OpenMP for Exascale Computing

Michael Wong
michaelw@ca.ibm.com
OpenMP CEO
Chair of WG21 SG5 Transactional Memory
ISO C+ +Standard, Head of Delegation for Canada and IBM
Acknowledgement and Disclaimer

• Numerous people internal and external to the OpenMP WG, in industry and academia, have made contributions, influenced ideas, written part of this presentations, and offered feedbacks to form part of this talk.
• I even lifted this acknowledgement and disclaimer from some of them.
• But I claim all credit for errors, and stupid mistakes. These are mine, all mine!
• Any opinions expressed in this presentation are my opinions and do not necessarily reflect the opinions of IBM.
Agenda

• OpenMP ARB Corporation
• A few features in 4.0 for exascale programming model
• Accelerators
  – OpenMP and OpenACC
• Affinity
• Tools and Debugging
• The future of OpenMP
• IWOMP 2014 and OpenMPCon 2014
OpenMP 4.0: A Significant Paradigm Shift in Parallel Programming

OpenMP 4.0 Specifications Released

The OpenMP 4.0 API Specification is released with Significant New Standard Features

The OpenMP 4.0 API supports the programming of accelerators, SIMD programming, and better optimization using thread affinity.

The OpenMP Consortium has released OpenMP API 4.0, a major upgrade of the OpenMP API standard language specifications. Besides several major enhancements, this release provides a new mechanism to describe regions of code where data and/or computation should be moved to another computing device.

Bronis R. de Supinski, Chair of the OpenMP Language Committee, stated that “OpenMP 4.0 API is a major advance that adds two new forms of parallelism in the form of device constructs and SIMD constructs. It also includes several significant extensions for the loop-based and task-based forms of parallelism already supported in the OpenMP 3.1 API.”

The 4.0 specification is now available on the OpenMP Specifications page.

Standard for parallel programming extends its reach

With this release, the OpenMP API specifications, the de-facto standard for parallel programming on shared memory systems, continues to extend its reach beyond pure HPC to include DSPs, real time systems, and accelerators. The OpenMP API aims to provide high-level parallel language support for a wide range of applications, from automotive and aeronautics to biotech, automation, robotics and financial analysis.

New features in the OpenMP 4.0 API include:

- Support for accelerators. The OpenMP 4.0 API specification effort included significant participation by all the major vendors in order to support a wide variety of compute devices. OpenMP API provides mechanisms to describe regions of code where data and/or computation should be moved to another computing device. Several prototypes for the accelerator proposal have already been implemented.

- SIMD constructs to vectorize both serial as well as parallelized loops. With the advent of SIMD units in all major processor chips, portable support for accessing them is essential. OpenMP 4.0 API provides mechanisms to describe when multiple iterations of the loop can be executed concurrently using SIMD instructions and to describe how to create versions of functions that can be invoked across SIMD lanes.
A brief history of OpenMP API

2014 onwards, more agile

Next OpenMP revision cycle: faster, more predictable
Less monolithic: Delivering concurrent TRs & language extensions
OpenMP is a living language
OpenMP Members growth

- From Dieter An Mey, RWTH Aachen 2012, since 2012 added
  - Red Hat/GCC
  - Barcelona SuperComputing Centre
  - University of Houston

26 members and growing
OpenMP internal Organization

OpenMP ARB

Language WG
Marketing WG

Today
Accel
Error
Task
Tools
Affinity
Fortran 2003
Memory
Model,
Loops,
Object
oriented

Future
TM
Async/Event
Interop
C++11
C11
The New Mission Statement of OpenMP

- OpenMP’s new mission statement
  - “Standardize directive-based multi-language high-level parallelism that is performant, productive and portable”
  - Updated from
    - "Standardize and unify shared memory, thread-level parallelism for HPC"
Agenda

• OpenMP ARB Corporation
• **A few features in 4.0 for exascale programming model**
• Accelerators
  – OpenMP and OpenACC
• Affinity
• Tools and Debugging
• The future of OpenMP
• IWOMP 2014 and OpenMPCon 2014
Goals

- Thread-rich computing environments are becoming more prevalent
  - more computing power, more threads
  - less memory relative to compute
- There is parallelism, it comes in many forms
  - hybrid MPI - OpenMP parallelism
  - mixed mode OpenMP / Pthread parallelism
  - nested OpenMP parallelism
- Have to exploit parallelism efficiently
  - providing ease of use for casual programmers
  - providing full control for power programmers
  - providing timing feedback
Impact of Overhead in Prevalent Threading Model

Programming Model

• MPI
  – distributed process across/within nodes
  – explicit user-managed communication

• Coarse-grain Parallel (OpenMP/Auto)
  – shared memory within nodes/cores
  – for outer parallel-loops

• Fine-grain Parallel (OpenMP/Auto)
  – shared memory within cores/nodes
  – for inner parallel-loops
OpenMP 4.0 features for Exascale

- Heterogeneous architecture: CPU with GPU, GPGPU, APU, DSP, co-processors, embedded processors
  - Address heterogeneity challenge of exascale: computing using accelerators
  - Exascale machines: lower energy, higher abstraction
- Improve locality: Handle nested parallelism: more control with thread affinity
  - More user input on how to map computation to threads
    - Currently: no affinity support provided by user
  - A new thread-affinity to OpenMP standard committee
  - Contributed reference implementation in research runtime
- Improve Tools interface: Provide timing feedback, Debugging API
  - User want to know where is the time spent
  - But with little or no overheads
  - Have a uniform way of debugging
Agenda

