Minnesota Supercomputing Institute
OpenMP
Motivation for Multithreading

More Cores

Wider Vectors

Source: https://software.intel.com/en-us/articles/intel-vectorization-tools
Motivation for Multithreading

Source: http://www.drdobbs.com
Shared Memory System

Source: https://portal.ivec.org/docs/Supercomputers/Magnus/Magnus_User_Guide
Current Architecture Trends

- Multi-socket nodes with rapidly increasing core counts.
- Memory per core decreases.
- Memory bandwidth per core decreases.
- Network bandwidth per core decreases.

Need a hybrid programming model with three levels of parallelism:

- MPI between nodes or sockets.
- Shared memory (such as OpenMP) on the nodes/sockets.
- Increase vectorization (SIMD) for lower level loops.
Strategies for a Shared Memory System

● Simple Parallelization: Collect Serial Calculations (see MSI tutorial “Parallel Computing Overview)

● Thread parallelization
  ○ A single program uses multiple “threads” which can communicate using shared memory.
  ○ Coding for thread parallelization often means using OpenMP (which is based on a POSIX threads implementation “under the covers”).

● Message Passing
  ○ Message passing frameworks such as MPI may be used, but are often not needed because thread parallel methods are sufficient.
Distributed Memory System: Cluster

Source: https://wiki.hpc.tulane.edu/trac/wiki/cypress/Programming/Mpi
Clusters at MSI

- **Mesabi**
  - About 17,700 total cores, on Intel Haswell processors.
  - 24 cores and 62 GB per node in the large primary queues.
  - Special queues with large memory (up to 1TB), and GPUs.
  - Allows node sharing: good for both small and large jobs.
  - mesabi.msi.umn.edu

- **Itasca**
  - About 9,000 total cores, on Intel Nehalem processors.
  - 8 cores and 22 GB per node in the large primary queue.
  - Special queues with larger memory and 16 cores per node.
  - itasca.msi.umn.edu

- **Interactive (Lab) Server**
  - About 500 total cores, on older hardware.
  - For interactive, or small single node jobs.
  - 8 cores and 15 GB per node in the primary queue.
  - lab.msi.umn.edu
Programming Difficulty

The general view of programming difficulty is that programming becomes more complicated in this order:

- Simple Parallelization: Collect Serial Calculations
- OpenMP (thread parallel)
- MPI (message passing)
- MPI + OpenMP (hybrid message passing + threads)
- Accelerators (GPUs using CUDA or OpenACC, PHIs)

(Note that GPUs can more easily be used via the nVidia supplied libraries, e.g. cuFFT, cuBLAS, cuSPARSE, etc. See: http://docs.nvidia.com/cuda/index.html#axzz3o0lsq3xx).

(Intel PHIs can be utilized using MKL, too).

The more difficult strategies can also yield larger speed increases, but it is important to examine the calculation type.
OpenMP

OpenMP is for parallelization on shared memory systems (at MSI, usually one node, composed of 2 sockets). OpenMP is an abbreviation for: **Open Multi-Processing**

OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs.

**OpenMP Compiler Directives**
- Interpreted when OpenMP compiler option is turned on.
- Each directive applies to the succeeding structured block.
Intel Haswell Xeon Socket Architecture

Source: http://www.theplatform.net/2015/05/05/intel-puts-more-compute-behind-xeon-e7-big-memory/

University of Minnesota
Haswell Architecture Details (lscpu)

In0005 % lscpu

Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 24
On-line CPU(s) list: 0-23
Thread(s) per core: 1
Core(s) per socket: 12
Socket(s): 2
NUMA node(s): 2
Vendor ID: GenuineIntel
CPU family: 6
Model: 63

Stepping: 2
CPU MHz: 2497.320
BogoMIPS: 4993.98
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 30720K
NUMA node0 CPU(s): 0-5,12-17
NUMA node1 CPU(s): 6-11,18-23
OpenMP: Arguments For & Against

