J Supercomput (2007) 39: 93–112
DOI 10.1007/s11227-007-0111-y

The memory behavior of cache oblivious stencil computations

Matteo Frigo · Volker Strumpen

Published online: 21 February 2007
© Springer Science+Business Media, LLC 2007

Abstract We present and evaluate a cache oblivious algorithm for stencil computations, which arise for example in finite-difference methods. Our algorithm applies to arbitrary stencils in n-dimensional spaces. On an "ideal cache" of size Z, our algorithm saves a factor of $\Theta(Z^{1/n})$ cache misses compared to a naive algorithm, and it exploits temporal locality optimally throughout the entire memory hierarchy. We evaluate our algorithm in terms of the number of cache misses, and demonstrate that the memory behavior agrees with our theoretical predictions. Our experimental evaluation is based on a finite-difference solution of a heat diffusion problem, as well as a Gauss-Seidel iteration and a 2-dimensional LBMHD program, both reformulated as cache oblivious stencil computations.

Keywords Cache oblivious algorithms · Stencil computations · Analysis of algorithms · Performance analysis · System simulation

1 Introduction

The goal of *cache oblivious algorithms* [14] is to use a memory hierarchy effectively without knowing parameters such as the number of cache levels and the size of each cache. Well-designed cache oblivious algorithms incur the minimum number of cache misses within each level of a memory hierarchy, and can deliver high performance across machines with different memory systems. Thus, cache oblivious algorithms are portable in the sense that they can deliver high performance without requiring machine-specific parameters to be tuned.

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

M. Frigo · V. Strumpen (✉)
IBM Austin Research Laboratory, 11501 Burnet Road, Austin, TX 78758, USA
e-mail: strumpen@us.ibm.com

We have recently developed [15] a cache oblivious algorithm for stencil computations in n-dimensional rectangular grids, for arbitrary n. At each time t, $0 \leq t < T$, a *stencil computation* updates a grid point based on the values of the point and some neighboring points to produce the value of the point at time $t + 1$. For sufficiently large spatial grids and time T, we proved [15] that our algorithm incurs $O(P/Z^{1/n})$ cache misses, where P is the total number of spacetime points computed and Z is the cache size, assuming an "ideal cache" (fully associative, optimal or LRU replacement policy) in which we ignore the effects of cache lines.

In this article, we recap our cache oblivious stencil algorithm, present specializations for $n = 1$ and $n = 2$ space dimensions as procedures written in the C programming language, and conduct an empirical study of the memory behavior. Specifically:

1. We compare the number of cache misses of a cache oblivious heat equation solver with its iterative counterpart on 1-, 2-, and 3-dimensional spatial grids, and on four different cache configurations. We show that our theoretical predictions are reasonably accurate even though real caches have limited associativity and large cache lines, and we discuss the behavior of the cache oblivious programs for "small" problems to which the theory does not apply.
2. We show that the cache oblivious stencil algorithm applies to the Gauss-Seidel iteration, even though Gauss-Seidel is not strictly a stencil computation because a point at time $t + 1$ depends upon some neighbors at time $t + 1$ (as opposed to t). Nevertheless, we show that the cache oblivious Gauss-Seidel algorithm behaves like a 1-dimensional stencil, as predicted by our theory.
3. To evaluate the end-to-end effect of cache oblivious algorithms on a more complex problem, we show that a cache oblivious implementation of the LBMHD (Lattice Boltzmann Magneto-Hydro-Dynamics) HPCS benchmark runs up to 4 times faster than a naive, iterative version on a Power4+ system.

Originally proposed in theoretical investigations, cache oblivious algorithms are now practically relevant as processor and memory speeds have diverged by orders of magnitude. In the past, cache aware algorithms, in particular *blocked algorithms*, have dominated the design of high-performance software, for example for linear algebra [3, 4, 6, 13, 16, 17]. Cache oblivious algorithms that incur the theoretically minimum number of cache misses exist for matrix multiplication [1, 14], FFT [2, 14], LU decomposition [9, 25], sorting [11, 14], and other problems [5, 7]. Toledo [25] experiments with a cache oblivious LU decomposition algorithm, and concludes that the cache oblivious algorithm is as good as any "cache aware" algorithm explicitly tuned to the memory hierarchy. Brodal and others [11] experiment with a variant of the cache oblivious funnelsort algorithm [14], and show that, for large problems, the cache oblivious algorithm is faster than an optimized implementation of quicksort on contemporary machines.

Cache oblivious algorithms for special cases of stencil computations have been proposed before. Bilardi and Preparata [8] discuss cache oblivious algorithms for the related problem of simulating large parallel machines on smaller machines in a spacetime-efficient manner. Their algorithms apply to 1-dimensional and 2-dimensional spaces and do not generalize to higher dimensions. In fact, the authors declare the 3-dimensional case, and implicitly higher dimensional spaces, to

be an open problem. Prokop [23] gives a cache oblivious stencil algorithm for a 3-point stencil in 1-dimensional space, and proves that the algorithm is optimal. His algorithm is restricted to square spacetime regions, and it does not extend to higher dimensions.

The remainder of this article is organized as follows. In Sect. 2, we summarize our cache oblivious stencil algorithm, and provide C programs for the 1-dimensional and 2-dimensional cases. We use an initial-value heat-diffusion problem to analyze the number of cache misses of 1-, 2-, and 3-dimensional stencil computations in Sect. 3. We present our cache oblivious Gauss-Seidel algorithm in Sect. 4. In Sect. 5 we analyze the cache oblivious formulation of a larger LBMHD program, and conclude in Sect. 6.

2 Cache oblivious stencil algorithm

In this section, we summarize our cache oblivious stencil algorithm [15], and we present C code for the 1- and 2-dimensional cases. The C code for the n-dimensional case with arbitrary n is listed in the Appendix.

