MPI AND OPENMP* ON THETA

Carlos Rosales-Fernandez

2019 ALCF Computational Performance Workshop

Legal Disclaimer & Optimization Notice

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

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.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Overview

This talk is not intended to teach basic MPI or OpenMP*, but rather focus on hybrid MPI+OpenMP execution and advanced OpenMP capabilities

- Hybrid Computing: brief introduction to MPI and OpenMP
- OpenMP tasking
- Using OpenMP SIMD instructions
- OpenMP affinity
  - Pure OpenMP
  - Hybrid MPI+OpenMP

Hybrid Computing

Modern computers require multiple levels of parallelism to be effective

- System level
  - Distributed over a network fabric (Cray* Dragonfly)
  - Explicit communication (MPI)
- Node level
  - Across cores on a shared memory platform
  - Across hardware threads within a core
- Core level
  - Using vector instructions (Intel® Advanced Vector Extensions 512 - AVX512)
  - Exploiting multiple issue capabilities

What is MPI?

MPI stands for Message Passing Interface.

- Multi-language message passing standard API for parallelism
- Portable and scalable model for distributed memory parallel programming
- Language support for C/C++/FORTRAN.
- Provides APIs and environment variables to control the execution of parallel codes.
- Latest specs and examples are available at http://www.mpi-forum.org
- Multiple implementations (Intel® MPI Library, Cray MPICH*, Mvapich2, OpenMPI, …)

MPI Programming Model

Once MPI is initialized a number of processes are spawned which execute the same code in parallel until explicit synchronization is requested.

Memory spaces are separate, and information must be explicitly exchanged.

There is no automatic workload division across processes (MPI ranks)

MPI defines API calls to establish explicit communication

- Point-to-point calls (send, receive, …)
- Collective calls (bcast, reduce, …)

Using the MPI API

A basic program requires only a handful of code modifications

- Including the right header or module
- MPI_Init
- MPI_Finalize

Notice that Fortran calls always have an extra argument, the error code:

```
CALL MPI_FINALIZE( errCode )
```

In C the error code is the return value:

```
errCode = MPI_Finalize();
```

```
USE MPI
...
CALL MPI_INIT( errCode )
CALL MPI_COMM_RANK( MPI_COMM_WORLD, rank, errCode )
WRITE(*,*)'Hi from rank ',rank
CALL MPI_FINALIZE( errCode )
```

```
#include <mpi.h>
...
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
printf( "Hi from rank %d\n", rank);
MPI_Finalize();
```

What is OpenMP*?

OpenMP stands for Open Multi-Processing. It provides:

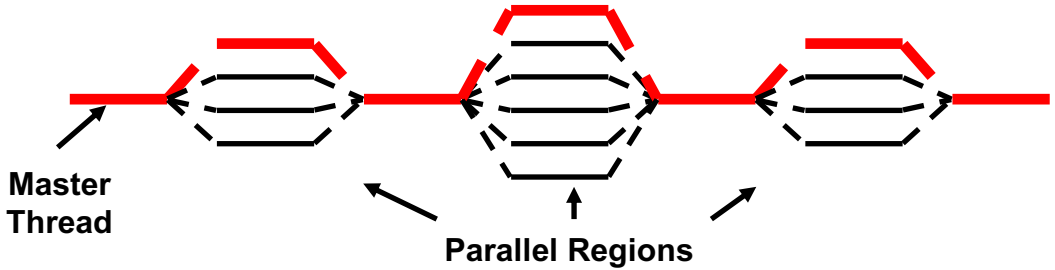
- Standardized directive-based multi-language high-level parallelism.
- Portable and Scalable model for shared-memory parallel programmers.
- Language support for C/C++/FORTRAN.
- Provides APIs and environment variables to control the execution of parallel regions.
- Latest specs and examples are available at http://www.openmp.org/specifications/.
- Supported by LLVM, Visual Studio Compiler, Intel Compiler, GNU GCC and others.

OpenMP* Programming Model

Real world applications are a mix of serial and inherently parallel regions.

OpenMP* provides **Fork-Join Parallelism** as a means to exploit inherent parallelism in an application within a **shared memory architecture**.

- Master thread executes in serial mode until a parallel construct is encountered.
- After the parallel region ends team threads synchronize and terminate, but master continues.



