SANDIA REPORT

SAND2011-9432
Unlimited Release
Printed December 2011

Optimization of CPAPR for x64 multicore

Benjamin Allan

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/ordering.htm

SAND2011-9432
Unlimited Release
Printed December 2011

Optimization of CPAPR for x64 multicore

Benjamin Allan
Sandia National Laboratories,
P. O. Box 969, Livermore CA 94551
{baallan@sandia.gov}

Abstract

I report the progress to date of my work on scaling the CPAPR algorithm and necessary supporting code to enable processing large (gigabyte to 100 gigabyte) data sets and benchmarking the same. Where possible, I also report background information possibly of relevance in future modifications of the code. The results include: minor repairs and additions to the TTB library for portability, algorithmic improvements relevant to both serial and multithreaded implementations, algorithmic improvements taking advantage of multithreading hardware, support library additions (binary IO routines) needed for efficiently and reproducibly benchmarking the algorithms. For this optimization work, no large scale data sets are available. Therefore, scalability of data synthesis algorithms is addressed as well.

Acknowledgment

This work was supported by the Sandia LDRD program. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Contents

1 Methodology and strategy 7
2 Code Development Results 9
2.1 Code Correctness and attendant build considerations 9
2.2 Serial algorithm changes 10
2.3 Code threading 12
3 Performance Results 17
3.1 CPAPR computation 17
3.2 Data generation 25
3.3 IO 26
4 Drivers 27
5 Conclusions and recommendations 29
5.1 Recommendations 29
References 31

Figures

1 Overall timing and breakdown on small and midsized cubic tensors 18
2 Large cube on 16-32 threads, stepping by 8. 19
3 Lopsided tensor on 8-32 threads, stepping by 4. 20
4 Pi performance on small and midsized cubes 21
5 Pi performance on a lopsided tensor, 1.3E7 nonzeros 22
6 Phi performance on small and midsized cubes 23
7 Phi performance on a lopsided tensor 24
8 Generation 25

Tables

1 Summary of largest data synthesis (32 core), 10000^3, Rank 50 26

This page intentionally left blank

1 Methodology and strategy

I translated the CPAPR algorithm [1] from Matlab code and applied the basic iterative procedure for any optimization project starting from serial code:

1. Verify (in this case in a limited way) baseline code correctness and portability
2. Profile a reasonable case to verify performance bottlenecks
3. Examine bottlenecks for coding practices that defeat compiler-based optimizations and repair them if needed
4. Examine bottlenecks for appropriate choice of algorithms/data structures and replace if needed
5. Test various threading tactics impact on performance while verifying correctness
6. Ensure reproducibility of input, output, and performance at scale

Based on recent experiences with MPI, OpenMP , auto-parallelization with Intel and PGI compilers, Intel Threaded building blocks, and a Sandia threadpool library, I selected OpenMP as the threading system for this optimization project. For this application, the choice of OpenMP in no way sacrifices performance relative to the other models listed; the differences are primarily syntactic. As the goal of the project is to create a library which is scalable in data set size, portable, compiler independent, hardware independent, and maintainable, OpenMP is the best standards-based choice. The inner loop arithmetic of CPAPR is simple addition and multiplication, with some indirect addressing. As such, we expect serial performance to be bounded by memory bandwidth and multicore performance to be bounded by coordination costs.

This page intentionally left blank

2 Code Development Results

2.1 Code Correctness and attendant build considerations

Testing revealed a few uninitialized memory uses; these were all patched. One such patch revealed a logic problem in handling zero values down stream that was referred to Ballard and Kolda for resolution. Several places in the code rely on the defined type *ttb_indx* to be an unsigned type, and a productized header should note that changing to signed type may invalidate the code. The sort function in Sptensor relies on arithmetic that could fail on large but reasonable data sets if the *ttb_indx* is 4 instead of 8 bytes. This should also be noted in productized headers.

Building across 4 compilers (Intel, PGI, 2 gcc versions) identified non-portable usages which have been fixed or are noted here. For all the checks listed in the rest of this section, checks are performed in select code areas, not comprehensively. A productized version of the library would apply these checks comprehensively across the source code, and then turn them on only for debugging and regression test builds.

Checks on file operations have been added in several IO routines; these checks should prevent possible hangs in IO code loops, a common problem when processing large data sets.

The generation of weighted random subscripts (coordinates) in the Sptensor is based on a weak C pseudorandom number generator, rand(). Use of this function was discovered to lead to cycling of the subscripts when generating data sets with more than 20 million elements. The most portable and maintainable replacement for this function is to use a Mersenne twister based algorithm from the boost::random package. Use of the boost generator is enabled by building the library with -D_HAVE_BOOST . Use of this flag option is recommended for all builds. In addition, when boost is not available, a periodic bump of the random number generator is applied to reduce the probability of cycling. Random tensor generation can be time consuming at large scale, so standard thread-safe versions of rand() are used when generation is done in parallel.

Support for detecting the appearance of IEEE infinity and NaN values in Ktensors has been added; use of this feature is enabled by compiling the library with the -D_TTB_CK_FINITE flag. One customer reported that due to using an old, non-ISO-compliant compiler, the standard C++ library functions for identifying non-finite numbers are not available on his platform; in deference to this customer, the code continues to use GNU C's finite() function.

Support for array bounds checking within Sptensors has been added; use of this feature is enabled by compiling the library with -D_TTB_USE_BOUND_CHECKS=1 .

Support for parallel quick sorting of generated Sptensors is implemented with OpenMP 3.0 task directives; use of this feature is enabled by compiling with -D_TTB_OMP_TASK which signifies that the compiler is known to have a reliable implementation of the task

directive. It appears there is no compile time way of detecting which OpenMP features are available and reliable. Literature review suggests OpenMP 2.0 syntax (known as nested parallel regions) is ineffective for parallelizing sorting of random data. The performance of the parallel quicksort implementation is optionally affected by another flag, -D_TTB_SP_SORT_MAGIC=k where k is a strictly positive integer; the meaning of this flag is discussed later.

Support for debugging parallel sorting and compression of randomly generated Sptensors is enabled by compiling the library with -D_TTB_DEBUG_SORT .

Support for multi-architecture binary IO has not been implemented; binary data files are platform specific. On little-endian machines, the data will be little-endian and on big-endian machines the data will be big-endian. Moving binary data files from x86 to PowerPC architectures is not supported. The binary data file format includes endian check data that are verified before additional data are read; misuse of a foreign formatted binary file will be detected and an error issued. The binary format includes data enabling a precision check; an error is issued if data is not of the precision expected. A productized version of the code might use some binary data portability library. Checksum verification is unimplemented for both binary and text data files; application level checksums need to be added to ensure reliable processing with large data of high consequence, as even self-checking server storage systems have been shown to have a high rate ($> 5\%$ of drives) of undetected bit errors.

None of the code tested in this project depends on portable BLAS or LAPACK interfaces, so the selection of these libraries made during build has no impact on performance.

2.2 Serial algorithm changes

Selective support for vectorizing fine-grained operations has been added by applying the C restrict keyword in relevant locations. Unfortunately, C++ compiler support of this C standard is very uneven (varying syntax, varying performance improvements), so we define the macro RESTRICT in a new header, TTB_portability.h based on the compiler detected with preprocessor directives. Currently restrict is used if PGI, Intel, or GNU compilers are detected; if the library is ported to other compilers, TTB_portability.h should be updated. If a supported compiler is not detected, the RESTRICT macro becomes empty and the keyword disappears from the code during preprocessing.

