMP–parallelized integer–typed variant of dgemm() named igemm(). The outer–most loop of the general matrix multiplication algorithm was marked for static, chunked parallelization by the compiler. Several compilers were used, including the Open64 compiler since it should theoretically produce the best possible machine code for the "Interlagos" processor at this time.

Reported runtimes are averaged across ten sequential invocations of the program in question. Timings represent the elapsed wall and CPU time surrounding just the computational routine being tested[4]. OpenMP thread counts were tested from 1 through 24, with each test being run as the sole task on a 24–core "Interlagos" compute node to avoid any resource contention (other than contention between OpenMP threads). SMP performance is measured in two ways:

1. The efficiency of the code (ratio of CPU time to wall time in excess of 100%).

2. The speedup produced by multiple threads (ratio of parallel wall time to serial wall time).

General matrix multiplication is an embarrassingly parallel algorithm so the efficiency is expected to scale somewhat perfectly (slope of 1) across thread counts; in practice, efficiency is also in part a test of a compiler's ability to produce object code that has optimal interaction with the processor architecture.

Efficiency compares "active" CPU time to elapsed real time, so significant negative degradation from the theoretical behavior implies threads spent time in a sleep state when they were expected to be running.

Speedup is the metric which is most desirable to scale per thread count. Ideally, for an embarrassingly parallel algorithm the efficiency and speedup would track exactly with thread count: for 24 threads, the CPU time should be 24 times the wall time and the wall time should be 1/24th the serial wall time. This is never the case, primarily due to increasing resource contention between the threads (e.g. shared caches, overlapping memory fetches). While contention cannot be eliminated, it can be mitigated by intelligent scheduling of instructions at compile time and run time.

Hardware

All 24 thread counts were tested on a node using the default BIOS configuration as supplied by Penguin Computing. In particular, the Supermicro motherboards have features that control the AMD "Turbo Core" functionality:

- HPC Mode: engage full Turbo Core frequency scaling on load (versus reacting progressively as load increases)

- Maximum P-state: frequency–scaling can be capped at states 0 through 4 (4 being lowest)

As delivered, the BIOS has HPC mode enabled with a cap at state P0. Thus, the full range of power states can be used for Turbo Core frequency scaling and scaling is delivered en masse on load. Mills nodes comprise dual AMD 6234 "Interlagos" processors at 12 cores (12C) each and a base frequency of 2.4 GHz.

Comparative testing of Magny–Cours was performed on a Penguin–built system containing a closely–related Supermicro chipset and dual AMD 6134 processors at 8 cores (8C) each and 2.3 GHz.

[4] An external, C-compiled timer function using the `gettimeofday()` and `getrusage()` functions was utilized for the sake of providing uniform timing across all tests. See **Listing 2** in Appendix 1.

Floating Point (dgemm)

To begin, a simple Fortran 90 program that allocates three (randomized) 4000 x 4000 double–precision floating–point matrices and multiplies them using `dgemm()` was compiled with the Intel 2011 `ifort` compiler. The `–mkl` flag was used to link–in the default Intel MKL library that accompanies the Intel 2011 suite. Special care must be exercised with the MKL with respect to OpenMP parallelism: by default, the library will attempt to use "dynamic" worker thread allocation that does not adhere to the limit imposed by the `OMP_NUM_THREADS` environment variable. The `OMP_DYNAMIC=FALSE` and `MKL_DYNAMIC=FALSE` environment variables are used to override this behavior.

It is safe to assume that the Intel 2011 compiler suite should not be necessarily well–suited to producing optimal object code for the AMD "Interlagos" processor architecture. Likewise, the MKL is also likely to be out–of–step with this platform. However, the prodigious number of commercial and academic software packages that are built atop the Intel runtime libraries necessitate our testing of it. Among the compilers present on Mills (Intel, Portland, and Open64), we might expect it to offer the worst–case performance profile.

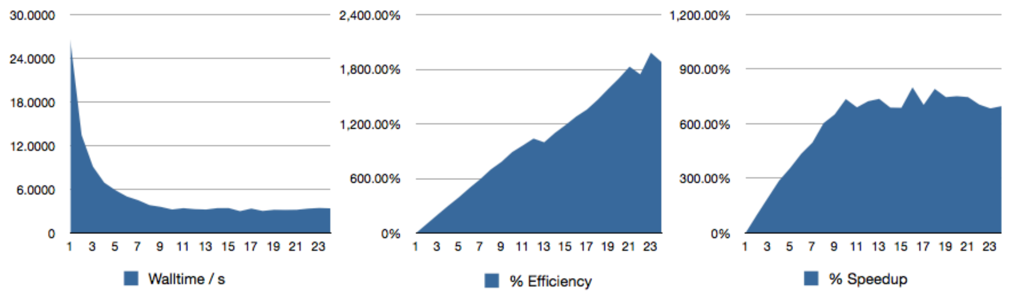

Figure 1: Average "Interlagos" performance data for Intel 2011 and MKL.