• OpenMP ARB Corporation
• A few features in 4.0 for exascale programming model
• **Accelerators**
  – OpenMP and OpenACC
• Affinity
• Tools and Debugging
• The future of OpenMP
• IWOMP 2014 and OpenMPCon 2014
OpenMP Accelerator Subcommittee

• Co-chairs Technical leads
  – Jame Beyers- Cray
  – Eric Stotzer - TI

• Active subcommittee members
  – Xinmin Tian – Intel
  – Ravi Narayanaswamy – Intel
  – Jeff Larkin – Nvidia
  – Kent Milfeld – TACC
  – Henry Jin – NASA
  – Kevin O’Brien – IBM
  – Alexandre Eichenberger, IBM
  – Christian Terboven– RWTH Aachen (courtesy for slides)
  – Michael Klemm – Intel (courtesy for slides)
  – Stephane Cheveau – CAPS
  – Convey, AMD, ORNL, TU Dresden,
Device Model

- One host
- Multiple accelerators/coprocessors
Glossary

- **Device**: an implementation-defined (logical) execution unit
- **League**: the set of threads teams created by a teams construct
- **Contention group**: threads of a team in a league and their descendant threads
- **Device data environment**: Data environment as defined by target data or target constructs
- **Mapped variable**: An original variable in a (host) data environment with a corresponding variable in a device data environment
- **Mapable type**: A type that is amenable for mapped variables. (Bitwise copyable plus additional restrictions.)
Transfer control from the host to the device

Syntax (C/C++)

```c
#pragma omp target [clause[,] clause],...
structured-block
```

Syntax (Fortran)

```fortran
!$omp target [clause[,] clause],...
structured-block
```

Clauses

- `device(scalar-integer-expression)`
- `map(alloc | to | from | tofrom: list)`
- `if(scalar-expr)`
Create a device data environment

Syntax (C/C++)
#pragma omp target data [clause[[], clause],...]
structured-block

Syntax (Fortran)
!$omp target data [clause[[], clause],...]
structured-block

Clauses
device(scalar-integer-expression)
map(alloc | to | from | tofrom: list)
if(scalar-expr)
target update Construct

- Issue data transfers between host and devices
- Syntax (C/C++)
  #pragma omp target update [clause[[[,] clause],...]
- Syntax (Fortran)
  !$omp target data update [clause[[[,] clause],...]
- Clauses
  device(scalar-integer-expression)
  to(list)
  from(list)
  if(scalar-expr)
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

- 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

- Use **target update** to request data transfers from within a target data region
Execution Model and Data Environment

- Data environment is lexically scoped
  - Data environment is destroyed at closing curly brace
  - Allocated buffers/data are automatically released

Host

Device

```
#pragma omp target \map(aloc: ...) \map(to: ...) \map(from: ...)
{ ... }
```
Offloading Computation

- Use target construct to
  - Transfer control from the host to the device
  - Establish a device data environment (if not yet done)
- Host thread waits until offloaded region completed
  - Use other OpenMP constructs for asynchronicity

```c
#pragma omp target map(to:b[0:count])) map(to:c,d) map(from:a[0:count])
{
  #pragma omp parallel for
  for (i=0; i<count; i++) {
    a[i] = b[i] * c + d;
  }
}
```
Data Environments

- Create a data environment to keep data on devices
  - Avoid frequent transfers or overlap computation/comm.
  - Pre-allocate temporary fields

```c
#pragma omp target data device(0) map(alloc:tmp[:N]) map(to:input[:N]) map(from:res)
{
#pragma omp target device(0)
#pragma omp parallel for
  for (i=0; i<N; i++)
    tmp[i] = some_computation(input[i], i);

  do_somes_other_stuff_on_host();

#pragma omp target device(0)
#pragma omp parallel for reduction(+:res)
  for (i=0; i<N; i++)
    res += final_computation(tmp[i], i)
}
```
declare target Construct

- Declare one or more functions to also be compiled for the target device

- Syntax (C/C++):
  
  ```
  #pragma omp declare target
  [function-definitions-or-declarations]
  #pragma omp end declare target
  ```

- Syntax (Fortran):
  
  ```
  !$omp declare target [(proc-name-list | list)]
  ```
Host and device functions

- The tagged functions will be compiled for
  - Host execution (as usual)
  - Target execution (to be invoked from offloaded code)

```c
#pragma omp declare target
float some_computation(float fl, int in) {
    // ... code ...
}

float final_computation(float fl, int in) {
    // ... code ...
}
#pragma omp end declare target
```

```
some_computation:
    ...
    movups %xmm2, (%rbx)
    movups %xmm3, (%rbx)
    ...
final_computation:
    ...
```

```
some_computation_device:
    ...
    vprefetch0 64(%r15)
    vaddps %zmm7, %zmm6, %zmm9
    ...
final_computation_device:
    ...
```
#pragma omp target data device(0) map(allocreg[tmp[:N]]) map(to:input[:N]) map(from:res)
{
  #pragma omp target device(0)
  #pragma omp parallel for
  for (i=0; i<N; i++)
  {
    tmp[i] = some_computation(input[i], i);

    update_input_array_on_the_host(input);
  }

  #pragma omp target update device(0) to(input[:N])

