Structured Parallel Programming

James Reinders
August 1, 2016, Pheasant Run, St Charles, IL
10:45-12:00
<table>
<thead>
<tr>
<th>Time</th>
<th>Presentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:30 am - 10:15 am</td>
<td><strong>Presentation: Computer Architecture Essentials</strong></td>
</tr>
<tr>
<td></td>
<td>Lecturer Room</td>
</tr>
<tr>
<td></td>
<td>James Reinders, Recently Semi-retired, Former Intel Director</td>
</tr>
<tr>
<td>10:45 am - 12:00 pm</td>
<td><strong>Presentation: Structured Parallel Programming</strong></td>
</tr>
<tr>
<td></td>
<td>Lecturer Room</td>
</tr>
<tr>
<td></td>
<td>James Reinders, Recently Semi-retired, Former Intel Director</td>
</tr>
<tr>
<td>1:00 pm - 1:45 pm</td>
<td><strong>Presentation: Performance: SIMD, Vectorization and Performance Tuning</strong></td>
</tr>
<tr>
<td></td>
<td>Lecturer Room</td>
</tr>
<tr>
<td></td>
<td>James Reinders, Recently Semi-retired, Former Intel Director</td>
</tr>
</tbody>
</table>

Knights Landing Clustering and Memory Modes, use and implications on the future of architecture and memory configurations.

Vectorization, current state of the art thinking, use and implications on the future of data parallelism through threading + SIMD instructions.
KEEP CALM AND THINK PARALLEL
Structured Parallel Programming

- Michael McCool
- Arch Robison
- James Reinders

Uses Cilk Plus and TBB as primary frameworks for examples.

Appendices concisely summarize Cilk Plus and TBB.

www.parallelbook.com

(pointers to teaching materials, ours and others!)
Parallel Patterns: Overview
Structured Programming with Patterns

• Patterns are “best practices” for solving specific problems.
• Patterns can be used to organize your code, leading to algorithms that are more scalable and maintainable.
• A pattern supports a particular “algorithmic structure” with an efficient implementation.
• Good parallel programming models support a set of useful parallel patterns with low-overhead implementations.
Some Basic Patterns

Serial: Sequence
→ Parallel: Superscalar Sequence
Serial: Iteration
→ Parallel: Map, Reduction, Scan, Recurrence…
(Serial) Sequence

A serial sequence is executed in the exact order given:

\[ F = f(A) ; \]
\[ G = g(F) ; \]
\[ B = h(G) ; \]
Superscalar Sequence

Developer writes “serial” code:

\[
\begin{align*}
F &= f(A) \\
G &= g(F) \\
H &= h(B, G) \\
R &= r(G) \\
P &= p(F) \\
Q &= q(F) \\
S &= s(H, R) \\
C &= t(S, P, Q)
\end{align*}
\]

- Tasks ordered only by data dependencies
- Tasks can run whenever input data is ready
(Serial) Iteration

The iteration pattern repeats some section of code as long as a condition holds:

```c
while (c) {
    f();
}
```

Each iteration can depend on values computed in any earlier iteration.

The loop can be terminated at any point based on computations in any iteration.
(Serial) Countable Iteration

The iteration pattern repeats some section of code a specific number of times

```c
for (i = 0; i<n; ++i) {
    f();
}
```

This is the same as

```c
i = 0;
while (i<n) {
    f();
    ++i;
}
```
Parallel “Iteration”

The serial iteration pattern actually maps to several different parallel patterns. It depends on whether and how iterations depend on each other… Most parallel patterns arising from iteration require a fixed number of invocations of the body, known in advance.
Map

- **Map** invokes a function on every element of an index set.
- The index set may be abstract or associated with the elements of an array.
- Corresponds to “parallel loop” where iterations are independent.

**Examples:** gamma correction and thresholding in images; color space conversions; Monte Carlo sampling; ray tracing.
Reduction

- *Reduce* combines every element in a collection into one using an *associative* operator:
  \[ x + (y + z) = (x + y) + z \]

- For example: *reduce* can be used to find the sum or maximum of an array.