Procedure `walk1` in Fig. 1 invokes procedure `kernel` on all spacetime points (t, x) in a certain region. Although we are ultimately interested in rectangular regions, procedure `walk1` operates on more general trapezoidal regions defined by the six integer parameters t_0, t_1, x_0, $\dot{x}_0$, x_1, and $\dot{x}_1$. Specifically, `walk1` visits all points (t, x) such that $t_0 \le t < t_1$ and $x_0 + \dot{x}_0(t - t_0) \le x < x_1 + \dot{x}_1(t - t_1)$. We define the ***trapezoid*** $\mathcal{T}(t_0, t_1, x_0, \dot{x}_0, x_1, \dot{x}_1)$ to be the set of integer pairs (t, x) such that $t_0 \le t < t_1$ and $x_0 + \dot{x}_0(t - t_0) \le x < x_1 + \dot{x}_1(t - t_0)$. (We use the Newtonian notation $\dot{x} = dx/dt$ to describe the slope of the sides of the trapezoid.) The ***height*** of the trapezoid is $\Delta t = t_1 - t_0$, and we define the ***width*** to be the average of the lengths of the two parallel sides, i.e. $w = (x_1 - x_0) + (\dot{x}_1 - \dot{x}_0)\Delta t/2$. The ***center*** of the trapezoid is point (t, x), where $t = (t_0 + t_1)/2$ and $x = (x_0 + x_1)/2 + (\dot{x}_0 + \dot{x}_1)\Delta t/4$, i.e. the average of the four corners.

```
void walk1(int t0, int t1, int x0, int ẋ0, int x1, int ẋ1)
{
    int Δt = t1 - t0;

    if (Δt == 1) {                                          /* base case */
        int x;
        for (x = x0; x < x1; ++x)
            kernel(t0, x);
    } else if (Δt > 1) {                                    /* recursion */
        if (2 * (x1 - x0) + (ẋ1 - ẋ0) * Δt >= 4 * ds * Δt) { /* space cut */
            int xm = (2 * (x0 + x1) + (2 * ds + ẋ0 + ẋ1) * Δt) / 4;
            walk1(t0, t1, x0, ẋ0, xm, -ds);
            walk1(t0, t1, xm, -ds, x1, ẋ1);
        } else {                                            /* time cut */
            int s = Δt / 2;
            walk1(t0, t0 + s, x0, ẋ0, x1, ẋ1);
            walk1(t0 + s, t1, x0 + ẋ0 * s, ẋ0, x1 + ẋ1 * s, ẋ1);
        }
    }
}
```

Fig. 1 Procedure `walk1` for traversing a 2-dimensional spacetime spanned by a 1-dimensional spatial grid and time

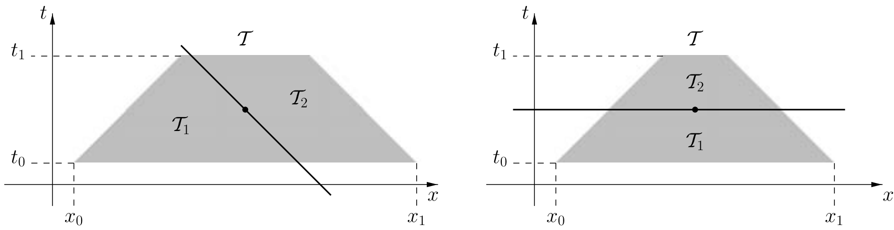

Fig. 2 Illustration of a space cut (left) and of a time cut (right). Procedure `walk1` cuts the trapezoid by means of lines through the trapezoid's *center*—the average of the coordinates of the four corners

Procedure `walk1` obeys this invariant: It visits point $(t+1, x)$ after visiting points $(t, x+k)$, for all k such that $-ds \le k \le ds$. Integer parameter $ds \ge 0$ is the ***stencil slope***, and is set by the user depending on the stencil shape. For example, in a 3-point stencil, spacetime point $(t+1, x)$ depends upon $(t, x-1)$, (t, x), and $(t, x+1)$, in which case ds should be set to 1. For a 5-point stencil where point $(t+1, x)$ also depends upon $(t, x \pm 2)$, ds should be set to 2.

Procedure `walk1` decomposes a trapezoid recursively into smaller trapezoids. In the base case $t_1 = t_0 + 1$, the trapezoid consists of a single row of spacetime points, which are traversed by means of a `for` loop. In the recursive case, if the trapezoid is "sufficiently wide," the procedure cuts trapezoid $\mathcal{T}$ by means of a line of slope $dx/dt = -ds$ (*space cut*) , chosen so that the two sub-trapezoids $\mathcal{T}_1$ and $\mathcal{T}_2$ are approximately of equal width. Otherwise, the procedure cuts the time dimension into two approximately equal halves (*time cut*). Figure 2 illustrates these cuts. The procedure traverses the two sub-trapezoids produced by the cuts in an order that respects the stencil dependencies.

Procedure `walk2` in Fig. 3 extends `walk1` to 2-dimensional stencils. This procedure traverses 3-dimensional spacetime trapezoids, specified by its ten arguments. The projections of such trapezoidal regions onto the (t, x) and (t, y) planes are trapezoids in the sense of procedure `walk1`. If `walk2` can perform a space cut on any space dimension, it does so, and otherwise it performs a time cut. This strategy extends in a straightforward fashion to arbitrary dimensions, as described in the Appendix.

In [15], we proved that our cache-oblivious stencil algorithm incurs $O(P/Z^{1/n})$ cache misses when traversing a $(n+1)$-dimensional trapezoidal spacetime region (n-dimensional space plus time) of P points, where Z is the size of the cache, provided that the cache is "ideal" and the region is sufficiently large. This number of cache misses matches the lower bound of Hong and Kung [18] within a constant factor. Informally, our bound holds because the algorithm decomposes the region into successively smaller regions. Once the surface of a region fits into cache, the algorithm traverses the whole region incurring a number of cache misses proportional to the surface of the region. The bound then follows from a surface vs. volume argument. We stress that the problem size for which the surface of a region fits into cache is not encoded in the algorithm (which is therefore cache oblivious), but it appears in the analysis only.

```
void walk2(int t0, int t1, int x0, int ẋ0, int x1, int ẋ1,
                           int y0, int ẏ0, int y1, int ẏ1)
{
   int Δt = t1 - t0;

   if (Δt == 1) {                                              /* base case */
      int x, y;
      for (x = x0; x < x1; x++)
         for (y = y0; y < y1; y++)
            kernel(t0, x, y);
   } else if (Δt > 1) {                                        /* recursion */
      if (2 * (x1 - x0) + (ẋ1 - ẋ0) * Δt >= 4 * ds * Δt) {         /* x cut */
         int xm = (2 * (x0 + x1) + (2 * ds + ẋ0 + ẋ1) * Δt) / 4;
         walk2(t0, t1, x0, ẋ0, xm, -ds, y0, ẏ0, y1, ẏ1);
         walk2(t0, t1, xm, -ds, x1, ẋ1, y0, ẏ0, y1, ẏ1);
      } else if (2 * (y1 - y0) + (ẏ1 - ẏ0) * Δt >= 4 * ds * Δt) {  /* y cut */
         int ym = (2 * (y0 + y1) + (2 * ds + ẏ0 + ẏ1) * Δt) / 4;
         walk2(t0, t1, x0, ẋ0, x1, ẋ1, y0, ẏ0, ym, -ds);
         walk2(t0, t1, x0, ẋ0, x1, ẋ1, ym, -ds, y1, ẏ1);
      } else {                                                 /* time cut */
         int s = Δt / 2;
         walk2(t0, t0 + s, x0, ẋ0, x1, ẋ1, y0, ẏ0, y1, ẏ1);
         walk2(t0 + s, t1, x0 + ẋ0 * s, ẋ0, x1 + ẋ1 * s, ẋ1,
                           y0 + ẏ0 * s, ẏ0, y1 + ẏ1 * s, ẏ1);
      }
   }
}
```