  #pragma omp target device(0)
  #pragma omp parallel for reduction(+:res)
  for (i=0; i<N; i++)
  {
    res += final_computation(input[i], tmp[i], i)
  }
Asynchronous Offloading

- Use existing OpenMP features to implement asynchronous offloads.

```c
#pragma omp parallel sections
{
#pragma omp task
{
#pragma omp target map(to:input[:N]) map(from:result[:N])
#pragma omp parallel for
   for (i=0; i<N; i++) {
       result[i] = some_computation(input[i], i);
   }
}
#pragma omp task
{
   do_something_important_on_host();
}
#pragma omp taskwait
}```
teams Construct

- Syntax (C/C++):
  
  
  ```
  #pragma omp teams [clause[[], clause],...]
  structured-block
  ```

- Syntax (Fortran):
  
  ```
  !$omp teams [clause[[], clause],...] structured-block
  ```

- Clauses
  
  ```
  num_teams(integer-expression)
  num_threads(integer-expression)
  default(shared | none)
  private(list), firstprivate(list)
  shared(list), reduction(operator : list)
  ```
Restrictions on 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` (see next slides)
  - `parallel`
  - `parallel for` (C/C++), `parallel do` (Fortran)
  - `parallel sections`
int main(int argc, const char* argv[]) {
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Define scalars n, a, b & initialize x, y

    for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
    }

    free(x); free(y); return 0;
}
int main(int argc, const char* argv[]) {
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Define scalars n, a, b & initialize x, y

#pragma omp target data map(to:x[0:n])
{

    for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
    }
}
free(x); free(y); return 0;
}
```c
int main(int argc, const char* argv[]) {
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Define scalars n, a, b & initialize x, y

#pragma omp target data map(to:x[0:n])
    {
#pragma omp target map(tofrom:y)
#pragma omp teams num_teams(num_blocks) num_threads(nthreads)

