Data Access Characteristics and Optimizations for Sun UltraSPARC T2 and T2+ systems

Georg Hager, Thomas Zeiser and Gerhard Wellein
Regionales Rechenzentrum Erlangen, University of Erlangen-Nuremberg
Martensstr. 1, 91058 Erlangen, Germany

September 4th, 2008

Abstract

Processor and system architectures that feature multiple memory controllers and/or ccNUMA characteristics are prone to show bottlenecks and erratic performance numbers on scientific codes. Although cache thrashing, aliasing conflicts, and ccNUMA locality and contention problems are well known for many types of systems, they take on peculiar forms on the new Sun UltraSPARC T2 and T2+ processors, which we use here as prototypical multi-core designs. We analyze performance patterns in low-level and application benchmarks and put some emphasis on a comparison of performance features between T2 and its successor. Furthermore we show ways to circumvent bottlenecks by careful data layout, placement and padding.

1 The Sun UltraSPARC T2 and T2+ processors

Trading high single core performance for a highly parallel single chip architecture is the basic idea of T2 as can be seen in Fig. 1: Eight simple in-order SPARC cores (running at 1.2 or 1.4 GHz) are connected to a shared, banked L2 cache and four independently operating dual channel FB-DIMM memory controllers through a non-blocking switch, thereby providing UMA access characteristics with scalable bandwidth. Such features were previously only available in shared-memory vector computers like the NEC SX series. To overcome the restrictions of in-order architectures and long memory latencies, each core is able to support up to eight threads, i.e. there are register sets, instruction pointers etc. to accommodate eight different machine states. There are two integer, two memory and one floating point pipeline per core. Although all eight threads can be interleaved across the floating point and memory pipes, each integer pipe is hardwired to a group of four threads. The CPU can switch between the threads in a group on a cycle-by-cycle basis, but only one thread per group is simultaneously active at any time. If a thread has to wait for resources like, e.g., memory references, it will be put in an inactive state until the resources become available which allows for effective latency hiding [1] but restricts each thread to a single outstanding cache miss. Running more than a single thread per core is therefore mandatory for most applications, and thread placement

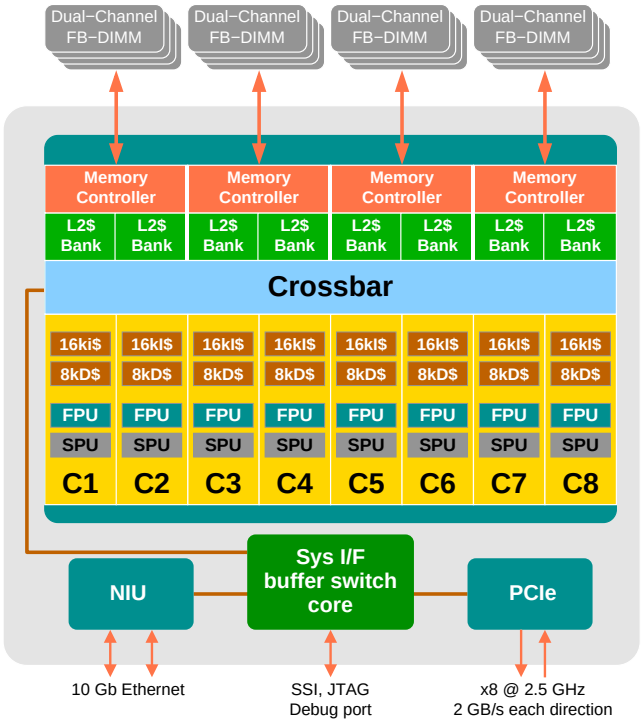

Figure 1: Block diagram of the Sun UltraSPARC T2 processor (see text for details). Picture by courtesy of Sun Microsystems.

("pinning") must be implemented. This can be done with the standard Solaris `processor_bind()` system call or, more conveniently but only available for OpenMP, using the `SUNW_MP_PROCBIND` environment variable.

Each memory controller is associated with two L2 banks. A very simple scheme is employed to map addresses to controllers and banks: Bits 8 and 7 of the physical memory address select the memory controller to use, while bit 6 determines the L2 bank [1, 2]. Consecutive 64-byte cache lines are thus served in turn by consecutive cache banks and memory controllers. Due to the fact that typical page sizes are at least 4 kB the distinction between physical and virtual addresses is of no importance here.

The aggregated nominal main memory bandwidth of 42 GB/s (read) and 21 GB/s (write) for a single socket is far ahead of most other general purpose CPUs and topped only by the NEC SX-8 vector series. Since there is only a single floating point unit (performing MULT or ADD operations) per core, the system balance of approximately 4 bytes/flop (assuming read) is the same as for the NEC SX-8 vector processor. In our experience, as shown in Sect. 2.1, only about one third of the theoretical bandwidth can actually be measured.

Recently, Sun Microsystems has released the UltraSPARC T2+ eight-core processor. With its four built-in coherence links (6.4 GB/s each per direction), it is designed to be used in four- and two-socket nodes (see Fig. 2 for a schematic layout of the latter). Due to the ccNUMA access characteristics, care must be taken to employ proper first-touch page placement in shared-memory par-

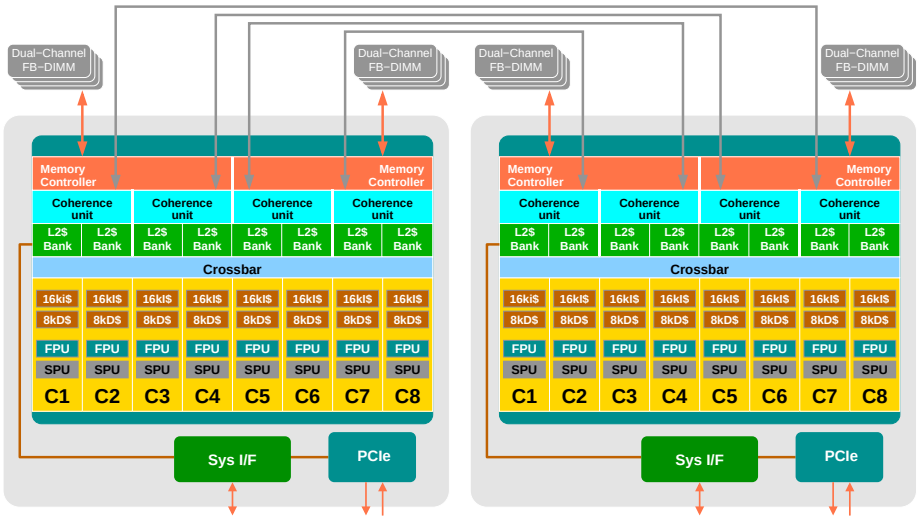

Figure 2: Architecture of a Sun UltraSPARC T2+ system with eight cores per socket and two ccNUMA locality domains. Each of the four coherence links can transport up to 6.4 GB/s per direction.

allel codes and the system must be configured accordingly[1]. As each processor chip has only two built-in memory controllers, the overall theoretical memory bandwidth in a two-socket system is the same as in a single-socket T2 configuration. For purely memory-bound code like the low-level STREAM benchmark one would thus expect only minor improvements due to the doubled number of outstanding misses. This will be discussed below. Other characteristics like peak performance or outstanding misses per core are the same as for T2.

