Preparing for Stampede: Programming Heterogeneous Many-Core Supercomputers

Dan Stanzione, Lars Koesterke, Bill Barth, Kent Milfeld
dan/lars/bbarth/milfeld@tacc.utexas.edu

XSEDE 12
July 16, 2012
OpenMP and Offloading
Outline

• Strategies for Migrating to Stampede
• OpenMP – refresh
• MIC Example Codes
• Heterogeneous Threading with OpenMP
• Strategies for Migrating to Stampede
• OpenMP – refresh
• MIC Example Codes
• Heterogeneous Threading with OpenMP
Application Migration to Stampede Cluster

User View of a Cluster

User View Stampede

Migration to Stampede

MPI on CPU and MIC (w/wo OpenMP)

Offloaded execution (threaded) to MIC

Symmetric Computing

Asymmetric Computing
Strategies

• Vector Reawakening
  – Pre-Sandy Bridge – DP vector units are only 2 wide
  – Sandy Bridge – DP vector units are 4 wide
  – MIC -- DP vector units are 8 wide

• Unvectorized loops lose 4x performance on Sandy Bridge and 8x performance on MIC!

• Evaluate performance with/without compiler vectorization turned on to assess overall vectorization

DP=double precision (8 bytes)
• Strategies for Migrating to Stampede
• OpenMP – refresh
• MIC Example Codes
• Heterogeneous Threading with OpenMP
What is OpenMP?

• OpenMP is an acronym for Open Multi-Processing
• An Application Programming Interface (API) for developing parallel programs in shared memory architectures
• Three primary components of the API are:
  – Compiler Directives
  – Runtime Library Routines
  – Environment Variables
• de facto standard -- specified for C, C++, and FORTRAN
• http://www.openmp.org/ has the specification, examples, tutorials and documentation
OpenMP

• All about executing concurrent work (tasks).
  – Tasks execute independently
  – Variable updates must be mutually exclusive
  – Synchronization through barriers

```c
... some work
  // We use loops for repetitive tasks
  for (i=0; i<N; i++)
    a[i] = b[i] + c[i];
... some more work
```

```c
... some work
  // We often update variables
  for (i=0; i<N; i++)
    sum = sum + b[i]*c[i];
... some more work
```

#pragma omp parallel ...
OpenMP Fork-Join Parallelism

- Programs begin as a single process: master thread
- Master thread executes in serial mode until the parallel region construct is encountered
- Master thread creates (forks) a team of parallel threads that simultaneously execute tasks in a parallel region
- After executing the statements in the parallel region, team threads synchronize and terminate (join) but master continues
Programming with OpenMP on Shared Memory Systems

Hardware Model: Multiple Cores

Shared

Core 0
Core 1
Core 2
Core M-1

Software Model: Threads in Parallel Region

Shared

Thread 0
Thread 1
Thread 2
Thread M
Thread M+1
Thread M+2
Thread 2M-1

M threads are usually mapped to M cores.

For HyperThreading, 2 SW threads are mapped to 2 HW threads on each core.

On MIC there are 4 Hardware threads/core.

= accessible by all threads

= private memory for thread x
Thread Memory Access

- Every thread has access to “global” (shared) memory
- All threads share the same address space
- Threads can synchronize through barriers (implicit/explicit), and can have exclusive access to shared variables.

Simultaneous updates to shared memory can create a race condition. Use mutual exclusion directives to avoid data race conditions. Here is the C/C++ syntax:

- `#pragma omp atomic` — for single statements
- `#pragma omp critical` — for code block
OpenMP Syntax

- OpenMP Directives: **Sentinel construct** and clauses
  
  \#pragma omp construct [clause [[,]clause]...]  \hspace{1cm} C
  
  !$omp construct [clause [[,]clause]...]  \hspace{1cm} F90

- Example
  
  \#pragma omp parallel private(i) reduction(+:sum)  \hspace{1cm} C
  
  !$omp parallel private(i) reduction(+:sum)  \hspace{1cm} F90

Most OpenMP constructs apply to a “structured block”, that is, a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
OpenMP Constructs

Parallel control structures
- governs flow of control in the program
  - parallel directive

Parallel control worksharing
- distributes work among threads
  - do
  - for
  - sections
  - single
  - directive

Control single task
- assigns work to a thread
  - task
  - directive

Data environment
- specifies variables as shared or private
  - shared
  - private
  - reduction clause

Synchronization
- coordinates thread execution
  - critical
  - atomic
  - barrier
  - directive

Runtime environment
- runtime functions
- environment variables
  - omp_set_num_threads()
  - omp_get_thread_num()
  - OMP_NUM_THREADS
  - OMP_SCHEDULE
- scheduling
total, dynamic, guided
Parallel Region & Worksharing

Use OpenMP directives to specify Parallel Region, Worksharing constructs, and Mutual Exclusion