    for (int i = 0; i < n; i += num_blocks) {
        for (int j = i; j < i + num_blocks; j++) {
            y[j] = a*x[j] + y[j];
        }
    }
    free(x); free(y); return 0;
}
```
distribute Construct

■ Syntax (C/C++):
  #pragma omp distribute [clause[[,] clause],...]
  for-loops

■ Syntax (Fortran):
  !$omp teams [clause[[,] clause],...]
  do-loops

■ Clauses
  private(list)
  firstprivate(list)
  collapse(n)
  dist_schedule(kind[, chunk_size])
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 the construct

- `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
int main(int argc, const char* argv[]) {
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Define scalars n, a, b & initialize x, y

#pragma omp target data map(to:x[0:n])
{
#pragma omp target map(tofrom:y)
#pragma omp teams num teams(num_blocks) num_threads(bsize)
    all do the same
#pragma omp distribute
    for (int i = 0; i < n; i += num_blocks){
        workshare (w/o barrier)
#pragma omp parallel for
    for (int i = i; i < i + num_blocks; i++) {
        workshare (w/ barrier)
            y[j] = a*x[j] + y[j];
    }
} } 
free(x); free(y); return 0; }
Combined Constructs

- The distribution patterns can be cumbersome

- OpenMP 4.0 defines combined constructs for typical code patterns
  - `distribute simd`
  - `distribute parallel for` (C/C++)
  - `distribute parallel for simd` (C/C++)
  - `distribute parallel do` (Fortran)
  - `distribute parallel do simd` (Fortran)
  - ... plus additional combinations for `teams` and `target`

- Avoids the need to do manual loop blocking
int main(int argc, const char* argv[]) {
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Define scalars n, a, b & initialize x, y

#pragma omp target map(to:x[0:n]) map(tofrom:y)
{
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
#pragma omp distribute parallel for
    for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
    }
}

    free(x); free(y); return 0;
}
int main(int argc, const char* argv[]) {
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Define scalars n, a, b & initialize x, y

    #pragma omp target map(to:x[0:n]) map(tofrom:y)
    {
        #pragma omp teams distribute parallel for \
          num_teams(num_blocks) num_threads(bsize)
          for (int i = 0; i < n; ++i){
              y[i] = a*x[i] + y[i];
          }
    }

    free(x); free(y); return 0;
}
Additional Runtime Support

- **Runtime support routines**
  - `void omp_set_default_device(int dev_num)`
  - `int omp_get_default_device(void)`
  - `int omp_get_num_devices(void);`
  - `int omp_get_num_teams(void)`
  - `int omp_get_team_num(void);`

- **Environment variable**
  - Control default device through `OMP_DEFAULT_DEVICE`
  - Accepts a non-negative integer value
int num_dev = omp_get_num_devices();
int chunksz = length / num_dev;
assert((length % num_dev) == 0);
#pragma omp parallel sections firstprivate(chunksz,num_dev)
{
    for (int dev = 0; dev < NUM_DEVICES; dev++) {
#pragma omp task firstprivate(dev)
        {
            int lb = dev * chunksz;
            int ub = (dev+1) * chunksz;
#pragma omp target device(dev) map(in:y[lb:chunksz]) map(out:x[lb:chunksz])
            {
#pragma omp parallel for
                for (int i = lb; i < ub; i++) {
                    x[i] = a * y[i];
                }
            }
        }
    }
}
OpenACC 1 compared to OpenMP 4.0 (by Dr. James Beyer)

**OpenACC 1**
- Parallel (offload)
  - Parallel (multiple “threads”)
- Kernels
- Data
- Loop
- Host data
- Cache
- Update
- Wait
- Declare

**OpenMP 4.0**
- Target
- Team/Parallel
- Target Data
- Distribute/Do/for/Simd
- Target Update
- Declare Target

Slide 41
Future OpenACC vs future OpenMP
(by Dr. James Beyer)

OpenACC2
• enter data
• exit data
• data api
• routine
• async wait
• parallel in parallel
• tile
• Linkable
• Device_type

OpenMP future
• Unstructured data environment
• declare target
• Parallel in parallel or team
• tile
• Linkable or Deferred_map
• Device_type
Agenda

• OpenMP ARB Corporation
• A few features in 4.0 for exascale programming model
• Accelerators
  – OpenMP and OpenACC
• Affinity
• Tools and Debugging
• The future of OpenMP
• IWOMP 2014 and OpenMPCon 2014
OpenMP Affinity Subcommittee

• Co-chairs Technical leads
  – Alexandre Eichenberger, IBM (courtesy for slides)
  – Christian Terboven – RWTH Aachen (courtesy for slides)

• Active subcommittee members
  – Dieter An Mey – RWTH Aachen
  – Michael Wong, IBM
  – Nowal Copty – Oracle
  – Kent Milfeld - TACC
Current Support for Thread Affinity

- All implementations use the core id naming scheme provided by the operating system.

- **Intel compilers:** `KMP_AFFINITY` environment variable
  - Enumerations of single core ids in any order, ranges of core ids, sets of core ids with and without a stride → threads may be bound to specific cores or a subset of the machine
  - Specifiers `scatter` and `compact` → thread distribution with given strategy

- **GNU compilers:** `GOMP_CPU_AFFINITY` environment variable
  - Enumerations of single core ids in any order, ranges of core ids with and without a stride → threads may be bound to specific cores or a subset of the machine.
Current Support for Thread Affinity

- **Oracle compilers**: `SUNW_MP_PROCBIND` environment variable
  - yes → automatic binding of threads to cores
  - Enumerations of single core ids in any order → threads may be bound to specific cores

- **PGI compilers**: `MP_BIND` and `MP_BLIST` environment variables
  - Binding has to be enabled explicitly
  - Enumerations of single core ids in any → threads are bound one-by-one in a round-robin manner.
Current Support for Thread Affinity

- **IBM compilers:** \texttt{XLSMPOPTS} environment variable
  
  → Definition of a \textit{system level detail}: \texttt{PROC, PROC\_CORE, L2CACHE or MCM} → granularity of thread affinity can be set

  → Either enumeration or list of resources → threads may be bound to specific resources or a subset of the machine

- Several compilers+runtimes offer flexible and powerful ways to bind threads to cores. However they all
  
  → lack support for Nested OpenMP

  → have no or very limited support to dynamically change the binding after the program has been started

  → offer similar behavior with different syntax → opportunity of standardization
Goals with Thread Affinity

- User often knows what he/she wants at a high level
  - get threads on separate cores to get more L1/L2 caches
  - get threads collocated on same core to maximize cache reuse

- Current runtimes have a fixed policy
  - runtime tries to even out load balance across the machine
  - this works well for single level of parallelism,
  - not as well for nested parallelism

- Want to allow users to specify where to get threads
  - broad policies that cover most cases

- Want to allow users to specify where threads are allowed to migrate
  - for load balancing purpose
Pre-thread binding Strategy

- Should have a basic understanding of the system topology
  - Intel MPI’s cpuinfo tool
    - Number of sockets (packages) and the mapping of OS processor ids to system cores
  - OpenMP-MPI Hwloc’s hwloc-ls or lstopo tool
    - Graphical representation of system topology, separated into NUMA nodes, mapping of ids used by the OS to CPU cores as well as caches

- Select the right binding strategy based on topology and the characteristics of application
  - Scatter threads far apart on different sockets
    - May improve aggregated memory bandwidth available
    - May improve combined cache size available
    - May decrease synchronization construct performance
  - Gather threads close together say on two adjacent cores that may share cache
    - May improve synchronization construct performance
    - May decrease memory bandwidth and cache size available
  - Or just use trial and error and select the best fit
Summary

- Define OpenMP Places
  - set of OpenMP threads running on one or more processors
  - can be defined by the user

- Define a set of OpenMP thread affinity policies
  - affinity defines which OpenMP threads are used in a parallel construct
    - SPREAD: spread OpenMP threads evenly among the places
    - CLOSE: pack OpenMP threads near master thread
    - MASTER: collocate OpenMP thread with master thread

- Goals
  - user has a way to specify where to execute its OpenMP threads for
    - better locality between OpenMP threads
    - less false sharing
    - more memory bandwidth
Places

- OpenMP runtime only deals with (abstract) places
  - there are one or more processors / hardware threads per place
  - runtime may move OpenMP threads within a place
  - runtime should not move OpenMP threads between places

- When executing a parallel region construct
  - affinity directives can impact which OpenMP threads are selected
  - i.e. on which place to pick OpenMP threads

- User is not required to give a list of places
  - there is a vendor-specific default list of places
  - when OpenMP is used in hybrid mode (e.g. MPI)
    - MPI-RUN script can specify a rank-specific list of places per MPI process

- User can specify a list of places
  - using high-level names (e.g. threads, cores, sockets) or low-level (numbers)
Defining Affinity Policies

- Affinity can be set on each individual parallel construct
  - “omp parallel proc_bind(master | close | spread)…”
  - affinity for that specific parallel region

- Default affinity can be set by an environment variable
  - OMP_PROC_BIND=‘spread, close’
  - defines default policy where first level-parallel constructs use spread, second-level (nested) parallel construct use close
  - values:
    - false: disabled, runtime may move thread among places
    - true: enabled, runtime should not move threads among places
    - list of master, compact, spread
OMP PLACES

- **Definition**
  - abstract name or ordered list of one or more places to be used by OpenMP
  - each place consists of a list of one or more resources
  - meaning of the abstract name or resource numbers is implementation dependent

- **Syntax**