**Pros:**

- **Programmability** - Easier to program/debug than MPI. Allows incremental introduction of OpenMP, one loop at a time.
- **Maintainability** - Code is easier to understand, so it may be more easily maintained. Allows for a single source version of code.
- **Minimal code modification** - Serial code usually doesn’t require modification. Can still run the program as a serial code.
- **Performance** - Most nodes on Itasca have 8 cores, and most nodes on Mesabi have 24 cores. An OpenMP application can use all cores on a node, giving a theoretical 24 fold performance improvement. Most modern laptops/servers have multi core CPUs.
- **Portability** - OpenMP is a standard not an implementation. OpenMP SIMD directive is the only portable way to force a loop to be vector.
OpenMP: Arguments For & Against

Cons:

**Memory** - Can only run on shared memory (usually one node). So can only use the memory on one node.

**Compiler Support** - Requires a compiler that supports OpenMP. All MSI compilers (Intel, GNU, PGI) support OpenMP.

**False sharing** - Possible data placement problem. Can be a problem if loops are not coded optimally.

**First touch** - Should try to initialize the memory in the same way you use it during computation. Discussed later.
OpenMP Terminology

Shared Memory Model:

- OpenMP is designed for multi-processor/core, shared memory machines. The underlying architecture can be shared memory UMA or NUMA.

Uniform Memory Access

Non-Uniform Memory Access

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP Terminology

Thread Based Parallelism:

- OpenMP programs accomplish parallelism exclusively through the use of threads.
- A thread of execution is the smallest unit of processing that can be scheduled by an operating system. The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is.
- Threads exist within the resources of a single process. Without the process, they cease to exist.
- Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP Terminology

Explicit Parallelism:

- OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
- Parallelization can be as simple as taking a serial program and inserting compiler directives....
- Or as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP

For parallelization on shared memory systems.

Source: http://en.wikipedia.org/wiki/OpenMP
OpenMP Terminology

OpenMP uses the fork-join model of parallel execution:

- All OpenMP programs begin as a single process: the `master thread`. The master thread executes sequentially until the first `parallel region` construct is encountered.
- **FORK**: the master thread then creates a team of parallel `threads`.
- The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
- **JOIN**: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.
- The number of parallel regions and the threads that comprise them are arbitrary.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP Terminology

Compiler Directive Based:
- Most OpenMP parallelism is specified through the use of compiler directives which are imbedded in C/C++ or Fortran source code.

Nested Parallelism:
- The API provides for the placement of parallel regions inside other parallel regions.
- Implementations may or may not support this feature.

Dynamic Threads:
- The API provides for the runtime environment to dynamically alter the number of threads used to execute parallel regions. Intended to promote more efficient use of resources, if possible.
- Implementations may or may not support this feature.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP Terminology

I/O:
- OpenMP specifies nothing about parallel I/O. This is particularly important if multiple threads attempt to write/read from the same file.
- If every thread conducts I/O to a different file, the issues are not as significant.
- It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.

Memory Model: FLUSH Often?
- OpenMP provides a "relaxed-consistency" and "temporary" view of thread memory (in their words). In other words, threads can "cache" their data and are not required to maintain exact consistency with real memory all of the time.
- When it is critical that all threads view a shared variable identically, the programmer is responsible for insuring that the variable is FLUSHed by all threads as needed.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP API Overview

Three Components:

- The OpenMP API is comprised of three distinct components. As of version 4.0:
  - Compiler Directives (44)
  - Runtime Library Routines (35)
  - Environment Variables (13)

- The application developer decides how to employ these components. In the simplest case, only a few of them are needed.

- Implementations differ in their support of all API components. For example, an implementation may state that it supports nested parallelism, but the API makes it clear that may be limited to a single thread - the master thread. Not exactly what the developer might expect?