Note the "bump" in the efficiency graph that occurs as the test transitions from 12 threads (half complement) to 13 threads, yielding two semi–linear regimes. Compare this to the speedup: in the first efficiency regime, the speedup also scales as the number of threads increases. However, in the second regime the speedup has flattened and fluctuates around the speedup produced at the 12–core level. The wall time is extremely flat in this region, implying that the addition of OpenMP threads is adding no performance benefit to the algorithm relative to running with just 12 threads.

Processor Affinity

Historically, one system–level optimization that was performed in order to increase the efficiency of HPC code was binding a program to a specific subset of the host's processor cores. With a *processor affinity* specified, the OS would not move the program thread(s) back and forth across the full complement of cores. This would generally increase throughput by minimizing the repeated eviction and refetching of program data in the caches. Linux allows one to specify affinity for a process by means of a processor mask:

```
taskset 0x00000041 ls -l
```

For a 24–core Mills node, the hexadecimal mask specifies that cores 0 and 1 on socket 1 should be used[5].

Using the Intel–compiled binary and two OpenMP threads, tests were made with different affinity strategies:

Strategy	Wall time / s	% Worse
Same “Bulldozer” Module	21.6876	60.0%
Same Socket	14.0181	3.4%
Unique Socket	13.5535	

The unique socket strategy is taken as the baseline since it should avoid the majority of resource contention. By comparison, running both threads on the same socket will have the threads competing for memory bandwidth; the performance hit is minimal, though. Binding the threads to the same “Bulldozer” module, on the other hand, produces contention between the threads for the module’s FPU resources. In fact, the nearly 22 second runtime for two threads is almost equal to the 26 second average runtime for a single thread. The baseline runtime (two threads across both sockets) is comparable to the two-thread average runtime (Figure 1) observed *without* explicitly setting processor affinity. So behind the scenes, Linux is doing a good job of distributing computational threads across the system and (in general) modifying a process’s affinity is unnecessary and potentially harmful.

Portland Compiler Suite

If the Intel 2011 compiler and MKL represents a “worst case” scenario, then it follows that tools that explicitly support the “Interlagos” architecture or AMD processors in general should offer better performance for the `dgemm()` example. The Portland Group 2012 (12.4) compiler suite was also used to compile the Fortran 90 source; the `–mp=numa` flag was used to link–in the PGI OpenMP runtime. Portland has a long–standing record of good support for AMD processors, and the 2012 suite includes support for the “Interlagos” processor. Likewise, the AMD Core Math Library (ACML) should be a better match for the AMD architecture than the Intel MKL: the ACML 5.0.0 library optimized for OpenMP parallelism was employed, both with and without support for the FMA4 additions to the ISA.

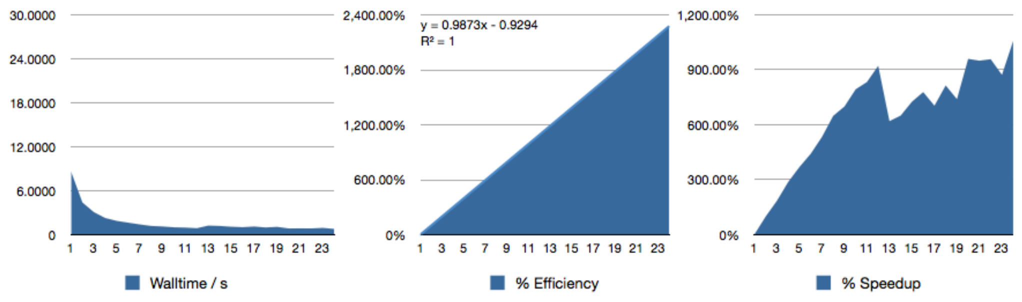

Figure 2: Average “Interlagos” performance for PGI 2012 (12.4) and ACML 5.0.0 (with FMA4)

The vertical axis ranges have been kept the same as for the Intel 2011 compiler suite to enable easy visual comparison of the results. The wall time has decreased an average of 69% relative to the Intel variant (ranging from 62% up to 78%).

[5] The bitmask for a dual–socket “Interlagos” system is four sequential sets of 6 bits (high to low order, left to right): [222222][222222][11111**1**][11111**1**]. The highlighted bits represent the two cores of a “Bulldozer” module.

The efficiency scales very well, with the slope of the trend line in Figure 2 very near 1. At 24 threads the efficiency is only 29% below a perfect 2300%, or just over 1% loss of efficiency per core. The speedup behavior is qualitatively similar to that which was observed for the Intel variant: a sharp drop in performance when 12 threads is exceeded. While the data does seem to trend toward a recovery of performance over the 13 to 24 thread range (versus the flatness in this range for the Intel variant), the behavior is non–uniform and barely exceeds a 1000% speedup for 24 threads (versus just over 900% at 12 threads).

