Parallelization Methods for Hierarchical SMP Systems

Larry Meadows(✉), Jeongnim Kim, and Alex Wells

Intel Corporation, Hillsboro, OR, USA
{lawrence.f.meadows,jeongnim.kim,alex.m.wells}@intel.com

Abstract. We discuss several parallelization methods for multi-level hierarchical SMP systems using a stencil-based finite difference code. Performance comparisons and suggestions for OpenMP runtime improvements are provided.

Keywords: Stencil · Nested parallelism · Runtime support

1 Introduction

Modern symmetric multi-processors (SMPs) have multiple levels of memory hierarchy and multiple levels of parallelism. In this paper we explore various methods to exploit those multiple levels including OpenMP, nested OpenMP, OpenMP 4 teams/distribute, and a higher-level C++ template library called SIMD building blocks (SBB). Additionally we explore various methods of load balancing including manual load balancing and the OpenMP `collapse` clause. As a result of these experiments we offer suggestions for OpenMP implementors.

We use the diffusion test code from [1]. Our work shows alternatives to the plesiochronous barriers used in Chap. 5 of [2], some of which may be more understandable to and usable by most OpenMP programmers.

2 The Test Code

The diffusion test code (hereafter referred to as just *diffusion*) is a simple 7-point stencil code in three dimensions, shown in Fig. 1.

The diffusion kernel is memory bandwidth bound. To see this we can compute the ratio of floats (or doubles) accessed to floating point operations. Each iteration has 7 loads, 1 store, 7 multiplies, and 6 adds. There are thus 13 floating point operations and 8 memory accesses, resulting in a ratio of about 2.5 bytes/flop (single precision). As an example, the current generation Intel® Xeon Phi™ coprocessor[1] has a peak floating point performance on the order of 1000E9 flops/second and a memory bandwidth on the order of 170E9 bytes/second,

[1] Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

```
for (int z = 0; z < N; ++z)
  for (int y = 0; y < N; ++y)
    for (int x = 0; x < N; ++x)
      f2[z][y][x] = cc * f1[z][y][x] +
                    cw * f1[z][y][x-1] +
                    ce * f1[z][y][x+1] +
                    cn * f1[z][y-1][x] +
                    cs * f1[z][y+1][x] +
                    cu * f1[z-1][y][x] +
                    cd * f1[z+1][y][x];
```

Fig. 1. Diffusion psuedocode

which is only about 0.17 bytes/flops, far less than required. Thus, our optimization efforts are focused on memory optimizations.

There are two memory optimizations: achieving maximum memory bandwidth from the processor, and exploiting reuse by tiling for cache. The former is largely done by the compiler, though we do use non-temporal stores (which are particularly helpful on the Intel® Xeon Phi™ coprocessor. The latter is accomplished in different ways depending on the particular code version.

Reuse occurs because of the ±1 subscript arithmetic. For the contiguous (unit-stride) X dimension, the reuse occurs automatically (spatial reuse). For the Y and Z dimensions, the current iteration reuses two elements from the previous iteration (temporal reuse); Y becomes $Y - 1$, and $Y + 1$ becomes Y. The key is to tile the loops so that the previous elements from the Y and Z loops remain in cache.

3 SIMD Building Blocks

SIMD Building Blocks (SBB) is a C++11 template library providing concepts of Containers, Accessors, Kernels, and Engines to abstract out different aspects of creating an efficient data parallel (SIMD + threading) program. The Containers encapsulate the memory data layout of an Array of "Plain Old Data" objects. Kernels represent the work inside a loop body and use Accessors with an array subscript operator (just like C++ arrays) to read from or write to the objects in the Containers. Engines visit a Kernel over an iteration space. Since these concepts are abstracted out, multiple concrete versions can exist and can encapsulate best known methods, thus avoiding common pitfalls in generating efficient SIMD code.

