OpenMP 4.0
Agenda

- Intro to OpenMP
- Support for Compute Devices
- SIMD
- Task Enhancements
- Thread Affinity
- Other improvements
- Demo
- Bibliography
Intro to OpenMP

OpenMP

- For shared memory systems
- For C, C++, and Fortran
- Works by adding compiler directives to code
- Developers have to control variables
- www.openmp.org
- Version 4.0 has been released July 2013
Intro to OpenMP

OpenMP 3.x

Features

- Create threads with shared and private memory
- Parallel sections and loops
- Different work scheduling algorithms for load balancing loops
- Lock, critical and atomic operations to avoid race conditions
- Combining results from different threads
- Nested parallelism
- Generating task to be executed by threads
Intro to OpenMP

OpenMP 4.0

New Features

▶ Support for compute devices
▶ SIMD support for vectorization
▶ Thread affinity support
▶ Thread cancellation
▶ Task enhancements
▶ Extended support for
  • Tasking (groups, dependencies, abort)
  • Reductions (user defined reductions)
  • Atomics (sequential consistency)
OpenMP and Multicore

- OpenMP is done for multicore architectures
- Memory and threading model map naturally
- Lightweight
- Mature, Widely available and used

The OpenMP Memory Model

- All threads have access to the same, *globally shared* memory
- Data in *private memory* is only accessible by the thread owning this memory
- No other thread sees the change(s) in *private memory*
- Data transfer is through shared memory and is 100% transparent to the application
Intro to OpenMP

OpenMP/Performance

- The transparency and ease of use of OpenMP hide “Two-faces”
  - Makes things pretty easy
  - May mask performance bottlenecks
- Dream: an OpenMP application “just performs well”
- This is not always the case

Trouble

- Two more obscure things can decrease performance:
  - cc-NUMA effect
  - False Sharing
- Neither of these are restricted to OpenMP
  - But they most show up because you used OpenMP
  - In any case they are important enough to cover here
False Sharing

- Storing data into a shared cache line invalidates the other copies of that line
- The system is not able to distinguish between changes within one individual line

Red Flags

- Be alert if **ALL** of these three conditions are met:
  - Shared data is *modified* by multiple processors
  - Multiple threads operate on the *same cache line(s)*
  - Update occurs *simultaneously* and very *frequently*

- Use local data where possible
- Shared *read-only* data does not lead to false sharing
cc-NUMA

- Important on cc-NUMA systems
  - If not optimal, longer memory access times and hotspots
- OpenMP does not provide support for cc-NUMA
- Placement comes from Operating System
- Linux use: *First Touch* placement policy by default
- Tools: numactl, KMP_AFFINITY
Support for Compute Devices
Support for Compute Devices

- Intro to OpenMP
- Support for Compute Devices
  - Introduction
  - New Directives
  - Explicit Offloading
- SIMD
- Task Enhancements
- Thread Affinity
- Other improvements
- Demo
- Bibliography
Support for Compute Devices

Introduction

- Goal: support a wide variety of compute devices (Intel Xeon Phi Coprocessor but not yet NVIDIA)
- OpenMP 4.0 adds mechanisms to describe regions of code where data and/or computation should be moved to another computing device
- `omp target`

Memory Model in OpenMP 4.0

- Device has its own data environment and its own shared memory
- Threads can be bundled in a teams of threads
- These threads can have memory shared among threads of the same team
Support for Compute Devices

OpenMP Language Terminology[1]

- **OpenMP threads**: A thread that is managed by the OpenMP runtime system
- **Processor**: Implementation defined hardware unit on which one or more OpenMP threads can execute.
- **Device**: An implementation defined logical execution engine (A device could have one or more processors).
- **Team**: A set of one or more threads participating in the execution of a parallel region.
- **League**: The set of thread teams created by a target construct or a teams construct
- **Contention group**: An initial thread and its descendent threads
- **Device data environment**: A data environment defined by a target data or target construct
- **Mapped variable**: An original variable in a data environment with a corresponding variable in a device data environment.
- **Mapable type**: A type that is valid for a mapped variable. If a type is composed from other types (such as the type of an array or structure element) and any of the other types are not mappable then the type is not mappable.
Support for Compute Devices