Fig. 3 Function `walk2` for traversing a 3-dimensional spacetime spanned by a 2-dimensional spatial grid and time

3 Heat diffusion

We employ an initial-value, heat-diffusion problem to illustrate and validate the theory of cache oblivious stencil computations empirically. Consider a simple form of the equation for 1-dimensional heat diffusion, approximated with the finite-difference equation [24]:

$$\frac{u(t+1,x)-u(t,x)}{\Delta t}=\frac{u(t,x+1)-2u(t,x)+u(t,x-1)}{(\Delta x)^2}\,. \tag{1}$$

Equation (1) describes the temperature $u(t,x)$ at space coordinate x and time t of, for example, an insulated rod with an initial temperature distribution at time $t=0$. We are interested in computing the values of $u(t,x)$ for $t=T$, given some initial values $u(0,x)$ for all x in the domain. We assume uniform grid spacings $\Delta x = 1/(N-1)$ and $\Delta t = 1/T$. Let the ends of the insulated rod be connected forming a ring, so that the resulting problem constitutes a periodic initial-value problem. (Our cache oblivious algorithm can be applied to boundary-value problems as well.)

Figure 4 shows a simple C program for computing the finite difference approximation. The kernel computation in function `kernel` consists of a 3-point stencil, as is clear from Eq. (1). During program execution, for each space coordinate x, we maintain only two spacetime points (t,x) and $(t+1,x)$, stored in memory locations $u[t \bmod 2][x]$ and $u[(t+1) \bmod 2][x]$, for some t that is not necessarily the same for distinct values of x, and that varies as the computation proceeds. This data organization corresponds to the standard programming practice of alternating between two arrays for even and odd t. We say that the data are maintained *in-place*, because we

Fig. 4 C program for computing heat diffusion according to Eq. (1)

```
double u[2][N];

void kernel(int t, int x)
{
   u[(t+1)%2][x] = u[t%2][x] +
       Δt/(Δx)^2 * (u[t%2][(x+N-1)%N] -
       2.0 * u[t%2][x] + u[t%2][(x+1)%N]);
}

void heat(void)
{
   int t, x;
   for (t = 0; t < T; t++)
      for (x = 0; x < N; x++)
         kernel(t, x);
}
```

Fig. 5 Cache oblivious program for computing heat diffusion, cf. Fig. 4

```
void kernel(int t, int x)
{
   u[(t+1)%2][x%N] = u[t%2][x%N] +
       Δt/(Δx)^2 * (u[t%2][(x+N-1)%N] -
       2.0*u[t%2][x%N] + u[t%2][(x+1)%N]);
}

void coheat(void)
{
   ds = 1;
   walk1(0, T, 0, 1, N, 1);   /* ẋ0 = ẋ1 = 1 */
}
```

reuse memory locations by overwriting those values not needed for future computations. For large problem size N, the cache misses incurred by accesses to temperature array u dominate the total number of cache misses.

We apply the kernel to all points of 2-dimensional spacetime $0 \leq t < T$, $0 \leq x < N$. The doubly nested loop in function `heat` implements the spacetime traversal in a simple yet inefficient manner if problem size N is larger than cache size Z. Note that the number of cache misses incurred by this program constitutes the worst case scenario: for $N \gg Z$, the two-fold nested loop visits $P = TN$ spacetime points, which causes $\Theta(TN)$ cache misses due to array u. As an alternative solution, we present our cache oblivious version in Fig. 5. Function `coheat` assigns the stencil slope $ds = 1$ associated with the 3-point stencil, and calls function `walk1` to perform the spacetime traversal. The new function `kernel` is called in the base case of `walk1`, as described in Sect. 2.

We evaluate the memory behavior of the heat equation solver by comparing load-miss counts of the naive, iterative program with those of the cache oblivious version in Tables 1, 2, and 3. Load-misses are the first-order effect of memory behavior and, therefore, reflect the primary difference between the two programs. Note that store misses do not add any information to the analysis, because, for stencil computations, the number of store misses is within a constant factor of the number of load misses. The numbers in Table 1 for 1-dimensional space ($n = 1$) stem from the naive program shown in Fig. 4 and from the cache oblivious program in Fig. 5. Tables 2 and 3 show the load misses for higher dimensional problems with $n = 2$ and $n = 3$. All numbers

Table 1 Load misses of 1-dimensional heat diffusion for $N = 60{,}000$ and $T = 1{,}000$

Z	2-way, 32 bytes/line			4-way, 32 bytes/line			2-way, 128 bytes/line			4-way, 128 bytes/line		
	naive	obliv	ratio	naive	obliv	ratio	naive	obliv	ratio	naive	obliv	ratio
16 K	15,001,050	105,239	142.5	15,001,050	93,083	161.2	3,751,039	108,425	34.6	3,751,039	24,085	155.7
32 K	15,001,050	51,388	291.9	15,001,050	45,798	327.5	3,751,039	50,632	74.1	3,751,039	11,626	322.6
64 K	15,001,050	16,356	917.2	15,001,050	16,389	915.3	3,751,039	4,140	906.0	3,751,039	4,160	901.7
128 K	15,001,050	15,663	957.7	15,001,050	15,567	963.6	3,751,039	3,947	950.4	3,751,039	3,919	957.1
256 K	15,001,050	15,559	964.1	15,001,050	15,559	964.1	3,751,039	3,917	957.6	3,751,039	3,917	957.6
512 K	15,001,050	15,555	964.4	15,001,050	15,555	964.4	3,751,039	3,916	957.9	3,751,039	3,916	957.9
1 M	15,048	15,049	1.0	15,048	15,049	1.0	3,788	3,787	1.0	3,788	3,787	1.0
2 M	15,048	15,049	1.0	15,048	15,049	1.0	3,788	3,787	1.0	3,788	3,787	1.0
4 M	15,048	15,049	1.0	15,048	15,049	1.0	3,788	3,787	1.0	3,788	3,787	1.0

