LSU CENTER FOR COMPUTATION & TECHNOLOGY

LSU INFORMATION TECHNOLOGY SERVICES

OpenMP programming Part II

Shaohao Chen

High performance computing @ Louisiana State University

Part II

- ❑ Optimization for performance
- ❑ Trouble shooting and debug
  - Common Misunderstandings and Frequent Errors
  - Debug
- ❑ Use OpenMP with Intel Xeon Phi
  - native mode
  - offloading

Optimization for performance

- It may be possible to quickly write a correctly functioning OpenMP program, but not so easy to create a program that provides the desired level of performance.
- The most intuitive implementation is often not the best one when it comes to performance, but the parallel inefficiency is not directly visible simply by inspecting the source.
- Programmers have developed some rules of thumb on how to write efficient code.

❑ Optimize serial code

- Memory access patterns: rowwise for C and columnwise for Fortran.
- Lower data precision if possible
- Common subexpression elimination
- Loop unrolling
- Loop fusion
- Loop tiling
- If-statement collapse

❑ Cases for optimizing OpenMP parallel codes will be introduced at the following slides.

Case 1.1: A reduced number of barriers

```
#pragma omp parallel shared(n,a,b,c,d,sum) private(i)
{
   #pragma omp for nowait
   for (i=0; i<n; i++)  a[i] += b[i];

   #pragma omp for nowait
   for (i=0; i<n; i++) c[i] += d[i];
   #pragma omp barrier

   #pragma omp for nowait reduction(+:sum)
   for (i=0; i<n; i++) sum += a[i] + c[i];
}  /*-- End of parallel region --*/
```

- Use the nowait clause where possible, carefully inserting explicit barriers at specific points in the program as needed.
- Here vectors a and c are independently updated. Therefore a thread that has finished its work in the first loop can safely enter the second loop.
- The barrier ensures that a and c have been updated before they are used.

Case 1.2: avoid using atomic/critical constructs in a large loop

- The atomic construct avoids the data racing condition. Therefore this code gives a correct result.
- But the additions are performed one by one and there is additional performance penalty.
- This code is even slower than a normal serial code!

```
sum = 0;
 #pragma omp parallel for shared(n,a,sum) private(i) // Optimization: use reduction instead of atomic
 for (i=0; i<n; i++)
 {
       #pragma omp atomic
        sum = sum + a[i];
 }   /*-- End of parallel for --*/
printf("Value of sum after parallel region: %d\n",sum);
```

Case 1.3: Avoid Large Critical Regions

```
#pragma omp parallel shared(a,b) private(c,d)
{
……
  #pragma omp critical
  {
    a += 2 * c;
    c = d * d;   // Optimization: move this line out of critical region
  }
}  /*-- End of parallel region --*/
```

- The more code contained in the critical region, the greater the likelihood that threads have to wait to enter it, and the longer the potential wait times.
- The first statement is protected by the critical region to avoid a data race of the shared variable a.
- The second statement however involves private data only. There is no data race. It should be removed from the critical region.

Case 1.4: Maximize Parallel Regions

- Overheads are associated with starting and terminating a parallel region.
- Large parallel regions offer more opportunities for using data in cache and provide a bigger context for other compiler optimizations.
- The code in the right panel is better, because it has fewer implied barriers, and there might be potential for cache data reuse between loops.

```
#pragma omp parallel for
for (.....){
/*-- Work-sharing loop 1 --*/
}
#pragma omp parallel for
for (.....){
/*-- Work-sharing loop 2 --*/
}
```

```
#pragma omp parallel
{
#pragma omp for /*-- Work-sharing loop 1 --*/
{ ...... }

#pragma omp for  /*-- Work-sharing loop 2 --*/
{ ...... }
}
```

Case 1.5: Avoid Parallel Regions in Inner Loops

- In the left panel, the overheads of the parallel region are incurred n^2 times.
- The code in the right panel is better, because the parallel construct overheads are minimized.

```
for (i=0; i<n; i++)
for (j=0; j<n; j++){
  #pragma omp parallel for
  for (k=0; k<n; k++){
   .........
  }
}
```

```
#pragma omp parallel
{
  for (i=0; i<n; i++)
  for (j=0; j<n; j++){
    #pragma omp for
    for (k=0; k<n; k++){
       .........
    }
  }
}
```