Beyond the requirements of the tests presented here one should be aware that the T2 chip also comprises on-chip PCIe-x8 and 10 Gb Ethernet connections as well as a cryptographic coprocessor. These features are reminiscent of the actual concept of the chip: It is geared towards commercial, database and typical server workloads. Consequently, one should not expect future versions to improve on HPC-relevant weaknesses of its design. The T2+ variant lacks the on-chip Ethernet hardware due to the additional space requirements for coherence logic.

2 Benchmarks and optimizations

This section describes the benchmarks that were used to pinpoint aliasing effects, performance results and optimization techniques. Measurements were performed on a Sun SPARC Enterprise T5120 system at RRZE and a pre-production SPARC Enterprise T5240 at Sun Microsystems, both equipped with PC2-5300 FBDIMM modules and running at 1.2 GHz. The latter system uses a T2+ processor in release 1.1; see Sect. 2.1.2 for a discussion.

The pre-production server was installed shortly before the official release of

[1]This means that 1 GB interleaving should be used so that successive 1 GB chunks of physical memory addresses are assigned to alternating locality domains. Whenever a physical page gets mapped to a logical address, it is taken from the pool in the local domain, if possible.

the platform. Results may change slightly on production hardware, but we do not expect any major changes.

2.1 McCalpin STREAM

The STREAM benchmark [3] is a widely used code to assess the memory bandwidth capabilities of a single processor or shared memory computer system. It performs OpenMP-parallel copy (`C(:)=A(:)`), scale (`B(:)=s*C(:)`), add (`C(:)=A(:)+B(:)`) and triad (`A(:)=B(:)+s*C(:)`) operations on double precision (DP) vectors `A`, `B`, and `C` at an array length that is large compared to all cache sizes. The standard Fortran code allows some variations as to how the data is allocated. If the arrays are put into a COMMON block, a configurable offset ("padding") can be inserted so that their base addresses vary with the offset in a defined way:

```
PARAMETER (N=20000000,offset=0, &
           ndim=N+offset,ntimes=10)
DOUBLE PRECISION a(ndim),b(ndim),c(ndim)
COMMON a,b,c
```

Performance results are reported as bandwidth numbers (GB/s). The required cache line read for ownership (RFO) on the store stream is not counted, so the actual data transfer bandwidth for, e.g., STREAM triad (copy) is a factor of 4/3 (3/2) larger than the reported number (some architectures provide means to bypass the cache on write misses or claim ownership of a cache line without a prior read).

Fig. 3 (lower panel) shows STREAM triad performance on 8, 16, 32 and 64 threads for T2 as well as 128-thread data for T2+ versus the offset parameter at an array size of $N = 2^{25}$.

2.1.1 STREAM on the T2

On the T2 there is a striking periodicity of 64 for 16 threads and above, and the 32 and 64 thread data shows an additional, albeit weaker variation with a period of 32. For this simple bandwidth-bound benchmark it does not seem possible to draw advantage from the T2's large memory bandwidth. The reasons for this shortcoming are as yet unclear; the processor does definitely not suffer from a lack of outstanding references as peak bandwidth does not change when going from 32 to 64 threads. It has been shown, however, that kernels which are almost exclusively dominated by loads can achieve somewhat larger bandwidths [4], which leads to the conclusion that at least part of the problem is caused by overhead for bidirectional transfers. This conjecture is substantiated by the significantly lower STREAM copy performance (upper panel in Fig. 3).

Performance starts off on a very low level at zero offset and returns to the same level at an offset of 64 which corresponds to 512 bytes. Considering that the array length is a power of two, the 512-byte periodicity reflects perfectly the mapping between memory addresses and memory controllers which is based on bits 8 and 7. Therefore, the starting addresses of arrays `A`, `B`, and `C` are mapped to the same memory controller if the offset is zero or a multiple of 64 DP words. This is even true for each single OpenMP chunk, which means that all threads hit exactly one memory controller at a time. As the loop count

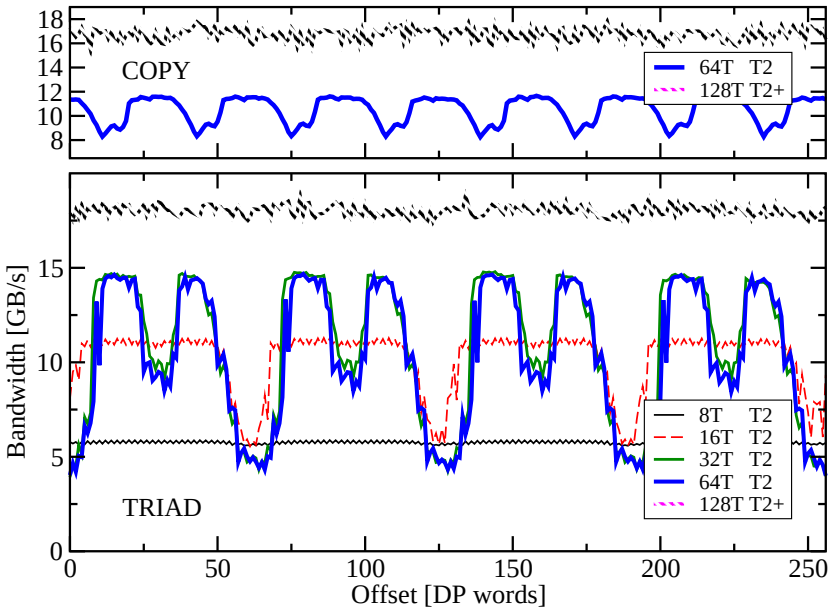

Figure 3: Lower panel: Parallel STREAM triad bandwidth at $N = 2^{25}$ and static OpenMP scheduling (no chunksize) for thread counts between 8 and 64 on T2 and 128 on T2+ versus array offset. Upper panel: STREAM copy bandwidth for 64 (128) threads on T2 (T2+). Threads were distributed equidistantly across cores.

proceeds, successive controllers are of course used in turn, but not concurrently. At odd multiples of 32, the situation is improved because bit 8 is different for array `B`'s base and thus two controllers are addressed, leading to an expected performance improvement of 100 %. The fact that 16 threads seem to suffer less under such conditions might be attributed to congestion effects.