For example, when speaking of efficient SIMD code, the terms "Array Of Structures" (AOS) and "Structure Of Arrays" (SOA) are often used. In order to utilize SIMD load/store instructions, the data must be in a SOA so that a vector register can be loaded with the same data members, instead of having to emit instructions to load each data lane separately or having to perform a gather over AOS data. However most object oriented code uses data in an AOS format.

Changing one's algorithms to work with SOA is cumbersome and difficult to maintain. With SBB, one can use an `SoaContainer` < `Object` > and the data will be stored in memory as SOA, but a kernel just sees an instance `Object`. This allows kernels to just work with the `Objects` and implement their algorithm and leave the complexities of SOA and data alignment to the container. SBB provides multi-dimensional 2d and 3d containers as well, with the added benefit of handling address calculation of multiple index variables through an accessor with multiple array subscript operators (just like a 2d or 3d c++ array). This often yields simpler kernel code.

Likewise the Engines abstract out iteration. Engines can be declared to generate scalar or SIMD code, to run single threaded, or with Intel® Threading Building Blocks (Intel® TBB), or with OpenMP threading. When an Engine runs a Kernel, it is given an iteration space and blocking size. The concrete engine can then divide the iteration space up into blocks and execute the Kernel over the blocks. 2d and 3d Engines are provide good cache blocking behavior out of the box. Switching threading models is as easy as changing a typedef and users can choose to make their own Engines that fit in SBB's framework (e.g., a team based threading Engine that uses threads on the same core to cooperatively work on the same block, as described in the next section).

4 Nested Threading

Intel® Xeon® processors and the Intel® Xeon Phi™ coprocessor consist of multiple cores. Each core has multiple hardware thread contexts, often called hyperthreads or simultaneous multi-threading (SMT) threads. The thread contexts each have their own register sets and other state but share all of the execution units and caches on the core.

In many cases it is helpful to have a two-level nested parallel structure that corresponds to the hardware threading structure. The outer level corresponds to the cores, and the inner level corresponds to the hardware threads within a core. The outer level determines the data decomposition (implicitly or explicitly), which in turn determines the data that resides in a particular core's caches. The inner level threads then cooperate on the data residing in the shared caches. This often reduces cache pressure since decomposition is per core, not per thread, and can lead to substantial speedups.

5 Code Variants

The following subsections describe the code variants used in the performance study. All of the codes use a common inner loop expressed in an inline function. The code for the inner loop is shown in Fig. 9 near the end of the paper.

The code in Fig. 9 incorporates some improvements from [2], namely alignment, streaming stores, and special treatment of the two boundary elements.

The latter improves vectorization efficiency and allows the alignment optimization by computing the 0th and nx-1th elements incorrectly as part of a vector operation, and then correcting the results with a scalar operation.

Note: the SBB implementation does not use the inline function. SBB templates take care of alignment and streaming stores, and handle the boundary elements with explicit halo regions.

5.1 Baseline

The baseline version is a modified version of the optimized code from [1]. An outline of the code is shown in Fig. 2. The outer loop is a timestep loop. Each iteration computes the stencil on the entire iteration space. There are two copies of the stencil array; one acting as an input array, and the other acting as an output array. The arrays are switched at the end of each timestep. A barrier is needed to ensure that all the threads have finished storing to the current output array before switching the two arrays.

```
#pragma omp parallel
for (int i = 0; i < count; ++i)
{
# define YBF 4
# define ZBF 4
# pragma omp for collapse(2) nowait
  for (int yy = 0; yy < ny; yy += YBF)
  for (int zz = 0; zz < nz; zz += ZBF) {
    int ymax = yy + YBF;
    int zmax = zz + YBF;
    for (int z = zz; z < zmax; z++) {
      for (int y = yy; y < ymax; y++) {
        diffusion_x_loop(f1_t, f2_t, nx, ny, nz, y, z,
          cc, cw, ce, cn, cs, ct, cb);
      }
    }
  }
  #pragma omp barrier
  REAL *t = f1_t;
  f1_t = f2_t;
  f2_t = t;
}
```

Fig. 2. Diffusion baseline

The Y and Z loops are tiled into 4×4 blocks. The 4×4 blocking factor was empirically determined using SBB and then back-ported to the various diffusion implementations. This loop nest is then collapsed and distributed amongst the threads. The inner loop is vectorized (see Fig. 9).

