OpenMP*: Beyond the Common Core
A hands on tutorial

Tim Mattson
Intel Corp.
timothy.g.mattson@ intel.com

Yun (Helen) He
Berkeley Lab
yhe@lbl.gov

∗ The name "OpenMP" is the property of the OpenMP Architecture Review Board.

Preliminaries: Part 1

- Disclosures
  - The views expressed in this tutorial are those of the people delivering the tutorial.
    - We are <u>not</u> speaking for our employers.
    - We are <u>not</u> speaking for the OpenMP ARB
- We take these tutorials VERY seriously:
  - Help us improve … tell us how you would make this tutorial better.

Preliminaries: Part 2

- Our plan for the day .. Active learning!
  - We will mix short lectures with short exercises.
  - You will use your laptop to connect to a multiprocessor server.
- Please follow these simple rules
  - Do the exercises that we assign and then change things around and experiment.
    - Embrace active learning!
  - Don't cheat: Do Not look at the solutions before you complete an exercise … even if you get really frustrated.

Cori - Cray XC40

- We will use the Cori system for hands on exercises today
- **9,688 Intel Knights Landing** compute nodes
  - 68 cores per node, 4 hardware threads per core
  - Larger vector units (512 bits) with more complex instructions
  - 96 GB DRAM, 16 GB on-package MCDRAM
- **2,388 Intel Xeon Haswell** compute nodes: 32 cores/node
- Cori KNL nodes are integrated with Haswell nodes on Aries network as **one system**
- Choices of Intel, Cray, and GNU compilers

Compile and run on Cori

- Exercises are at
  - % cd $SCRATCH   # or another local directory
  - % cp -r /project/projectdirs/training//OpenMP_May2018 .
  - % cd OpenMP_May2008/C   (or …/Fortran)
  - % make
- The default compiler is Intel. Use compiler wrappers (ftn, cc, and CC) and the OpenMP compiler flag to build, such as:
  - % cc -qopenmp mycode.c
- To use another compiler, such as gcc:
  - % module swap PrgEnv-intel PrgEnv-gnu
  - % cc -fopenmp mycode.c
- To run on a compute node with an interactive batch session:
  - Haswell node: % salloc -N 1 -C haswell -q interactive -t 1:00:00
  - KNL node: % salloc -N 1 -C knl -q interactive -t 1:00:00
  - Pure OpenMP code: % ./a.out
  - Hybrid MPI/OpenMP code: % srun -n .. -c … --cpu_bind=cores ./a.out
  - You will need to set OMP_NUM_THREADS and other OpenMP environment variables when necessary
  - We will also talk about -n, -c settings later in the Affinity section

Outline

- The common core: a quick review
- OpenMP Tasks
- The divide and conquer pattern
- Task group, task loops, and more
- Threadprivate
- The other workshare constructs
- Do across loops
- The OpenMP Memory model
- Point to point synchronization, atomic, and locks
- NUMA systems
- Thread affinity
- Hybrid MPI/OpenMP
- More about process and thread affinity
- A quick survey of the rest of OpenMP

OpenMP[*] overview:

`C$OMP FLUSH`

`#pragma omp critical`

`C$OMP THREADPRIVATE(/ABC/)`

`CALL OMP_SET_NUM_THREADS(10)`

`C$OM`

`C$OM`

`C$`

`ED`

`C`

`#p`

`:$OMP BARRIER`

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes established SMP practice + vectorization and heterogeneous device programming

`C$OMP PARALLEL COPYIN(/blk/)`

`C$OMP DO lastprivate(XX)`

`Nthrds = OMP_GET_NUM_PROCS()`

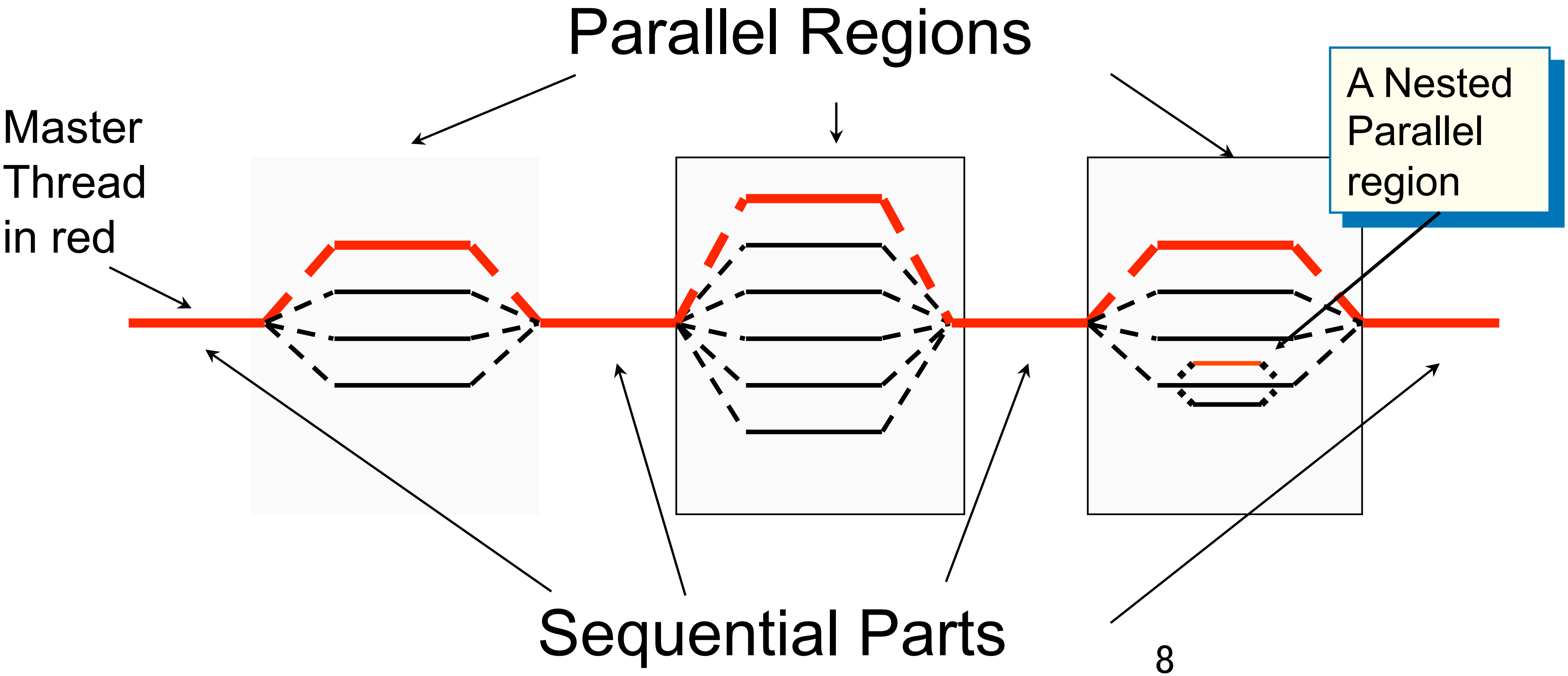
`omp_set_lock(lck)`

[*] The name “OpenMP” is the property of the OpenMP Architecture Review Board.

OpenMP programming model:

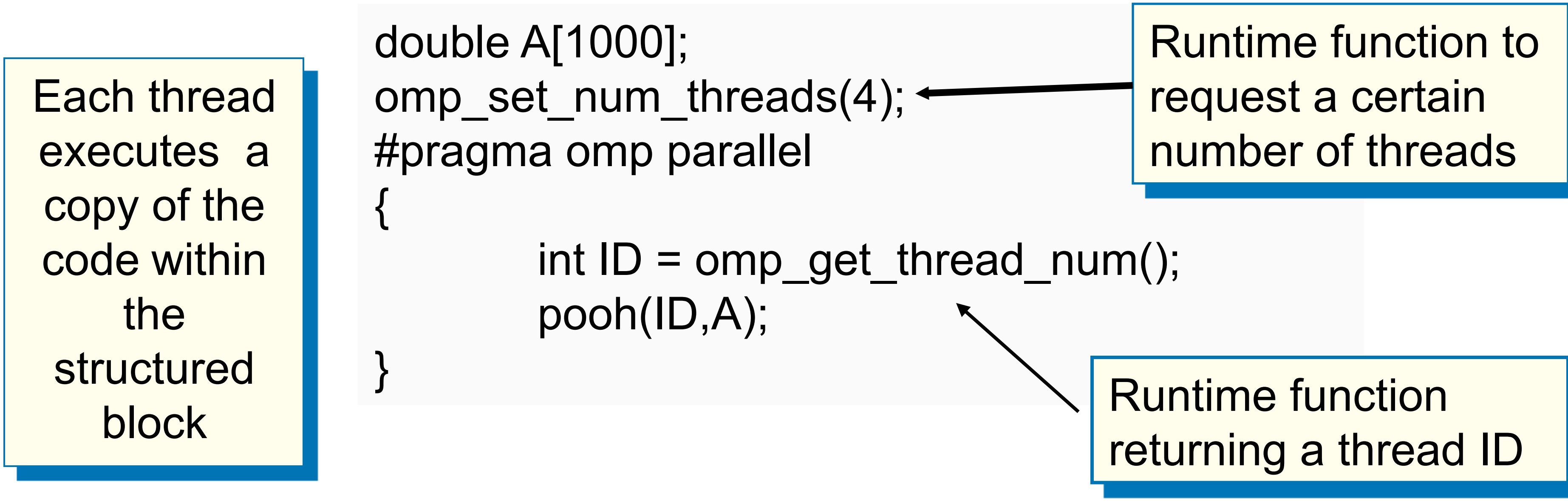
Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



Thread creation: Parallel regions

- You create threads in OpenMP* with the parallel construct.
- For example, To create a 4 thread Parallel region:



```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
            int ID = omp_get_thread_num();
            pooh(ID,A);
}
```

Each thread executes a copy of the code within the structured block

Runtime function to request a certain number of threads

Runtime function returning a thread ID

- Each thread calls pooh(ID,A) for `ID = 0` to `3`

* The name "OpenMP" is the property of the OpenMP Architecture Review Board

The worksharing-loop constructs

- The worksharing-loop construct splits up loop iterations among the threads in a team

```
#pragma omp parallel

{
#pragma omp for
          for (I=0;I<N;I++){
                    NEAT_STUFF(I);
          }
}
```

Worksharing-Loop construct name:

- •C/C++: for
- •Fortran: do

The loop control index I is made “private” to each thread by default.

Threads wait here until all threads are finished with the parallel loop before any proceed past the end of the loop

Reduction

- OpenMP reduction clause:

  reduction (op : list)

- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in “list” must be shared in the enclosing parallel region.

```
 double  ave=0.0, A[MAX];    int i;
#pragma omp parallel for reduction (+:ave)
  for (i=0;i< MAX; i++) {
         ave + = A[i];
  }
  ave = ave/MAX;
```

Do you understand Reduction?

- What does the following code print?

```
int j = 2;
float sum = 1;
float Avec[100];

// Initialize Avec to a set of random values
Initialize(Avec, 100);

#pragma omp parallel for reduction(+:sum)
For (j=0; j<100; j++)
    sum *= Avec[j];

printf(“ sum = %f \n”,(float)sum);
```

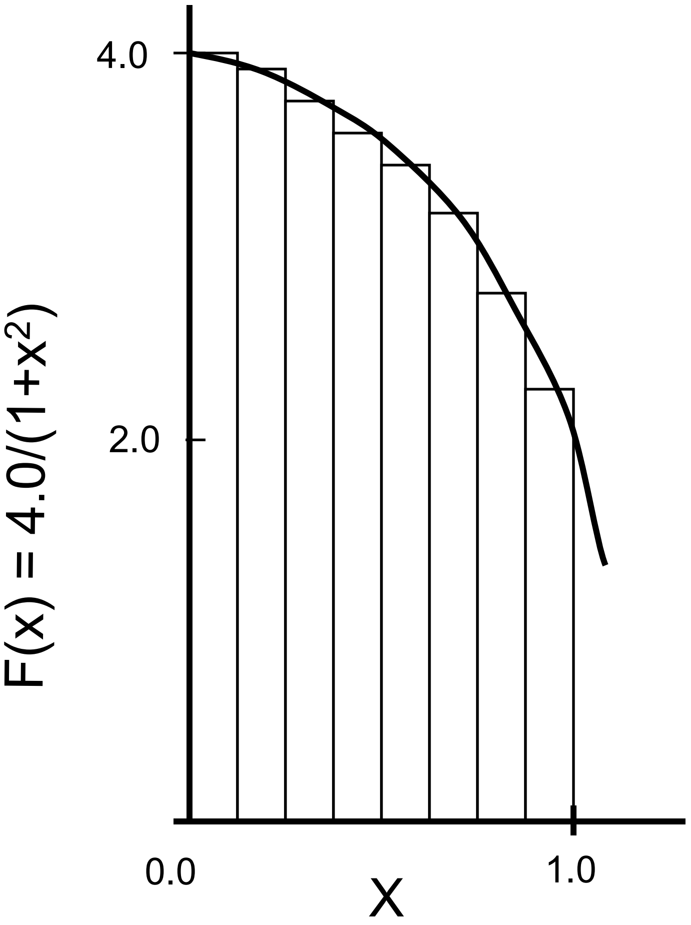
Exercise: Pi with loops and a reduction

- Start with the serial pi program (pi.c) and parallelize it with a worksharing-loop construct
- Your goal is to minimize the number of changes made to the serial program.

```
#pragma omp parallel
#pragma omp for
#pragma omp parallel for
#pragma omp for reduction(op:list)
#pragma omp critical
int omp_get_num_threads();
int omp_get_thread_num();
double omp_get_wtime();
```

Remember: OpenMP makes the loop control index in a loop workshare construct private for you … you don't need to do this yourself

Numerical integration: the pi program



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial PI program

```
static long num_steps = 100000;
double step;
int main ()
{           int i;     double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;

            for (i=0; i< num_steps; i++){
                       x = (i+0.5)*step;
                       sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

Exercise: for more experienced OpenMP programmers

- Consider the program linked.c
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program using anything you choose in OpenMP *other than* tasks.

```
p = listhead ;
while (p) {
  process(p);
  p=next(p) ;
}
```

Assume that items can be processed independently

Example: Pi with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{    int i;          double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps;
      #pragma omp parallel
      {
           double x;
          #pragma omp for reduction(+:sum)
              for (i=0;i< num_steps; i++){
                      x = (i+0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
              }
       }
         pi = step * sum;
}
```

Create a team of threads … without a parallel construct, you'll never have more than one thread

Create a scalar local to each thread to hold value of x at the center of each interval

Break up loop iterations and assign them to threads … setting up a reduction into sum. Note … the loop index is local to a thread by default.

Linked lists without tasks

- See the file Linked_omp25.c

```
while (p != NULL) {
     p = p->next;
      count++;
}
p = head;
for(i=0; i<count; i++) {
      parr[i] = p;
      p = p->next;
   }
#pragma omp parallel
{
     #pragma omp for schedule(static,1)
     for(i=0; i<count; i++)
       processwork(parr[i]);
}
```

Count number of items in the linked list

Copy pointer to each node into an array

Process nodes in parallel with a for loop

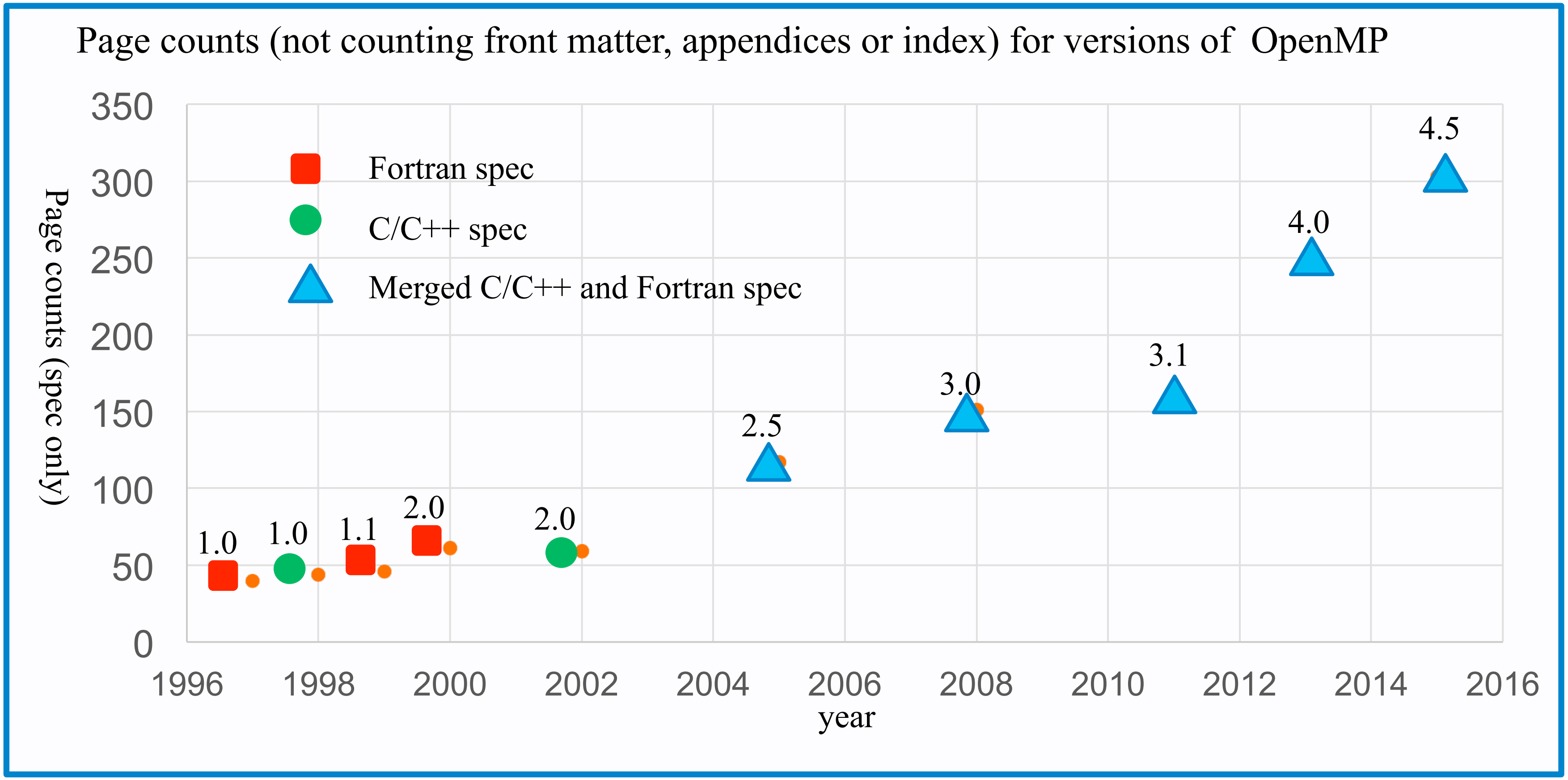
	Default schedule	Static,1
One Thread	48 seconds	45 seconds
Two Threads	39 seconds	28 seconds

Results on an Intel dual core 1.83 GHz CPU, Intel IA-32 compiler 10.1 build 2

The growth of complexity in OpenMP

- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science.

- The complexity has grown considerably over the years!