Table 2 Load misses of 2-dimensional heat diffusion for $N = 1000$ and $T = 100$. All miss counts are expressed in thousands

Z	2-way, 32 bytes/line			4-way, 32 bytes/line			2-way, 128 bytes/line			4-way, 128 bytes/line		
	naive	obliv	ratio	naive	obliv	ratio	naive	obliv	ratio	naive	obliv	ratio
16 K	75,200	8,135	9.2	75,200	7,491	10.0	18,950	5,436	3.5	18,950	3,026	6.3
32 K	25,288	4,990	5.1	25,210	4,863	5.2	6,499	2,911	2.2	6,445	1,782	3.6
64 K	25,150	3,283	7.7	25,025	3,381	7.4	6,361	1,066	6.0	6,256	1,068	5.9
128 K	25,101	2,870	8.7	25,025	2,324	10.8	6,312	863	7.3	6,256	677	9.2
256 K	25,076	1,572	16.0	25,025	1,666	15.0	6,287	442	14.2	6,256	469	13.3
512 K	25,025	1,066	23.5	25,025	1,120	22.3	6,256	284	22.0	6,256	299	20.9
1 M	25,025	1,033	24.2	25,025	701	35.7	6,256	270	23.2	6,256	176	35.5
2 M	25,025	680	36.8	25,025	696	35.9	6,256	170	36.6	6,256	174	35.8
4 M	25,025	313	79.7	25,025	359	69.6	6,256	78	79.6	6,256	90	69.2

Table 3 Load misses of 3-dimensional heat diffusion for $N = 100$ and $T = 100$. All miss counts are expressed in thousands

Z	2-way, 32 bytes/line			4-way, 32 bytes/line			2-way, 128 bytes/line			4-way, 128 byte/line		
	naive	obliv	ratio	naive	obliv	ratio	naive	obliv	ratio	naive	obliv	ratio
16 K	75,018	46,446	1.6	75,016	43,230	1.7	18,766	25,366	0.7	18,762	23,887	0.8
32 K	75,016	28,829	2.6	75,016	29,352	2.6	18,762	16,293	1.2	18,762	16,418	1.1
64 K	75,016	23,695	3.2	75,016	21,160	3.5	18,762	13,021	1.4	18,762	10,858	1.7
128 K	75,016	16,167	4.6	75,016	16,507	4.5	18,762	7,552	2.5	18,762	7,697	2.4
256 K	50,205	12,064	4.2	75,016	12,207	6.1	12,573	4,960	2.5	18,762	4,966	3.8
512 K	25,270	9,567	2.6	25,253	9,399	2.7	6,319	3,662	1.7	6,313	3,471	1.8
1 M	25,253	7,420	3.4	25,253	7,564	3.3	6,314	2,577	2.4	6,313	2,600	2.4
2 M	25,253	6,321	4.0	25,253	5,601	4.5	6,313	2,177	2.9	6,313	1,839	3.4
4 M	25,253	4,451	5.7	25,253	4,481	5.6	6,313	1,360	4.6	6,313	1,371	4.6

are generated by IBM's full-system simulator Mambo [10], simulating one Power4 processor with four different memory subsystem configurations, described in detail below. All programs were compiled with `gcc -O2 -m32`, version 3.4.0, and run under Linux-2.6.7.

Each of the tables presents the load misses of four different cache configurations: (1) a 2-way set-associative cache with a line size of 32 bytes, (2) a 4-way set-associative cache with 32 bytes per line, (3) a 2-way set-associative cache with 128 bytes per line, and (4) a 4-way set-associative cache with 128 bytes per line.[1] For each of the four configurations, we vary the total cache size Z between 16 Kbytes and 4 Mbytes. We report the absolute number of load misses of the naive iterative version, the cache oblivious version, and the ratio of the two counts. The ***ratio*** is the factor by which the cache oblivious version reduces the number of load misses compared to the naive version.

The load-miss data enable us to compare theory and practice of cache oblivious algorithms. The theory is based on a fully associative "ideal cache," it ignores the effects of line sizes, and it assumes that the working set is too large to fit into cache. In contrast, we simulated realistic caches with limited associativity and large line sizes, and both large as well as small problems. The data in the three tables lead us to the following observations:

1. The number of cache misses of the naive version is a stepwise constant function of cache size Z. For increasing Z, the number of load misses drops once a number of rows, planes, or the entire working set fit into the cache.
2. The number of load misses of the cache oblivious version is significantly smaller than those of the naive version if the working set does not fit into cache. Furthermore, for increasing cache size Z, the number of load misses $\Theta(P/Z^{1/n})$ decreases monotonically according to the theory.
3. The benefit of increasing associativity from 2-way to 4-way is negligible for the smaller line size of 32 bytes, yet noticeable for small caches with a large line size of 128 bytes.
4. Increasing the line size by a factor of 4 approaches the ideal reduction of the number of load misses by a factor of 4 in the naive version. The benefit for the cache oblivious version is less pronounced. This is not surprising, since the loop structure of the naive version introduces the artificial property of spatial locality while the cache oblivious version exploits the algorithmically fundamental property of temporal locality.
5. We only modeled one cache level in our simulations, but the results can also be interpreted as L2 misses on systems where L2 is inclusive of L1, for example.

In the following, we discuss the discrepancies between the predicted and the observed number of load misses. Our theory states that the number of load misses for n-dimensional heat diffusion is $O(P/Z^{1/n})$. This statement is the result of an asymptotic analysis, which assumes that N and T are large compared to cache size Z. We now discuss how the theory breaks down when either N or T are small.

[1] The last two configurations reflect the cache parameters of the Power4 and Power5 processors, respectively.

