J Supercomput (2012) 62:946–966
DOI 10.1007/s11227-012-0764-z

Hierarchical parallelization and optimization of high-order stencil computations on multicore clusters

Hikmet Dursun · Manaschai Kunaseth · Ken-ichi Nomura · Jacqueline Chame · Robert F. Lucas · Chun Chen · Mary Hall · Rajiv K. Kalia · Aiichiro Nakano · Priya Vashishta

Published online: 18 April 2012

H. Dursun (✉) · M. Kunaseth · K. Nomura · R.K. Kalia · A. Nakano · P. Vashishta
Collaboratory for Advanced Computing and Simulations, Department of Computer Science, Department of Physics & Astronomy, Department of Chemical Engineering & Materials Science, University of Southern California, Los Angeles, CA 90089, USA
e-mail: hdursun@usc.edu

M. Kunaseth
e-mail: kunaseth@usc.edu

K. Nomura
e-mail: knomura@usc.edu

R.K. Kalia
e-mail: rkalia@usc.edu

A. Nakano
e-mail: anakano@usc.edu

P. Vashishta
e-mail: priyav@usc.edu

J. Chame · R.F. Lucas
Information Sciences Institute, University of Southern California, Suite 1001, 4676 Admiralty Way, Marina del Rey, CA 90292, USA

J. Chame
e-mail: jchame@isi.edu

R.F. Lucas
e-mail: rflucas@isi.edu

C. Chen · M. Hall
School of Computing, University of Utah, Salt Lake City, UT 84112, USA

C. Chen
e-mail: chunchen@cs.utah.edu

M. Hall
e-mail: mhall@cs.utah.edu

Abstract We present a scalable parallelization scheme for high-order stencil computations that also optimizes memory behavior on multicore clusters. Our multilevel approach combines: (i) inter-node parallelization via spatial decomposition; (ii) inter-core parallelization via multithreading and explicit non-uniform memory access (NUMA) control; (iii) data locality optimizations through auto-tuned tiling for efficient use of hierarchical memory; and (iv) register blocking and data parallelism via single-instruction multiple-data techniques to utilize registers and exploit data locality. The scheme is applied to a sixth-order stencil based finite-difference time-domain code. Weak-scaling parallel efficiency is over 98 % on 32,768 BlueGene/P processors. Multithreading with explicit NUMA control attains 9.9-fold speedup on a dual 12-core AMD Opteron system. Data locality optimizations achieve 7.7-fold reduction of the last level cache miss rate of Intel Nehalem, whereas register blocking increases data parallelism and thereby achieves 5.9 Gflops performance on a single core. Register blocking + multithreading optimizations achieve 5.8-fold speedup on a single quadcore Nehalem.

Keywords Stencil computation · PDE solvers · Finite differences · Structured grid · NUMA · Blocking · Multithreading · Single instruction multiple data parallelism · Message passing · Spatial decomposition

1 Introduction

As hierarchical multicore processors with complex computational and memory organizations emerge as a result of the quest for simultaneous performance and power-efficiency improvement, a challenge faced by software developers and application scientists is the adaptation of algorithms to effectively utilize this underlying hardware for broad computational applications. The emerging multicore paradigm has given us unprecedented supercomputing power [1], such as IBM BlueGene L and P, Road Runner, and Cray Jaguar, through scaling at multiple levels, and in particular, multiple cores per node, interconnected into hierarchical systems of up to more than 100,000 cores. All have complex memory hierarchies, where some memory is shared across cores, some is dedicated, and some requires explicit management in software. While at different scales, the features of these architectures are mirrored in commercial microprocessors, which represent their constituent nodes, and are often combined into clusters of nodes as targets of high-end applications. From an application programmer's perspective, we hypothesize that such architectures can be viewed as hierarchical computational units with corresponding hierarchical storage that is explicitly or implicitly managed by software. The computation hierarchy includes support for fine-grain data-parallelism, through single instruction multiple data (SIMD) multimedia extensions such as Streaming SIMD Extensions (SSE) for Intel and AMD platforms and Altivec for PowerPC, through the synergistic processing elements of the IBM Cell, or through the streaming processors of an NVIDIA graphics processing unit (GPU). Across cores, thread-level parallelism permits potentially independent computation on related data, while across nodes, coarse-grain parallelism on independent data can be exploited. Data locality is critical to achieving high performance on such architectures, so memory structures including registers, multilevel

caches, and storage buffers should be carefully managed to match the hierarchical parallel constructs.

As a common computational kernel in a variety of scientific and engineering applications [2–4], stencil computation (SC) has extensively been studied. For example, Nguyen et al. [5] have implemented a hybrid blocking algorithm which combines spatial and temporal blocks [6] to optimize a 7-point nearest neighbor kernel on Intel Nehalem and NVIDIA GPU. Williams et al. [7] have optimized a lattice Boltzmann application on leading multicore platforms, including Intel Clovertown, AMD Opteron, and STI Cell. Datta et al. [8] have devised cache-oblivious algorithms and Peng et al. [9] have used data layout reordering approach to optimize first-order stencil computations on a variety of state-of-the-art architectures, including NVIDIA GPU and IBM Cell BE processor. Other approaches to SC optimization include tiling [10] and iteration skewing [11–13]. However, there has been little research on optimizing high-order stencil computations (HOSC). HOSC pattern is characterized by large memory footprint stencils, which usually span multiple levels of parallelization ranging from data to inter-core to inter-node levels and involve frequent high-memory stride accesses. The pivotal role of HOSC in broad applications and the emergence of a wide landscape of heterogeneous multicore architectures have motivated us to develop a unified parallelization strategy that scales on massively parallel multicore supercomputers and perform systematic performance optimization on each of its hierarchical levels.

In our earlier work [14], we have focused on system-level optimizations for HOSC. In this paper, we instead emphasize processor-level optimization techniques and propose a hierarchical scalable parallelization scheme that exploits the floating-point performance of the computational units through efficient use of the hierarchical memory levels in modern multicore processors. Our scheme combines: (i) inter-node parallelization via spatial decomposition; (ii) inter-core parallelization via multithreading using explicit non-uniform memory access (NUMA) control; (iii) data locality optimizations through auto-tuned tiling for efficient use of hierarchical memory; and (iv) register blocking (RB) and data parallelism via SIMD techniques to utilize registers and exploit data locality. We illustrate the hierarchical scalable parallelization scheme by applying it to a sixth-order stencil based finite-difference time-domain (FDTD) application, which is used in various areas such as photonics [15] and quantum dynamics [16]. We obtain good overall strong scalability on our test platforms, with even superlinear speedups on the Intel architecture. Excellent weak-scalability is also achieved on the 256 processor Intel Clovertown-based cluster and 32,768 processors of BlueGene/P. Intra-node optimization using multithreading and SIMD parallelization achieves a speedup of 5.83 for 8 threads on a single quadcore Intel Nehalem node. We attain 9.93-fold speedup with NUMA control on a dual 12-core AMD 6172 Opteron Magny-Cours system. We achieve 7.7-fold reduction of the last level cache miss rate, and 55 % of the theoretical peak performance on a single core of Intel Nehalem by incorporating loop tiling for Translation Lookaside Buffer (TLB), caches and registers, and explicit use of SSE instructions.

This paper is organized as follows. An overview of the SC is given in Sect. 2 together with the description of the kernel and its application in FDTD method. Section 3 describes our hierarchical parallelization strategy and the optimization tech-

niques. Section 4 presents the scaling and optimization performance of our scheme and their corresponding analysis. Finally, we summarize our study in Sect. 5.

2 High-order stencil computation

This section first introduces HOSC, and then provides a detail of our application and experimental kernel used in the subsequent sections.

