PROCEEDINGS OF SCIENCE

Experiences with OpenMP in tmLQCD

A. Deuzeman
Albert Einstein Center for Fund. Physics, University of Bern, CH-3012 Bern, Switzerland
E-mail: albert.deuzeman@gmail.com

K. Jansen
NIC, DESY, Zeuthen, Platanenallee 6, D-15738 Zeuthen, Germany
E-mail: karl.jansen@desy.de

B. Kostrzewa[*]
Humboldt Universität zu Berlin, Institut für Physik, Newstonstr. 15, 12489 Berlin, Germany
NIC, DESY, Zeuthen, Platanenallee 6, D-15738 Zeuthen, Germany
E-mail: bartosz.kostrzewa@desy.de

C. Urbach
HISKP (Theory), Rheinische Friedrich-Wilhelms Universität Bonn, Germany
E-mail: urbach@hiskp.uni-bonn.de

PoS(LATTICE 2013)416

An overview is given of the lessons learned from the introduction of multi-threading using OpenMP in tmLQCD. In particular, programming style, performance measurements, cache misses, scaling, thread distribution for hybrid codes, race conditions, the overlapping of communication and computation and the measurement and reduction of certain overheads are discussed. Performance measurements and sampling profiles are given for different implementations of the hopping matrix computational kernel.

HU-EP-13/60, DESY 13-217, SFB/CPP-13-93

31st International Symposium on Lattice Field Theory - LATTICE 2013
July 29 - August 3, 2013
Mainz, Germany

[*] Speaker.

© Copyright owned by the author(s) under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike Licence.

The tmLQCD software suite [1] is a collection of programs for gauge configuration generation and Dirac matrix inversion for various types of Wilson fermions, most notably including the twisted mass term [2]. For an overview of recently added features, please see [3], presented at this conference. In the following the source-code will be referred to frequently and the interested reader is encouraged to download or browse it on github. [1]

The hopping matrix can be found in `operator/Hopping_Matrix.c`, which includes the files `operator/halfspinor_body.c` and `operator/hopping_body_dbl.c` using 'Weyl-type' spinors with overlapping communication and computation and full four-component spinors without overlapping respectively, for a more complete explanation, see contribution [3] at this conference. Unless mentioned otherwise, all measurements are done on 32 nodes of BlueGene/Q with 1 process per node and 64 threads per process with a node-local lattice volume of 12^4 using the tmLQCD benchmark application.

1. Introducing OpenMP into tmLQCD

In the period leading up to the installation of the BlueGene/Q machine (BG/Q), JuQueen, at JSC Juelich, it was concluded that a multi-threaded version of tmLQCD was essential for good performance in the near future. In the following, a summary of the lessons that were learned through the process of introducing OpenMP across the entire code-base will be attempted.

Using Scoping Rules – Specifying thread-local and thread-global data in OpenMP is usually achieved through the directives `shared`, `private` and others which allow for more fine-grained control. In C/C++, one can exploit variable scoping rules to automate the process by defining thread-local variables only inside `parallel` sections and thread-global variables only outside of them. This way one can avoid potential bugs resulting from forgetting to update the variable lists as the code evolves. As will be seen further below, while simplifying, this approach could however render it more difficult to introduce coarsened parallelism as an approach to reducing thread-management overhead through so-called *orphaned* [4] directives. It can also be argued that the explicit listing of thread-private and thread-global variables aids in understanding, but in a language where variable scope is a central concept, this may be judged as a weak argument.

2. Optimizing the Hopping Matrix

Performance Measurements – The hopping matrix was the first function to be multi-threaded and performance measurements were carried out on multi-core Intel® CPUs, the results of which are shown in figure 1. It was found that for a problem that fits into cache on a single node of this machine, the OpenMP version of both the full-spinor and the half-spinor hopping matrices outperforms the pure MPI version with a corresponding number of processes. Once multiple nodes are required, however, the pure MPI version outperforms the hybrid OpenMP/MPI implementation on this architecture. The authors believe that this regression originates from the loss of symmetry and additional thread synchronization required when the communication is dispatched by one of the threads. This suspicion is supported by the measurement of the half-spinor hybrid code with

[1] The version of the software with the overlapping kernel using MPI or SPI communication is currently found in the repository "`github.com/urbach/tmLQCD`" in the "`InterleavedNDTwistedClover`" branch.