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP API Overview

Compiler Directives:
- Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise - usually by specifying the appropriate compiler flag, as discussed in the Compiling section later.
- OpenMP compiler directives are used for various purposes:
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization of work among threads
- Compiler directives have the following syntax:
  - **sentinel**  **directive-name**  **[clause, ...]**
- For example:
  - Fortran:  
    ```
    !$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)
    ```
  - C/C++:
    ```
    #pragma omp parallel default(shared) private(beta,pi)
    ```

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP API Overview

Run-time Library Routines:

- The OpenMP API includes an ever-growing number of run-time library routines.
- These routines are used for a variety of purposes:
  - Setting and querying the number of threads
  - Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  - Setting and querying the dynamic threads feature
  - Querying if in a parallel region, and at what level
  - Setting and querying nested parallelism
  - Setting, initializing and terminating locks and nested locks
  - Querying wall clock time and resolution
- For C/C++, all of the run-time library routines are actual subroutines. For Fortran, some are actually functions, and some are subroutines. For example:
  - Fortran: INTEGER FUNCTION OMP_GET_NUM_THREADS()
  - C/C++: #include <omp.h>
    intomp_get_num_threads(void)

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP API Overview

Run-time Library Routines:

- Note that for C/C++, you usually need to include the `<omp.h>` header file.
- Fortran routines are not case sensitive, but C/C++ routines are.
- The run-time library routines are briefly discussed as an overview in the Run-Time Library Routines section, and in more detail in Appendix A.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP API Overview

Environment Variables:

- OpenMP provides several environment variables for controlling the execution of parallel code at run-time.
- These environment variables can be used to control such things as:
  - Setting the number of threads
  - Specifying how loop iterations are divided
  - Binding threads to processors
  - Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
  - Enabling/disabling dynamic threads
  - Setting thread stack size
  - Setting thread wait policy

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP API Overview

Environment Variables:

- Setting OpenMP environment variables is done the same way you set any other environment variables, and depends upon which shell you use. For example:
  
csh/tcsh: `setenv OMP_NUM_THREADS 8`
sh/bash: `export OMP_NUM_THREADS=8`

OpenMP environment variables are discussed in the Environment Variables section.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP Syntax

Fortran: case insensitive
– Add: use omp_lib or include “omp_lib.h”
– Fixed format
  ● Sentinel directive [clauses]
  ● Sentinel could be: !$OMP, *$OMP, c$OMP
– Free format
  ● !$OMP directive [clauses]

C/C++: case sensitive
– Add: #include “omp.h”
  ● #pragma omp directive [clauses] newline

Source: http://en.wikipedia.org/wiki/OpenMP
OpenMP Simple Example

Simple OpenMP example (code.c):

```c
#include <omp.h>
#include <stdio.h>
int main()
{
#pragma omp parallel
printf("Hello World from thread = %d, nthreads = %d\n",
omp_get_thread_num(), omp_get_num_threads());
}
```

Source: http://en.wikipedia.org/wiki/OpenMP
OpenMP Compilation

OpenMP compilation examples:

- **Intel (icc, ifort):** icc -openmp -o omp_hello code.c
- **GNU (gcc, g++, gfortran):** gcc -fopenmp -o omp_hello code.c
- **PGI (pgcc, pgCC, pgf77, pgf90):** pgcc -mp -o omp_hello code.c

Default behavior for number of threads (when OMP_NUM_THREADS not set):
- One thread for PGI.
- For Intel/GNU, as many threads as available cores.

Source: http://en.wikipedia.org/wiki/OpenMP
OpenMP Execution Example

OpenMP execution example:
$ export OMP_NUM_THREADS=3
$ ./omp_hello
Hello World from thread = 0, nthreads = 3
Hello World from thread = 2, nthreads = 3
Hello World from thread = 1, nthreads = 3

Source: http://en.wikipedia.org/wiki/OpenMP
PARALLEL Region Construct

Purpose:
- A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.

```
#pragma omp parallel [clause ...] newline
  if (scalar_expression)
  private (list)
  shared (list)
  default (shared | none)
  firstprivate (list)
  reduction (operator: list)
  copyin (list)
  num_threads (integer-expression)
```

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP C/C++ Directives

Notes:

- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.
- If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined. (In OpenMP 4.0, error handling is better defined.)

