OpenMP* on Knights Landing

John Pennycook, Intel Corporation

May 2018, ALCF Performance Workshop

Acknowledgements:
Alex Duran, Jason Sewall, Carlos Rosales-Fernandez

Agenda

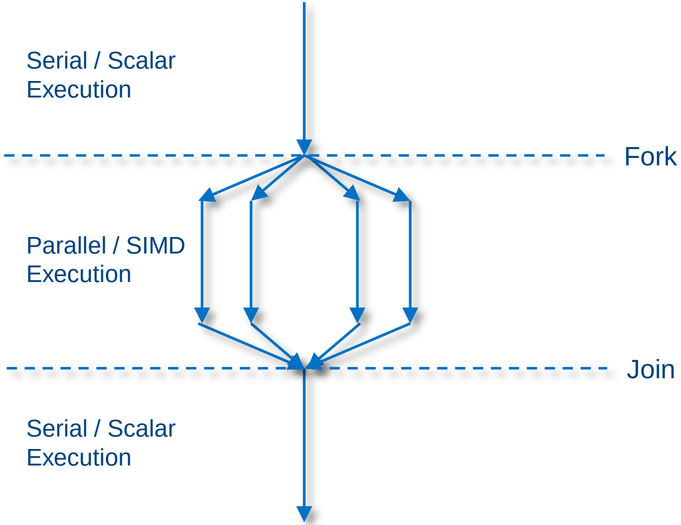
- Think Parallel & SIMD
- Threading
  - Affinity
  - Nested Threading
  - Scheduling & Explicit Worksharing
- Vectorization

Think Parallel & SIMD

Birds-Eye View of OpenMP*

- OpenMP Directives
  - Indicate parallelism opportunities.
  - Compilers not supporting OpenMP are free to ignore directives.

```
#pragma omp parallel shared(A, B, C)
{
    int tid = omp_get_thread_num();
    printf(“Hello from thread %d\n”, tid);

    #pragma omp for simd
    for (int i = 0; i < size; i++) {
        C[i] = A[i] + B[i]
    }
}
```


Serial / Scalar Execution
Fork
Parallel / SIMD Execution
Join
Serial / Scalar Execution