PoS(LATTICE 2013)416

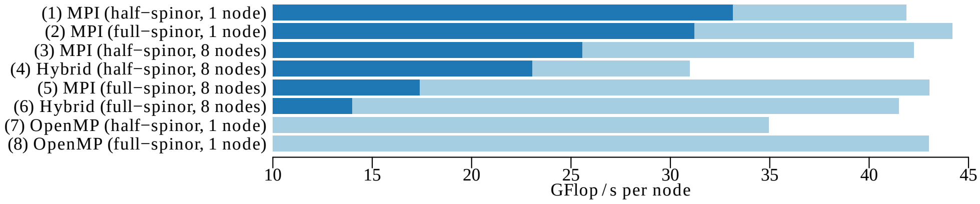


Figure 1: Performance of the tmLQCD hopping matrix on a dual-socket Intel Xeon X5560 Infiniband cluster with a lattice volume per node of 16×8^3. The lighter bars show performance with communication routines explicitly disabled. Hybrid refers to 2 processes per node and uses a kernel which attempts to overlap communication and computation.

communication disabled (light bar 4) but all the OpenMP structure still in place. It is possible that optimization of MPI parameters might produce higher performance because the message size is clearly larger for the hybrid version than for the pure MPI version. The various overheads will be discussed in detail below.

Although not studied in depth for these cases, the performance regression persists on Cray XC30 but appears to be quite mild on SuperMUC, allowing scaling to larger machine partitions through the usage of the hybrid code there. As will be discussed in more detail below, on BG/Q the hybrid version consistently outperforms the MPI-only implementation because overlapping of communication and computation works very well on this machine.

An important conclusion that can be taken away from these results is that for hybrid parallelizations, measurements need to be carried out on each target platform taking into account different compilers, communication libraries and even the problem size as will be discussed below.

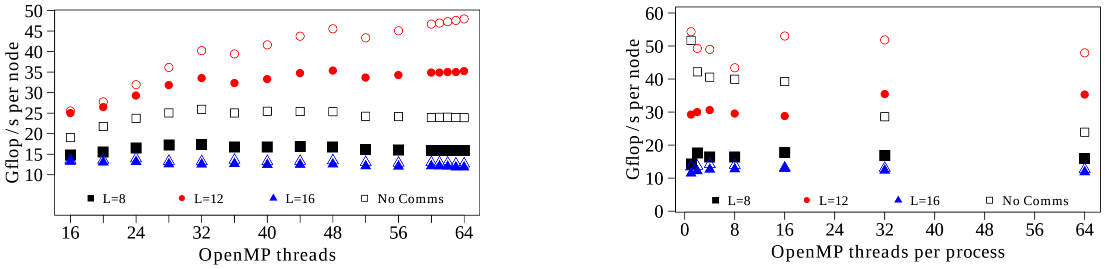
Cache Misses – During these initial measurements it was noticed through the usage of the Intel® VTune™ Performance Analyzer that in the full-spinor version, a large number of cache misses and resulting thread idling occurred from lookups of neighbourhood lattice indices which are stored in two-dimensional arrays `g_iup[x][mu]` and `g_idn[x][mu]`. While a direct lookup like `xp1=g_iup[x][1]` did not pose a problem, the compiler could not optimize access to a double lookup like `xm0p1=g_iup[g_idn[x][0]][1]`.

A dedicated array of indices was added, precomputed during program initialization, which contains the lattice indices in the order required by the hopping matrix and which is accessed in this way: `ix=g_hi[16*x+n]`, where `n` is a constant integer depending on the operation in this line. This simple optimization resulted in a performance boost of almost 20%.

Scaling with Thread Distribution – The performance of the hopping matrix as a function of the total number of threads and their distribution across processes was studied on BG/Q, the results are shown in figure 2. Note that the more efficient SPI communication was not used for these tests.

One conclusion which can drawn from these measurements is that for local lattice volumes which fill the L2 cache close to maximally, a configuration with 1 process per node and 64 threads is fastest. On the other hand, when the L2 cache is not fully exploited, configurations with 4 threads per process or 32 threads in total seem to be beneficial. Comparing in panel (b) performance with communication for $L = 12$ for 16 and 32 threads per process, MPI seems to benefit from using fewer processes per node as the pure floating point performance is comparable. On the other hand, for the $L = 8$ measurement, performance without communication is severely degraded, suggesting inefficiencies in multi-threading. It is conceivable, therefore, that for small local lattice volumes with optimized task placement, fewer threads per process and more processes are optimal.