Finally, at "skewed" offsets the addresses of different streams in one thread and also between threads ensure a rather uniform utilization of all four memory controllers. Surprisingly, this condition seems to hold in an optimal way for only about half of all offsets.

2.1.2 STREAM on the T2+

The STREAM performance data for T2+ displays a completely different behaviour (128-thread data in Fig. 3). For release 1.1 of this processor, Sun Microsystems decided to change the prioritization of read vs. write accesses within the memory interface which led to different access characteristics in comparison to T2 [2], but we have confirmed (using low-level code which uses no write operations at all) that this change has no influence on alignment issues. Nevertheless, the performance variations with varying array offset have since vanished (apart from minor noise). One must emphasize, however, that all currently available systems using the T2 still show the effect, and we are not aware of any plans to release an update similar to the T2+. In what follows, it will consequently be regarded as a major optimization issue. For the T2+ we will present performance

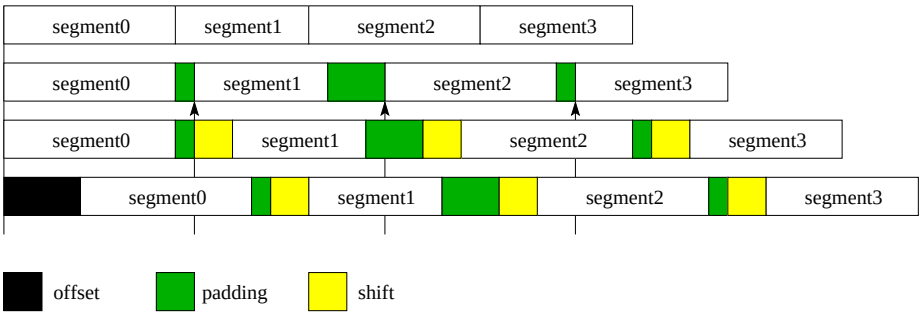

Figure 4: Parameters for alignments and offsets of array segments in the `seg_array` data structure.

data without any special alignment provisions, unless otherwise noted.

Comparing maximum achievable performance, the two-socket T2+ system shows between 25 % (for triad) and 50 % (for copy) bandwidth improvement versus T2. Also, taking the real data transfer due to the cache line RFO into account one must conclude that the bandwidth advantage for the read-dominated triad has disappeared; STREAM copy is slightly faster. The reasons for this change are unknown as of now.

2.2 Vector triad

One could argue that using an array length of 2^{25} and powers of two for thread counts on STREAM measurements are bound to provoke aliasing conflicts. Even on the T2+, which does not suffer from aliasing problems on memory access, the ccNUMA system architecture forces the programmer to employ correct first-touch memory placement. In this case it is useful to know the characteristics of the coherent links between the sockets for judging the influence of non-local accesses. In order to fathom aliasing and locality effects on both architectures, we turn to a more flexible framework for bandwidth assessment and use a self-developed vector triad code. In the subsequent sections, a 2D relaxation solver and finally an implementation of the lattice-Boltzmann algorithm will be used to implement the developed optimizations in a more application-centered setting.

The vector triad is quite similar to the STREAM triad benchmark but features three instead of two read streams (`A(:)=B(:)+C(:)*D(:)`)[5]. We have developed a flexible C++ framework in which all arrays and also OpenMP chunks can be aligned on definite address boundaries and then shifted by configurable amounts (see Fig. 4). The array base is aligned to some boundary by allocating memory using the standard `posix_memalign()` libc function (leftmost border in Fig. 4). The data is then divided into segments, not necessarily of equal size, and padding is inserted in order to align each segment except the first to another specific boundary (arrows in Fig. 4). After that, a constant amount of additional padding ("shift") is added before each segment and finally the whole data block is shifted by some offset. Thereby it is, e.g., possible to align an array to a memory page and then shift a segment that would be assigned to thread t by $t \cdot 128$ bytes.

Although the segmented data structure can be equipped with a standard

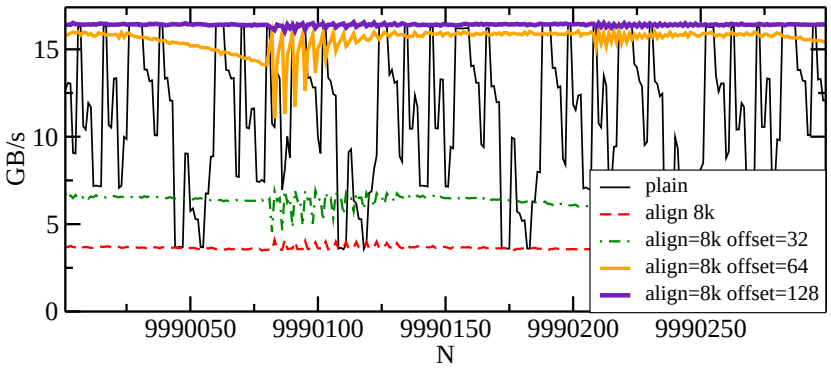

Figure 5: Vector triad performance on T2 (64 threads) vs. array length for plain arrays with no restrictions, 8 kB alignment for all arrays and different byte offsets for array base addresses (arrays `B`, `C` and `D` are shifted by one, two, and three times the indicated offset, respectively).

bidirectional iterator, its use is discouraged in loop kernels because of the required conditional branches in, e.g., `operator++()`. Instead, low-level loops are handled by a C++ programming technique called *segmented iterators* [6, 7] which enables the design of STL-style generic algorithms with performance characteristics equivalent to standard C or Fortran versions. OpenMP parallelization directives are applied to the loop over all segments, and a separate function is called to handle a single segment:

```
seg_array a,b,c,d; // parameters omitted
...
typedef seg_array::iterator it;
typedef seg_array::local_iterator lit;
typedef seg_array::segment_iterator sit;
sit ai = a.begin().segment();
// ... same for bi, ci, di
#pragma omp parallel for schedule(...)
for(int s=0; s < N_SEGMENTS; s++) {
  lit alb = (ai+s)->begin();
  lit ale = (ai+s)->end();
  lit blb = (bi+s)->begin();
...
  triad(alb, blb, clb, dlb, ale);
}
```

The `triad()` function performs the actual low-level array operations and is actually a generic dispatching algorithm that can handle both segmented and local iterators. Details about the template mechanism and its general use for high performance kernels are omitted for brevity and will be published elsewhere [8].