The initial scheme for weighted generation of random tensors depended on a linear search of an array of cumulative sums with the length of a tensor dimension. For large dimensions, this search dominates the generation time. I replaced it with a binary search, making the time for generating unsorted, nonunique tensors essentially negligible.

Loop unrolling and vectorization in the Sptensor cp_apr_pi operation are defeated by the object-oriented code organization (by function calls in the inner loop). Inlining is not a reliable way to obtain performance in this case. The most efficient of the parallelized variants

eliminates these function calls by applying knowledge of the library data structures. The original inefficient version is retained and is the default mode of computation when threading is not used. The optimization friendly inner loop rewrites could be made the default for non-threaded code; if this is done, the original version should be retained for regression checks because the optimized version depends on storage layout choices made in Sptensor, FacMatrix and other classes.

2.2.1 A quick overview of key memory traffic patterns in array and tensor based codes

Kolda, Ballard, and Chi chose data structures algorithms with the intent to enable efficient inner loops and cache data motion for key steps in CPAPR. To aid understanding the descriptions of algorithms given further on, we define a few key terms and data motion patterns. We do not aim to be comprehensive or tutorial here.

- A *streaming calculation* is the sequential flow of a contiguous chunk of the array through a calculation, with the same operations being applied to each data element and no dependence of the output to one element on the data in any other element.
- A *streaming read or write* is the sequential flow of a contiguous chunk of elements from one hardware unit (cache, processor, storage) into another.
- A *gather* is when a large number of elements (at random or stenciled locations) contribute to a computation but are not overwritten; also known as a reduction.
- A *scatter* is when a large number of elements (at random or stenciled locations) are overwritten with the result of a computation.
- A *random read or write* is arbitrarily ordered access to array elements; it is generally associated with indirect addressing schemes and gather or scatter operations.
- A *cache miss* is when data is needed which is not in the current level of cache being queried for the data.
- *Hardware locality* is the degree of closeness of bulk memory hardware to the processor accessing it. In the multi-socket machines (e.g. Dell T7500, SGI Ultraviolet, and most of the recent Sandia cluster nodes), memory directly attached to the processor socket (local memory) is faster to access than remote memory attached to another socket on the same board. As applications allocate and use memory, placement of the data can affect bandwidth available for streaming operations. Control of memory placement is not well standardized, so functions needed to manage placement well are non-portable and complex.

Streaming operations are the simplest to parallelize, but most interesting algorithms are a combination of streaming and gather/scatter. With care, most reductions can be parallelized, but locking or intermediate result caching is always required. Gathers that require

access outside cache induce significant delays in the absence of expensive arithmetic that overlaps the memory traffic. Sparse tensor arithmetic invariably requires indirect addressing for some subset of operations, depending on the storage scheme selected. Sorting a sparse tensor may reduce delays due to cache misses in some algorithms. When multiple threads used to decompose a single streaming task are also using the same cache, the result is a shortened cache residence time. If there is no data reuse, this shortened residence time has little impact on efficiency. Scatters from multiple threads require locking or some other method to eliminate race conditions and data corruption resulting from conflicting writes.

2.3 Code threading

I applied and studied threading support in three high-level functions of Sptensor: cp_apr_pi , cp_apr_phi , and quicksort. These present an interesting fraction of the spectrum of the challenges inherent in threading and optimizing numerical code on x86-based processors. Fortunately for the functions in question, the code is compact and we have little reason to be concerned with the performance of the processor instruction cache system unless the code is ported to a processor with little or no instruction cache. The serial throughput of the code depends on the amount of arithmetic needed per tensor element, the available bandwidth of the read and write operations, and the access patterns of the read and write operations that drive cache behavior. The parallel throughput of the code depends additionally on the assignment of calculations to threads and the choice of algorithms and data structures used to eliminate race conditions.

2.3.1 A quick overview of threading as used in this work

The interested reader can find numerous tutorial and reference resources for thread programming online. Here we describe briefly the three standardized threading patterns we applied, so that those unfamiliar with OpenMP can examine and understand the source code.

The first two patterns apply to simple loops. Consider a loop of length L where each trip performs some roughly fixed amount of work. When assigning the work of this loop to a fixed number of threads, P, there are two common work patterns. These are identified in the OpenMP standard as 'static scheduling' and 'dynamic scheduling'. Static scheduling divides the work into P contiguous chunks within the range [0,L) sized as evenly as possible. Dynamic scheduling dispatches work in fixed size chunks no bigger than L to each thread as the thread becomes available. Dynamic scheduling is typically preferred when the cache architecture of the machine is such that for a large or small L, static chunks of size L/P result in unfavorable cache behavior and lowered throughput. OpenMP 2.0 and later support both these work patterns. Code written following these pattern as expressed in OpenMP is portable across all the compilers we tested and is reported to be portable to Microsoft compilers.

The third pattern handles arbitrary workloads. Consider an algorithm where there is no fixed work quantity associated with inner iterations or recursions. This is typical of certain graph algorithms and most sorting algorithms. These workloads can be threaded by splitting at iteration or recursion points and processing the resulting subtasks with a task queue. Identification of effective task splitting methods is heuristic and algorithm dependent. Manually implementing a task queue and guaranteeing its correctness and termination is nontrivial. Fortunately, OpenMP 3.0 and later provide a simple way to assign tasks to code regions.

Should OpenMP prove unacceptable as an implementation choice for bulk threading on some future platform, the threading directives in the code give clear documentation of the information and control flows which must be maintained in any alternative threading implementation. Folklore exists which suggests OpenMP implementations manage threads inefficiently, creating and destroying threads each time a parallel region is entered. I verified that this is not true for our usage of the current OpenMP implementations tested; all the implementations create a pool of threads on entry to the first parallel region and schedule or idle them as needed.

Enabling a CMake based build with OpenMP requires compiler-specific flags, as listed by compiler here. These flags should be set in the environment when CMake is run. A variety of examples are provided in TTB_cpp/example_builds/.* of the source code.

- GCC: CFLAGS=-fopenmp; CXXFLAGS=-fopenmp; LDFLAGS="-fopenmp -lgomp"
- Intel: CFLAGS=-openmp; CXXFLAGS=-openmp
- PGI: CFLAGS=-mp; CXXFLAGS=-mp

For Intel and PGI compilers, the same flags used for compiling objects must be used for linking executables.

The effects of hardware locality on threaded performance are often noticeable, as the data for a thread may be almost entirely in memory non-local to the processor. Ideal memory placement for one step of a complex algorithm may be very non-ideal for another even though they use the same object classes. Most Linux kernels apply a first-touch principle in assigning virtual memory pages to hardware. It is difficult to guarantee the effectiveness of this strategy without reengineering the code to make separate allocations for each thread, as the Linux kernel is free to migrate pages as needed. The widely available program wrapper *numactl* provides a limited ability to guide Linux kernel page assignment policy, but it is not suitable for a general application-oriented user.

For good performance on large numeric data sets, the default Linux page size of 4 kilobytes is widely regarded as far too small, sometimes entailing performance reductions of 50% and more. Threaded programs can be particularly sensitive to this. However, making use of larger page sizes (2MB or 1GB) requires system administration support and is not a

viable option for most users. I did not yet run any experiments to determine performance sensitivity to page size.

