Workstealing and Nested Parallelism in SMP Systems

Larry Meadows(✉), Simon J. Pennycook, Alex Duran, Terry Wilmarth, and Jim Cownie

Intel Corporation, Hillsboro, OR, USA
{lawrence.f.meadows,john.pennycook,alejandro.duran,
terry.l.wilmarth,james.h.cownie}@intel.com

Abstract. We present a workstealing scheduler and show its use in two separate areas: (1) to enable hierarchical parallelism and per-core load balancing in stencil codes, and (2) to reduce overhead in per-thread load balancing in particle codes.

Keywords: Stencil · Nested parallelism · Runtime support

1 Introduction

Modern symmetric multiprocessors (SMPs) have cores with multiple hardware threads per core and share caches at multiple levels. Effective programming for such systems requires that code be structured so that threads and cores cooperate rather than compete for these shared resources.

Section 2 introduces some terminology. Section 3 motivates the need for the workstealing scheduler using a simple two-dimensional loop and discusses its implementation. In Sects. 4 and 5 we introduce the ISO-3DFD stencil code, show several implementations exploiting hierarchical parallelism and the workstealing scheduler, and give performance results. Section 6 describes possible extensions to OpenMP* that are motivated by the stencil implementations. Section 7 shows how the workstealing scheduler can be used in particle codes. Finally we conclude with Sect. 8.

2 Terminology

In our terminology, a core is a single hardware processor. Each core can execute multiple independent hardware threads ("hyperthreads"), which are interleaved in the core's pipeline. Threads on a core share all of the core's resources, including all levels of cache. We assume that OpenMP threads are tightly bound to cores so that the operating system cannot move them, and that there is no oversubscription. Therefore if we have a core that can execute four threads (e.g., on an Intel® Xeon Phi™ processor or coprocessor), the OpenMP runtime will create at most four threads bound to that core.

N. Maruyama et al. (Eds.): IWOMP 2016, LNCS 9903, pp. 47–60, 2016.
DOI: 10.1007/978-3-319-45550-1_4

3 Static Workstealing Scheduler

3.1 Motivation

Consider a code that loops over a two-dimensional iteration space:

```
for (int i = 0; i < N; ++i)
  for (int j = 0; j < M; ++j)
    work(i, j);
```

If the iterations of the nested loop are independent, then we can easily parallelize the loop nest and exploit parallelism in both loops using the OpenMP `collapse` directive:

```
#pragma omp parallel for schedule(static) collapse(2)
for (int i = 0; i < N; ++i)
  for (int j = 0; j < M; ++j)
    work(i, j);
```

The combined iteration space of length $N * M$ is divided among the OpenMP threads according to the schedule clause. If required, the compiler generates code to recover the `i` and `j` indices for each iteration using % and / operators.

In many cases we would prefer to distribute the iterations by core, rather than by thread. This can improve locality and load balancing. It also enables cooperation among the threads on a single core, as we will see in a later section. Since OpenMP does not provide such an iteration distribution, we simulate it by hand like this:

```
#include"omp_sched.h"
Percore cores[maxCores];
#pragma omp parallel
{
  Sched sch(N*M, cores);
  int block;
  while ((block = sch.nextiter()) != -1) {
    int i = block / N;
    int j = block % N;
    coreWork(i, j);
  }
}
```

Here the function `coreWork` will be called with the same i and j values in all of the threads on a given core. It must therefore internally distribute the work for the i, j iteration over those threads.

The scheduler `Sched` is described in the next section.

3.2 Scheduler Implementation

The static workstealing scheduler is implemented as two C++ classes: `Sched` and `Percore`. `Percore` contains data that is shared among the OpenMP threads that execute on the same core. The scheduler uses an array of `Percore` objects, one for each core. Since the `Percore` array is shared it must be declared outside an OpenMP parallel region or allocated on the heap. A predefined constant `maxCores` is defined to simplify declaration:

```
Percore cores[maxCores];
```

The `Sched` class must be instantiated inside a parallel region, resulting in one instance per OpenMP thread. It is constructed with the number of iterations of the parallel loop and the `Percore` array:

```
Sched sch(niter, cores);
```

Every cooperating instance of `Sched` must be constructed with the same arguments.