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP C/C++ Directives

How Many Threads?
- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  1. Evaluation of the IF clause
  2. Setting of the NUM_THREADS clause
  3. Use of the `omp_set_num_threads()` library function
  4. Setting of the OMP_NUM_THREADS environment variable
  5. Implementation default - usually the number of CPUs on a node, though it could be dynamic (see next bullet).
- Threads are numbered from 0 (master thread) to N-1

Dynamic Threads:
- Use the `omp_get_dynamic()` function to determine if dynamic threads are enabled.
- If supported, the two methods available for enabling dynamic threads are:
  1. The `omp_set_dynamic()` library routine
  2. Setting of the OMP_DYNAMIC environment variable to TRUE

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP C/C++ Directives

Clauses:
- **IF** clause: If present, it must evaluate to .TRUE. (Fortran) or non-zero (C/C++) in order for a team of threads to be created. Otherwise, the region is executed serially by the master thread.
- The remaining clauses are described in detail later, in the Data Scope Attribute Clauses section.

Restrictions:
- A parallel region must be a structured block that does not span multiple routines or code files
- It is illegal to branch (goto) into or out of a parallel region
- Only a single IF clause is permitted
- Only a single NUM_THREADS clause is permitted
- A program must not depend upon the ordering of the clauses

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP C/C++ Directives

Work-Sharing Constructs

- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs do not launch new threads.
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP C/C++ Directives

Work-Sharing Constructs

Types of Work-Sharing Constructs:

- **DO / for** - shares iterations of a loop across the team. Represents a type of "data parallelism".

- **SECTIONS** - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".

- **SINGLE** - serializes a section of code

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP C/C++ Directives

Work-Sharing Constructs

Restrictions:

- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
- Work-sharing constructs must be encountered by all members of a team or none at all.
- Successive work-sharing constructs must be encountered in the same order by all members of a team.

Source: https://computing.llnl.gov/tutorials/openMP/
Work-Sharing Constructs

DO / for Directive

Purpose:
- The DO / for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.

Source: https://computing.llnl.gov/tutorials/openMP/
DO / for Directive

- Format:
- #pragma omp for [clause ...] newline
  schedule (type [,chunk])
  ordered
  private (list)
  firstprivate (list)
  lastprivate (list)
  shared (list)
  reduction (operator: list)
  collapse (n)
  nowait

  for_loop

Source: https://computing.llnl.gov/tutorials/openMP/
DO / for Directive

Clauses:

**SCHEDULE**: Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent. For a discussion on how one type of scheduling may be more optimal than others, see [http://openmp.org/forum/viewtopic.php?f=3&t=83](http://openmp.org/forum/viewtopic.php?f=3&t=83).

- **STATIC**: Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads. This is usually the default schedule.

- **DYNAMIC**: Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1. This schedule option has significant overhead.

Source: [https://computing.llnl.gov/tutorials/openMP/](https://computing.llnl.gov/tutorials/openMP/)
DO / for Directive

Clauses:

**SCHEDULE (cont.):**

- **GUIDED:** Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread. The size of the initial block is proportional to:
  
  - number_of_iterations / number_of_threads
  
  Subsequent blocks are proportional to:
  
  - number_of_iterations_remaining / number_of_threads
  
  The chunk parameter defines the minimum block size. The default chunk size is 1.

- **RUNTIME:** The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. Example: export OMP_SCHEDULE="guided, 100"
  
  It is illegal to specify a chunk size for this clause in the source code.

- **AUTO:** The scheduling decision is delegated to the compiler and/or runtime system.