2.3.2 The Pi calculation

The cp_apr_pi algorithm applied to an Nd dimensional sparse tensor X computes an update of Nd-1 components of Pi. Pi is an array proportional in size to the number of nonzeros in X and the update is to scale an element of Pi by the corresponding current values in the approximant M being computed. This amounts in memory traffic terms to a gather from a random read of M and a streaming read/write of Pi. The read location in M is determined from the subscripts of the nonzeros in X, thus sorting X may somewhat improve the cache performance of this algorithm.

The Pi calculation algorithm is parallelized by splitting the work into contiguous chunks of non-zeros from X. Several parallelizations of the algorithm are coded, and at present the scheduling directive used is *runtime* which in the absence of user intervention defaults to "dynamic,1" for GNU OpenMP and vendor dependent policies on other compilers. Performance behavior with varying dynamic schedules is easily tested by setting the environment variable OMP_SCHEDULE =dynamic,chunksize where chunksize is a strictly positive integer. When productized, this directive should be converted to static or a large dynamic chunk size, as the default behavior with GCC performs very poorly.

The first parallelization applies a simple *omp parallel for* directive to the serial version of the code. This version performs unscalably, as the threads all compete for write access to a temporary (*subs*) used to query the indices of elements of X. In the absence of OpenMP , this parallelization degenerates to the serial version of the code. Using the parallel version of this can be tested by compiling with OpenMP enabled and the flag -D_TTB_OMP_CPAPR_PI=kpi where kpi is 1.

The second parallelization eliminates the conflicting write and locking of *subs*. Using this version can be tested by compiling with OpenMP enabled and the flag -D_TTB_OMP_CPAPR_PI=kpi where kpi is 2.

The third parallelization extends the second and eliminates barriers to instruction level parallelism by applying knowledge of the data structure of X and of M. This version is the default if OpenMP is used.

Even if OpenMP is enabled, passing -D_TTB_OMP_CPAPR_PI=kpi where kpi is 0 or 4 will generate a serial version of the Pi calculation. Version 0 is the original serial code and version 4 is the serial code with all inner-loop functions eliminated in favor of pointer arithmetic; the pointer arithmetic is brittle in that a change in the storage conventions of FacMatrix or Ktensor will invalidate it.

In the event that a library builder specifies an invalid value of kpi, the build of cp_apr_pi will exit with an error message. Independent of the specific parallelization option chosen is the

option to apply a first-touch scheme to Pi memory allocation. Passing -D_TTB_DISTRIBUTE_PI to the C++ compiler will enable statically parallelized initialization of the Pi data array.

2.3.3 The Phi calculation

The cp_apr_phi algorithm applied to an Nd dimensional sparse tensor X computes a factor phi as a sum of count-weighted, scaled products of Pi and the approximant M computed in cp_apr_pi . This amounts in memory traffic terms to a streaming read of Pi, a streaming read of X, a random read of M with addressing driven by the subscripts of elements of X, and a random read/write of Phi also driven as M.

The Phi calculation algorithm is parallelized by splitting the work into contiguous chunks of non-zeros from X. Four implementations of the algorithm are coded, and at present the scheduling directive used for all is *static*, as preliminary studies indicated no benefit from dynamic scheduling. Note that unlike the Pi calculation, random writes are involved and care must be taken to eliminate race conditions. Each thread uses a temporary, private Phi FacMatrix, *tmpPhi*, to accumulate partial results. TmpPhi dominates the memory usage when large numbers of threads are used. For example, on a $10000i^3$ tensor with $7.7x10^8$ nonzeros, the tensor X is 24GB and the total memory used is 310GB.

The first implementation is the original serial code. It is compiled if OpenMP is not detected at compile time or if the user builds with -D_TTB_OMP_CPAPR_PHI=kphi with kphi is 0. No parallelization is used in this case.

The second implementation uses an OpenMP critical section (a lock) to serialize the adding each tmpPhi to the output Phi. Using this version can be tested by compiling with OpenMP enabled and -D_TTB_OMP_CPAPR_PHI=kphi where kphi is 1. This serialization strongly dominates the phi cost as the number of threads grows.

The third implementation streams the sum over tmpPhi operation using a new operator added to FacMatrix, *plusAll*. The implementation of plusAll is parallelizable. One parallelization has been implemented, decomposing the loop over the data array index and summing elements across all the tmpPhi elements for a given index. This reduction scheme, while far more scalable than the serialized version, is unlikely to scale well (in naive form) across multiple ultraviolet nodes. Using this version can be tested by compiling with OpenMP enabled and -D_TTB_OMP_CPAPR_PHI=kphi where kphi is 2.

The fourth implementations is as the third, but modified to eliminate barriers to instruction level parallelism by eliminating pointer lookup calls in the inner loop. This change provides a 15% decrease in runtime for the function as a whole compared to the third version. This is the default version and corresponds to kphi = 3.

In the event that a library builder specifies an invalid value of kphi, the build of cp_apr_phi will exit with an error message.

2.3.4 Tensor generation

Generating test tensors to model sparse count data is done by generating random integer subscripts (within the size range of each tensor dimension) which may be redundant, sorting the entire tensor by subscript indices, and then merging redundant entries. Subscripts which appear once will have a value of 1, merging 2 redundant entries yields a value of 2, and so on.

Large scale random number generation in serial has difficulties which are addressed in the earlier code correctness section. Parallel subscript generation adds the requirement of generator thread safety and distinct seeds for each thread, but is essentially an easily threaded streaming write. I parallelized the generation phase for loop with the *runtime* directive, as was the case with the cp_apr_pi calculation. The generator seed (or set of seeds) is reused each time a *generate* call is made. This seed choice guarantees that for a given dimension, size, and number of threads, the same tensor will always be generated, an appropriate choice for benchmark work. A choice less likely to produce duplicate tensors if multiple random tensors are needed in the same run would include using the system time or the bits of the tensor object pointer for the seed base.

Parallel sorting requires irregularly sized work chunks and is discussed separately below. Detection and merging of the redundant entries after sorting can be parallelized at the cost of redundant shifts of the separately compressed sections. As the serial cost of this operation is presently negligible relative to the parallel sorting cost, this parallelization is not yet implemented.

2.3.5 Tensor sorting

Parallel sorting is a widely studied topic, as the irregular workload of divide and conquer algorithms leads to a many heuristic variations. The failure of the OpenMP 2.0 standard to adequately support irregular benchmarks including quicksort lead directly to inclusion of the task queue in OpenMP 3.0. The serial quicksort originally implemented for Sptensor makes an arbitrary partitioning of the unsorted data and recursively continues partitioning and moving data until all partitions are size 1 and the data is therefore sorted. On large random data, these partitions are usually very imbalanced.

To parallelize this algorithm, after a review of several public attempts in the literature, I added the function *quicksort_p* which introduces a thread task for each subrange defined after the partition operation. This ultimately adds very many very small tasks to the OpenMP queue unless one places a minimum partition size below which the recursion continues in serial only. The size cutoff is arbitrary and heuristic and dependent on the processor hardware. A brief check of performance on a large test tensor on ultraviolet suggests that a good cutoff is in the range between 20,000 and 2,000,000. The default set in the code is 100,000; it can be tuned at build time as mentioned in the earlier correctness discussion with -D_TTB_SP_SORT_MAGIC=k .

