The Intel® Compiler(2): Parallelism etc

Martyn Corden
Developer Products Division
Software & Services Group
Intel Corporation

June 2010
Agenda

• Memory considerations

• A few language specific features
  – C/C++
  – Fortran

• Parallelism
  – libraries
  – Auto-parallelization
  – OpenMP
    – Trouble-shooting OpenMP code
Compiling for 64 bits

• Not a lot changes, but..
  – Beware variables that may contain addresses
  – e.g. use `size_t`

• Memory models for static data
  – Default assumes <2GB static data (-mcmodel small)
  – If > 2GB static data, build with
    `-mcmodel medium -shared-intel` (slight performance impact)
    – When a shared library is built, the compiler driver automatically adds `-fpic` to build position-independent code
      – Resulting shared library supports all memory models.
  – Code built with `-mcmodel medium` should only be linked to static libraries that are also built with `-mcmodel medium`
  – Relocation errors typically indicate a memory model mismatch

• Often simpler to use dynamically allocated data
  – Usually the case for C/C++, but not for older Fortran
Compilation time, memory usage

• Computer-generated functions are sometimes large
  – Often dense code; large, explicit expressions; many repeated sub-expressions
  – Combinatorics may cause long compile times and large memory requirement as the compiler tries to optimize
    – May see “internal threshold was exceeded” message
      – Means some optimizations were skipped, to reduce memory requirement and/or compile time
      – Can try –override-limits if have plenty of memory (& time!)
    – 32 bit compilers are limited to 2 GB on IA32, 4 GB on Intel 64
      – “out of memory” message if this is exceeded.
  – To reduce memory footprint and compilation time:
    – Try splitting up very large functions
    – Use temporary variables for repeated expressions
    – Compile at lower optimization if not important for performance (e.g. initialization).
C/C++ Compiler for Linux*

• Compatibility with GCC:
  – C/C++ source code compatibility
    – Except very few extensions (e.g. nested functions in C)
    – Tracks gcc support for C++0x
  – Command line compatibility
    – Achieved for most important options
  – Object Code interoperability
    – achieved – even for complex C++ and OpenMP*
    – Uses GNU C++ RTL

• Compatibility test: Linux kernel build
  – ICC 10.x and kernel 2.6.x: No manual changes to source code required but requires ‘wrapper’ script
C++0x Features

New C++0x features enabled by switch
-std=c++0x

- Lambda functions
- variadic templates
- extern templates
- static assertions
- RVALUE references
- C99-compliant preprocessor
- __func__ predefined identifier
- ... and some more

The feature list corresponds to the functionality offered by the switch -std=c++0x of the GNU GCC 4.3 compiler. See http://gcc.gnu.org/gcc-4.3/cxx0x_status.html for details
C++ Lambda Functions
Will be part of future C++0x Standard

• Refers to lambda calculus/functional programming
• *Lambda* is the Lisp term for a function with no name
  – Compiler creates a *closure*, an anonymous function object with an *environment* comprising elements of the local context
  – Simplifies dealing with functions that are used once only
  – Greatly facilitates use of template libraries with loop constructs such as STL or Intel® TBB

```
std::vector<int> someList;
int total = 0;
std::for_each( someList.begin(), someList.end(), [&](int x) {total += x;});
std::cout << total;
```
C99 support

- restricted pointers (**restrict** keyword).
- variable-length Arrays
  - void func(int size1, int size2, double a[][size2], ...)
- flexible array members
- complex number support (**_Complex** keyword)
- hexadecimal floating-point constants
- compound literals
- designated initializers
- mixed declarations and code
- macros with a variable number of arguments
- inline functions (**inline** keyword)
- boolean type (**_Bool** keyword)

- Not yet: 128 bit long doubles   (see later)
Using Intel® IPP to accelerate valarray operations

- A valarray is simply an array of values. It is templated on its value type and was added to C++ Standard for performance reasons – part of the STL
- The operations on the value types are designed to exploit low level hardware features in form of vectorization
- Intel® C++ Compiler Version 11.0 provides an optimized replacement valarray header file
- No source changes required to use
- Provide parallelism by vectorization through std headers (perf_header/c++)