Although C++ is used for administrative purposes, the (purely serial) inner benchmark kernel can be written in C or Fortran and even compiled separately without OpenMP, so as to produce the possibly most efficient machine code. In our implementation of the segmented triad we choose the number of segments

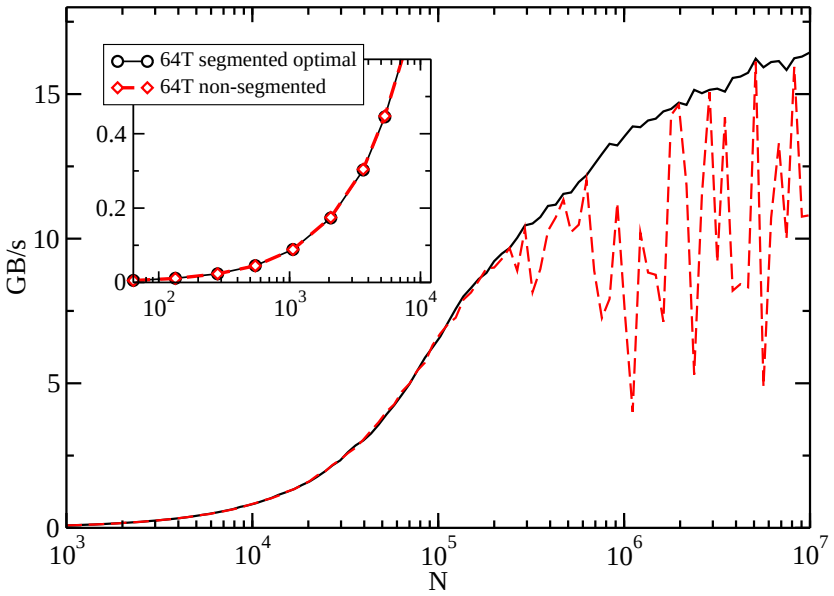

Figure 6: Performance overhead of segmented iterators vs. plain OpenMP on T2 (64 threads). For clarity, not all values of N were scanned. Inset: Enlarged small-N region.

equal to the number of OpenMP threads and do manual scheduling with segment sizes of $\lfloor N/t \rfloor + 1$ and $\lfloor N/t \rfloor$, respectively. The segmented array class constructor takes care of optimal ccNUMA placement as well by employing parallel first touch on the allocated memory segments. It is even possible to use the container for objects with dynamic content by utilizing *placement new* in a parallel loop across all container objects.

2.2.1 Vector triad and aliasing conflicts on T2

For the T2 processor, Fig. 5 shows vector triad performance in GB/s versus array length, using different alignment constraints. The interval on the N axis was chosen to clearly show essential features without superimposed small-N startup effects. In the "plain" case, no special arrangements were made and arrays were allocated as continuous blocks using `malloc()`. This results in very erratic performance behaviour with a periodicity of 64 DP words, showing "hard" upper and lower limits at roughly 16 and 3.7 GB/s, respectively. Aligning all arrays to page boundaries (8 kB), one can force an especially bad situation that corresponds to the zero offset case for the STREAM triad (bottom line). On the other hand, by choosing suitable offsets for `B`, `C`, and `D` (128, 256 and 384 bytes, respectively, in the optimal case), one can achieve a nearly perfectly balanced utilization of all four memory controllers that causes no breakdowns at all (top line). In this case it is not even required to use padding and shifts (see Fig. 4) for the segments as the large number of streams (three for reading, one for writing) ensures that even the single thread features optimal access patterns if the offset is chosen correctly.

Although of minor importance here, padding to 16-byte boundaries can

greatly improve performance of memory-bound kernels on x86 architectures. Current x86 designs like AMD Opteron and Intel Xeon (Netburst as well as Core2 architectures) feature a *non-temporal store* in the SSE2 instruction set extension that writes a complete 16-byte wide SSE register directly to memory without the need for RFO on a write miss ("cache bypass"). Using this instruction in a memory-bound streaming loop kernel leads to improved performance, but the store addresses must all be multiples of 16[2]. In cases where it is not possible to achieve this simply by choosing an appropriate OpenMP chunk size, manual alignment of OpenMP chunks can be used. Moreover, aliasing effects as described above for the UltraSPARC T2 processor can also be observed on standard x86 systems, albeit at a much lesser extent [8].

The performance overhead incurred by segmented iterators is negligible even for tight loops like the vector triad. Fig. 6 shows a comparison between plain OpenMP and segmented triad performance with 64 threads. Optimal alignment was chosen in the latter case. Obviously, the cost of starting a parallel region is clearly dominating, which is not surprising since the data distribution (segmentation, first touch placement) has already been completed on array allocation and the segmented iterators have been designed to show pointer performance in inner kernels. See the next section for a discussion of OpenMP overhead in the multi-socket T2+ configuration.

2.2.2 Vector triad performance and ccNUMA characteristics on T2+

Like any other ccNUMA-type system, the two-socket T2+ node (Fig. 2) shows the typical characteristics of such architectures: Non-local accesses across the coherent links suffer from bandwidth and latency penalties, and severe memory bus contention can occur if pages are accessed by two sockets concurrently. Although correct page placement by first touch (or other means like system-dependent libraries) can ameliorate such effects in many cases, this is not always possible and it is important to know the characteristics of the system under "unfortunate" conditions. Fig. 7 shows results for the vector triad with 64 and 128 threads on the T2+ (T2 data with optimal alignment and 64 threads is included for reference). The following observations are worth noting:

- With 64 threads on a single socket, T2+ is able to achieve about 70 % of the saturation bandwidth on T2, although the latter features twice the number of memory controllers. As shown previously, this cannot be a consequence of the increased number of outstanding misses but must be attributed to other optimizations in the memory interface.

- Unfortunately, saturation bandwidth does not scale perfectly across locality domains, and there are strong fluctuations whenever both domains are used. This leads to a mere 20 %–25 % improvement in overall system bandwidth compared to T2. The fluctuations cannot be removed by any choice of alignment or padding and will also be present on more high-level codes (see below). However, data that was obtained on a single socket always shows very smooth behaviour. See Sect. 2.3.2 for further discussion.

[2]The latest AMD Opteron implementation called "Barcelona" and its relatives even provide a scalar non-temporal store instruction, but this has not been adopted by Intel so far.