- Vectorization may require that the operator *also* be *commutative*:
  \[ x + y = y + x \]

Examples: averaging of Monte Carlo samples; convergence testing; image comparison metrics; matrix operations.
Scan

- Scan computes all partial reductions of a collection

\[
A[0] = B[0] + \text{init}; \\
\text{for } (i=1; i<n; ++i) \{ \\
\quad A[i] = B[i] + A[i-1]; \\
\}
\]

- Operator must be (at least) associative.
- Diagram shows one possible parallel implementation using three-phase strategy

Examples: random number generation, pack, tabulated integration, time series analysis
Geometric Decomposition/Partition

- **Geometric decomposition** breaks an input collection into sub-collections
- **Partition** is a special case where sub-collections do not overlap
- Does not move data, it just provides an alternative “view” of its organization

**Examples:** JPG and other macroblock compression; divide-and-conquer matrix multiplication; coherency optimization for cone-beam recon.
Stencil

• *Stencil* applies a function to neighbourhoods of an array.

• Neighbourhoods are given by set of relative offsets.

• Boundary conditions need to be considered.

**Examples:** image filtering including convolution, median, anisotropic diffusion
Implementing Stencil

Vectorization can include converting regular reads into a set of shifts.

Strip-mining reuses previously read inputs within serialized chunks.
nD Stencil

- *nD Stencil* applies a function to neighbourhoods of an nD array
- Neighbourhoods are given by set of relative offsets
- Boundary conditions need to be considered

**Examples:** image filtering including convolution, median, anisotropic diffusion; simulation including fluid flow, electromagnetic, and financial PDE solvers, lattice QCD
- **Pipeline** uses a sequence of stages that transform a flow of data
- Some stages may retain state
- Data can be consumed and produced incrementally: “online”

**Examples:** image filtering, data compression and decompression, signal processing
Fork-Join: Efficient Nesting

- Fork-join can be nested
- Spreads cost of work distribution and synchronization.
- This is how `cilk_for`, and `tbb::parallel_for` are implemented.

Recursive fork-join enables high parallelism.
KEEP CALM
USE TASKS
AVOID THREADS
Choosing a non-proprietary parallel abstraction

<table>
<thead>
<tr>
<th>non-proprietary</th>
<th>BLAS, FFTW</th>
<th>MPI</th>
<th>OpenMP®</th>
<th>TBB</th>
<th>Cilk™ Plus</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog. lang.</td>
<td>Fortran, C, C++</td>
<td>Fortran, C, C++</td>
<td>Fortran or C</td>
<td>C++</td>
<td>C++</td>
</tr>
</tbody>
</table>

Use abstractions !!!

Avoid direct programming to the low level interfaces (like pthreads).

PROGRAM IN TASKS, NOT THREADS

Is OpenCL® low level? For HPC – YES.
## Choosing a non-proprietary parallel abstraction

<table>
<thead>
<tr>
<th>non-proprietary</th>
<th>BLAS, FFTW</th>
<th>MPI</th>
<th>OpenMP&lt;sup&gt;®&lt;/sup&gt;</th>
<th>TBB</th>
<th>Cilk™ Plus</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog. lang.</td>
<td>Fortran, C, C++</td>
<td>Fortran, C, C++</td>
<td>Fortran or C</td>
<td>C++</td>
<td>C++</td>
</tr>
</tbody>
</table>

Choose First
(limited functions)
Choosing a non-proprietary *parallel abstraction*

<table>
<thead>
<tr>
<th>non-proprietary</th>
<th>BLAS, FFTW</th>
<th>MPI</th>
<th>OpenMP®</th>
<th>TBB</th>
<th>Cilk™ Plus</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog. lang.</td>
<td>Fortran, C, C++</td>
<td>Fortran, C, C++</td>
<td>Fortran or C</td>
<td>C++</td>
<td>C++</td>
</tr>
</tbody>
</table>