The public member functions of class valarray (“what you can do with valarrays”) can be found e.g. at http://www.aoc.nrao.edu/~tjuerges/ALMA/STL/html/classstd_1_1valarray.html
Example for valarray usage

```cpp
#include <valarray>
int N = 1000;
std::valarray<float> va(N), vb(N), vc(N);
float tsum, dotp;

va = vb + vc; // array addition
va = sin(vb) + cos(vc); // trig. functions
va = vb * 2.0f - vc; // do math operation for corresponding elements
tsum = va.sum(); // Summarize the entire valarray using sum,min,max
dotp = (va * vb).sum(); // Express complicated concepts simply: dotproduct
```

How to use?

```bash
icpc -use-intel-optimized-headers source.cpp
```

Includes headers, links needed libraries
Some useful Fortran features

- **-cxxlib**
  - link to default C++ RTL (for mixed language apps)

- **-fexceptions**
  - allows C++ exceptions to propagate through Fortran calls

- **-assume buffered_io**
  - export FORT_BUFFERED=true

- **-convert big_endian**
  - Converts data from/to big endian format on input/output

- **-traceback**
  - For low-overhead stack trace in case of runtime failure

- **-mkl** to link the MKL libraries
Stack Overflow

• Automatic and temporary arrays are put on the stack for performance reasons with the Intel® Fortran Compiler.
• This can result in a segmentation fault (due to stack overflow) if the array is large.
• Two Solutions:
  – Increase your stack size
    – ulimit -s unlimited (for bash) or limit stacksize unlimited (for csh) (note you may need to specify a large # instead of unlimited on some OS).
  – Put the arrays on the heap
    – For a small performance penalty -heap-arrays
Debugging Switches (Fortran)

- Check calling interfaces
  - -gen-interfaces, -warn interfaces

- Runtime checks
  - -check
    - Uninitialized variables
    - Array bounds checking
    - Dissociated or uninitialized pointers
  - -traceback
    - Lightweight stack trace (without -g)

- Debugging Switches (in addition to -g -debug all)
  - -debug-parameters (make PARAMETER symbols visible)
  - -debug extended (debugging of optimized code)
Most Fortran 2003 features

- IEEE_arithmetic, IEEE_exceptions
- ISO_C_Binding
- Object oriented features
- Many new intrinsics
- Environment access
- Allocatable components of derived types
- Asynchronous I/O
- Procedure pointers (mostly)

not yet:
- User-defined derived type I/O
- Parameterized derived types
Levels of Parallelism

- **SIMD instructions**
  - Compiler can vectorize loops automatically

- **Instruction level**
  - Processor schedules, you don’t see it

- **Threading** (usually shared memory)
  - Native Linux* or Windows* threads
  - OpenMP*
  - Simplest for multi-core

- **Message Passing** (usually distributed memory)
  - Various MPI; also CoArray Fortran, ...
  - Problems that are too large for shared memory

- “embarrassingly parallel” multiprocessing
Providing Choice of Parallel Methods

Classification is indicative only, categories overlap...

- **Vector Instructions**
  - Vectorizer for SSE
  - Vectorizer for AVX

- **Data parallelism**
  - Ct
  - Open MP

- **Task parallelism**
  - Cilk++
  - TBB

- **Cluster Parallelism**
  - MPI
  - CoArray Fortran

**Resource Management**

**Multi-core IA Platforms**

**Many-core IA Platforms**
Ways to Introduce Threading

• Threaded libraries, e.g. Intel® MKL, IPP
  – Easy and effective, if it fits your problem

• Auto-parallelization by the compiler
  – Easy to do, but limited in scope
    – “simple” loops where compiler can prove it is safe

• Asynchronous I/O

• Native threads
  – Mostly used for task-level parallelism
  – Not so easy to program and debug

• OpenMP
  – Designed to facilitate data-level parallelism
  – (relatively) easier programming and debugging
  – Some support for task parallelism, especially in OpenMP 3.0
  – portable