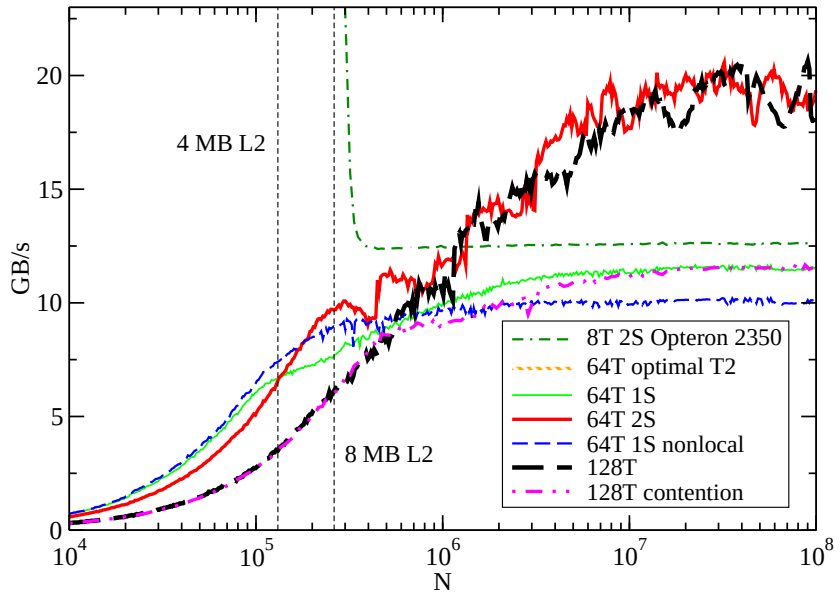

Figure 7: OpenMP startup overhead and locality/contention effects on UltraSPARC T2+, demonstrated with the vector triad. In the 64 threads case, data on one and two sockets (1S/2S) is given for optimal page placement, and one-socket data for purely non-local access. For 128 threads, data for optimal placement and pure contention (all pages in one locality domain) is shown. 64-thread data on T2 with optimal alignment and 8-thread data for a two-socket AMD Opteron system (see text for details) is included for reference.

- Running 64 threads on one socket while all memory pages are placed into the remote locality domain ("nonlocal" data in Fig. 7) shows a 20 % bandwidth penalty for large N. Not surprisingly, when using 128 threads on the same data performance saturates at the same level. This shows, as in any other ccNUMA architecture, that a fast inter-domain network alone is not sufficient to avoid memory bus contention effects.

- Generally, the L2 cache is too small to have any significant impact on small-N performance for both systems (4 MB and 8 MB cache size limits are shown for reference in Fig. 7). The reason is that OpenMP parallel region startup overhead dominates for small N. Comparing the 64-thread data on two sockets (2S) with 128 threads we conclude that adding more threads adds to startup overhead proportionally. Moreover, startup is considerably faster when all threads in a team are located on the same socket (see 64-thread data on one socket vs. two sockets). This is completely in line with experience from standard x86-based multi-core multi-socket systems.

For reference, vector triad performance for eight threads on a two-socket AMD Opteron 2350 system (2 GHz "Barcelona" quad-cores, 512 kB L2 per core, 2 MB shared L3 per socket) is also shown in Fig. 7. Note that non-temporal stores were not used here and would further improve memory bandwidth by about 20 %

for large N. At small N (in cache), eight Opteron cores achieve an aggregated triad bandwidth of > 90 GB/s.

Although those low-level benchmarks show that memory performance can be quite competitive on T2 and T2+ systems, they also reveal one of the weaknesses of trading single-core performance for massive thread parallelism: OpenMP startup overhead becomes an important factor to consider if the number of threads gets large. Dynamic thread number adjustment should thus be considered but bears the potential for new complexities due to the required modifications to thread/process pinning (even more so on the ccNUMA-based T2+ nodes).

2.3 2D relaxation solver

For the vector triad, solving the aliasing problems on the UltraSPARC T2 processor merely required a correct choice of the offset between the four different data streams. There are cases, however, where this will not suffice. This happens, e.g., when the number of concurrent load/store streams is not large enough to address all memory controllers concurrently with a single thread. As an example and an intermediate step towards more complex applications we consider a simple 2D Jacobian heat equation solver using a five-point stencil on a quadratic $N \times N$ domain:

```
#pragma omp parallel for schedule(...)
for(int i=1; i < N-1; i++) {
  for(int j=1; j < N-1; j++)
    dest[i][j] = (source[i-1][j]
                  + source[i+1][j]
                  + source[i][j-1]
                  + source[i][j+1])*0.25;
}
```

With four loads, one store and four floating-point operations, this kernel has an application balance (ratio of bytes loaded or stored vs. flops) of 10 bytes/flop, much smaller than the vector triad from the previous section (16 bytes/flop). However, three of the four source operands needed at the current index can be obtained from cache or registers, given that the amount of cache available per thread is large enough to accommodate at least two successive rows. If this condition is fulfilled, the actual data transfer to and from memory amounts to only 4 bytes/flop (6 bytes/flop with RFO). Comparing with the achievable T2 STREAM copy bandwidth (Fig. 3) of roughly 18 GB/s (including RFO) one should expect a performance of about 3 GF/s, which corresponds to 750 million lattice site updates per second (MLUPs/s).

Implementing the segmented iterator technique is straightforward and is only used here to enforce the desired alignment constraints: Each source and destination row is a separate segment which is subject to the alignment options described in Sect. 2.2. The parallel OpenMP loop runs over rows so that scheduling can be done in the standard way. The low-level kernel is parametrized with iterators pointing to the three current source rows and the destination row (any template syntax is again omitted):

```
typedef seg_array::iterator it;
```

```
typedef seg_array::local_iterator lit;
typedef seg_array::segment_iterator sit;
sit si = source.begin().segment();
sit di = dest.begin().segment();
#pragma omp parallel for schedule(...)
for(int i=1; i < N-1; i++) {
  lit dl = (di + i)->begin();
  lit sa = (si+i-1)->begin();
  lit sb = (si+i-1)->begin();
  lit sl = (si + i)->begin();
  relax_line(dl, sa, sb, sl, N);
}
```

The `relax_line()` function is again purely serial:

```
void relax_line(lit &dl, lit &sa,
lit &sb, lit &sl, int N){
  for(int j=1; j < N-1; j++)
    dl[j] = (sa[j] + sb[j]
           + sl[j-1] + sl[j+1])*0.25;
}
```

In a 3D formulation, two additional arguments (rows) to `relax_line()` would be required. The number of segments equals the number of rows N and is hence not directly connected to the number of threads t.

2.3.1 Relaxation solver and alignment optimization on T2

Fig. 8 shows performance results on UltraSPARC T2 in MLUPs/s for up to 64 threads using the most optimal set of alignment parameters:

- Each segment, i.e. each row, is aligned to a 512 byte boundary using appropriate padding.
- By using shift=128, the base addresses of successive segments are shifted versus each other so as to address different controllers.
- As destination row i can only be updated by reading source row $i+1$ first, there is a natural offset between read and write streams and no further provisions are required to make sure that they address different controllers if shift=128.
- An OpenMP schedule of "static,1" (round robin) has to be used for optimal performance. This is because the 4 MB L2 cache of the processor is too small to accommodate a sufficient number of rows when using 64 threads if the addresses are too far apart, i.e. if the domain is too large (see also the comparison with T2+ below for some more discussion regarding this point). Certainly, this problem could easily be resolved by employing spatial blocking.