Note that in the baseline case, the load balance and data distribution are implicitly determined by the threads executing the collapsed for loop. We explore various other methods to improve load balance and data distribution in the implementations that follow.

```
#pragma omp parallel
{
  int z0, ze, y0, ye;
  // compute Y and Z begin and end points y0,ye,z0,ze
  ...
  for (int i = 0; i < count; ++i) {
    for (int yy = y0; yy < ye; yy += YBF)
    for (int zz = z0; zz < ze; zz += ZBF)
    {
      int z1 = zz + ZBF;
      int y1 = yy + YBF;
      for (int y = yy; y < y1; y++) {
        for (int z = zz; z < z1; z++) {
          diffusion_x_loop(f1_t, f2_t, nx, ny, nz, y, z,
            cc, cw, ce, cn, cs, ct, cb);
        }
    }

    #pragma omp barrier
    REAL *t = f1_t;
    f1_t = f2_t;
    f2_t = t;
  }
}       // parallel
```

Fig. 3. Diffusion 2d decomposition

5.2 Hand Decomposed

The hand decomposed version of diffusion is very similar to the baseline version, except that the blocks are distributed by hand. An outline of the code is in Fig. 3. We begin by obtaining the thread number *mythread*, and then compute our position in the Z and Y dimensions. Our position in the Y dimension is $mod(mythread, nHT)$ and our position in the Z dimension is $mythread/nHT$ (using integer division) where nHT is the number of threads per core: 2 for the Intel® Xeon® processor and 4 for the Intel® Xeon Phi™ coprocessor. Then we distribute the blocks in each dimension as evenly as possible. Finally, there are two nested outer loops (equivalent to the collapsed loops in the baseline code) that iterate over the Z and Y blocks, and two nested inner loops that iterate over the elements in each block.

One advantage of this hand decomposition is that a core always has all the Y blocks for each Z block. The collapsed loop might split some blocks between threads, since it distributes the full $Z * Y$ iteration space without regard to the original loop nesting.

One disadvantage of this decomposition is that the load balance is not always as good. For example, consider a problem size of 512 and a 60-core Intel® Xeon Phi® part. There are 512/4 or 128 blocks in each dimension. The number of threads in the Z dimension is 60, so the first 8 cores will each get 3 blocks of Z, and the last 52 will get 2 blocks of Z. Thus 52 of the cores will have 2/3 as much work to do as the other 8 cores.

In the collapsed case, there are 512*512/4 or 65536 iterations divided amongst 240 threads, resulting in 1092 or 1093 iterations per thread; the load imbalance is far less severe.

5.3 Nested Parallelism

As stated earlier, it is often useful to have the threads within a core cooperate on a single block of data residing in that core's caches. Nested OpenMP seems to be a good choice for this. Because current implementations of nested OpenMP have relatively high overhead, we placed the inner parallel region further out in the code than might have been desirable (this is explained more in the next section). This led to the code in Fig. 4.

```
#pragma omp parallel
{
  // compute decomposition in Z
  ...
  #pragma omp parallel for
  for (int yy = 0; yy < ny; yy += ybf)
  for (int zz = z0; zz < ze; zz += zbf)
  {
    int z1 = zz + zbf;
    if (z1 > nz) z1 = nz;
    int y1 = yy + ybf;
    if (y1 > ny) y1 = ny;
    for (int z = zz; z < z1; z++) {
      for (int y = yy; y < y1; y++) {
        diffusion_x_loop(f1_t, f2_t, nx, ny, nz, y, z,
          cc, cw, ce, cn, cs, ct, cb);
      }
    }
  }
}       // parallel
```

Fig. 4. Nested OpenMP

The parallel region enclosing the code in Fig. 4 computes its piece of the Z dimension just as described in the section on hand decomposition. Then the inner parallel for divides the blocks in the Y dimension so that each thread is working on 1/4 (or 1/2 for the processor) of the Y elements, but only on the Z elements for that core.