OpenMP* Constructs

Basic Components

Parallel - thread creation

- parallel

Work Sharing - work distribution among threads

- do, for, sections, single

Data Sharing - variable treatment in parallel regions and serial/parallel transitions

- shared, private

Synchronization - thread execution coordination

- critical, atomic, barrier

Advanced Functionality

- Tasking, SIMD, Affinity, Devices (offload)

Runtime functions and control

```
!$OMP PARALLEL
    !$OMP DO
    do i = 1, N
        a(i) = b(i) + c(i);
    end do
!$OMP END PARALLEL
```

```
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < N; i++)
    {
        a[i] = b[i] + c[i];
    }
}
```

MPI and OpenMP Working Together - Initialization

MPI treats calls in threaded code in different ways depending on the initialization used:

```
int MPI_Init_thread( int *argc, char ***argv, int required, int *provided )

    MPI_INIT_THREAD( required, provided, ierror )
```

The allowed values for the **required** level of thread support are:

- **MPI_THREAD_SINGLE** Only one thread will execute.
- **MPI_THREAD_FUNNELED** The process may be multi-threaded, but only the main thread will make MPI calls (all MPI calls are funneled to the main thread).
- **MPI_THREAD_SERIALIZED** The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are serialized).
- **MPI_THREAD_MULTIPLE** Multiple threads may call MPI, with no restrictions. No direct support for thread IDs, so tags must be used with care to ensure correctness.

MPI+OpenMP - Common Implementations

MPI_THREAD_SERIALIZED

- Only one thread can execute
- Simple implementation using single construct
- Implicit synchronization already present in single construct

```
!$OMP SINGLE
    CALL MPI_XXX( ... )
!$OMP END PARALLEL
```

```
 #pragma omp single
{
   MPI_Xxx( ... );
}
```

MPI_THREAD_FUNNELED

- Only master thread can execute
- Simple implementation using master construct
- Master has no implied barrier, so an explicit synchronization is required

```
!$OMP MASTER
    CALL MPI_XXX( ... )
!$OMP END PARALLEL
!$OMP BARRIER
```

```
#pragma omp master
{
   MPI_Xxx( ... );
}
#pragma omp barrier
```

OPENMP* TASKING CONCEPTS

Some Background

Prior to standard version 3.0, OpenMP* was focused exclusively on Data Parallelism, distributing work over threads executing the same code.

This work sharing model presented some limitations

- A need for a known loop count
- Very limited ability for dynamic scheduling
- Inconvenient for naturally task-parallel problems (dependencies, nesting)

Task parallelism constructs were introduced to complement the already existing set that supported data parallelism

Task parallelism is particularly useful in irregular computing

What is an OpenMP* Task?

From the standard document: "*specific instance of executable code and its data environment*"

- Explicit task: work generated by the task construct
- Implicit task: threads of a parallel region

In this section of the talk I will be only discussing explicit tasks.

By default tasks are deferrable, so the generating thread may execute it immediately or queue it

```
#pragma omp task
myfunc();

#pragma omp task
for(int i = 0; i < N; i++){ … }
```

Task Synchronization

Sibling tasks

The taskwait construct can be used to wait for deferred task completion at some point in the code

```
#pragma omp task
myfunc();

#pragma omp task
for(int i = 0; i < N; i++){ … }

#pragma omp taskwait
```

Nested tasks

Synchronizing siblings and their descendants requires a taskgroup

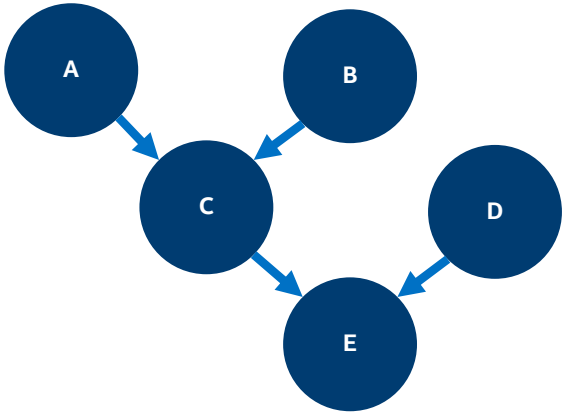
```
#pragma omp taskgroup
{
  #pragma omp task
  myfunc();

  #pragma omp task
  {
    for(int i = 0; i < N; i++){
      #pragma omp task
      nestedfunc();
    }
  }
}
```

Task Decomposition

Often an application can be decomposed into tasks which can execute simultaneously.