Note that these parameters are the same for all problem sizes and can be obtained by analyzing the data access properties of the loop kernel, together

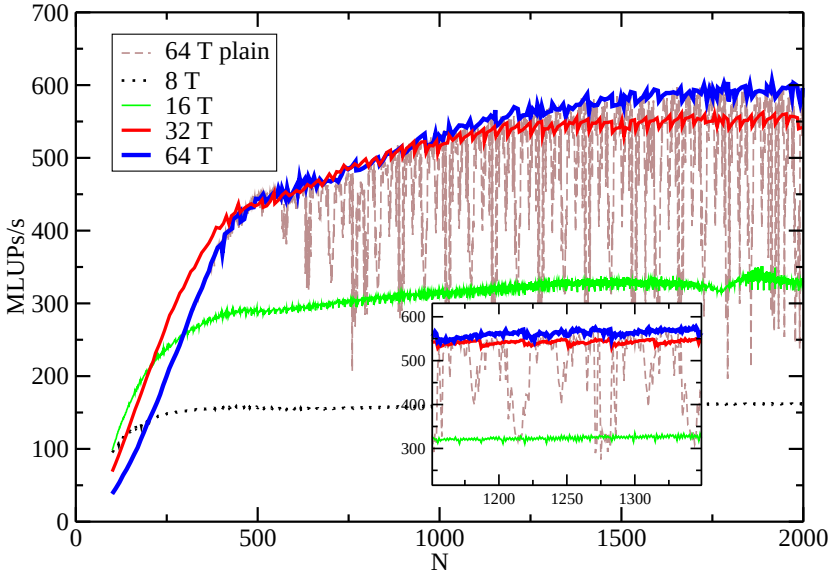

Figure 8: Performance and scaling of 2D heat equation relaxation solver versus problem size with optimal alignment and "static,1" scheduling on UltraSPARC T2. See text for parameters. "Plain" data with no alignment optimizations is shown for reference.

with some knowledge about the mapping between addresses and memory controllers. No "trial and error" is required. The maximum performance of about 600 MLUPs/s is just 20 % below expectations from STREAM copy bandwidth.

For reference, 64-thread data with no optimizations is included. The typical periodicity of 64 or 32 versus N is clearly visible in the latter case. The residual "jitter" on the optimized data, especially for large thread counts, is due to the number of rows not being a multiple of the number of threads. This effect can be expected to become more pronounced in the 3D case and will be discussed in Sect. 2.4 on lattice-Boltzmann.

2.3.2 Relaxation solver on T2+

Fig. 9 shows a performance comparison between T2 (using optimal alignment as described above) and T2+ for a single socket using 64 threads in all cases. The characteristic performance drop on T2 at $N \approx 6000$ for static,1 and $N \approx 3000$ for static scheduling results from the L2 cache being too small to hold two successive rows of the read stream, as mentioned above. However, the T2+ data for static,1 scheduling does not show this drop at all. One may thus conjecture that there has been some change in the T2+ cache organization, but this information is not available from Sun.

In Fig. 10 we present two-socket T2+ performance data. Obviously, the small cache size per thread starts to show very early when all 128 threads are used, so that 64 threads with static scheduling are best to use at small to intermediate problem sizes. For $N \gtrsim 8000$, however, 64- and 128-thread performance with static scheduling coincide, as could be expected from the low-level bandwidth

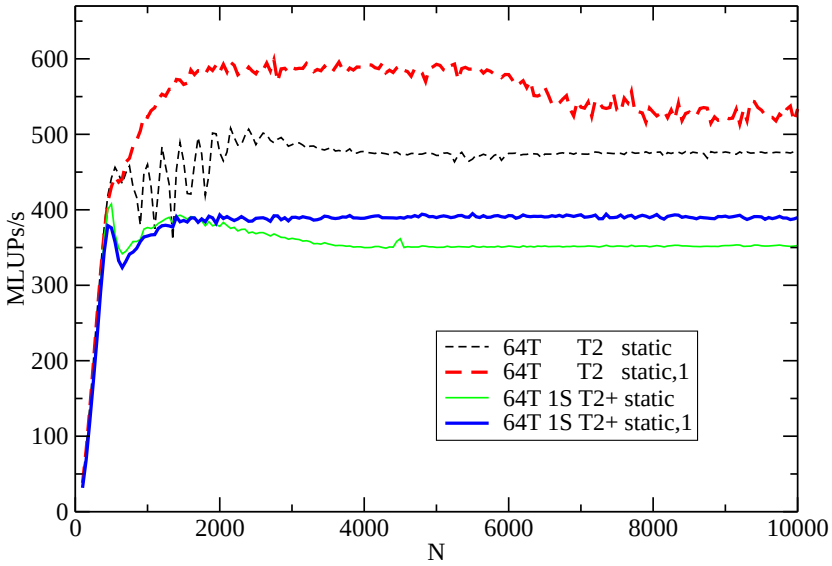

Figure 9: Single socket Performance comparison between UltraSPARC T2 (optimal alignment, dashed) and T2+ (solid) for the relaxation solver benchmark.

measurements shown earlier.

Interestingly, in contrast to the situation on a single socket, static scheduling yields better performance for intermediate N. In order to identify a possible reason for this, we removed the write operation from the relaxation iteration. The result is shown in the inset of Fig. 10 and confirms that the mediocre performance, strong fluctuations and characteristic "jumps" up to $N \approx 8000$ are a consequence of write traffic. Whether the corresponding increase in coherence (snoop) activity or ccNUMA boundary effects are responsible for the performance characteristics can not be answered as of today; a more thorough investigation, possibly using hardware performance counters, will be conducted. It is also worth noting that at large $N \gtrsim 8000$, round robin scheduling on 128 threads performs best.

2.4 Lattice-Boltzmann algorithm (LBM)

The advantage of using a special data structure to address alignment problems is its generality and applicability to non-regular problems (e.g., segments of different size). It is, however, in some cases possible to circumvent aliasing effects just by choosing the right data layout. As an example we consider a lattice-Boltzmann benchmark that has been developed out of a production code in order to study various optimizations [9]. For these tests we use a 3D model with 19 distribution functions (D3Q19) on a cubic domain with two disjoint grids (or "toggle arrays"). There is a choice as to which data layout to employ for the cartesian array holding the distribution functions. On cache-based architectures the propagation-optimized "IJKv" data layout, often referred to as "structure of arrays", is usually the best choice where I, J and K are cartesian coordinates and v denotes the distribution function index. The computational kernel using