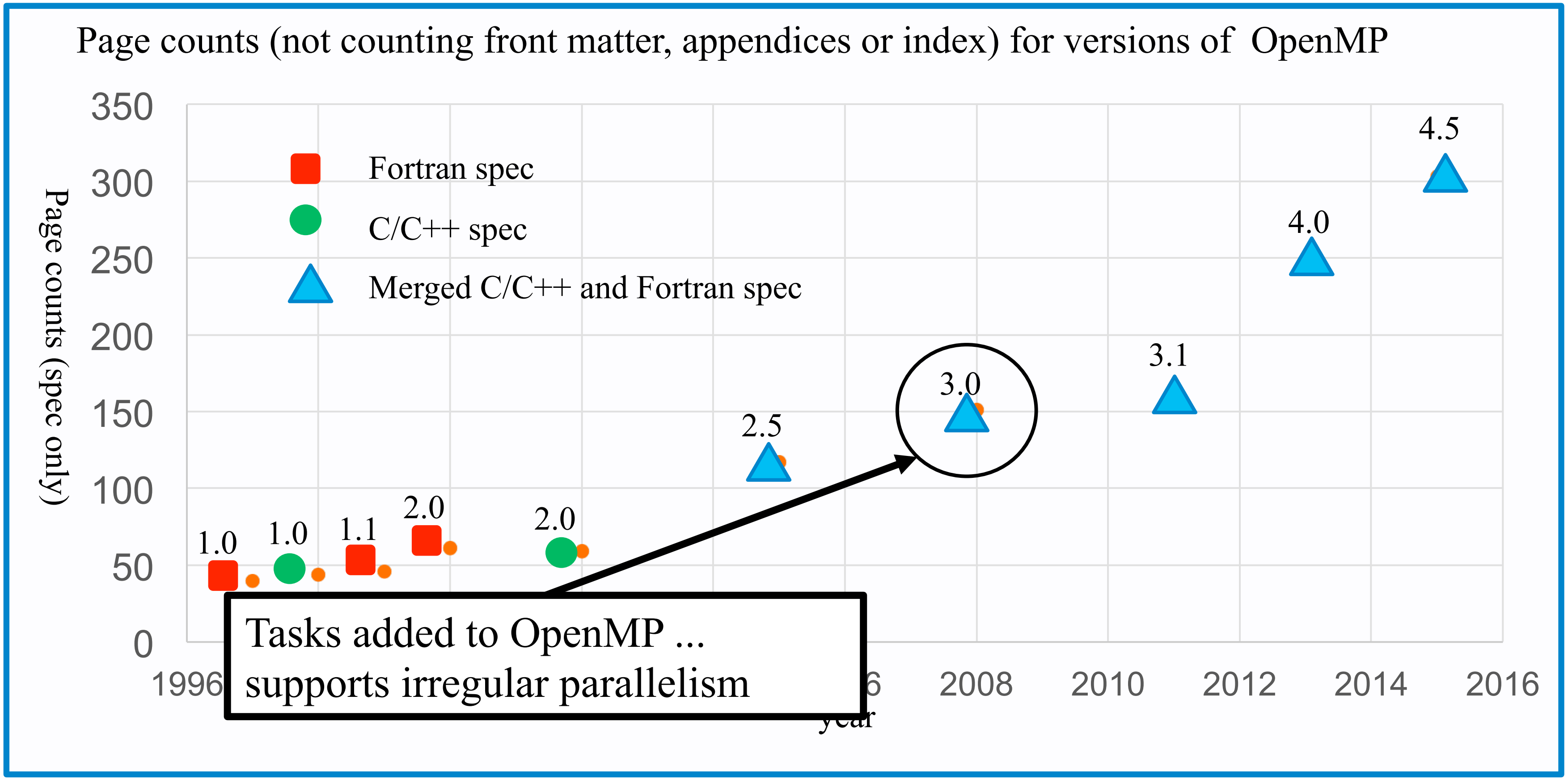


The complexity of the full spec is overwhelming, so we focus on the 16 constructs most OpenMP programmers restrict themselves to … the so called "OpenMP Common Core"

The growth of complexity in OpenMP

- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science.

- The complexity has grown considerably over the years!

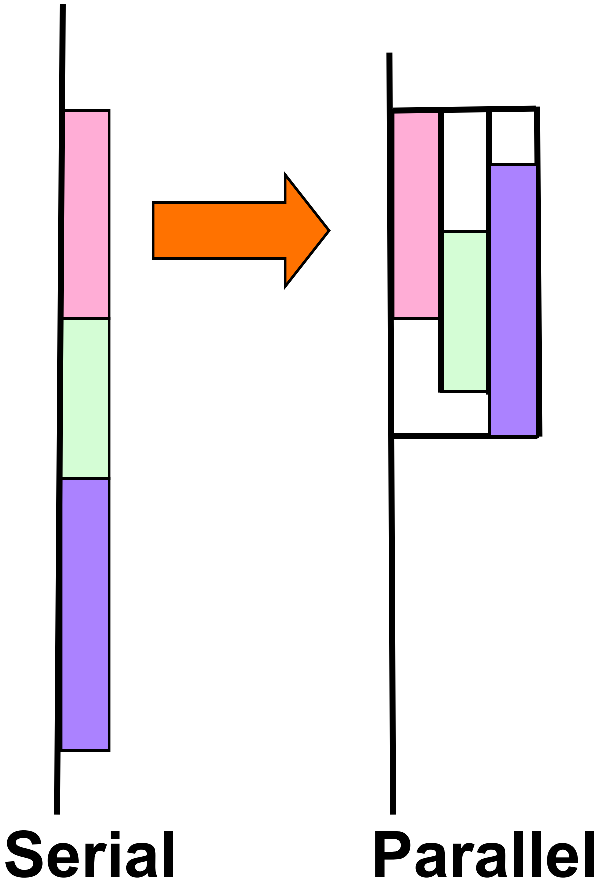


Tasks added to OpenMP ...
supports irregular parallelism

The complexity of the full spec is overwhelming, so we focus on the 19 constructs most OpenMP programmers restrict themselves to … the so called "OpenMP Common Core"

What are tasks?

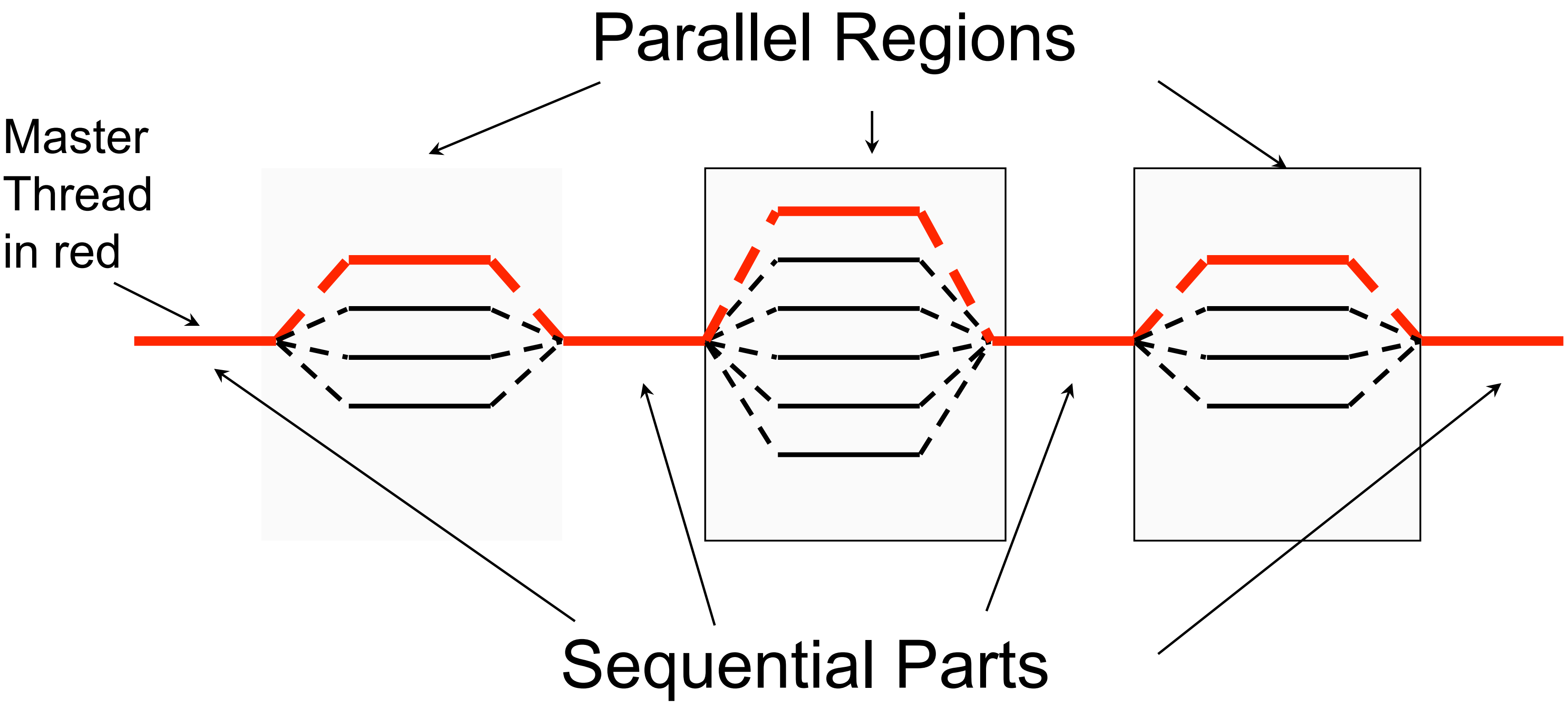
- Tasks are independent units of work
- Tasks are composed of:
  - code to execute
  - data to compute with
- Threads are assigned to perform the work of each task.
  - The thread that encounters the task construct may execute the task immediately.
  - The threads may defer execution until later



Adding tasks to OpenMP required major changes to the specification

Fork-Join Parallelism:

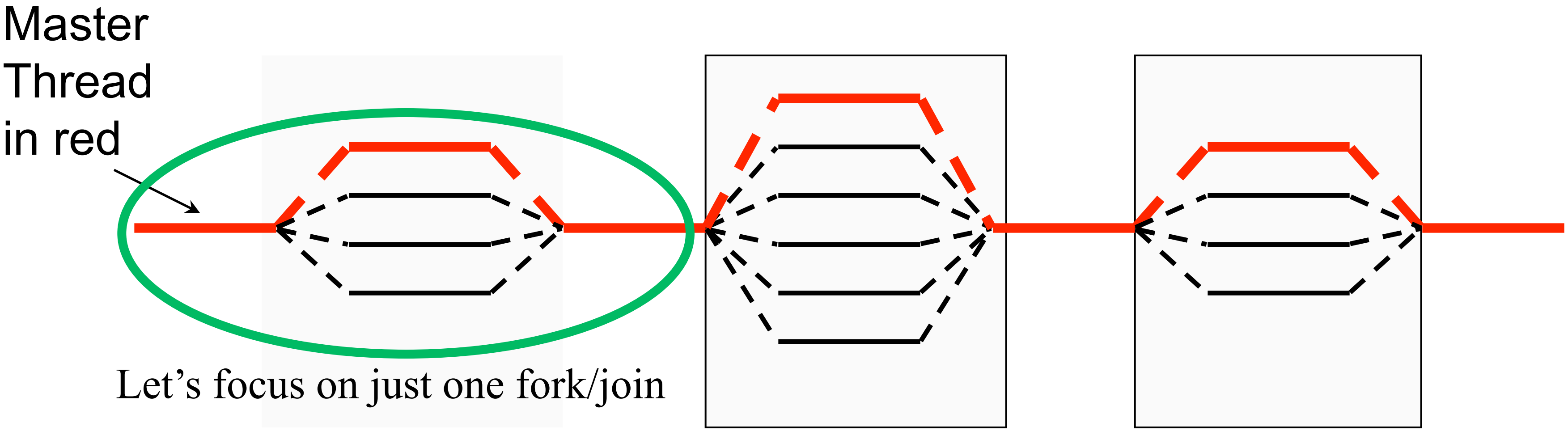
- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



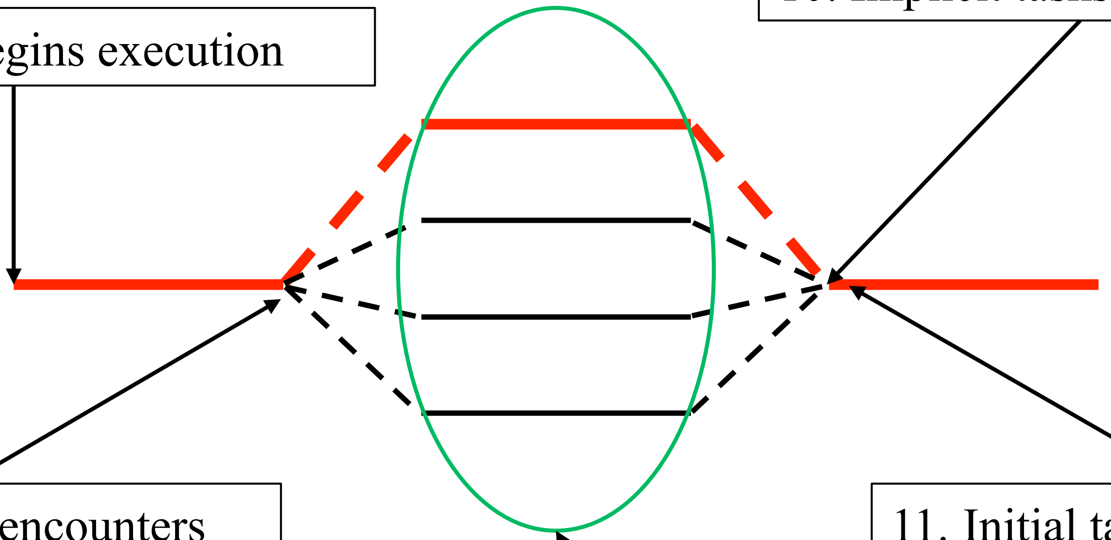
Adding tasks to OpenMP required major changes to the specification

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



Low Level details of OpenMP

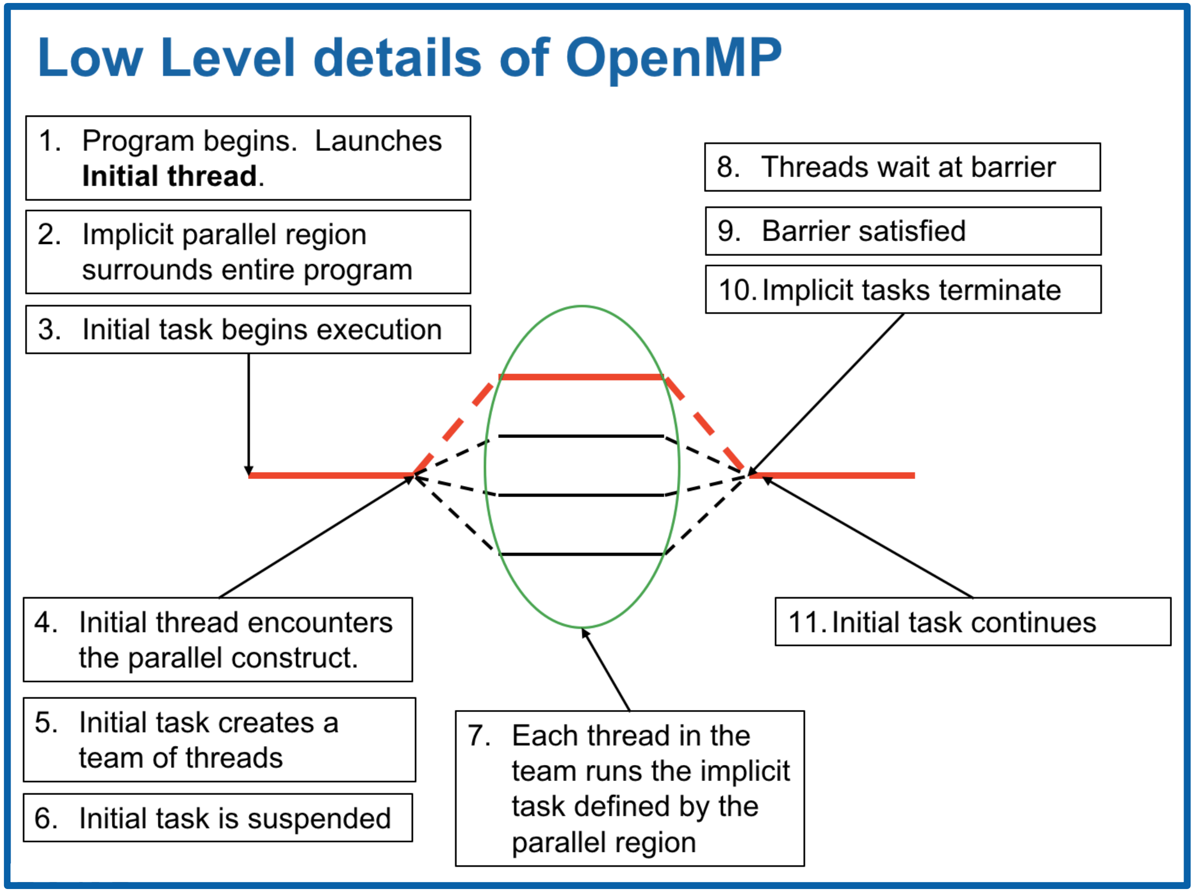


1. Program begins. Launches **Initial thread**.
2. Implicit parallel region surrounds entire program
3. Initial task begins execution
4. Initial thread encounters the parallel construct.
5. Initial task creates a team of threads
6. Initial task is suspended
7. Each thread in the team runs the implicit task defined by the parallel region
8. Threads wait at barrier
9. Barrier satisfied
10. Implicit tasks terminate
11. Initial task continues

Why all this complexity around tasks?

Remember: a language specification is written for people who implement the language … they have ZERO tolerance for ANY ambiguity.

By defining a thread as an execution entity that runs tasks, we can define semantics in terms of tasks and consistently apply them everywhere.



Low Level details of OpenMP

1. Program begins. Launches **Initial thread**.
2. Implicit parallel region surrounds entire program
3. Initial task begins execution
4. Initial thread encounters the parallel construct.
5. Initial task creates a team of threads
6. Initial task is suspended
7. Each thread in the team runs the implicit task defined by the parallel region
8. Threads wait at barrier
9. Barrier satisfied
10. Implicit tasks terminate
11. Initial task continues

While all these initial threads, implicit tasks, and such are confusing to the programmer, they actually make life easier for people who implement OpenMP.

The OpenMP Common Core: Most OpenMP programs only use these 19 items

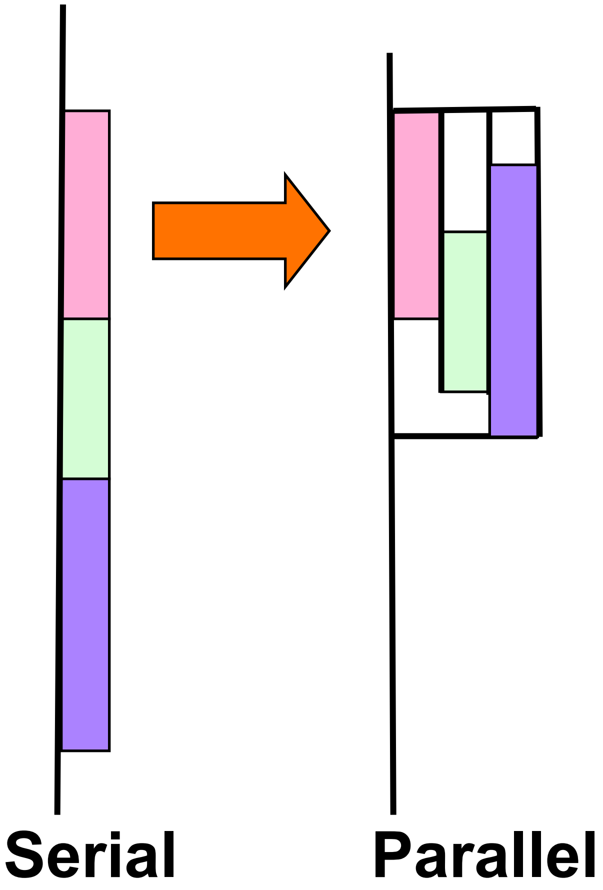
OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads
int omp_get_thread_num() int omp_get_num_threads()	Create threads with a parallel region and split up the work using the number of threads and thread ID
double omp_get_wtime()	Speedup and Amdahl's law. False Sharing and other performance issues
setenv OMP_NUM_THREADS N	Internal control variables. Setting the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies
reduction(op:list)	Reductions of values across a team of threads
schedule(dynamic [,chunk]) schedule (static [,chunk])	Loop schedules, loop overheads and load balance
private(list), firstprivate(list), shared(list)	Data environment
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers. The flush concept (but not the concept)
#pragma omp single	Workshare with a single thread
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.

Outline

- The common core: a quick review
- OpenMP Tasks
- The divide and conquer pattern
- Task group, task loops, and more
- Threadprivate
- The other workshare constructs
- Do across loops
- The OpenMP Memory model
- Point to point synchronization, atomic, and locks
- NUMA systems
- Thread affinity
- Hybrid MPI/OpenMP
- More about process and thread affinity
- A quick survey of the rest of OpenMP

What are tasks?