Source: https://computing.llnl.gov/tutorials/openMP/
DO / for Directive

Clauses:

SCHEDULE (cont.):

- **NO WAIT / nowait**: If specified, then threads do not synchronize at the end of the parallel loop.
- **ORDERED**: Specifies that the iterations of the loop must be executed as they would be in a serial program.
- **COLLAPSE**: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

Other Clauses:

- **PRIVATE**: Variable list indicates which variables are private to a thread (each thread has their own copy).
- **SHARED**: Variable list indicates which variables are shared (only one copy). Default.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP C/C++ Directives

Other Clauses (cont.):

- **FIRSTPRIVATE**: Initialize a variable from the serial part of the code (private clause doesn't initialize the variable).
- **LASTPRIVATE**: Thread that executes the ending loop index copies its value to the master (serial) thread.
- Other clauses are described in detail later, in the [Data Scope Attribute Clauses](https://computing.llnl.gov/tutorials/openMP/) section.

Source: https://computing.llnl.gov/tutorials/openMP/
C / C++ - for Directive Example

#include <omp.h>
#define CHUNKSIZE 100
#define N 1000

main ()
{
    int i, chunk;
    float a[N], b[N], c[N];
    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;
    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
        #pragma omp for schedule(dynamic,chunk) nowait
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];
    } /* end of parallel section */
} /* end of main */
OpenMP C/C++ Directives

Combined Parallel Work-Sharing Constructs

- OpenMP provides three directives that are merely conveniences:
  - PARALLEL DO / parallel for
  - PARALLEL SECTIONS
  - PARALLEL WORKSHARE (fortran only)

- For the most part, these directives behave identically to an individual PARALLEL directive being immediately followed by a separate work-sharing directive.

- Most of the rules, clauses and restrictions that apply to both directives are in effect. See the OpenMP API for details.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP Thread Control

Barrier
Each thread waits at the barrier until all threads reach the barrier.

```c
#pragma omp parallel private(myid, istart, iend)
{
  myrange(myid, nthreads, &istart, &iend);
  for(i=istart; i<=iend; i++){
    a[i] = a[i] - b[i];
  }
#pragma omp barrier
dowork(a);
}
```

Single
Similar to Master except runs only on the first thread to reach it.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP Thread Control

Master
A section of code that runs only on the master (thread with rank=0).

```c
#pragma omp parallel private(myid, istart, iend)
{
    myrange(myid, nthreads, global_start, global_end, &istart, &iend);
    for(i=istart; i<=iend; i++){
        a[i] = b[i];
    }
    #pragma omp barrier
    #pragma omp master
    {
        n = global_end - global_start + 1;
        write_size = fwrite(a, 1, n, file_pointer);
    }
    do_work(istart, iend);
}
```

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP Thread Control

Sections/Section
A section of code that is run by only one thread. Sections are performed in parallel.

```c
#pragma omp parallel
{
 #pragma omp sections
 {
    #pragma omp section
    init_field(field);

    #pragma omp section
    check_grid(grid);

   }
}
```

Source: https://computing.llnl.gov/tutorials/openMP/
MSI Hardware (MAXCORES)

The number of cores per node varies with the type of Xeon on that node. We define:

MAXCORES=number of cores/node
e.g. Nehalem=8 (Itasca), Westmere=12, SandyBridge=16, IvyBridge=20, Haswell=24 (Mesabi).

Since MAXCORES will not vary within the PBS job (the Itasca and Mesabi queues are split by Xeon type), you can determine this at the start of the PBS job (in bash):

MAXCORES=`grep "core id" /proc/cpuinfo | wc -l`
export OMP_NUM_THREADS=$MAXCORES
OpenMP Parallel Loop

The following example demonstrates how to parallelize a simple loop using the parallel loop construct. The loop iteration variable is private by default, so it is not necessary to specify it explicitly in a private clause.

```c
void simple(int n, float *a, float *b) {
    int i;
    #pragma omp parallel for
    for (i=1; i<n; i++) /* i is private by default */
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Source: http://openmp.org/mp-documents/openmp-examples-4.0.2.pdf
OpenMP Parallel Loop

Threads share the work in loop parallelism. For example, using 8 threads with n=800 under the default “static” scheduling:
- thread 0 has iterations i=1-100
- thread 1 has iterations i=101-200, etc.