Small N In Table 1, the number of load misses of the naive version drops for cache sizes $Z \geq 1$ Mbyte to the asymptotic limit of the cache oblivious version, because the working set fits into cache. In general, if the working set is smaller than the cache size, the number of load misses of a stencil computation is the number of compulsory misses for both the naive and the cache oblivious version. In the 1-dimensional problem, the number of load misses is $\Theta(N)$, because N temperature values of array u dominate the working set, and occupy $N \cdot 8$ bytes of memory. Thus, for $N = 60{,}000$ and 32 bytes per line, we expect 15,000 load misses, and for 128 bytes per line 3,750 load misses. These numbers agree well with those in Table 1.

Small T In Table 1, the ratio of load misses of the naive and the cache oblivious versions appears to approach 1,000 as Z increases to 512 Kbytes. This limit coincides with the number of time steps $T = 1{,}000$, and is quite easy to explain. In a spacetime region with height $T < \Theta(Z)$, we can only reduce the number of cache misses by a factor of $\Theta(T)$ rather than $\Theta(Z^{1/n})$, because each value can be reused at most $O(T)$ times. Thus, the number of load misses of the cache oblivious version becomes $\Theta(P/T)$ for small T. We observe that the limiting ratio $T = 1{,}000$ holds in all four cache configurations.

For 1-dimensional stencils, these two effects can be unified formally. If N is small, the number of cache misses is $\Theta(N)$. If T is small, the number of cache misses is $\Theta(P/T)$, which is also $\Theta(N)$ because the number of spacetime points is $P = TN$. Therefore, we conclude that the number of cache misses is

$$\Theta\left(\frac{P}{\min(T, Z)}\right) \tag{2}$$

in the 1-dimensional case.

Figure 6 illustrates the cache behavior of the 1-dimensional heat diffusion problem. We plot the load misses of the naive and cache oblivious version for a 4-way set associative cache with 32 bytes per line (the second group of columns in Table 1). The plot clearly distinguishes two regions: (1) region $N, T \gg Z$, where our theory predicts $\Theta(P/Z)$ load misses, and (2) region $N, T \lesssim Z$, where Eq. (2) prescribes $\Theta(P/T)$ load misses.

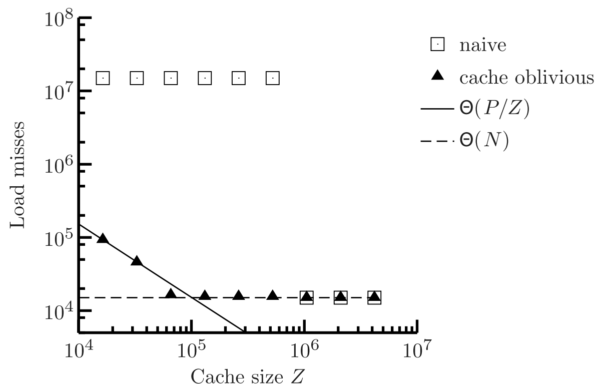

Fig. 6 Load misses of 1-dimensional heat diffusion for the naive and cache oblivious versions, and fitted predictions

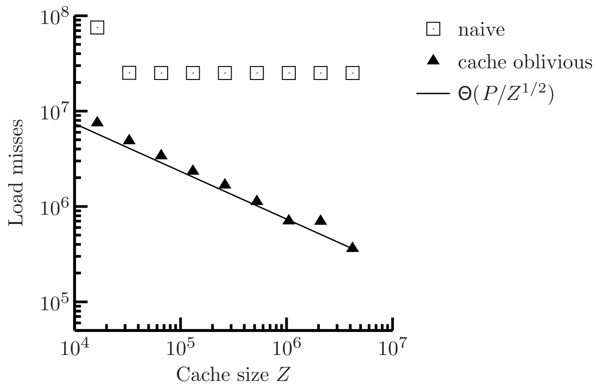

Fig. 7 Load misses of 2-dimensional heat diffusion for the naive and cache oblivious versions, and fitted predictions

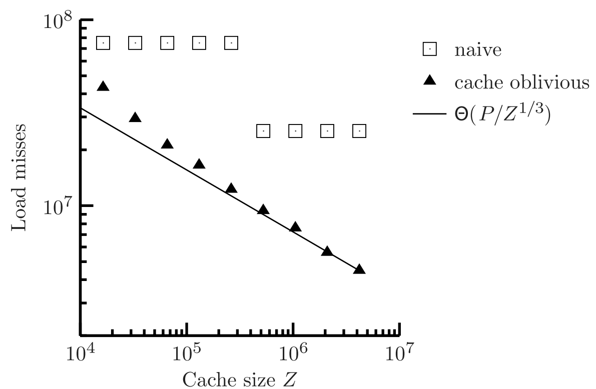

Fig. 8 Load misses of 3-dimensional heat diffusion for the naive and cache oblivious versions, and fitted predictions

Table 2 and Fig. 7 present the number of load misses for the 2-dimensional heat diffusion problem. Here, N is the rank of both space dimensions, resulting in $P = TN^2$ spacetime points. The size of the working set is proportional to N^2 and does not even fit into the largest cache. Furthermore, the number of time steps $T = 100$ is just large enough for the ratio to reach about 80% of its upper bound at value 100. Theory predicts that the cache oblivious version has $\Theta(P/\sqrt{Z})$ load misses, and N and T are large enough that the theoretical assumptions are satisfied. The fitted curve in Fig. 7 shows that theory matches experiments quite well.

The load miss data of the naive version in Table 2 demonstrate the inefficient memory behavior when compared to the cache oblivious version. The number of load misses remains flat for all cache sizes, with the exception of the drop from $Z = 16$ Kbyte to $Z = 32$ Kbyte by a factor of three. This drop can be explained by the fact that three rows of the domain fit into 32 Kbyte but not into 16 Kbyte, and enable reuse of matrix rows from previous iterations. Finally, we note that the qualitative behavior of both the naive and the cache oblivious versions is the same for all cache configurations, respectively.

Table 3 and Fig. 8 show the load misses for the 3-dimensional heat diffusion problem with $P = TN^3$ spacetime points, where $N = T = 100$. Figure 8 shows the fitted curve for the number of load misses $\Theta(P/\sqrt[3]{Z})$, as predicted by our theory for the cache oblivious version. As in the 2-dimensional case, N and T are large enough for

the theory to be applicable, and the load miss data approach the fitted curve asymptotically in Z.

We observe that the number of load misses of the naive version remains flat, except for the drop from $Z = 256$ Kbytes to $Z = 512$ Kbytes. This drop is caused by an effect analogous to the one that occurs in the 2-dimensional naive version. Here, in case of the 3-dimensional problem, three planes of the domain fit into 512 Kbytes but not into 256 Kbytes, and enable a partial form of data reuse.