- Task construct: a structured block of code + a data environment
- Inside a parallel region, a thread encountering a task construct will package up the code block and its data for execution
- The task is executed immediately, or deferred for later execution.
- Tasks can be nested: i.e. a task may itself generate tasks.



A common Pattern is to have one thread create the tasks while the other threads wait at a barrier and execute the tasks

Single worksharing Construct

- The **single** construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implied at the end of the single block (can remove the barrier with a *nowait* clause).

```
#pragma omp parallel
{
        do_many_things();
#pragma omp single
        {   exchange_boundaries();  }
        do_many_other_things();

}
```

Data copying: Copyprivate

Used with a single region to broadcast values of privates from one member of a team to the rest of the team

```
#include <omp.h>
void input_parameters (int, int); // fetch values of input parameters
void do_work(int, int);

void main()
{
   int Nsize, choice;

   #pragma omp parallel private (Nsize, choice)
   {
      #pragma omp single copyprivate (Nsize, choice)
            input_parameters (*Nsize, *choice);

      do_work(Nsize, choice);
   }
}
```

Task Directive

```
#pragma omp task [clauses]
          structured-block
```

```
#pragma omp parallel
{
  #pragma omp single
   {
     #pragma omp task
        fred();
     #pragma omp task
        daisy();
     #pragma omp task
        billy();
   }
}
```

Create some threads

One Thread packages tasks

Tasks executed by some thread in some order

All tasks complete before this barrier is released

When/where are tasks complete?

- At thread barriers (explicit or implicit)
  - applies to all tasks generated in the current parallel region up to the barrier

- At taskwait directive
  - i.e. Wait until all tasks defined in the current task have completed.

    ```
    #pragma omp taskwait
    ```
  - Note: applies only to tasks generated in the current task, not to “descendants” .

Example

```
#pragma omp parallel
{
   #pragma omp single
    {
       #pragma omp task
          fred();
       #pragma omp task
          daisy();
       #pragma taskwait
       #pragma omp task
          billy();
    }
}
```

fred() and **daisy()** must complete before **billy()** starts

The task construct (OpenMP 4.5)

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

Generates an explicit task

where *clause* is one of the following:

if(*[task :]scalar-expression***)**

untied

default(shared | none)

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

final(scalar-expression)

mergeable

depend(*dependence-type* **:** *list***)**

priority(*priority-value***)**

The evolution of the task construct

OpenMP 3.0 (May'08)

OpenMP 3.1 (Jul'11)

OpenMP 4.0 (Jul'13)

OpenMP 4.5 (Nov'15)

The task construct (OpenMP 4.5)

#pragma omp task *[clause[[,]clause]…]*
structured-block

Generates an explicit task

where *clause* is one of the following:

if(*[* **task :***]scalar-expression***)**

untied

default(shared | none)

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

final(scalar-expression)

mergeable

depend(*dependence-type* **:** *list***)**

priority(*priority-value***)**

Consider the data environment associated with a task

The evolution of the task construct

OpenMP 3.0

OpenMP 3.1

OpenMP 4.0

OpenMP 4.5

Data scoping with tasks

- Variables can be shared, private or firstprivate with respect to task
  - If a variable is shared on a task construct, the references to it inside the construct are to the storage with that name at the point where the task was encountered
  - If a variable is private on a task construct, the references to it inside the construct are to new uninitialized storage that is created when the task is executed
  - If a variable is firstprivate on a construct, the references to it inside the construct are to new storage that is created and initialized with the value of the existing storage of that name when the task is encountered

Data scoping defaults

- The behavior you want for tasks is usually firstprivate, because the task may not be executed until later (and variables may have gone out of scope)
  - Variables that are private when the task construct is encountered are firstprivate by default
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared by default

```
#pragma omp parallel shared(A) private(B)
{
    ...
#pragma omp task
    {
        int C;
        compute(A, B, C);
    }
}
```

A is shared
B is firstprivate
C is private

Exercise: Linked List

- Start from your serial linked list program (linked.c)

```
p = listhead ;
while (p) {
  process(p);
  p=next(p) ;
}
```

- Parallelize it using tasks

```
#pragma omp parallel
#pragma omp taskwait
#pragma omp parallel firstprivate(x) shared(y)
#pragma omp task
#pragma omp single
```

Parallel linked list traversal

```
#pragma omp parallel
{
  #pragma omp single
   {
    p = listhead ;
    while (p) {
       #pragma omp task firstprivate(p)
             {
               process (p);
             }
       p=next (p) ;
     }
   }
}
```

Only one thread packages tasks

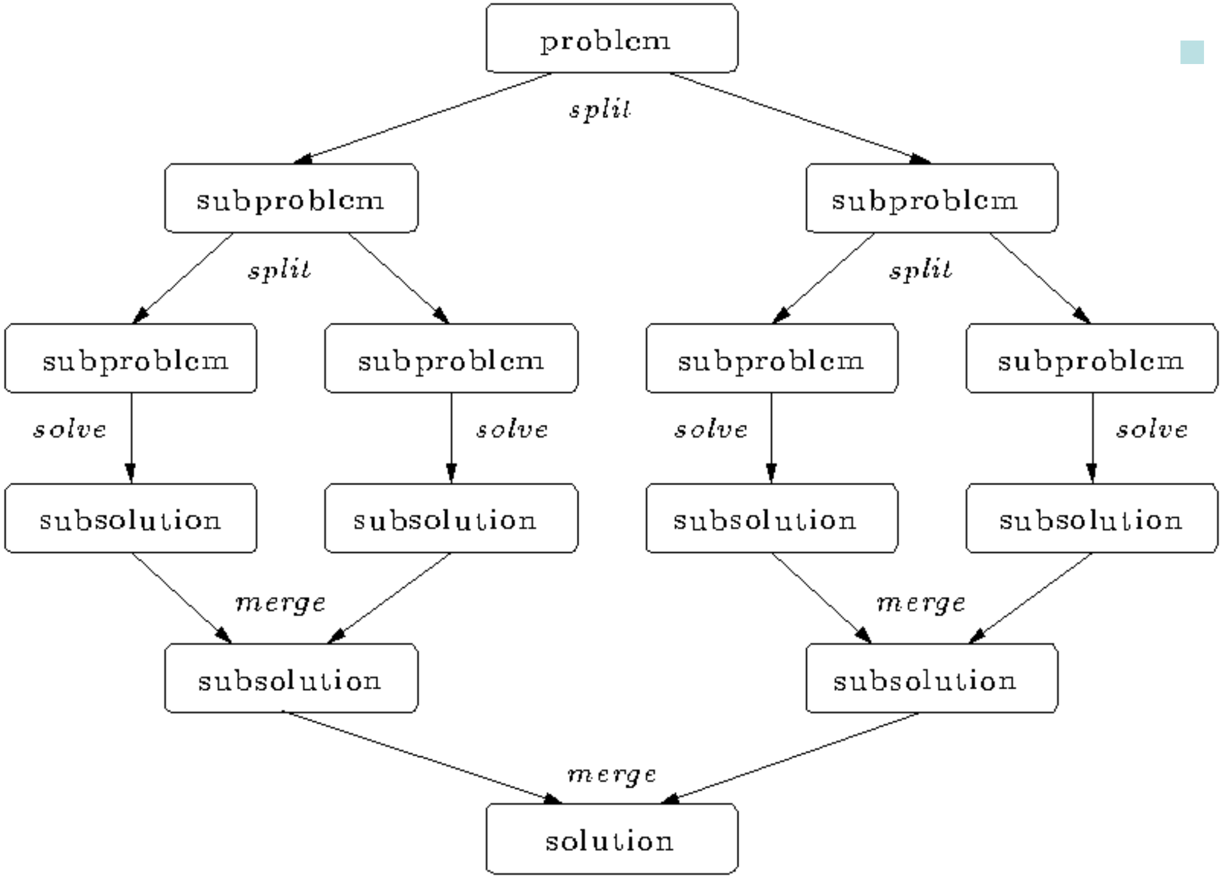
makes a copy of **`p`** when the task is packaged

Outline

- The common core: a quick review
- OpenMP Tasks
- The divide and conquer pattern
- Task group, task loops, and more
- Threadprivate
- The other workshare constructs
- Do across loops
- The OpenMP Memory model
- Point to point synchronization, atomic, and locks
- NUMA systems
- Thread affinity
- Hybrid MPI/OpenMP
- More about process and thread affinity
- A quick survey of the rest of OpenMP

Divide and conquer

- Split the problem into smaller sub-problems; continue until the sub-problems can be solve directly



- 3 Options:
  - □ Do work as you split into sub-problems
  - □ Do work only at the leaves
  - □ Do work as you recombine

Example: Fibonacci numbers
A classic divide and conquer problem

```
int fib (int n)
{
  int x,y;
  if (n < 2) return n;               Direct Solve

  x = fib(n-1);                      Split
  y = fib (n-2);

  return (x+y);                      Merge
}

Int main()
{
  int NW = 5000;
  fib(NW);
}
```

- $F_n = F_{n-1} + F_{n-2}$
- Inefficient $O(n^2)$ recursive implementation!

Parallel Fibonacci

```
int fib (int n)
{   int x,y;
  if (n < 2) return n;

#pragma omp task shared(x)
  x = fib(n-1);
#pragma omp task shared(y)
  y = fib (n-2);
#pragma omp taskwait
  return (x+y);
}

Int main()
{  int NW = 5000;
  #pragma omp parallel
  {
     #pragma omp single
         fib(NW);
  }
}
```

- Binary tree of tasks
- Traversed using a recursive function
- A task cannot complete until all tasks below it in the tree are complete (enforced with taskwait)
- `x,y` are local, and so by default they are private to current task
  - must be shared on child tasks so they don't create their own firstprivate copies at this level!

Parallel Fibonacci again

```
int fib ( int n )
{
int x,y;
   if ( n < 2 ) return n;
#pragma omp task shared(x) if(n>30)
   x = fib(n-1);
#pragma omp task shared(y) if(n>30)
   y = fib(n-2);
#pragma omp taskwait
   return x+y
}
int main()
{  int NN = 5000;
   #pragma omp parallel
   {
       #pragma omp master
          fib(NN);
   }
}
```

Stop creating tasks at some level in the tree.

Exercise: Pi with tasks

- Start from the basic serial pi program, pi.c or pi.f
  - First create a serial divide-and-conquer/recursive solution.
  - Parallelize the recursive program using OpenMP tasks

```
#pragma omp parallel
#pragma omp task
#pragma omp taskwait
#pragma omp single
double omp_get_wtime()
int omp_get_thread_num();
int omp_get_num_threads();
```

Hints:

- Think carefully about what you want the direct solve case to be.
- Make life easy on yourself for the splitting and specialize to a Power-of-two number of steps

Program: OpenMP tasks

```
#include <omp.h>
static long num_steps = 1024*1024;
#define MIN_BLK  1024
double pi_comp(int Nstart,int Nfinish,double step)
{   int i,iblk;
   double x, sum = 0.0,sum1, sum2;
   if (Nfinish-Nstart < MIN_BLK){
      for (i=Nstart;i< Nfinish; i++){
         x = (i+0.5)*step;
         sum = sum + 4.0/(1.0+x*x);
      }
   }
   else{
      iblk = Nfinish-Nstart;
      #pragma omp task shared(sum1)
           sum1 = pi_comp(Nstart,         Nfinish-iblk/2,step);
      #pragma omp task shared(sum2)
            sum2 = pi_comp(Nfinish-iblk/2, Nfinish,       step);
      #pragma omp taskwait
         sum = sum1 + sum2;
   }return sum;
}
```

```
int main ()
{
  int i;
  double step, pi, sum;
   step = 1.0/(double) num_steps;
   #pragma omp parallel
   {
      #pragma omp single
         sum =
             pi_comp(0,num_steps,step);
   }
     pi = step * sum;
}
```

Results*: pi with tasks

threads	1st SPMD	SPMD critical	PI Loop	Pi tasks
1	1.86	1.87	1.91	1.87
2	1.03	1.00	1.02	1.00
3	1.08	0.68	0.80	0.76
4	0.97	0.53	0.68	0.52

*Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core^{TM} i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Using tasks

- Don’t use tasks for things already well supported by OpenMP
  - e.g. standard do/for loops
  - the overhead of using tasks is greater

- Don’t expect miracles from the runtime
  - best results usually obtained where the user controls the number and granularity of tasks

Outline

- The common core: a quick review
- OpenMP Tasks
- The divide and conquer pattern
- Task group, task loops, and more
- Threadprivate
- The other workshare constructs
- Do across loops
- The OpenMP Memory model
- Point to point synchronization, atomic, and locks
- NUMA systems
- Thread affinity
- Hybrid MPI/OpenMP
- More about process and thread affinity
- A quick survey of the rest of OpenMP

Task definitions

- Task: a specific instance of executable code and its data environment.
- Task region: all the code encountered during the execution of a task.
- When a task construct is encountered by a thread, the generated task may be:
  - Deferred: executed by some thread independently of generating task.
  - Undeferred: completes execution before the generating task continues.
  - Included: Undeferred and executed by the thread that encounters the task construct.
- Tasks once started may suspend, wait, and restart.
  - Tied tasks: if a thread is suspended, the same thread will restart the thread at a later time.
  - Untied tasks: if a task is suspended, any thread in the binding team may restart the thread at a later time.

The task construct (OpenMP 4.5)

#pragma omp task *[clause[[,]clause]...]*
structured-block

Generates an explicit task

where *clause* is one of the following:

if(*[* **task :***]scalar-expression***)**

untied

default(shared | none)

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

final(scalar-expression)

mergeable

depend(*dependence-type* **:** *list***)**

priority(*priority-value***)**

The evolution of the task construct

OpenMP 3.0

OpenMP 3.1

OpenMP 4.0

OpenMP 4.5

Task dependencies

!$omp task depend（*type*：*list*）

where *type* is in, out or inout and *list* is a list of variables.

- list may contain subarrays: OpenMP 4.0 includes a syntax for C/C++
- in: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an out or inout clause
- out or inout: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an in, out or inout clause

Task dependencies example

```
#pragma omp task depend (out:a)
  { ... } //writes a
#pragma omp task depend (out:b)
  { ... } //writes b
#pragma omp task depend (in:a,b)
  { ... } //reads a and b
```

- The first two tasks can execute in parallel
- The third task cannot start until the first two are complete

The task construct: the newer/rarely used clauses

untied

The created task, if suspended, can be executed by a different thread

final(scalar-expression)

If the scalar-expression is true, generated tasks are undeferred and execute immediately by the encountering thread.

mergeable

The task is mergable if it is undeferred and included (i.e. uses the parent tasks data environment).

priority(*priority-value*)

Gives a hint to the compiler to schedule tasks with a larger priority value (>0) before tasks with a lower value.

OpenMP 3.0 **OpenMP 3.1** **OpenMP 4.0** **OpenMP 4.5**

Waiting for tasks to complete

```
#pragma omp taskwait
```

OpenMP 3.0

Causes current task region to suspend and wait for completion of all the child tasks created before the taskwait to complete

- A standalone directive
- Defines a task scheduling point

```
#pragma omp taskgroup
          structured-block
```

OpenMP 4.0

A thread encounters the taskgroup construct. It executes the code in the structured block.

That thread suspends and waits at the end of the taskgroup region until all child tasks and any of their descendant tasks are complete.

Task switching

- Consider the following example ... Where the program may generate so many tasks that the internal data structures managing tasks overflow.

```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
    #pragma omp task
      process(item[i]);
}
```

- Solution … Task switching; Threads can switch to other tasks at certain points called ***task scheduling*** points.
- With Task switching, a thread can
  - Execute an already generated task … to “drain the task pool”
  - Execute the encountered task immediately (instead of deferring task execution for later)

Explicit task scheduling

#pragma omp taskyield

OpenMP 3.1

Tells the OpenMP runtime that the current task can be suspended in favor of execution of a different task

- A standalone directive
- Defines an explicit task scheduling point

A function that only one task at a time can execute (mutual exclusion)

```
#include <omp.h>
void something_useful ( void );
void mutual_excl_op( void );
void foo ( omp_lock_t * lock, int n )
{   for (int  i = 0; i < n; i++ )
    #pragma omp task
    {    something_useful();
         while ( !omp_test_lock(lock) ) {
                #pragma omp taskyield
         }
         mutual_excl_op();
         omp_unset_lock(lock);
    }
}
```

Grab a lock if you can, return if you can't

Tell the runtime it can suspend current task and schedule another

Release the lock that protected mutual_excl_op()

Task scheduling Points

- Task switching can only occur at Task Scheduling points.
- Task scheduling points happen …
  - After generation of an explicit task
  - After completion of a task region
  - In a taskyield region
  - In a taskwait region
  - At the end of a taskgropup or barrier
  - In and around regions associated with target constructs (not discussed here).
- At a task scheduling point, *any of* the following can happen for any tasks bound to the current team
  - Begin execution of a tied or untied task
  - Resume any suspended task (tied or untied)

Task Scheduling Details

- An included task is executed immediately after generation of the task
- Scheduling of new tied tasks is constrained by the set of task regions that are currently tied to the thread, and that are not suspended in a barrier region.
  - If this set is empty, any new tied task may be scheduled.
  - Otherwise, a new tied task may be scheduled only if it is a descendent task of every task in the set.
- A dependent task shall not be scheduled until its task dependences are fulfilled.
- When an explicit task is generated by a construct containing an if clause for which the expression evaluated to false, and the previous constraints are already met, the task is executed immediately after generation of the task.

Task Execution around task scheduling points

- Think of a task as a set of “task regions” between task scheduling points
- Each “task region” executes uninterrupted from start to end in the order they are encountered.
- A correct program must behave correctly and consistently with all conceivable scheduling sequences that are compatible with the rules above.
  - If multiple “task regions” between scheduling points modify values in threadprivate storage, a data race is produced and the state of threadprivate storage is not defined.
  - Lock acquire and release in different task regions may break program-order lock protocols and deadlock.