False sharing

- Cache coherence mechanism: When a cache line is modified by one processor, other caches holding a copy of the same line are notified that the line has been modified elsewhere. At such a point, the copy of the line on other processors is invalidated.
- False sharing: When two or more threads update different data elements in the same cache line simultaneously, they interfere with each other.
- Note that a modest amount of false sharing does not have a significant impact on performance. However, if some or all of the threads update the same cache line frequently, performance degrades.
- Typically, the computing results in false sharing cases are still correct.
- False sharing is likely to significantly impact performance under the following conditions:

  1. Shared data is modified by multiple threads.

  2. The access pattern is such that multiple threads modify the same cache line(s).

  3. These modifications occur in rapid succession.

Case 1.6: Avoid false sharing (I)

- Example I of false sharing case:

```
#pragma omp parallel for shared(Nthreads,a) schedule(static,1)
for (int i=0; i<Nthreads; i++)   a[i] += i;        // Optimization: use a[i][0] instead of a[i]
```

- Each thread has its own copy of a[i], thus there is no data race and the computing result is correct.
- But all elements of a accesses to the same cache line, which results in false sharing and thus degrades the performance.

- This case can be optimized by array padding: Accesses to different elements a[i][0] are now separated by a cache line. As a result, the update of an element no longer affects other elements.

Case 1.7: Avoid false sharing (II)

- Example II of false sharing:

```
#pragma omp parallel shared(a,b)   // Optimization: variable a should be private.
{
    a = b + 1;
    ……
}
```

- Variable b is not modified, thus it does not cause false sharing.
- Variable a is modified, thus it causes false sharing.
- If there are a number of such initializations, they could reduce program performance. In a more efficient implementation, variable a is declared and used as a private variable instead.

Trouble shooting

- Up to now, we can see that it is easy to develop an OepnMP parallel program from a serial program. But it still remains the programmer's responsibility to identify and properly express the parallelism.
- One of the biggest drawbacks of shared-memory parallel programming is the high potential to meet a data race condition. In the case of race condition, the thread reading the value might get the old value or the updated one, or some other erroneous value if the update requires more than one store operation. This usually leads to indeterministic behavior, with the program producing different results from run to run.
- In the following, some common misunderstandings and frequent Errors will be analyzed.

Case 2.1: data race due to loop-carried dependence

- It is good to parallelize an iteration-independent loop

```
#pragma omp parallel for shared(n,a,b) private(i)
 for (i=0; i<n-1; i++)   a[i] = a[i] + b[i];
```

- But it could induce a race condition to parallelize an iteration-dependent loop

```
#pragma omp parallel for shared(n,a,b) private(i)
 for (i=0; i<n-1; i++)   a[i] = a[i+1] + b[i];
```

Different threads could simultaneously execute the statement a[i] = a[i+1] + b[i] for different values of i. Thus there arises the distinct possibility that for some value of i, the thread responsible for executing iteration i+1 does so before iteration i is executed. When the statement is executed for iteration i, the new value of a[i+1] is read, leading to an incorrect result.

Case 2.2: data race due to implied sharing

- By default, most variables (except loop variables) declared outside the parallel region are shared.
- Data race: multiple threads simultaneously store a different value in the same variable X.

```
int X;     // shared by default
#pragma omp parallel     // Correction: explicitly specify the data-sharing attributes
{
    int Xlocal = omp_get_thread_num();
    X = omp_get_thread_num();        /*-- Data race --*/
    printf("Xlocal = %d X = %d\n", Xlocal, X);
}  /*-- End of parallel region --*/
```

- It is better to explicitly specify the data-sharing attributes of variables and not rely on the default data-sharing attribute.
- For good performance, it is often best to minimize sharing of variables.