4 Gauss-Seidel

In this section we study a cache oblivious algorithm of the Gauss-Seidel method [16] for banded linear systems of equations $Ax = b$. Equation (3) defines the Gauss-Seidel iteration.[2] The Gauss-Seidel method is not usually regarded as a stencil computation. Nevertheless, it can be cast into our cache oblivious stencil framework, and so can other iterative methods, including the successive over-relaxation (SOR) method, that have the same structure of Gauss-Seidel.

This equation defines the stencil computation for element x_i:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=0}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{N-1} a_{ij} x_j^{(k)}\right) \qquad \text{for } \begin{array}{l} i = 0, 1, \ldots, N-1, \\ k = 0, 1, \ldots. \end{array} \tag{3}$$

Within each iteration k, the method updates all elements x_i by looping over index i in the order $[0, 1, \ldots, N-1]$. Due to this particular order, the computation of $x_i^{(k+1)}$ can be organized in-place, and references values $x_j^{(k+1)}$ of the current iteration $(k+1)$ for $0 \le j < i$, and values $x_j^{(k)}$ for $i < j < N$ of the previous iteration k.

Equation (3) can be fine-tuned for banded systems. The lower bound of the first sum and the upper bound of the second sum are adjusted to reflect the structure of matrix A with bandwidth Q. Element x_i at iteration $(k+1)$ depends on these neighboring values:

$$x_i^{(k+1)} = f\left(x_{i-Q}^{(k+1)}, \ldots, x_{i-2}^{(k+1)}, x_{i-1}^{(k+1)}, x_{i+1}^{(k)}, x_{i+2}^{(k)}, \ldots, x_{i+Q}^{(k)}\right). \tag{4}$$

By storing only non-zero subdiagonals of A, we can reduce the storage requirements to less than $(2Q+1)N$ elements [16]. Band matrix A and the right-hand side b are read-only and, therefore, do not generate any dependencies. Vector x, however, is updated in-place by overwriting value x_i of iteration k with the new value of iteration $(k+1)$.

Figure 9 shows a cache oblivious program for the Gauss-Seidel iteration. We run a fixed number of 10 iterations, as may be desired when applying the method as a smoother of high-frequency oscillations, for instance. We associate element $x_i^{(k)}$ with spacetime point (t, x) by mapping index i into the x-coordinate and iteration count k

[2] Note the name clash between space coordinate x and vector x of unknowns in the standard notation $Ax = b$. We attempt to mitigate potential confusion by virtue of context and lexical scoping.

Fig. 9 Cache oblivious Gauss-Seidel iteration for band matrixes of order N and bandwidth Q, running 10 iterations

```
void kernel(int k, int i)
{
   int j;

   x[i] = 0.0;
   for (j = MAX(0,i-Q); j < i; j++)
      x[i] += A(i,j) * x[j];
   for (j = i+1; j < MIN(i+Q+1,N); j++)
      x[i] += A(i,j) * x[j];
   x[i] = (b[i]-x[i]) / A(i,i);
}

void co_gauss_seidel(void)
{
   ds = Q;
   walk1(0, 10, 0, 0, N, 0);
}
```

into the t-coordinate. Thus, element $x_i^{(k+1)}$ is mapped to point $(t+1, x)$, $x_{i-1}^{(k+1)}$ to $(t+1, x-1)$, $x_{i+1}^{(k)}$ to $(t, x+1)$, etc. For bandwidth Q, the stencil is an unsymmetric $(2Q+1)$-point stencil which requires stencil slope $ds = Q$.

Table 4 compares the load miss counts of the naive Gauss-Seidel program [16] with the cache oblivious version of Fig. 9. For problem size $N = 15{,}000$ and bandwidth $Q = 8$, the working set fits into the caches of size $Z = 4\,M$. By comparison with Table 1 we confirm that the Gauss-Seidel program behaves like a 1-dimensional problem ($n = 1$). Furthermore, since the number of iterations is 10, we find that the ratio of cache misses is limited by $T = 10$. Finally, we mention that we have observed runtime improvements up to a factor of 4 on a Power4+ system due to cache obliviousness.

5 Two-dimensional LBMHD

In this section, we show that a cache oblivious version of the 2-dimensional LBMHD (Lattice-Boltzmann Magneto-Hydro-Dynamics) HPCS benchmark [20, 21] runs up to 4 times faster than the naive version on a Power4+ system. Lattice Boltzmann methods [12, 19, 22] solve the Boltzmann transport equation for modeling the distribution of particles in physical systems. The LBMHD benchmark applies lattice Boltzmann methods to the evolution of a 2-dimensional conducting fluid.

We applied our cache oblivious stencil algorithm to the LBMHD benchmark. The original code is written in FORTRAN, and its computational kernel consists of two separate routines: `stream` (particle redistribution) and `collision`. We rewrote the program in C and merged functions `stream` and `collision` into a single computational kernel that has the form of a 2-dimensional 13-point stencil. In the process, we eliminated temporary arrays and the associated redundant data copies that the original code used to communicate between the `stream` and `collision` routines. Our kernel procedure comprises 309 floating point operations and 63 memory loads, and it is numerically equivalent to the original one. We produced both a cache oblivious implementation, using the 2-dimensional traversal procedure from Fig. 3, and a

Table 4 Load misses of Gauss-Seidel with $N = 15{,}000$ and $Q = 8$

Z	2-way, 32 byte/line			4-way, 32 bytes/line			2-way, 128 bytes/line			4-way, 128 bytes/line		
	naive	obliv	ratio	naive	obliv	ratio	naive	obliv	ratio	naive	obliv	ratio
16 K	712,492	221,616	3.2	712,492	215,298	3.3	181,479	86,412	2.1	178,179	62,534	2.8
32 K	712,492	160,918	4.4	712,492	95,970	7.4	178,179	50,243	3.5	178,179	25,176	7.1
64 K	712,492	157,052	4.5	712,492	75,354	9.5	178,179	42,434	4.2	178,179	19,089	9.3
128 K	712,492	73,129	9.7	712,492	74,881	9.5	178,179	18,251	9.8	178,179	18,790	9.5
256 K	712,492	71,478	10.0	712,492	71,466	10.0	178,179	17,980	9.9	178,179	17,975	9.9
512 K	712,492	71,464	10.0	712,492	71,460	10.0	178,179	17,975	9.9	178,179	17,974	9.9
1 M	712,492	71,456	10.0	712,492	71,456	10.0	178,179	17,973	9.9	178,179	17,973	9.9
2 M	225,536	71,452	3.2	328,370	71,452	4.6	56,452	17,961	3.1	82,193	17,962	4.6
4 M	71,289	71,425	1.0	71,289	71,383	1.0	17,847	17,940	1.0	17,847	17,916	1.0