New Directives and Functions for Devices

- `omp target [map]` make a region to execute on device
- `omp teams` creates a league of thread teams
  - The `omp_get_num_teams` routine returns the number of teams executing in a `teams` region
  - The `omp_get_team_num` routine returns the team number, which is an integer between 0 and one less than the value returned by `omp_get_num_teams`
- `omp distribute` distributes a loop over the teams in the league
- `omp declare target / omp end declare target` marks function(s) that can be called on the device
Support for Compute Devices

target Construct

- Transfert control from the host to the device
- Syntax Fortran
  ```fortran
  !$omp target[clause[[], clause],...]
  block
  ```
- Syntax C/C++
  ```c
  #pragma omp target[clause[[], clause],...]
  block
  ```
- Clauses
  ```plaintext
  device(scalar-integer-expression)
  map(alloc | to | from | tofrom: list)
  if(scalar-expr)
  ```
Support for Compute Devices

Data mapping

- Host memory and device memory are distinct
- OpenMP 4.0 allows host and device memory to be “shared”
- To manage the both, the relation between variables on host and on target:
  - **to** existing host variables copied to a corresponding variable in the target before
  - **from** target variables copied back to a corresponding variable in the host after
  - **tofrom** both from and to
  - **alloc** neither from nor to, but ensure the variable exists on the both without links
Support for Compute Devices

Execution Model

- The **target construct** transfers the control flow to the target device
  - The transfer clauses control direction of data flow
  - Array notation is used to describe array length
  - Establish a device data environment
  - Host thread waits until offloaded region completed
    - Use other OpenMP constructs for asynchronicity

- The **target data** construct creates a scoped device data environment
  - The transfer clauses control direction of data flow
  - The device data environment is valid through the lifetime of the target data region
  - Create a *data* environment to keep data on devices
    - Avoid frequent transfer or overlap computation/comm
    - Pre-allocate temporary fields

- Use **target update** to request data transfers from within a target data region
Support for Compute Devices

Explicit Offloading

- Programmer explicitly control data and function movement between the Host and Phi(s)
  - data are copied (not shared)
  - Must be bitwise copy-able (pointers not relocated)
- Works with Fortran, C and C++
Support for Compute Devices

**target declare Construct**

- Declare one or more functions to also be compiled for the target device
- **Syntax (Fortran)**

  ```fortran
  !$omp declare target[(proc-name-list|list)]
  ```

- **Syntax C/C++**

  ```c
  #pragma omp declare target
       [function-declarations]
  #pragma omp end declare target
  ```

- The tagged functions will be compiled for
  - Host execution
  - Target execution
teams Construct

- Creates a league of thread teams
  - The master thread of each team executes the teams region
  - Number of teams is specified with num_teams()
  - Each team executes num_threads() threads
- A teams constructs must be “perfectly” nested in a target construct
  - No statements or directives outside the teams construct
- Only special OpenMP constructs can be nested inside a teams construct:
  - distribute
  - parallel
  - parallel for or parallel do
  - parallel sections
Support for Compute Devices

distribute Construct

- New kind of worksharing construct
  - Distribute the iterations of the associated loops across the master threads of a teams construct
  - No implicit barrier at the end of th construct

- Syntax Fortran

```fortran
!$omp distribute [clause[, clause],...] do-loops
```

- Syntax C/C++

```c
#pragma omp distribute[clause[, clause],...] for-loops
#pragma omp end declare target
```

dist_schedule(kind[, chunk_size])

- if specified scheduling kind must be static
- chunks are distributed in round-robin fashion of chunks with size chunk_size
- if no chunk size specified, chunks are of (almost) equal size; each team receives at least one chunk
program nbthreads
    use omp_lib
    integer host_threads, trgt_threads
    host_threads = omp_get_max_threads()
!$omp target map(from:target_threads)
    trgt_threads = omp_get_max_threads();
!$omp end target
    print *, "host threads =", host_threads
    print *, "trgt threads =", trgt_threads
end program nbthreads