`Sched` has two member functions: `nextiter()` and `nextiter1()`. Both of these function enumerate the iterations for which the `Sched` instance was constructed. `nextiter()` returns the same value to all hyperthreads on the same core and contains an implicit barrier on them. `nextiter1()` returns a value to only the calling thread and there is no implicit barrier. There is a third member function `corebarrier()` that barriers the threads in a core. It uses the barrier described in [5,6].

Initially iterations are distributed as they would be in a static schedule, except that they are distributed over cores rather than threads. The `Percore` object for a given core contains the start and end iteration for that core. When one of the iteration functions is called, it first checks to see if there are any remaining iterations on the core to which the calling thread is bound. If so, it atomically increments the start value and returns that iteration. Otherwise, it traverses the `Percore` array looking at each core. As soon as it finds a core that has available iterations, it atomically decrements the end value (thus stealing an iteration from the end of its range) and returns that iteration. If a thread has traversed all cores and found no iterations to steal, the iteration function returns −1, indicating that there are no more iterations to start. See Fig. 1 for the stealing algorithm.

```
iter_t _nextiter()
{
    iter_t ret = getiter(*core);        // my core
    if (ret != -1)
        return ret;

    // need to steal, start in a random place
    int startcore = _rdtsc() % ncores;
    int i = startcore;
    do
    {
        if (i != mycore)
        {
            ret = stealiter(base[i]);
            if (ret != -1)
                return ret;
        }
        i = (i == ncores-1) ? 0 : i+1; // wraparound
    } while (i != startcore);
    return -1;
}
```

Fig. 1. Stealing algorithm

3.3 OpenMP Scheduler Constraints

Prior to OpenMP 4.5, the description of a dynamic schedule in OpenMP was subject to interpretation. In particular it was not immediately apparent whether code liks this should always succeed, or whether schedules which would cause it to abort are legal.

```
#pragma omp parallel
{
    int myHighestIteration = -1;
#pragma omp for schedule(dynamic)
    for (int i=0; i<1000; i++)
    {
        if (i < myHighestIteration)
            abort();
        else
            myHighestIteration = i;
    }
}
```

Fig. 2. OpenMP scheduler monotonicity test

This is an important question for the static workstealing scheduler, since under load-imbalance it will generate schedules that would cause this test to abort. (Consider the thread whose static schedule includes the serially final iteration; if it ever steals from another thread the iteration it steals must be lower than one it has already seen.)

In OpenMP 4.5 control of this property of the scheduler (known as "monotonicity") can be explicitly expressed by the programmer using the new schedule modifiers (`monotonic` and `nonmonotonic`). In addition, OpenMP 4.5 includes the statement of intent that in OpenMP 5.0 an unmodifed dynamic loop schedule can legally be treated as though it had the nonmonotonic qualifier. These improvements to the OpenMP standard make it clear that a static workstealing scheduler, like that described here, can be used inside the OpenMP runtime, and, therefore, that its performance should be of interest to people who will never rewrite their code to include their own scheduler (Fig. 2).

4 ISO-3DFD Test Code

The ISO-3DFD stencil code (hereafter referred to as *ISO-3DFD*) is a 16th order stencil in space (second order in time) finite difference implementation used to solve the isotropic acoustic wave equation. The baseline code is shown in Fig. 3.

We can use a roofline model [2] to estimate the performance of ISO-3DFD. Each iteration of the loop as written above has 27 multiplies, 51 adds, and 51 4-byte loads, for an arithmetic intensity of 78/4*51 or 0.38 flops/byte. We use five different systems in this article:

SNB Intel® Xeon® Processor E5-2680 v1

```
int dimn1n2 = n1*n2;
for(int iz=0; iz<n3; iz++)
  for(int iy=0; iy<n2; iy++)
    for(int ix=0; ix<n1; ix++) {
      float value = ptr_prev[offset]*coeff[0];
      for(int ir=1; ir<=8; ir++) {
        value += coeff[ir] * (ptr_prev[offset + ir] +
                              ptr_prev[offset - ir]);
        value += coeff[ir] * (ptr_prev[offset + ir*n1] +
                              ptr_prev[offset - ir*n1]);
        value += coeff[ir] * (ptr_prev[offset + ir*dimn1n2] +
                              ptr_prev[offset - ir*dimn1n2]);
      }
      ptr_next[offset] = 2.0f* ptr_prev[offset] - ptr_next[offset] +
                         value*ptr_vel[offset];
    }
```

Fig. 3. ISO-3DFD pseudocode

HSW Intel® Xeon® Processor E5-2697 v3
BDW Intel® Xeon® Processor E5-2699 v4
KNC Intel® Xeon Phi™ C0PRQ-7120
KNL the Intel processor codenamed Knights Landing in a preproduction system, B0 stepping, 1.4 GHz, 68 cores, 16 GB MCDRAM, 96 GB DDR in quadrant/flat mode.