The reason for this situation seems clear: when there are many threads per process the OpenMP



(a) 1 process per node and increasing numbers of threads.

(b) The number of threads times the number of processes per node is kept constant at 64.

Figure 2: Performance on 32 nodes of BG/Q in GFlop/s per node using an overlapping OpenMP/MPI hopping matrix for different node-local lattice volumes L^4. Empty symbols correspond to inter-process communication being disabled.

overhead is very large and this can only be mitigated by a large workload for each thread, allowing scaling (a) with the number of threads for $L = 12$ which is not seen for $L = 8$. By contrast, when the local lattice volume is very small ($L = 8$), it makes sense to absolutely minimize the OpenMP overhead, as shown by the significant loss of pure floating point performance as the number of threads per process is increased in panel (b).

Race conditions and Memory Locking – Algorithms such as the hopping matrix can be efficiently implemented in a "push-style" fashion where a loop over the volume "pushes" computation results to neighbouring lattice sites. When the work loop is shared amongst threads, as any lattice site is the neighbour of 8 other lattice sites, multiple threads can potentially attempt to write the same memory location, thus requiring some form of memory locking. This is usually done through `critical` directives or by updating memory "atomically" using the `atomic` directive. While the former approach is safe, it results in extreme overheads [5] which are unacceptable in a lattice QCD code. The usage of `atomic` directives seems to have low overhead, but correctness is not guaranteed [6, pp.380-390] when there are multiple writing and reading threads and the same memory location could be accessed from several `atomic` statements on different source-code lines (which would be the case for the hopping matrix, for example). Further, the set of operations which can be carried out atomically is very limited.

An alternative to locking is the addition of "halo memory" for *each* lattice point as pictorially depicted in figure 3. Because each lattice point now has independent neighbourhood memory, the conflict situation discussed above cannot occur. This implementation, however, requires significantly more memory and the results for the different halos need to be accumulated in a second loop over the volume. Since the tmLQCD half-spinor hopping matrix was already implemented in this manner even before the introduction of threads, no performance comparison has been carried out.

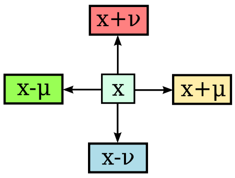


Figure 3: Pictorial representation of the concept of "halo memory".

In the computation of the fermionic derivative during Hybrid Monte Carlo integration, tmLQCD does not use halo memory but the `atomic` directive when updating the components of the real-valued derivative field. Very high statistics comparisons of serial and multi-threaded HMC runs have shown that there seems to be no noticeable effect of the danger discussed in [6]. A dedicated study of the potential for errors from this kind of set of `atomic` statements is planned, as well as a test of halo memory for this operation.

PoS(LATTICE 2013)416

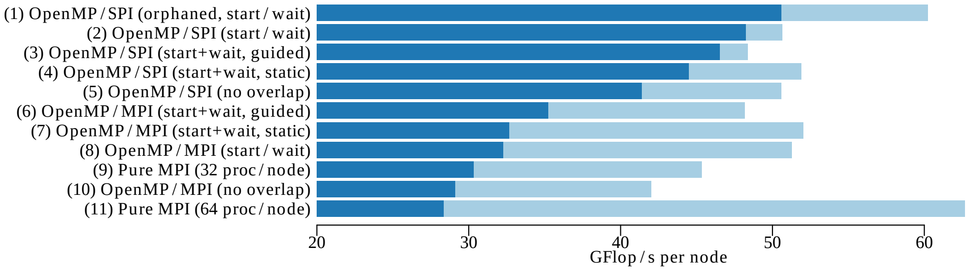


Figure 4: Performance measurements for different versions of the hopping matrix. The lighter bar shows measurements without communication. Unless noted otherwise, the hybrid kernel attempts to overlap communication and computation.

Overlapping Communication and Computation – The instructions for tackling this problem are usually as follows: compute the data which is to be communicated (the "surface"), start the communication using non-blocking communicators, perform the part of the computation which does not depend on communicated data (the "body"), call MPI_Wait(all) and finally compute the remaining results. Unfortunately it seems that in the cases that were tested by the authors (using MPI_Isend/recv), all the communication is done during MPI_Wait(all), no matter how much time is spent in the body computation, thereby completely defeating the purpose of attempting to overlap in the first place.