**Choose First**
(limited functions)

**Cluster**
(distributed memory)
Choosing a non-proprietary *parallel abstraction*

<table>
<thead>
<tr>
<th>non-proprietary</th>
<th>BLAS, FFTW</th>
<th>MPI</th>
<th>OpenMP®</th>
<th>TBB</th>
<th>Cilk™ Plus</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog. lang.</td>
<td>Fortran, C, C++</td>
<td>Fortran, C, C++</td>
<td>Fortran or C</td>
<td>C++</td>
<td>C++</td>
</tr>
</tbody>
</table>

Choose First (limited functions)

Cluster (distributed memory)

Node (shared memory)
Choosing a non-proprietary *parallel abstraction*

<table>
<thead>
<tr>
<th>non-proprietary</th>
<th>BLAS, FFTW</th>
<th>MPI</th>
<th>OpenMP®</th>
<th>TBB</th>
<th>Cilk™ Plus</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog. lang.</td>
<td>Fortran, C, C++</td>
<td>Fortran, C, C++</td>
<td>Fortran or C</td>
<td>C++</td>
<td>C++</td>
</tr>
</tbody>
</table>

Choose First (limited functions)

Cluster (distributed memory)

Node (shared memory)

Up and coming for C++ (keywords, compilers)

Because… you just have to expect “more”

Affect future C++ standards? (2021?)
Choosing a non-proprietary *parallel abstraction*

<table>
<thead>
<tr>
<th>non-proprietary</th>
<th>BLAS, FFTW</th>
<th>MPI</th>
<th>OpenMP*</th>
<th>TBB</th>
<th>Cilk™ Plus</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog. lang.</td>
<td>Fortran, C, C++</td>
<td>Fortran, C, C++</td>
<td>Fortran or C</td>
<td>C++</td>
<td>C++</td>
</tr>
<tr>
<td>implemented</td>
<td>vendor libraries</td>
<td>many</td>
<td>in compiler</td>
<td>portable</td>
<td>in compiler</td>
</tr>
<tr>
<td>supported by</td>
<td>most vendors</td>
<td>open src &amp; vendors</td>
<td>most compilers</td>
<td>ported most everywhere</td>
<td>gcc and Intel (llvm future)</td>
</tr>
</tbody>
</table>

Compare...

<table>
<thead>
<tr>
<th>proprietary</th>
<th>NVidia®CUDA</th>
<th>NVidia OpenACC</th>
<th>Intel LEO</th>
</tr>
</thead>
<tbody>
<tr>
<td>purpose</td>
<td>data parallel</td>
<td>offload</td>
<td>offload</td>
</tr>
<tr>
<td>target (perf.)</td>
<td>NVidia GPUs</td>
<td>NVidia GPUs</td>
<td>portable</td>
</tr>
<tr>
<td>alternative</td>
<td>OpenCL*</td>
<td>OpenMP 4.0</td>
<td>OpenMP 4.0</td>
</tr>
</tbody>
</table>
Choosing a non-proprietary parallel abstraction

<table>
<thead>
<tr>
<th></th>
<th>BLAS, FFTW</th>
<th>MPI</th>
<th>OpenMP*</th>
<th>TBB</th>
<th>Cilk™ Plus</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog. lang.</td>
<td>Fortran, C, C++</td>
<td>Fortran, C, C++</td>
<td>Fortran or C</td>
<td>C++</td>
<td>C++</td>
</tr>
<tr>
<td>implemented</td>
<td>vendor libraries</td>
<td>many</td>
<td>in compiler</td>
<td>portable</td>
<td>in compiler</td>
</tr>
<tr>
<td>supported by</td>
<td>most vendors</td>
<td>open src &amp; vendors</td>
<td>most compilers</td>
<td>ported most everywhere</td>
<td>gcc and Intel (Illum future)</td>
</tr>
</tbody>
</table>