Compilation

$ ifort -openmp nbthreads.f90
$./a.out
$ srun -N1 -A BULL_MIC ./a.out
    host threads = 16
    trgt threads = 236
program teams
use omp_lib
integer starti, i

 !$omp target teams num_teams(59), thread_limit(4)
 !$omp distribute
 do starti=1,59

 !$omp parallel do
 do i=1,4
   print *,i, omp_get_team_num(), omp_get_thread_num()
 end do
 !$omp end parallel do
end do
 !$omp end target teams distribute
end program teams

Compilation

```
~/2014/OpenMP4$ ./teams2 | sort | grep "team= 5"
i= 1 team= 5 thread= 0
i= 2 team= 5 thread= 1
i= 3 team= 5 thread= 2
i= 4 team= 5 thread= 3
```
Support for Compute Devices
SIMD
SIMD

- Intro to OpenMP
- Support for Compute Devices
- SIMD
  - Introduction
  - SIMD Constructs
- Task Enhancements
- Thread Affinity
- Other improvements
- Demo
- Bibliography
SIMD on Intel Architecture

- SSE: 128 bit, with 2 DP and 4 SP
- AVX: 256 bit, with 4 DP and 8 SP
- MIC/AVX-2: 512 bit with 8 DP and 16 SP
Once upon a time before OpenMP 4.0

Support required vendor-specific extensions
- Programming models like Intel Cilk Plus
- Compiler pragmas (#pragma vector)
- Low-level constructs (_mm_add_pd())

```c
#pragma omp parallel for
#pragma vector always
#pragma ivdep
do i = 1, N
    a(i) = b(i) + ...
end do
```

“You need to trust your compiler to do the right things”
**SIMD Constructs**

- SIMD: Single Instruction Multiple Data
- OpenMP can enable vectorization of both serial as well as parallelized loops
- *vectorization* i.e. processing multiple elements of an array at the same time
- Done using **SIMD** instructions

**Targets loops**

- Can target inner or outer loops
- Developer asserts loop is suitable for SIMD (no loop-carried dependencies and iterations can be evaluated in parallel)
- Developer should validate results
- **Developer responsible for results**
New Directives for SIMD Loop Construct

- **omp simd** marks a loop to be executed using SIMD lanes
- **omp declare simd** marks a function that can be called from a SIMD loop
- **omp parallel for simd** marks a loop for thread work-sharing as well as SIMDing

Subroutine somme(A,B,n)

```fortran
implicit none
integer :: n, i
real (kind=8) :: A(n), B(n)
real (kind=8) :: som
som = 0.0
!$omp simd reduction(+:som)
do i=1,n
   som = som + A(i)*B(i)
end do
print *, "som =", som
end subroutine somme
```
SIMD pragma clauses

- **reduction(operator:v1,v2,...)**
  - v1, v2, ... are reduction variables for operation “operator”
  - operator include computing average or sum of arrays into a single scalar value

- **linear(v1:steps1, v2:step2,...)**
  - declares one or more list items to be private to a SIMD lane and to have a linear relationship with respect to the iteration space of a loop

- **safelen(length)**
  - no two iterations executed concurrently with SIMD instructions can have a greater distance in the logical iteration space than this value

- **aligned(v1:alignment, v2:alignment)**
  - declares that the object to which each list item points is aligned to the number of bytes expressed in the optional parameter of the aligned clause.

- **collapse(number of loops)**
  - Nested loop iterations are collapsed into one loop with a larger iteration space.

- **private(v1, v2, ...), lastprivate (v1, v2, ...)**
  - declares one or more list items to be private to an implicit task or to a SIMD lane, lastprivate causes the corresponding original list item to be updated after the end of the region.
Why Linear/Uniform

- Unless uniform or linear are specified each parameter to the function will be treated as a vector.

```
# pragma omp declare simd uniform(a) linear(i:1)
void test(float *a, int i)
```

- `a` is a pointer
- `i` is a sequence of integers (i, i+1,i+2...)
- `a[i]` is a unit-stride load/store

```
# pragma omp declare simd
void test(float *a, int i)
```