3 Performance Results

In the following results, the platforms are noted as:

- 2x4, a dual socket Dell T7500 workstation with two Intel Xeon E5640 quad core @ 2.67GHz, 24GB RAM, Ubuntu Linux 10.04.3 x86_64 kernel 2.6.32-33-generic, and the following 64 bit compilers: GCC 4.6.2, Intel XE icpc 12.0.1.07, Portland Group pgCC 11.7-0.
- 4x8, a quad socket SGI Ultraviolet server node (not connected to other nodes) with 4 Intel Xeon X7550 oct core @ 2.0 GHz, 512 GB RAM, SGI provided SUSE Linux Enterprise Server 11.1 x86_64 kernel 2.6.32.43-0.4-default, and the following 64 bit compilers: GCC 4.6.2, Intel XE icpc 12.1.0.233 Build 20110811, Portland Group pgCC 11.7-0.

Both platforms run in NUMA mode with hyperthreading disabled. Timing data presented are elapsed wallclock times collected from benchmarks executed with user-exclusive use of the machine, but the normal complement of multi-user services idling (and periodically waking up) in the background. Thus the numbers here may have some value in estimating practical time requirements for similar problems, but they are not the absolute best that the hardware could achieve by further tuning the code and the operating system. At present timing data are collected with boost timing functions. OpenMP timing routines could, with some rewrites, replace boost if it is unavailable. For all loops using directive schedule(runtime), static scheduling was specified via the environment variable OMP_SCHEDULE .

3.1 CPAPR computation

In Figure 1 are the results of scaling tests on the 4x8 machine, marching from 8 to 32 processor in steps of 4. Comparisons can be made among lines of the same color across these plots to understand the effect of problem size on a particular function and compiler combination. In each data line of the plots, a spike at 8 threads shows the comparison between using 8 cores in the same socket and 8 cores selected round-robin across all 4 sockets; the end of the spike is the single-socket value. At small sizes (24k,48k) the processors are clearly starved for work and multithreading is at best a break-even choice for the commercial compilers. At larger sizes, the phi operation scales well, leaving the pi operation to dominate runtime for high thread counts.

The number of nonzeros used in the cube tests is slightly less than the number requested and listed on the plot, due to the generation process used. The actual counts for the four plots are, respectively: 23926, 47667, 231509, and 447608. The data for all cases were generated off-line and reloaded from files for each test.

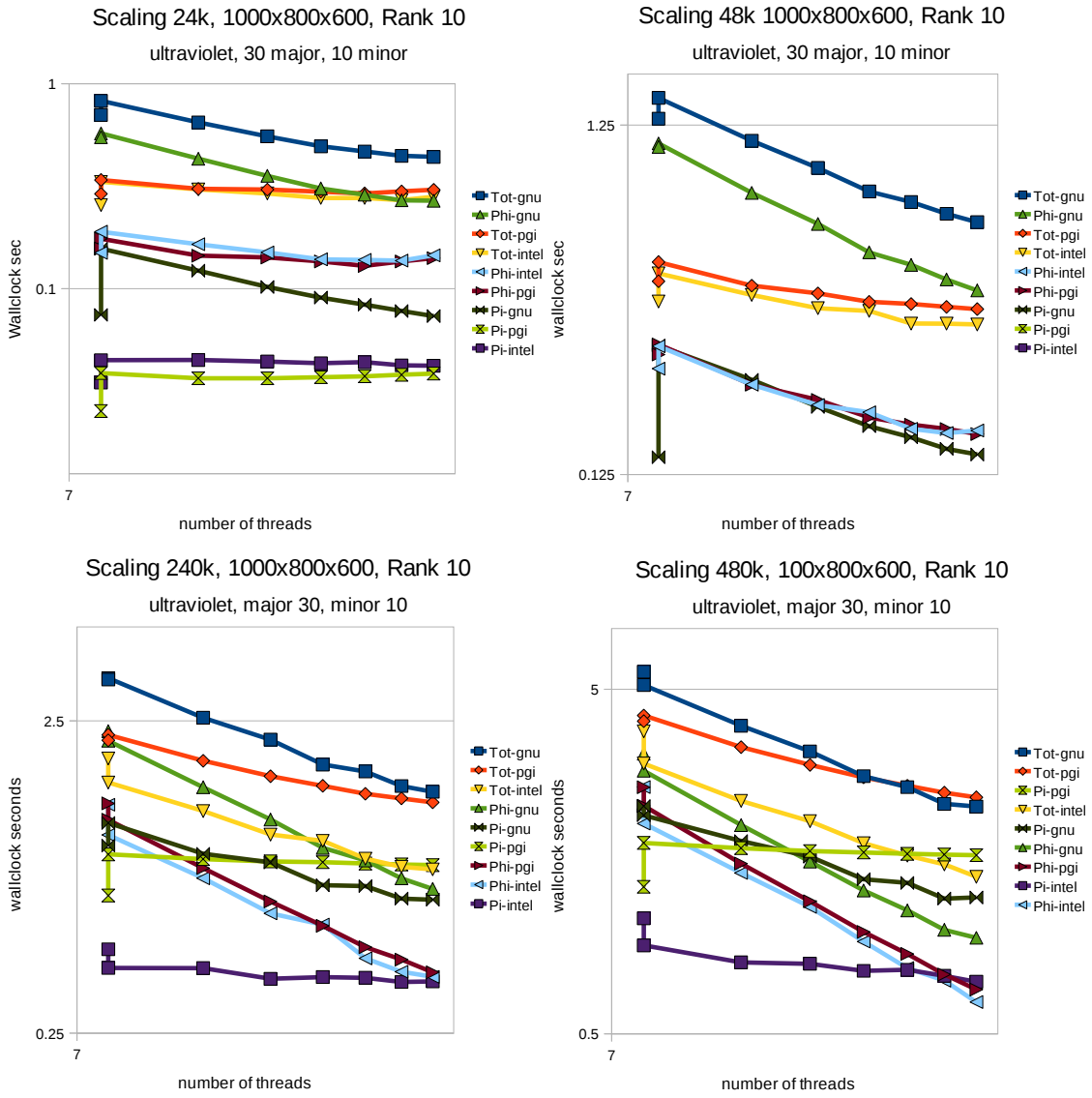

Figure 1: Overall timing and breakdown on small and midsized cubic tensors

In the cube tests, the Intel compiler substantially outperforms the others for midsized tensors. This result holds for the huge cube in Figure 2, but is less evident in the lopsided tensor shown in Figure 3. In both these large tensor tests, the Pi operation scales poorly, and it becomes the dominant cost beyond 16 threads in the large cube test.