Compare...
Choosing a non-proprietary parallel abstraction

<table>
<thead>
<tr>
<th>non-proprietary</th>
<th>BLAS, FFTW</th>
<th>MPI</th>
<th>OpenMP*</th>
<th>TBB</th>
<th>Cilk™ Plus</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog. lang.</td>
<td>Fortran, C, C++</td>
<td>Fortran, C, C++</td>
<td>Fortran or C</td>
<td>C++</td>
<td>C++</td>
</tr>
<tr>
<td>implemented</td>
<td>vendor libraries</td>
<td>many</td>
<td>in compiler</td>
<td>portable</td>
<td>in compiler</td>
</tr>
<tr>
<td>supported by</td>
<td>most vendors</td>
<td>open src &amp; vendors</td>
<td>most compilers</td>
<td>ported most everywhere</td>
<td>gcc and Intel (llvm future)</td>
</tr>
<tr>
<td>composable?</td>
<td>usually</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>memory</td>
<td>shared/distributed</td>
<td>distributed</td>
<td>shared (in implementations)</td>
<td>shared memory</td>
<td>shared memory</td>
</tr>
<tr>
<td>tasks</td>
<td>yes</td>
<td>n/a</td>
<td>YES</td>
<td>YES</td>
<td>limited keywords, TBB</td>
</tr>
<tr>
<td>explicit SIMD</td>
<td>internal</td>
<td>n/a</td>
<td>YES (OpenMP 4.0: SIMD)</td>
<td>use compiler options, OpenMP directives, or Cilk Plus keywords</td>
<td>keywords</td>
</tr>
<tr>
<td>offload</td>
<td>some</td>
<td>n/a</td>
<td>YES (OpenMP 4.0: SIMD)</td>
<td>use Cilk Plus or OpenMP</td>
<td>keywords</td>
</tr>
</tbody>
</table>

Best options for Performance and Performance Portability
For TBB - we asked ourselves:

- How should C++ be extended?
  - “templates / generic programming”

- What do we want to solve?
  - Abstraction with good performance (scalability)
  - Abstraction that steers toward easier (less) debugging
  - Abstraction that is readable
Intel® Threading Building Blocks (Intel® TBB)

C++ Library for parallel programming
• Takes care of managing multitasking

Runtime library
• Scalability to available number of threads

Cross-platform
• Windows*, Linux*, Mac OS* and others

http://threadingbuildingblocks.org/
### Rich Feature Set for Parallelism

<table>
<thead>
<tr>
<th>Generic Parallel Algorithms</th>
<th>Flow Graph</th>
<th>Concurrent Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efficient scalable way to exploit the power of multi-core without having to start from scratch.</td>
<td>A set of classes to express parallelism as a graph of compute dependencies and/or data flow</td>
<td>Concurrent access, and a scalable alternative to serial containers with external locking</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task Scheduler</th>
<th>Flow Graph</th>
<th>Concurrent Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sophisticated work scheduling engine that empowers parallel algorithms and the flow graph</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flow Graph</th>
<th>Concurrent Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Concurrent access, and a scalable alternative to serial containers with external locking</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Synchronization Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic operations, a variety of mutexes with different properties, condition variables</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task Scheduler</th>
<th>Flow Graph</th>
<th>Concurrent Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sophisticated work scheduling engine that empowers parallel algorithms and the flow graph</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalable memory manager and false-sharing free allocators</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task Scheduler</th>
<th>Flow Graph</th>
<th>Concurrent Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sophisticated work scheduling engine that empowers parallel algorithms and the flow graph</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Synchronization Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic operations, a variety of mutexes with different properties, condition variables</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task Scheduler</th>
<th>Flow Graph</th>
<th>Concurrent Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sophisticated work scheduling engine that empowers parallel algorithms and the flow graph</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalable memory manager and false-sharing free allocators</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task Scheduler</th>
<th>Flow Graph</th>
<th>Concurrent Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sophisticated work scheduling engine that empowers parallel algorithms and the flow graph</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Synchronization Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic operations, a variety of mutexes with different properties, condition variables</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task Scheduler</th>
<th>Flow Graph</th>
<th>Concurrent Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sophisticated work scheduling engine that empowers parallel algorithms and the flow graph</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalable memory manager and false-sharing free allocators</td>
</tr>
</tbody>
</table>
Rich Feature Set for Parallelism

**Generic Parallel Algorithms**
- Efficient scalable way to exploit the power of multi-core without having to start from scratch.