2.1 Stencil computation

Accuracy of SC generally depends on the order of its stencil. Applications employing SC mostly involve partial differential equation solvers, which are based on finite-difference and multigrid methods, to study a vast array of computer simulations such as photonics [15] and acoustic wave propagation [17], quantum dynamics [16], and financial analytics [18]. Due to its pivotal role in computational sciences, SC is included in a number of benchmark suites such as PARKBENCH [19] and NAS Parallel Benchmarks [20]. Implementation of special purpose stencil compilers [21–23] and development of compiler optimizations [24] highlight the common use of SC based methods.

In SC, values $u^{(t)}(\mathbf{r})$ are assigned to a set of discrete grid points $\boldsymbol{\Omega} = \{\mathbf{r}\}$ for a number of simulation time steps $t \in [1, N_{step}]$. SC routine sweeps over $\boldsymbol{\Omega}$ iteratively to compute values of the grid points at the next iteration, $u^{(t+1)}(\mathbf{r})$, as a function of the values of the neighboring nodal set at time t, $\boldsymbol{\Omega}' = \{\mathbf{r}' \mid \mathbf{r}' \in neighbor(\mathbf{r})\}$, determined by the stencil geometry. According to the geometric arrangement of the nodal group $neighbor(\mathbf{r})$, SC may be classified as follows: First, the order of a stencil, n, is defined as the distance between the grid point of interest, $\mathbf{r}$, and the farthest grid point in $neighbor(\mathbf{r})$ along a certain axis (In a finite-difference application, the order increases with required level of precision). Second, we define the size of a stencil as the cardinality $|\{\mathbf{r}' \mid \mathbf{r}' \in neighbor(\mathbf{r})\}|$, i.e., the number of grid points involved in each stencil iteration. Third, the footprint of a stencil is defined by the cardinality of minimum bounding orthorhombic volume, which includes all involved grid points per stencil. For example, Fig. 1 shows a third order, 13-point SC whose footprint is $7^2 = 49$ on a 2-dimensional lattice. Such a stencil is widely used in high-order finite-difference calculations [25, 26]. In Fig. 1, the grid point of interest, $\mathbf{r}$, is shown as a large solid circle while the set of neighbor points, excluding $\mathbf{r}$, $\{\mathbf{r}' \mid \mathbf{r}' \in neighbor(\mathbf{r})\} - \{\mathbf{r}\}$, is illustrated as small solid circles. Open circles show the other lattice sites within the stencil footprint, which are not used for calculation of $u^{(t+1)}(\mathbf{r})$.

A typical computation for updating the value of the central grid point shown in Fig. 1 is

$$\begin{aligned} u_{i,j}^{(t+1)} = {} & c_{-3} \times u_{i-3,j}^{(t)} + c_{-2} \times u_{i-2,j}^{(t)} + c_{-1} \times u_{i-1,j}^{(t)} + c_3 \times u_{i+3,j}^{(t)} \\ & + c_2 \times u_{i+2,j}^{(t)} + c_1 \times u_{i+1,j}^{(t)} + c_{-3} \times u_{i,j-3}^{(t)} + c_{-2} \times u_{i,j-2}^{(t)} \\ & + c_{-1} \times u_{i,j-1}^{(t)} + c_3 \times u_{i,j+3}^{(t)} + c_2 \times u_{i,j+2}^{(t)} + c_1 \times u_{i,j+1}^{(t)} + c_0 \times u_{i,j}^{(t)} \end{aligned}$$

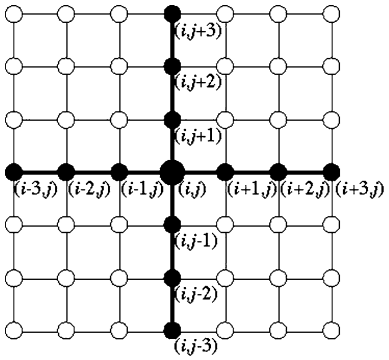

Fig. 1 Third order, 13-point SC whose footprint is 7^2 on a 2-dimensional lattice

where the subscripts i, j of $u^{(t)}$ represent the coordinates of grid points, and the coefficients are denoted as c. The coefficients c are time independent and only depend on the finite differencing method.

2.2 A high-order stencil application for FDTD method

Our experimental SC application simulates the time evolution of a field such as electromagnetic field or electronic wave function. The simulation methodology is based on a 3D equivalent of the stencil in Fig. 1 and computes spatial derivatives on uniform grids using a finite-difference method. The 3D stencil kernel is highly off-diagonal, i.e., each grid point interacts with other grid points in each of the x, y, and z Cartesian directions only. In addition, the stencil is sixth order and involves 37 points (footprint is $13^3 = 2,197$).

A typical problem space may involve several hundreds of grid nodes in each dimension amounting to millions of grid points in total. The FDTD application reported here uses $384^3 = 56.62$ million points. For each grid point, the code allocates 5 floats to hold temporary arrays and intermediate physical quantities and the result, therefore its memory footprint is 5×4 bytes $\times\ 384^3 = 1.13$ GB. Thus, the application not only involves heavy computations, but also uses three orders of magnitude larger memory than the cache size of multicore processors in the current HPC literature.

Figure 2 shows the pseudocode of a computational kernel of the FDTD application. In the kernel, the triply-nested loop in 3D Cartesian coordinates updates the value of each target grid points, $u^{(t+1)}$, based on values of neighboring grid points at time t, i.e., $u^{(t)}$. The size of the problem domain is $(n_x - 2n) \times (n_y - 2n) \times (n_z - 2n)$, where n is the stencil order and n_x, n_y, and n_z are the numbers of grid points in the three Cartesian coordinates. Stride in memory space is a critical factor to optimize SC application. In our kernel, the allocation for both $u^{(t)}$ and $u^{(t+1)}$ are 3-dimensional, 16-bytes aligned dynamic arrays, where x is the unit stride direction, y is n_x stride, and z is the highest stride $n_x \times n_y$. The scopes of each loop are reduced by $2n$ ($n = 6$ in our case) to avoid complex boundary conditions from the experimental kernel.

Fig. 2 Pseudocode of the high-order stencil kernel

```
1: procedure naiveHOSC(n, n_x, n_y, n_z, c, u^(t), u^(t+1))
2:     for i = n to n_x - n do
3:       for j = n to n_y - n do
4:         for k = n to n_z - n do
5:           for i_n = -n to n do                    //X sweep
6:             u^(t+1)_{i,j,k} ← u^(t+1)_{i,j,k} + c_{i_n} * u^(t)_{i+i_n,j,k}
7:           end for
8:           for j_n = -n to n do                    //Y sweep
9:             u^(t+1)_{i,j,k} ← u^(t+1)_{i,j,k} + c_{j_n} * u^(t)_{i,j+j_n,k}
10:          end for
11:          for k_n = -n to n do                    //Z sweep
12:            u^(t+1)_{i,j,k} ← u^(t+1)_{i,j,k} + c_{k_n} * u^(t)_{i,j,k+k_n}
13:          end for
14:        end for
15:      end for
16:    end for
17: end procedure
```

3 Hierarchical parallelization

This section outlines our top-down approach to application tuning at system-, node-, and microarchitecture-levels. We discuss our parallelization framework that combines: (i) inter-node parallelization via spatial decomposition; (ii) intra-node parallelization via multithreading; (iii) data locality optimizations through tiling; and (iv) register blocking (RB) and data parallelism via SIMD techniques to utilize registers.

3.1 Inter-node parallelism by spatial decomposition

Our parallel implementation essentially retains the computational methodology of the original sequential code. All of the existing subroutines and the data structures are retained. Our parallelization is based on spatial decomposition, where the 3D data space is decomposed into a mutually exclusive and collectively exhaustive set of subdomains. We distribute the data by assigning each subdomain to a compute node in the network, and employ an owner-computes rule.