C

#pragma omp parallel

//end of parallel block

Use parallel ... end parallel for F90
Use parallel {...} for C

parallel do/for
parallel sections

<table>
<thead>
<tr>
<th>Code block</th>
<th>Each thread executes</th>
</tr>
</thead>
<tbody>
<tr>
<td>do / for</td>
<td>Worksharing</td>
</tr>
<tr>
<td>sections</td>
<td>Worksharing</td>
</tr>
<tr>
<td>single</td>
<td>One Thread (Worksharing)</td>
</tr>
<tr>
<td>critical</td>
<td>One Thread at a time</td>
</tr>
<tr>
<td>atomic</td>
<td>One Thread at a time</td>
</tr>
</tbody>
</table>

A single worksharing construct (e.g. a do/for) may be combined on a parallel directive line.
### Worksharing: Loop

#### C/C++

```c
1 #pragma parallel for
2 for (i=0; i<N; i++)
3 {
4     a[i] = b[i] + c[i];
5 }
```

#### General form:

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

**Line 1**  Team of threads formed (parallel region).

**Line 2-5** Loop iterations are split among threads. Implied barrier at end of block {}.

Each loop iteration must be independent of other iterations.
F90

1  !$omp parallel do
2  do i=1,N
3    a(i) = b(i) + c(i)
4  enddo
5  !$omp end parallel do

General form:

1  !$omp parallel
d2  !$omp do
do i=1,N
d3    a(i) = b(i) + c(i)
d4  enddo
5  !$omp end parallel

Line 1  Team of threads formed (parallel region).
Line 2-4 Loop iterations are split among threads.
Line 5  (Optional) end of parallel loop (implied barrier at enddo).

Each loop iteration must be independent of other iterations.
• Strategies for Migrating to Stampede
• OpenMP – refresh
• MIC Example Codes
• Heterogeneous Threading with OpenMP
Programming Models

Ready to use on day one!

• TBB’s will be available to C++ programmers

• MKL will be available
  – Automatic offloading by compiler for some MKL features

• Cilk Plus
  – Useful for task-parallel programming (add-on to OpenMP)
  – May become available for Fortran users as well

• OpenMP
  – TACC expects that OpenMP will be the most interesting programming model for our HPC users
MIC Programming with OpenMP

• MIC specific `pragma` precedes OpenMP pragma
  – Fortran: `!dir$ omp offload target(mic) <...>`
  – C: `#pragma offload target(mic) <...>`

• All data transfer is handled by the compiler (optional keywords provide user control)

• Offload and OpenMP section will go over examples for:
  1. Automatic data management
  2. Manual data management
  3. I/O from within offloaded region
     • Data can “stream” through the MIC; no need to leave MIC to fetch new data
     • Also very helpful when debugging (print statements)
  4. Offloading a subroutine and using MKL
Example 1

2-D array (a) is filled with data on the coprocessor

Data management handled automatically by compiler

Memory for (a) allocated on coprocessor

Private variables (i, j, x) are created

Result is copied back

```fortran
integer            :: n = 1024         ! Size
real, dimension(:,,:), allocatable :: a ! Array
integer            :: i, j             ! Index
real               :: x                ! Scalar
allocate(a(n,n)) ! Allocation

!$ use omp_lib ! OpenMP
!$omp parallel do shared(a, n), & ! Par. region
private(x, i, j), schedule(dynamic)
do j=1, n
   do i=j, n
      x = real(i + j); a(i,j) = x
   enddo
enddo
```

```fortran
#include <omp.h> /* C example */
#include <stdlib.h>
int main() {
   const int n = 1024; /* Size of the array */
   float   a[n][n];    /* Array */
   int     i, j;       /* Index variables */
   float   x;          /* Scalar */

   #pragma offload target(mic) ! Offloading
   #pragma omp parallel for shared(a), & ! Par. region
      private(x), schedule(dynamic)
   for(i=0;i<n;i++) {
      for(j=i;j<n;j++) {
         x = (float)(i + j); a[i][j] = x;
      }
   }
}
```
Data management by programmer

Copy in without deallocation:
\[
\text{in}(a: \text{free}_\text{if}(0))
\]
First and second use without any data movement:
\[
\text{nocopy}(a)
\]
Finally, copy out without allocation:
\[
\text{out}(a: \text{alloc}_\text{if}(0))
\]

Example 2

Stencil update and reduction with persistent data

```
! Allocate, transfer, no deallocation
!dir$ omp offload target(mic) in(a: free_if(0))

! Offloading: no alloc. no trans, no dealloc
!dir$ omp offload target(mic) nocopy(a)
(!$omp parallel do shared(a)
do j=2, n-1
   do i=2, n-1
      a(i,j) = 0.5*(a(i+1,j) + a(i-1,j) + &
                     a(i,j-1) + a(i,j+1))
   enddo
endo
endo

sum = 0. ! host code between offloaded regions

! Offloading: no alloc., no trans, no dealloc
!dec$ omp offload target(mic) nocopy(a)
(!$omp parallel do shared(a) reduction(+:sum)
do j=1, n
   do i=1, n
      sum = sum + a(i,j)
   enddo; enddo

! No alloc., transfer, deallocate
!dir$ omp offload target(mic) out(a: alloc_if(0))
```
Example 3