**Flow Graph**
- A set of classes to express parallelism as a graph of compute dependencies and/or data flow

**Concurrent Containers**
- Concurrent access, and a scalable alternative to serial containers with external locking

**Synchronization Primitives**
- Atomic operations, a variety of mutexes with different properties, condition variables

**Task Scheduler**
- Sophisticated work scheduling engine that empowers parallel algorithms and the flow graph

**Threads and Synchronization**

**Memory Allocation**
- Scalable memory manager and false-sharing free allocators

**Parallel algorithms and data structures**

**Threads and synchronization**

**Memory allocation and task scheduling**

---

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

**Loop parallelization**
- `parallel_for`
- `parallel_reduce`
  - load balanced parallel execution
  - fixed number of independent iterations
- `parallel_scan`
  - computes parallel prefix
  \[ y[i] = y[i-1] \text{ op } x[i] \]

**Parallel Algorithms for Streams**
- `parallel_do`
  - Use for unstructured stream or pile of work
  - Can add additional work to pile while running
- `parallel_for_each`
  - `parallel_do` without an additional work feeder
- `pipeline / parallel_pipeline`
  - Linear pipeline of stages
  - Each stage can be parallel or serial in-order or serial out-of-order.
  - Uses cache efficiently

**Parallel function invocation**
- `parallel_invoke`
  - Parallel execution of a number of user-specified functions

**Computational graph**
- `flow::graph`
  - Implements dependencies between nodes
  - Pass messages between nodes
Parallel For
tbb::parallel_for

Has several forms.

- **parallel_for(lower, upper, functor);**
  - Execute *functor(i)* for all \( i \in [\text{lower}, \text{upper}) \)
- **parallel_for(lower, upper, stride, functor);**
  - Execute *functor(i)* for all \( i \in \{\text{lower}, \text{lower}+\text{stride}, \text{lower}+2*\text{stride}, \ldots\} \)
- **parallel_for(range, functor);**
  - Execute *functor(subrange)* for all subrange in range

#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>
#define N 10

inline int Prime(int & x) {
  int limit, factor = 3;
  limit = (long)(sqrtf((float)x)+0.5f);
  while ( (factor <= limit) && (x % factor))
    factor ++;
  x = (factor > limit ? x : 0);
}

int main (){  
  int a[N];
  // initialize array here...
  tbb::parallel_for (0, N, 1,
    [&] (int i){
      Prime (a[i]);
    });
  return 0;
}
Recursive parallelism

Split range...

.. recursively...

... until \( \leq \text{GrainSize} \)

\([\text{Data}, \text{Data}+\text{GrainSize})\]

\([\text{Data}, \text{Data}+\text{N}/2)\]

\([\text{Data}, \text{Data}+\text{N}/k)\]

\([\text{Data}+\text{N}/2, \text{Data}+\text{N})\]

\([\text{Data}+\text{N}/2, \text{Data}+\text{N})\]

tasks available to thieves
Rich Feature Set for Parallelism

<table>
<thead>
<tr>
<th>Generic Parallel Algorithms</th>
<th>Flow Graph</th>
<th>Concurrent Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efficient scalable way to exploit the power of multi-core without having to start from scratch.</td>
<td>A set of classes to express parallelism as a graph of compute dependencies and/or data flow</td>
<td>Concurrent access, and a scalable alternative to serial containers with external locking</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sophisticated work scheduling engine that empowers parallel algorithms and the flow graph</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Synchronization Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic operations, a variety of mutexes with different properties, condition variables</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread Local Storage</th>
<th>Threads</th>
<th>Miscellaneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlimited number of thread-local variables</td>
<td>OS API wrappers</td>
<td>Thread-safe timers and exception classes</td>
</tr>
</tbody>
</table>

Memory Allocation

- Scalable memory manager and false-sharing free allocators
The MOST popular abstract parallelism model for C++
The MOST popular abstract parallelism model for C++

Down with OpenMP!
Sorry OpenMP

You just do not cut it.

(for C++)
Sorry OpenMP

Down with OpenMP!