- `a` is a vector of pointers
- `i` is a vector
- `a[i]` is a gather/scatter
Restrictions applying pragma omp simd

- Applied to for loops only
- Induction variables should be signed or unsigned int
- The associated loops must be structured blocks
- A program must not branch into or out of a SIMD region.
- No OpenMP construct can appear inside a simd region
SIMD Worksharing Construct

- Parallelize and vectorize a loop nest
  - Distribute a loop’s iteration space across a thread team
  - Subdivide loop chunks to fit a SIMD vector register

- Syntax for C/C++
  ```
  #pragma omp for simd [clause[,, clause], …]
  for-loops
  ```

- Syntax for Fortran
  ```
  !$omp do simd [clause[,, clause], …]
  do-loops
  ```
subroutine somme(A,B,n)
    implicit none
    integer :: n, i
    real (kind=8) :: A(n), B(n)
    real (kind=8) :: som
    som = 0.0
!$omp do simd reduction(+:som)
    do i=1,n
        som = som + A(i) * B(i)
    end do
print *, "som=", som
end subroutine somme
Function Vectorization new keyword

- `simdlen(length)` generate function to support a given vector length
- `uniform(argument-list)` argument has a constant value between the iterations of a given loop
- `inbranch` function always called from inside an `if` statement
- `notinbranch` function never called inside an `if` statement

Function Vectorization keyword

- `linear(argument-list[:linear-step])`
- `aligned(argument-list[:alignment])`
- `reduction(operator:list)`
Task Enhancements
Task Enhancements

- Intro to OpenMP
- Support for Compute Devices
- SIMD
- Task Enhancements
  - Introduction
  - Task Construct
- Thread Affinity
- Other improvements
- Demo
- Bibliography
Task Enhancements

New with OpenMP4

- Can abort parallel OpenMP execution by conditional cancellation at implicit and user-defined cancellation points
- Task can be grouped. Task groups can be aborted to reflect completion of cooperative tasking activities
- Task-to-task synchronization is supported through the specification of task dependency
Task Enhancements

The OpenMP Task Construct

- Each encountering thread/task creates a new task
  - Task can be nested into another task directive, into a worksharing construct

- Data clauses
  - `shared(list)`
  - `private(list)`, `firstprivate(list)`
  - `default(shared|none)`

```c
!$omp task [clause]
block
!$omp end task
```

```c
#pragma omp task [clause]
block
```

© Atos
Task Enhancements

Barrier and Taskwait

- **Openmp barrier**
  - All tasks created by any thread of the current *Team* are guaranteed to be completed at barrier exit

- **Task barrier: taskwait**
  - Encountering task is suspended until child tasks are complete. Applies only to direct childs, not descendants

- **taskyield** directive specifies that the current task can be suspended in favor of execution of a different task
Task Enhancements

Overview of New Directives and Functions for Tasks

- `omp cancel parallel|for|sections|taskgroup`
  - starts cancellation of all tasks in the same construct
- `omp cancelation point parallel|for|sections|taskgroup`
  - marks a point at which this task may be canceled
- `omp taskgroup`
  - marks a region such that all tasks started in it belong to a group
- `omp task depend([in|out|inout]: variable)` clause
  - marks that a task depends on other task
**Task Enhancements**

**depend**(type:list) clause on task construct

type is in, out or inout and list is a list of variables

▶ in the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an out or inout clause.

▶ out or inout : the generated task will be a dependent task of all previously generated sibling task that reference at least one of list items in in, out or inout clause.

```
program example
  integer :: x
  x = 1
!$omp parallel
!$omp single
!$omp task shared(x) depend(out: x)
  x = 2
!$omp end task
!$omp task shared(x) depend(in: x)
  print*, "x = ", x
!$omp end task
!$omp end single
!$omp end parallel
end program
```

“The program will always print ‘x = 2’, because the depend clauses enforce the ordering of the tasks. If the depend clauses had been omitted, then the tasks could execute in any order and the program would have a race condition.” [2]
Task Enhancements

Tasks in OpenMP: Data Scoping

- Some rules from *Parallel Regions* apply;
  - Static and global variables are shared
  - Automatic Storage (local) variables are private