Programmer responsible for thread safety
Many components of MKL have threaded versions
- Based on the compiler’s OpenMP runtime library
- Level 1,2 and 3 BLAS, LAPACK
- Sparse BLAS
- Discrete Fourier Transforms
- Vector math and random number functions
- Direct sparse solvers, e.g. PARDISO

Link threaded or non-threaded interface
- libmkl_intel_thread.a or libmkl_sequential.a
- Use the link line advisor at

Set the number of threads
- export MKL_NUM_THREADS or OMP_NUM_THREADS
- Call mkl_set_num_threads or omp_set_num_threads
Intel® Math Kernel Library delivers nearly linear scaling

BLAS Threaded Performance
Intel® Xeon® Quad-Core Processor

- Intel MKL 8 Threads
- Intel MKL 4 Threads
- Intel MKL 2 Threads
- Intel MKL 1 Thread
- ATLAS 8 Threads
- ATLAS 1 Thread

Quad-core Intel® Xeon® Processor E5472
3.0GHz, 8Mb L2 Cache, 16Gb Memory
RedHat Linux Enterprise Server 5.0
Intel® MKL 10.1; Atlas 3.8.0 DGEMM function

Source: Intel Corporation. Test results aggregated with overall performance scores based on geometric mean. Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, refer to www.intel.com/performance/resources/benchmark_limitations.htm.

Software & Services Group, Developer Products Division

Copyright © 2010, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners.
Auto-parallelization

• The compiler can thread simple loops automatically
  – Build with –parallel
  – Need at least –O2  (cf OpenMP works at –O0)
  – Loops must fulfill “simple” conditions
  – reports which loops parallelized, & if not, why not
    – -par-report2  etc
  – tune parallelization cost model with -par-thresholdn
    – default is n=100;  try n=99
  – Directives can help compiler: #pragma parallel [always]

• Based on the same RTL threading calls as OpenMP:
  – call _kmpe_fork_call
  – These are wrappers around the low-level pthreads libraries
  – Recognizes same OpenMP environment variables
Conditions for Auto-parallelization

• Loop count known at entry (no while or DO WHILE)
  – But not necessarily at compile time
  – No jumps into or out of loop
• Loop iterations are independent
  – **No function calls** (or prove no side effects)
    – except if inlined
  – No aliasing
    (accessing same variable via different pointers)
  – No structures such as \( X(I+1) = Y(I+1) + X(I) \)
  – Reductions are allowed
    – But partial sums may lead to rounding differences
• Enough work to amortize parallel overhead
  – \( O(10^4) \) cycles
• Conditions for OpenMP loops are similar
  – But **you** are responsible for dependencies, not compiler!!
Example: matrix multiply

```c
void matmul(double a[][NUM], double b[][NUM],
        double c[][NUM], int* n) {
    int i,j,k;

    for(i=0;i<NUM;i++) {
        for (k=0;k<NUM;k++) {
            for (j=0;j<NUM;j++) {
                c[i][j] = c[i][j] + a[i][k]*b[k][j];
            }
        }
    }
}
```

```
icc -parallel -par-report2 -vec-report2 -fargument-noalias -c matmul.c DNUM=500
matmul.c(5): (col. 4) remark: LOOP WAS AUTO-PARALLELIZED.
...
matmul.c(7): (col. 10) remark: LOOP WAS VECTORIZED.
```
OPENMP - advantages

• Standardized API based on compiler directives
  – Latest version is 3.0
  – C++ and Fortran, Linux* and Windows*
  – Directives treated as comments by non-OpenMP compiler
  – Single source base for serial and parallel implementations
    – Helps debugging
  – Allows incremental parallelism
  – OpenMP rules make checking tools a little easier
OpenMP Programming Model

Fork-Join Parallelism:

Master thread spawns a team of threads as needed.
Parallelism is added incrementally: that is, the sequential program evolves into a parallel program. Threads are not destroyed, but are returned to a pool.
OPENMP – where to thread

• Start by mapping out high level structure
• Where does your program spend the most time?
  – If you don’t know, do quick performance analysis
    – VTune, PTU, gprof, ...
  – If only x% of your program is parallel, the speedup is always less than x%, however many cores and threads.
• Prefer data parallelism
  – Easier to load balance
  – Easier to scale to more cores
• Favor coarse grain (high level) parallelism
  – E.g. outer loop of nest, slowest varying grid coordinate, high level driver routines
  – Reduces overhead
  – Improves data locality and reuse for each thread
  – Can’t parallelize iterative loops such as time stepping.
Example: Square_Charge