The performance improvement with Portland/ACML is impressive: what role does the FMA4 functionality of "Interlagos" play in the increase? The program was relinked against the non-FMA4 ACML 5.0.0 library to answer this question. The *decrease in wall time* was graphed for each thread count.

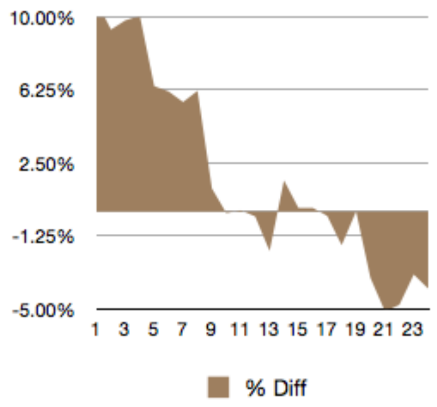

Figure 3: Percent ***decrease*** *in wall time using the non-FMA4 ACML 5.0.0 library versus the FMA4-enabled version.*

Running from 1 to 12 threads, the FMA4 instructions go from dramatically **decreasing** the wall time to having no effect. Once again at the transition to 13 threads the behavior changes radically: in general, above 12 threads the FMA4 instructions negatively affect the wall time.

From these data alone, the Portland compiler and ACML library appear to be a good choice on Mills. The observed thread dependence for the FMA4–enabled variant of ACML 5.0.0 would imply that care must be exercised to match the ACML library to the runtime environment: when running jobs with more than 12 threads, the non–FMA4 library may be preferable.

Open64 Compiler

With the Open64 compiler and the 5.1.0 version of the ACML (with FMA4) the timing results defy explanation: the parallel `dgemm()` routine always yields the same wall time for a run, but the CPU time scales by number of threads. This uniquely strange behavior was also evident when using the non–FMA4 ACML library — though the wall time did suffer in the absence of FMA4 instructions.

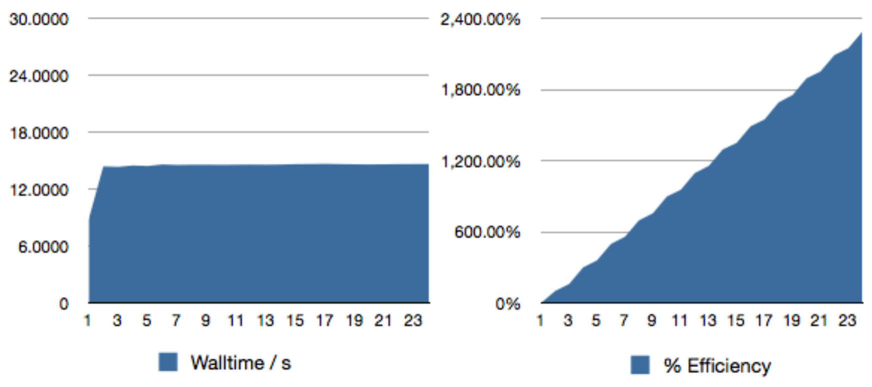

Figure 4: Performance under the Open64 compiler and ACML 5.1.0.

The figure shows that the best performance for the Open64/ACML 5.1.0 variant came from the serial run. Every parallel run saw a wall time of approximately 14 seconds and CPU time of 14 times the thread count.

Some discussion was found online of the ACML 5.1.0 library's negative performance gains relative to the later versions of the 4.x series of the library. Among the affected routines mentioned was `dgemm()`. One forum containing such complaints was AMD's own developer forum: no "answer" was ever added by AMD staff who participated in the conversation. With a lack of a release newer than 5.1.0, the path toward better performance with Open64/ACML is apparently to move back to the 4.4.0 release of the library[6]. Indeed, repeating the `dgemm()` timing with ACML 4.4.0 for Open64 yielded behavior inline with expectation (though the Portland/ACML combination still easily beat the wall times).

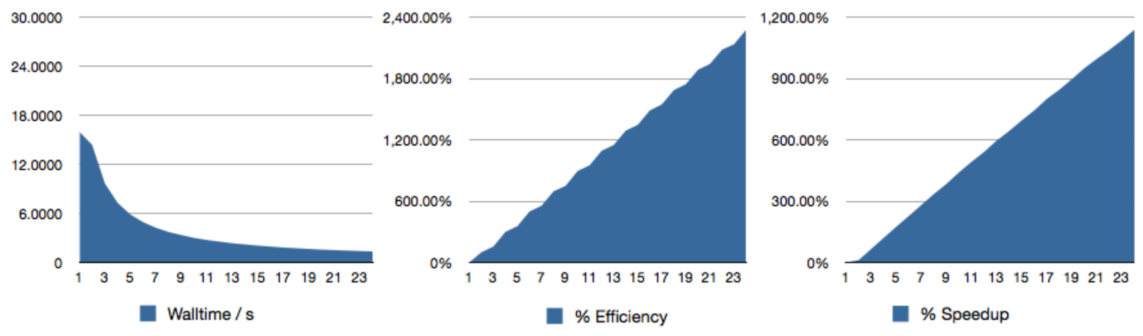

Figure 5: Performance under the Open64 compiler and ACML 4.4.0

Magny–Cours

To determine how the "Interlagos" processor compares to the previous generation "Magny–Cours" processor, the `dgemm()` example was moved to another cluster where it was built using Portland 10.9 and ACML 4.3.0. Once again the `-mp=numa` flag was used to link–in the OpenMP runtime.