(for C++)
The next few slides are based on following paper from WHPCF’14:

STAC-A2 on Intel Architecture: From Scalar Code to Heterogeneous Application

Evgeny Fiksman       Sania Salahuddin
evgeny.fiksman@intel.com       sania.salahuddin@intel.com

SC’14, New Orleans, November 16th, 2014
STAC-A2 overview (https://stacresearch.com/)

- A vendor independent market risk analysis benchmark
- Defined by Securities Technology Analysis Center (STAC*)
- Calculate “Greeks” – sensitivity of the option price to changes in parameters of the underlying market
- Heston option pricing model & Least Squares Monte Carlo of Longstaff & Schwartz
- Benchmark Metrics
  - Speed (GREEKS.TIME.COLD/WARM)
  - Workload scalability (MAX_ASSETS, MAX_PATHS)
- Power & Space efficiency
- Quality
TBB used on STAC-A2 Benchmark – beat OpenMP

130829 and 140507 use identical hardware
140507 and 140814 use identical source code
This is portable code: no “intrinsics”
~1.45x from each HW generation, SW change worth at least 2 HW generations

Parallelization choices matter
Hold on!!!

Who is the invited keynote speaker for OpenMP conference in September 2015?
How did Intel TBB beat OpenMP annotations on STAC-A2?

OpenMP annotations work well when
- You control the whole machine
- You have one level of parallelism
- You want to take low level control of scheduling, placement,…

Intel TBB tends to out perform OpenMP when…
- You don’t know about the machine you’ll run on
- You have many levels of parallelism (recursive, or in libraries)
- You’re happy to let the runtime handle things

Both are portable: Intel TBB does not require compiler support.
Both are reasonably performance portable in practice, although TBB is composable – which can be a significant advantage in perf. port.
OpenMP is very popular – and works very well on technical applications (like HPC) with C and Fortran. But, for C++… TBB is better.

*I was having a little fun… to make a point.*

We love TBB!!!

Long live OpenMP!
Nested parallelism is important to exploit.

Trending: more and more so.
OpenMP Nested Parallelism: HOT TEAMS

NESTED PARALLEL:

By DEFAULT, any parallel worker that executes a parallel construct does that work inside the same worker thread.

PRO: controlled memory footprint (including stack space)

CON: no load balancing
OpenMP Nested Parallelism: HOT TEAMS

OpenMP worker threads – created ONCE PER PROGRAM

Additional level(s) created and released repeatedly

NESTED PARALLEL:

TURN ON NESTING (no code changes – done with environment variables)

PRO: load balancing

CON: high overhead, potential oversubscription (runaway memory/stack usage being the key issue)
Table of Contents...

Foreword
Introduction
Numerical Weather Prediction Optimization
WRF Goddard Microphysics Scheme Optimization
Pairwise DNA Sequence Alignment Optimization
Accelerated Structural Bioinformatics for Drug Discovery
Amber PME Molecular Dynamics Optimization
Low Latency Solutions for Financial Services
Parallel Numerical Methods in Finance
Wilson Dslash Kernel From Lattice QCD Optimization

Cosmic Microwave Background Analysis: Nested Parallelism In Practice
Visual Search Optimization
Radio Frequency Ray Tracing
Exploring Use of the Reserved Core
High Performance Python Offloading
Fast Matrix Computations on Asynchronous Streams
MPI-3 Shared Memory Programming Introduction
Coarse-Grain OpenMP for Scalable Hybrid Parallelism
Exploiting Multilevel Parallelism with OpenMP
OpenCL: There and Back Again
OpenMP vs. OpenCL: Difference in Performance?
Prefetch Tuning Optimizations
SIMD functions via OpenMP
Vectorization Advice
Portable Explicit Vectorization Intrinsics
Power Analysis for Applications and Data Centers
OpenMP Nested Parallelism: HOT TEAMS