We collectively call SNB, HSW and BDW big cores, and KNC and KNL small cores.

Table 1 shows the five systems, their Stream benchmark [7] figures, the projected performance from the roofline model, and the actual, measured, performance. Note: Only one socket is used on the big core systems to avoid complications from NUMA effects.

From Table 1 we see that the actual performance exceeds that predicted by the roofline model. This is because, by using bandwidth from main memory, the roofline model is implicitly assuming that all the loads miss cache. In fact, as we can see from the stencil pattern, there is a lot of potential temporal and spatial reuse from one iteration to the next in all three dimensions. In particular the X dimension has a lot of spatial reuse because of accesses to the same few

Table 1. Roofline prediction vs. measured performance

System	Stream bandwidth (GB/s)	Roofline prediction (GF/s)	Measured performance (GF/s)
SNB	39	15	25
HSW	59	22	44
BDW	52[a]	20	39
KNC	177	67	137
KNL	490	186	275

[a] The memory configuration on this system is non-optimal, with optimal configuration the stream bandwidth is around 70 GB/s per socket.

Table 2. Roofline using LLC BW

System	LLC bytes/ clock/core	# Cores used	Freq (MHz)	LLC BW (GB/Sec)	Roofline prediction (GF/Sec)	Measured performance (GF/Sec)
SNB	11	8	2700	237	90	25
HSW	11	14	2600	400	152	44
BDW	11	22	2200	532	173	39
KNC	14	60	1238	1040	395	137
KNL	32	64	1400	2867	1089	275

cache lines offset by between 1 and 8 4-byte floats in both positive and negative directions.

We can refine the roofline model to get a more accurate performance estimate by looking at bandwidth from the last level cache. On the small cores the last level cache is the L2 cache, while on the big cores it is the L3 cache. Table 2 shows the roofline performance using this alternate bandwidth metric (LLC bandwidth was measured by a simple test program performing a vectorized single-precision summation). This roofline assumes that all accesses hit in the last-level cache. Tables 1 and 2 bound the expected performance of ISO-3DFD.

5 ISO-3DFD Optimization

As we have seen, there is a lot of temporal reuse in ISO-3DFD and we would like to capture that reuse to improve performance. The usual way to do this is to tile the loops so that reused cache-lines are closer together in time and thus more likely to stay in the LLC. This is fairly easy in ISO-3DFD, as shown in Fig. 4. The tile sizes `tilex`, `tiley`, and `tilez` are chosen by experimentation. Tile sizes should be fairly small so that the collapsed loop has significantly more iterations than threads to improve load balancing (dynamic scheduling or static workstealing can also be used as we will see later).

5.1 Nested Parallelism vs. Hand Threading

The collapsed loop in Fig. 4 distributes iterations by thread, not by core. To obtain iteration distribution by core, we can use either the static workstealing scheduler described previously, or nested parallelism. The two methods are shown side-by-side in Figs. 5 and 6.

Both implementations distribute the tiles among the cores and then distribute the work for the tile over the threads in the core. The code in Fig. 5 uses nested OpenMP. It assumes that the threads in the outer team are placed one per core, and that the threads in the inner teams are on the same core as their master thread. The code in Fig. 6 accomplishes the same effect by precomputing

```
#pragma omp parallel for collapse(3)
for(int iiz=0; iiz<n3; iiz+=tilez)
for(int iiy=0; iiy<n2; iiy+=tiley)
for(int iix=0; iix<n1; iix+=tilex) {
  int zmax = std::min(iiz+tilez, n3);
  int ymax = std::min(iiy+tiley, n2);
  int xmax = std::min(iix+tilex, n1);
  for (int iz=iiz; iz<zmax; ++iz)
  for (int iy=iiy; iy<ymax; ++iy) {
    #pragma omp simd
    for (int ix=iix; ix<xmax; ++ix) {
      int offset = idx(ix, iy, iz, n1, n2, n3);
      float value = ptr_prev[offset]*coeff[0];
      #pragma unroll(8)
      for(int ir=1; ir<=8; ir++) {
        value += coeff[ir] * (ptr_prev[offset + ir] +
                              ptr_prev[offset - ir]);
        value += coeff[ir] * (ptr_prev[offset + ir*xn1] +
                              ptr_prev[offset - ir*xn1]);
        value += coeff[ir] * (ptr_prev[offset + ir*xn1n2] +
                              ptr_prev[offset - ir*xn1n2]);
      }
      ptr_next[offset] = 2.0f*ptr_prev[offset] - ptr_next[offset] +
                         value*ptr_vel[offset];
    }
  }
}
```