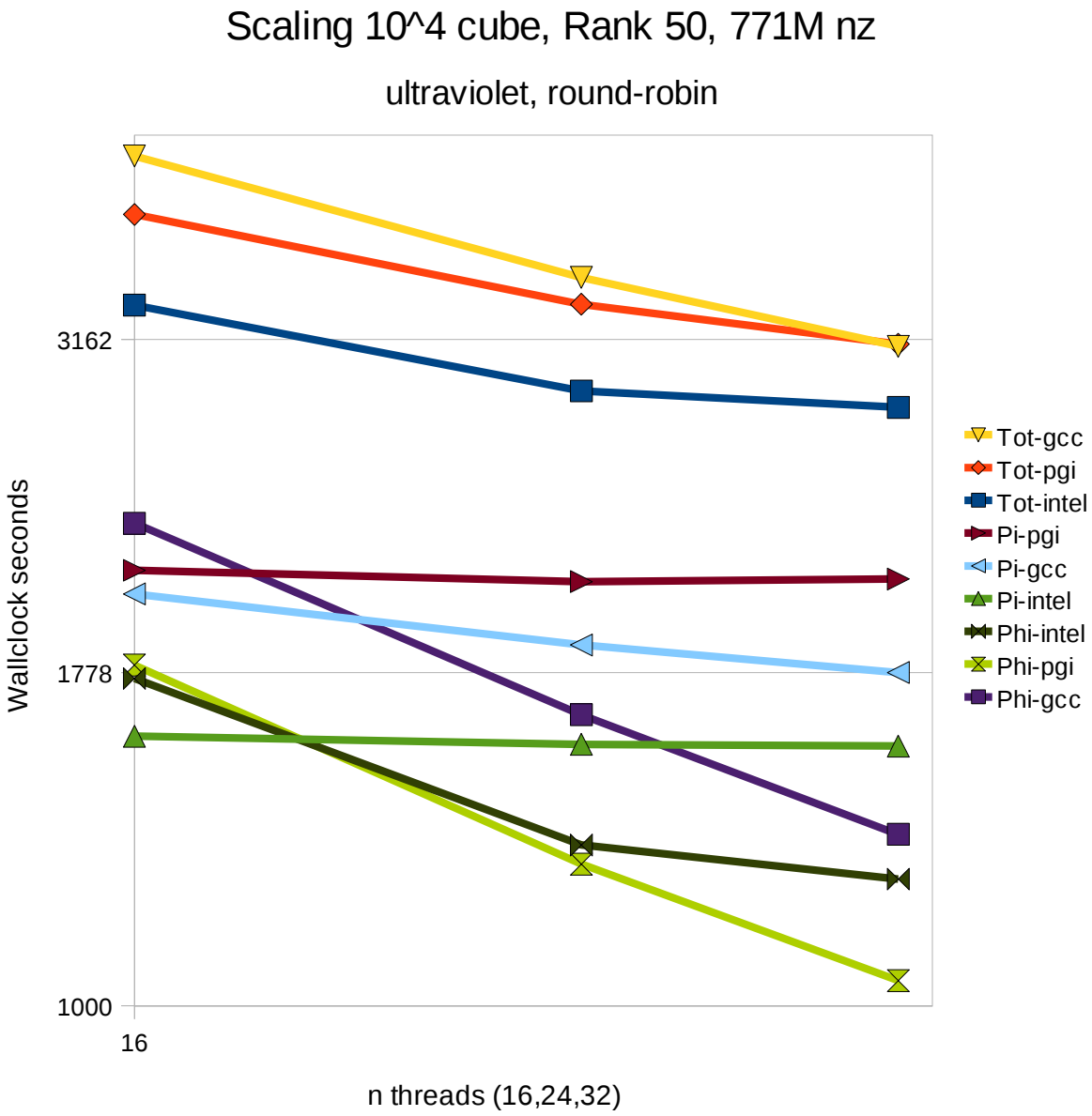

Figure 2: Large cube on 16-32 threads, stepping by 8.

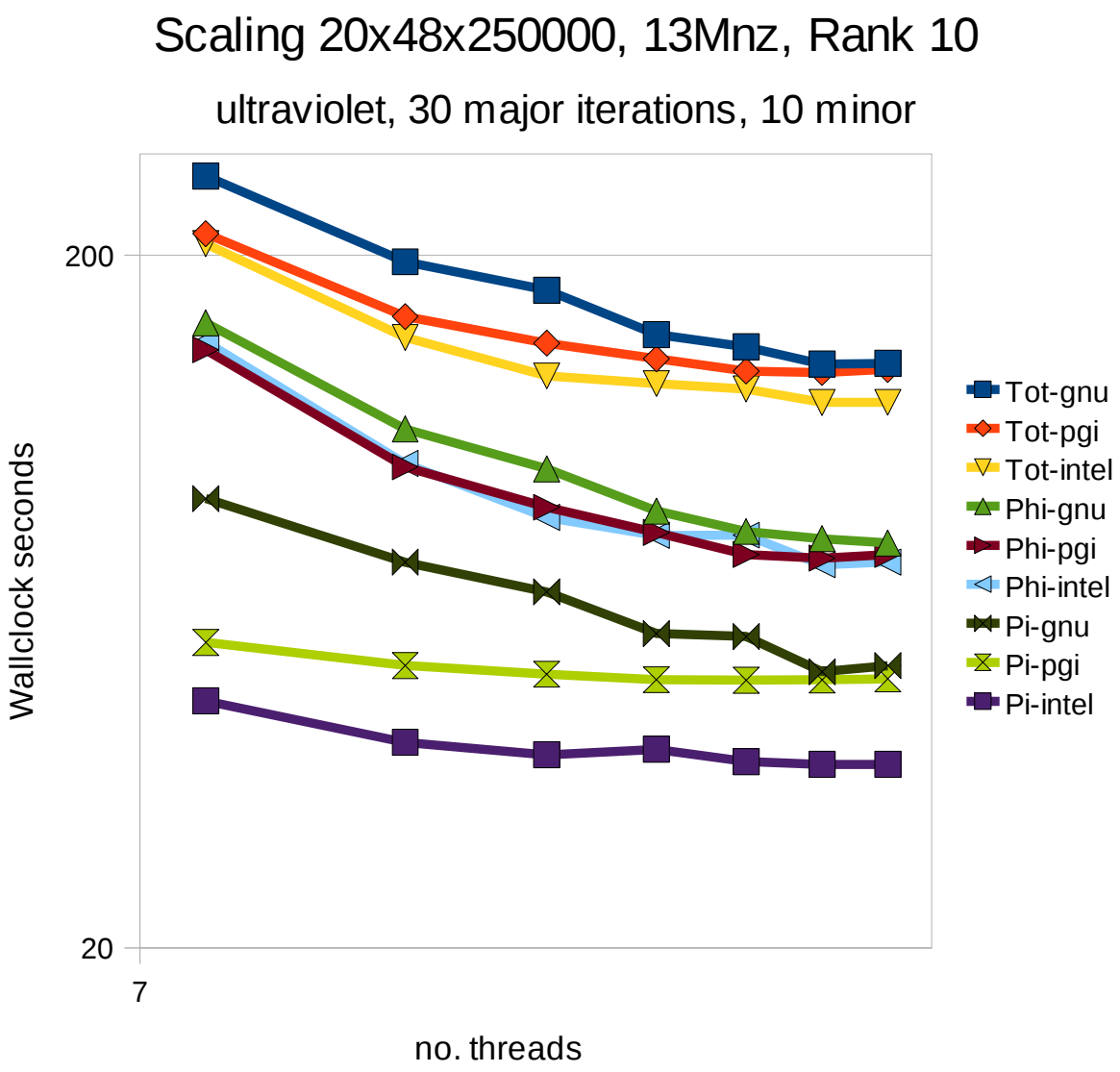

Figure 3: Lopsided tensor on 8-32 threads, stepping by 4.

3.1.1 CPAPR Pi

Here we present the Pi performance data separately for clarity, though it is the same data that appears on the previous plots. The Pi function is an example of the general claim that scalable performance is not portable. The time available did not permit a detailed examination of the generated assembly code to better understand what strategy the PGI compiler took that lead to a performance breakdown or why the GCC compiler at 32 cores is worse than the Intel on 8 cores. The Pi data of the large cube test overall plot is sufficiently clear that it is not plotted separately here. The amount of arithmetic and number of memory writes in the inner loop of the Pi calculation is so small at a decomposition rank of 10 or 50 that this kernel gets little benefit from the multicore hardware.