- calculates the electrostatic potential at a series of points in a plane due to a uniform square distribution of charge
  - Essentially, a 2D integral over the square

<table>
<thead>
<tr>
<th>Square_charge</th>
<th>loops over points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twod_int</td>
<td>integrate over y</td>
</tr>
<tr>
<td>Trap_int</td>
<td>integrate over x</td>
</tr>
<tr>
<td>Func</td>
<td>calculates 1/r potential</td>
</tr>
</tbody>
</table>

- Inline func()
- Vectorize loop over x
- Thread loop over y
  - Avoid race conditions
- Could instead thread loop over points, or use MPI
OpenMP: How do threads interact?

• OpenMP is a shared memory model
  – Threads communicate by sharing variables
• Unintended sharing of data causes race conditions:
  – Race condition: when the program outcome changes as the threads are scheduled differently
• To control race conditions...
  – Use synchronization to protect data conflicts
• Synchronization is expensive so...
  – Change how data is accessed to minimize the need for synchronization
OPENMP – data

• Identify which data are **shared** between threads, which need a **separate copy** for each thread.

• It’s helpful (but not required) to make shared data explicitly global in Modules or common blocks, thread private data as local and automatic.

• Dynamic allocation is OK (malloc, ALLOCATE)
  – Allocate in serial region if shared
  – Allocate in parallel region if each thread needs own copy

• Each thread gets its own private stack, but the heap is shared by all threads.
OPENMP – data scoping

- Distinguish lexically explicit parallel regions
  - from subroutines called from these
  (that may contain no OpenMP directives)

- Explicit !$OMP PARALLEL to !$OMP END PARALLEL
  - Everything defaults to SHARED (except loop index)
  - Local data: can change with PRIVATE clause
  - Global data: can declare common blocks, module variables as !$OMP THREADPRIVATE
  - Initial values inside parallel region undefined
    - unless use FIRSTPRIVATE (local variables)
    - or COPYIN (globals)
  - Final values after parallel region undefined
    - unless use LASTPRIVATE (local variables)
  - Functions called must be threadsafe
Thread Safety

• A threadsafe function can be called simultaneously from multiple threads, and still give correct results
  – Potential conflicts must either be protected (synchronized) or avoided (by privatization)
  – Static local data: each thread may access same copy!
  – Automatic data: each thread has own copy & stack

• ifort defaults:
  – Local scalars are automatic
  – Local arrays are static

• When compiling with –openmp, default changes
  – Local arrays are automatic
  – Same as compiling with –auto
  – This may increase the required stack size
    – Beware segmentation faults
Making a function thread safe

• With the compiler
  – Compile with -openmp
  – Compile with -auto
    – May have less impact on serial optimization

• In source code
  – Use AUTOMATIC keyword in declarations
    – But doesn’t cover compiler-generated temporaries
  – Declare function as RECURSIVE
    – Covers whole function, including compiler-generated code
    – Best way to make code thread safe if you don’t want to depend on build options

• In either case:
  – don’t use -save or SAVE keyword
  – Avoid global variables,
    – Or don’t write to them unless synchronized
Thread Safe Code

• OpenMP has various synchronization constructs to protect operations that are potentially unsafe
  – REDUCTION clause
  – !$OMP CRITICAL
  – !$OMP SINGLE
  – etc
Thread Safe Libraries

• The Intel® Math Kernel library is threadsafe
  – Sequential version as well as threaded version

• The Intel Fortran runtime library has two versions
  – The default is **not** threadsafe  (libifcore)
    – Build with `-threads` to link to threadsafe version  (libifcoremt)
  – If you build with `-openmp`, the threadsafe version is linked by default
Performance considerations