[6] Even the 5.0.0 release — which performs very well under the Portland compiler — produced the constant wall time profile a'la ACML 5.1.0 for Open64.

At lower thread counts the “Interlagos” processor does show shorter wall times than its older “Magny–Cours” relative. Unfortunately, the usual transition at 13 threads is where its advantage disappears. The speedup graph shows the “Magny–Cours” parallel performance scaling better across all 16 of its cores than the “Interlagos” does beyond its 12–core half complement. Taken together, the “Bulldozer” design definitely starves floating–point intensive code of FPU resources relative to the previous generation of the Opteron processor, even where an optimizing compiler and native numerical library are employed.

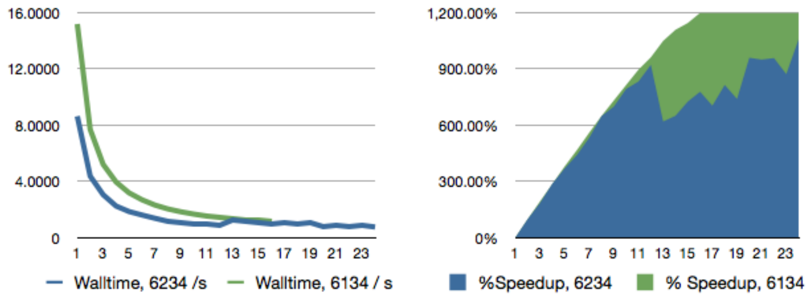

Figure 6: Performance on “Magny–Cours” (green) versus “Interlagos” (blue).

Multiple Jobs Per Node

The results thus far apply to a node that is running *only* a single job: how is performance affected when multiple OpenMP jobs are run concurrently? To answer this question, several tests were made employing two instances of the Intel 2011/MKL variant run concurrently with thread counts summing to 24. The wall time for each is compared against the runtime for that thread count when run solo on the node:

Thread Counts	Wall times / s	% Longer wall time
12 + 12	6.456, 6.524	100.14%, 102.24%
8 + 16	7.319, 6.413	93.71%, 117.32%
4 + 20	9.377, 4.133	36.48%, 32.48%
8 + 8 + 8	11.9180, 12.6020, 13.3310	202.64%, 220.01%, 238.52%
6 + 6 + 6 + 6	16.3540, 16.6960, 17.6450, 18.5980	231.39%, 238.32%, 257.55%, 276.86%
12 + 12 (per-socket affinity)	4.048, 4.095	25.49%, 26.95%

Note that (for the dual job tests) as the difference between the two thread counts increases the effect on the lower thread–count run in the pair is expected to decrease since its counterpart will finish more quickly than it and relinquish all resources to the slower job. For a host with 24 cores one would typically expect two 12–thread jobs to contend for memory bandwidth and have some additional latency due to cache mismatches. A correlation between the jobs’ taking *twice* as long when run concurrently and the “Bulldozer” module’s having *two cores* share an FPU seems apt.

The situation is actually *worse* for the Portland/ACML variant:

Thread Counts	Wall times / s	% Longer wall time
12 + 12	7.022, 7.026	702.20%, 702.60%
6 + 6 + 6 + 6	14.994, 15.004, 15.012, 15.018	839.06%, 839.69%, 840.19%, 840.56%
12 + 12 (per-socket affinity)	1.388, 1.402	64.28%, 65.94%

Whereas the Intel/MKL variant had two processes ineffectually filling the shared FPU pipelines (and thus, were already spending large amounts of time waiting for FPU pipelines to clear), the more optimal instruction scheduling in the Portland/ACML variant was disrupted to an enormous degree by the presence of multiple processes.

For both tests, the performance degradation can be mitigated to a great extent by setting processor affinities to bind each 12–thread process to a specific socket (see Listing 1); the affinity result is highlighted in blue in the tables above.

Magny–Cours Comparison

Running two 8–thread jobs on the 16 core Magny–Cours test system (no processor affinity specified) produced wall times that were longer by only 5.67% relative to a solo 8–thread job. Running four 4–thread jobs concurrently, the runtime was lengthened an average of just 3.25%.