Amdahl’s Law

$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}}$$

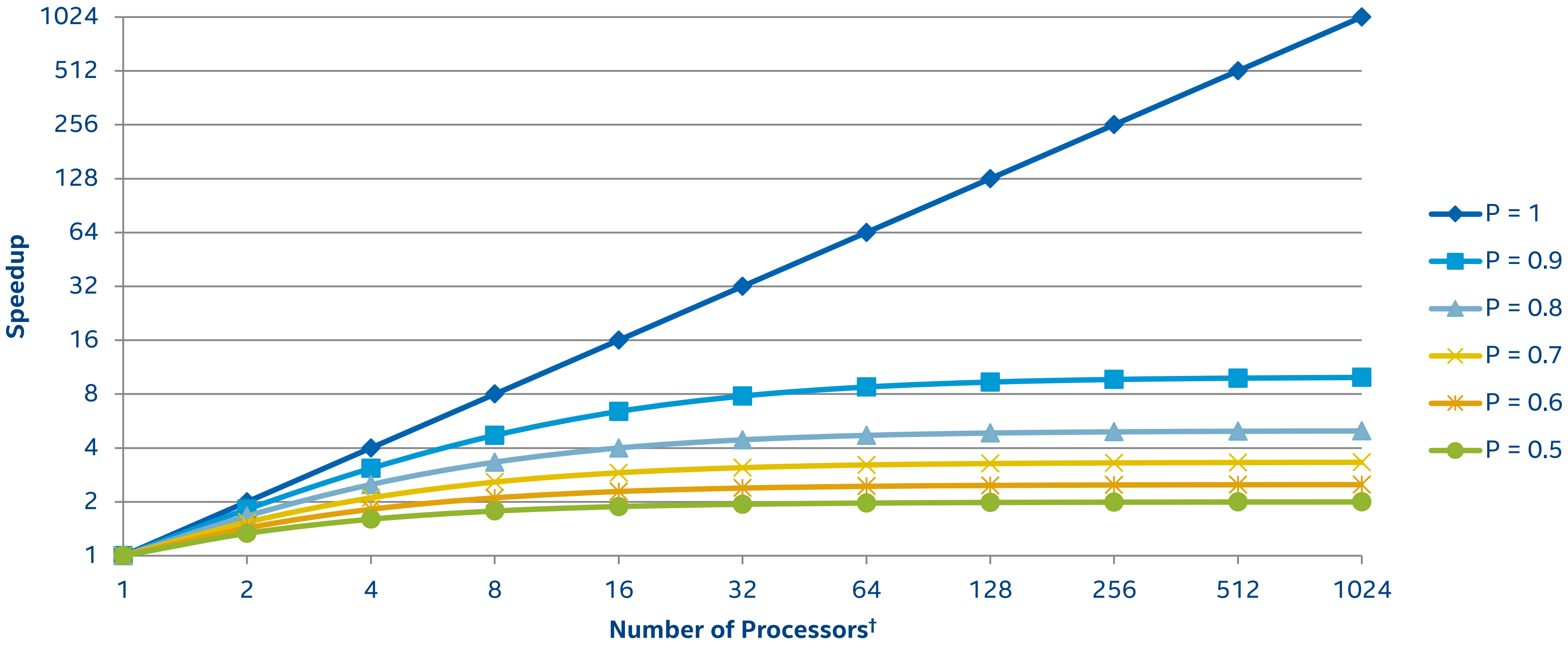
where:

- $S(N)$ = speedup on N processors
- P = fraction of code that can be parallelised
- N = number of processors

The speedup of “strong scaling” applications is governed by Amdahl’s Law.

As $N \to \infty$, $S(N) \to \frac{1}{(1-P)}$.

Impact of Amdahl’s Law



† Amdahl’s Law applies to Cores and SIMD, too!

Think Parallel & SIMD!

- Key takeaways from Amdahl's Law:
  - Maximize P to maximize efficiency and performance at scale
  - Threads/SIMD "bolted on" to serial/scalar applications will not scale.
- Adding pragmas and crossing fingers rarely solves the problem.
  - Consider how hardware should be used before worrying about the implementation.
  - Think of pragmas as a short-hand for telling the compiler what to do.
- Revisit algorithms and throw out assumptions.
  - A parallel implementation of a "slower" algorithm may be faster!

Threading with OpenMP*

Standard OpenMP* Affinity Controls

- OpenMP has two standard environment variables for affinity:
  - OMP_PLACES={place}[,{place}…]*
    Similar to KMP_HW_SUBSET; defines virtual cores to be used by OpenMP.
    Places can be a list of hardware threads or standard short-hands (threads, cores, sockets).
  - OMP_PROC_BIND=[spread | close]
    Similar to KMP_AFFINITY; defines binding of threads to places.

- Decoder Ring:
  - KMP_AFFINITY proclist ≈ OMP_PLACES with list of hardware threads
  - KMP_AFFINITY scatter ≈ spread; compact ≈ close
  - KMP_HW_SUBSET ≈ OMP_PLACES with standard places

OpenMP* Affinity on Theta

- For pure OpenMP* based codes the most effective way to set affinity is to disable affinity in aprun and then use OpenMP settings to bind threads.
- Disabling affinity with aprun is simple:

```
$ aprun -n 1 -N 1 -cc none ./exe
```

- Now threads can be pinned to specific hardware resources using the OMP_PLACES and OMP_PROC_BIND environment variables.
- Rich set of options with lots of flexibility and configuration granularity, but a few simple setups cover the vast majority of production cases.

Affinity Examples

- KMP_HW_SUBSET=64c,1t KMP_AFFINITY=compact,granularity=core
  Launch 64 threads, one per physical core.