- Start with optimized serial code, vectorized inner loops, etc. (-O3 -xsse4.2 -ipo ...)
- Ensure sufficient parallel work
- Minimize data sharing between threads
  - Unless read-only
- Avoid false sharing of cache lines
  - Each thread thinks its copy of A(i,j) may have been invalidated
  - Reversing subscripts of A improves data locality for each thread
    - Contiguous memory access also permits vectorization of inner loop
- Scheduling options
  - Consider DYNAMIC or GUIDED if work is unevenly distributed between loop iterations

!$OMP parallel do
do i=1,nthreads
  do j=1,1000
    A(i,j) = A(i,j) + ..
Timers for threaded apps

• The Fortran standard timer `CPU_TIME` returns “processor time”
  – It sums time over all threads/cores
  – Like the “User time” in the Linux “time” command
  – So threaded apps don’t seem to run faster than serial ones😊

• The Fortran intrinsic subroutine `SYSTEM_CLOCK` returns data from the real time clock
  – Does not sum over cores
  – Like the “real time” in the Linux “time” command
  – Can be used to measure the speedup due to threading

• `dclock` (Intel-specific function) can also be used
Thread Affinity Interface

- Allows OpenMP threads to be bound to physical or logical cores
  - export environment variable KMP_AFFINITY=
    - physical use all physical cores before assigning logical cores (hyperthreads)
    - compact assign threads to consecutive cores on same socket (eg to benefit from shared cache)
    - scatter assign threads to cores on alternating sockets (eg to maximize channels to memory)
  - Helps optimize access to memory or cache
  - Particularly important if Hyperthreading is enabled
    - else some physical cores may be idle while others run multiple threads

- See compiler documentation for more detail
NUMA considerations

• Want memory allocated “close” to where it will be used
  – “first touch” determines where allocated
  – So initialize data in an OpenMP loop in the same way you plan to use it later:

```c
!$OMP parallel do
do i=1,n
do j=1,m
  A(j,i) = 0.0
enddo
docommenddo
docommenddo

!$OMP parallel do
do i=1,n
do j=1,m
dowork(A(j,i))
enddo
```

• Remember to set KMP_AFFINITY
Common problems

• **Insufficient stack size**
  - Most frequently reported OpenMP issue!
  - Typical symptom: *seg fault* during initialization

• For whole program (shared + local data):
  - Increase shell limits to large value
    - (address space, memory allocated only if needed)
  - Bash : `ulimit -s [value in KB]` or `[unlimited]`
    - Can only increase once!
  - C shell: `limit stacksize 1000000` (for 1 GB)
  - Windows*: `/F:100000000` (value in bytes)
    - Typical OS defaults ~10 MB

• For individual thread (thread local data only)
  - `export OMP_STACKSIZE=[size]`, default 4m (4 MB)
  - Actually allocates memory, so don’t make too large
Tips for Debugging OpenMP apps

• Run with `OMP_NUM_THREADS=1`
  - Generate threaded code, but run with a single thread
  - If still fails, excludes race conditions or other synchronization issues as cause
  - If it works, try Thread Checker

• Build with `-openmp-stubs -auto`
  - RTL calls are resolved; but no threaded code is generated
  - allocate local arrays on the stack, as for OpenMP
  - If works, check for missing FIRSTPRIVATE, LASTPRIVATE
  - If still fails, eliminates threaded code generation as cause
  - If works without `-auto`, implicates changed memory model
    - Perhaps insufficient stack size
    - Perhaps values not preserved between successive calls
Tips for Debugging OpenMP apps

• If debugging with PRINT statements
  – print out the thread number with \texttt{omp\_get\_thread\_num()} 
  – remember to USE \texttt{OMP\_LIB} \texttt{(declares this integer!)} 
  – the internal I/O buffers are threadsafe (with –openmp), but the order of print statements from different threads is undetermined.