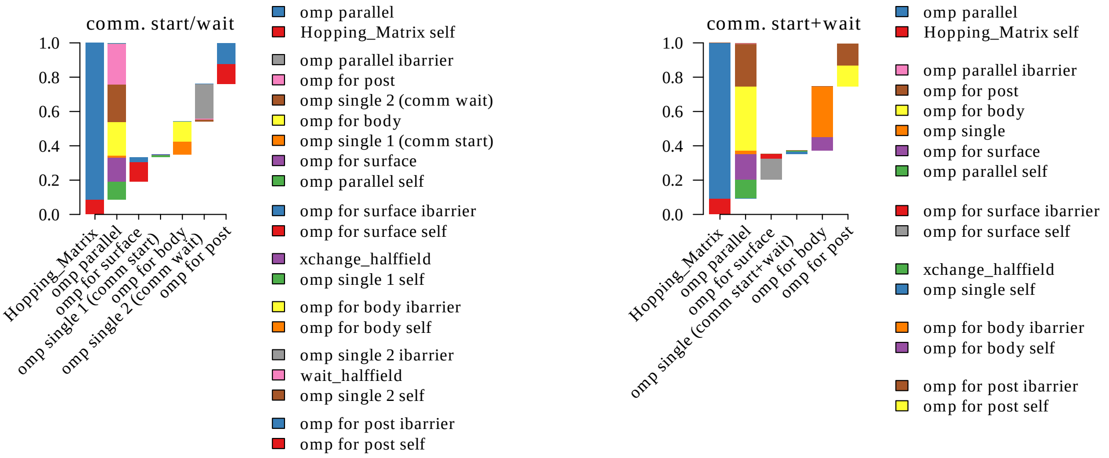
In a hybrid application, however, one could instruct one (or multiple) thread(s) to initiate the communication in a section with a `nowait` directive and then immediately call MPI_Wait(all), thereby actually starting the communication. When static thread scheduling is used in the body loop, this however results in a loss of performance as the other threads have to wait for the communicating thread(s) to rejoin after doing its (their) work on the body computation. It was found that employing `guided,32` scheduling in the body computation significantly increases performance. In addition, this implementation theoretically benefits from smaller overhead because it requires one fewer synchronization as the communicating thread(s) is (are) 'caught' by the implicit OpenMP barrier at the end of the body loop.

An overview of the structure of the "start→compute→wait" and "start+wait→compute" implementations is shown in figure 5, with the time spent in each sub-part sampled using SCALASCA [7]. It must be noted here that the difference in overheads between static and guided scheduling discussed above is only marginally reflected in the respective SCALASCA samplings (note: only `guided` shown here!), although the effect on performance is apparently substantial.

A summary of performance measurements on BG/Q using different implementations of an overlapping kernel is shown in figure 4. The hybrid code without overlapping (10) is only marginally faster than the pure MPI code (11). A code which attempts to overlap following the 'usual' instructions is somewhat faster (8). The implementation using only one `single` section is faster still (7) but only through the optimization of the thread scheduling is performance improved significantly (6). Note that performance without communication drops (compare the light bars in 6 and 7), because `guided` scheduling is less efficient. These relationships between the different hybrid implementations persist on different machines like the Cray XC30 or SuperMUC, with the caveat that the performance benefit from overlapping is only around 100 MFlop/s per core and the pure MPI version without overlapping outperforms all of them, at least on the machine partitions that were tested.

This picture is complicated by the fact that when using SPI for communication, the "naive"

PoS(LATTICE 2013)416



(a) **(b)**

Figure 5: SCALASCA samplings of time spent in the sub-parts of the half-spinor hopping matrix in the tmLQCD benchmark application. The legend gives the division of each bar where the qualifier "self" refers to time spent in a given part which is not spent in any of its sub-parts. Panel (a) shows the distribution in the function where non-blocking communicators are called before and MPI_Waitall is called after the body loop. Panel (b) shows the situation when the same thread is used to start and wait for communication, with implicit synchronization at the end of the body loop.

implementation using two `single` sections is faster than the one discussed above as shown by bars 2 and 3. The authors assume that using SPI for Remote Direct Memory Access (RDMA) actually launches the communication without requiring many CPU cycles. The loss of performance from `guided` scheduling in the body then causes overall performance to drop more (compare light bars in 4 and 3), than is gained by the removal of one extra synchronization (compare light bars in 4 and 2). An RDMA implementation using one-sided MPI communicators was not attempted yet.