- KMP_HW_SUBSET=64c,4t KMP_AFFINITY=compact,granularity=core
  Launch 256 threads, four per physical core.

- OMP_NUM_THREADS=64 OMP_PLACES="cores(64)" OMP_PROC_BIND=spread
  Launch 64 threads, one per physical core.

- OMP_NUM_THREADS=256 OMP_PLACES={0,68,136,204}:64 OMP_PROC_BIND=close
  Launch 256 threads, four per physical core.

Hybrid MPI + OpenMP* Affinity on Theta

- When using hybrid applications aprun must be configured to create pinning ranges for each MPI task, and then OpenMP variables may be set to control thread pinning within each rank processor range.

- Example: 4 MPI ranks,16 threads per rank, 8 nodes

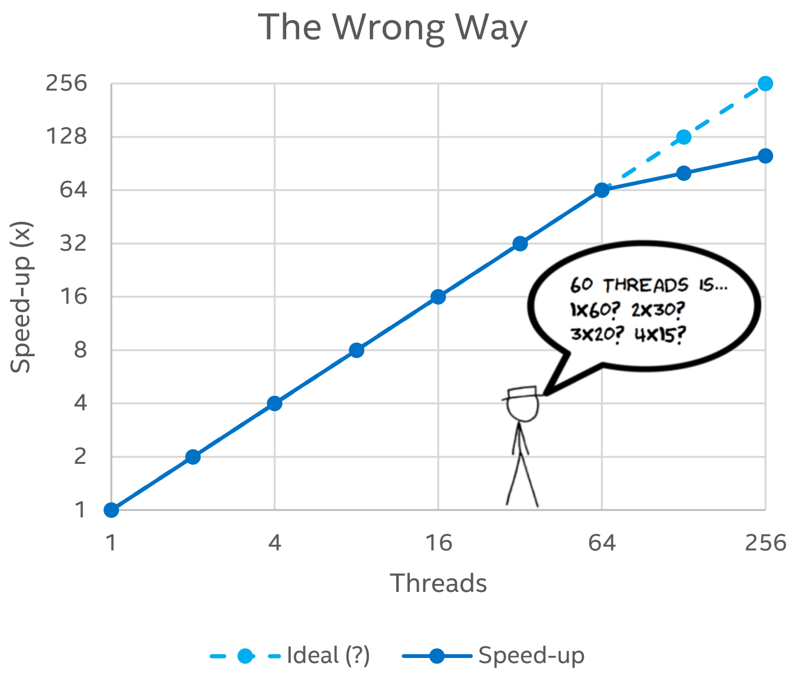
```
export OMP_NUM_THREADS=16
export OMP_PLACES=cores
export OMP_PROC_BIND=spread
aprun -n 32 -N 4 -cc depth -d 64 -j 4 ./exe
```