The cache footprint for each core for the Intel® Xeon Phi™ coprocessor is thus $NY * NZ/60 * NX * 4$ bytes, which is over 8 MB for the 512^3 problem and thus far too large for L2. It would probably be better to tile the Y loop again, or to attempt the solution shown in the next section.

```
for (int z = zz+ymythread; z < z1; z+=nHTs) {
  for (int y = yy; y < y1; y++) {
    diffusion_x_loop(f1_t, f2_t, nx, ny, nz, y, z,
      cc, cw, ce, cn, cs, ct, cb);
  }
}
```

Fig. 5. Hand nested inner loops

5.4 Hand Nested

In the hand nested case, we divide the Z dimension as before, but we use a trick to ensure that the hardware threads on a core cooperate on the same block of data. The outer loops are the same as those in Fig. 4, but the inner loops are different as shown in Fig. 5. Here `nHTs` is the number of threads per core and `ymythread` is the thread number within the core. We divide the work amongst the threads by assigning the Z iterations round-robin to the threads (note that each thread gets only one iteration on the coprocessor).

So that threads that finish early don't race ahead to the next block, we follow the code above with a core barrier. This barrier is not required for correctness. The core barrier uses two 4-byte words. Each thread sets its byte in the first word, then waits until all the bytes are set. It then does the same for the second word. The value set by a thread toggles between 0 and 1 every time the barrier is encountered, thus removing any need for re-initialization. Two words are needed in case of back-to-back barriers (an alternate formulation using only one word is available but somewhat more complicated). The core barrier is quite fast (under 200 clocks) since all the threads on the core share a cache. The core barrier is also described in [3].

The hand nested code is really the code we want to use with nested OpenMP, but the overhead of nested OpenMP is too high. Again looking at the 512^3 case, the computation performed by the tile loop in Fig. 4 is $4*4*512/16$ or 512 vector loop iterations (in other words, 16 calls to `diffusion_x_loop`). Using reference data we can compute the cost of a loop iteration as roughly between 5000 and 15000 clock cycles. Currently nested OpenMP overheads are greater than 500 clocks for fork-join of a nested parallel region, making nested OpenMP unusable at such fine granularity.

5.5 Crew and Teams

Finally we experimented with two different lower-overhead implementations. Crew is a very lightweight experimental nested threading model for OpenMP that exists in the Intel® C++ Composer XE compiler. Crew creates one OpenMP thread per core, and one extra thread for each additional hyperthread (3 extra threads for the coprocessor). The notation `#pragma intel_crew parallel for` then causes the main thread and the additional threads to divide the work for the following loop; however, no nested OpenMP regions are created and a dynamic scheduling policy is used. The lack of nested OpenMP overhead greatly reduces the overhead for nested parallelism.

Teams are designed for device code, but using `#pragma device if(0)` causes the code to be executed on the host. Creation of teams involves less overhead than nested OpenMP. Unfortunately, there is a need for a barrier of all of the threads in the team at each timestep, and the current definition of the teams construct does not allow a barrier. Thus the answers from the teams implementation are incorrect.

In both the crew case and the teams case, we parallelized the same loop as in the nested OpenMP case (rather than our preferred loop as was done in the hand nested case). Otherwise the code looks the same as the OpenMP nested code.

We ran the crew experiments only on the Intel® Xeon Phi™ coprocessor, and the teams experiments only on the Intel® Xeon® processor. The results are included for completeness.