The taskloop construct (OpenMP 4.5)

#pragma omp taskloop *[clause[[,]clause]…]*
structured-block

where *clause* is one of the following:

if([taskloop :]scalar-expr)

shared(list)

private(list)

firstprivate(list)

lastprivate(list)

default(shared | none)

grainsize(grain-size)

num_tasks(num-tasks)

collapse(n)

final(scalar-expr)

priority(priority-value)

untied

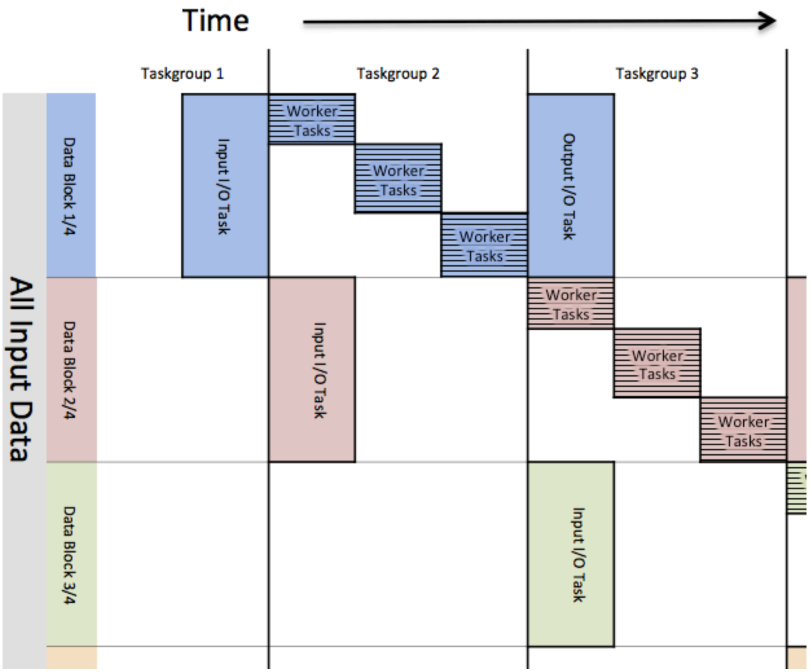
mergeable

nogroup

- The structured block contains loops in the standard form
- Loop iterations are turned into tasks that execute within a taskgroup (unless the nogroup clause is present)
- Grainsize specifies the number of iterations per task
- Num_tasks stipulates the number of tasks to create (unless there are too few loop iterations)

HMMER3: task and taskgroup to Overlap I/O and Compute

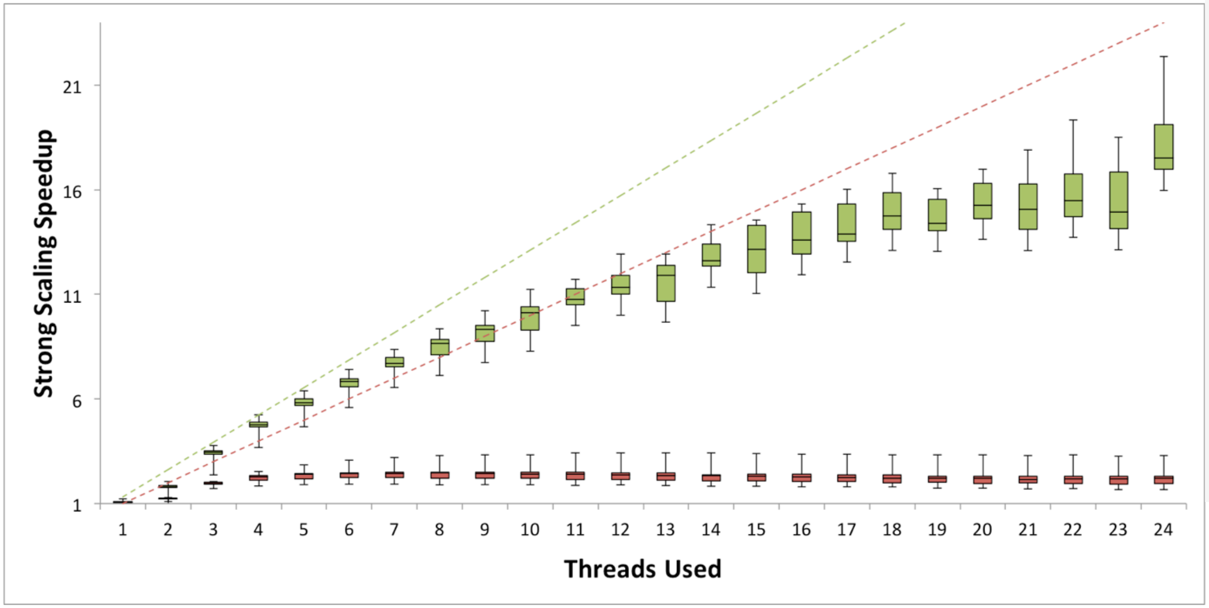
```
#pragma omp parallel {
    #pragma omp single {
        #pragma omp task { load_seq_buffer(); }
        #pragma omp task { load_hmm_buffer(); }
        #pragma omp taskwait
        while( more HMMs ) {
            #pragma omp task { write_output();
                              load_hmm_buffer(); }
            while( more sequences ) {
                #pragma omp taskgroup {
                    #pragma omp task
                    { load_seq_buffer(); }
                    for ( each hmm in hmm_buffer )
                        #pragma omp task
                        { task_kernel(); }
                    swap_I/
                    O_and_working_seq_buffers();
                }
            }
            #pragma omp taskwait
            swap_I/O_and_working_hmm_buffers();
        }
    }
}
```



Courtesy of William Arndt, NERSC

HMMER3: use OpenMP task directives

- Replace pthread implementation limited by performance of master thread
  - OpenMP tasks facilitate overlap of I/O and Compute
  - Forking of child tasks and task groups allow simple work stealing implementation
- Thread scaling result on 1 Edison node (24 cores of Intel Xeon Ivy Bridge)



- pthread HMMER3 Red
- OpenMP HMMER3 Green

- Dashed lines show theoretical peak (two lines because serial performance is also improved)

Courtesy of Willaim Arndt, NERSC

Outline

- The common core: a quick review
- OpenMP Tasks
- The divide and conquer pattern
- Task group, task loops, and more
- Threadprivate
- The other workshare constructs
- Do across loops
- The OpenMP Memory model
- Point to point synchronization, atomic, and locks
- NUMA systems
- Thread affinity
- Hybrid MPI/OpenMP
- More about process and thread affinity
- A quick survey of the rest of OpenMP

Data sharing: Threadprivate

- Makes global data private to a thread
  - Fortran: **COMMON** blocks
  - C: File scope and static variables, static class members
- Different from making them **PRIVATE**
  - with **PRIVATE** global variables are masked.
  - THREADPRIVATE preserves global scope within each thread
- Threadprivate variables can be initialized using **COPYIN** or at time of definition (using language-defined initialization capabilities)

A threadprivate example (C)

Use threadprivate to create a counter for each thread.

```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
    counter++;
    return (counter);
}
```

Data copying: Copyin

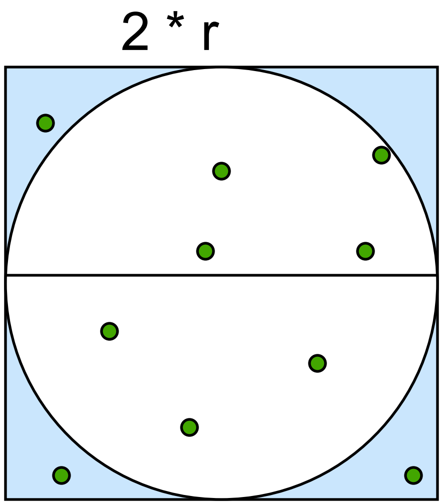
You initialize threadprivate data using a copyin clause.

```
      parameter (N=1000)
      common/buf/A(N)
!$OMP THREADPRIVATE(/buf/)

!$ Initialize the A array
      call init_data(N,A)

!$OMP PARALLEL COPYIN(A)

 … Now each thread sees threadprivate array A initialized
 … to the global value set in the subroutine init_data()

!$OMP END PARALLEL

end
```

Exercise: Monte Carlo calculations

Using random numbers to solve tough problems

- Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
- Example: Computing π with a digital dart board:



N= 10	π = 2.8
N=100	π = 3.16
N= 1000	π = 3.148

- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:

  $A_c = r^2 * \pi$

  $A_s = (2*r) * (2*r) = 4 * r^2$

  $P = A_c/A_s = \pi /4$
- Compute π by randomly choosing points; π is four times the fraction that falls in the circle

Exercise: Monte Carlo pi (cont)

- We provide three files for this exercise
  - pi_mc.c: the Monte Carlo method pi program
  - random.c: a simple random number generator
  - random.h: include file for random number generator
- Create a parallel version of this program without changing the interfaces to functions in random.c
  - This is an exercise in modular software … why should a user of your random number generator have to know any details of the generator or make any changes to how the generator is called as they move to a multithreaded program?
  - The random number generator must be thread-safe.
- Extra Credit:
  - Make your random number generator numerically correct (non-overlapping sequences of pseudo-random numbers).

Parallel Programmers love Monte Carlo algorithms

Embarrassingly parallel: the parallelism is so easy its embarrassing.

Add two lines and you have a parallel program.

```
#include “omp.h”
static long num_trials = 10000;
int main ()
{
   long i;      long Ncirc = 0;       double pi, x, y;
   double r = 1.0;   // radius of circle. Side of squrare is 2*r
   seed(0,-r, r);  // The circle and square are centered at the origin
   #pragma omp parallel for private (x, y) reduction (+:Ncirc)
   for(i=0;i<num_trials; i++)
   {
      x = random();         y = random();
      if ( x*x + y*y) <= r*r)   Ncirc++;
    }

    pi = 4.0 * ((double)Ncirc/(double)num_trials);
    printf("\n %d trials, pi is %f \n",num_trials, pi);
}
```

Linear Congruential Generator (LCG)

- LCG: Easy to write, cheap to compute, portable, OK quality

```
random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
random_last = random_next;
```

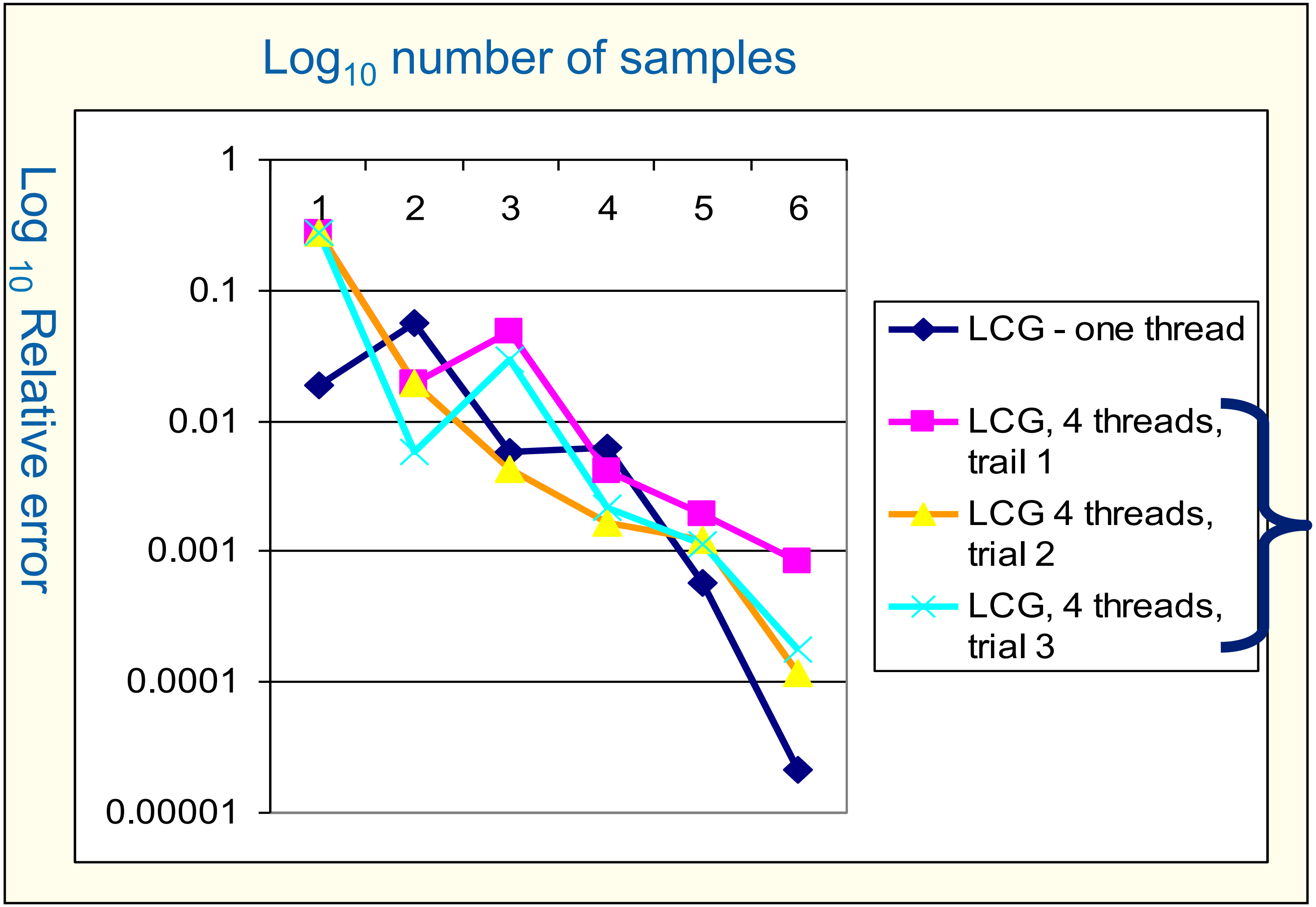
- If you pick the multiplier and addend correctly, LCG has a period of PMOD.
- Picking good LCG parameters is complicated, so look it up (Numerical Recipes is a good source). I used the following:
  - MULTIPLIER = 1366
  - ADDEND = 150889
  - PMOD = 714025

LCG code

```
static long MULTIPLIER  = 1366;
static long ADDEND      = 150889;
static long PMOD        = 714025;
long random_last = 0;
double random ()
{
    long random_next;

    random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
    random_last = random_next;

   return  ((double)random_next/(double)PMOD);
}
```

Seed the pseudo random sequence by setting random_last

Running the PI_MC program with LCG generator



Run the same program the same way and get different answers!

That is not acceptable!

Issue: my LCG generator is not threadsafe

Program written using the Intel C/C++ compiler (10.0.659.2005) in Microsoft Visual studio 2005 (8.0.50727.42) and running on a dual-core laptop (Intel T2400 @ 1.83 Ghz with 2 GB RAM) running Microsoft Windows XP.

LCG code: threadsafe version

```
static long MULTIPLIER  = 1366;
static long ADDEND      = 150889;
static long PMOD        = 714025;
long random_last = 0;
#pragma omp threadprivate(random_last)
double random ()
{
    long random_next;

    random_next = (MULTIPLIER  * random_last + AD
    random_last = random_next;

   return  ((double)random_next/(double)PMOD);
}
```

random_last carries state between random number computations,

To make the generator threadsafe, make random_last threadprivate so each thread has its own copy.

Outline

- The common core: a quick review
- OpenMP Tasks
- The divide and conquer pattern
- Task group, task loops, and more
- Threadprivate
- The other workshare constructs
- Do across loops
- The OpenMP Memory model
- Point to point synchronization, atomic, and locks
- NUMA systems
- Thread affinity
- Hybrid MPI/OpenMP
- More about process and thread affinity
- A quick survey of the rest of OpenMP

The loop worksharing constructs

- The loop worksharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel

{
#pragma omp for
          for (I=0;I<N;I++){
                    NEAT_STUFF(I);
          }
}
```

The variable I is made "private" to each thread by default. You could do this explicitly with a "private(I)" clause

Loop construct name:

- C/C++: for
- Fortran: do

Loop worksharing constructs:
The schedule clause

- The schedule clause affects how loop iterations are mapped onto threads
  - schedule(static [,chunk])
    - Deal-out blocks of iterations of size “chunk” to each thread.
  - schedule(dynamic[,chunk])
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
  - schedule(guided[,chunk])
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
  - schedule(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library).
  - schedule(auto)
    - Schedule is left up to the runtime to choose (does not have to be any of the above).

OpenMP 4.5 added modifiers monotonic, nonmontonic and simd.

loop work-sharing constructs:

The schedule clause

Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead
AUTO	When the runtime can "learn" from previous executions of the same loop

Least work at runtime : scheduling done at compile-time

Most work at runtime : complex scheduling logic used at run-time

Nested loops

- **For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the collapse clause:**

```
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
   for (int j=0; j<M; j++) {
             .....
   }
}
```

Number of loops to be parallelized, counting from the outside

- Will form a single loop of length NxM and then parallelize that.
- Useful if N is O(no. of threads) so parallelizing the outer loop makes balancing the load difficult.

Sections worksharing Construct

- The *Sections* worksharing construct gives a different structured block to each thread.

```
#pragma omp parallel
{

  #pragma omp sections
  {
  #pragma omp section
        X_calculation();
  #pragma omp section
        y_calculation();
  #pragma omp section
        z_calculation();
  }

}
```

By default, there is a barrier at the end of the “omp sections”. Use the “nowait” clause to turn off the barrier.

Array sections with reduce

```
#include <stdio.h>
#define N 100
void init(int n, float (*b)[N]);
int main(){
int i,j; float a[N], b[N][N]; init(N,b);
for(i=0; i<N; i++)  a[i]=0.0e0;

#pragma omp parallel for reduction(+:a[0:N]) private(j)
for(i=0; i<N; i++){
   for(j=0; j<N; j++){
          a[j] += b[i][j];
   }
}
printf(" a[0] a[N-1]: %f %f\n", a[0], a[N-1]);
return 0;
```

Works the same as any other reduce … a private array is formed for each thread, element wise combination across threads and then with original array at the end

Outline

- The common core: a quick review
- OpenMP Tasks
- The divide and conquer pattern
- Task group, task loops, and more
- Threadprivate
- The other workshare constructs
- Do across loops
- The OpenMP Memory model
- Point to point synchronization, atomic, and locks
- NUMA systems
- Thread affinity
- Hybrid MPI/OpenMP
- More about process and thread affinity
- A quick survey of the rest of OpenMP

Parallel loop with ordered region

- An **ordered clause** on a loop worksharing construct
  - indicates that the loop contains an ordered region

- The **ordered construct** defines an ordered region
  - The Statements in ordered region execute in iteration order

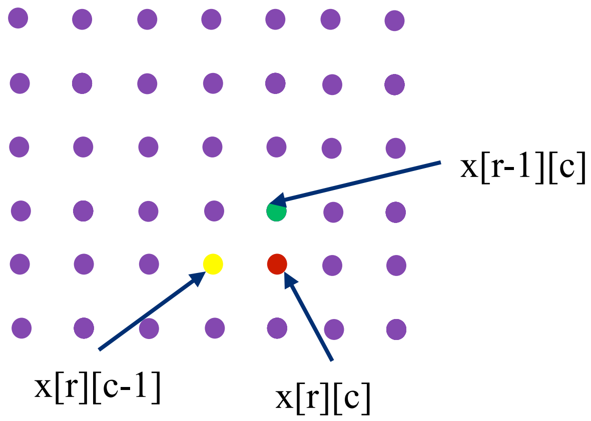
```
#pragma omp for ordered
    for (i=0; i<N; i++) {
      float res = work(i);
      #pragma omp ordered
      {
        printf("result for %d was %f\n", i, res);
        fflush(stdout);
      }
    }
```

Parallelizing nested loops

- Will these nested parallel loops execute correctly?

```
#pragma omp parallel for collapse(2)
for (r=1; r<N; r++) {
   for (c=1; c<N; c++) {

    x[r][c] += fn(x[r-1][c], x[r][c-1]);

  }
}
```

An array section of x



- Pattern of dependencies between elements of x prevent straightforward parallelization
- is there a way to manage the synchronization so we can parallelize this loop?

Ordered stand-alone directive

- Specifies cross-iteration dependencies in a doacross loop nest … i.e. loop level parallelism over nested loops with a regular pattern of synchronization to manage dependencies.

```
#pragma omp ordered depend(sink : vec)
#pragma omp ordered depend(source)
```

vec is a comma separated list of decencies … one per loop involved in the dependencies

- **Depend** clauses specify the order the threads execute **ordered** regions.
  - The **sink** *dependence-type*
    - specifies a cross-iteration dependence, where the iteration vector *vec* indicates the iteration that satisfies the dependence.
  - The **source** *dependence-type*
    - specifies the cross-iteration dependences that arise from the current iteration.

Parallelizing DOACROSS loops

2 loops contribute to the pattern of dependencies … so the dependency relations for each depend(sink) is of length 2

```
#pragma omp for ordered(2) collapse(2)
    for (r=1; r<N; r++) {
        for (c=1; c<N; c++) {
            // other parallel work ...
            #pragma omp ordered depend(sink:r-1,c) \
                                depend(sink:r,c-1)
                x[r][c] += fn(x[r-1][c], x[r][c-1]);
            #pragma omp ordered depend(source)
        }
    }
```

Threads wait here until x[r-1][c] and x[r][c-1] have been released

x[r][c] is complete and released for use by other threads

Outline

- The common core: a quick review
- OpenMP Tasks
- The divide and conquer pattern
- Task group, task loops, and more
- Threadprivate
- The other workshare constructs
- Do across loops
- ➡ The OpenMP Memory model
- Point to point synchronization, atomic, and locks
- NUMA systems
- Thread affinity
- Hybrid MPI/OpenMP
- More about process and thread affinity
- A quick survey of the rest of OpenMP

Synchronization

Synchronization is used to impose order constraints and to protect access to shared data

- High level synchronization:
  - **critical**
  - **barrier**
  - atomic
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)

Covered in the common core

Synchronization: atomic

- Atomic provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```
#pragma omp parallel
{
        double B;
        B =  DOIT();


 #pragma omp atomic
             X +=  big_ugly(B);
}
```

Synchronization: atomic

- Atomic provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

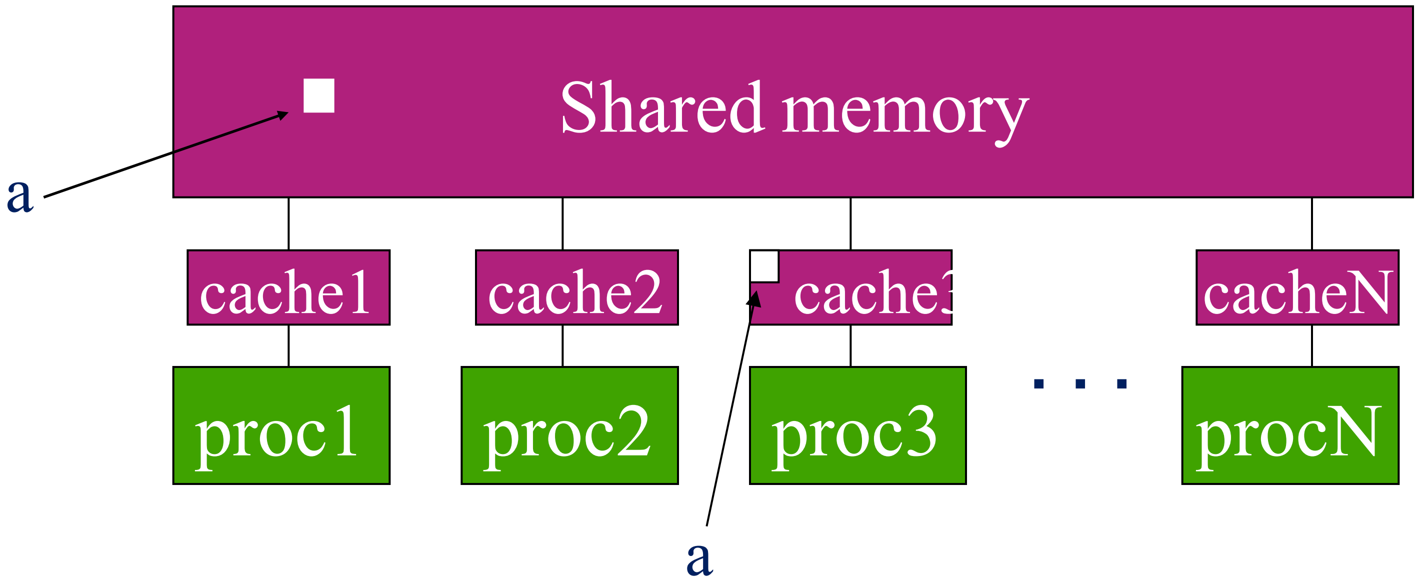
```
#pragma omp parallel
{
        double B, tmp;
        B =  DOIT();
        tmp = big_ugly(B);
 #pragma omp atomic
             X +=  tmp;
}
```

Atomic only protects the read/update of X

Additional forms of atomic were added in 3.1 (discussed later)

OpenMP memory model

- OpenMP supports a shared memory model
- All threads share an address space, where variable can be stored or retrieved:



- Threads maintain their own temporary view of memory as well … the details of which are not defined in OpenMP but this temporary view typically resides in caches, registers, write-buffers, etc.