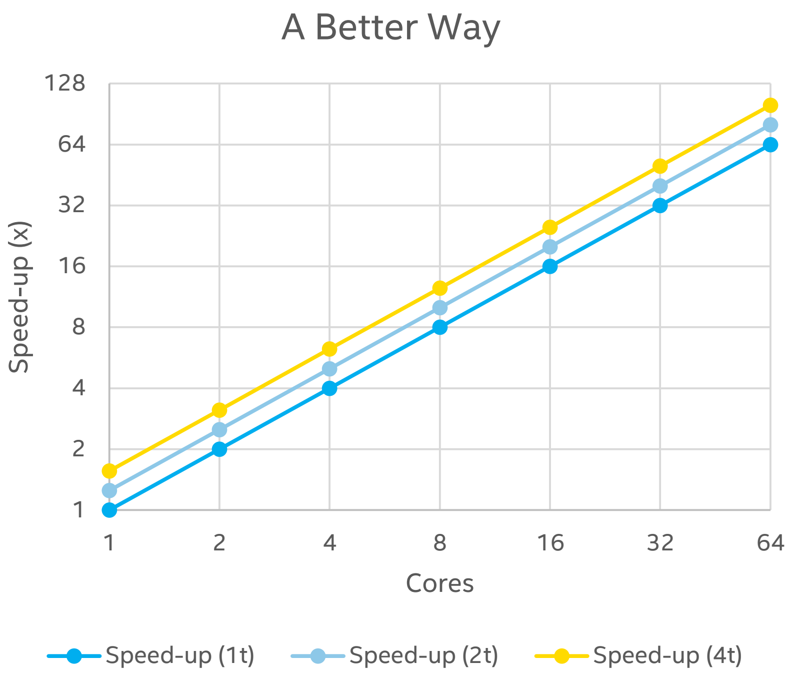
How to Plot OpenMP* Scaling Results

- Why does how we plot scaling results matter?
  - Clarity of presentation
  - May confuse/bias interpretation

- y-axis is clearly performance (or speed-up) but what about x-axis?
  - Threads?
  - Cores?
  - Something else?

How to Plot OpenMP* Scaling Results





Conflates threads/cores and obscures affinity.

Separates scaling from hyperthread gains.

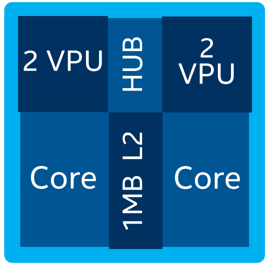
https://software.intel.com/en-us/blogs/2016/12/02/how-to-plot-openmp-scaling-results

Nested Threading and Locality

- Recall that KNL cores are grouped into **tiles**, with two cores sharing an L2.

- Effective capacity depends on locality:
  - 2 cores sharing no data => 2 x 512 KB
  - 2 cores sharing all data => 1 x 1 MB

- Ensuring **good locality** (e.g. through blocking or nested parallelism) is likely to improve performance.



```
#pragma omp parallel for num_threads(ntiles)
for (int i = 0; i < N; ++i)
{
      #pragma omp parallel for num_threads(8)
      for (int j = 0; j < M; ++j)
      {
            …
      }
}
```

Nested Threading – Considerations

- Nested threading has historically been slow due to fork-join overheads.
  - Ensure that there is sufficient work per thread to amortize this.
  - Performance will differ across OpenMP* runtimes.

- OpenMP environment variables:
  - OMP_NESTED=true
  - OMP_NUM_THREADS=64,4

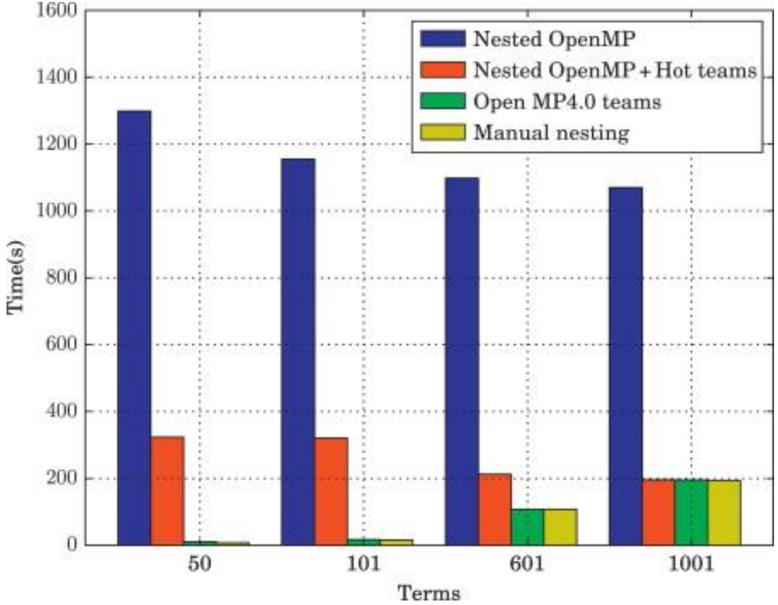
Enable nested threading and set the number of threads at each level of nesting.