Case 2.3: data race due to missing private declaration

- The variables i and x are not explicitly declared as private.
- A loop variable is implicitly declared to be private according to the OpenMP default data-sharing rules.
- But the normal variable x is shared by default. This leads to a data race condition.

```
int i;
double h, x, sum=0.0;
h = 1.0/(double) n;
#pragma omp prarallel for reduction(+:sum) shared(h)  // Correction: private(x) should be added.
for (i=1; i <= n; i++) {
   x = h * ((double)i - 0.5);
   sum += (1.0 / (1.0 + x*x));
}
pi = h * sum;
```

Case 2.4: a loop variable that is implicitly shared

- In C, the index variables of the parallel for-loop (i in this case) are private by default, but this does not extend to the index variables of loops at a deeper nesting level (j in this case). This results in undefined runtime behavior.

```
int i, j;
#pragma omp parallel for   // correct version 1: private(j) should be explicitly added.
for (i=0; i<n; i++)
   for (j=0; j<m; j++) {       // correct version 2: declare j here,  for (int j=0; j<m; j++)
        a[i][j] = compute(i,j);
}
```

- In Fortran, loop index variables are private by default, because variables cannot be declared locally in a code block, such as a loop.

Case 2.5: incorrect use of the private clause

- First, variable b is used but not initialized within the parallel loop.
- Second, the values of variables a and b are undefined after the parallel loop.

```
int i, a, b=0;
#pragma omp parallel for private(i,a,b)
// Correction:  #pragma omp parallel for private(i) firstprivate(b) lastprivate(a,b)
for (i=0; i<n; i++)  {
    b++;
    a = b+i;
}   /*-- End of parallel for --*/
c = a + b;
```

- The firstprivate and lastprivate clauses should be used in this case.

Case 2.6: incorrect use of the master construct

- This code fragment implicitly assumes that variable Xinit is available to all threads after it is initialized by the master thread. This is incorrect. The master thread might not have executed the assignment when another thread reaches it.

```
int Xinit, Xlocal;
#pragma omp parallel shared(Xinit) private(Xlocal)
{
    #pragma omp master // correct version 1: use single construct instead, #pragma omp single
    {
        Xinit = 10;
    }
    // correct version 2: insert a barrier here, #pragma omp barrier
    Xlocal = Xinit;    /*-- Xinit might not be available for other threads yet --*/
}   /*-- End of parallel region --*/
```

Case 2.7: incorrect assumptions about work scheduling

- The nowait clause can help to increase performance by removing unnecessary barriers at the end of work-sharing constructs, however, care must be taken not to rely on assumptions about which thread executes which loop iterations.
- A compiler may choose to employ different strategies for dealing with remainder iterations in order to take advantage of memory alignment.
- The second loop might read values of array b that have not yet been written to in the first loop, even with static work scheduling, if n is not a multiple of the number of threads.

```
#pragma omp for schedule(static) nowait  // Correction: remove nowait clause
for (i=0; i<n; i++)   b[i] = (a[i] + a[i-1]) / 2.0;
#pragma omp for schedule(static) nowait
for (i=0; i<n; i++)   z[i] = sqrt(b[i]);
```

Case 2.8: incorrectly nested directives

- Nested parallelism is implemented at the level of parallel regions, not work-sharing constructs.

```
#pragma omp parallel shared(n,a,b)
{
    #pragma omp for
    for (int i=0; i<n; i++) {
        a[i] = i + 1;
        #pragma omp for   // Correction: parallel should be added, #pragma omp parallel for
        for (int j=0; j<n; j++)  b[i][j] = a[i]*2.0;
    }
} /*-- End of parallel region --*/
```

Case 2.9: illegal use of the barrier

- The barrier is not encountered by all threads in the team, and therefore this is not illegal.

```
#pragma omp parallel
{
    if ( omp_get_thread_num() == 0 ){
        .....
        #pragma omp barrier  // Correction: the barrier should be out of the if-else region
    }
    else{
        .....
        #pragma omp barrier
    }
}  /*-- End of parallel region --*/
```

- Also, a barrier should not be in a work-sharing construct, a critical section, or a master construct.