Following the Directed Acyclic Graph (DAG) shown on the right:

- Tasks A, B and C can start executing simultaneously.
- Task C can only be executed after A and B complete execution.
- Task E can only be executed after C and D complete execution.




```
a = A();
b = B();
c = C(a,b);
d = D();
printf( "%f\n", E(c,d) );
```

Parallel Execution of Tasks

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        a = A();
        #pragma omp task
        b = B();
        #pragma omp task
        d = D();
    }
}
c = C(a, b);
printf ( "%f\n", E(c,d) );
```

Start parallel region, forking N threads

Use a single thread to generate the tasks

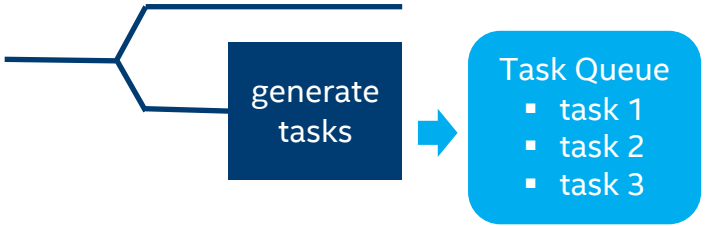
Each independent code section may be defined as a task

Once generated, each task may be performed by any available thread in the parallel region

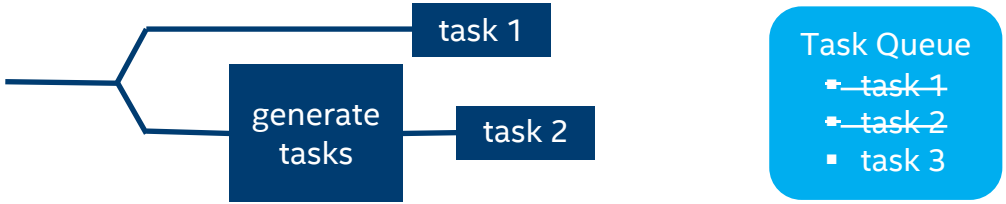
Task Generation and Execution

1. Threads are spawned from master

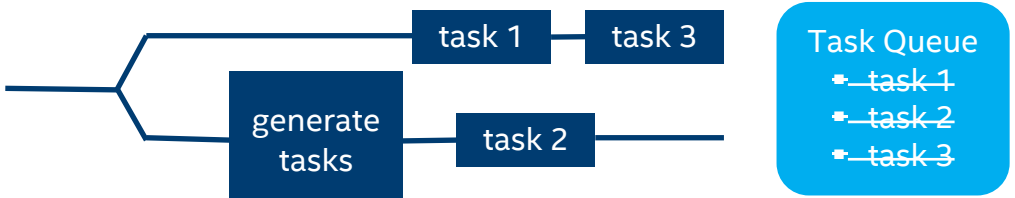
2. Work queue is generated by single thread



3. Tasks in queue are assigned to threads and executed



4. Process continues until queue is empty (or sync point)



Better Scheduling with Depend Clause

```
#pragma omp parallel
{
    #pragma omp single
    {
     #pragma omp task depend(out:a)
     a = A();
     #pragma omp task depend(out:b)
     b = B();
     #pragma omp task depend(out:d)
     d = D();
     #pragma omp task depend(in:a,b) depend(out:c)
     c = C(a, b);
     #pragma omp task depend(in:c,d)
     printf ( "%f\n", E(c,d) );
   }
}
```

depend clause allows to specify dependencies among tasks

depend(<in|out|inout>:<variables>)

Based on dependences C() can start executing once A() and B() are done.

Using the depend clause makes it possible to execute C() and D() simultaneously

Parallelize Recursions

```
void merge_sort_openmp(int a[], int tmp[], int first, int last)
{
  if (first < last) {
    int middle = (first + last + 1) / 2;
    if (last - first < 5000) {
      merge_sort(a, tmp, first, middle - 1);
      merge_sort(a, tmp, middle, last);
    } else {
      #pragma omp task
      merge_sort_openmp(a, tmp, first, middle - 1);
      #pragma omp task
      merge_sort_openmp(a, tmp, middle, last);
      #pragma omp taskwait
    }
    merge(a, tmp, first, middle, last);
  }
}
```

Merge sort is common recursive algorithm

- Its recursive nature used to pose a challenge in terms of expressing the parallelism.
- OpenMP* Tasking helps express the parallelism in recursive calls as shown below.
- Explicit taskwait synchronization forces a wait until all sibling tasks complete execution.
- Merging phase can't start until all the tasks spawned above have completed.

Other Interesting Tasking Tidbits

Tasks can be stopped and continued (at scheduling points). By default tasks are tied so they can only be continued by the same thread that started them (hot cache). This behavior can be overridden with the untied clause