- Intel environment variables:
  - KMP_HOT_TEAMS=1
  - KMP_HOT_TEAMS_MAX_LEVEL=2

Permit teams of threads to stay "alive" once created; accelerates fork-join of nested threads.

Nested Threading – Performance Impact

- Impact of nested threading depends on ability of threads to share data.
- Multiple implementations possible:
  - Nested OpenMP* parallel regions
  - OpenMP "teams" construct
  - Explicit (manual) nested threading
- Expect to spend some time fine-tuning nesting behavior.



From "Cosmic Microwave Background Analysis: Nested Parallelism in Practice", in "High Performance Parallel Pearls: Volume 2". Used with permission.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance. See System Configuration(s) slide at back of deck.

Benchmark results were obtained prior to implementation of recent software patches and firmware updates intended to address exploits referred to as "Spectre" and "Meltdown". Implementation of these updates may make these results inapplicable to your device or system.

OpenMP* Schedules

```
schedule([modifier[, modifier] : ]kind[, chunk_size])
```

Kinds:

static	Round-robin distribution of chunks.
dynamic	Threads request chunks from a queue dynamically.
guided	Like dynamic, but decreasing chunk size amortizes overheads of acquiring new chunks.

Modifiers:

monotonic	Chunks assigned to a thread are executed in increasing order of iterations.
nonmonotonic	Chunks may be assigned to a thread in any order.
simd	All threads except the first and last are assigned a number of iterations divisible by the SIMD length.

Other modifiers/kinds are available. See: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

OpenMP* Schedules – Examples

- schedule(monotonic:dynamic, 8)
  Threads work on 8 iterations at a time. To ensure monotonicity, threads likely take work from a single, shared queue.

- schedule(nonmonotonic:dynamic, 16)
  Threads work on 16 iterations at a time. Since monotonicity is not required, runtime is free to use a work-stealing scheduler.

- schedule(simd:static)
  Threads work on ≈(niterations / nthreads) iterations each, with exact work assignment adjusted to permit efficient use of simd instructions.

Explicit Worksharing in OpenMP*

- Naively adding OpenMP pragmas to existing loops may restrict performance, since runtime cannot exploit domain knowledge.

- OpenMP provides mechanisms for explicit worksharing (similar to pthreads/MPI):
  - omp_get_max_threads():
    The maximum number of threads a parallel region can use.
  - omp_get_num_threads():
    The number of threads in the enclosing parallel region. Will be 1 if called from a serial region!
  - omp_get_thread_num():
    The thread id (`tid`) of the calling thread.

Examples:

```
int ntiles = omp_get_max_threads();

#pragma omp parallel for
for (int i = 0; i < ntiles; ++i)
{
  for (int i = 0; i < Ni; ++i)
  {
    for (int j = 0; j < Nj; ++j)
    {
      // work for this tile
    }
  }
}

#pragma omp parallel
{
  int tid = omp_get_thread_num();
  foo(tiles[tid]);
}
```

Explicit Worksharing in OpenMP* – N-Body

Implicit Worksharing:

```
#pragma omp parallel for
for (int i = 0; i < natoms; ++i)
{
  for (int j = 0; j < nneigh[i]; ++j)
  {
    int jj = neighbors[i][j];
    float f = compute_force(i, jj);
    force[i] += f;
    #pragma omp atomic
    force[jj] -= f;
  }
}
```

Explicit Worksharing:

```
#pragma omp parallel for
for (int t = 0; t < ntiles; ++t)
{
  for (int i = 0; i < natoms[t]; ++i)
  {
    for (int j = 0; j < nneigh[t][i]; ++j)
    {
      int jj = neighbors[t][i][j];
      float f = compute_force(i, jj);
      force[t][i] += f;
      force[t][jj] -= f;
    }
  }
}
```

Domain decomposition guarantees no write-conflicts.
Already favored at the MPI level; why throw away the insight for OpenMP?