Case 2.10: a deadlock situation

```
work1(){
   /*-- Some work performed here --*/
   #pragma omp barrier   // Correction: remove this barrier
}
work2(){
   /*-- Some work performed here --*/
}

main(){
   #pragma omp parallel sections
   {
      #pragma omp section
      work1();
      #pragma omp section
      work2();
   }   // An implicit barrier
}
```

- If executed by two threads, this program never finishes.
- Thread1 executing work1 waits forever in the explicit barrier, which thread2 will never encounter.
- Thread2 executing work2 waits forever in the implicit barrier at the end of the parallel sections construct, which thread1 will never encounter.
- Note: Do not insert a barrier that is not encountered by all threads of the same team.

Case 2.11: an extra exclamation mark in Fortran code

- As a result the compiler ignores the private clause. Loop variable i is private by default, as intended, but variable cl is shared. This introduces a data race.

```
!$OMP PARALLEL SHARED(n,a,b,c)     ! This code is to calculate dot product of two vectors
!!$OMP& PRIVATE(i,cl)    ! Correction: delete the first exclamation mark !
!$OMP DO
   do i = 1, n
      cl = cl + b(i)*a(i)    ! calculate dot product for local parts of the vectors simultaneously
   end do
!$OMP END DO
!$OMP CRITICAL
   c = c + cl          ! add up all parts one by one
!$OMP END CRITICAL
!$OMP END PARALLEL
```

Case 2.12: missing a curly bracket in C code

- It is very likely an error was made in the definition of the second parallel region.
- Without the curly bracket, only the statement following the parallel directive is executed in parallel. In this case, function work4 is executed by the master thread only.

```
#pragma omp parallel
{
   work1();    /*-- Executed in parallel --*/
   work2();    /*-- Executed in parallel --*/
}
#pragma omp parallel        // Correction: a curly bracket { should be added.
    work3();    /*-- Executed in parallel --*/
    work4();   /*-- Executed sequentially --*/
}
```

Case 2.13: a function call that is not thread-safe

```
int icount;   // global variable, default to be shared

void lib_func() {
// Correction: use atomic or critical constructs here
   icount++;
   do_lib_work();
}

main (){
   #pragma omp parallel
   {
      lib_func();
   } /*-- End of parallel region -- */
}
```

- The library keeps track of how often its routines are called by incrementing a global counter, which is default to be shared.
- If this function is executed by multiple threads within a parallel region, all threads read and modify the shared counter variable, leading to a race condition.

Case 2.14: unsafe use of a shared C++ object

```
class anInt {    // Declare and define a class
    public:
    int x;
    anInt(int i = 0){ x = i; };
    void addInt (int y){ x = x + y; }   // Correction 1: use critical here
};

main(){
    anInt a(10);    // This class object is shared by default
    #pragma omp parallel
    {
        a.addInt(5);   // Correction 2: use critical here
    }
}
```

- The class objects and methods in C++ are default to be shared. If they are used within OpenMP parallel regions, race conditions can result.
- In order to make the code thread-safe, the invocation of the method or the update of the shared variable within the method should be enclosed by a critical region.

Debug

❑ Verification of the Sequential Version

- The first step when debugging a parallel application should always be the verification of the sequential version.
- Run the sequential code first and make sure that the computing result is correct.
- Run the loops backwards. If the result is wrong, the loops cannot be executed in parallel.

❑ Verification of the Parallel Code

- Run the OpenMP version of the program on one thread. If the error shows up then, there is most likely a basic error in the code.
- Selectively enable/disable OpenMP directives to zoom in on the part of the program where the error originates.

Keeping Sequential and Parallel Programs as a Single Source Code

- Conditional compilation in C

```
#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif
    . . . . . .
int TID = omp_get_thread_num();
```

- Conditional compilation in Fortran

```
#ifdef _OPENMP
    use omp_lib
#endif
    . . . . . .
integer:: TID
    . . . . . .
#ifdef _OPENMP
    TID = omp_get_thread_num()
#else
    TID = 0
#endif
```

- If one does not compile using the OpenMP option (flag), the OpenMP directives are simply ignored, and a sequential executable is generated.

Debug tools

- ❑ GNU gdb debugger
- ❑ DDT by Allinea
- ❑ TotalView by Etnus

A debugger allows the user to:

- stop a running program more or less at any point in the source
- examine and also change variables to do a “what if” kind of analysis.
- monitor the change of a specific memory location.