• Debug with –O0 –openmp
  – Unlike most other optimizations, OpenMP threading is not disabled at –O0
Further Information

Threading Fortran applications for parallel performance on multi-core systems

Developing Multithreaded Applications: A Platform Consistent Approach

- The Intel® C++ and Fortran Compiler User and Reference Guides,
- And the User Forums and Knowledge Base,
  http://software.intel.com/en-us/articles/tools
Summary

• Comprehensive set of tools for multi-core and cluster parallelism from Intel for x86 architecture
  – Best performance on Intel architecture, and competitive performance on AMD systems
  – Intel tools can be used to standardize x86 development C++/Fortran development

• Our focus is on
  – Best Performance
  – Comprehensive coverage of parallelism
  – Ease of use
  – Compatibility and software investment protection

Visit http://intel.com/software/products
INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

Intel, the Intel logo, Itanium, Pentium, Intel Xeon, Intel Core, Intel Centrino and VTune are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries.

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

Copyright © 2010. Intel Corporation.
Lambda Function and Intel® TBB
Serial Example

- static void SerialApplyFoo( float a[], size_t n ) {
  for( size_t i=0; i!=n; ++i )
    Foo(a[i]);
}

Will be parallelized by dividing iteration space of i into chunks using Intel® TBB
Lambda Function and Intel® TBB
Parallel without \( \lambda \)-function Feature

- class ApplyFoo {
  float *const my_a;
  public:
  ApplyFoo( float *a ) : my_a(a) {}
  void operator() (blocked_range<int>& range) const {
    float *a = my_a;
    for( int i = range.begin(); i != range.end(); ++i )
      Foo(a[i]);
  }
};

- void ParallelApplyFoo(float a[], size_t n) {
  parallel_for( blocked_range<int>(0, n),
                ApplyFoo(a),
                auto_partitioner() );
}
Lambda Function and Intel® TBB†
Using λ-Functions Feature - Much simpler !!

void ParallelApplyFoo(float a[], size_t n )
{
    parallel_for( blocked_range<int>( 0, n ),
        [](const blocked_range<size_t>& range)
        {
            for( int i= range.begin(); i!=range.end(); ++i )
                Foo(a[i]);
        },
        auto_partitioner() );

† Please note, that the initial beta release will be shipped with Intel TBB 2.0; thus the lambda functionality cannot be tested with Intel TBB initially during the beta program. TBB 2.1 which does support lambda functionality will be included in a beta update.

blue = original code
red = provided by TBB
black = boilerplate for library
Standard OpenMP Environment Variables

- **OMP_STACKSIZE**
  - Set thread stack size, default 4MB (Intel64)
- **OMP_SCHEDULE = static | dynamic | guided | runtime**
  - Set scheduling mode at runtime (default static)
- **OMP_NESTED**
  - Enable nested parallelism (default disabled)
- **OMP_DYNAMIC**
  - Enable dynamic adjustment of the number of threads (default disabled)
- **OMP_NUM_THREADS**
  - Set number of threads (defaults to number of processors)
- **OMP_THREAD_LIMIT**
  - Maximum number of threads for a parallel region
OpenMP* 3.0 Support

- Intel® Compilers 11.0 for C++ and Fortran fully compliant with OpenMP 3.0
  - Standard released May 2008
    - See [www.openmp.org](http://www.openmp.org)

- Numerous extensions, including:
  - Tasking for unstructured parallelism
  - Better support for nested parallelism
  - Collapsing of nested loops
  - Enhanced loop scheduling control
OpenMP* 3.0 TASK Construct
The most important new OpenMP feature

- OpenMP 2.0 focused mostly on loop based, data parallelism
- OpenMP 3.0 TASK construct expands on OpenMP parallel sections
  - TASKs supported for both C/C++ and Fortran
- A task has
  - Code to execute
  - A data environment (it owns its data)
  - An assigned thread that executes the code and uses the data
- Two activities: packaging and execution
  - Each encountering thread packages a new instance of a task (code & data)
  - Some thread in the team executes the task at some later time
OpenMP* 3.0: Nested Parallelism

• Better support for nested parallelism
• Per-thread internal control variables
  – Allows, for example, calling `omp_set_num_threads()` inside a parallel region.
  – Controls the team sizes for next level of parallelism
• Library routines to determine depth of nesting, IDs of parent/grandparent etc. threads, team sizes of parent/grandparent etc. teams
  • `omp_get_active_level()`
  • `omp_get_ancestor(level)`
  • `omp_get_teamsize(level)`
• Nesting more important as number of cores increases