Explicit Worksharing in OpenMP* – Stencils

Implicit Worksharing:

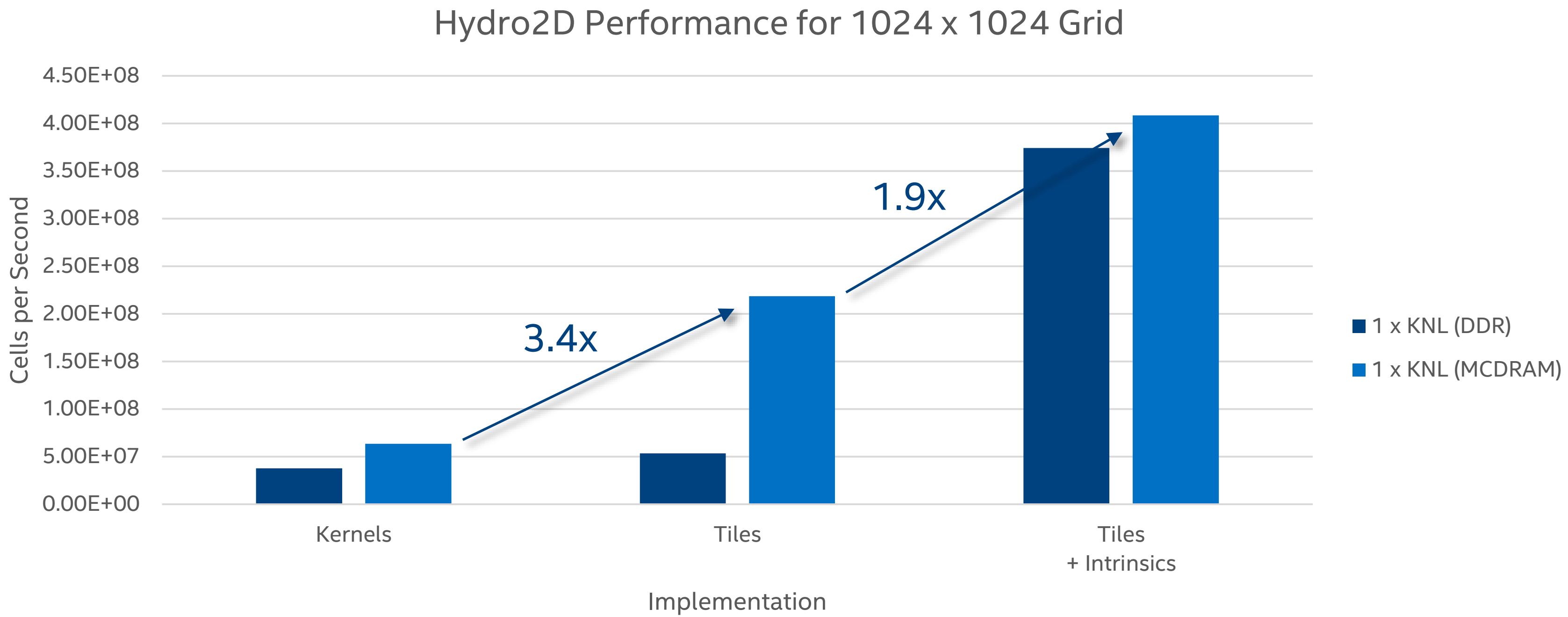
```
#pragma omp parallel for
for (int i = 0; i < N+1; ++i)
{
  flux[i] = foo(cell[i-1], cell[i]);
}


#pragma omp parallel for
for (int i = 0; i < N; ++i)
{
  cell[i] = bar(flux[i], flux[i+1]);
}
```

Explicit Worksharing:

```
#pragma omp parallel for
for (int t = 0; t < ntiles; ++t)
{
  float flux[2];
  flux[0] = foo(cell[-1], cell[0]);
  for (int i = 0; i < N; ++i)
  {
    flux[1] = foo(cell[i], cell[i+1]);
    cell[i] = bar(flux[0], flux[1]);
    flux[0] = flux[1];
  }
}
```

Domain decomposition permits loop fusion / stencil chaining.
Reduces memory footprint and bandwidth requirements.