- if shared scoping is not inherited:
  - Orphaned Task variables are *firstprivate* by defaults
  - Non-Orphanded Task variables inherit the *shared* attribute
  - Variables are *firstprivate* unless *shared* in the enclosing context

Taskgroup Construct

- `taskgroup` v.s. `taskwait`
  - `taskwait` only joins the child tasks
  - `taskgroup` joins the child and decedent tasks
Task Enhancements
Thread Affinity

Get Info on the System Topology

Before all, you should have a basic understanding of the system topology:

- Intel MPI’s `cpuinfo` tool
  - Delivers information about the number of sockets or packages and the mapping of processor ids used by the operating system to cpu cores
- OpenMPI’s `hwloc-ls` or `lstopo` tool
  - Display a graphical representation of the system topology, separated into NUMA nodes

Binding Strategy

- Putting threads far apart (i.e. on different sockets)
  - May improve the aggregated memory bandwidth available to your application
  - May improve the combined cache size available to your application
  - May decrease performance of synchronization constructs
- Putting threads close together (i.e. on two adjacent cores which possibly shared some caches)
  - May improve performance of synchronization constructs
  - May decrease the available memory bandwidth and cache size
Thread Affinity

Intel Compiler
Use environment variable **KMP_AFFINITY**
- **KMP_AFFINITY=scatter**: Put threads far apart
- **KMP_AFFINITY=compact**: Put threads close together
- **KMP_AFFINITY=<core_list>**: Bind threads in the order in which they are started to the cores given in the list, one thread per core
- Add **verbose** to print out binding information to stdout

Gnu Compiler
Use environment variable **GOMP_CPU_AFFINITY**
- **GOMP_CPU_AFFINITY=<core_list>**: Bind threads in the order in which they are started to the cores given in the list, one thread per core.
Thread Affinity

- Better support of affinity
- Can be used to get better locality, less false sharing and more memory bandwidth
- Define OpenMP Place: Variable OMP_PLACES to specify platform-specific data
  - set OpenMP threads running on one or more processors
  - can be defined by the user
  - pre-defined place:
    - threads: one place per hyper-thread
    - cores: one place exists per physical core
    - sockets: one place per processor package
- Define set of OpenMP thread affinity policies
  - New clause for omp parallel: proc_bind. Possibles values: false, true, master, close, spread
  - SPREAD: spread OpenMP threads evenly among places
  - CLOSE: pack OpenMP threads near master thread
  - MASTER: collocate OpenMP thread with master thread
  - OMP_PROC_BIND
- Goals: locality between OpenMP Threads/less false sharing/memory bandwidth
Other improvements

- You can define your own reductions (not yet with intel compiler)
  - Before version 4.0 OpenMP supported only reductions with base language operators and intrinsic procedures. With OpenMP 4.0 API end-user reductions are now supported `omp declare reduction`

- Sequentially consistent atomics
  - The use of the `seq_cst` clause forces the atomically performed operation to include an implicit flush operation for all variables
    - `omp atomic [read|write|update|capture] [ seq_cst]`
    - `omp atomic [capture] [ seq_cst]`

- Optional dump all internal variables at program start
  - `OMP_DISPLAY_ENV=TRUE|FALSE|VERBOSE`
Other improvements
© Bull. All rights reserved

- Users Restricted Rights - Use, duplication or disclosure restricted.
- Any copy of these documents should keep all copyright, logos and other proprietary notices contained herein.
- This publication may include technical inaccuracies or typographical errors.
- This publication is provided "AS IS" without any warranty either expressed or implied including but not limited to the implied warranties of merchantabilities or fitness of the described product.
- Course Material Licensing Terms : No sublicensing rights.
- For other licensing needs, please contact Bull
Bibliography

- OpenMP.
  *OpenMP Application Program Interface*, version 4.0 édition, 2013.

- OpenMP Architecture Review Board.
  Version 4.0.1.

- James Reinders et Jim Jeffers, éditeurs.
  *High Performance Parallelism Pearls*, chapitre Dynamic load balancing OpenMP4.0.
  MK, 2014.

- SC2013, éditeur.