GNU gdb

- gdb *program* -- start gdb debugger
- run *[arglist]* -- start your program [with arglist]
- break *[file:]line* -- set breakpoint at line number [in file]
- break *[file:]:func* -- set breakpoint at a function [in file]
- delete *[n]* -- delete all breakpoints [or breakpoint n]
- print *expr* --- display the value of an expression
- c -- continue running your program
- next -- next line, stepping over function calls
- step -- next line, stepping into function calls
- help *[command]* – list classes of commands or describe command
- quit or q or Ctr-d -- exit gdb
- gdb work sheet: http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf

Use OpenMP with Xeon Phi

- ❑ native mode
- ❑ offload

  explicit offload

  auto offload

Intel Xeon Phi coprocessor (accelerator)

(parameters for Xeon Phi 7120P)

- Add-on to CPU-based system
- PCI express (6.66 ~ 6.93 GB/s)
- IP-addressable
- 16 GB memory
- 61 x86 64-bit cores (244 threads)
- single-core 1.2 GHz
- 512-bit vector registers
- 1.208 TeraFLOPS = 61 cores * 1.238 GHz * 16 DP FLOPs/cycle/core

Current: Knight Corner (KNC)

Next generation (2016): Knight Landing (KNL)

LSU CENTER FOR COMPUTATION & TECHNOLOGY

LSU INFORMATION TECHNOLOGY SERVICES

LSU INFORMATION TECHNOLOGY SERVICES

A typical compute node on SuperMIC

Two Intel® Xeon® CPUs	One Intel® Xeon Phi™ coprocessor	Two Intel® Xeon Phi™ coprocessors
• host	◈ mic0	◈ mic1
20 cores	61 cores (244 logical)	61 cores (244 logical)
64 GB memory	16 GB memory	16 GB memory

❑ Theoretical maximum acceleration:

One Xeon Phi / Two Xeons = 1208 GFLOPS / 448 GFLOPS = 2.7

(Tow Xeons + Two Xeon Phis) / Two Xeons = (2*1208 + 448) GFLOPS / 448 GFLOPS = 6.4

Native mode

Computer codes
execute natively
intel
Xeon Phi Coprocessor

❑ An example: vector addition, parallelized with OpenMP.

- No change to normal CPU source codes.

❑ Compilation

- Always compile codes on the host. Compiler is not available on Xeon Phi.
- icc -O3 -openmp vector_omp.c -o vec.omp.cpu # CPU binary
- icc -O3 -openmp -mmic vector_omp.c -o vec.omp.mic # MIC binary

❑ execute CPU binary on the host

- export OMP_NUM_THREADS=20 # set OepnMP threads on host. Maximum is 20.
- ./vec.omp.cpu # run on the host