Listing 1: A script that runs two concurrent instances of a program, binding them to socket 0 and 1, respectively.

```
#!/bin/bash

export OMP_DYMANIC=FALSE
export MKL_DYNAMIC=FALSE

export OMP_NUM_THREADS=12
taskset 0x000fff ./matmul_test &
taskset 0xfff000 ./matmul_test &
wait
```

Conclusions

For floating–point intensive code, the following recommendations are distilled from the tests cited above:

1. When possible, recompile your code with the Portland compiler suite and use the ACML library.
2. Avoid multiple jobs per node: submit your OpenMP ACML jobs using the `-pe threads 24` argument to Grid Engine.
3. Since you will be using all 24 cores, some additional performance should be realized by linking against the ACML library that lacks FMA4 instructions.

Item 2 may seem like a waste of resources, but consider running four (4) Portland/ACML `dgemm()` tests with a single node available. Running 24 cores per job, the four runs would happen sequentially. The expected total wall time would be around 3 seconds (4 runs, 0.7456 seconds per run). Figuring that there is no perceived benefit to using more than 12 cores for these jobs, they are run instead with `-pe threads 12` and are scheduled two at a time on the node. Given the dual–job wall time cited above, the total wall time would be around 14 seconds (2 runs of 2 jobs each, wall time 7 seconds per pair) — or 4.7 times longer. Becoming slightly more sophisticated, processor affinity is applied to each pair, producing a total wall time of around 3 seconds (2 runs of 2 jobs each, wall time 1.4 seconds per pair). This is about the same as running the four jobs with 24 cores each. So to augment the above recommendation:

4. To effectively run 12 thread jobs at two concurrent per node you must bind each process to a single socket using the `taskset` command.

Integer Performance (igemm)

Not all workloads on Mills may be floating–point, so a series of similar tests were performed using an integer–oriented variant on `dgemm()`. The `igemm()` C function multiplies two integer matrices and places the results in another. No special attention is given to overflow conditions. Like Fortran, the matrices are column–major; the outer loop over columns of the result matrix is marked for OpenMP static, block parallelization.

Please note: In no way was the `igemm()` function rigorously optimized. No quantitative performance comparisons should be made between it and the previously presented data for the `dgemm()` function. For this reason, plots of the wall time will not be presented. Quantitative comparisons will be made between variants of `igemm()`, though.

As a baseline for comparison among compilers, the native GCC 4.4.5 compiler on the cluster was used to build the program (with the `-O3 -fopenmp` options). GCC is usually considered to be the least–optimal choice of compiler when speed/performance is desired.

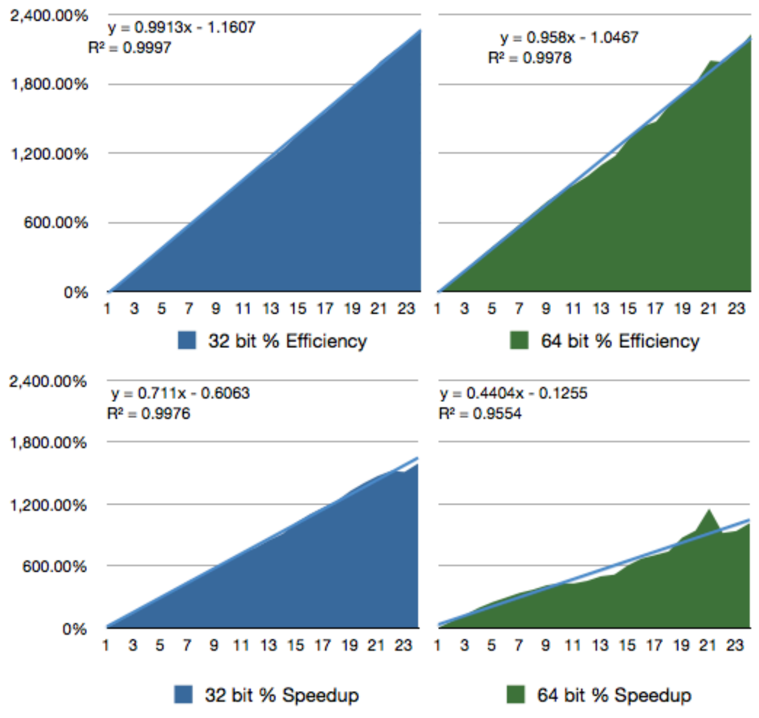

Figure 7: Efficiency and speedup of igemm() compiled with GCC.

For both the 32– and 64–bit variant the efficiency of the threaded code scales well. The speedup scales linearly across the entire range and does not show the same transitioning behavior when 13 threads is reached. The negative deviation from unit slope is likely due to memory latency (cache misses) since the code is not fully optimized for performance. For the same matrix size the 64-bit variant must move double the number of bytes between cache and memory, so it should be expected to have longer wall time (on average around 50% longer than the 32–bit variant). The speedup suffered more for the 64–bit variant since each additional thread triggers cache misses that cost approximately twice as much latency as the 32–bit variant.