5.6 SBB

The SBB version in Fig. 6 uses template programming to generate multiple versions of the same diffusion kernel (`diffusionOdd`) by varying Containers (AOS, SOA, Tiled), Accesors (YBF, ZBF), and Engines (OpenMP, TBB). Here, we present a set of data generated using `Soa3dContainer` and `OpenMP` engine and YBF = ZBF = 4. Essentially, this SBB version is the same as the baseline version. In fact, the optimal block sizes of the baseline code are "auto-tuned" based on the extensive SBB data. C++ 11 features (auto, lambda functions) and predfined `SBB_` macros faciliate compact and efficient codes that can be easily incorporate into the existing C++ applications.

6 Performance Experiments

We used two different systems for our experiments. The Intel® Xeon® system is a dual socket E5-2697 v3 (formerly code-named Haswell) @ 2.60 GHz. Each socket has 14 cores with two hardware cores per thread. The Intel® Xeon Phi™ system is a B1PRQ-7110 P/X @ 1.10 GHz. We used 60 of the 61 cores, each with four hardware threads.

Tables 1 and 2 contains the raw data (in GFlops/Second) for the two systems. Figures 7 and 8 show the performance in charts. The data in the charts is

```
//containers: SOA for 3D with +-1 Halo regions
const int StencilHaloSize = 1;
using Container=sbb::Soa3dContainer<float, StencilHaloSize, Allocator>;

Container inputContainer(nx,ny,nz);
Container outputContainer(nx,ny,nz);

//iterator space
sbb::Block3dBounds iterationSpace;
iterationSpace.d1.set(StencilHaloSize,nx-StencilHaloSize);// for d2 & d3

//block size
sbb::Block3dSize blockSize;
blockSize.d1=nx; blockSize.d2=YBF; blockSize.d3=ZBF;

auto in = inputContainer.access();
auto out = outputContainer.access();

//Define 2 kernels
//"Odd" that reads from "in" and writes to "out"
SBB_KERNEL_BEGIN(diffusionOdd)
  SBB_NON_TEMPORAL_BEGIN
  SBB_ITER_D321_BEGIN(z, y, x)
  {
    float result = cc* in[z][y][x] + cw* in[z][y][x-1] + ...;
    out[z][y][x] = result;
  }
  SBB_ITER_END
  SBB_NON_TEMPORAL_END
SBB_KERNEL_END
// "Even" that reads from "out" and writes to "in"
...

sbb::OpenMp3dEngine<sbb::VectorCode> the3dEngine;
for(int i=0; i<count; ++i)
{
  the3dEngine.run(diffusionOdd, iterationSpace, blockSize);
  the3dEngine.run(diffusionEven, iterationSpace, blockSize);
}
```

Fig. 6. SBB diffusion code

normalized to the memory bandwidth (Stream Triad) for the platform: 108 GB/s and 158 GB/s, respectively.

On the Intel® Xeon® coprocessor, the hand-nested code is clearly the best except in the 256^3 case, where it is still competitive. One surprise is how well the nested version performs.