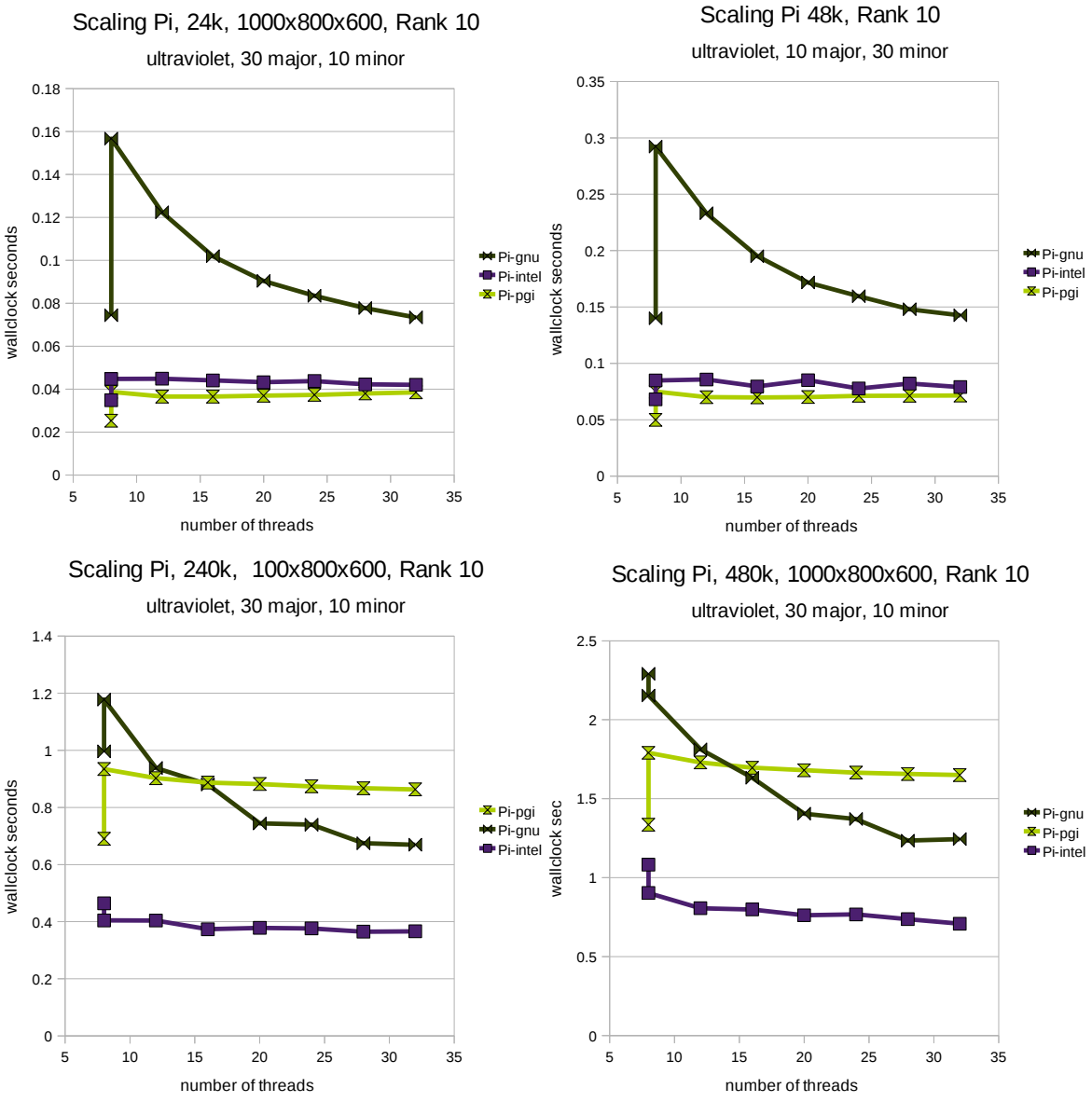

Figure 4: Pi performance on small and midsized cubes

Figure 5: Pi performance on a lopsided tensor, 1.3E7 nonzeros

3.1.2 CPAPR Phi

Here we present the same Phi data seen in the overall plots, but separately and on linear axes for clarity. Clearly in the midsized cubes, the spike down at 8 threads shows that distribution of threads over sockets improves the performance for all the compilers, relieving cache contention somewhat.