Portland Compiler

The program was also compiled using the Portland Group C compiler (`pgcc`) from the same 2012 (12.4) suite. Optimizations were made at the generic `–O4` level and OpenMP parallelism was enabled using the `–mp=numa` flag. The `numa` parameter requests that `pgcc` link–in an OpenMP runtime library that automatically sets OS processor affinities to prevent threads from migrating between processors.

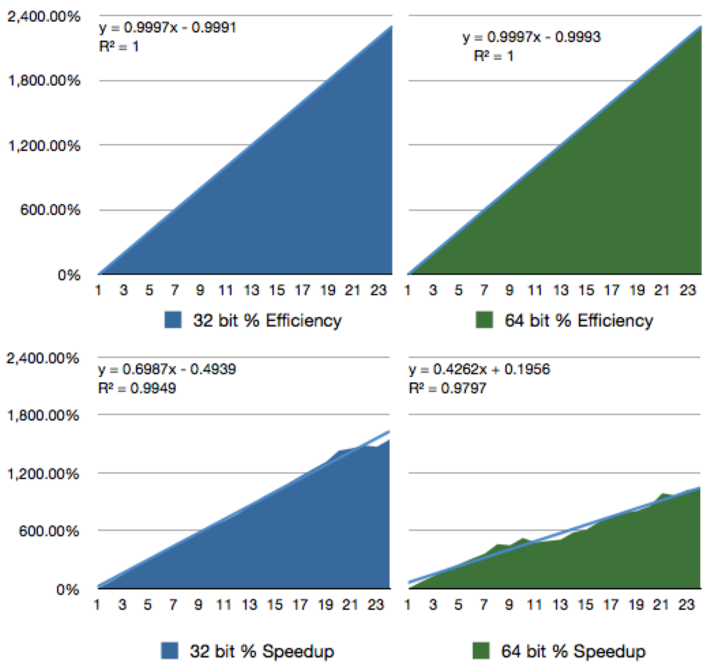

Figure 8: Efficiency and speedup of igemm() compiled with the Portland compiler.

As in the `dgemm()` tests, the Portland C compiler produces near–perfect efficiency in the parallel code it generates. By comparison to GCC, though, the speedup of the code as it scales up is not significantly better. In fact, in terms of wall time the GCC versions marginally edged–out their Portland counterparts (see Fig. 9).

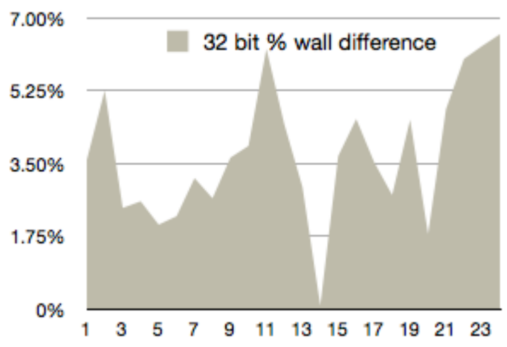

Figure 9: Portland wall time as a percentage above that of GCC.

Open64 Compiler

AMD officially recommends the Open64 compiler suite as a high–performance, production quality compiler for their hardware. Its pedigree descends from the Silicon Graphics MIPSPro compiler that originally helped drive SGI's presence in the HPC market, and actually makes its way to the University of Delaware for some of its later development work! The AMD site prominently mentions features like multicore scalability optimizations and code analysis optimizations. The latest version does include the capability for FMA4 (`-mfma4`) and AVX (`-mavx`) optimizations.

The `igemm()` code was compiled using `opencc` with the `-O3 -mp` flags. The resulting executable produced efficiency and speedup results inline with the GCC compiler. The wall times for Open64 were radically decreased relative to GCC (and thus, Portland as well) with a full 20 seconds shaved off the serial run time of GCC (24 seconds off the Portland run time).

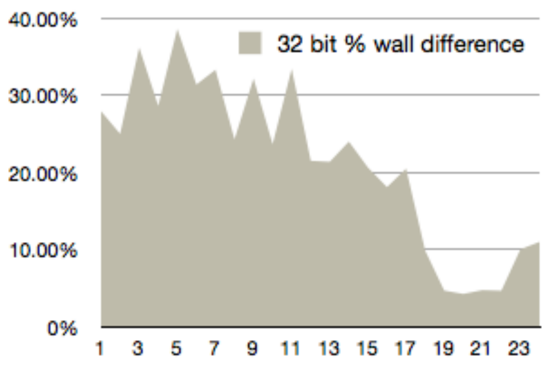

Figure 10: GCC wall time as a percentage above that of Open64.

Conclusions

In all cases for integer–centric work loads the efficiency and speedup scaled uniformly across the full range of OpenMP threads. This is in stark contrast to the floating–point speedups which invariably showed two performance regimes situated on either side of the half–complement of 12 cores. However, this behavior is to be expected: relative to the changes to floating–point, integer behavior is not quite so different on "Interlagos" as on previous versions of the Opteron processor.