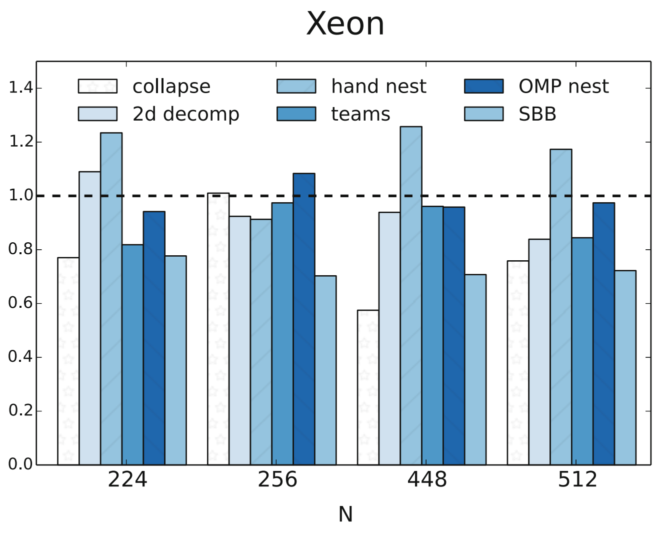

Fig. 7. Intel® Xeon® processor normalized performance

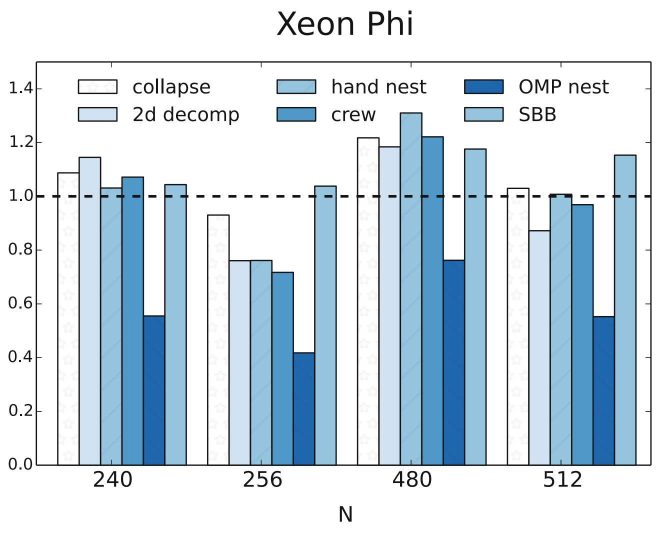

Fig. 8. Intel® Xeon Phi™ coprocessor normalized performance

```
static inline void
diffusion_x_loop(const REAL *f1_t, REAL *f2_t,
    int nx, int ny, int nz,
    int y, int z,
    REAL cc, REAL cw, REAL ce, REAL cn, REAL cs, REAL ct, REAL cb)
{
    int x;
    int c, w, e, n, s, b, t;
    const REAL *restrict pc; ...
    x = 0;
    c = x + y * (nx + NXP_DELTA) + z * (nx + NXP_DELTA) * ny;
    w = c - 1;
    e = c + 1;
    n = (y == 0) ? c : c - (nx + NXP_DELTA);
    s = (y == ny - 1) ? c : c + (nx + NXP_DELTA);
    b = (z == 0) ? c : c - (nx + NXP_DELTA) * ny;
    t = (z == nz - 1) ? c : c + (nx + NXP_DELTA) * ny;
    pc = &f1_t[c]; ...
    poc = &f2_t[c];
    __assume_aligned(pc, CACHE_LINE_SIZE);
    ...
    #pragma simd
    for (x = 0; x < N_REALS_PER_CACHE_LINE; ++x)
      poc[x] = cc * pc[x] +
        cw * pw[x] + ce * pe[x] +
        cs * ps[x] + cn * pn[x] + ct * pt[x] + cb * pb[x];
    // element 0
    poc[0] = cc * pc[0] +
      cw * pc[0] + ce * pe[0] +
      cs * ps[0] + cn * pn[0] + ct * pt[0] + cb * pb[0];
# pragma vector nontemporal
# pragma simd
    for (x = N_REALS_PER_CACHE_LINE; x < nx; x++)
    {
      poc[x] = cc * pc[x] +
        cw * pw[x] + ce * pe[x] +
        cs * ps[x] + cn * pn[x] + ct * pt[x] + cb * pb[x];
    }
    // element nx-1
    poc[nx-1] = cc * pc[nx-1] +
      cw * pw[nx-1] + ce * pc[nx-1] +
      cs * ps[nx-1] + cn * pn[nx-1] +
      ct * pt[nx-1] + cb * pb[nx-1];
}
```