In a parallel processing environment, it is vital to consider the time complexity of the communication and the computation dictated by the underlying algorithm, to achieve efficient parallelism. For a d-dimensional stencil problem of order n and global grid size n_g to run on p processors, the communication complexity is $O(n(n_g/p)^{(d-1)/d})$, whereas computation associated by the subdomain is proportional to the number of owned grid points, therefore its time complexity is $O(n \times n_g/p)$ (which is linear in n, assuming a stencil geometrically similar to that shown in Fig. 1). For a 3D problem, they reduce to $O(n((n_x \times n_y \times n_z)/p)^{2/3})$ and $O(n(n_x \times n_y \times n_z)/p)$, which are proportional to the surface area and volume of the underlying subdomain, respectively. Accordingly, subdomains are chosen to be the optimal orthorhombic box in the 3D domain minimizing the surface-to-volume ratio, $O((p/(n_x \times n_y \times n_z))^{1/3})$, for a given number of processors (e.g., a cubic subvolume, if the processor count is cube of an integer).

```
1:  for t = 1 to N_step
2:    if parity = sendfirst
3:      send sendBuffer to neighbor
4:      receive recvBuffer from neighbor
5:    else
6:      receive recvBuffer from neighbor
7:      send sendBuffer to neighbor
8:    end if
9:    forall r ∈ Ω_p do
10:     compute stencil in owned space
11:   end for
12: end for
```

Fig. 3 Pseudocode for spatial decomposition

The pseudocode in Fig. 3 represents our spatial-decomposition approach. We implement the message passing using Message Passing Interface (MPI) [27]. A typical SC requires boundary grid points to be exchanged among processors. Therefore, each subdomain is augmented with surrounding layers of buffer grids used for data transfer. The *parity* of a process determines if it first sends, i.e., a *sendfirst*, its own grid points at the boundary, *sendBuffer*, or receives the neighboring grid points from its neighbor process to update its receive buffer, *recvBuffer*. Once the buffer layers are exchanged, each process computes the stencil for their own grid points, i.e., $r \in \Omega_p$. Line 10 of Fig. 3 encapsulates the SC in lines 5–13 of Fig. 2.

3.2 Intra-node parallelism by multithreading

We base our implementation on hierarchical spatial decomposition: (i) inter-node parallelization with the higher-level spatial decomposition into domains based on message passing; and (ii) intra-node parallelization with the lower-level spatial decomposition within each domain through multithreading. We implement 3 subroutines to handle x, y, and z directional sweeps inside the main loop shown in Fig. 2. The procedure *zSweepTh* in Fig. 4 shows a thread spawned to perform z directional sweep. The *zSweepTh* thread receives n_{tx} as a parameter, which has lower and upper bounds of its assigned block, $n_{tx,l}$ and $n_{tx,u}$, respectively. This divides the global grid domain in the x direction. Recall that x is the unit-stride direction, y has n_x stride, and z has the highest stride of $n_x \times n_y$. Other threads reuse the method in *zSweepTh*, i.e., their innermost loop of the triply nested loop body is the direction of computation, second loop is the direction that has the lower stride among the other two Cartesian coordinates, and the outermost loop being the higher stride dimension.

Notice that in line 5 of Fig. 4, we pack the grid points $u^{(t)}$ into $packed\_u^{(t)}$. This is in order to pay the non-contiguous memory access penalty only once for higher-stride dimensions, i.e., y and z. Therefore, x thread does not implement this packing loop. In line 8, we represent the stencil to be a function f of coefficients c and the neighbor points of the grid point $packed\_u_k^{(t)}$. Here, the accessed neighbors are naturally the z neighbors since *zSweepTh* only implements the z-directional computation. Threads of the same direction can independently execute and perform updates on their assigned subdomain without introducing any locks until finally they are joined.

We utilize the native Linux kernel NUMA policy support through `numactl` utility. Numerical values of the NUMA control parameters are shown in the Table 1.

```
1: procedure zSweepTh(n, n_tx, n_y, n_z, c, u^(t), u^(t+1))
2:     for j = n to n_y - n do
3:       for i = n_tx,l to n_tx,u do
4:         for k = n to n_z - n do
5:           packed_u_k^(t) ← u_i,j,k^(t)
6:         end for
7:         for k = n to n_z - n do
8:           u_i,j,k^(t+1) ← f(c, neighbor(packed_u_k^(t)))
9:         end for
10:      end for
11:    end for
12: end procedure
```

Fig. 4 Pseudocode for the threaded code section. Since threads store their data in the $u^{(t+1)}$ array, they avoid possible race conditions and the program can exploit thread-level parallelism (TLP)

Table 1 Parameter uses to enforce NUMA control

Number of threads	Value of N
1–6	0
7–12	0,1
13–18	0,1,2
≥ 19	0,1,2,3

The `numactl` enforces the threads to run only on a specified number of NUMA processors while a round-robin memory allocation is being performed only on the same NUMA processors that threads are allowed to run. Specifically, we use `numactl --interleave=`N `--cpunodebind=`N command, where the parameter N specifies the set of processors on which the round-robin memory interleaving policy is to be enforced.

Note that on our Magny-Cours machine, there are 2 multi-chip module (MCM) packages. Each MCM packs 2 NUMA processors with 6 physical cores each. Therefore, the parameter N specified in Table 1 ensures that the memory and threads are only allocated/spawned on the corresponding NUMA processors among the available four. In addition, since each thread is not pinned to a specific core, round-robin allocation will average out the access time of all threads even if they are switching between cores.

3.3 Loop unrolling and tiling optimizations

The stencil code in Fig. 2 has data reuse in all loops but traverses a very large memory footprint, which prevents the reuse to be fully exploited in the core's memory hierarchy. In this subsection, we discuss locality optimizations for exploiting data reuse.

To compute a single value, $u_{i,j,k}^{(t+1)}$, the stencil code reads $2n$ elements from each dimension of the 3-dimensional array $u^{(t)}$. The computation of $u_{i,j,k}^{(t+1)}$ in one of the dimensions reuses $2n-1$ elements in that dimension and reads $2n$ new values in each of the other two dimensions. For example, the computation of $u_{i+1,j,k}^{(t+1)}$ reuses $2n-1$ elements in x, and reads $2n$ new elements in y and z (plus a new element in x,

```
1: procedure unrollHOSC(n,n_x,n_y,n_z,c,u^(t),u^(t+1))
2:     for k = n to n_z - n do
3:         for j = n to n_y - n do
4:             for i = n to n_x - n do
5:                 u^(t+1)_{i,j,k} ← f(c,neighbor(u^(t)_{i,j,k}))
6:             end for
7:         end for
8:     end for
9: end procedure
```

Fig. 5 Pseudocode for unrolled code. Function f represents stencil computation

$u^{(t)}_{i+1+n,j,k}$). Therefore, each of the loops z, y, and x reuses data across iterations. However, the data reused by different iterations of the outer loops may not be in the cache(s) because of the large amount of data accessed in between.

The size of the memory footprint traversed by a stencil code is another factor preventing data locality. The memory footprint of $u^{(t)}$ in one iteration of loop x spans a memory region of $(2n - 1) \times 576$ kB, or 6.336 MB for $n = 6$, which is on the order of the shared data cache of most quadcore architectures. Assuming that each 3-dimensional array is allocated in a contiguous region of memory, two elements $u^{(t)}_{i,j,k}$ and $u^{(t)}_{i,j,k+1}$ are 576 kB apart. A few iterations of loop z touch more than a memory page, even for the large page size of the Intel Nehalem.