```c
void simple(int n, float *a, float *b)
{
    int i;
    #pragma omp parallel for
    for (i=1; i<n; i++) /* i is private by default */
        b[i] = (a[i] + a[i-1]) / 4.0;
}
```

Source: http://openmp.org/mp-documents/openmp-examples-4.0.2.pdf
OpenMP Parallel Loop

_OPENMP is defined if the code is compiled with the openmp flag.

#ifdef _OPENMP
    #include <omp.h>
#endif
OpenMP Reduction Example

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define SUM_INIT 0
int main()
{
   int i, n = 250;
   int sum, a[n];
   int ref = SUM_INIT + (n-1)*n/2;
   for (i=0; i<n; i++)
      a[i] = i;

Source: http://openmp.org/examples/Using-OpenMP-Examples-Distr.zip
Reduction Example (cont.)

```c
#pragma omp parallel
{
    #pragma omp single
    printf("Number of threads is %d\n",omp_get_num_threads());
}
sum = SUM_INIT;
printf("Value of sum prior to parallel region: %d\n",sum);
#pragma omp parallel for default(none) shared(n,a) reduction(+:sum)
    for (i=0; i<n; i++)
        sum += a[i];
/*-- End of parallel reduction --*/

printf("Value of sum after parallel region: %d\n",sum);
printf("Check results: sum = %d (should be %d)\n",sum,ref);
return(0);
}
```

Source: http://openmp.org/examples/Using-OpenMP-Examples-Distr.zip
OpenMP 4.0

OpenMP 4.0 was released in July 2013.

Intel OpenMP 4.0 features:


OpenMP 4.0 examples:

http://openmp.org/mp-documents/OpenMP_Examples_4.0.1.pdf
OpenMP 4.0 target Directive

OpenMP 4.0 was released in July 2013. The major new features included in OpenMP 4.0 are:

• Support for accelerators. The OpenMP 4.0 specification effort included significant participation by all the major vendors in order to support a wide variety of compute devices.

OpenMP provides mechanisms (target) 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. Currently works for the Intel PHIs using Intel compilers, but not on nVidia GPUs.
OpenMP 4.0 SIMD Directive

• 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 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.

Basic form: `omp simd:` applied to a loop to indicate that multiple iterations of the loop can be executed concurrently using SIMD instructions.

OpenMP 4.0 SIMD Directive

For best efficiency in all cases (serial, OpenMP, MPI, etc.), you should use SIMD (vectors) on the innermost loops. On Haswell, each core is able to retrieve 256 bits per clock in vector mode, but only 64 bits per clock without vectorization. So memory bandwidth (which limits most applications’ performance) is up to 4 times slower without vectorization.

Here is Intel’s video describing how to use the OpenMP 4.0 SIMD directive:

OpenMP 4.0 Error Directives

• Error handling. OpenMP 4.0 defines error handling capabilities to improve the resiliency and stability of OpenMP applications in the presence of system-level, runtime-level, and user-defined errors. Features to abort parallel OpenMP execution cleanly have been defined, based on conditional cancellation and user-defined cancellation points.

cancel directive:

#pragma omp cancel [clause[[],]clause] ...
!$omp cancel [clause[[],]clause] ...
Clauses: parallel, sections, for, do
OpenMP 4.0 Affinity Directive

- **Thread affinity.** OpenMP 4.0 provides mechanisms to define where to execute OpenMP threads. Platform-specific data and algorithm-specific properties are separated, offering a deterministic behavior and simplicity in use. The advantages for the user are better locality, less false sharing and more memory bandwidth. More info here:

http://openmp.org/forum/viewtopic.php?f=3&t=1731

Set environment variable OMP_PLACES to describe a list of places and the hardware threads of each place:

OMP_PLACES = threads | cores | sockets

- threads: place maps to hardware thread.
- cores: place maps to core (may have multiple threads).
- sockets: place maps to socket (may have multiple cores).

e.g.: export OMP_PLACES=cores

The "cores" abstract name should give you one place per physical core.
OpenMP 4.0 Affinity Directive

proc_bind(master|close|spread) clause of parallel to specify policy of assigning OpenMP threads to places. Allows programmer to control the placement of individual loops.

- **master**: All threads of the team go to the place of the master thread.
- **close**: Assign threads to place close to the place of parent thread.
- **spread**: Subpartition parent place sets, and then assign threads to place close to the new places.