Explicit Worksharing in OpenMP* – Stencils



Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance. See System Configuration(s) slide at back of deck.

Benchmark results were obtained prior to implementation of recent software patches and firmware updates intended to address exploits referred to as "Spectre" and "Meltdown". Implementation of these updates may make these results inapplicable to your device or system.

Explicit Worksharing in OpenMP* – Manual Nesting

Implicit Worksharing:

```
#pragma omp parallel for num_threads(64)
for (int i = 0; i < N; ++i)
{
  #pragma omp parallel for num_threads(4)
  for (int j = 0; j < M; ++j)
  {
    // work
  }
}
```

Explicit Worksharing:

```
#pragma omp parallel num_threads(256)
{
  int tid = omp_get_thread_num();
  int cid = tid / 4;
  int lid = tid % 4;
  int il = (N/64)*cid;
  int iu = il + (N/64);
  int jl = (M/4)*lid;
  int ju = jl + (M/64);
  for (int i = il; i < iu; ++i)
  {
    for (int j = jl; j < ju; ++j)
    {
      // work
    }
  }
}
```

Removes fork-join overhead of inner parallel loop.
Can employ a specialized barrier for threads on same core (see Parallel Pearls 2).

Vectorization with OpenMP*

Explicit Vectorization – OpenMP* SIMD Loops

#pragma omp simd / !$omp simd => for/do loop is a SIMD loop.

safelen (*length*)	Maximum distance between two iterations executed concurrently by a SIMD instruction.
linear (*list[:linear-step]*)	List items are private and have a linear relationship with respect to the iteration space.
aligned (*list[:alignment]*)	List items are aligned to a platform-dependent value (or the value of the optional parameter).

private (*list*), lastprivate (*list*), reduction (*reduction-identifier:list*) and collapse (*n*) are also supported, with functionality matching that of omp for.

See: OpenMP 4.0 Specification http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

Explicit Vectorization – OpenMP* SIMD Loops

Forced Vectorization

```
#pragma omp simd
for (int i = 0; i < N; ++i)
{
  output[i] = foo();
  printf(“output[i] = %d\n”, output[i]);
}
```

Outer-Loop Vectorization

```
#pragma omp simd
for (int i = 0; i < N; ++i)
{
  while (condition dependent on i)
  {
    // work
  }
}
```

Semi-Automatic Vectorization

```
#pragma omp parallel num_threads(64)
for (int i = 0; i < N; i += VLEN)
{
  float tmp[VLEN];
  #pragma omp simd simdlen(VLEN)
  for (int v = 0; v < VLEN; ++v)
  {
    // work
  }
}
```

Explicit Vectorization – OpenMP* SIMD Functions

#pragma omp declare simd / !$omp declare simd => function called from a SIMD loop.

simdlen (*length*)	Maximum number of concurrent arguments to the function (i.e. maximum SIMD width).
uniform (*argument-list*)	List items have the same value for all SIMD lanes, and can therefore be broadcast.
inbranch notinbranch	Function always called inside a conditional. Function never called inside a conditional.

linear (*argument-list[:linear-step]*) and aligned (*argument-list[:alignment]*) are also supported, with functionality matching that of omp simd.

See: OpenMP 4.0 Specification http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

Explicit Vectorization – OpenMP* SIMD Functions

- Possible to declare multiple SIMD implementations (i.e. vector variants) of a single function.

- Compiler selects the best match based on contextual information at the call site.