```
"OMP PLACES"  = <abstract-name> | <place-list>
<abstract-name>  = <word> [ "(" <pnum> ")" ] | ... | <word> [ "(" <pnum> ")" ]
<place-list>  = <place-interval> "," <place-list> | <place-interval>
<place-interval>  = <place> ":" <pnum> ":" <num> | <place> ":" <pnum> | <place> | "!" <place>
<place>  = "{" <resource-list> ""}
<resource-list>  = <resource-interval> "," <resource-list> | <resource-interval>
<resource-interval>  = <resource> ":" <pnum> ":" <num> | <resource> ":" <pnum> | <resource> | "!" <resource>
<resource>  = non-negative integer
<num>  = integer
<pnum>  = positive integer
<word>  = implementation dependent name, possibly qualified with a "(" <pnum> ")"
How to use Place Lists

• Consider a system with 2 chips, 4 cores, and 8 hardware-threads

  – One place per hardware-thread
    • OMP_PLACES="{0},{1},{2},…{15}"
    • OMP_PLACES="threads(16)" # 16 threads
    • OMP_PLACES="threads" # as many threads as available

  – One place per core, including both hardware-threads
    • OMP_PLACES="{0,1},{2,3},{4,5},{6,7},{8,9},{10,11},{12,13},{14,15}"
    • OMP_PLACES="{0:2}:8:2"
    • OMP_PLACES="cores(4)" # 4 cores
    • OMP_PLACES="cores" # as many cores as available

  – One place per chip, excluding one hardware-thread per chip
    • OMP_PLACES="{1:7},{9:7}" # exclude hardware thread 0 and 8
MASTER Affinity

- For best data locality
  - select OpenMP threads in the same place as the master

- Examples of “omp parallel proc_bind(master)”
  - master 2*
  - master 4

* technically: omp parallel num_threads(2) proc_bind(master)
CLOSE Affinity

- For data locality, load-balancing, and more dedicated-resources
  - select OpenMP threads near the place of the master
  - wrap around once each place has received one OpenMP thread

- Examples of “omp parallel proc_bind(close)”
  - compact 2*
  - compact 4
  - compact 16

* technically: omp parallel num_threads(2) proc_bind(compact)
SPREAD Affinity

- For load balancing, most dedicated hardware resources
  - spread OpenMP threads as evenly as possible among places
  - create sub-partition of the place list
    - subsequent threads will only be allocated within sub-partition
- Examples of “omp parallel proc_bind(spread)”
  - spread 2
  - spread 4
  - spread 8
  - spread 16
Objective:
- separate cores for outer-loop,
- same cores for inner-loops (avoid false sharing)

Spread with nested compact
- spread create sub-partition
- same place work within respective partition

Examples
- initial
- spread 4
- master 4
Nested Parallel with SPREAD-CLOSE

- Objective:
  - separate cores for outer-loop,
  - near cores for inner-loops (still distributing threads for load balancing)

- Spread with nested compact
  - spread create sub-partition
  - compact work within respective partition

- Examples
  - initial
  - spread 4
  - compact 4
**Nested Parallelism SPREAD-SPREAD-CLOSE**

- **Objective:** multiple-levels of nested parallelism
- **Spread with nested compact**
  - spread create sub-partition
  - compact work within respective partition

**Examples**
- **Initial**
- **spread 2**
- **spread 4**
- **compact 4**
Asking for More OpenMP Threads than Available

• Asking for more threads than available:
  – determining “if threads are available or not” is implementation defined
  – when not enough available, providing threads is implementation-defined
  – possible policies:
    • provide fewer OpenMP threads than requested
    • break sub-partition to provide OpenMP threads from other partitions
  – some implementation may always have threads available
    • provided it respects the OMP_THREAD_LIMIT upper limit

• When we may not have enough available threads:
  – affinity MASTER asks for more than can be provided in its place
  – affinity CLOSE/SPREAD asks for more than can be provided in its partition
  – may occur even when OMP_THREAD_LIMIT threads have not yet been used
Assignment of Thread Numbers

- Left to right, same order as places in list

In the above example:
- consider a user requesting 16 threads with proc_bind(spread) in a system with 8 places.
- this user will know exactly which thread (as identified with thread_num) executes on which places with which other thread.
Summary