```
#pragma omp task untied
```

You may introduce your own scheduling points using the taskyield directive

```
#pragma omp taskyield
```

The taskloop directive may be used to schedule loop iterations as independent tasks with a single generator (Intel® Compiler version 18+)

```
#pragma omp taskloop [[grainsize|numtask] [untied] [nogroups] [priority]]
for( i = 0; i < N; i++){ …}
```

Tasking Summary

Introduced to enable task-parallelism in shared memory architectures

Mostly used in irregular computing

Tasks are typically generated by a single thread

Dependencies can be specified to improve scheduling efficiency

Untied task generators can ensure progress

First-private is default data-sharing attribute

Shared variables remain shared

VECTORIZATION WITH OPENMP* SIMD

OpenMP* SIMD

A few critical capabilities were introduced in OpenMP* with the standard specification 4.0 (not an exhaustive list!)

- ~~Target Constructs : Accelerator support~~
- ~~Task Groups/Dependencies : Runtime task dependencies & synchronization~~
- SIMD : fine grained data level parallelism
- Affinity : Pinning workers to cores/HW threads

Refinements to SIMD were also introduced in specification 4.5

SIMD is of critical importance on Theta due to the 512bit width of the KNL processors

Affinity is also of critical importance with 256 threads per socket

The OpenMP* SIMD directive

```
#pragma omp simd [clause]
for(int i = 0; i < N; i++)
{
   ...
}
```

```
!$omp simd [clause]
do i = 1, N
   ...
end do
!$omp end simd
```

WARNING: The compiler ignores dependencies when using the simd directive .

Multiple clauses available

- safelen(length)
- simdlen(length)
- linear(list[:linear-step])
- aligned(list[:alignment])
- private(list)
- lastprivate(list)
- reduction(op: list)
- collapse(n)

Details and Limitations

Do/For-loop has to be in “canonical loop form” (see OpenMP 4.0 API:2.6)

`safelen(n)` : The compiler can assume a vectorization for a vector of length of n to be safe

`simdlen(n)` : Preferred vector length

`linear(var:step)` : For every iteration of the original scalar loop var is incremented by step. Therefore it will be incremented by step * vector_length for the vectorized loop.

`aligned(var:base)`: Assert that var is aligned to base bytes; (default is architecture specific alignment)

SIMD Example

This example instructs the compiler to ignore data dependencies, asserts array alignment, and indirectly mitigates the control flow dependence.

OpenMP* SIMD must be enabled at compilation time with either -qopenmp or -qopenmp-simd flags

```
#pragma omp simd safelen(32) aligned(a:64, b:64)
for(int i = 0; i < N; i++)
{
  a[i] = (a[i] > 1.0) ? a[i]*b[i] : a[i+off]*b[i];
}
```

SIMD Enabled Functions

Applying the declare simd construct to a function creates one or more versions of the function that can process multiple arguments using SIMD instructions from a single invocation from a SIMD loop.

```
#pragma omp declare simd [clause]
double work(double *a,double *b,int off);
```

```
function work(a,b,off)
!$omp omp declare simd [clause]
implicit none
integer           :: off
double precision :: a(*), b(*)
 ...
end function
```

Multiple clause options

- simdlen(length)
- linear(list[:linear-step])
- aligned(list[:alignment])
- uniform(list)
- inbranch
- notinbranch

SIMD Enabled Function Example

```
#pragma omp declare simd simdlen(16) notinbranch uniform(a, b, off)
double work( double *a, double *b, int i, int off )
{
  return (a[i] > 1.0) ?  a[i]*b[i] : a[i + off]*b[i];
}

void vec2( double *a, double *b, int off, int len )
{
  #pragma omp simd safelen(64) aligned(a:64, b:64)
  for( int i = 0; i < len; i++ )
  {
     a[i] = work( a, b, i, off );
  }
}
```

SIMD + Threads

By combining syntax we can both parallelize and vectorize a loop:

```
#pragma omp parallel for simd [clause]