Flush operation

- Defines a sequence point at which a thread enforces a consistent view of memory.

- For variables visible to other threads and associated with the flush operation (the **flush-set**)
  - The compiler can't move loads/stores of the flush-set around a flush:
    - All previous read/writes of the flush-set by this thread have completed
    - No subsequent read/writes of the flush-set by this thread have occurred
  - Variables in the flush set are moved from temporary storage to shared memory.
  - Reads of variables in the flush set following the flush are loaded from shared memory.

IMPORTANT POINT: The flush makes the calling threads temporary view match the view in shared memory. Flush by itself does not force synchronization.

Memory consistency: flush example

- Flush forces data to be updated in memory so other threads see the most recent value

```
double A;

A = compute();

#pragma omp flush(A)

  // flush to memory to make sure other
  //  threads can pick up the right value
```

Flush without a list: flush set is all thread visible variables

Flush with a list: flush set is the list of variables

Note: OpenMP's flush is analogous to a fence in other shared memory APIs

Flush and synchronization

- A flush operation is implied by OpenMP synchronizations, e.g.,
  - at entry/exit of parallel regions
  - at implicit and explicit barriers
  - at entry/exit of critical regions
  - whenever a lock is set or unset

  ….

  (but not at entry to worksharing regions or entry/exit of master regions)

Outline

- The common core: a quick review
- OpenMP Tasks
- The divide and conquer pattern
- Task group, task loops, and more
- Threadprivate
- The other workshare constructs
- Do across loops
- The OpenMP Memory model
- Point to point synchronization, atomic, and locks
- NUMA systems
- Thread affinity
- Hybrid MPI/OpenMP
- More about process and thread affinity
- A quick survey of the rest of OpenMP

Exercise: prod_cons.c

- Parallelize a producer/consumer program
  - One thread produces values that another thread consumes.

```
int main()
{
  double *A, sum, runtime;     int flag = 0;

  A = (double *) malloc(N*sizeof(double));

  runtime = omp_get_wtime();

  fill_rand(N, A);        // Producer: fill an array of data

  sum = Sum_array(N, A);  // Consumer: sum the array

  runtime = omp_get_wtime() - runtime;

  printf(" In %lf secs, The sum is %lf \n",runtime,sum);
}
```

- Often used with a stream of produced values to implement “pipeline parallelism”
- The key is to implement pairwise synchronization between threads

How would you modify prod_cons.c so we use two threads: one to fill the array (producer) and another con sum the array (consumer).

Pairwise synchronization in OpenMP

- OpenMP lacks synchronization constructs that work between pairs of threads.
- When needed, you have to build it yourself.
- Pairwise synchronization
  - Use a shared flag variable
  - Reader spins waiting for the new flag value
  - Use flushes to force updates to and from memory

Exercise: Producer/consumer

Start from the serial version of proc_cons.c, parallelize the program and use flush to make data sharing between threads race free

```
int main()
{
   double *A, sum, runtime;     int numthreads, flag = 0;
   A = (double *)malloc(N*sizeof(double));
   #pragma omp parallel sections
   {
     #pragma omp section
      {
        fill_rand(N, A);

        flag = 1;

      }
      #pragma omp section
      {

        while (flag == 0){

        }

        sum = Sum_array(N, A);
      }
    }
}
```

Put the flushes in the right places to make this program race-free.

Do you need any other synchronization constructs to make this work?

Solution (try 1): Producer/consumer

```
int main()
{
    double *A, sum, runtime;     int numthreads, flag = 0;
    A = (double *)malloc(N*sizeof(double));
    #pragma omp parallel sections
    {
       #pragma omp section
        {
           fill_rand(N, A);
           #pragma omp flush
           flag = 1;
           #pragma omp flush (flag)
        }
        #pragma omp section
        {
           #pragma omp flush (flag)
           while (flag == 0){
                #pragma omp flush (flag)
           }
           #pragma omp flush
           sum = Sum_array(N, A);
        }
      }
}
```

Use flag to Signal when the "produced" value is ready

Flush forces refresh to memory; guarantees that the other thread sees the new value of A

Flush needed on both "reader" and "writer" sides of the communication

Notice you must put the flush inside the while loop to make sure the updated flag variable is seen

This program works with the x86 memory model (loads and stores use relaxed atomics), but it technically has a race … on the store and later load of flag

The OpenMP 3.1 atomics (1 of 2)

- Atomic was expanded to cover the full range of common scenarios where you need to protect a memory operation so it occurs atomically:

```
# pragma omp atomic [read | write | update | capture]
```

- Atomic can protect loads

```
# pragma omp atomic read
          v = x;
```

- Atomic can protect stores

```
# pragma omp atomic write
          x = expr;
```

- Atomic can protect updates to a storage location (this is the default behavior … i.e. when you don’t provide a clause)

```
# pragma omp atomic update
          x++;  or ++x;  or x--;  or –x;  or
          x binop= expr; or x = x binop expr;
```

This is the original OpenMP atomic

The OpenMP 3.1 atomics (2 of 2)

- Atomic can protect the assignment of a value (its capture) AND an associated update operation:

```
# pragma omp atomic capture
        statement or structured block
```

- Where the statement is one of the following forms:

```
v = x++;     v = ++x;     v = x--;     v =  –x;     v = x binop expr;
```

- Where the structured block is one of the following forms:

```
{v = x;  x binop = expr;}
{v=x;    x=x binop expr;}
{v = x;   x++;}
{++x;     v=x:}
{v = x;    x--;}
{--x;       v = x;}
{x  binop = expr;     v = x;}
{X = x binop expr;   v = x;}
{v=x;     ++x:}
{x++;      v = x;}
{v= x;     --x;}
{x--;       v = x;}
```

The capture semantics in atomic were added to map onto common hardware supported atomic operations and to support modern lock free algorithms

Atomics and synchronization flags

```
int main()
{   double *A, sum, runtime;
    int numthreads, flag = 0, flg_tmp;
    A = (double *)malloc(N*sizeof(double));
    #pragma omp parallel sections
    {
       #pragma omp section
        {  fill_rand(N, A);
           #pragma omp flush
           #pragma omp atomic write
                   flag = 1;
           #pragma omp flush (flag)
        }
        #pragma omp section
        {  while (1){
               #pragma omp flush(flag)
               #pragma omp atomic read
                     flg_tmp= flag;
               if (flg_tmp==1) break;
           }
           #pragma omp flush
           sum = Sum_array(N, A);
        }
      }
}
```

This program is truly race free … the reads and writes of flag are protected so the two threads cannot conflict

Still painful and error prone due to all of the flushes that are required

OpenMP 4.0 Atomic: Sequential consistency

4.0

- Sequential consistency:
  - The order of loads and stores in a race-free program appear in some interleaved order and all threads in the team see this same order.
- OpenMP 4.0 added an optional clause to atomics
  - #pragma omp atomic [read | write | update | capture] [**seq_cst**]
- In more pragmatic terms:
  - If the seq_cst clause is included, OpenMP adds a flush without an argument list to the atomic operation so you don't need to.
- In terms of the C++'11 memory model:
  - Use of the seq_cst clause makes atomics follow the sequentially consistent memory order.
  - Leaving off the seq_cst clause makes the atomics relaxed.

Advice to programmers: save yourself a world of hurt … let OpenMP take care of your flushes for you whenever possible … use seq_cst

Atomics and synchronization flags (4.0)

```
int main()
{   double *A, sum, runtime;
    int numthreads, flag = 0, flg_tmp;
    A = (double *)malloc(N*sizeof(double));
    #pragma omp parallel sections
    {
       #pragma omp section
        {  fill_rand(N, A);

           #pragma omp atomic write seq_cst
                   flag = 1;

        }
        #pragma omp section
        {  while (1){

               #pragma omp atomic read seq_cst
                     flg_tmp= flag;
                if (flg_tmp==1) break;
            }

            sum = Sum_array(N, A);
        }
      }
}
```

This program is truly race free … the reads and writes of flag are protected so the two threads cannot conflict – and you do not use any explicit flush constructs (OpenMP does them for you)

Synchronization: Lock routines

A lock implies a memory fence (a “flush”) of all thread visible variables

- Simple Lock routines:
  - A simple lock is available if it is unset.
    - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_destroy_lock()
- Nested Locks
  - A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function
    - **omp_init_nest_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), omp_destroy_nest_lock()**

Note: a thread always accesses the most recent copy of the lock, so you don’t need to use a flush on the lock variable.

Locks with hints were added in OpenMP 4.5 to suggest a lock strategy based on intended use (e.g. contended, unconteded, speculative,, unspeculative)

Synchronization: Simple locks

- Example: conflicts are rare, but to play it safe, we must assure mutual exclusion for updates to histogram elements.

```
#pragma omp parallel for
 for(i=0;i<NBUCKETS; i++){
      omp_init_lock(&hist_locks[i]);   hist[i] = 0;
 }
 #pragma omp parallel for
 for(i=0;i<NVALS;i++){
    ival = (int)  sample(arr[i]);
    omp_set_lock(&hist_locks[ival]);
         hist[ival]++;
    omp_unset_lock(&hist_locks[ival]);
   }

for(i=0;i<NBUCKETS; i++)
 omp_destroy_lock(&hist_locks[i]);
```

One lock per element of hist

Enforce mutual exclusion on update to hist array

Free-up storage when done.

Exercise: Histograms

- Consider the file hist.c.
  - The program generates a sequence of pseudo random numbers, does some work for each one (compute a Fibonacci number), then puts the pseudorandom number into a bin in a histogram.
- Parallelize the program and evaluate the performance.

```
#pragma omp parallel for
#pragma omp critical
# pragma omp atomic [read | write | update | capture]
omp_lock_t lck;
omp_init_lock(&lck)
omp_set_lock(&lck)
omp_unset_lock(&lck)
omp_test_lock(&lck)
omp_destroy_lock(&lck)
```

These are thread safe and can be called inside a parallel region

Synchronization: Simple locks

- Example: conflicts are rare, but to play it safe, we must assure mutual exclusion for updates to histogram elements.

```
omp_lock_t hist_locks[NBUCKETS];
#pragma omp parallel for
 for(i=0;i<NBUCKETS; i++){
       omp_init_lock(&hist_locks[i]);    hist[i] = 0;
 }
 #pragma omp parallel for
 for(i=0;i<NVALS;i++){
     ival = (int)  sample(arr[i]);
     omp_set_lock(&hist_locks[ival]);
          hist[ival]++;
     omp_unset_lock(&hist_locks[ival]);
   }

for(i=0;i<NBUCKETS; i++)
  omp_destroy_lock(&hist_locks[i]);
```

One lock per element of hist

Enforce mutual exclusion on update to hist array

Free-up storage when done.

Outline

- The common core: a quick review
- OpenMP Tasks
- The divide and conquer pattern
- Task group, task loops, and more
- Threadprivate
- The other workshare constructs
- Do across loops
- The OpenMP Memory model
- Point to point synchronization, atomic, and locks
- ➡ NUMA systems
- Thread affinity
- Hybrid MPI/OpenMP
- More about process and thread affinity
- A quick survey of the rest of OpenMP

Common architectures

- Shared Memory Architecture
  - Multiple CPUs share global memory
  - Uniform Memory Access (UMA) or SMP (Symmetric Multiprocessor)
    - Equal cost to any location in memory
  - Non-Uniform Memory Access (NUMA)
    - Unequal cost across memory locations
  - Typical Shared Memory Programming Model: OpenMP, Pthreads, …

- Distributed Memory Architecture
  - Each CPU has own memory
  - Typical Message Passing Programming Model: MPI, …

- Hybrid Architecture
  - Shared memory node or socket
  - Distributed memory architecture between nodes
  - Typical Hybrid Programming Model: hybrid MPI/OpenMP, ...

Current architecture trend

- 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 for lower level loop structures

NUMA systems

- Most systems today are Non-Uniform Memory Access (NUMA)
- Accessing memory in remote NUMA is slower than accessing memory in local NUMA
- Accessing High Bandwidth Memory is faster than DDR

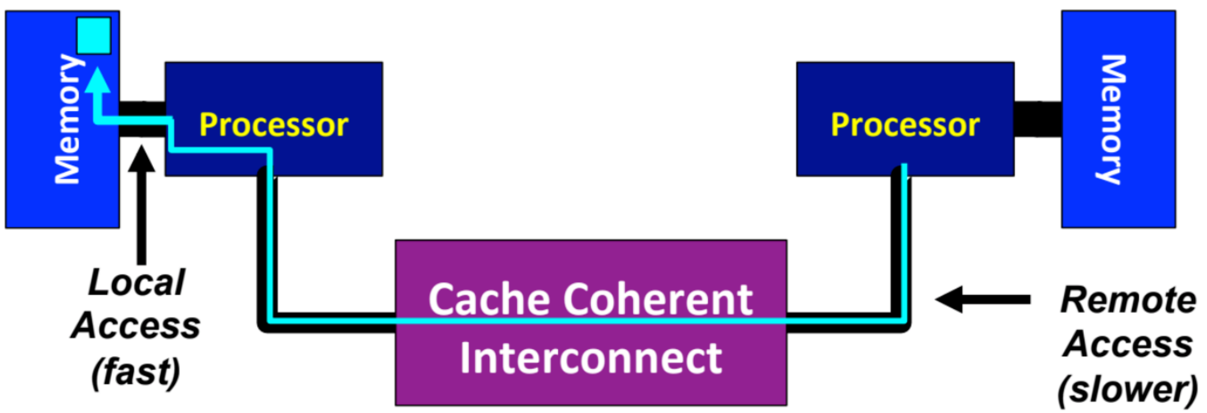
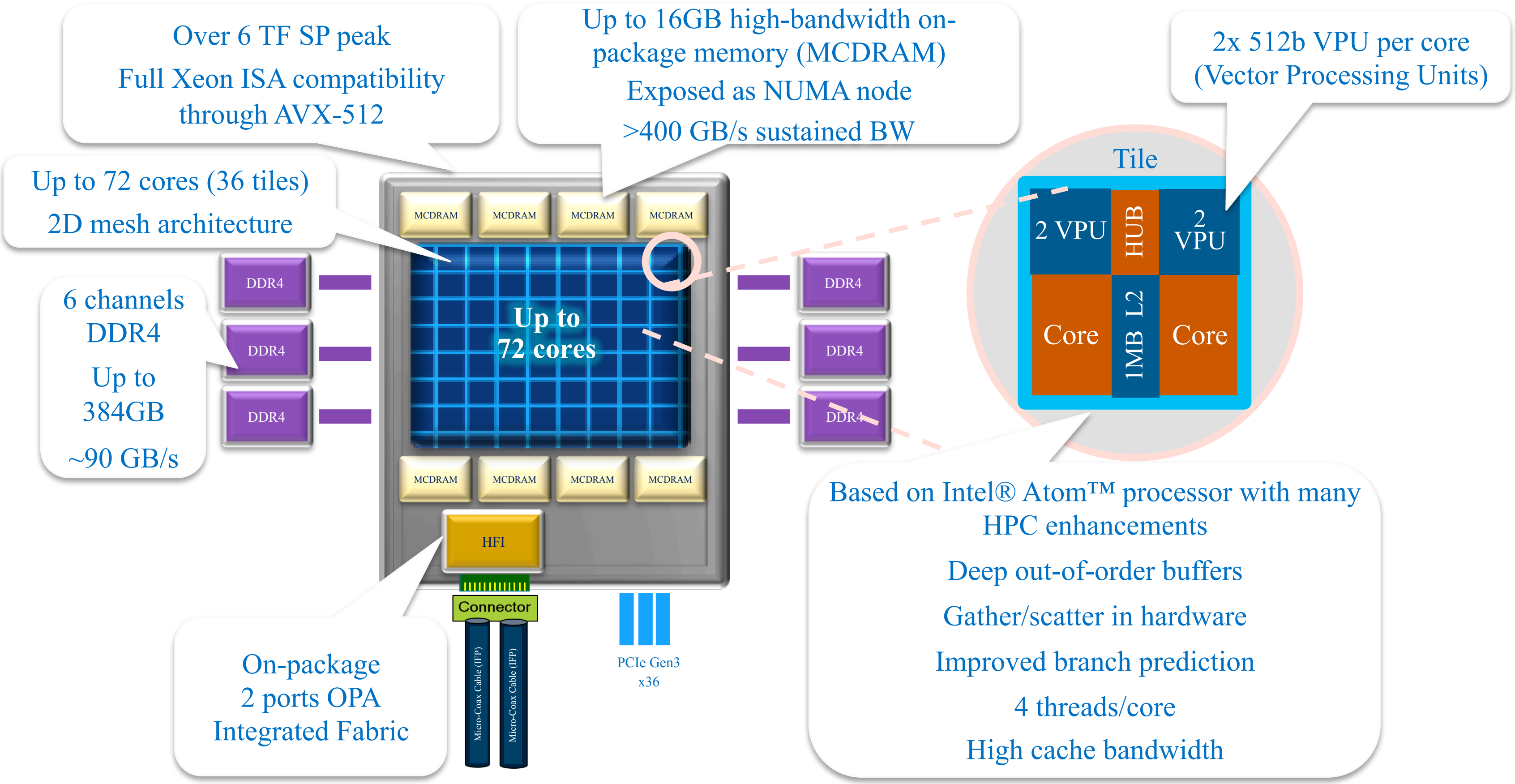


Diagram courtesy Ruud van der Pas

All modern CPUs include caches therefore all modern systems are NUMA even though we often pretend they are UMA

NUMA example: the Intel® Xeon Phi™ processor



Over 6 TF SP peak

Full Xeon ISA compatibility through AVX-512

Up to 16GB high-bandwidth on-package memory (MCDRAM)

Exposed as NUMA node

>400 GB/s sustained BW

2x 512b VPU per core (Vector Processing Units)

Up to 72 cores (36 tiles)