To take advantage of the reuse available in the SC, we use code transformations that improve locality. Loop tiling is a code transformation that reorders loop iterations to bring accesses to the same data, cache line and memory pages closer in time. We apply loop tiling to SC targeting different levels of memory hierarchy: TLB, last level cache (LLC), and SIMD register file. Before tiling, we apply loop unrolling to the three loops at the innermost level (i_n, j_n, and k_n) of Fig. 2 with a factor of $2n$, i.e., we fully unroll the loops. Line 5 in Fig. 5 represents the computation in the loop body after unrolling.

Fully unrolling loop body eliminates loop overhead and exposes more data reuse in the loop body that we will exploit using SIMD registers. The resulting code is a 3-deep loop nest, as shown in Fig. 5.

TLB: To reuse more data in each memory page and decrease the number of TLB misses, we apply tiling to loop k. Tile size *zTile* should be chosen such that the number of pages touched by loop iterations within the tile is smaller than the number of TLB entries.

LLC: The 8 MB L3 cache of our experimental platform (Intel Nehalem) can keep 13 "planes" of grid points, where a plane is a region of 384×384 elements (576 kB) along dimensions y and x (each value of z corresponds to a plane in (y, x)). However, to exploit reuse in all three dimensions, we tile loop j so that the data accessed within a tile of size *zTile* $\times$ *yTile* fits in the L3 cache (the data includes elements of $u^{(t)}$, $u^{(t+1)}$ and other temporary data required by the computation). Note that, although in this section we discuss locality optimizations in the context of a single thread on a single core, when combining these optimizations with parallelism, the tile sizes should be chosen carefully to prevent conflicts in the shared cache, i.e., L3 in 3-level cache Intel Nehalem (L2 in earlier quadcore architectures).

Figure 6 shows the SC after loop tiling is applied to loops k and j. In Fig. 6, macro *MIN* returns the smaller of its parameters, so that the two higher-stride directional

```
1: procedure tiledHOSC(n,n_x,n_y,n_z,c,u^(t),u^(t+1))
2:     for k = n to n_z - n by zTile do
3:     for j = n to n_y - n by yTile do
4:       for kk = k to MIN((kk + zTile),(n_z - n)) do
5:       for jj = j  to MIN((jj + yTile),(n_y - n)) do
6:           for ii = n to n_x - n do
7:               u^(t+1)_(ii,jj,kk) ← f(c,neighbor(u^(t)_(ii,jj,kk)))
8:           end for
9:       end for
10:      end for
11:    end for
12:    end for
13: end procedure
```

Fig. 6 Pseudocode for tiling implementation

loops, i.e., y and z, are subdivided into smaller loops of size *zTile* and *yTile*. The tiling parameters, *zTile* and *yTile*, should be large enough to amortize the cost of the added loops but the size of the tiles should not exceed physically available cache size. In fact, for a 3-dimensional SC with m physical quantities (e.g., pressure and/or temperature) of type *datatype* per grid point, the tiling parameters should be chosen such that $yTile \times zTile \times m \times n \times sizeof(datatype) \leq$ effective cache size, where we refer to effective cache size instead of physical size to account for unoccupied cache slots due to memory mapping rules.

In contrast to some existing tiling approaches that tile all three loops [28], the pseudocode in Fig. 6 does not apply tiling to the unit-stride dimension. Our tests have shown that tiling the two high-stride dimension loops is enough to have the most reuse across tiles. In fact, tiling all three loops expands the tile boundaries and cannot amortize the increased number of tile execution. This comes from the fact that in stencil codes, it is only required to preserve the group reuse at $2n + 1$ distance.

Code generation and tuning: We use a code transformation tool, CHiLL [29], to generate optimized code variants of the SC. CHiLL is a transformation framework that allows the user to specify code transformations using a high-level script interface. CHiLL supports code transformations such as loop tiling, interchanging, unrolling, fusion and distribution, data copying, and data prefetching. CHiLL takes as input the original code and a script specifying code transformations, and automatically generates a transformed code. Optimization parameters such as tile sizes can be specified as integer values or as unbound parameters to be determined later. In our optimized code variant (*stencil_2dTiling*), loops z and y are tiled with unbound parameters *zTile* and *yTile*. We use empirical search to determine the optimum tile sizes *zTile* and *yTile*, RB, and SIMD parallelization.

3.4 Register blocking

In SC, it is not possible to have small strides for all directions of computation at the same time. For example, the stride in the z direction is 576 kB in our kernel, whereas the x direction has a unit stride, i.e., 4 bytes.

To exploit SIMD parallelism and spatial reuse in the x direction, we rearrange the computation to perform updates to the grid points that are contiguous in the

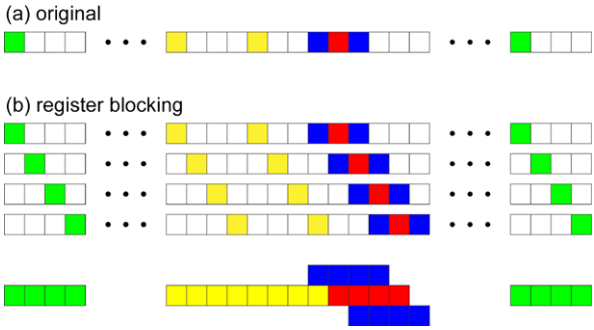

Fig. 7 Memory access pattern of register blocking compared to the original access pattern

SIMDized code. Figure 7 schematically represents our approach to RB for HOSC. In the original kernel in Fig. 7(a), the red square indicates target grid points to be updated at certain time. Blue, yellow, and green squares respectively show memory locations of the nearest neighbor points in x, y, and z Cartesian coordinates. During each iteration of the loop ii (in Fig. 6), the red square is updated by accessing all neighbors (not necessarily the nearest neighbors, and the stencil order determines the distance of the furthest accessed neighbor.) Memory stride for each direction is 1, n_x, and $n_x \times n_y$, respectively, using the same notation as in Fig. 6. In contrast to the original access pattern shown in Fig. 7(a), RB deals with a chunk of target grid points contiguous in the x direction. The same size of neighboring cells is fetched to update the block of target grids, which maximally utilize registers (see Fig. 7(b)).

In a more abstract level, with the RB technique, we load blocks of memory to machine register files. In RB implementation, instead of computing $u^{(t+1)}_{i,j,k}$ one by one, we accumulate the contributions of $u^{(t)}_{i,j,k+n}$, $u^{(t)}_{i+1,j,k+n}$, $u^{(t)}_{i+2,j,k+n}$, $u^{(t)}_{i+3,j,k+n}$ on $u^{(t+1)}_{i,j,k}$, $u^{(t+1)}_{i+1,j,k}$, $u^{(t+1)}_{i+2,j,k}$, $u^{(t+1)}_{i+3,j,k}$, where both of 4 float blocks are contiguous and 16 bytes aligned in memory, and can be packed into 16 bytes SIMD registers. We manually implement RB using Intel SSE3 intrinsics. RB eliminates 3 memory accesses to the neighboring grid points per 16-byte block and accordingly increases performance. It should also be noted that the compiler fails to generate code using Intel SSE instructions for floating point operations in the tiling mode due to the complex loop body of the high-order stencil. Therefore, RB enhances the opportunity for better instruction scheduling with increased instruction level parallelism.