Fig. 9. Diffusion inner loop

Table 1. Intel® Xeon® processor (GF/Sec)

	224	256	448	512
Collapse	83.2	109.1	62.1	81.9
2d decomp	117.7	99.8	101.4	90.6
Hand nest	133.3	98.6	135.8	126.7
Teams	88.4	105.2	103.8	91.2
OMP nest	101.7	117.0	103.5	105.2
Sbb	83.9	75.9	76.4	78.0

Table 2. Intel® Xeon Phi™ coprocessor (GF/Sec)

	240	256	480	512
Collapse	171.8	147.0	192.4	162.7
2d decomp	180.9	120.2	187.1	137.8
Hand nest	162.9	120.3	207.0	159.2
Crew	169.3	113.3	193.0	153.1
OMP nest	87.7	66.0	120.4	87.3
Sbb	165.0	163.5	185.0	180.0

On the Intel® Xeon Phi™ coprocessor, the hand nested version is still most competitive. One pleasant surprise is how well the original blocked and collapsed version performs; this is nice because it is relatively easy to write compared to the others. This data also demonstrates the strength of SBB. For users that are comfortable with C++ template libraries, SBB is an excellent choice that also enables easy experimentation with different containers and blocking factors.

7 Conclusions and Future Work

Nested OpenMP is a natural way to exploit hardware with two-level threading, but the overhead is currently prohibitive for very fine grained threading, especially on the current generation of the Intel® Xeon Phi™ coprocessor. Alternatives such as hand nesting are possible, but can be tricky to write and to maintain, and may not be portable (especially with respect to performance).

Some of the performance issues with nested OpenMP are inherent in the OpenMP specification (thread teams, various query functions, ICVs, etc.) and it would be beneficial to consider changes or additions to the specification to make nested parallelism more lightweight. Other performance issues are related to quality of implementation. Hopefully this paper gives more impetus to the developers to improve their implementations.

Some of the OpenMP requirements can be relaxed when using OpenMP 4.0 teams, but currently those have restrictions (most importantly, that they need to be used in a device region) that make them unsuitable for general use. Making teams more general is a possible alternative to loosening the requirements for nested OpenMP.

One problem with picking diffusion for this paper is that there isn't enough work in the stencil. Real codes (e.g., various oil and gas codes) have far more complex stencils. Our experience has shown that in these cases, nested threading and blocking have much greater impact.

With the exception of SBB, none of these codes address load balancing, which is significant when the problem size doesn't match the number of cores and threads available. This is clearly evident in the 256^3 and 512^3 problems run on the Intel$^{®}$ Xeon Phi^{TM} coprocessor, where SBB is the best performer. More work is needed to create a pure OpenMP dynamically scheduled code that still has good cache locality. Attempts using OpenMP tasking did not achieve good performance. We attribute this to high overhead when creating OpenMP tasks.

There are a few anomalies in the data: core threading on the Intel$^{®}$ Xeon$^{®}$ processor is poor for the 256^3 problem; nested parallelism on the Intel$^{®}$ Xeon Phi^{TM} coprocessor is far worse than on the processor; SBB performance is worse on the processor than on the coprocessor; tiled/collapsed code performs reasonably well on the coprocessor but not as well on the processor. We are investigating these issues.

SBB is a useful alternative for C++ programmers and is reasonably easy to use compared to some of the more complicated techniques.

References

1. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High-Performance Programming. Morgan Kauffman, Boston (2013)
2. Dempsey, J.: High performance parallelism perls. In: Jeffers, J., Reinders, J. (eds.) Pesiochronous Phasing Barriers, pp. 87–115. Morgan Kauffman, Boston (2015)
3. Briggs, J., et al.: Separable projection integrals for higher-order correlators of the cosmic microwave sky: acceleration by factors exceeding 100. Cornell University Library. http://arxiv.org/abs/1503.08809