2D mesh architecture

6 channels DDR4

Up to 384GB

~90 GB/s

Based on Intel® Atom™ processor with many HPC enhancements

Deep out-of-order buffers

Gather/scatter in hardware

Improved branch prediction

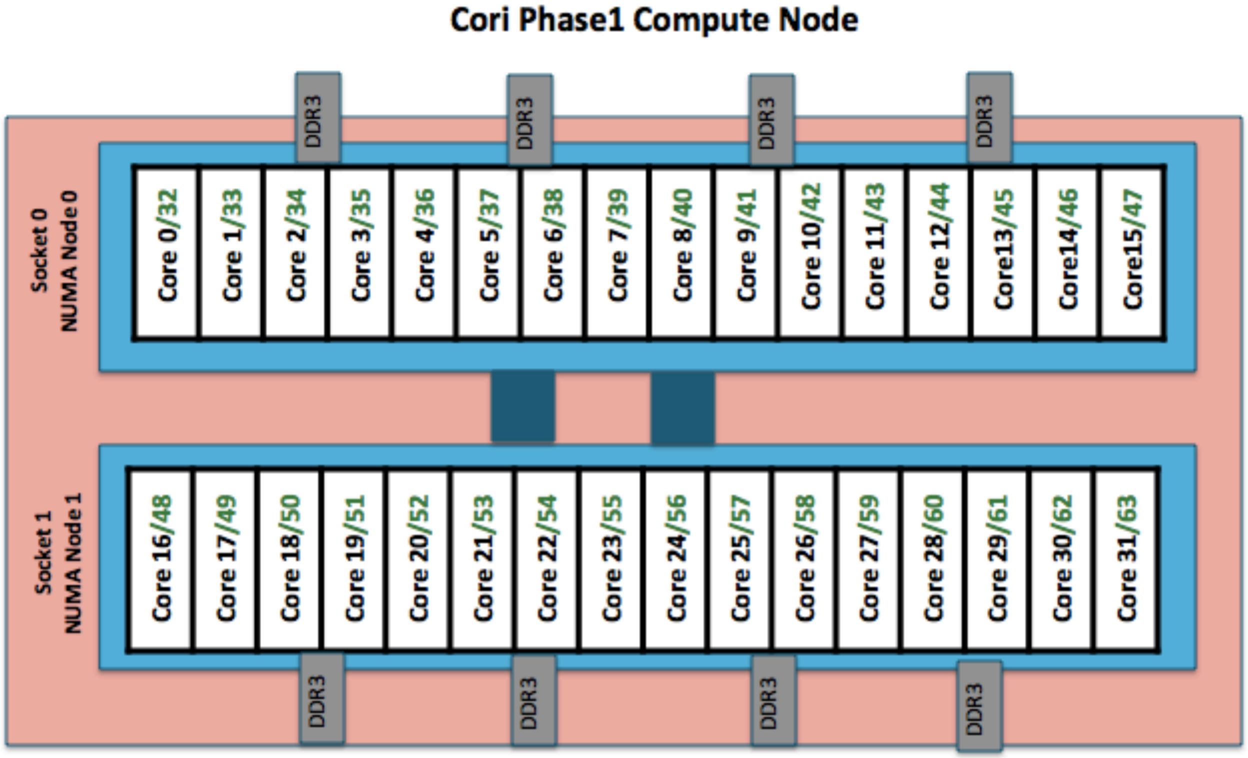
4 threads/core

High cache bandwidth

On-package
2 ports OPA
Integrated Fabric

- Diagram is for conceptual purposes only and only illustrates a CPU and memory – it is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.

Example compute nodes (Intel Haswell*)



- **An Intel Haswell node has 32 cores (64 CPUs), 128 MB DDR memory.**
- **2 NUMA domains per node, 16 cores per NUMA domain. 2 hardware threads (CPUs) per core.**
- **Memory bandwidth is non-homogeneous among NUMA domains.**
  - **CPUs 0-15, 32-47 are closer to memory in NUMA domain 0, farther to memory in NUMA domain 1.**
  - **CPUs 16-31, 48-64 are closer to memory in NUMA domain 1, farther to memory in NUMA domain 0.**

*Haswell: 16-core **Intel® Xeon™ Processor E5-2698 v3 at 2.3 GHz**

Tools to check compute node info

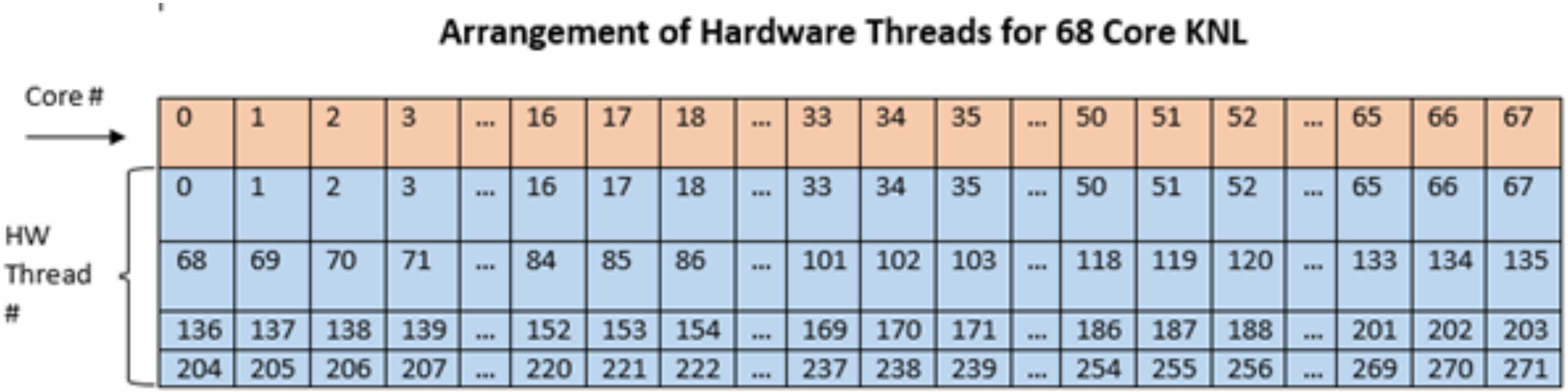
- **numactl:** controls NUMA policy for processes or shared memory
  - **numactl -H:** provides NUMA info of the CPUs

Cori Haswell node example
32 cores, 2 sockets

```
% numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
node 0 size: 64430 MB
node 0 free: 63002 MB
node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 48 49 50 51 52 53 54 55 56 57 58 59
60 61 62 63
node 1 size: 64635 MB
node 1 free: 63395 MB
node distances:node   0   1
0:  10  21
1:  21  10
```

Example compute nodes (Cori KNL*)

- **A Cori KNL node has 68 cores/272 CPUs, 96 GB DDR memory, 16 GB high bandwidth on package memory (MCDRAM).**
- **Three cluster modes, all-to-all, quadrant, sub-NUMA clustering, are available at boot time to configure the KNL mesh interconnect.**



Arrangement of Hardware Threads for 68 Core KNL

Core # →	0	1	2	3	...	16	17	18	...	33	34	35	...	50	51	52	...	65	66	67
HW Thread #	0	1	2	3	...	16	17	18	...	33	34	35	...	50	51	52	...	65	66	67
	68	69	70	71	...	84	85	86	...	101	102	103	...	118	119	120	...	133	134	135
	136	137	138	139	...	152	153	154	...	169	170	171	...	186	187	188	...	201	202	203
	204	205	206	207	...	220	221	222	...	237	238	239	...	254	255	256	...	269	270	271

- A quad,cache node has only 1 NUMA node with all CPUs on the NUMA node 0 (DDR memory). The MCDRAM is hidden from the "numactl -H" result since it is a cache.
- A quad,flat node has only 2 NUMA nodes with all CPUs on the NUMA node 0 (DDR memory). And NUMA node 1 has MCDRAM only.
- A snc2,flat node has 4 NUMA domains with DDR memory and all CPUs on NUMA nodes 0 and 1. (NUMA node 0 has physical cores 0 to 33 and all corresponding hyperthreads, and NUMA node 1 has physical cores 34 to 67 and all corresponding hyperthreads). NUMA nodes 2 and 3 have MCDRAM only.

*KNL: Intel® Xeon Phi™ processor 7250 with 68 cores @ 1.4 GHz

Intel KNL quad,flat node example

% numactl –H

Cori KNL quad,flat node example
64 cores, 272 hardware threads

```
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81
82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116
117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146
147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175
176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204
205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234
235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263
264 265 266 267 268 269 270 271
node 0 size: 96723 MB
node 0 free: 93924 MB
node 1 cpus:
node 1 size: 16157 MB
node 1 free: 16088 MB
node distances:
node   0   1
  0:  10  31
  1:  31  10
```

- The quad,flat mode has only 2 NUMA nodes with all CPUs on the NUMA node 0 (DDR memory).
- And NUMA node 1 has MCDRAM (high bandwidth memory).

Outline

- The common core: a quick review
- OpenMP Tasks
- The divide and conquer pattern
- Task group, task loops, and more
- Threadprivate
- The other workshare constructs
- Do across loops
- The OpenMP Memory model
- Point to point synchronization, atomic, and locks
- NUMA systems
- Thread affinity
- Hybrid MPI/OpenMP
- More about process and thread affinity
- A quick survey of the rest of OpenMP

What do Data Locality and Affinity mean in OpenMP

- Data Locality
  - Memory Locality: allocate memory as close as possible to the core on which the task that requested the memory is running
  - Cache Locality: use data in cache as much as possible
- Affinity
  - Process Affinity: bind processes (MPI tasks, etc.) to CPUs
  - Thread Affinity: further binding threads to CPUs that are allocated to their parent process
- Correct process, thread and memory affinity is the basis for getting optimal performance.

Memory Locality

- Memory access from different NUMA domains are different
  - Accessing memory in remote NUMA is slower than accessing memory in local NUMA
  - Accessing High Bandwidth Memory on KNL* is faster than DDR
- OpenMP does not explicitly map data across shared memories
- Memory locality is important since it impacts both memory and intra-node performance

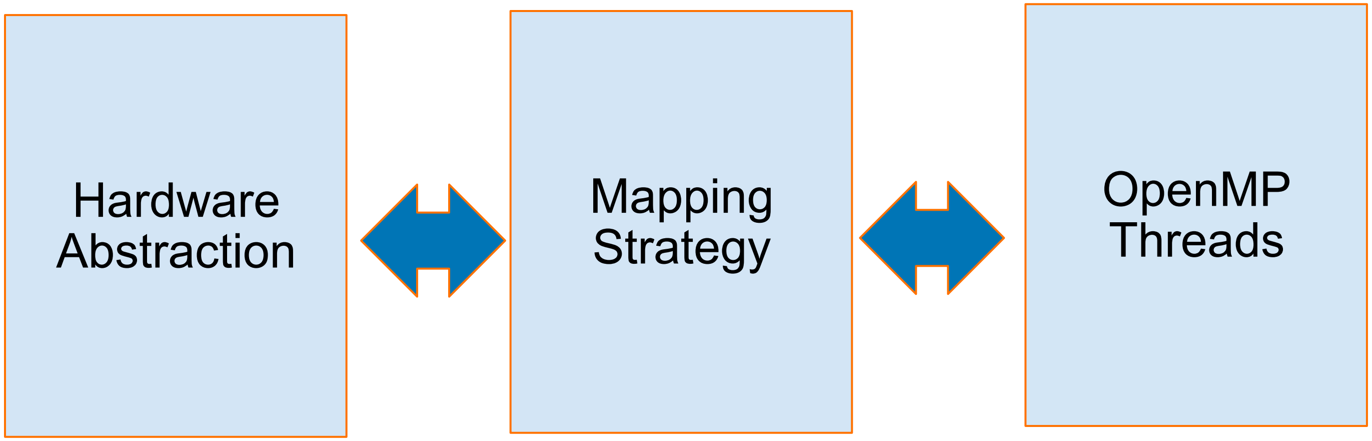
*KNL: Intel® Xeon Phi™ processor 7250 with 68 cores @ 1.4 Ghz … the "bootable" version that sits in a socket, not a co-processor

OpenMP Thread Affinity

- OpenMP provides a mechanism to map threads to hardware execution units (e.g. hardware threads, cores, sockets), with the following goals:
  - Maximize resource utilization
  - Minimize thread contention for the same hardware resource
  - Maximize local accesses and minimize remote memory accesses in NUMA
- Develop strategies for memory latency programs *vs.* memory bandwidth bound programs
  - Cache reuse by threads
  - Bandwidth aggregation
  - Reduce thread synchronization overheads
- Bind OpenMP threads to the hardware threads or cores

OpenMP thread affinity concepts

- Three main concepts:



OMP_PLACES
Environment Variable
(e.g. threads, cores, sockets)

OMP_PROC_BIND
Environment Variable
Or
proc_bind() clause
of parallel region

OMP_NUM_THREADS
Environment Variable
Or
num_threads() clause
of parallel region

Courtesy of Oscar Hernandez, ORNL

Runtime Environment Variable: OMP_PLACES (1)

- OpenMP 4.0 added OMP_PLACES environment variable
  - controls thread allocation
  - defines a series of places to which the threads can be assigned
- OMP_PLACES can be
  - threads: each place corresponds to a single hardware thread on the target machine.
  - cores: each place corresponds to a single core (having one or more hardware threads) on the target machine.
  - sockets: each place corresponds to a single socket (consisting of one or more cores) on the target machine.
  - A list with explicit CPU ids (see next slide)
- Examples:
  - export OMP_PLACES=threads
  - export OMP_PLACES=cores

Runtime Environment Variable: OMP_PLACES (2)

- OMP_PLACES can also be
  - A list with explicit place values of CPU ids, such as:
    - "{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}”
    - “{0:4},{4:4},{8:4},{12:4}” (default stride is 1)
    - Format: {lower-bound:length:stride}. Thus, specifying {0:3:2} is the same as specifying {0,2,4}
- Examples:
  - export OMP_PLACES=“ {0:4:2},{1:4:2}” (which is equivalent to “{0,2,4,6},{1,3,5,7}”)
  - export OMP_PLACES=“{0:8:1}” (which is equivalent to “{0,1,2,3,4,5,6,7}”

Runtime Environment Variable: OMP_PROC_BIND

- Controls thread affinity within and between OpenMP places
- OpenMP 3.1 only has OMP_PROC_BIND, either TRUE or FALSE.
  - If true, the runtime will not move threads around between processors.
- OpenMP 4.0 still allows the above. Added options:
  - close: bind threads close to the master thread
  - spread: bind threads as evenly distributed (spread) as possible
  - master: bind threads to the same place as the master thread

    (Can be used when master thread is bound to core or socket)
- Examples:
  - OMP_PROC_BIND=spread
  - OMP_PROC_BIND=spread,close (for nested levels)

Considerations for OMP_PROC_BIND choices

- Selecting the “right” binding is dependent on the architecture topology but also on the application characteristics
- Putting threads apart (“spread”, e.g. different sockets)
  - Can help to improve aggregated memory bandwidth
  - Combine the cache sizes across cores
  - May increase the overhead of synchronization across far apart threads
  - Aggregates memory bandwidth to/from accelerator(s)
- Putting threads near (“close”, e,g. hardware threads or cores sharing caches)
  - Reverse impact as “spread”
  - Good for synchronization and data reuse
  - May decrease total memory bandwidth

Runtime Environment Variable: OMP_PROC_BIND (2)

- Prototype example: 4 cores total, 2 hyperthreads per core, 4 OpenMP threads
- none: no affinity setting.
- close: Bind threads as close to each other as possible

Node	Core 0		Core 1		Core 2		Core 3	
	HT1	HT2	HT1	HT2	HT1	HT2	HT1	HT2
Thread	0	1	2	3				

- spread: Bind threads as far apart as possible.

Node	Core 0		Core 1		Core 2		Core 3	
	HT1	HT2	HT1	HT2	HT1	HT2	HT1	HT2
Thread	0		1		2		3	

- master: bind threads to the same place as the master thread

Memory Affinity: “First Touch” memory

Step 1.1 Initialization by master thread only

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

Step 1.2 Initialization by all threads

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

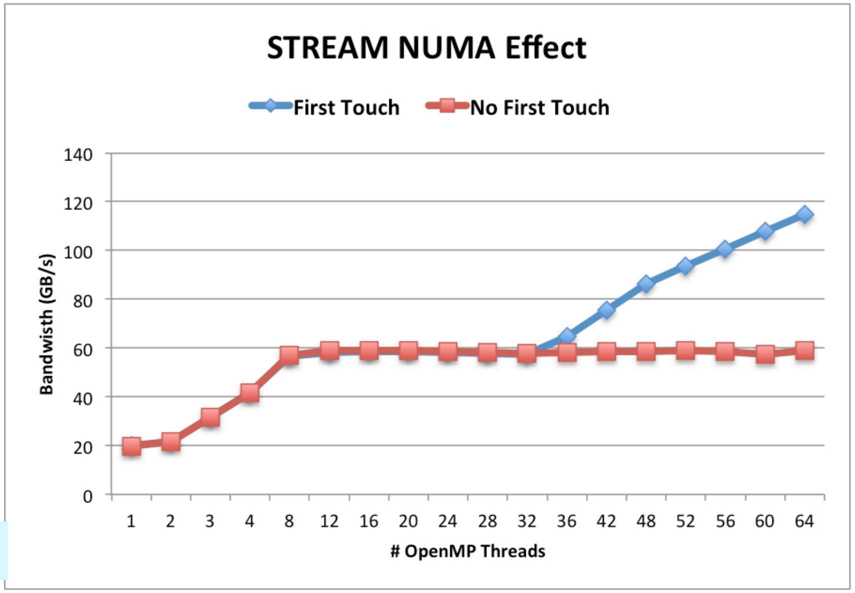
Step 2 Compute

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

OMP_PROC_BIND=close

- Memory affinity is not defined when memory was allocated, instead it will be defined at initialization.
- Memory will be local to the thread which initializes it. This is called **first touch** policy.

Red: step 1.1 + step 2. No First Touch
Blue: step 1.2 + step 2. First Touch

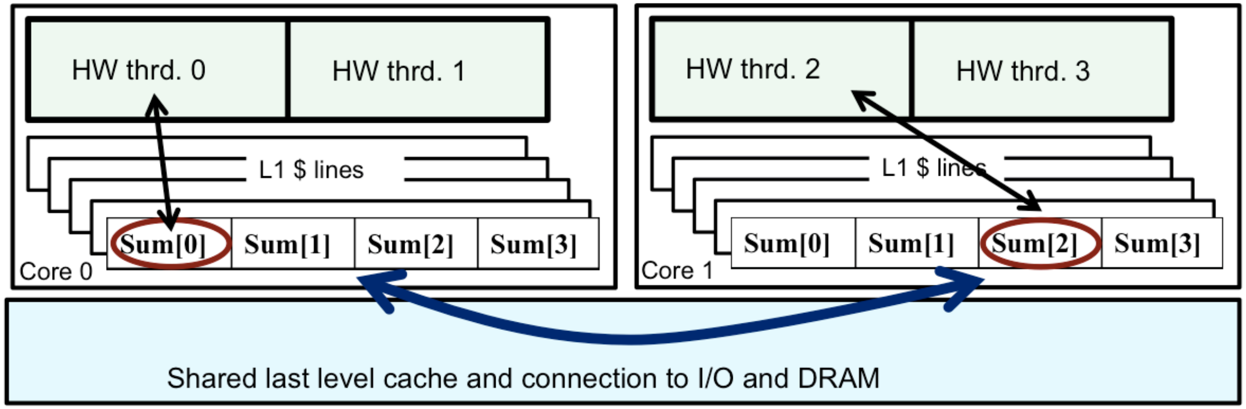


“Perfect Touch” is hard

- Hard to do “perfect touch” for real applications.
- General recommendation is to use number of threads fewer than number of CPUs per NUMA domain.
- For example: 16 cores (32 CPUs) per NUMA domain. Sample run options:
  - 2 MPI tasks, 1 MPI task per NUMA domain, with 32 OpenMP threads (if using hyperthreads) or 16 OpenMP threads (if not using hyperthreads) per MPI task
  - 4 MPI tasks, 2 MPI tasks per NUMA domain, with 16 OpenMP threads (if using hyperthreads) or 8 OpenMP threads (if not using hyperthreads) per MPI task
  - …

Cache coherence and false sharing

- ccNUMA node: cache-coherence NUMA node.
- Data from memory are accessed via cache lines.
- Multiple threads hold local copies of the same (global) data in their caches. Cache coherence ensures the local copy to be consistent with the global data.
- Main copy needs to be updated when a thread writes to local copy.
- Writes to same cache line from different threads is called false sharing or cache thrashing, since it needs to be done in serial. Use atomic or critical to avoid race condition.