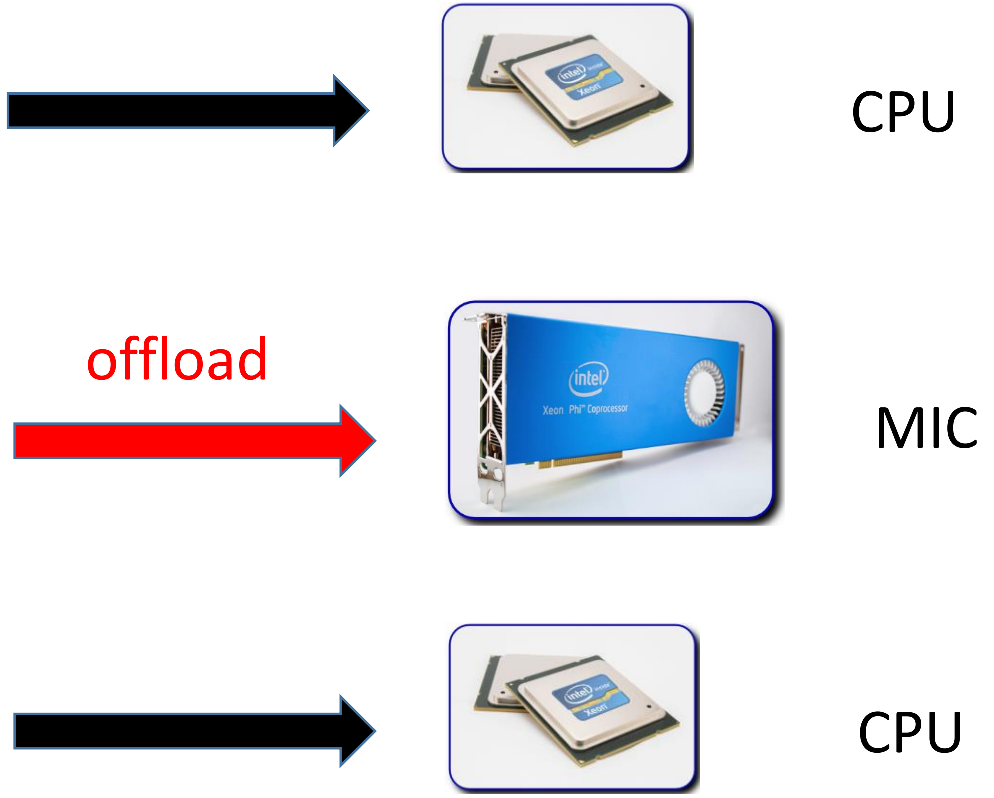
❑ execute MIC binary on Xeon Phi natively

- ssh mic0 # login mic0
- export LD_LIBRARY_PATH=/usr/local/compilers/Intel/composer_xe_2013.5.192/compiler/lib/mic
  # specify libs for MIC
- export OMP_NUM_THREADS=244 # Set OepnMP threads on mic0. Maximum is 244.
- ./vec.omp.mic # Run natively on mic0

Offload

❑ Example for explicit offload

```
int totalProcs;
int maxThreads;
#pragma offload target(mic:0)
 { // begin offload block
   totalProcs = omp_get_num_procs();
   maxThreads = omp_get_max_threads();
 } // end offload block
printf( "   total procs: %d\n", totalProcs );
printf( "   max threads: %d\n", maxThreads );
```



Explicit Offload: compile and run

❑ Compile

- The same as compiling normal CPU codes. Without -mmic.
- icc -openmp name.c -o name.off # C
- ifort -openmp name.f90 -o name.off # Fortran

❑ Execute offloading jobs from the host

- export MIC_ENV_PREFIX=MIC # set the prefix if launch from the host.
- export MIC_OMP_NUM_THREADS=240 # set number of threads for MIC (The default is the maximum, that is 240, not 244. Leave one core with 4 threads to execute offloading.)
- ./name.off # launch from the host

Offload an OpenMP region

- ❑ Spread OpenMP threads to 240 workers (logical threads) of MIC.
- ❑ Offload with explicit control of data transfer (in C)

```
#pragma offload target(mic:0)  in( a ) out( c, d ) inout( b )
 #pragma omp parallel for
 for ( i=0; i<100000; i++ )  {
     c[i] =  a[i] + b[i];
     d[i] =  a[i] - b[i];
     b[i] = -b[i];
 }
```