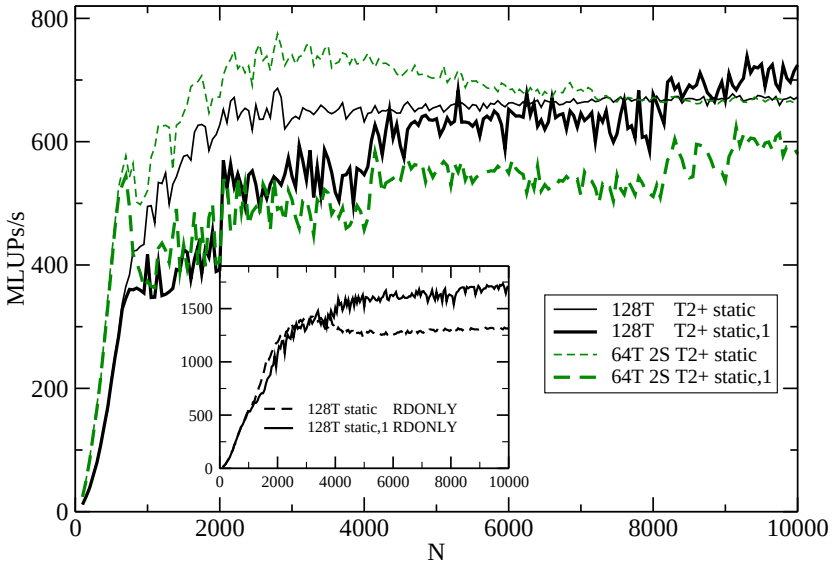

Figure 10: Relaxation solver performance on UltraSPARC T2 on 64 (2 sockets) and 128 threads with OpenMP static and static,1 scheduling,respectively. Inset: Synthetic benchmark with write operations removed.

this layout is sketched in Fig. 11. Evidently, the 19 read and 19 write streams are traversed with unit stride in this case.

2.4.1 LBM on the T2

Judging from the achievable STREAM copy memory bandwidth (≈18 GB/s including RFO) and the required load/store traffic for a single lattice site update (456 bytes including RFO), one would expect an LBM performance of roughly 40 MLUPs/s . These kinds of estimates usually give good approximations for standard multi-core architectures [10] if the kernel is really memory-bound.

Fig. 12 shows performance results in MLUPs/s for LBM on a cubic domain of extent N^3 for the standard IJKv layout as well as for an alternative IvJK layout. Obviously the latter choice yields twice the performance than IJKv and also smoother behaviour over a wide range of domain sizes. As the loop nest is parallelized on the outer level, the fortunate number of 19 distribution functions leads to an automatic skew between streams when doing the 19 neighbour updates. The large number of concurrent stride-1 write streams is of course instrumental in achieving this effect.

There are two residual peculiarities worth noting. First, if the 1D domain size is a multiple of 64 (minus two boundary layers), the well-known cache thrashing effects are ruinous. This could be eliminated by padding the first array dimension. Second, the sawtooth-like performance pattern is a "modulo effect" which emerges from N not being a multiple of the number of threads. A simple way to remove the pattern is to coalesce several outer loop levels in order to lengthen the OpenMP parallel loop. Results for up to 64 threads and two-way coalescing are also shown in Fig. 12 and corroborate the call for extensions of the OpenMP standard towards more flexible options for parallel execution of

```
real*8 f(0:N+1,0:N+1,0:N+1,0:18,0:1)
logical fluidCell(1:N,1:N,1:N)
real*8 dens, ne, ...
!$OMP PARALLEL DO PRIVATE(...)
do z=1,N
 do y=1,N; do x=1,N
  if ( fluidCell(x,y,z) )  then
   ! read distributions from local cell
   ! and  calculate moments
   dens=f(x,y,z,0,t)+f(x,y,z,1,t)+ &
        f(x,y,z,2,t)+...
   ...
   ! compute non-equilibrium parts
   ne0=...
   ...
   ! write updates to neighbouring cells
   f(x  ,y  ,z  , 0,tN)=f(x,y,z, 0,t)*...
   f(x+1,y+1,z  , 1,tN)=f(x,y,z, 1,t)*...
   ...
   f(x  ,y-1,z-1,18,tN)=f(x,y,z,18,t)*...
  endif
 enddo; enddo
enddo
!$OMP END PARALLEL DO
```

Figure 11: Computational kernel for the IJKv layout D3Q19 LBM.

loop nests. Luckily, the recently adopted OpenMP 3.0 standard provides basic support for this.

However, even when these optimizations are employed, the system falls short of the performance expectations derived from STREAM by a factor of 1.5. As for the reason one may speculate that the T2's arithmetic units are a limiting factor due to the rather low code balance of LBM of ≈ 2.5 bytes/flop, so that the code is not memory-bound on this processor. This conclusion is supported by the observation that LBM performance does not change if the benchmark is carried out in single precision (the SPARC core's peak performance is identical for single and double precision). More cores or a larger peak performance per core should thus improve the results. See the next section for details.

Interestingly, comparing 32- and 64-thread performance in Figs. 3, 8 and 12 we conclude that the smaller the application balance in bytes/flop the larger the gain when using 64 instead of 32 threads. This is contrary to expectations as strongly memory-bound kernels should benefit from a larger number of outstanding references.

2.4.2 LBM on the T2+

In Fig. 13 we compare the best possible LBM variant on T2 (64 threads, lower graph) with the same code on T2+, using 128 threads. Performance saturates at ≈43 MLUPs/s, a 65 % boost versus T2. This is much more than could be ex-

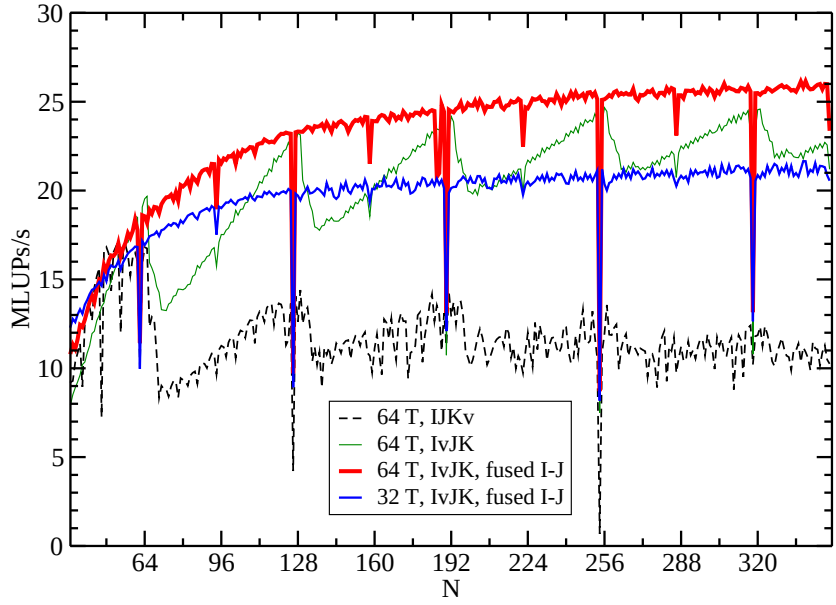

Figure 12: LBM performance versus domain size (cubic) at up to 64 threads on UltraSPARC T2 using different data layouts and scheduling methodologies. The "modulo variation" can be eliminated by coalescing the outer loop pair (top curve).

pected from the STREAM triad or even STREAM copy comparisons presented in Sect. 2.1.2. In the previous section it was speculated that the D3Q19 lattice-Boltzmann algorithm is not limited by memory bandwidth on the T2. The large improvement seen on T2+ strongly supports this conjecture. The UltraSPARC T2 processor is the only microprocessor on which the D3Q19 LBM is compute bound, a feature which it shares exclusively with vector machines like the NEC SX-8.