- Optimization report (`-qopt-report=5`) includes function matching report:

```
remark #15489: --- begin vector function matching report ---
remark #15490: Function call:
               add(float *, float *, float *, int) with
               simdlen=16, actual parameter types:
               (uniform,uniform,uniform,linear:1)
remark #15492: A suitable vector variant was found (out of 6)
               with zmm, simdlen=16, unmasked,
               formal parameter types:
               (uniform,uniform,uniform,linear:1)
remark #15493: --- end vector function matching report ---
```

- Compiler may emulate a SIMD function by calling another function with smaller `simdlen` multiple times.

```
#pragma omp declare simd simdlen(16)
#pragma omp declare simd simdlen(16)
 uniform(left, right, out)
#pragma omp declare simd simdlen(16)
 uniform(left, right, out) linear(i:1)
void add(float* left, float* right,
         float* out, int i)
{
    out[i] = left[i] + right[i];
}

void foo(float* a, float* b, float* c,
         int N)
{
    #pragma omp simd
    for (int i = 0; i < N; ++i)
    {
        add(a, b, c, i);
    }
}
```

Summary

Summary

- OpenMP* is a great tool for adding thread/SIMD parallelism to an application.

- Adding pragmas to existing code is the bare minimum you can do
  - ….and may not be successful if the parallelism is “bolted on” or too limited in scope

- High performance OpenMP codes pay attention to:
  - Hardware resources (affinity, nesting, tasking, memory bandwidth)
  - Runtime overheads (scheduling, explicit worksharing, tasks)
  - Parallel algorithm design (explicit worksharing, tasks)

Legal Disclaimers

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. § For more information go to www.intel.com/benchmarks.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at www.intel.com.

§ Configurations:
Slide 18 - Measured by University of Cambridge on 2 x Intel® Xeon® processor E5-4650L, 8 cores, 2.6 GHz, Intel® Xeon Phi™ coprocessor 5110P, 60 cores, 1.053 GHz, Intel® Composer XE 2015, Source: "Cosmic Microwave Background Analysis: Nested Parallelism in Practice", in High Performance Parallelism Pearls: Volume 2: Multicore and Many-core Programming Approaches
Slide 24 - Measured by Intel on Intel® Xeon Phi™ coprocessor 5110P, 60 cores, 1.053 GHz, Intel® Composer XE 2016, Source: Intel

Intel, the Intel logo, Look Inside, Xeon, Xeon Phi, are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

intel®

KMP_HW_SUBSET Environment Variable

- Previously called KMP_PLACE_THREADS (now deprecated).
- Restricts the resources available to the OpenMP runtime **without** applying any particular thread affinity.

```
<placement>  := <level> [<level_list>]
<level_list> := <separator><level>[<level_list>]
<separator>  := ‘,’|’x’
<level>      := <positive_integer><level_code>
<level_code> :=
<thread>|<core>|<cache>|<numa>|<socket>
<thread>     := ‘t’|’T’|’thread’
<core>       := ‘c’|’C’|’core’
<cache>      := ‘l1’|’L1’|’cache1’|’l2’|…
<numa>       := ‘n’|’N’|’numa’
<socket>     := ‘s’|’S’|’socket’
```

Require hwloc topology
KMP_TOPOLOGY_METHOD=hwloc

KMP_AFFINITY Environment Variable

- Controls how OpenMP threads are mapped and pinned to hardware threads.
  - compact:
    All virtual cores are assigned threads before moving to the next physical core.
  - scatter:
    Physical cores are assigned threads round-robin before virtual cores.
  - explicit,proclist=[...]:
    Threads are mapped to cores in the specified proclist.
  - granularity=[fine | thread | core]:
    Threads are pinned to virtual cores (fine | thread) or physical cores (core).
  - verbose
    Print the thread-to-core mapping at the start of the run.

- Manual mapping/pinning is very complex; in 99% of cases I'd recommend:
  KMP_AFFINITY=compact,granularity=fine + appropriate KMP_HW_SUBSET.

https://software.intel.com/en-us/node/522691