4 Performance test results

Inter-node scalability tests have been carried out on (i) Intel Xeon and AMD Opteron platforms at the High Performance Computing and Communications (HPCC) Center of the University of Southern California (USC) and (ii) the IBM BlueGene/P at Argonne National Laboratory. The Intel dual quadcore Xeon E5420 (Harpertown) processors are clocked at 2.5 GHz, featuring a 4×32 kB 8-way set associative L1 data and instruction cache and 2×6144 kB 24-way set associative L2 cache. The AMD dual dualcore Opteron 270 is clocked at 2 GHz with 128 kB of L1 and 1 MB L2

cache per core. The Xeon platform deploys 12 GB memory and 10-gigabit Myrinet interconnects, and the Opteron platform 4 GB memory with 2-gigabit Myrinet. The BlueGene/P has four nodes on a chip, where each node has 2 GB DDR2 DRAM and four 450 POWER PC processors clocked at 850 MHz, featuring a 32 kB instruction and data L1 cache, a 2 kB L2 cache and a shared 8 MB L3 cache. BlueGene/P also allows users to specify arrangement of MPI processes to make use of its 3D torus network topology. We have examined $TXYZ$ and $XYZT$ mapping orders, where XYZ represents 3D indices in the torus network while T $(= 0, 1, 2, 3)$ corresponds the number of cores on one node. For a spatial decomposition scheme, though the best performance is expected by mapping process arrangement exactly on the torus structure [30], $TXYZ$ mapping often performs better, retaining more communications locally.

We perform our single-processor and single-core benchmarks on Intel Nehalem (core i7 920), which is clocked at 2.67 GHz. On a single-die quadcore Nehalem, there is 8 MB of shared L3 cache, 256 kB of L2 cache per core and 64 kB of L1 cache (divided into a 32 kB instruction cache and a 32 kB data cache). Intel Nehalem drops the front side bus (FSB) in favor of Intel QuickPath Interconnect (QPI) and, in doing so, brings the memory controller on-die. Cores on the Intel Nehalem die communicate through QPI that offers 25.6 GB/s of bandwidth, which is more than twice the theoretical bandwidth of Harpertown's FSB. Intel Nehalem's cores are capable of Simultaneous Multi-Threading (SMT), i.e., each core can execute 2 threads simultaneously, opposed to single thread per core on Harpertown.

In addition, we investigate the effect of NUMA on multithreading performance. We perform multithreading benchmarks with and without NUMA control on a dual 12-core Magny-Cours AMD Opteron 6172 machine. A single Magny-Cours Opteron combines two 6-core dies (or processors) on an MCM package. The cores are clocked at 2.1 GHz with 128 kB/core for L1 cache (64 kB for instruction and 64 kB for data), 512 kB/core L2 cache, and share a 6 MB/die L3 cache.

4.1 Strong scaling

Algorithms harnessing sound strong-scalability may accelerate overall application performance by increasing the number of processors, and are desirable to fully utilize many-core architectures. Here, we define the strong-scaling speedup as the ratio of the time to solve a problem on one processor to that on p processors for the same problem size. Figure 8(a) shows the total execution and communication times on BlueGene/P with $TXYZ$ and $XYZT$ network mappings as a function of the number of processors. Figures 8(b) and 8(c) plot strong-scaling speedup on the three platforms described above. Measurements are done with a fixed global problem size of 400^3 grid points. We obtain good overall strong-scalability by increasing processor count on all platforms. Furthermore, we observe superlinear speedups on the Intel architecture. This may be explained as an effect of aggregate cache size as discussed below.

Sequential or shared-memory implementations suffer from a large memory footprint per process. For example, our FDTD application uses 384^3 grid points (1.13 GB data in total), which is well beyond the current cache sizes. The spatial decomposition

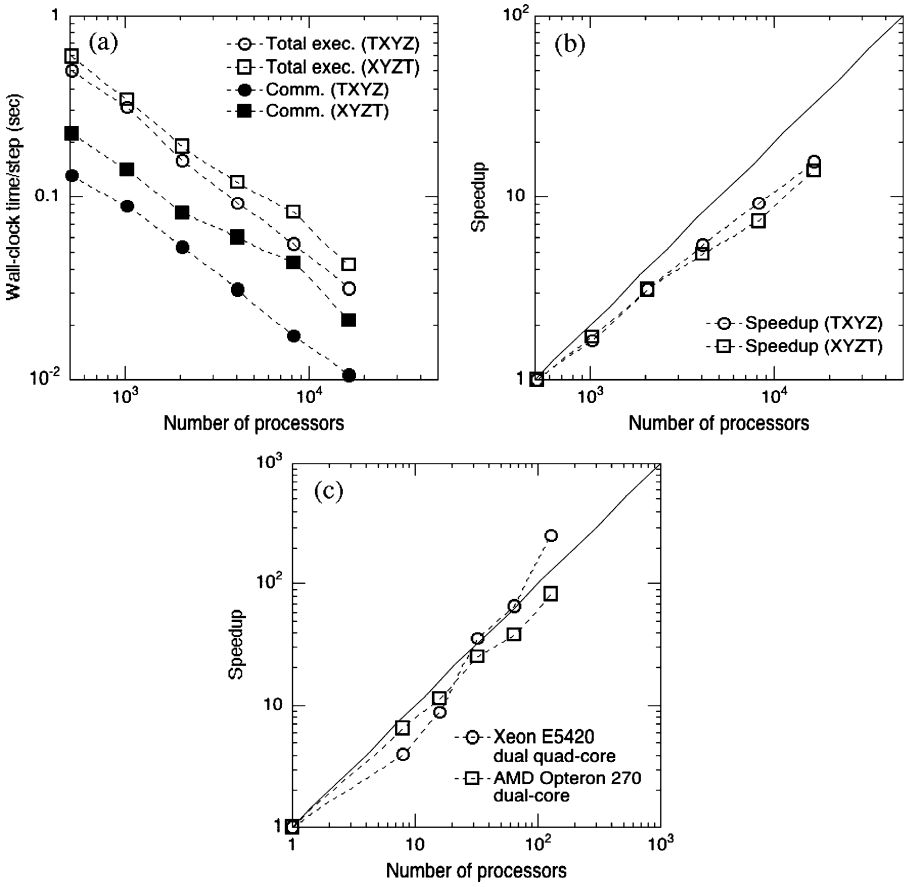

Fig. 8 Strong-scaling benchmark on BlueGene/P, Intel Xeon E5420 and AMD Opteron 270 based clusters. (**a**) The wall-clock time as a function of the number of processors (up to 32,768) of BlueGene/P. (**b**) Speedup compared to 512 core performance. (**c**) Speedup compared to single core performance on the Intel Xeon E5420 and AMD Opteron 270 platforms. The Intel Xeon E5420 exhibits superlinear speedup for relatively small number of processors

scheme avoids this problem by increasing the processor count, and thus decreasing the volume of each subdomain. Substantial performance gain is expected, when the sub-domain size becomes small enough to fit into the cache. We use the Intel Vtune Performance Analyzer to monitor cache and TLB misses in the original FDTD code. We find that high-stride computation accounts for more than 25 % of core cycles throughout the thread execution during page-walks, indicating that a high TLB miss rate results in greater effective memory access time. The cache effect is most pronounced in the benchmark on Intel Xeon E5420 (Harpertown) architecture that features a shared 6 MB L2 cache per chip that accommodates two cores. This amounts to 12 MB of cache per MCM, 6 times more than 2 MB (2×1 MB) L2 cache on AMD Opteron. Therefore, Harpertown outperforms AMD Opteron and shows the superlinear-scaling with the processor counts over 32 (see Fig. 8(c)).