Of the compilers tested, the Open64 C compiler is the clear winner. Much like employing the AMD Core Math Library had the greatest positive effect on the `dgemm()` tests, for C code using AMD's choice compiler provided an enormous performance boost to the program without the manual addition of any explicit code.

Appendix 1: Code Listings

Listing 2: Fortran-callable `timer()` subroutine.

```
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
#include <sys/resource.h>

void
timer_()
{
  static struct rusage    __RUSAGE_START;
  static struct timeval   __WALL_START;
  static int              __TIMER_CALLED = 0;

  if ( ! __TIMER_CALLED ) {
    getrusage(RUSAGE_SELF, &__RUSAGE_START);
    gettimeofday(&__WALL_START, NULL);
    __TIMER_CALLED = 1;
  } else {
    struct rusage       __RUSAGE_END;
    struct timeval      __WALL_END;
    double              dt_cpu, dt_wall;

    getrusage(RUSAGE_SELF, &__RUSAGE_END);
    gettimeofday(&__WALL_END, NULL);

    dt_wall = (__WALL_END.tv_sec - __WALL_START.tv_sec) +
              (__WALL_END.tv_usec - __WALL_START.tv_usec) * 1e-6;
    dt_cpu =  (__RUSAGE_END.ru_utime.tv_sec - __RUSAGE_START.ru_utime.tv_sec) +
              (__RUSAGE_END.ru_utime.tv_usec - __RUSAGE_START.ru_utime.tv_usec) * 1e-6 +
              (__RUSAGE_END.ru_stime.tv_sec - __RUSAGE_START.ru_stime.tv_sec) +
              (__RUSAGE_END.ru_stime.tv_usec - __RUSAGE_START.ru_stime.tv_usec) * 1e-6;
    printf(
        "CPU,WALL,%%EFFICIENCY =%10.3lf%10.3lf%10.1lf\n",
        dt_cpu,
        dt_wall,
        100.0 * (dt_cpu / dt_wall - 1.0)
      );
    __RUSAGE_START = __RUSAGE_END;
    __WALL_START = __WALL_END;
  }
}
```

Appendix 2: Grid Engine and Processor Affinity

Grid Engine is capable of tracking and assigning Linux processor affinities to jobs. The `-binding` flag to the `qsub` command enables this feature for a job and also determines in what capacity Grid Engine will implement the affinity. There are three binding methods:

- **set**: Grid Engine executes the job using the `taskset` command and an affinity mask it assigns
- **env**: the `SGE_BINDING` environment variable will be present in the job's environment
- **pe**: bindings are made per–node and are listed in the fourth column of the `pe_hostfile`

The **set** method is the default and the simplest since it requires no changes to the job script. In the latter two methods it is up to the job itself to apply affinity limits to its environment. The `man` page for `qsub` contains a full description of the `-binding` flag, including discussion of the "binding strategy" parameter that dictates how core selection should be made across a node's processors.

Internally, Grid Engine maintains a static complex attribute named `m_topology` that represents the layout of a node's processor cores using a string of upper–case characters: **S**ocket, **C**ore, and **T**hread: a 24–core node on Mills would be represented as

```
SCTTCTTCTTCTTCTTCTTSCTTCTTCTTCTTCTTCTT
```

This string shows that the "Bulldozer" modules in an Interlagos processor actually look like single cores with two dispatch threads and will be treated as such by Grid Engine's affinity logic. The dynamic complex attribute named `m_topology_inuse` contains the same string but any components that have been allocated to a job by means of the `-binding` flag are lower–cased, as in

```
ScttcttCTTCTTCTTCTTSCTTCTTCTTCTTCTTCTT
```

for a job that used the `qsub` flags

```
-pe threads 4 -binding set striding:2:1
```

Notice that the binding logic does not descend to the processor thread level, but instead allocates four threads associated with two neighboring cores. This establishes two simple "rules" for using this Grid Engine functionality with OpenMP/threaded jobs on Mills:

1. Choose an even number of threads for your job (call it *N*).
2. For the core count in the binding strategy use *N/2* and a stride of 1.

Appendix 3: Open64 Thread Binding

The Linux facilities for core binding were discussed earlier in this document. The `taskset` command can be used to request that the OS restrict the subset of processor cores on which the threads of a process may execute. This technique is often used in HPC to promote better cache efficiency.

For the Non–Uniform Memory Access (NUMA) memory model used by AMD Opteron processors:

- Each processor socket "owns" and controls access to a fraction of the total RAM
- A processor incurs the lowest latency when accessing its own RAM
- A processor incurs greater latency when accessing another processor's RAM

These factors exacerbate the problem of locality: whereas core binding may help to increase cache efficiency, it typically does nothing to ensure that the data entering/leaving that cache are in RAM that is local to the processor containing those cores. For the utmost efficiency, one might like to exercise fine–grain control over both the core and memory localization for a job.