```c
void work();
int main()
{
    #pragma omp parallel proc_bind(spread) num_threads(4)
    {
        work();
    }
    return 0;
}
```
Affinity

- Avoid thread migration for better data locality.
All scripts on MSI systems should set env var OMP_PROC_BIND=true
although the current system gcc (4.4.6) does not use it, gcc versions 4.7 and above do use it (Intel compiled apps, too).

Setting OMP_PROC_BIND=true will eliminate cache invalidations due to processes switching to different cores. It will not necessarily force OPTIMAL thread to core binding (to do this you need to set KMP_AFFINITY for Intel compiled apps, GOMP_CPU_AFFINITY for gcc compiled apps). OMP_PROC_BIND was added in OpenMP 3.1.
The Intel® runtime library has the ability to bind OpenMP threads to physical processing units. The interface is controlled using the KMP_AFFINITY environment variable. Depending on the system (machine) topology, application, and operating system, thread affinity can have a dramatic effect on the application speed.

A full description of KMP_AFFINITY is found here:


http://software.intel.com/en-us/node/463446
Affinity

MSI staff found KMP_AFFINITY=compact to be optimal when hyper-threading is not enabled. When hyper-threading is enabled, KMP_AFFINITY=scatter was optimal. Every application has different performance characteristics, so we recommend that you try running your application with these different settings (compact, scatter) to see which is optimal for it.

At MSI, a compiler which currently has OpenMP 4.0 support is the latest Intel/cluster module, loaded using:
% module load intel/cluster
Affinity

The GNU compilers have an environment variable called GOMP_CPU_AFFINITY similar to KMP_AFFINITY, although you must specify the CPU numbers explicitly.

See:

http://gcc.gnu.org/onlinedocs/gcc-4.3.0/libgomp/GOMP_CPU_AFFINITY.html
OpenMP 4.0 Intel Features

Intel OpenMP 4.0 features description (includes a first touch example):

http://openmp.org/sc13/OpenMP4.0_Intro_YonghongYan_SC13.pdf

Source: https://www.nersc.gov/assets/pubs_presos/hybridMPIOpenMP20150323.pdf
OpenMP 4.0 Intel Features

Intel OpenMP 4.0 examples (It includes examples that demonstrate use of the `proc_bind` clause to control thread binding for a team of threads in a parallel region.):

http://openmp.org/wp/2014/02/updated-openmp-40-examples/

Source: https://www.nersc.gov/assets/pubs_presos/hybridMPIOpenMP20150323.pdf
OpenMP Optimization Tips

Memory Affinity: “First Touch” Memory

- Memory affinity: Allocate memory as close as possible to the core on which the task that requested the memory is running.
- Memory affinity is not decided by the memory allocation, but by the initialization. Memory will be local to the thread which initializes it. This is called “first touch” policy.
- Hard to do “perfect touch” for real applications. Instead, use a number of threads fewer than number of cores per NUMA domain.

Source: https://www.nersc.gov/assets/pubs_presos/hybridMPIOpenMP20150323.pdf
OpenMP Optimization Tips

Memory Affinity: “First Touch” Memory Example

Initialization

#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
  a[j] = 1.0;
  b[j] = 2.0;
  c[j] = 0.0;
}

Compute

#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
  a[j] = b[j] + d * c[j];
}