- Give the user more fine-grain control
  - which hardware thread / core / chip to use
  - which thread to select for a given parallel region
    - e.g. spread vs. close
  - where threads are allowed to migrate (within a place)

- Proposal should satisfy different users:
  - users wanting full control can have it
  - users wanting to give general hints don’t have to go through too much details

- Ongoing work
  - implemented in our research OpenMP runtime (avail to LLNL/ANL/Juelich)
  - part of OpenMP 4.0
Agenda

• OpenMP ARB Corporation
• A few features in 4.0 for exascale programming model
• Accelerators
  – OpenMP and OpenACC
• Affinity
• **Tools and Debugging**
• The future of OpenMP
• IWOMP 2014 and OpenMPCon 2014
OpenMP Tools Subcommittee

- Executive lead
  - Martin Schulz - LLNL

- Technical leads
  - Alexandre Eichenberger - IBM
  - John Mellor-Crummey – Rice (courtesy for slides)

- Active subcommittee members
  - Nawal Copty - Oracle
  - Robert Dietrich - TU Dresden
  - Xu Liu – Rice (courtesy for slides)
  - Eugene Loh - Oracle
  - Daniel Lorenz - Juelich
OpenMP’s fork-join parallelism
Challenges in programming with OpenMP

- Performance problems in OpenMP programs
  - insufficient parallelism
  - serialization
  - load imbalance

Performance tools are needed to understand inefficiencies
Previous work

- Instrumentation-based tools: TAU, Scalasca
  - high measurement overhead using POMP API for OpenMP
- Sampling-based tools: Intel Vtune
  - no specific OpenMP support
- Hybrid tools: Solaris Studio/Oracle Collector API for OpenMP
  - support low-overhead, sampling-based measurement
  - insufficient support for statically-linked applications
  - insufficient mechanisms to blame root causes of performance losses
    - attributes waiting to “symptoms” rather than “causes”

Goal: build a tool that avoids the shortcomings of existing tools
OMPT OpenMP tools API status

Implementations

- GNU OpenMP runtime: early prototype
- IBM lightweight OpenMP runtime: a full implementation
- Open-source Intel OpenMP runtime: emerging full implementation
Without OMPT: separate views for different threads

Worker threads don’t know the full user-level context for work
Call stack snapshot in OpenMP threads

- Regions in gray have distributed calling contexts.
- Problem: monitoring tiny regions can be expensive.

Online deferred context resolution.
Results of deferred context construction

```
#include <omp.h>

#define N 10

int main()
{
    omp_set_nested(1);
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2)
    {
        fib(N+3);
        report_num_threads(omp_get_level());
        #pragma omp parallel num_threads(2)
        {
            fib(N+3);
            report_num_threads(omp_get_level());
        }
        fib(N+3);
    }
    fib(N+3);
}
```
Problem: meaningless hotspots

hotspot is `do_wait`, but don’t know why
Undirected blame

- *do_wait* is the symptom
- Causes are working threads
  - e.g. some threads have more work than other threads
- Blaming *do_wait* to working threads

```
F
ork

  do_wait

  do_wait

Join

  2 idle threads
  3 working threads

each working thread blames itself for 2/3 of a sample of idleness
```
Directed blame

- Thread waiting at lock is the symptom
- Cause is the lock holder
- Blame lock waiting to lock holder

![Diagram showing directed blame process]

- Acquire lock
- Lock wait
- Release lock

Locked samples indexed by lock address
Lock holder accepts these samples at lock release
Example: Directed Blame Shifting for Locks

Blame a lock holder for delaying waiting threads

- Charge all samples that threads receive while awaiting a lock to the lock itself
- When releasing a lock, accept blame at the lock

almost all blame for the waiting is attributed here (cause)

all of the waiting occurs here (symptom)
Outline

• OMPT - emerging performance tool API for OpenMP
  – overview and goals
  – state tracking
  – event notification
  – API

• Next steps
OMPT Design Objectives

• Enable tools to gather information and associate costs with application source and runtime system
  – provide an interface sufficient to construct low-overhead performance tools based on asynchronous sampling
  – enable a profiler that uses call stack unwinding to identify which frames in its callstack correspond to routines in OpenMP runtime
  – associate activity of a thread at any point in time with a state
    • enable performance tools to monitor behavior

• Negligible overhead if OMPT interface is not in use

• Define support for trace-based performance tools

• Don’t impose an unreasonable development burden on
  – runtime implementers
  – tool developers
OMPT Performance Tools API

Overview and Goals

• Create a standardized performance tool interface for OpenMP
  – prerequisite for portable performance tools
  – goal: inclusion in the OpenMP standard
  – role model: PMPI and MPI_T

• Focus on minimal set of functionality
  – provide essential support for sampling-based tools
  – only require support for tools attached at link-time or program launch

• Minimize runtime cost
  – reduce cost in runtime and tool where possible
  – enable integration into optimized runtimes
  – make support for higher-overhead features optional
    • callbacks for blame shifting
    • callbacks for full-featured tracing tools
Major OMPT Functionality

Draft standard interface for performance tools

• State tracking
  – have runtime track keep track of thread states (e.g., idle, parallel)
  – allow tools to query this state at any time (async signal safe)
  – provide (limited) persistent storage for tool data in runtime system