Table 5 Performance of our iterative and cache-oblivious implementations of LBMHD on a 1.45 GHz Power4+ with 5.8 Gflop/s peak performance

Problem Size N		1024	2048	4096	8192
iterative	Gflop/s	0.43	0.32	0.32	0.29
	% peak	7.4	5.5	5.5	5.0
cache	Gflop/s	1.1	1.2	1.3	1.2
oblivious	% peak	19.7	20.7	22.4	20.7
Gflop/s ratio (obliv/iter)		2.6	3.7	4.0	4.2

naive, iterative version based on a straightforward nested loop for traversing spacetime and the same unified kernel.

Table 5 reports the performance of both the cache oblivious and the iterative versions. We produced these data on a 1.45 GHz Power4+ machine. Our kernel runs at about 1.55 Gflop/s on small problem sizes that fit into the L1-cache. We observe in Table 5 that the performance degradation of large problems is minimal for the cache oblivious implementation. Even though our kernel has a different structure than the original code, the performance of our iterative version is consistent with the results reported in [21] for the original LBMHD benchmark on a similar machine, suggesting that the iterative implementation is limited by the memory performance of the machine. We report observing speedups reaching factor 8 of our cache oblivious program versus the original FORTRAN code on a Power4+ system.

Table 6 compares the number of load misses of the iterative LBMHD program with the cache oblivious version. LBMHD has a memory footprint of 27 double precision numbers per spacetime point. Consequently, for problem size $N = 1{,}024$, we find that five rows of the problem domain fit into a cache of size $Z = 1$ Mbyte. Thus, the number of load misses in the iterative version drops steeply for cache sizes around $Z = 1$ Mbyte. More importantly, we observe that the number of cache misses in the cache oblivious version decreases, as expected, proportionally to the square root of the cache size, since LBMHD is a 2-dimensional problem. It is of course hard to correlate the ratios of the number of load misses in Table 6 with the speedups reported in Table 5, given the complexity of todays computer systems.

6 Conclusions

We present an empirical analysis of the memory behavior of cache oblivious stencil computations. To enable a meaningful performance evaluation, we focus our efforts on the number of load misses. We use a heat diffusion problem and a Gauss-Seidel iteration to illustrate the formulation of cache oblivious stencil computations based on our spacetime traversal routines. We conclude that the theoretically predicted cache miss counts agree with our experiments. Furthermore, we interpret those cache miss counts where the assumptions of the asymptotic theory do not hold, because the problem size and number of time steps is too small. Our cache oblivious formulation of a 2-dimensional LBMHD program provides evidence for the practical value of cache oblivious algorithms. The cache oblivious version is up to a factor of 4 faster than the iterative version, and runs up to 8 times faster than the original program on a Power4+ system.

Table 6 Load misses of LBMHD for $N = 1024$ and $T = 50$; all miss counts to be multiplied by 10^6

Z	2-way, 32 bytes/line			4-way, 32 bytes/line			2-way, 128 bytes/line			4-way, 128 bytes/line		
	naive	obliv	ratio	naive	obliv	ratio	naive	obliv	ratio	naive	obliv	ratio
16 K	1,233	774	1.6	1,194	682	1.8	544	462	1.2	448	349	1.3
32 K	1,205	578	2.1	1,193	521	2.3	469	257	1.8	443	213	2.1
64 K	1,193	400	3.0	1,193	390	3.1	443	155	2.9	443	151	2.9
128 K	1,193	331	3.6	1,193	284	4.2	443	124	3.6	443	102	4.3
256 K	1,193	297	4.0	1,193	231	5.2	443	109	4.1	443	80	5.5
512 K	1,155	265	4.3	1,189	206	5.8	443	95	4.7	443	70	6.3
1 M	609	226	2.7	627	176	3.6	305	78	3.9	372	59	6.4
2 M	367	186	2.0	354	130	2.7	97	63	1.5	89	42	2.1
4 M	357	100	3.6	354	106	3.4	91	32	2.9	89	33	2.7

Appendix Multidimensional spacetime traversal

The `walk` procedure in Fig. 10 computes n-dimensional stencils, where $n > 0$ is the number of space dimensions (i.e., excluding time) [15]. The n-dimensional trapezoid $\mathcal{T}(t_0, t_1, x_0^{(i)}, \dot{x}_0^{(i)}, x_1^{(i)}, \dot{x}_1^{(i)})$, where $0 \le i < n$, is the set of integer tuples $(t, x^{(0)}, x^{(1)}, \ldots, x^{(n-1)})$ such that $t_0 \le t < t_1$ and $x_0^{(i)} + \dot{x}_0^{(i)}(t - t_0) \le x^{(i)} < x_1^{(i)} + \dot{x}_1^{(i)}(t - t_0)$ for all $0 \le i < n$. Informally, for each dimension i, the projection of the multi-dimensional trapezoid onto the $(t, x^{(i)})$ plane looks like a 1-dimensional trapezoid. Consequently, we can apply the same recursive decomposition that we used in procedure `walk1` for the 1-dimensional case: if any dimension i permits a

```
typedef struct { int x0, ẋ0, x1, ẋ1 } C;

void walk(int t0, int t1, C c[n])
{
    int Δt = t1 - t0;

    if (Δt == 1) {
        basecase(t0, c);
    } else if (Δt > 1) {
        C *p;

        /* for all dimensions, try to cut space */
        for (p = c; p < c + n; ++p) {
            int x0 = p->x0, x1 = p->x1, ẋ0 = p->ẋ0, ẋ1 = p->ẋ1;
            if (2 * (x1 - x0) + (ẋ1 - ẋ0) * Δt >= 4 * ds * Δt) {
                /* cut space dimension *p */
                C save = *p;    /* save configuration *p */
                int xm = (2 * (x0 + x1) + (2 * ds + ẋ0 + ẋ1) * Δt) / 4;
                *p = (C){ x0, ẋ0, xm, -ds };   walk(t0, t1, c);
                *p = (C){ xm, -ds, x1, ẋ1 };   walk(t0, t1, c);
                *p = save;      /* restore configuration *p */
                return;
            }
        }

        {
            /* because no space cut is possible, cut time */
            int s = Δt / 2;
            C newc[n];
            int i;

            walk(t0, t0 + s, c);

            for (i = 0; i < n; ++i) {
                newc[i] = (C){ c[i].x0 + c[i].ẋ0 * s, c[i].ẋ0,
                               c[i].x1 + c[i].ẋ1 * s, c[i].ẋ1 };
            }

            walk(t0 + s, t1, newc);
        }
    }
}
```

Fig. 10 A C99 implementation of the multi-dimensional `walk` procedure. The code assumes that n is a compile-time constant. The base case and the definition of the slope ds are not shown

space cut in the $(t, x^{(i)})$ plane, then cut space in dimension i. Otherwise, if none of the space dimensions can be split, cut time in the same fashion as in the 1-dimensional case.