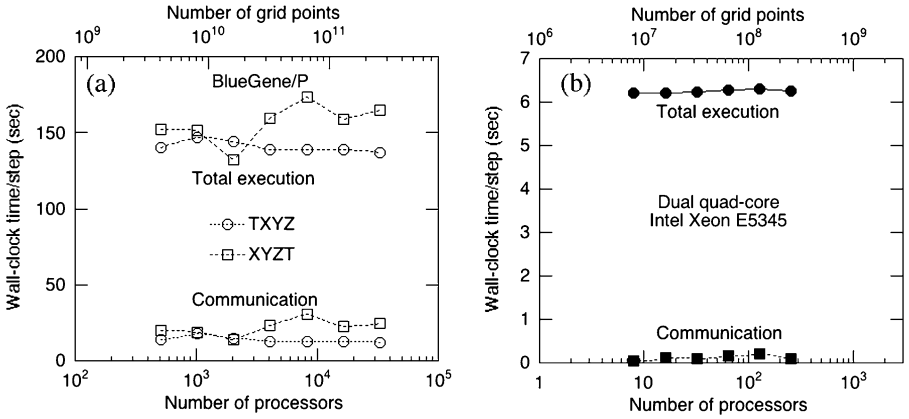

Fig. 9 Weak-scaling performance on (**a**) BlueGene/P with 200^3 grid points per process and (**b**) a dual quadcore Intel Xeon E5345 cluster with 100^3 grid points per process

4.2 Weak scaling

Next, we test weak-scaling parallel efficiency of our parallel SC. We define the weak-scaling parallel efficiency as the running time on 1 processor divided by that on p processors for a *fixed problem size per processor*. Figures 9(a) and 9(b) plot the total execution time and communication time per SC step on the BlueGene/P and Intel Xeon E5345 (Intel Clovertown) platforms. The quadcore Intel Clovertown architecture is clocked at 2.33 GHz and features 256 kB/8 MB of L1/L2 cache.

In Fig. 9, the number of grid points per process (i.e., grain size) is 200^3 (160 MB/process) for BlueGene/P and 100^3 (20 MB/process) for the Intel Clovertown cluster. We observe excellent weak-scalability, nearly constant performance up to 256 processors on the Intel Clovertown-based cluster and 32,768 processors on BlueGene/P with TXYZ network mapping. The XYZT network mapping shows fluctuations in the total and communication times which is due to higher network resource sharing with other processes. The sharing is more effective for the XYZT mapping since the TXYZ topology closely matches our algorithmic spatial decomposition scheme.

4.3 Multithreading

In addition to the massive inter-node scalability demonstrated above, our parallelization strategy involves the lower levels of optimization: First, we use multithreading explained in Sect. 3.2, implemented with POSIX thread. Next, we vectorize the loop for construction of $packed\_u_k^{(t)}$ and computation of the stencil $f(c, neighbor(packed\_u_k^{(t)}))$ inside the triply nested for loops in Fig. 4 by using SSE3 intrinsics on the Intel Nehalem platform. We use Intel C compiler (icc) version 11 with O3 optimization level for our benchmarks.

Figure 10(a) shows the reduction in clock time spent per simulation step due to multithreading and SIMD optimizations. Corresponding speedups are shown in

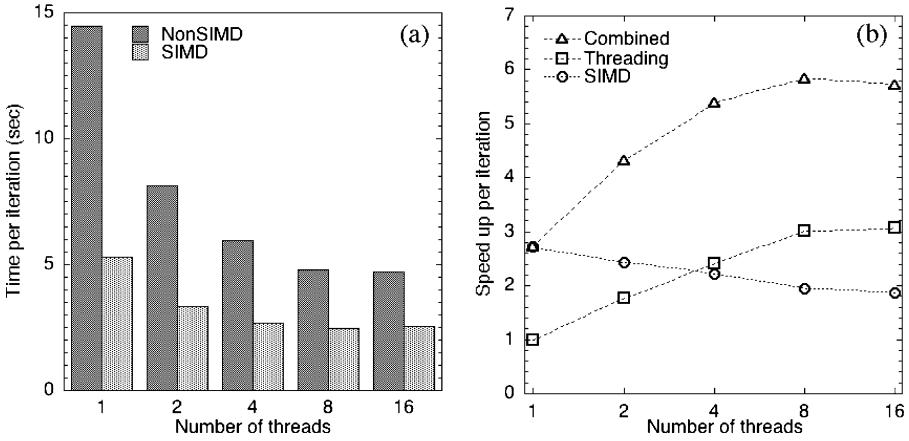

Fig. 10 (**a**) The wall-clock time per iteration for non-SIMD and SIMDized codes as a function of the number of threads on quadcore Intel Nehalem. (**b**) Breakdown of the speedups due to multithreading and data-level parallelism along with the combined intra-node speedup. The best observed intra-node speedup is 5.83 with 8 threads on 4 cores

Fig. 10(b). To delineate the performances of multithreading and SIMDization, we define a performance metric as follows. We use the clock time for one simulation step of the single threaded, non-vectorized code to be the sequential run time T_s. We denote the parallel run time, $T_p(NUM\_THREADS)$, to be the clock time required for execution of one time step of the algorithm as a function of spawned thread number, in presence of both multithreading and SIMD optimizations. Then combined speedup, S_c, shown in Fig. 10(b) is the ratio T_s/T_p as a function of the number of threads. We remove SIMD optimizations to quantify the effect of multithreading only, and measure the parallel running times for a variety of thread numbers, and state the multithreading speedup, S_t, with respect to T_s. Finally, we attribute the excess speedup, S_c/S_t, to SIMD optimizations.

Figure 10(b) shows the best speedup of 5.83 for 8 threads on a single quadcore Intel Nehalem node. It should be noted that multithreading speedup continues to increase until 8 threads on 4 cores. This is because Intel Nehalem cores feature simultaneous multi-threading technology (SMT) that enables each core to run two threads at the same time. Increased L3 cache sharing and thread management cost dominates at 16 threads to yield moderate performance degradation.

Figure 11 shows the multithreading speedup with and without NUMA control as the number of threads varies from 1 to 48. When NUMA control is used (red line), the speedup steadily increases with the number of threads and reaches the peak speedup at 9.9-fold when 24 threads are used. When 25 threads are spawned, the speedup sharply drops to 8.3-fold before gradually increasing to 9.3-fold at 48 threads. This sharp drop signifies that the thread management cost affects the performance adversely if more than 24 threads are spawned when only 24 physical cores are available. Unlike Intel Nehalem processor, in Magny-Cours architecture the performance gain from memory latency hiding via redundant threading is dominated by the context-switching penalty when more than 1 thread per core is spawned.

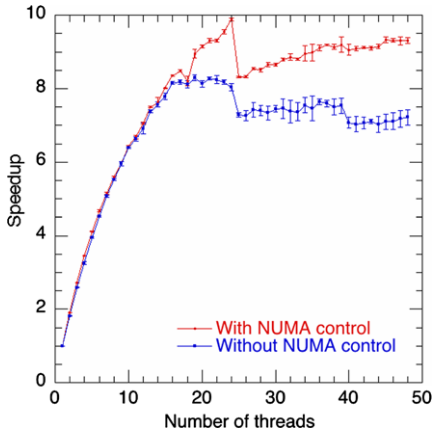

Fig. 11 Multithreading speedup with and without NUMA control on dual 12-core AMD Opteron system

On the other hand, multithreading without NUMA control (blue line) on AMD Magny-Cours shows inferior speedup compared to the one with NUMA control. When the number of threads is less than 18, the speedup without NUMA control is 2.3 % less on average. However, when 19 to 24 threads are used, the speedup without NUMA saturates and is clearly below speedup with NUMA control (12.3 % slower on average). A dramatic drop is observed when 25 threads are used, which is similar to the speedup with NUMA control. Performance without NUMA control is significantly less than the performance with NUMA control (18.1 % slower on average) when 25 or more threads are used. Also, the error margins, which show the standard deviation of speedup over 10 runs, for the case without NUMA is larger reflecting the stochastic nature of memory access.