• Call stack interpretation
  – provide hooks to enable recovery of complete calling context
    for computations in worker threads
    • hooks to support reconstruction of application-level call stacks
  – support identification of OpenMP runtime stack frames

• Event notification
  – provide callback mechanism for predefined events
  – support a few mandatory notifications and many optional ones
Runtime State Tracking

- OpenMP runtime keeps track of its own state
  - predefined states on next slide
- Query routine
  - `ompt_state_t ompt_get_state(ompt_wait_id_t *wait_id)`
  - routine must be async signal safe
- Wait IDs
  - only returned for states that signify waiting
  - identifies the cause for waiting
    - e.g., address of a user lock or implicit lock for a critical region/atomic
### Predefined States

```c
/* work states (0..15) */
ompt_state_work_serial = 0x00, /* serial work */
ompt_state_work_parallel = 0x01, /* parallel work */
ompt_state_work_reduction = 0x02, /* performing a reduction */

/* idle (16..31) */
ompt_state_idle = 0x10, /* waiting for work */

/* overhead states (32..63) */
ompt_state_overhead = 0x20, /* non-wait overhead */

/* barrier wait states (64..79) */
ompt_state_wait_barrier = 0x40, /* waiting at any barrier */
ompt_state_wait_explicit_barrier = 0x41, /* waiting at an explicit barrier */

/* task wait states (80..95) */
ompt_state_wait_taskwait = 0x50, /* waiting at a taskwait */
ompt_state_wait_taskgroup = 0x51, /* waiting at a taskgroup */

/* wait states mutex (96..111) */
ompt_state_wait_lock = 0x60, /* waiting for lock */
ompt_state_wait_nest_lock = 0x61, /* waiting for nest lock */
ompt_state_wait_critical = 0x62, /* waiting for critical */
ompt_state_wait_atomic = 0x63, /* waiting for atomic */
ompt_state_wait_ordered = 0x64, /* waiting for ordered */

/* miscellaneous (112..127) */
ompt_state_undefined = 0x70, /* undefined thread state */
ompt_state_first = 0x71, /* initial enumeration state */
```
OMPT Event Notifications

- Mandatory events
- Blame-shifting events (optional)
- Trace events (optional)
## Mandatory Events

**Essential support for any performance tool**

<table>
<thead>
<tr>
<th>Events</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>create/exit event pairs</td>
</tr>
<tr>
<td>Parallel regions</td>
<td></td>
</tr>
<tr>
<td>Tasks</td>
<td>singleton events</td>
</tr>
<tr>
<td>Runtime shutdown</td>
<td></td>
</tr>
<tr>
<td>User-level control API</td>
<td>e.g., support tool start/stop</td>
</tr>
</tbody>
</table>

**Thread Events**

- Start/stop event pair
- Creation/exit event pair
- Singleton event

**Parallel Region Events**

- Start/stop event pair
- Creation/exit event pair
- Singleton event

**Task Events**

- Start/stop event pair
- Creation/exit event pair
- Singleton event
Blame-shifting Events (Optional)

Support designed for sampling-based performance tools

- **Idle**
- **Wait**
  - barrier
  - taskwait
  - taskgroup wait

• begin/end event pairs


- **Release**
  - lock
  - nest lock
  - critical
  - atomic
  - ordered section

• singleton events
| ompt_event_implicit_task_create | ompt_event_taskgroup_end |
| ompt_event_implicit_task_exit   | ompt_event_release_nest_lock_prev |
| ompt_event_task_switch          | ompt_event_wait_lock          |
| ompt_event_loop_begin           | ompt_event_wait_nest_lock     |
| ompt_event_loop_end             | ompt_event_wait_critical     |
| ompt_event_section_begin        | ompt_event_wait_atomic       |
| ompt_event_section_end          | ompt_event_wait_ordered      |
| ompt_event_single_in_block_begin| ompt_event_acquired_lock     |
| ompt_event_single_in_block_end  | ompt_event_acquired_nest_lock_first |
| ompt_event_single_others_begin  | ompt_event_acquired_nest_lock_next |
| ompt_event_single_others_end    | ompt_event_acquired_critical |
| ompt_event_master_begin         | ompt_event_acquired_atomic   |
| ompt_event_master_end           | ompt_event_acquired_ordered  |
| ompt_event_barrier_begin        | ompt_event_init_lock         |
| ompt_event_barrier_end          | ompt_event_init_nest_lock    |
| ompt_event_taskwait_begin       | ompt_event_destroy_lock      |
| ompt_event_taskwait_end         | ompt_event_destroy_nest_lock |
| ompt_event_taskgroup_begin      | ompt_event_flush             |
Parallel Region and Task IDs

• Each parallel region instance has a unique ID
  – region IDs are not required to be consecutive
• Ability to query parallel region IDs
  – `ompt_parallel_id_t ompt_get_parallel_id(int ancestor_level)`
  – async signal safe
  – current region: ancestor_level = 0
  – query IDs of ancestor regions using higher ancestor levels
• Each task instance has a unique ID
  – task IDs are not required to be consecutive
  – both explicit and implicit tasks receive an ID
• Ability to query task IDs
  – `ompt_task_id_t ompt_get_task_id(int ancestor_level)`
  – async signal safe
  – current task: ancestor_level = 0
  – query IDs of ancestor task/region using higher ancestor levels
Call Stack Interpretation