3. Measuring and Reducing Overheads

The OpenMP overhead in the hopping matrix can be estimated by comparing pure floating point performance (with communication routines explicitly disabled) using 64 processes per node and the respective performance when using 1 process and 64 threads. Comparing thus the light bars of 11 and 7 in figure 4, one can estimate that the OpenMP overhead impacts pure floating point performance per node by around 11 GFlop/s on BG/Q.

A breakdown of the overhead can be estimated from the SCALASCA samplings shown in figure 5, for instance by looking at the "self" contribution to `omp parallel` sections. Further overheads are found in the implicit barriers marked "ibarrier" at the end of loops and `single` sections without `nowait`. For loops, the "self" measurement represents the actual "work" done.

One surprising result from these measurements is that when the tmLQCD inverter is sampled using SCALASCA, it records a significant overhead for thread forking even though thread recycling should be taking place. This is not seen in samplings of the tmLQCD benchmark application, which simply calls the Hopping_Matrix function many times. This suggests that when `parallel` sections in the same function occur repeatedly, this overhead drops out.

Still, the "self" contribution to `omp parallel` (which can be seen to reflect the thread management overhead), constitutes a significant portion of total runtime. It can be significantly reduced through the usage of *orphaned* directives, e.g. `omp pragma for` directives inside functions

PoS(LATTICE 2013)416

which are not syntactically enclosed in a `parallel` section. Instead, threads are launched one (or several) level(s) further up, for example when a solver is called. This has been tested for the hopping matrix benchmark, resulting in measurement 1 in figure 4. The underwhelming result with communication stems from a recent network performance regression on JuQueen, but the light bar indicates a significant reduction of the overhead discussed above. Managing this kind of parallelism using OpenMP can be very demanding because thread-local and thread-global data now needs to be tracked over several function hierarchies. Perhaps in this situation it might be beneficial to state the locality of variables using `private` and `shared` directives instead of relying on scoping rules.

Finally, it is worthwhile to look at overheads on different machines and for different types of functions. A SCALASCA sampling of the (real) scalar product of two spinor vectors is shown in figure 6. This computationally simple function suffers from substantial thread management overhead and a significant amount of time is spent in the barrier at the end of the for loop. Whether optimization would result in measurable improvement seems to depend strongly on the architecture and number of threads, as shown by the comparison between BG/Q and SuperMUC, where the latter is dominated completely by the collective MPI communication. More fine-grained measurements of these and other overheads are planned through SCALASCA event tracing.

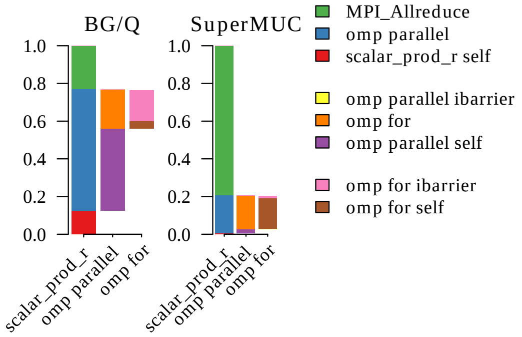


Figure 6: SCALASCA sampling of the sclar_prod_r function on 512 (256) nodes of BG/Q (SuperMUC), using 1 (2) tasks per node and 64 (8) threads per process during an inversion on a lattice volume of $48^3 \times 96$.

PoS(LATTICE 2013)416

Acknowledgements

B.K. acknowledges full financial support by the National Research Fund, Luxembourg under AFR Ph.D. grant 27773315. This work is supported in part by DFG and NSFC (CRC 110). This talk was part of a coding session sponsored partially by the PRACE-2IP project, grant: RI-283493.

References

[1] K. Jansen and C. Urbach. *Comput.Phys.Commun.* (2009). vol. 180:2717–2738

[2] R. Frezzotti and G. C. Rossi. *Journal of High Energy Physics* (2004). vol. 2004(08):007

[3] A. Abdel-Rehim et al. In: *31st International Symposium on Lattice Field Theory*, no. 414 in PoS(LATTICE 2013)

[4] O. Tatebe et al. In: M. Valero et al. (editors), *High Performance Computing*, vol. 1940 of *Lecture Notes in Computer Science*, pp. 471–481 (Springer Berlin Heidelberg) (2000)

[5] J. M. Bull. In: *Proceedings of First European Workshop on OpenMP*, vol. 8 (1999) p. 49

[6] G. Bronevetsky and B. Supinski. *International Journal of Parallel Programming* (2007). vol. 35(4):335–392

[7] M. Geimer et al. *Concurrency Computat.: Pract. Exper.* (2010). vol. 22:702–719