Fig. 4. ISO-3DFD tiled pseudocode

the core and hyperthread for each OpenMP thread. Hand threading assumes that OpenMP thread affinity is set properly to mirror the internal mapping to cores and hyperthreads. The distribution of the work within the core is done explicitly in the Y loop using the thread number within the core.

```
#pragma omp parallel for collapse(3)
for(int iiz=0; iiz<n3; iiz+=tilez)
for(int iiy=0; iiy<n2; iiy+=tiley)
for(int iix=0; iix<n1; iix+=tilex) {
#pragma omp parallel
  {

  int myht = omp_get_thread_num();









  int zmax = std::min(iiz+tilez, n3);
  int ymax = std::min(iiy+tiley, n2);
  int xmax = std::min(iix+tilex, n1);
  for (int iz=iiz; iz<zmax; ++iz)
  for (int iy=iiy+myht; iy<ymax; iy+=nHT) {
    #pragma omp simd
    for (int ix=iix; ix<xmax; ++ix) {
      ...
    }
  }
  }
}
```

Fig. 5. Nested OpenMP

```
Percore cores[maxCores];
#pragma omp parallel
{
int nblocksz = (n3 + tilez - 1) / tilez;
int nblocksy = (n2 + tiley - 1) / tiley;
int nblocksx = (n1 + tilex - 1) / tilex;
nblocks = nblocksz * nblocksy * nblocksx;
Sched sch(nblocks, cores);
int myht = sch.myht;
int block;
while ((block = sch.nextiter()) != -1) {
  int iiz = block / (nblocksy * nblocksx);
  int rem = block % (nblocksy * nblocksx);
  int iiy = rem   / (nblocksx);
  int iix = rem   % (nblocksx);
  iix *= tilex; iiy *= tiley; iiz *= tilez;
  int zmax = std::min(iiz+tilez, n3);
  int ymax = std::min(iiy+tiley, n2);
  int xmax = std::min(iix+tilex, n1);
  for (int iz=iiz; iz<zmax; ++iz)
  for (int iy=iiy+myht; iy<ymax; iy+=nHT) {
    #pragma omp simd
    for (int ix=iix; ix<xmax; ++ix) {
      ...
    }
  }
}
}
```

Fig. 6. Hand threaded

5.2 Performance Results

Table 3 compares the different implementions of ISO-3DFD. The implementations are:

Baseline The baseline implementation in Fig. 3.
Tiled The tiled implementation in Fig. 4.
Nested The nested implementation in Fig. 5.
Hand The hand-threaded implementation in Fig. 6.
Scheduler Three implementations used to evalua te the workstealing scheduler. The code is hand-tiled and the loop over the tiles is parallelized in one of three ways:
Static - Use OpenMP `#pragma omp for schedule(static)`
Workstealing - Use static workstealing scheduler
Dynamic - Use OpenMP `#pragma omp for schedule(dynamic)`

Table 4 gives the performance results for the different implementations on the test platforms.

On big cores, tiling alone gives a significant improvement over the baseline. This is because the large L3 cache is able to hold almost all of the reused data. Small cores do not have an L3 cache and the smaller L2 cache is not able to enable much reuse, so tiling does not generally help here.

Turning to the scheduler columns, we see that OpenMP dynamic scheduling is generally superior to either OpenMP static scheduling or static workstealing on big cores (though this might change if we looked at a multi-socket big-core case). The shared L3 improves the performance of the atomic operations needed in dynamic scheduling, and the static workstealing implementation introduces additional overhead. However on the small cores, Static workstealing is clearly superior to either OpenMP static or OpenMP dynamic scheduling since it requires many fewer atomic operations.

This also helps to explain why nested parallelism (which uses `#pragma omp parallel for schedule(dynamic)`) performs better than hand threading (which uses static workstealing) on the big cores.