https://www.nersc.gov/assets/pubs_presos/hybridMPIOpenMP20150323.pdf
OpenMP Optimization Tips

- False sharing can cause significant performance degradation. See:


- In general it should be fastest to parallelize the outer loop only, provided there is sufficient parallelism to keep the threads busy and load balanced. Parallelizing the inner loop only adds an overhead for every parallel region encountered which (although dependent on the implementation and the number of threads) is typically of the order of tens of microseconds.
OpenMP References

OpenMP standard, summary cards, examples:  
http://www.openmp.org/

Recommended tutorial:  https://computing.llnl.gov/tutorials/openMP/

MSI hardware descriptions:  https://www.msi.umn.edu/content/hpc
MSI queue descriptions:  https://www.msi.umn.edu/queues

Source: http://openmp.org/mp-documents/openmp-examples-4.0.2.pdf
# Mesabi: Hardware Hierarchy

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>core</td>
<td></td>
<td>2.5 GHz clock; up to 16 *s &amp; *s per clock</td>
</tr>
<tr>
<td>Processor</td>
<td></td>
<td>12 cores; 30 MB Cache memory</td>
</tr>
<tr>
<td>Node</td>
<td></td>
<td>2 processors (sockets) &amp; 64+ GB shared memory</td>
</tr>
<tr>
<td>Level 1</td>
<td>Switch: Leaf</td>
<td>24 nodes (or fewer) 1x EDR to each node</td>
</tr>
<tr>
<td>Level 2</td>
<td>Switch: Island</td>
<td>8 Leafs 6x FDR to each leaf</td>
</tr>
<tr>
<td>Level 3</td>
<td>Switch: Cluster</td>
<td>4 Islands 12x FDR to each island</td>
</tr>
</tbody>
</table>
### MPI: Software Hierarchy

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>Scheduled work in time slices (by the OS)</td>
</tr>
<tr>
<td>Process</td>
<td>1 MPI rank: variables arrays, IO streams, one or more threads</td>
</tr>
<tr>
<td>Application</td>
<td>1 or more Ranks (processes)</td>
</tr>
<tr>
<td></td>
<td>MPI communicators</td>
</tr>
<tr>
<td>Workflow</td>
<td>1 or more applications, scripts, ...</td>
</tr>
</tbody>
</table>
MPI: Arguments For & Against

**Pros:**
- **Performance**
  - Strong Scaling & Cache Coherency
- **scalability**
  - more cores AND more memory
- **flexibility**
  - hardware topology & heterogeneity
- **portability**
  - MPI is a standard not an implementation

**Cons:**
- Need to restructure your code
  - However: possibly not by much
- May need to restructure IO & data formats
  - However: may lead to much faster (parallel) IO
  - Can fall back to Rank 0 doing all IO
Current Architecture Trends

Need a hybrid programming model with three levels of parallelism

– MPI between nodes or sockets (if multiple nodes are required).
– Shared memory (such as OpenMP) on the nodes/sockets.
– Increase vectorization (SIMD) for lower level loops.
Hybrid MPI/OpenMP Applications

See:

https://www.nersc.gov/assets/pubs_presos/hybridMPIOpenMP20150323.pdf

Application Optimization Strategy: A general strategy for performance optimization of an application is:

1. Serial Optimization: Compiler options, profile code (gprof, Intel VTune, Allinea MAP (module allinea-tools/5.0-40932)), etc.
2. Increase vectorization (SIMD) for all vectorizable lower level loops. See: https://software.intel.com/en-us/articles/intel-vectorization-tools
3. Implement shared memory threading (using OpenMP or pthreads) on a node/socket.
4. Implement MPI between nodes or sockets.