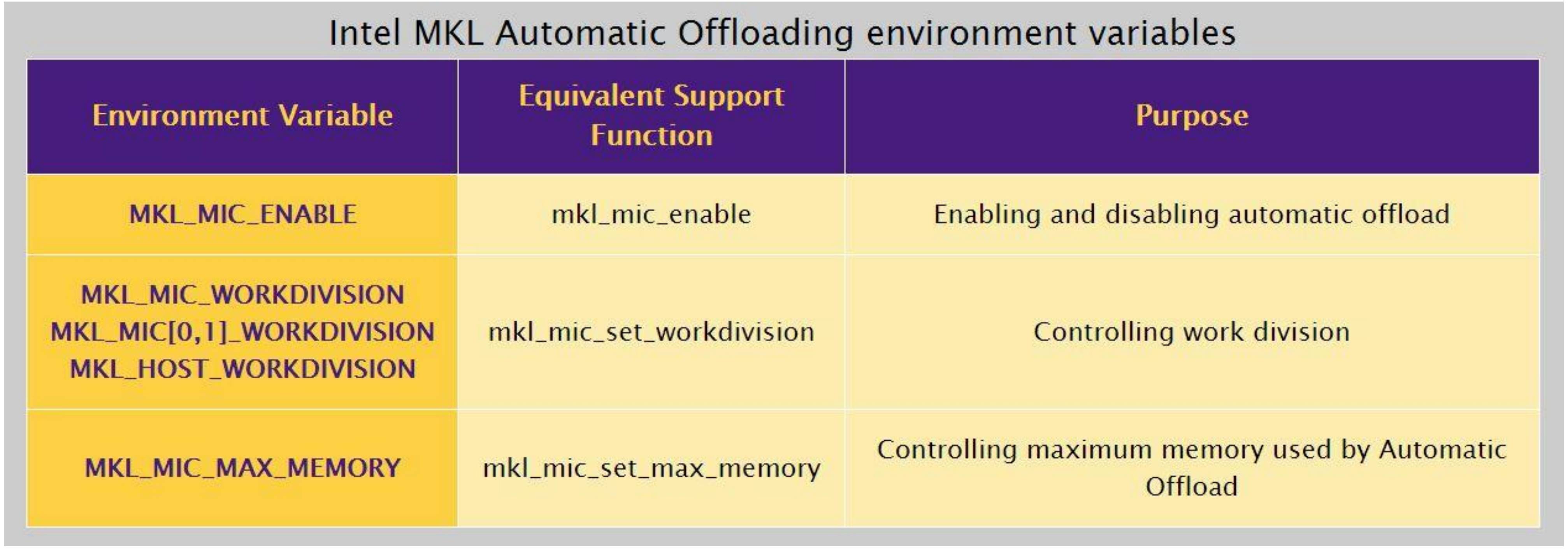
❑ Offload with explicit control of data transfer (in Fortran)

```
!dir$ offload target(mic)  in( a ), out( c, d ), inout( b )
!$omp parallel do
     do i=1,N
        c(i) =  a(i) + b(i)
        d(i) =  a(i) - b(i)
        b(i) = -b(i)
     end do
!$omp end parallel do
```

Automatic offload with Intel MKL

- Intel Math Kernel Library (MKL):

Highly vectorized and threaded Linear Algebra, Fast Fourier Transforms (FFT), Vector Math and Statistics functions.



Intel MKL Automatic Offloading environment variables

Environment Variable	Equivalent Support Function	Purpose
MKL_MIC_ENABLE	mkl_mic_enable	Enabling and disabling automatic offload
MKL_MIC_WORKDIVISION MKL_MIC[0,1]_WORKDIVISION MKL_HOST_WORKDIVISION	mkl_mic_set_workdivision	Controlling work division
MKL_MIC_MAX_MEMORY	mkl_mic_set_max_memory	Controlling maximum memory used by Automatic Offload

an example for auto offload

- Matrix product and addition: C = alpha*A*B + beta*C

- For C

```
……
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, p, alpha, A, p, B,
n, beta, C, n);    // Double-precision General Matrix Multiplication
……
```