On the small cores we were able to improve the performance of nested parallelism by adjusting the tile size. The small cores have reduced single-thread performance and many more cores which combine to increase the overhead of

Table 3. ISO-3DFD implementation comparison

	Baseline	Tiled	Nested	Hand	Static	Worksteal	Dynamic
Outer loop structure	collapse(2) Z,Y	collapse(3) Z,Y,X	hand-collapse Z,Y,X	hand-collapse Z,Y,X	hand-collapse Z,Y,X	hand-collapse Z,Y,X	hand-collapse Z,Y,X
Inner loop structure	simd X	serial Z,Y simd X	serial Z,Y simd X	serial Z,Y simd X	serial Z,Y simd X	serial Z,Y simd X	serial Z,Y simd X
Cooperative threading	none	none	Y loop	Y loop	none	none	none

Table 4. ISO-3DFD performance (GF/s)

System	Threads	Indep threading		Coop threading		Scheduler		
		Baseline	Tiled	Nested	Hand	Static	Worksteal	Dynamic
SNB	1	25	55	59	57	61	57	55
	2	21	47	59	57	49	56	52
HSW	1	39	109	109	106	113	113	118
	2	35	107	119	108	96	107	109
BDW	1	39	155	156	119	117	121	154
	2	45	158	167	122	108	120	156
KNC	1	49	40	38	51	48	52	40
	2	76	70	79	82	67	89	70
	4	93	86	101	107	64	115	86
KNL	1	256	119	159	264	250	274	122
	2	179	132	148	291	245	318	135
	4	166	169	209	274	196	196	173

nesting, so the amount of work per tile needs to be larger to compensate. However, even with this change, the hand threaded implementation with static workstealing outperforms nested parallelism.

6 OpenMP Extension to Loop Scheduling

As we have seen, the current OpenMP support is limited when trying to map the iterations of loops to modern hardware and forces programmers wanting the maximum performance to code their own loop scheduling policies by hand. To improve OpenMP support we propose two sets of extensions to the loop scheduling mechanisms: hierarchical loop scheduling and multi-dimensional chunking.

6.1 Hierarchical Loop Scheduling

In existing and future architectures not all hardware threads are peers. Hardware threads are organized in a hierarchy in which threads in the same level share some resources together. For example, in many architectures several hardware threads are part of the same core (sharing some part of the cache hierarchy) and these cores are part of the same NUMA domain (sharing a privileged access to some system memory). These logical groupings create a hierarchy of groups of threads.

Exploiting this hierarchy when distributing the iterations of a given loop nest is becoming increasingly important to obtain the best performance. Current OpenMP provides the mechanisms (e.g., `OMP_PLACES` and `OMP_PROC_BIND`) that allow the programmer to lay out the different OpenMP threads across the

hardware thread hierarchy. But it has no provisions to ensure that the scheduling of iterations from a loop can exploit these carefully planned layouts.

Furthermore, the scheduling decisions at each level of the hardware hierarchy are likely to be different. For example, a programmer might want to dynamically distribute relatively large groups of iterations between the different cores but then statically distribute the iterations of each group between the threads of each core, or to statically distribute large groups of iterations across NUMA domains while dynamically scheduling those iterations inside a NUMA domain.

While it is possible to code these patterns today in OpenMP using nested parallelism the unnecessary fork-join overheads make it impractical in many cases. We argue that these patterns should be supported as loop scheduling options that do not require nested parallelism. We therefore propose two extensions to the existing OpenMP loop construct:

- add a new `schedule-groupsizes` clause. This clause contains a list of positive integer expressions that are group sizes. The first group size defines how the threads of the team are divided into groups. Each subsequent group size specified defines how the previous group is divided. This creates a hierarchy of groups that will be used for scheduling the iterations of the loop. The usage of this clause must be coordinated with the thread affinity mechanism to obtain good results.
- extend the current syntax of the schedule clause from a single `schedule-kind` to a list of `schedule-kind`s. The `schedule-kind` that will be applied to each level of the group hierarchy is implicitly defined by the order in the list (i.e., the first schedule kind applies to the root group, the second schedule kind applies to the next group level, etc.).

6.2 Multi-dimensional Chunking

Another common OpenMP limitation is that in a nest of loops OpenMP only offers the options to either schedule the iterations of the outer loop of the nest or collapse the iteration space of some outer subset of the nest and schedule the resulting iterations of the collapse operation. In the first case, the loop scheduling is akin to a one-dimensional tiling of the loop nest, while in the second case the created tiles can have irregular shapes that do not favor locality.