- I/O from within offloaded region
- File opened/closed by one thread (omp single)
- Threads read from file (omp critical)
- Threads may read in parallel (not shown)
  - Parallel file system
  - Threads read different parts of file, stored on different targets

```c
#pragma offload target(mic) //Offload region
#pragma omp parallel
{
    #pragma omp single /* Open File */
    {
        printf("Opening file in offload region\n");
        f1 = fopen("/var/tmp/mydata/list.dat","r");
    }

    #pragma omp for
    for(i=1;i<n;i++) {
        #pragma omp critical
        {
            fscanf(f1,"%f",&a[i]);
            a[i] = sqrt(a[i]);
        }
    }

    #pragma omp single
    {
        printf("Closing file in offload region\n");
        fclose(f1);
    }
}
```
Example 4

• Two routines `sgemm` (MKL) and `my_sgemm`
• Both called with `offload` directive
  – Explicit data movement used for `my_sgemm`
  – Input: `in(a, b)`
  – Output: `out(d)`
• Use **special attribute** to have routine compiled for the coprocessor, or specify library use.

```
! Snippet from the Main Program
!dir$ attributes offload:mic :: sgemm

!dir$ offload target(mic) !Offload to MIC
call &
sgemm('N','N',n,n,n,alpha,a,n,b,n,beta,c,n)

! Offload to accelerator with explicit 
! clauses for the data movement
!dir$ offload target(mic) in(a,b) out(d)
call my_sgemm(d,a,b)

! Snippet from the Hand-coded subprogram
!dir$ attributes offload:mic :: my_sgemm
subroutine my_sgemm(d,a,b)
real, dimension(:, :) :: a, b, d
!$omp parallel do
do j=1, n
  do i=1, n
    d(i,j) = 0.0
  do k=1, n
    d(i,j) = d(i,j)+a(i,k)*b(k,j)
  enddo; enddo; endo
endo subroutine
```
• Strategies for Migrating to Stampede
• OpenMP – refresh
• MIC Example Codes
• Heterogeneous Threading with OpenMP
OpenMP Parallel – homogeneous threading

C/C++

```c
#pragma parallel private(id)
{
    id=omp_get_num_thread();
    foo(id);
}
```

F90

```f90
!$omp parallel private(id)
    id=omp_get_num_thread()
    foo(id)
!$omp end parallel
```
OpenMP Parallel – heterogeneous threading, serial
“offload()” is any form of routine or pragma that does offloading.

```
#define parallel
{
    #pragma omp single
    { offload();}
    #pragma omp for
    for(i=0; i<N; i++){
        ...
    }
}
```
OpenMP Parallel – heterogeneous threading, concurrent

C/C++

```c
#pragma parallel
{
    #pragma omp single no wait
    { offload(); }  
    #pragma omp for schedule(dynamic)
    for(i=0; i<N; i++){ ... }
}
```

F90

```fortran
!$omp parallel
    !$omp single
        call offload();
    !$omp end single no wait
    !$omp do schedule (dynamic)
    do i=1,N; ...  end do
    !$omp end parallel
```
```c
#include <omp.h>
#include <stdio.h>

int main() {
    const int N=100000000;
    int i, id, nt, N_mic, N_cpu;
    float *a;

    a = (float *) malloc(N*sizeof(float));
    for(i=0;i<N;i++)a[i]=-1.0; a[0]=1.0;

    N_mic = N/2; N_cpu = N/2;
    nt = 16; omp_set_num_threads(nt);

    #pragma omp parallel private(id)
    {
        #pragma omp single nowait
        {
            #pragma offload target(MIC:) inout(a:length(N_MIC)
            #pragma omp parallel for
            for(i=0;i<N_mic;i++) { a[i]=(float)i; }
        }

        #pragma omp for schedule(dynamic,N/nt)
        for(i=N_cpu;i<N;i++) { a[i]=(float)i; }
    }

    printf("a[0],a[N-1] %f %f\n",a[0],a[N-1]);
}
```
OpenMP 3.0 supports nested parallelism, older implementations may ignore the nesting and serialize inner parallel regions.

A nested parallel region can specify any number of threads to be used for the thread team, new id’s are assigned.
omp_set_nested(1);
omp_set_max_active_levels(2);
omp_set_num_threads(5);

#pragma omp parallel private(id)
{
    printf("reporting in from %d\n", omp_get_thread_num());

    #pragma omp sections nowait
    {
        #pragma omp section
        {
            for (i = 0; i < 2; i++)
            {
                #pragma omp task
                foo(i);
            }
        }
    }

    #pragma omp section
    {
        #pragma omp parallel for num_threads(3)
        for (i = 0; i < 3; i++)
        {
            bar(i);
        }
    }
}

Sections allow 1 generating thread in each section.

Nested level re-defines a thread team with new thread ids.
(Worksharing team is no longer dependent upon original parallel region team size.)

task allows “execute at will” scheduling.