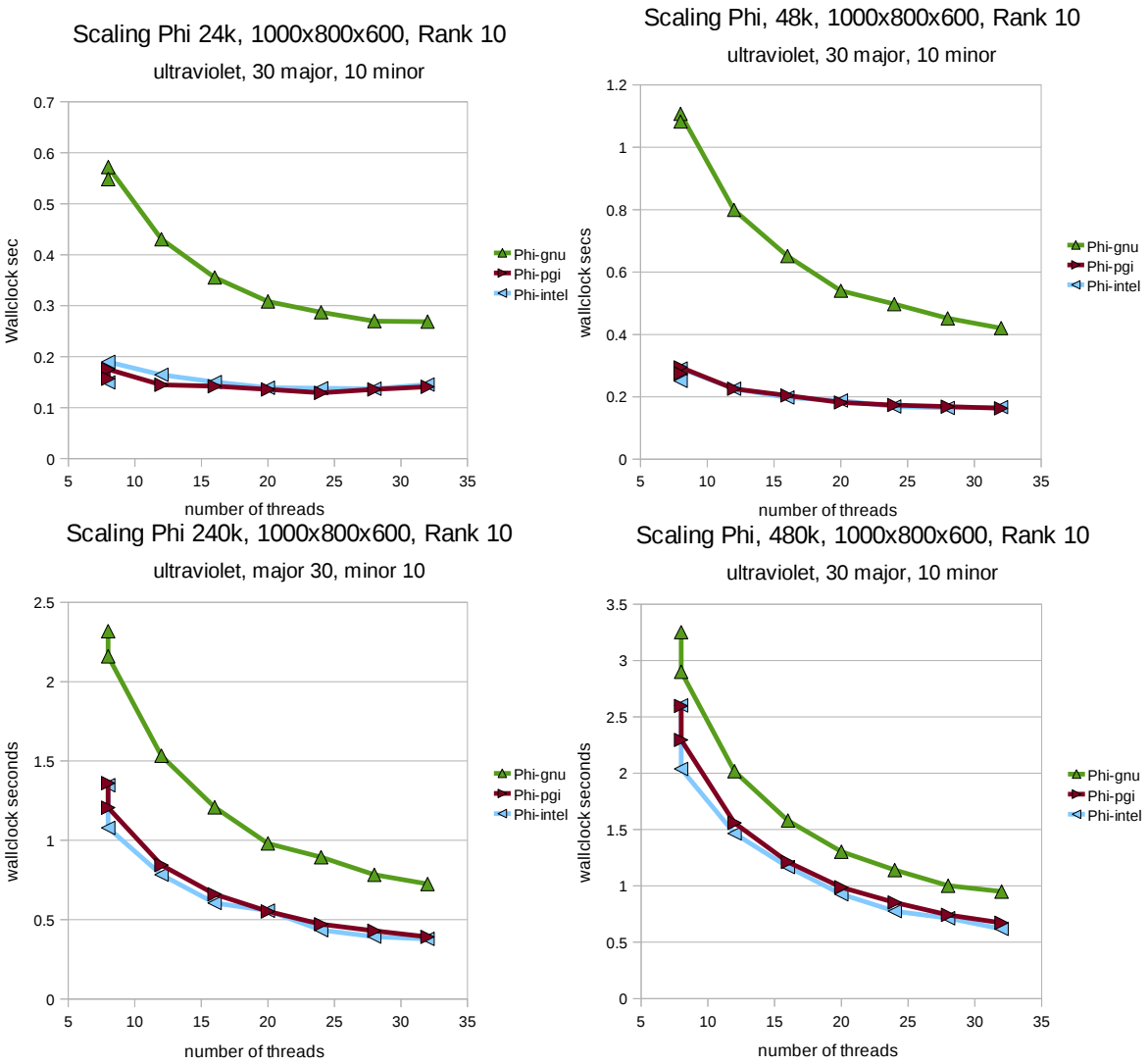

Figure 6: Phi performance on small and midsized cubes

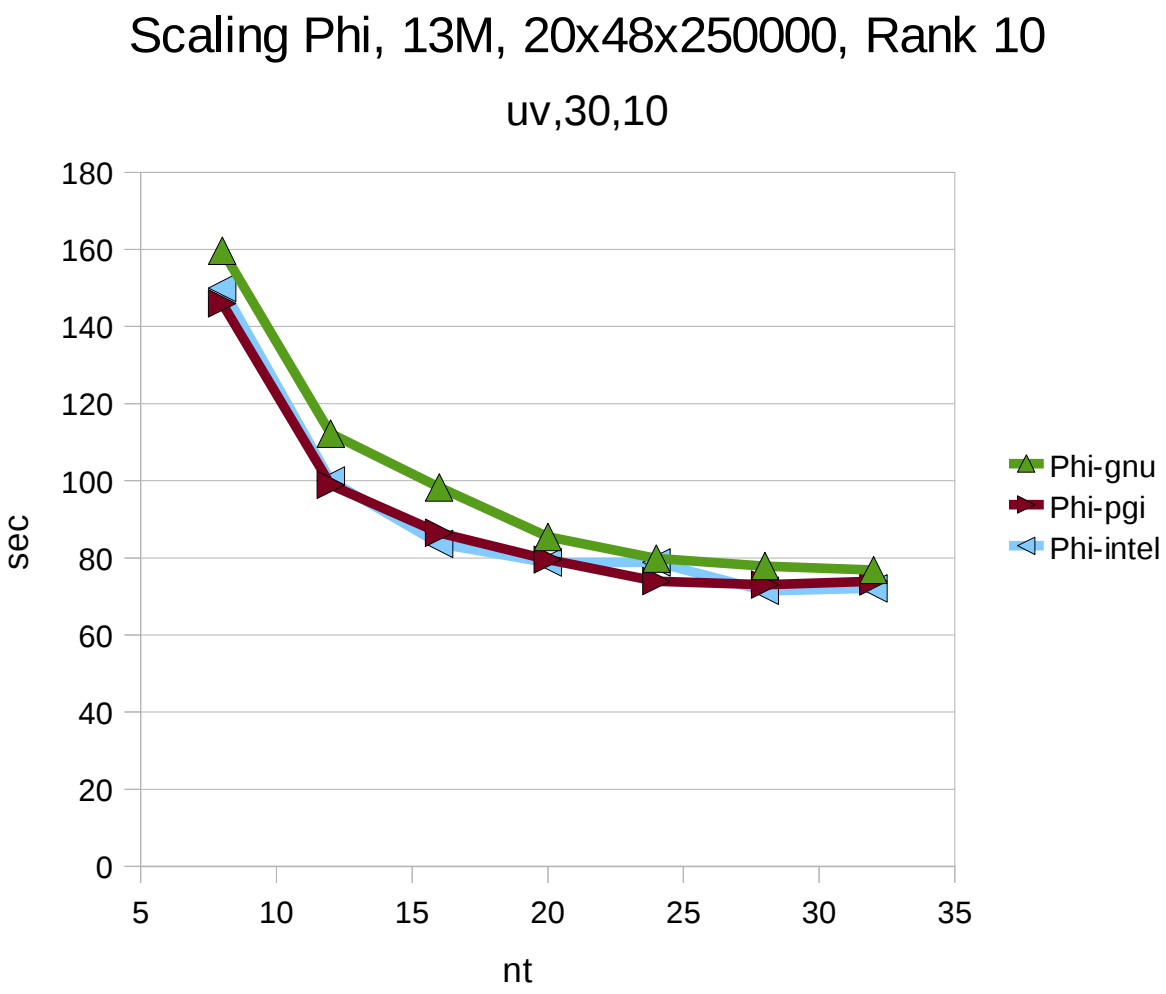

Figure 7: Phi performance on a lopsided tensor

3.2 Data generation

Data generation on ultraviolet follows the trends shown in the smaller 2x4 hardware study of Figure 8. Random generation of elements is a small fraction of the total data synthesis cost. Sorting and merging elements for large data is adequately scaled by handing off independent contiguous chunks to single threads. For thread counts up to four, both single socket (local) and round-robin thread allocation data are plotted. Unlike previous plots, the vertical axis is the log rate of nonzeros processed. Sorting clearly dominates the cost.