The best speedup achieved without using NUMA control is 8.3-fold at 19 threads, which is 16.1 % slower compared to the best speedup with NUMA control. On average, NUMA control improves speedup by 11.5 %. This result indicates the benefit of NUMA control on NUMA architectures.

4.4 Single-core performance

We use a script to empirically search for the optimal values of *yTile* and *zTile* parameters to maximize the floating point performance of *stencil_2dTiling* and its variant featuring RB. In this empirical method, the parameter space is defined by tile sizes $\langle yTile, zTile \rangle$, bounded by cache and TLB capacity constraints. Our tiling approach essentially keeps cache lines closer to the core (in cache). All measurements are performed using one core of Intel Core i7 920 CPU. Table 2 details the cache performance improvement because of our tiling implementation. As a reference, Table 2 lists the performance of the experimental kernel that incorporates loop-unrolling transformations labeled as original.Each row shows the number of CPU events per 1,000 retired instructions, i.e., the executed non-speculative instructions that are actually needed by the program flow. The data in the rows are normalized with respect to 1,000 retired instructions during the code execution. The first row shows the number of retired loads that miss the LLC per 1,000 retired instructions at Intel Nehalem

Table 2 Performance comparison of original and tiled codes

Per 1000 instructions	Original	2dTiling	Improvement
LLC_MISS	0.42	0.054	7.77
Local DRAM	0.54	0.115	4.69
DTLB	0.41	0.084	4.88

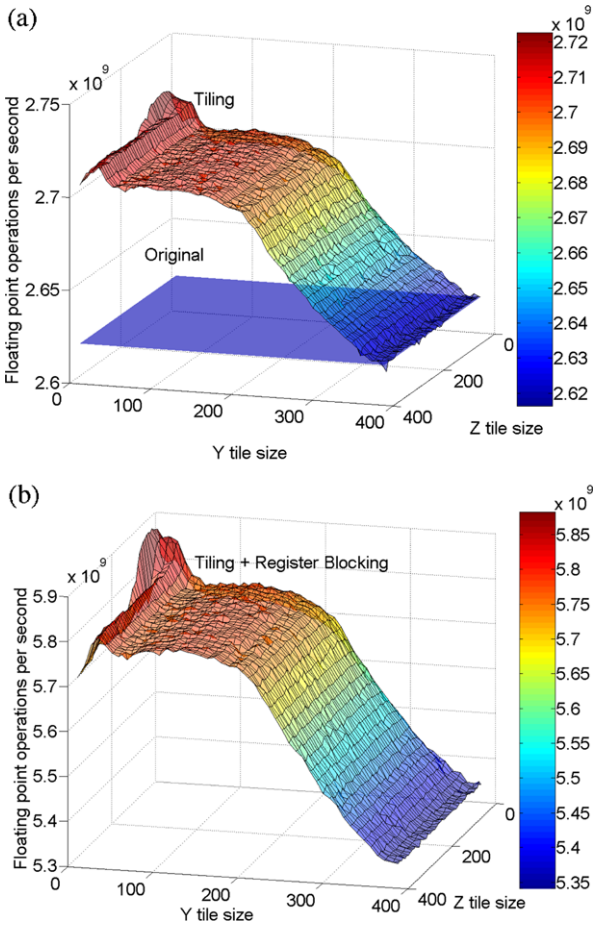

Fig. 12 Floating-point operations per second performance at Intel Nehalem. (**a**) Comparison of the original and tiled codes. (**b**) The performance of tiling for TLB, cache and registers with SIMD implementation. The best single core performance is 5.9 GFlops

core. We observe a 7.7-fold improvement in the LLC miss rate for the tiling implementation with the best tile sizes used. The second row shows number of memory load instructions retired where the memory reference missed the L1, L2, and LLC caches and required a local socket memory reference per 1,000 retired loads during the execution. We see that the tiling version of the code shows improvement by decreasing local memory access rate by a factor of 4.6. The third row shows the number of retired loads that missed the DTLB per 1,000 retired instructions. We also observe that tiling improves DTLB miss per 1,000 instructions by 4.8-fold. We collect perfor-

Table 3 Comparison of original and optimized code at the assembly level

Assembly for 1 float multiply from original code
1: movss 0160 (%rsp), %xmm14
2: mulss %xmm14, %xmm13
3: addss %xmm13, %xmm10
4: unpcklps %xmm8, %xmm8
5: movlhps %xmm8, %xmm8
6: movups 010 (%rax, %r13, 4), %xmm11
7: mulps %xmm8, %xmm11
8: addps %xmm11, %xmm14
9: movss 0210 (%rsp), %xmm6
10: mulss %xmm6, %xmm8
Assembly for 4 float vector multiply in SIMD code variant
1: movaps (%rdx), %xmm10
2: movaps 010 (%rbp, %rax, 4), %xmm0
3: mulps %xmm0, %xmm10

mance numbers via Intel VTune Performance Analyzer. The Linux kernel version in our system is 2.6.28-19 (LLC patch for Intel Nehalem is applied).

Figure 12(a) shows the variation of the performance (in terms of the floating-point operations per second) of *stencil_2dTiling* with tile sizes compared with that of the original loop-unrolling transformation code. We observe the dominant effect of y tile size, with the smaller y tile size achieving the better performance. The best observed performance is 2.72 Gflops, whereas the theoretical peak performance is 2.67×10^9 cycles/second $\times$ 4 flops/cycle $=$ 10.68 Gflops. This corresponds to 25 % of the peak performance. Auto-tuning through CHiLL generates similar flops performance.

Figure 12(b) shows the tile-size dependency of the performance of the code variant incorporating both tiling and RB optimizations with the explicit use of SSE instructions. The best performance we have achieved is 5.9 Gflops, which corresponds to over 55 % of the theoretical peak performance.

The above results show more than 2-fold improvement by the SSE featuring code variant with respect to the tiling version. This performance enhancement may be attributed to the effective use of Intel SSE instructions for floating-point operations. In fact, we use Intel C compiler (icc) version 11 for all codes with the best optimization level (-O3), but the compiler fails to vectorize the nested loop body for the HOSC. Our analysis with VTune Performance Analyzer reveals that the SIMD variant reduces the number of instructions that retire the execution by more than 2-fold with respect to both original and tiling versions of the same code. To better understand this, we have examined the assembly codes for floating-point operations inside the loop body. Table 3 shows a typical assembly code for a single floating-point multiply generated from the original code and vector multiply of a four packed single-precision floating-point values in the SIMD code variant. Both codes load coefficient c and grid value $u^{(t)}_{i,j,k}$ and perform a multiply operation. Original code executes extra instructions to calculate array indices whereas SIMD variant uses 1 array index calculation

per 4 floats multiply. Note the use of *unpcklps* and *movlhps* in the original code to unpack and move single-precision floating-point values, and use of *movups* for moving unaligned data. Also one single-precision floating-point multiply is performed by *mulss* instruction. In the SIMD code variant, *movaps* is used to load a 128-bit memory location to an XMM register at once, i.e., aligned load operation loads 4 float values at a time, then performs a packed multiply, *mulps*, to concurrently multiply 4 packed floats.