What a programmer would frequently like is to distribute multi-dimensional tiles between the threads that cooperate in the worksharing construct. Today programmers are required to manually modify their code to apply tiling optimization and then apply a loop worksharing with the `collapse` clause as shown in Fig. 4.

Because by creating chunks of iterations for loop scheduling purposes OpenMP is implicitly supporting 1D tiling, we propose to extend loop scheduling semantics to support multi-dimensional tiling. This requires that the chunks of iterations assigned to threads contain not just a subset of iterations of the outer most loop, but a tuple of iteration subsets for other loops in the nest.

To express this we propose to extend the existing `schedule-kind`s to accept not just one chunksize expression but a list of them. The first chunksize is applied to the first loop in the nest, the second chunksize to the next loop in the next, etc. The special value $*$ is allowed in one dimension to imply that chunksize is equal to the number of iterations in that dimension. This is important in combination with the previous hierarchical scheduling proposal as the number of iterations below the root level might be unknown.

Given a certain nest of m loops and a `schedule` clause with chunk sizes $C_1, ..., C_m$ the iteration space of the loop nest is $N_1 x..xN_m$ where N_i is the number of iterations for the i loop in the nest (1 being the outermost loop and m the innermost). This iteration space is partitioned in chunks of size $C_1 x...xC_m$ except for the uppermost boundaries where they can have less iterations. Then these chunks are distributed to the threads in the team following the schedule-kind specified in the schedule clause (i.e., statically or dynamically).

A new static (without chunksize) schedule-kind that has the same semantics as the existing `static` schedule-kind but can be applied to multiple loops (i.e., create tiles of approximately the same size for each thread) could also be added to OpenMP. It is unclear to us how useful this would be, as in practice a programmer usually wants to create these tiles to match the size of a specific hardware resource (e.g., the L2 cache).

6.3 Example

Combining the two proposals we can apply them together to the ISO3DFD code as show in Fig. 7. We use the multi-dimensional chunking to create 3D tiles of size $tilex*tiley*tilez$. Then we use the hierarchical scheduling to distribute them dynamically across groups of HT threads (which correspond to cores assuming a close thread placement and that HT is the number of threads per core used). Then threads inside each core cooperate to execute the iterations of the tile. Note that the `static(*,*,1)` expresses that only those iterations of the ix loop (which are also SIMDized) are distributed among threads of each core using a static schedule with chunksize 1. The code generated from this new version should be equivalent to that of Fig. 6, but is significantly easier to write!

```
#pragma omp parallel for schedule-groupsizes(HT) \
                         schedule(dynamic(tilez,tiley,tilex),static(*,*,1))
for ( iz = 0; iz < zmax; iz++)
    for ( iy = 0; iy < ymax; iy++)
       #pragma omp simd
        for ( ix = 0; ix < xmax; ix++ )
        {
           ...
        }
```

Fig. 7. ISO-3DFD parallelization with the proposed OpenMP extensions

7 Static Workstealing and Particle Codes

Particle codes often include some interaction between the particles and a mesh (i.e., a discretized version of the space in which they are travelling). For example, the Particle Mesh Ewald (PME) method accumulates charges at mesh points in order to approximate long-range interactions, and in particle transport codes each particle will update several "tally" values for any cells that it encounters while traversing the mesh.

The simplest way to implement these particle-mesh interactions in parallel is to loop over particles, using some method of guaranteeing atomicity (e.g., hardware atomics, locks or transactions) to handle write-conflicts during updates to a single (shared) mesh data structure. Although in some cases it is possible to use some algorithmic restructuring (e.g., using coloring [8]) instead, such approaches are more complicated to implement and may incur other runtime overheads or decrease the amount of available parallelism. We restrict our discussion to the simplest implementation.

In order to improve cache locality, it is desirable to have some way to group and then iterate over particles spatially, and this is commonly achieved by sorting all of the particles at some fixed frequency [9]. However, the number of particles in each region of space is not guaranteed to be uniform, and in some simulations the amount of computation per particle is not fixed (e.g. particles may represent different atoms, or different regions of space may have different material properties). As a result it is often necessary to use dynamic scheduling of some sort to overcome the load imbalance.