Procedure `walk` encodes a multi-dimensional trapezoid by means of an array of tuples of type `C`, the ***configuration tuple*** for one space dimension. Figure 10 hides the traversal of the n-dimensional base case in procedure `basecase`. We leave it as a programming exercise to develop this procedure, which visits all points of the rectangular parallelepiped at time step t_0 in all space dimensions by calling application specific procedure `kernel`; see the base cases in procedures `walk1` and `walk2`.

References

1. Aggarwal A, Alpern B, Chandra AK, Snir M (1987) A model for hierarchical memory. In: 19th ACM symposium on theory of computing, New York, May 1987, pp 305–314
2. Aggarwal A, Vitter JS (1988) The input/output complexity of sorting and related problems. Commun ACM 31(9):1116–1127
3. Alpern B, Carter L, Ferrante J (1995) Space-limited procedures: a methodology for portable high-performance. In: Conference on programming models for massively parallel computers, Berlin, Germany, October 1995. IEEE Computer Society, pp 10–17
4. Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1999) LAPACK users' guide. Society for Industrial and Applied Mathematics, Philadelphia, 3rd edn. http://www.netlib.org/lapack/lug/lapack_lug.html
5. Arge L, Bender MA, Demaine ED, Holland-Minkley B, Munro JI (2002) Cache-oblivious priority queue and graph algorithm applications. In: 34th ACM symposium on theory of computing. ACM Press, Montréal, Canada, 2002, pp 268–276
6. Bailey DH (1993) RISC microprocessors and scientific computing. In: Supercomputing'93, Portland, OR, November 1993, pp 645–654
7. Bender MA, Demaine ED, Farach-Colton M (2000) Cache-oblivious B-trees. In: Symposium on foundations of computer science, IEEE Computer Society, Redondo Beach, CA, November 2000, pp 399–409
8. Bilardi G, Preparata FP (1995) Upper bounds to processor-time tradeoffs under bounded-speed message propagation. In: 7th ACM symposium on parallel algorithms and architectures, ACM Press, Santa Barbara, 1995, pp 185–194
9. Blumofe RD, Frigo M, Joerg CF, Leiserson CE, Randall KH (1996) An analysis of dag-consistent distributed shared-memory algorithms. In: 8th ACM symposium on parallel algorithms and architectures, Padua, Italy, June 1996, pp 297–308
10. Bohrer P, Elnozahy M, Gheith A, Lefurgy C, Nakra T, Peterson J, Rajamony R, Rockhold R, Shafi H, Simpson R, Speight E, Sudeep K, Van Hensbergen E, Zhang L (2004) Mambo: a full system simulator for the PowerPC architecture. SIGMETRICS Perform Eval Rev 31(4):8–12
11. Brodal GS, Fagerberg R, Vinther K (2004) Engineering a cache-oblivious sorting algorithm. In: 6th Workshop on algorithm engineering and experiments SIAM, New Orleans, LA, January 2004, pp 4–17
12. Chen S, Doolen GD, Eggert KG (1994) Lattice-Boltzmann fluid dynamics: a versatile tool for multiphase and other complicated flows. Los Alamos Sci 22:98–19
13. Dongarra JJ, Moler CB, Bunch JR, Stewart GW (1979) LINPACK users' guide. Society for Industrial and Applied Mathematics, Philadelphia
14. Frigo M, Leiserson CE, Prokop H, Ramachandran S (1999) Cache-oblivious algorithms. In: 40th symposium on foundations of computer science, New York, NY, October 1999. ACM Press
15. Frigo M, Strumpen V (2005) Cache oblivious stencil computations. In: International conference on supercomputing, Boston, MA, June 2005. ACM Press, pp 361–366
16. Golub GH, van Loan CF (1996) Matrix computations, 3rd edn. Johns Hopkins University Press, Baltimore
17. Goto K, van de Geijn R (2001) On Reducing TLB Misses in Matrix Multiplication. Technical Report TR-2002-55, Department of Computer Sciences, The University of Texas at Austin (FLAME Working Note #9)

18. Hong J-W, Kung HT (1981) I/O complexity: the red-blue pebbling game. In: 13th ACM Symposium on Theory of Computing, Milwaukee, WI, May 1981, pp 326–333
19. Kowarschik M (2004) Data locality optimizations for iterative numerical algorithms and cellular automata on hierarchical memory architectures. PhD thesis, Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, Universität Erlangen-Nürnberg, Erlangen, Germany, July 2004
20. Macnab A, Vahala G, Vahala L, Pavlo P (2002) Lattice Boltzmann model for dissipative MHD. In: 29th EPS conference on controlled fusion and plasma physics, vol 26B, Montreux, Switzerland, June 2002
21. Oliker L, Canning A, Carter J, Shalf J, Ethier S (2004) Scientific computations on modern parallel vector systems. In: Supercomputing'04, Pittsburgh, PA, November 2004, IEEE. http://www.sc-conference.org/sc2004/papers.html
22. Pohl T, Deserno F, Thürey N, Rüde U, Lammers P, Wellein G, Zeiser T (2004) Performance evaluation of parallel large-scale lattice Boltzmann applications on three supercomputing architectures. In: Supercomputing'04, Pittsburgh, PA, November 2004, IEEE. http://www.sc-conference.org/sc2004/papers.html
23. Prokop H (1999) Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 1999
24. Smith GD (1985) Numerical solution of partial differential equations: finite difference methods, 3rd edn. Oxford University Press, Oxford
25. Toledo S (1997) Locality of reference in LU decomposition with partial pivoting. SIAM J Matrix Anal Appl 18(4):1065–1081