!$omp parallel do simd [clause]
```

Where the clauses are those valid for either a do/for directive or a simd directive.

Loop will be distributed among threads using chunks that are multiples of the vector size

AFFINITY CONTROL WITH OPENMP*

Thread Affinity in OpenMP*

OpenMP* 4.0 introduces the concept of Places and Policies

- Set of threads running on one or more processors
- Places can be defined by the user
- Predefined places available: threads, cores, sockets
- Predefined policies : spread, close, master

And means to control these settings

- Environment variables OMP_PLACES and OMP_PROC_BIND
- Clause `proc_bind` for parallel regions

Optimal settings depend on application and workload

Pure OpenMP* on Theta

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

Disabling affinity with aprun is simple:

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

Now threads can be pinned to specific hardware resources using the OMP_PLACES and OMP_PROC_BIND environmental variables.

Rich set of options with lots of flexibility and configuration granularity, but a few simple setups cover the vast majority of production cases.

Pinning Step 1: OMP_PLACES

Two levels of granularity. You may specify a policy:

```
OMP_PLACES=<policy>
```

Where policy may be

- sockets : threads are allowed to float on sockets (multiple cores)
- cores : threads are allowed to float on cores (multiple logical processors)
- threads : threads are bound to specific logical processors

Or you may specify a list:

```
OMP_PLACES={lower_bound:length:stride}:repeat:increment
```

Pinning Step 2: OMP_PROC_BIND

To specify how threads are bound within the defined places use:

```
OMP_PROC_BIND=<policy>
```

Where policy must be chosen from:

- close : threads paced consecutively, as near to the master place as possible
- spread : threads spread equally on hardware to use most resources
- master : threads placed on master place to enhance locality

Note that specifying master could lead to heavy oversubscription of hardware resources, depending on the defined places.

It is possible to print out your pining specification as interpreted by OpenMP* using

```
OMP_DISPLAY_ENV=true
```

Some examples

```
OMP_NUM_THREADS=4; OMP_PLACES="{0:4:2}"
```

Bound to [0] [2] [4] [6]

```
OMP_NUM_THREADS=4; OMP_PLACES=threads; OMP_PROC_BIND=close
```

Bound to [0] [64] [128] [192]

```
OMP_NUM_THREADS=4; OMP_PLACES=threads; OMP_PROC_BIND=spread
```

Bound to [0] [16] [32] [48]

```
OMP_NUM_THREADS=4; OMP_PLACES=cores; OMP_PROC_BIND=spread
```

Bound to [0,64,128,192] [16, 80, 144, 208] [32, 96, 160, 224] [48, 112, 176, 240]

Hybrid MPI + OpenMP*

When using hybrid applications aprun must be configured to create pinning ranges for each MPI task, and then OpenMP variables may be set to control thread pinning within each rank processor range. Example: 4 MPI tasks, 16 , 8 nodes

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

	Thread 0	Thread 1	…	Thread 15
Task 0	[0, 64, 128, 192]	[1, 65, 129, 193]	…	[15, 79, 143, 207]
Task 1	[16, 80, 144, 208]	[17, 81, 145, 209]	…	[31, 95, 159, 223]
Task 2	[32, 96, 160, 224]	[33, 97, 161, 225]	…	[47, 111, 175, 239]
Task3	[48, 112, 176, 240]	[49, 113, 177, 241]	…	[63, 127, 191, 255]

NUMA considerations

Locality

- Local memory accesses reduce latency.
- Use Linux first touch policy to your advantage by initializing data in an OpenMP* loop in the same way that it will be used later.

MCDRAM

- Provides higher bandwidth
- Important to make a conscious choice if running on flat mode

If running on flat mode you may use numactl to attach to the numa node 1 (MCDRAM) :

```
aprun -n <ntot> -N <ppn> numactl --membind=1    ./exe
aprun -n <ntot> -N <ppn> numactl --preferred=1 ./exe
```

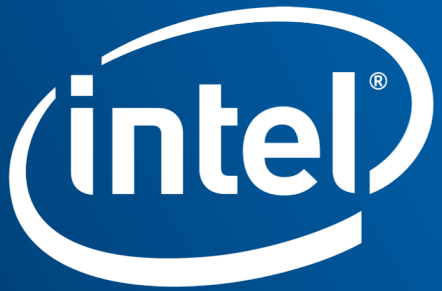
Recommended settings for Theta

The following setup is recommended for jobs using up to 4 threads per core

```
OMP_PLACES=cores

OMP_PROC_BIND=spread

aprun -n <totalTasks> -N <tasksPerNode> -cc depth -d 256/<tasksPerNode> -j 4
```

If using multiple threads per core you may want to test the effect of changing the default wait policy to passive:

```
OMP_WAIT_POLICY=passive
```



Software