5 Summary

We have developed a multilevel optimization scheme for high-order stencil computations that combines: (i) inter-node parallelization via spatial decomposition; (ii) intercore parallelization via multithreading; (iii) data locality optimizations through autotuned tiling for efficient use of hierarchical memory; and (iv) RB and data parallelism via SIMD techniques to utilize registers and exploit data locality. We have applied our optimization scheme to a sixth-order stencil based FDTD code. Our benchmarks on 32,768 BlueGene/P processors achieved over 98 % weak-scaling parallel efficiency. We have also observed superlinear strong scaling due to increasing aggregate caches on an Intel Xeon cluster. RB+multithreading optimizations have achieved 5.8-fold speedup on a single quadcore Intel Nehalem Core i7 920. A speedup of 9.9-fold is obtained on a dual Magny-Cours Opteron with explicit NUMA control. Data locality optimizations achieve 7.7-fold reduction of LLC miss rate on Intel Nehalem, whereas RB increases data parallelism and thereby achieving 5.9 Gflops performance on a single core, which is over 55 % of its peak performance.

Acknowledgements This work was partially supported by NSF PetaApps/EMT/CMMI, DOE SciDAC/SciDAC-e/BES/INCITE, and DTRA. Performance tests were carried out at the Collaboratory for Advanced Computing and Simulations and High Performance Computing Center of the University of Southern California. This research also used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

1. Barker KJ, Davis K, Hoisie A, Kerbyson DJ, Lang M, Pakin S, Sancho JC (2008) Entering the petaflop era: the architecture and performance of Roadrunner. In: Proceedings of the 2008 international conference for high performance computing, networking, storage and analysis, Austin, Texas. IEEE Comput Soc, Los Alamitos
2. Carrington L, Komatitsch D, Laurenzano M, Tikir MM, Michea D, Goff NL, Snavely A, Tromp J (2008) High-frequency simulations of global seismic wave propagation using SPECFEM3D_GLOBE on 62K processors. In: Proceedings of the 2008 international conference for high performance computing, networking, storage and analysis, Austin, Texas. IEEE Comput Soc, Los Alamitos
3. Komatitsch D, Erlebacher G, Göddeke D, Michéa D (2010) High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. J Comput Phys 229:7692–7714
4. Zhao S, Wei GW (2004) High-order FDTD methods via derivative matching for Maxwell's equations with material interfaces. J Comput Phys 200:60–103
5. Nguyen A, Satish N, Chhugani J, Kim C, Dubey P (2010) 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In: Proceedings of the 2010 international conference for high performance computing, networking, storage and analysis, New Orleans, Louisiana. IEEE Comput Soc, Los Alamitos

6. Wellein G, Hager G, Zeiser T, Wittmann M, Fehske H (2009) Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In: Proceedings of the 2009 IEEE international computer software and applications conference, Seattle, Washington. IEEE Comput Soc, Los Alamitos
7. Williams S, Carter J, Oliker L, Shalf J, Yelick K (2009) Optimization of a lattice Boltzmann computation on state-of-the-art multicore platforms. J Parallel Distrib Comput 69:762–777
8. Datta K, Murphy M, Volkov V, Williams S, Carter J, Oliker L, Patterson D, Shalf J, Yelick K (2008) Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 international conference for high performance computing, networking, storage and analysis, Austin, Texas. IEEE Comput Soc, Los Alamitos
9. Peng L, Seymour R, Nomura K, Kalia RK, Nakano A, Vashishta P, Loddoch A, Netzband M, Volz WR, Wong CC (2009) High-order stencil computations on multicore clusters. In: Proceedings of the 23rd IEEE international parallel and distributed processing symposium, Rome, Italy. IEEE Comput Soc, Los Alamitos
10. Rivera G, Tseng C-W (2000) Tiling optimizations for 3D scientific computations. In: Proceedings of the 2000 international conference for high performance computing, networking, storage and analysis, Dallas, Texas. IEEE Comput Soc, Los Alamitos
11. Frigo M, Strumpen V (2005) Cache oblivious stencil computations. In: Proceedings of the 2005 international conference on supercomputing, Cambridge, Massachusetts. IEEE Comput Soc, Los Alamitos
12. Wonnacott D (2000) Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In: Proceedings of the 14th IEEE international parallel and distributed processing symposium, Cancun, Mexico
13. Renganarayana L, Harthikote-Matha M, Dewri R, Rajopadhye SV (2007) Towards optimal multi-level tiling for stencil computations. In: Proceedings of the 21st IEEE international parallel and distributed processing symposium, Long Beach, California. IEEE Comput Soc, Los Alamitos
14. Dursun H, Nomura K, Peng L, Seymour R, Wang W, Kalia RK, Nakano A, Vashishta P (2009) A multilevel parallelization framework for high-order stencil computations. In: Proceedings of the 15th international Euro-Par conference on parallel processing, Delft, The Netherlands. Springer, Berlin
15. Shen G, Cangellaris AC (2007) A new FDTD stencil for reduced numerical anisotropy in the computer modeling of wave phenomena: research articles. Int J RF Microw Comput-Aided Eng 17:447–454
16. Nakano A, Vashishta P, Kalia RK (1994) Multiresolution molecular dynamics algorithm for realistic materials modeling on parallel computers. Comput Phys Commun 83:197–214
17. Parker M, Ketcham S, Cudney H (2007) Acoustic wave propagation in urban environments. In: Proceedings of the 2007 DoD high performance computing modernization program users group conference, Pittsburgh, Pennsylvania. IEEE Comput Soc, Los Alamitos
18. Dang DM, Christara CC, Jackson KR (2010) Pricing multi-asset American options on graphics processing units using a PDE approach. In: 3rd workshop on high performance computational finance (WHPCF), in conjunction with the 2010 international conference for high performance computing, networking, storage and analysis, New Orleans, Louisiana. IEEE Comput Soc, Los Alamitos
19. PARKBENCH: PARallel kernels and BENCHmarks. Available from http://www.netlib.org/parkbench
20. Bailey D, Barton J, Laninski T, Simon H (1991) The NAS parallel benchmarks. NASA Ames Research Center, Moffett Field
21. Bromley M, Heller S, McNerney T, Steele JGL (1991) Fortran at ten gigaflops: the connection machine convolution compiler. In: Proceedings of the ACM SIGPLAN 1991 conference on programming language design and implementation, Toronto, Ontario, Canada. ACM, New York
22. Roth G, Mellor-Crummey J, Kennedy K, Brickner RG (1997) Compiling stencils in high performance Fortran. In: Proceedings of the 1997 international conference for high performance computing, networking, storage and analysis, San Jose, CA. IEEE Comput Soc, Los Alamitos
23. Bordawekar R, Choudhary A, Ramanujam J (1996) Automatic optimization of communication in compiling out-of-core stencil codes. In: Proceedings of the 1996 international conference for high performance computing, networking, storage and analysis, Philadelphia, Pennsylvania, United States. IEEE Comput Soc, Los Alamitos
24. Ramanujam J, Krishnamurthy S, Hong J, Kandemir M (2002) Address code and arithmetic optimizations for embedded systems. In: Proceedings of the 2002 conference on Asia south pacific design automation/VLSI design, Bangalore, India. IEEE Comput Soc, Los Alamitos

25. Shimojo F, Kalia RK, Nakano A, Vashishta P (2008) Divide-and-conquer density functional theory on hierarchical real-space grids: parallel implementation and applications. Phys Rev B, Condens Matter Mater Phys 77:085103
26. Stathopoulos A, Öğüt S, Saad Y, Chelikowsky JR, Kim H (2000) Parallel methods and tools for predicting material properties. Comput Sci Eng 2:19–32
27. Snir M, Otto S (1998) MPI: the complete reference: the MPI core. MIT Press, Cambridge
28. Lam MS, Wolf ME (2004) A data locality optimizing algorithm. ACM SIGPLAN Not 39:442–459
29. Chen C, Chame J, Hall M (2008) CHiLL: a framework for composing high-level loop transformations. USC computer science technical report
30. IBM (2008) IBM system Blue Gene Solution: Blue Gene/P application development