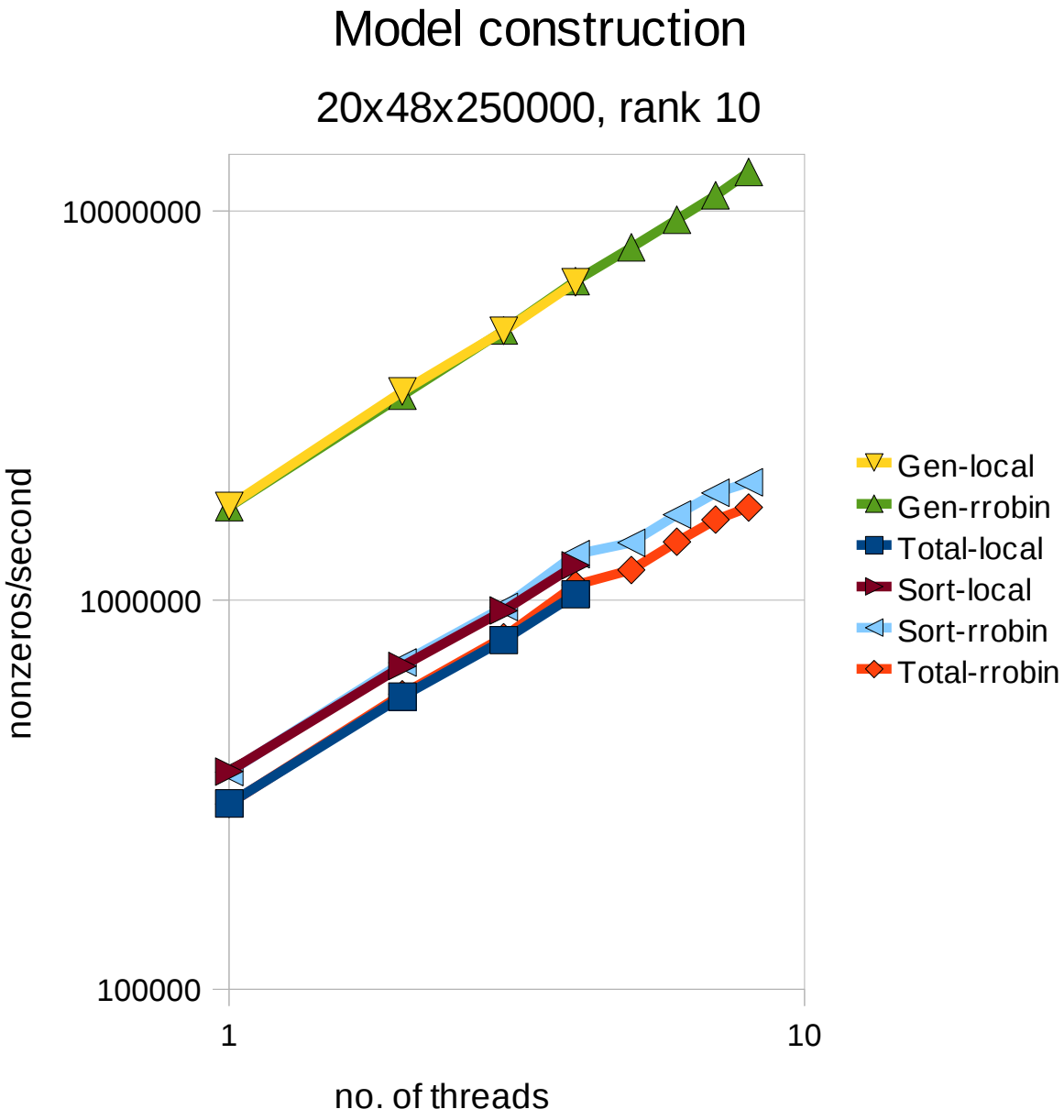

Figure 8: Generation

For the largest data set generated on ultraviolet, we obtained the data in Table 1.

Table 1: Summary of largest data synthesis (32 core), 10000^3, Rank 50

requested elements	$2x10^9$
resulting elements	$0.771x10^9$
total seconds	793
generation seconds	88
sorting seconds	705
nz/sec/core	78000

3.3 IO

Binary IO streams numeric data to memory, while readable text IO requires complex format conversions and error checking. Text based IO is far slower than binary IO for numeric data. IO performance for large files is expected to scale roughly linearly in data size, absent a parallel IO library, so I present no scaling data. When generating the largest model (24 GB), data write time was about one minute out of 14 minutes total generator runtime.

4 Drivers

Three drivers have been created in TTB_cpp/TTB_parallel, gen_cp_apr_input, dump_cp_apr_input, and test_cp_apr_special. All three respond to the command line argument *–help* with a usage message. Generating large data test models (a paired Sptensor and Ktensor in separate files) is done with gen_cp_apr_input. Identifying the contents of a binary data file or dumping the contents to text format is done with dump_cp_apr_input. Loading (or generating anew) test models and running the cp_apr algorithm is done with test_cp_apr_special. Each driver has a correspondingly named single C++ source file that links with the TTB libraries.

Presently all the drivers are built with a Makefile independent of the CMake build system. Care must be taken when using the makefile, as its compiler selection and compiler flags do not automatically track those used when building the TTB libraries.

This page intentionally left blank

5 Conclusions and recommendations

On 32 cores, overall wallclock cost for decomposing a large tensor is less than 10% of the serial decomposition cost. This improvement comes at a memory cost for localized temporaries which scales linearly in the number of processors. In testing of sorted tensor decomposition with cp_apr, the sum of Pi and Phi operations accounts for over 91% of runtime on 32 cores, and a higher percentage on fewer cores, meaning further parallelization without significant data structure changes cannot be expected to substantially improve overall performance. The optimal memory layout for the Pi computation may be different from that for the Phi computation. The current code is tuned to the memory traffic pattern of the Phi operation, attempting with first-touch and thread-local allocation to reduce the amount of off-socket data handled by the Phi computation.

As the code is currently organized, it is likely that advanced multicore technologies (e.g. Intel MIC) would provide additional benefit for large models only if memory bandwidth is expanded to keep the cores fed and the amount of memory connected to a single socket is sufficient to hold the entire Pi and M factor matrices.

Here is a list of secondary operations that present immediate opportunities for further parallelism, but that have not yet been threaded. In porting to hardware supporting a larger number of remote threads, these functions may be expected to appear as bottlenecks. In many of them, adding naive thread support will be optimizing for large data sets and pessimizing for small data sets. Here small data, for 2010 Xeon processors, is data such that the product of the tensor decomposition rank and the size of the largest individual dimension of the approximated tensor X is less than 1000. Each operation is expected to exhibit linear improvement when handling large data, based on inspection, although several of them are reduction operators.

- FacMatrix::times (both scalar and vector versions)
- FacMatrix::cp_apr_kkt
- Ktensor::normalize
- Ktensor::distribute
- Array::norm

5.1 Recommendations

1. Extract the CPAPR logic used in the test driver and move it into a function on the Sptensor class. Alternatively, move it into a separate class with the core algorithm and several methods for setting options that control parallelism strategies, allocation strategies, and collection of performance data. The latter approach would facilitate

rapid experimentation on future platforms. It would also provide an appropriate class to contain the factor match scoring algorithms.

2. The effect of the first-touch implementation on both Pi and Phi computations should be better studied. Now that basic threading approaches have been established, advanced performance instrumentation to characterize cache performance and remote vs local memory access is appropriate. When OpenMP is present, first-touch oriented initialization of the data and index arrays in Sptensor is applied, independent of the argument *parallel* passed to the resize operations during construction; this needs to be fixed for thread-locally allocated Sptensors to work as expected.

3. The memory overhead of tmpPhi can most likely be reduced in the case where indices of X are sorted. A scan of the sparse index data for each thread will reveal what portion of its temporary factor matrix will be written, and only that chunk of the temporary factor need be allocated.

4. More instruction-level parallelism and better cache use could be extracted by shifting the implementation from 8 byte to 4 byte data storage. Care must be taken that loops over arrays are written over indices of type *size_t* and not *ttb_indx*, or integer wrap-around will lead to silently incorrect algorithms. It seems unlikely that the multiplicative update scheme is sensitive to the errors induced by single precision floating point in a few hundred to a few thousand steps.

5. Care should be taken to include at least the Intel compiler, as the scaling studies for a single compiler (particularly GCC) can be misleading regarding the possible total performance of the code on a given hardware. If the latest generation of AMD processors (Bulldozer) is tested, the PGI and AMD compilers should be included in the testing. For Sandia Linux users, PGI and Intel compilers are available on host compilers.sandia.gov.

6. Users on single-core platforms or otherwise determined to use the code in serial mode should compile with -D_TTB_OMP_CPAPR_PI=kpi where kpi = 4.

7. Until the OpenMP 3.0 task implementations become more reliable, the data generation and sorting application should be built with GCC 4.6.2 (or later) and a separate build with Intel compilers for solving (with no -D_TTB_OMP_TASK).

References

[1] Eric C. Chi and Tamara G. Kolda. On tensors, sparsity, and nonnegative factorizations. arXiv:1112.2414 [math.NA], December 2011.

DISTRIBUTION:

1	Benjamin Allan, 08961	MS 9158
1	Tamara Kolda, 08966	MS 9159
1	Nicole Lemaster, 08961	MS 9158
1	RIM (Reports Management), 09532 (electronic copy)	MS 0899

Sandia National Laboratories