- False sharing significantly degrade performance. Tips for avoiding:
  - use private variables
  - pad arrays so each thread will update a different cache line
  - use critical or atomic for update

False sharing example

- Array A is shared. Chunk size is 1.
- Updating A[0] or A[1] requires entire cache line update to retain cache coherency.
- Loop essentially becomes serial.

False sharing

```
int A[N];
#pragma omp parallel for schedule (static, 1)
for (i=0; i<N; i++) {
 A[i] += i;
```

- One solution: Pad arrays so elements you use are on distinct cache lines.

No false sharing, array is padded

```
int A[N][cache_line_size];
#pragma omp parallel for schedule (static, 1)
for (i=0; i<N; i++) {
 A[i][0] += i;
```

Cache Locality

- Cache locality means to use data in cache as much as possible
- Tips often used in real codes
  - Pin threads and associate threads onto regions of system
  - Exploit “first touch” data policy
  - Privatize data
  - Optimize code for cache
    - Use a memory stride of 1
      - Fortran: column-major order
      - C: row-major order
    - Access variable elements in the same order as they are stored in memory
    - Interchange loops or index orders if necessary
- If performance is bad, look for false sharing
  - Occurs frequently, performance degradation can be catastrophic

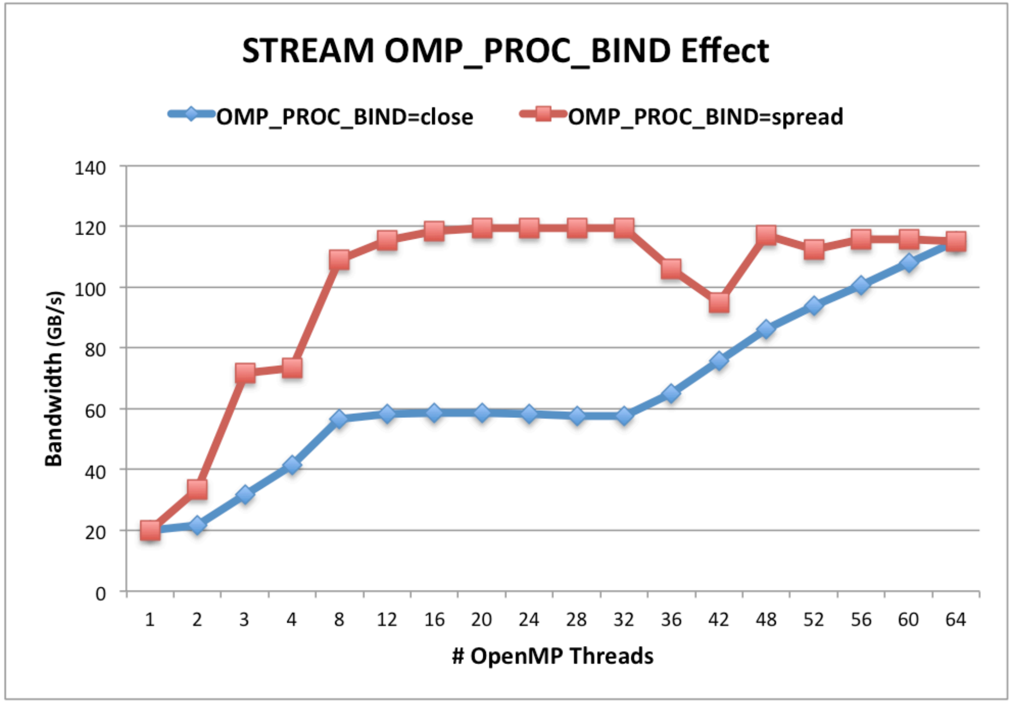
OMP_PROC_BIND choices for STREAM

OMP_NUM_THREADS=32
OMP_PLACES=threads

OMP_PROC_BIND=close
Threads 0 to 31 bind to CPUs 0,32,1,33,2,34,…15,47. All threads are in the first socket. The second socket is idle. Not optimal.

OMP_PROC_BIND=spread
Threads 0 to 31 bind to CPUs 0,1,2,… to 31. Both sockets and memory are used to maximize memory bandwidth.

Blue: OMP_PROC_BIND=close
Red: OMP_PROC_BIND=spread
Both with First Touch



Other Runtime Environment Variables for affinity support

- OMP_NUM_THREADS
- OMP_THREAD_LIMIT
- OMP_NESTED
- OMP_MAX_ACTIVE_LEVELS

- Names are upper case, values are case insensitive

Affinity clauses for OpenMP parallel construct

- The “num_threads” and “proc_bind” clauses can be used
  - The values set with these clauses take precedence over values set by runtime environment variables
- Helps code portability
- Examples:
  - C/C++:

    #pragma omp parallel *num_threads(2) proc_bind(spread)*
  - Fortran:

    !$omp parallel *num_threads (2) proc_bind (spread)*

    ...

    !$omp end parallel

Runtime APIs for thread affinity support

- OpenMP 4.5 added runtime functions to determine the effect of thread affinity clauses
- Query functions for OpenMP thread affinity were added
  - **omp_get_num_places**: returns the number of places
  - **omp_get_place_num_procs**: returns number of processors in the given place
  - **omp_get_place_proc_ids**: returns the ids of the processors in the given place
  - **omp_get_place_num:** returns the place number of the place to which the current thread is bound
  - **omp_get_partition_num_places:** returns the number of places in the current partition
  - **omp_get_partition_place_nums:** returns the list of place numbers corresponding to the places in the current partition

Other Runtime APIs for thread affinity support

- omp_get_nested, omp_set_nested
- omp_get_thread_limit
- omp_get_level
- omp_get_active_level
- omp_get_max_active_levels, omp_set_max_active_levels
- omp_get_proc_bind, omp_set_proc_bind
- omp_get_num_threads, omp_set_num_threads
- omp_get_max_threads

Exercise: “First Touch” with STREAM benchmark

- STREAM benchmark codes: stream.c, stream.f
- Check the source codes to see if “first touch” is implemented
- With “first touch” on (stream.c) and off (stream_nft.c), experiment with different OMP_NUM_THREADS and OMP_PROC_BIND settings to understand how “first touch” and OMP_PROC_BIND choices affect STREAM memory bandwidth results on Haswell (look at the Best Rate for Triad in the output).
- Compare your results with a few data points on the two STREAM plots shown earlier in this slide deck.

Sample nested OpenMP program

```
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level)
{
    #pragma omp single {
         printf("Level %d: number of threads in the
team: %d\n", level, omp_get_num_threads());
        }
}
int main()
{
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2) {
        report_num_threads(1);
        #pragma omp parallel num_threads(2) {
            report_num_threads(2);
            #pragma omp parallel num_threads(2) {
                report_num_threads(3);
            }
        }
    }
    return(0);
}
```

```
% a.out
Level 1: number of threads in the team: 2
Level 2: number of threads in the team: 1
Level 3: number of threads in the team: 1
Level 2: number of threads in the team: 1
Level 3: number of threads in the team: 1
```

```
% export OMP_NESTED=true
% export OMP_MAX_ACTIVE_LEVELS=3
% a.out
Level 1: number of threads in the team: 2
Level 2: number of threads in the team: 2
Level 2: number of threads in the team: 2
Level 3: number of threads in the team: 2
Level 3: number of threads in the team: 2
Level 3: number of threads in the team: 2
Level 3: number of threads in the team: 2
```

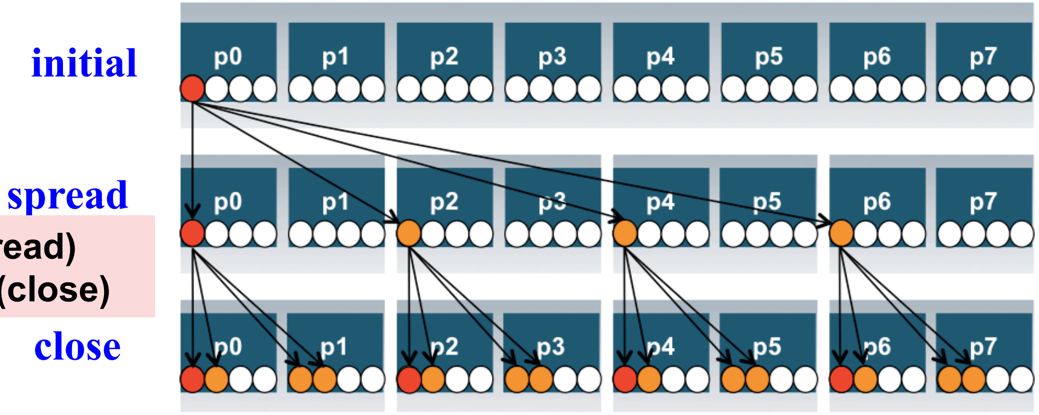
Level 0: P0
Level 1: P0 P1
Level 2: P0 P2; P1 P3
Level 3: P0 P4; P2 P5; P1 P6; P3 P7

Process and Thread Affinity in nested OpenMP

- A combination of OpenMP environment variables and run time flags are needed for different compilers and different batch schedulers on different systems.

Illustration of a system with:
2 sockets, 4 cores per socket,
4 hyper-threads per core

#pragma omp parallel proc_bind(spread)
#pragma omp parallel proc_bind(close)



Example: Use Intel compiler with SLURM on Cori Haswell:

```
export OMP_NESTED=true
export OMP_MAX_ACTIVE_LEVELS=2
export  OMP_NUM_THREADS=4,4
export OMP_PROC_BIND=spread,close
export OMP_PLACES=threads
srun -n 4 -c 16 --cpu_bind=cores ./nested.intel.cori
```

- Use num_threads clause in source codes to set threads for nested regions.
- For most other non-nested regions, use OMP_NUM_THREADS environment variable for simplicity and flexibility.

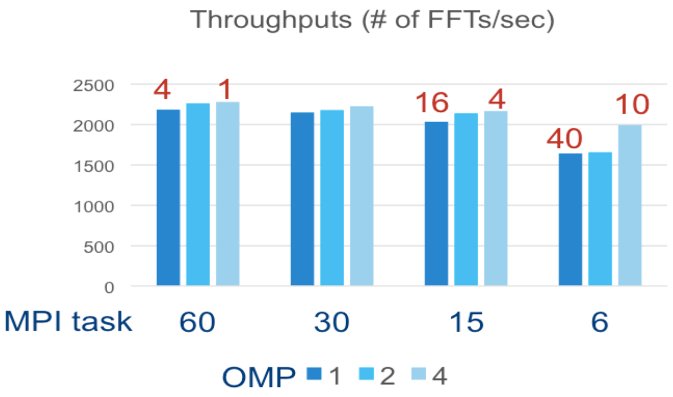
When to use nested OpenMP

- Beneficial to use nested OpenMP to allow more fine-grained thread parallelism.
- Some application teams are exploring with nested OpenMP to allow more fine-grained thread parallelism.
  - Hybrid MPI/OpenMP not using node fully packed
  - Top level OpenMP loop does not use all available threads
  - Multiple levels of OpenMP loops are not easily collapsed
  - Certain computational intensive kernels could use more threads
  - MKL can use extra cores with nested OpenMP
- Nested level can be arbitrarily deep.

Use multiple threads in MKL

- By Default, in OpenMP parallel regions, only 1 thread will be used for MKL calls.
  - MKL_DYNAMICS is true by default
- Nested OpenMP can be used to enable multiple threads for MKL calls. Treat MKL as a nested inner OpenMP region.
- Sample settings

```
export OMP_NESTED=true
export OMP_PLACES=cores
export OMP_PROC_BIND=spread,close
export OMP_NUM_THREADS=6,4
export MKL_DYNAMICS=false
export OMP_MAX_ACTIVE_LEVELS=2
```



$$N_{MKL} = 240/(N_{MPI} * OMP)$$

FFT3D on KNC, Ng=64^3 *example*

Courtesy of Jeongnim Kim, Intel

*KNC: Intel® Xeon Phi™ processor (Knights Corner) … the first generation co-processor version of the chip.

Exercise: affinity choices

- Pure OpenMP code: xthi-omp.c
- Nested OpenMP code: xthi-nested-omp.c
- Sample output:

```
% ./xthi-omp |sort –k4n
Hello from thread 0, on nid00011. (core affinity = 0)
Hello from thread 1, on nid00011. (core affinity = 4)
Hello from thread 2, on nid00011. (core affinity = 8) ...
```

- Experiment with different OMP_NUM_THREADS, OMP_PROC_BIND, and OMP_PLACES settings to check thread affinity on different compute node architectures (for example, Cori Haswell and KNL).
- Make sure to understand the CPU output values in “core affinity=xxx” report.

Essential runtime settings for KNL MCDRAM Memory Affinity

- In quad, cache mode, no special setting is needed to use MCDRAM
- In quad,flat mode, using quad,flat as an example
  - NUMA node 1 is MCDRAM
- Enforced memory mapping to MCDRAM
  - If using >16 GB, malloc will fail
  - Use "**numactl -m 1 ./myapp**" as the executable
- (instead of "./myapp")
- Preferred memory mapping to MCDRAM:
  - If using >16 GB, malloc will spill to DDR
  - Use "**numactl -p 1 ./myapp**" as the executable (instead of "./myapp")

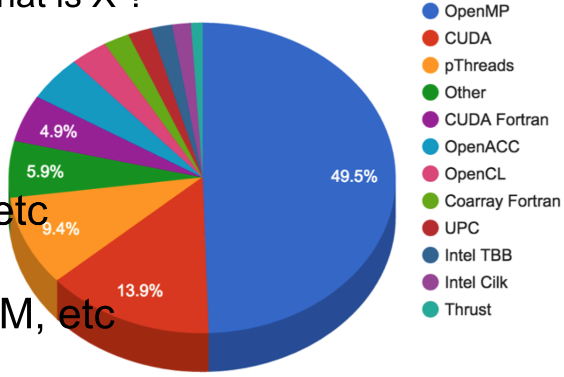
Outline

- The common core: a quick review
- OpenMP Tasks
- The divide and conquer pattern
- Task group, task loops, and more
- Threadprivate
- The other workshare constructs
- Do across loops
- The OpenMP Memory model
- Point to point synchronization, atomic, and locks
- NUMA systems
- Thread affinity
- Hybrid MPI/OpenMP
- More about process and thread affinity
- A quick survey of the rest of OpenMP

Choice of programming models for modern HPC systems

- MPI was developed primarily for inter-address space (inter means between or among)
- OpenMP was developed for shared memory or intra-node, and now supports accelerators as well (intra means within)
- Hybrid Programming (MPI+X) is when we use a solution with different programming models for inter *vs.* intra-node parallelism
- Several solutions including
  - Pure MPI
  - MPI + Shared Memory (OpenMP)
  - MPI + Accelerator programming
    - OpenMP 4.5 shared memory + offload, OpenACC, CUDA, etc
  - MPI message passing + MPI shared memory
  - PGAS: UPC/UPC++, Fortran 2008 coarrays, GA, OpenSHMEM, etc
  - Runtime tasks (Legion, HPX, HiHat (draft), etc)
  - Other hybrid based on Kokkos, Raja, SYCL, C++17 (C++20 draft)

NERSC data from 2015:
When asked: If you use MPI + X, what is X ?



Courtesy of Alice Koniges and Oscar Hernandez

Why Hybrid MPI + OpenMP?

- Homogeneous and Heterogeneous systems have large core counts per node. For example:
  - NERSC Cori: Xeon Phi (KNL) 68 cores, 4 hardware threads per core. Total of 272 threads per node
  - ORNL Summit: Total 176 (SMT4) Power9 threads + 6 Volta GPUs per node
- Application may run with MPI everywhere, but possibly not good performance
  - Needs hybrid programming to manage threading, improve SIMD, accelerator programming
- Hybrid MPI/OpenMP is a recommended programming model to achieve scaling capability and code portability, new trend
- Incremental parallelism with OpenMP for cores and accelerators
- Some applications have two levels of parallelism naturally; advanced OpenMP features extend beyond the two-level model
- Some problems have a natural restriction on the number of MPI tasks
- Avoids extra communication overhead within the node
- Adds fine granularity (larger message sizes) and allows increased dynamic load balancing across MPI tasks

Hybrid MPI/OpenMP reduces memory usage

- Many applications will not fit into the node memory using Pure MPI (e.g. per core) because of the memory overhead for each MPI task
- Smaller number of MPI processes. Save the memory needed for the executables and process stack copies.
- Larger domain for each MPI process, so fewer ghost cells
  - e.g. Combine 16 10x10 domains to one 40x40. Assume 2 ghost layers.
  - Total grid size: Original: 16x14x14=3136, new: 44x44=1936.
- Save memory for MPI buffers due to smaller number of MPI tasks.
- Fewer messages, larger message sizes, and smaller MPI all-to-all communication sizes improve performance.

Example of Hybrid Code

```
Program hybrid
 call MPI_INIT_THREAD (required, provided, ierr)
 call MPI_COMM_RANK (…)
 call MPI_COMM_SIZE (…)
  …  some computation and MPI communication
  call OMP_SET_NUM_THREADS(4)
 !$OMP PARALLEL DO PRIVATE(i)
 !$OMP&                    SHARED(n)
    do i=1,n
        … computation
    enddo
  !$OMP END PARALLEL DO
  …  some computation and MPI communication
 call MPI_FINALIZE (ierr)
 end
```

Supported levels of thread safety

- Defined by MPI standard in the form of **commitments** a multithreaded **application** makes to the MPI implementation. Not specific to hybrid MPI/OpenMP.
- Use MPI_INIT_THREAD (required, provided, ierr), as an alternative to MPI_INIT (ierr)
  - IN: “required”, desired level of thread support (integer)
  - OUT: “provided”, provided level of thread support (integer)
  - Returned “provided” maybe lower than “required”
- Thread support levels:
  - MPI_THREAD_SINGLE: Only one thread will execute
  - MPI_THREAD_FUNNELED: Process may be multi-threaded, but only master thread will make MPI calls (all MPI calls are ’’funneled" to master thread)
  - MPI_THREAD_SERIALIZED: Process may be multi-threaded, 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

Thread support levels

environment variable MPICH_MAX_THREAD_SAFETY	thread support level
not set	MPI_THREAD_SINGLE
single	MPI_THREAD_SINGLE
funneled	MPI_THREAD_FUNNELED
serialized	MPI_THREAD_SERIALIZED
multiple	MPI_THREAD_MULTIPLE

- Different compilers may have different max level of thread support
- Make sure to set the environment variable in order to get the desired thread level. Otherwise, you may get a lower level than desired even if the compiler supports it

MPI calls inside OMP MASTER

- MPI_THREAD_FUNNELED is required
- OMP MASTER does not include any barrier. If the application needs a barrier (e.g., to prevent race conditions between the buffer usage in the MPI call and some numerical buffer read or write in other threads) then explicit OMP BARRIERs may be needed
- Such barriers would imply that all other threads are sleeping while the master thread does MPI communication! (may not be able to saturate the inter-node bandwidth)

```
!$OMP BARRIER
!$OMP MASTER
    call MPI_xxx(...)
!$OMP END MASTER
!$OMP BARRIER
```

MPI calls inside OMP SINGLE

- MPI_THREAD_SERIALIZED is required
- OMP_BARRIER or an implicit barrier is needed at the beginning since OMP_SINGLE only guarantees synchronization at the end
- It also implies all other threads are sleeping while one thread does MPI communication! (may not be able to saturate the inter-node bandwidth)

```
!$OMP BARRIER
!$OMP SINGLE
    call MPI_xxx(…)
!$OMP END SINGLE
```

THREAD FUNNELED/SERIALIZED vs. Pure MPI

- FUNNELED/SERIALIZED:
  - All other threads are sleeping while single thread communicating
  - Only one thread communicating maybe not able to saturate the inter-node bandwidth
- Pure MPI:
  - Every CPU communicating may over saturate the inter-node bandwidth
- Overlap communication with computation!