Chapter 18: Exploiting Multilevel Parallelism with OpenMP

Nested OpenMP is an optional feature of the OpenMP standard. Its support is subject to the compilers and runtime libraries. The default is to ignore OpenMP parallel regions within a running parallel region; in OpenMP parlance, the nested regions are serialized. This can be overridden by setting OMP_NESTED=true. The Intel OpenMP runtime has greatly improved performance for nested OpenMP since releasing Intel Composer XE 15.1 with so-called HOT_TEAMS. They are enabled in our experiments by setting these environment variables:

```
export KMP_HOT_TEAMS_MODE=1
export KMP_HOT_TEAMS_MAX_LEVEL=2
export MKL_DYNAMIC=false
```

Note that we set MKL_DYNAMIC=false for DGEMM or FFT when they are used.

**HOT TEAMS MOTIVATION**

Hot teams is an extension to OpenMP supported by the Intel runtime. It reduces the overhead of OpenMP parallelism. It works with standard OpenMP code but changes it. It is a logical extension that may inspire similar capabilities in other implementations.

To understand “hot teams,” it is important to know that any modern implementation of OpenMP, in order to avoid the cost of creating and destroying pthreads, has the OpenMP runtime maintain a pool of OS threads (pthreads on Linux) that it has already created. This is standard practice in OpenMP runtimes because OS thread creation is normally quite expensive. However, OpenMP also has a concept of a thread team, which is the set of pthreads that will execute

```
export OMP_NESTED=true
export OMP_NUM_THREADS=12,4
export OMP_PLACES=threads
export OMP_PROC_BIND=spread,close
mpirun -np 5 ./myapp
```
Chapter 10: Cosmic Microwave Background Analysis: Nested Parallelism In Practice

Chapter 10
COSMIC MICROWAVE BACKGROUND ANALYSIS

costs are prohibitively expensive when the nested regions are encountered often, such as when the
treads are spawned for an inner-most loop.

There is, however, support for an experimental feature in the Intel® OpenMP runtime (Version 15
Update 1 or later) known as “hot teams” that is able to reduce these overheads, by keeping a pool of
threads alive (but idle) during the execution of the non-nested parallel code. The use of hot teams is
controlled by two environment variables: KMP_HOT_TEAMS_MODE and KMP_HOT_TEAMS_MAX_LEVEL.
To keep unused team members alive when team sizes change we set KMP_HOT_TEAMS_MODE=1, and because
we have two levels of parallelism we set KMP_HOT_TEAMS_MAX_LEVEL=2.

Care must also be taken with thread affinity settings. OpenMP 4.0 provides new environment vari-
ables for handling the physical placement of threads, OMP_PROC_BIND and OMP_PLACES, and these are
compatible with nested parallel regions. To place team leaders on separate cores, and team members on
the same core, we set OMP_PROC_BIND=spread, close and OMP_PLACES=threads.
KEEP CALM AND ASK SOME QUESTIONS
James Reinders. Parallel Programming Enthusiast

James has been involved in multiple engineering, research and educational efforts to increase use of parallel programming throughout the industry. James worked 10,001 days as an Intel employee 1989-2016, and contributed to numerous projects including the world's first TeraFLOP/s supercomputer (ASCI Red), first 3 TeraFLOP/s supercomputer (ASCI Red upgrade), the world's first TeraFLOP/s microprocessor (Intel® Xeon Phi™ coprocessor) and the world's first 3 TeraFLOP/s microprocessor (Intel® Xeon Phi™ Processor). James been an author on numerous technical books, including VTune™ Performance Analyzer Essentials (Intel Press, 2005), Intel® Threading Building Blocks (O'Reilly Media, 2007), Structured Parallel Programming (Morgan Kaufmann, 2012), Intel® Xeon Phi™ Coprocessor High Performance Programming (Morgan Kaufmann, 2013), Multithreading for Visual Effects (A K Peters/CRC Press, 2014), High Performance Parallelism Pearls Volume 1 (Morgan Kaufmann, Nov. 2014), High Performance Parallelism Pearls Volume 2 (Morgan Kaufmann, Aug. 2015), and Intel® Xeon Phi™ Processor High Performance Programming - Knights Landing Edition (Morgan Kaufmann, 2016).