3 Conclusions

We have analyzed the performance of the Sun UltraSPARC T2 and T2+ multi-core processors using low-level and application benchmarks. Aliasing conflicts when accessing memory on T2 could be attributed to the simple mapping of memory controllers to physical addresses. Consequently, bandwidth-intensive code tends to show large performance fluctuations with respect to problem size. Using explicit alignment and padding techniques we were able to remedy aliasing conflicts for a simple vector triad benchmark and a 2D Jacobi heat equation solver. For a D3Q19 lattice-Boltzmann algorithm we could show that an appropriate choice of data layout removes most of the aliasing. Comparing a single-socket T2 system to a dual-socket T2+ node we have demonstrated that most of the aliasing problems have vanished at the price of a doubled number of threads and the typical ccNUMA performance features. On both architectures — but especially for T2+ — OpenMP startup overhead can play a dominant role at small problem sizes due to the large thread numbers. At large prob-

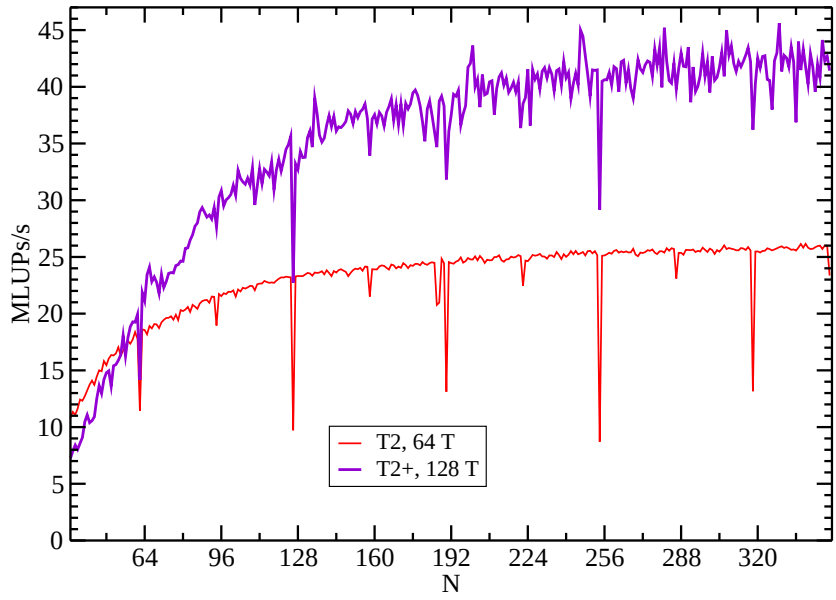

Figure 13: Comparison of best LBM performance on UltraSPARC T2 (same data as in Fig. 12) versus T2+. There is significant performance gain from using twice the number of cores at the same theoretical memory bandwidth.

lem sizes the small amount of L2 cache available per thread will make spatial blocking optimizations mandatory for many stencil-based applications. Moreover, the two-socket T2+ system shows performance peculiarities not related to aliasing conflicts whenever there is write traffic on both memory domains. Finally we demonstrated that a D3Q19 lattice-Boltzmann flow solver is limited by compute performance on T2 and by memory bandwidth on T2+.

We believe all demonstrated performance features and optimizations to be very relevant on large-scale systems because predictable one-node performance is essential for getting good parallel efficiency.

Finally one must emphasize that in the light of upcoming massively multi-core, multi-threaded designs, the rigid and only slowly evolving OpenMP programming model might not be the ultimate solution for shared-memory parallel programming in the future. More "lightweight" paradigms like, e.g., Threading Building Blocks (TBB) [11], could provide a promising alternative.

Acknowledgements

We are indebted to Sun Microsystems for providing access to an UltraSPARC T2+ system. We also wish to thank Rick Hetherington, Denis Sheahan, Sumti Jairath, Constantin Gonzalez and Ram Kunda from Sun Microsystems, and Samuel Williams from UCB for valuable discussions.

References

[1] Sun Microsystems: *OpenSPARC T2 Core Microarchitecture Specification.* `http://opensparc-t2.sunsource.net/specs/OpenSPARCT2_Core_Micro_Arch.pdf`

[2] Sun Microsystems, private communication.

[3] `http://www.cs.virginia.edu/stream/`

[4] S. W. Williams, L. Oliker, R. Vuduc, K. Yelick, J. Demmel and J. Shalf: *Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms.* Proceedings of SC07, Reno, Nevada, Nov. 10–16, 2007.

[5] W. Schönauer: *Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers.* Self-edition, Karlsruhe (2000), `http://www.rz.uni-karlsruhe.de/~rx03/book`

[6] M. H. Austern: *Segmented Iterators and Hierarchical Algorithms.* In M. Jazayeri, R. G. K. Loos, and D. R. Musser (ed.), *Generic programming: International Seminar on Generic Programming,* Dagstuhl Castle, Germany, Apr 28 – May 1, 1998, Selected Papers, Springer, 2000. Series: Lecture Notes in Computer Science, Vol. 1766. ISBN: 978-3-540-41090-4

[7] H. Stengel: *C++ programming techniques for High Performance Computing on systems with non-uniform memory access using OpenMP.* Diploma thesis, University of Applied Sciences Nuremberg, 2007 (in German). `http://www.hpc.rrze.uni-erlangen.de/Projekte/numa.shtml`

[8] G. Hager et al.: *Using segmented iterators for locality and alignment optimizations on current cache-based architectures.* In preparation.

[9] G. Wellein, T. Zeiser, S. Donath and G. Hager: *On the Single Processor Performance of Simple Lattice Boltzmann Kernels.* Computers & Fluids **35**, 910–919, 2006.

[10] S. Donath, K. Iglberger, G. Wellein, T. Zeiser, A. Nitsure, U. Rüde: *Performance comparison of different parallel lattice Boltzmann implementations on multi-core multi-socket systems.* Accepted for publication in Int. J. Comp. Sci. Eng., 2007.

[11] J. Reinders: *Intel Threading Building Blocks.* O'Reilly, 2007. ISBN-13: 978-0-596-51480-8