- For Fortran

```
……
CALL DGEMM('N','N',M,N,P,ALPHA,A,M,B,P,BETA,C,M)
……
```

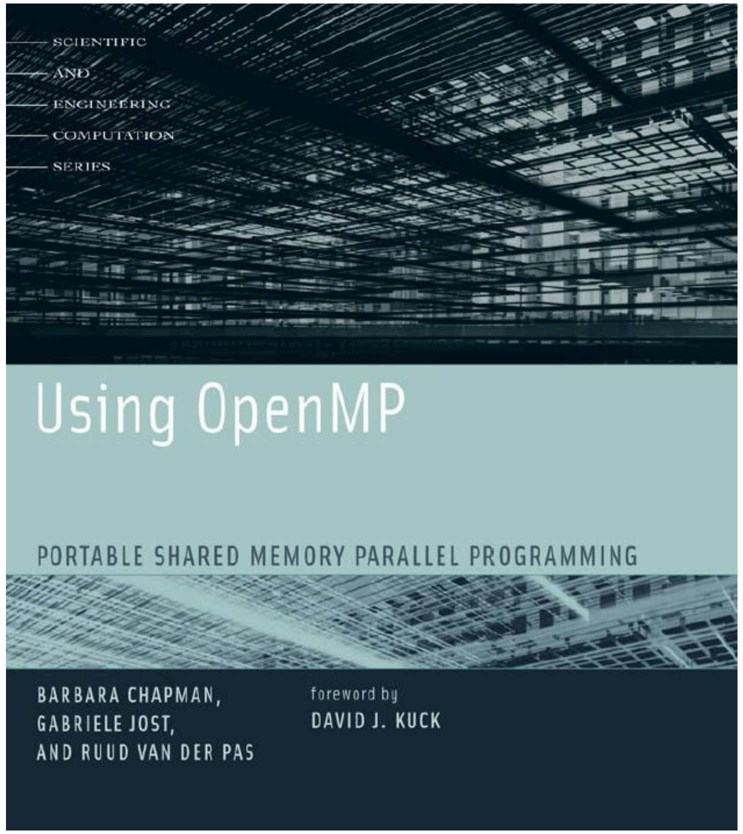
Auto offload: compile and run

- Compilation (the same as normal CPU code)
  - icc -openmp -mkl ao_intel.c -o ao_intel
  - ifort -openmp -mkl ao_intel.f -o ao_intel

- Run auto-offloading jobs
  - export MKL_MIC_ENABLE=1 # enable auto offload, also set the prefix MIC_
  - export OMP_NUM_THREADS=20 # set CPU threads
  - export MIC_OMP_NUM_THREADS=240 # set MIC threads from the host
  - ./ao_intel

- Depending on the problem size and the current status of the devices, the MKL runtime will determine how to divide the work between the host CPU's and the Xeon Phi's

References

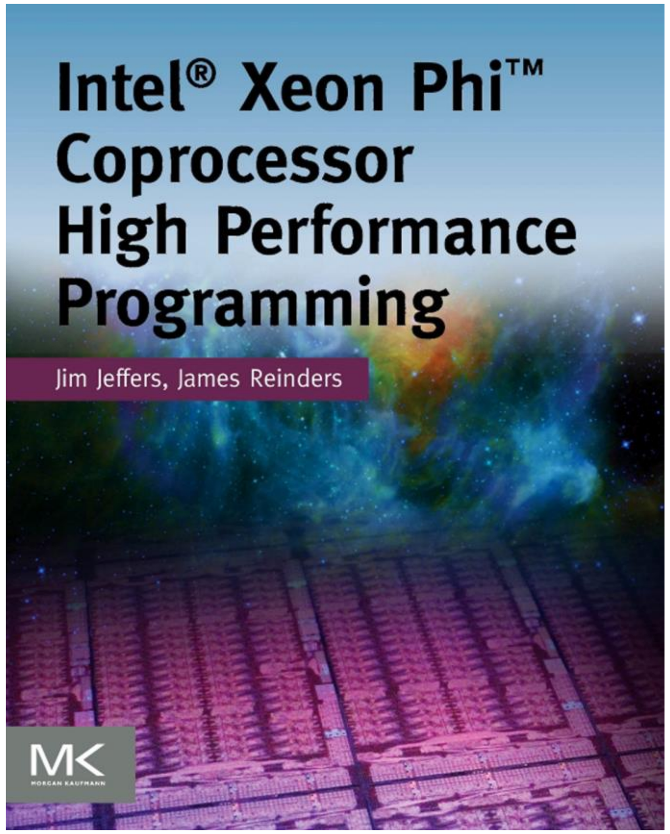

SCIENTIFIC AND ENGINEERING COMPUTATION SERIES

Using OpenMP

PORTABLE SHARED MEMORY PARALLEL PROGRAMMING

BARBARA CHAPMAN, GABRIELE JOST, AND RUUD VAN DER PAS

foreword by DAVID J. KUCK


Intel® Xeon Phi™ Coprocessor High Performance Programming

Jim Jeffers, James Reinders

- User guide of SuperMIC: http://www.hpc.lsu.edu/docs/guides.php?system=SuperMIC