7.1 Application of Static Workstealing

A purely static schedule ensures that threads are working on particles from disjoint sections of the mesh (thus reducing the probability of write-conflicts), but does not account for load imbalance; at the other extreme, a purely dynamic schedule handles load imbalance well, but makes no guarantees about scheduling (potentially increasing the probability of write-conflicts). The static workstealing schedule proposed here strikes a good balance between the two: each thread is initially assigned work from one region of the mesh, but is able to steal work from another region if/when necessary.

Distributing work per core instead of per thread provides additional benefits, by ensuring that the threads executing on each core are (initially) working on particles from the same region. The primary effect of this is decreased latency for mesh data accesses through cache re-use. A secondary effect is improved atomics throughput, since an atomic update to a cache line is fastest when the line is already held in modified/exclusive state by the updating core [10].

Table 5 compares the performance of static, dynamic, guided and static workstealing schedules for a Monte Carlo particle transport benchmark developed by the French Alternative Energies and Atomic Energy Commission (CEA) [11].

Note that all of these schedules had to be implemented by hand (i.e., without using OpenMP pragmas) due to the structure of the loops involved; however,

Table 5. Monte Carlo particle transport benchmark performance (Mega events/second)

System	Threads	Scheduler			
		Static	Dynamic	Guided	Worksteal
SNB	2	84	108	87	113
HSW	2	112	163	125	185
BDW	2	211	281	244	327
KNC	4	191	315	225	378
KNL	2	278	413	335	588

every effort was made to ensure that the implementation was representative. Static workstealing provides a clear performance benefit across all of the architectures tested, and should be expected to provide greater benefits where inter-core communication is more expensive (e.g., in dual-socket systems).

8 Conclusions and Future Work

We have shown that static workstealing performs well across the board on the small core platforms and on imbalanced problems on big cores. With the introduction of the `nonmonotonic` modifier for dynamic schedules in OpenMP 4.5 (and the statement that `nonmonotonic` will become the default dynamic schedule in OpenMP 5.0), the static workstealing implementation is now legal in OpenMP implementations. Given the performance it shows on our examples, we expect that it will become the default implementation for dynamic schedules in many runtimes.

We have also proposed simple extensions to OpenMP which would allow the expression of loop tiling and the choice of appropriate schedules at each level of a closely nested OpenMP loop-nest. These extensions would allow the benefits which we have demonstrated from these techniques to be more easily achieved by OpenMP programmers.

While our stencil performance falls short of the roofline model, especially on the small cores, absolute stencil performance is beyond the scope of this paper. We expect to include in-depth analysis of the performance shortfall in a future publication.

References

1. Andreolli, C.: Eight Optimizations for 3-Dimensional Finite Difference (3DFD) Code with an Isotropic (ISO). https://software.intel.com/en-us/articles/eight-optimizations-for-3-dimensional-finite-difference-3dfd-code-with-an-isotropic-iso. Accessed 21 Oct 2014
2. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. CACM **52**(4), 65 (2009)

3. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High-Performance Programming. Morgan Kauffman, Boston (2013)
4. Dempsey, J.: Plesiochronous phasing barriers. In: Jeffers, J., Reinders, J. (eds.) High Performance Parallelism Pearls, pp. 87–115. Morgan Kauffman, Boston (2015)
5. 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
6. Meadows, L., Kim, J., Wells, A.: Parallelization methods for hierarchical SMP systems. In: Terboven, C., et al. (eds.) IWOMP 2015. LNCS, vol. 9342, pp. 247–259. Springer, Heidelberg (2015). doi:10.1007/978-3-319-24595-9_18
7. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995
8. Sbalzarini, I.F., Walther, J.H., Bergdorf, M., Hieber, S.E., Kotsalis, E.M., Koumoutsakos, P.: PPM a highly efficient parallel particlemesh library for the simulation of continuum systems. J. Comput. Phys. **215**(2), 566 (2006)
9. Madduri, K., Im, E.-J., Ibrahim, K.Z., Williams, S., Ethier, S., Oliker, L.: Gyrokinetic particle-in-cell optimization on emerging multi- and manycore platforms. Parallel Comput. **37**(9), 501 (2011)
10. Schweizer, H., Besta, M., Hoefler, T.: Evaluating the cost of atomic operations on modern architectures. In: Proceedings of Parallel Architectures and Compilation (2015)
11. Dureau, D., Poëtte, G.: Hybrid parallel programming models for AMR neutron Monte-Carlo transport. In: Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo (2013)