- Tool saves some frame information to support stack unwinding

```c
typedef struct ompt_frame_t {
    void *reenter_runtime_frame;
    void *exit_runtime_frame;
} ompt_frame_t;
```

- per task; lifetime: duration of task
- `ompt_frame_t *ompt_get_task_frame(int ancestor_level)`
- async signal safe

- Reenter_runtime_frame
  - set each time a current task enters the runtime to create a new task
  - points to the stack above the return address of the last user frame

- Exit_runtime_frame
  - set when a task exits the runtime to execute user code
  - points to the stack above the return address of the last runtime frame
Call Stack Interpretation Example
Miscellaneous API Features

• Tool-facing API functions
  – initialization
    • int ompt_initialize(ompt_function_lookup_t ompt_fn_lookup)
    • int ompt_set_callback(ompt_event_t e, ompt_callback_t cb)
  – tool support version inquiry
    • int ompt_get_ompt_version(void)
  – state enumeration
    • int ompt Enumerate_state(int current_state, int *next_state, const char **next_state_name)

• User-facing API functions
  – version inquiry
    • int ompt_get_runtime_version(char *buffer, int length)
  – tool control
    • void ompt_control(uint64_t command, uint64_t modifier)

• OMPD debugger support shared-library locations
  – char **ompd_dll_locations
    • argv-style list of filename strings
Outline

• OMPT - emerging performance tool API for OpenMP
  – overview and goals
  – state tracking
  – event notification
  – API

• Next steps
Next Steps

- Consolidate recent changes into the API description
  - functions to access thread, task, and region data omitted because of objections by some tool developers
  - omitted asynchronous access to parallel region and task “functions”
  - new type signatures for some callbacks
  - new signature for initialization
- Submit it to OpenMP language committee for comment
  - turn it into an official OpenMP TR
- Runtime implementations
  - IBM will release OMPT interface on BG/Q and Power
  - Rice and Oregon will finish draft of OMPT in Intel runtime
- Tools
  - Rice University’s HPCToolkit
    - OpenMP branch will be folded into trunk
  - University of Oregon’s Tau
Agenda

• OpenMP ARB Corporation
• A few features in 4.0 for exascale programming model
• Accelerators
  – OpenMP and OpenACC
• Affinity
• Tools and Debugging
• **The future of OpenMP**
• IWOMP 2014 and OpenMPCon 2014
What did we accomplish in OpenMP 4.0?

• Broad form of accelerator support
• SIMD
• Cancellation (start of a full error model)
• Task dependencies and task groups
• Thread Affinity
• User-defined reductions
• Initial Fortran 2003
• C/C++ array sections
• Sequentially Consistently Atomics
• Display initial OpenMP internal control variable state
OpenMP future features

- OpenMP Tools: Profilers and Debuggers
- Consumer style parallelism: event/async/futures
- Enhance Accelerator support/fpga
  - Multiple device type, linkable
- Additional Looping constructs
- Transactional Memory, Speculative Execution
- Task Model refinements
- CPU Affinity
- Common Array Shaping
- Full Error Model
- Interoperability
- Rebase to new C/C++/Fortran Standards
Agenda

• OpenMP ARB Corporation
• A few features in 4.0 for exascale programming model
• Accelerators
  – OpenMP and OpenACC
• Affinity
• Tools and Debugging
• The future of OpenMP
• IWOMP 2014 and OpenMPCon 2014
• **Salvador** is the largest city on the northeast coast of Brazil
  – The capital of the Northeastern Brazilian state of Bahia
  – It is also known as Brazil's capital of happiness
• **Salvador** was the first colonial capital of Brazil
  – The city is one of the oldest in the Americas

• **Getting There (SSA):**
  – Direct flights from US (Miami) and Europe (Lisbon, Madrid, & Frankfurt)
  – Alternatively, fly to Rio (GIG) or Sao Paulo (GRU) and connect to Salvador (SSA)

• **Average Temperatures in September:**
  – Average high: 27 °C / 81 °F
  – Daily mean: 25 °C / 77 °F
  – Average low 22 °C / 72 °F
Key Dates

- Submission: **May 2, 2014**
  - Automatic extension to May 9, 2014
- Reviews due May 30, 2014
- Communication to authors: June 6, 2014
- Final version: **June 20**
• The Design of OpenMP Thread Affinity
  – Alexandre E. Eichenberger, Christian Terboven, Michael Wong, and Dieter an Mey

• OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis
  – Alexandre E. Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Lohz, Daniel Lorenzk, and other members of the OpenMP Tools Working Group

• Early Experiences With The OpenMP Accelerator Model
  – Chunhua Liao, Yonghong Yan, Bronis R. de Supinski, Daniel J. Quinlan and Barbara Chapman

• Experimenting with Low-Overhead OpenMP Runtime on BG/Q
  – Alexandre E. Eichenberger and Kevin O’Brien
My blogs and email address

- My Blogs: [http://ibm.co/pCvPHR](http://ibm.co/pCvPHR)
- STM: [https://sites.google.com/site/tmforcplusplus/](https://sites.google.com/site/tmforcplusplus/)

- Chair of WG21 SG5 Transactional Memory
- IBM and Canada C++ Standard Head of Delegation
- ISOcpp.org Director, Vice President

**Tell us how you use OpenMP:**