The Open64 OpenMP runtime declares two additional environment variables that control NUMA affinity for a job. The feature is enabled by setting `O64_OMP_AFFINITY` to `TRUE` in the environment. As usual, `OMP_NUM_THREADS` provides the maximum number of worker threads. For *N* worker threads, the variable `O64_OMP_AFFINITY_MAP` must be set to a list of *N* core numbers separated by commas. In a 24–core node the core numbers would range from zero through 23; the NUMA layout of the node can be displayed by logging–in to a compute node and using the `numactl --hardware` command (see Listing 3). The 24–core node is divided into four NUMA nodes. Nodes 0 and 1 encompass cores on the first physical CPU and nodes 2 and 3 encompass the cores on the second physical CPU. Under the "Bulldozer" architecture, the first six cores ("cpus" in the output of `numactl`) are the left[7] core in each two–core module. Thus, to run a 6 thread OpenMP job with heightened NUMA locality, one might choose to bind the threads and memory to the "cpus" of NUMA node 0 by setting `O64_OMP_AFFINITY_MAP` to `0,1,2,3,4,5` in the environment. Note that if there are fewer core numbers in `O64_OMP_AFFINITY_MAP` than the thread count in `OMP_NUM_THREADS`, the number of cores listed in `O64_OMP_AFFINITY_MAP` will act as the limit.

The `numactl` command can also be used to exercise such fine–grain control over the execution of any command. Consult the `man` page for more details on its usage in that capacity. Listing 4 shows that a process with a "preference" for NUMA node 0 makes exclusive use of memory associated with that node. Executing the same program with an allocation greater than the free RAM on node 0 would lead eventually to usage of another NUMA node for mapped pages: hence, a "preference" is not a strict policy that precludes other NUMA nodes. The `numactl` command also allows a program to be strictly bound to specific NUMA nodes for core, memory, or both core and memory localization. Listing 5 shows a program which allocates and accesses 20 GB of memory and illustrates that the memory pages it accesses are stringently corralled on NUMA node 0.

The `numactl` command could be used with the **env** Grid Engine binding method discussed in Appendix 2 to affect both core and memory binding for OpenMP/threaded jobs.

[7] An arbitrary designation based on the AMD illustrations of "Bulldozer" components.

Listing 3: Output of the `numactl --hardware` command.

```
[user@n011 ~]$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5
node 0 size: 16382 MB
node 0 free: 15426 MB
node 1 cpus: 6 7 8 9 10 11
node 1 size: 16384 MB
node 1 free: 15688 MB
node 2 cpus: 12 13 14 15 16 17
node 2 size: 16384 MB
node 2 free: 15704 MB
node 3 cpus: 18 19 20 21 22 23
node 3 size: 16384 MB
node 3 free: 15632 MB
node distances:
node   0   1   2   3
  0:  10  16  16  16
  1:  16  10  16  16
  2:  16  16  10  16
  3:  16  16  16  10
```

Listing 4: Using `numactl` to localize memory access. Notice that nodes 1, 2, and 3 remain the same as in Listing 3. The maximum difference between Listing 3 and 4 for node 0 is approximately 10000 MB (exactly what test-mem allocated) which corresponds to the full range being active in RAM and not paged to disk, etc.

```
[user@n011 ~]$ numactl --preferred=0 ./test-mem
Allocating 140734520624056 bytes of memory...accessing for 5 minutes...
^Z
[user@n011 ~]$ bg
  :
[frey@n011 ~]$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5
node 0 size: 16382 MB
node 0 free: 5406 MB
node 1 cpus: 6 7 8 9 10 11
node 1 size: 16384 MB
node 1 free: 15690 MB
node 2 cpus: 12 13 14 15 16 17
node 2 size: 16384 MB
node 2 free: 15704 MB
node 3 cpus: 18 19 20 21 22 23
node 3 size: 16384 MB
node 3 free: 15633 MB
  :
```

Listing 5: Using node "binding" rather than "preference" with `numactl` forces the program to ONLY map memory pages to the memory associated with node 0. If the program allocates more than 16 GB of memory, then the OS will eventually need to swap pages in and out of NUMA node 0 (versus moving to another node under the "preference" mode).

```
[user@n011 ~]$ numactl --cpunodebind=0 --membind=0 ./test-mem
Allocating 140734520624056 bytes of memory...accessing for 5 minutes...
^Z
[user@n011 ~]$ bg
  :
[frey@n011 ~]$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5
node 0 size: 16382 MB
node 0 free: 110 MB
node 1 cpus: 6 7 8 9 10 11
node 1 size: 16384 MB
node 1 free: 15692 MB
node 2 cpus: 12 13 14 15 16 17
node 2 size: 16384 MB
node 2 free: 15708 MB
node 3 cpus: 18 19 20 21 22 23
node 3 size: 16384 MB
node 3 free: 15636 MB
  :
```