Overlap communication and computation

- Is a good strategy for improving performance
  - Use MPI inside parallel region with thread-safe MPI
- Need at least MPI_THREAD_FUNNELED
- Many “easy” hybrid programs only need MPI_THREAD_FUNNELED
  - Simplest and least error-prone way is to use MPI outside parallel region, and allow only master thread to communicate between MPI tasks
  - While this single master is making MPI calls, other threads are computing
- Must be able to separate codes that can run before or after ghost zone or halo info is received. Can be very hard conceptually
- May lose compiler optimizations such as vectorization

```
!$OMP PARALLEL
   if (my_thread_rank < 1) then
      call MPI_xxx(…)
   else
      do some computation
   endif
!$OMP END PARALLEL
```

PARSEC: overlap comp and comm (1)

Original Force Pseudocode

```
do type
  do atom
    calc A & B
    reduceAll A & B to master
    calc Δforce = f(A&B) on master
    store force on master
  end atom
end type
```

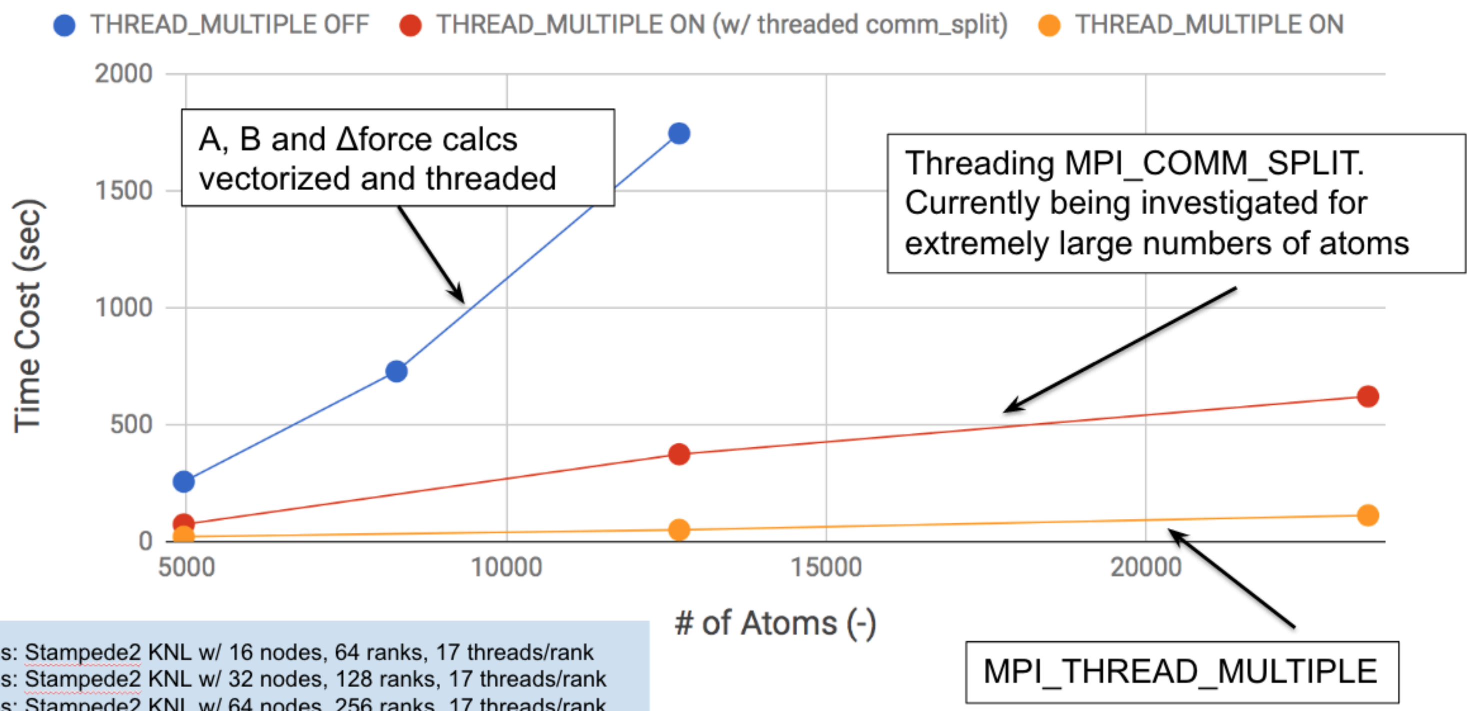
- Preemptively create an **array of comms**, one for each atom, to allow mpi ranks without data to move to the next atom
- **Atom loop is threaded**, allowing multiple atoms to be solved simultaneously
- Use **MPI_THREAD_MULTIPLE,** multiple threads call MPI

Improved Version with MPI_THREAD_MULTIPLE

```
do type
  MPI_COMM_SPLIT(atom, rank_has_data)
  !$OMP DO
  do atom
    if comm(atom) = MPI_COMM_NULL, cycle
    calc A & B
    reduceAll(comm(atom), A)
    calc Δforce = f(A&B)
    reduceAll(comm(atom), Δforce)
    store locally with master of comm(atom)
  end atom
  !$OMP END DO
end type
```

Courtesy of Kevin Gott, NERSC

PARSEC: overlap comp and comm (2)



A, B and Δforce calcs vectorized and threaded

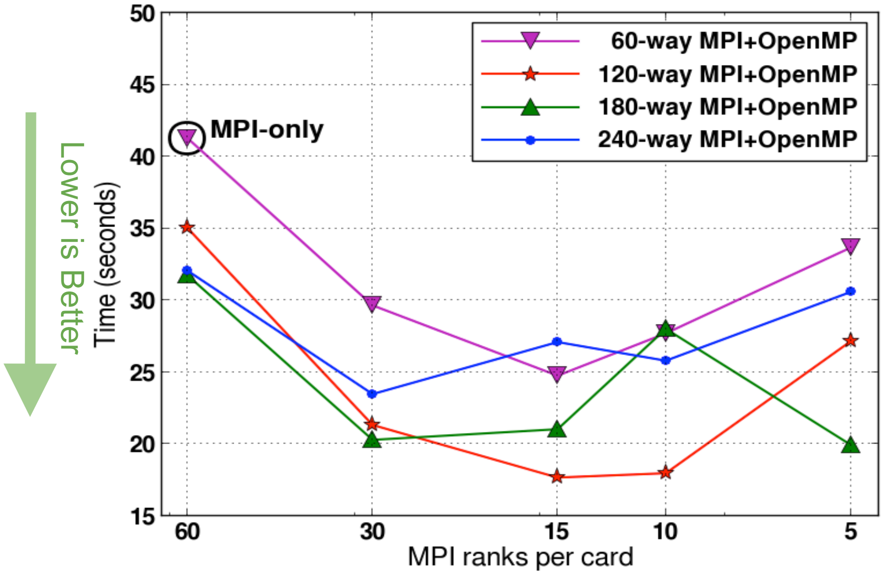
Threading MPI_COMM_SPLIT. Currently being investigated for extremely large numbers of atoms

MPI_THREAD_MULTIPLE

5k atoms: Stampede2 KNL w/ 16 nodes, 64 ranks, 17 threads/rank
8k atoms: Stampede2 KNL w/ 32 nodes, 128 ranks, 17 threads/rank
13k atoms: Stampede2 KNL w/ 64 nodes, 256 ranks, 17 threads/rank
23k atoms: Stampede2 KNL w/ 256 nodes, 512 ranks, 34 threads/rank

Courtesy of Kevin Gott, NERSC

MPI *vs.* OpenMP scaling analysis for optimal configuration



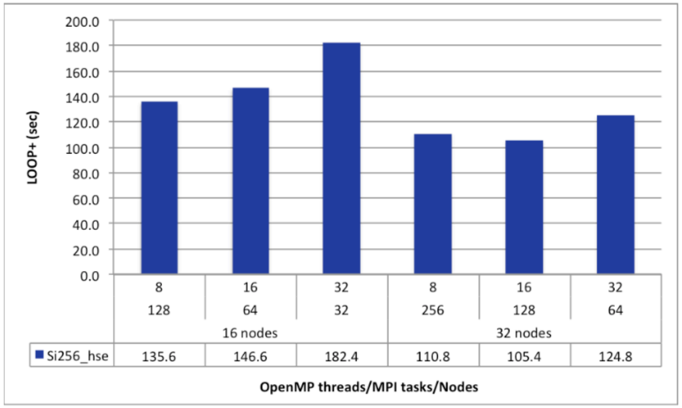
Courtesy of Chris Daley, NERSC

- Each line represents multiple runs using fixed total number of cores = #MPI tasks x #OpenMP threads/task
- Scaling may depend on the kernel algorithms and problem sizes
- In this test case, 15 MPI tasks with 8 OpenMP threads per task is optimal

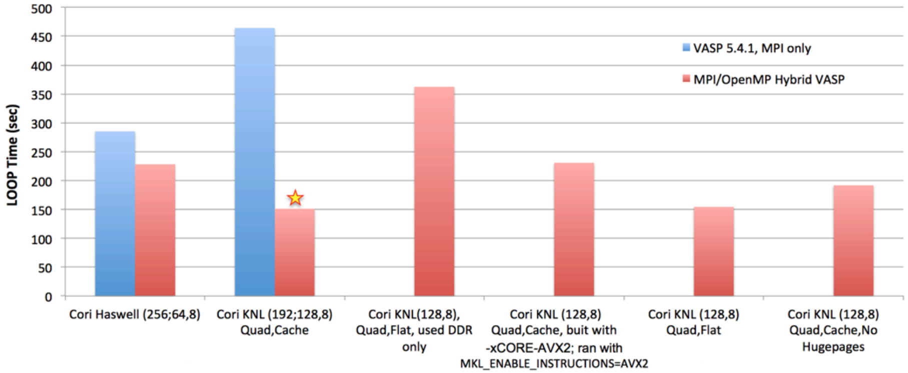
Find the sweet spot for hybrid MPI/OpenMP

VASP: MPI/OpenMP Scaling Study

- **Original MPI parallelization**
  - –Over the bands (high level)
  - –Over Fourier coefficient of the bands (low level)
- **MPI + OpenMP parallelization**
  - –MPI over bands (high level)
  - –OpenMP threading over the coefficients of bands, either by explicitly adding OpenMP directives or via using threaded FFTW and LAPACK/BLAS3 libraries
  - –No nested OpenMP
  - –SIMD vectorization is deployed extensively
  - –MPI/OpenMP scaling study to find the sweet spot
  - –Other tuning options



MPI/OpenMP hybrid VASP outperforms the pure MPI code by 2-3 times on Cori KNL



All runs used 8 Haswell or KNL nodes on Cori. The numbers inside the "()", [num;] num,num, are the number of MPI tasks used for the MPI only VASP 5.4.1, if present; the MPI tasks, OpenMP threads per task used to run the Hybrid VASP.

Courtesy of Zhengji Zhao,NERSC

Best practices for Hybrid MPI/OpenMP (1)

- With sequential code, decompose with MPI first, then add OpenMP
- Use profiling tools to find hotspots. Add OpenMP and check correctness incrementally
- Choose between fine grain or coarse grain parallelism implementation
- Reduce number of OpenMP parallel regions to reduce overhead costs
- Parallelize outer loop and consider loop collapse, loop fusion or loop permutation to give all threads enough work, and to optimize thread cache locality. Use NOWAIT clause if possible
- Minimize shared variables, minimize serial/critical/barrier sections
- Pay attention to load imbalance. If needed, try dynamic scheduling or implement own load balance scheme

Best practices for Hybrid MPI/OpenMP (2)

- Decide whether to overlap MPI communication with thread computation
- Simplest and least error-prone way is to use MPI outside parallel region, and allow only master thread to communicate between MPI tasks. MPI_THREAD_FUNNELED is usually the best choice.
- Consider OpenMP TASKing
- Consider nested OpenMP
- Consider OpenMP SIMD for better vectorization
- Experiment with different combinations of MPI tasks and number of threads per task. Less MPI tasks may not saturate inter-node bandwidth
- Be aware of NUMA domains
- Leave some cores idle on purpose, for memory capacity or bandwidth capacity

Why Hybrid MPI/OpenMP code is sometimes slower?

- All threads are idle except one while MPI communication
  - Need overlap comp and comm for better performance
  - Critical Section for shared variables
- Thread creation overhead
- Cache coherence, false sharing
- Data placement, NUMA effects
- Natural one level parallelism problems
- Pure OpenMP code performs worse than pure MPI within node
- Lack of optimized OpenMP compilers/libraries

Hybrid programming with MPI + OpenMP is a viable and efficient model

- MPI+OpenMP interoperability can happen in multiple ways – Funneled and Serialized modes are most common where a single thread makes MPI calls at a time
- THREAD_MULTIPLE is becoming increasingly common where multiple threads can make MPI calls simultaneously ("fully multi-threaded")
  - Now provided by almost all implementations
  - Optimization is important
- Other options such as "MPI everywhere" are also possible, especially with advanced MPI options
  - Solutions with no MPI (not covered here) are also emerging (HPX)
- Improvements such as "endpoints" (multiple addressable communication entities within a single MPI process) may eventually lead to more options than just funneled, serialized, and multiple

Courtesy of Alice Koniges

Outline

- The common core: a quick review
- OpenMP Tasks
- The divide and conquer pattern
- Task group, task loops, and more
- Threadprivate
- The other workshare constructs
- Do across loops
- The OpenMP Memory model
- Point to point synchronization, atomic, and locks
- NUMA systems
- Thread affinity
- Hybrid MPI/OpenMP
- ➡ More about process and thread affinity
- A quick survey of the rest of OpenMP

A naive “srun” causes a mess in process/ thread affinity

Example: 16 MPI tasks x 8 OpenMP threads per task on a single 68-core KNL quad,cache node

```
% export OMP_NUM_THREADS=8
% export OMP_PROC_BIND=spread    (other choice are “close”,”master”,”true”,”false”)
% export OMP_PLACES=threads   (other choices are: cores, sockets, and various ways to specify
explicit lists)
```

```
% srun -n 16  ./xthi |sort -k4n,6n
   Hello from rank 0, thread 0, on nid02304. (core affinity = 0)
   Hello from rank 0, thread 1, on nid02304. (core affinity = 144)  (on physical core 8)
   Hello from rank 0, thread 2, on nid02304. (core affinity = 17)
   Hello from rank 0, thread 3, on nid02304. (core affinity = 161)       (on physical core 25)
   Hello from rank 0, thread 4, on nid02304. (core affinity = 34)
   Hello from rank 0, thread 5, on nid02304. (core affinity = 178)       (on physical core 42)
   Hello from rank 0, thread 6, on nid02304. (core affinity = 51)
   Hello from rank 0, thread 7, on nid02304. (core affinity = 195)       (on physical core 59)
   Hello from rank 1, thread 0, on nid02304. (core affinity = 0)
   Hello from rank 1, thread 1, on nid02304. (core affinity = 144)   (on physical core 8)
    ...
```

It is a mess!

Cori KNL: the importance of “srun” -c and --cpu_bind Options

- The reason: 68 is not divisible by #MPI tasks!
  - Each MPI task is getting 68x4 / #MPI tasks of logical cores as the domain size
  - MPI tasks are crossing tile boundaries
- Let’s set number of logical cores per MPI task (-c) manually by wasting extra 4 cores on purpose, which is 256 / #MPI tasks
  - Meaning to use 64 cores only on the 68-core KNL node, and spread the logical cores allocated to each MPI task evenly among these 64 cores

```
% srun -n 16 -c 16 --cpu_bind=cores ./xthi
Hello from rank 0, thread 0, on nid09244. (core affinity = 0)
Hello from rank 0, thread 1, on nid09244. (core affinity = 136) (on physical core 0)
Hello from rank 0, thread 2, on nid09244. (core affinity = 1)
Hello from rank 0, thread 3, on nid09244. (core affinity = 137) (on physical core 1)
 …
```

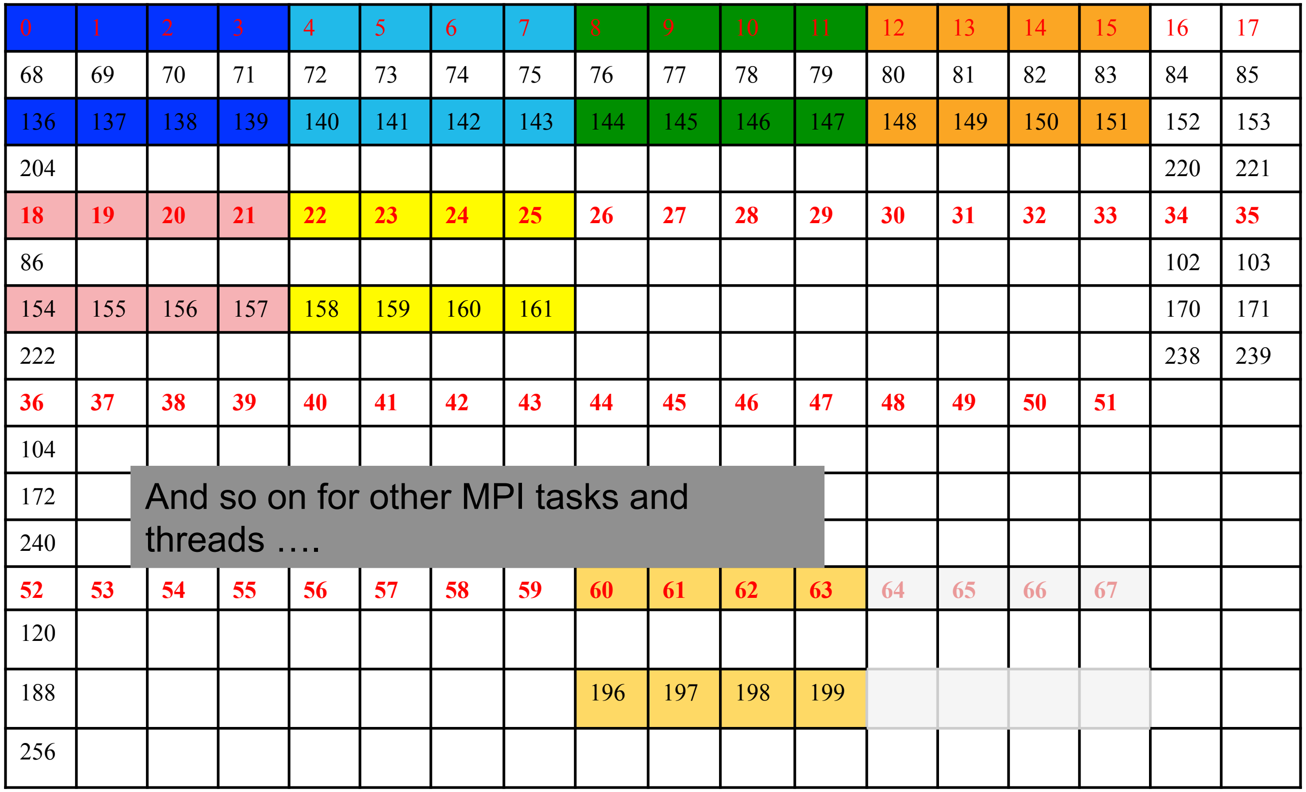
  - Now It looks good!

```
Similarly with Intel MPI:
% export I_MPI_PIN_DOMAIN=16
% mpirun -n 16 ./xthi
```

Now it looks good!

Process/thread affinity are good! (Marked first 6 and last MPI tasks only)

- 🟦 MPI rank 0
- 🟦 MPI rank 1
- 🟩 MPI rank 2
- 🟧 MPI rank 3
- 🟥 MPI rank 4
- 🟨 MPI rank 5
- ….
- 🟨 MPI rank 15



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85
136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153
204																220	221
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
86																102	103
154	155	156	157	158	159	160	161									170	171
222																238	239
36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51		
104																	
172																	
240																	
52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67		
120																	
188								196	197	198	199						
256																	

And so on for other MPI tasks and threads ….

Cori: essential runtime settings for process/thread affinity

- Use srun -c and --cpu_bind flags to bind tasks to CPUs
  - **-c <n> (or --cpus-per-task=n) allocates (reserves) n CPUs per task (process). It helps to evenly spread MPI tasks**
  - Use **--cpu_bind=cores** (no hyperthreads) or **--cpu_bind=threads** (if hyperthreads are used)
- Use OpenMP envs, OMP_PROC_BIND and OMP_PLACES to fine pin each thread to a subset of CPUs allocated to the host task
  - Different compilers may have different default values for them
  - The following provide compatible thread affinity among Intel, GNU and Cray compilers:

OMP_PROC_BIND=true # Specifying threads may not be moved between CPUs

OMP_PLACES=threads # Specifying a thread should be placed in a single CPU

Cori: essential runtime settings for MCDRAM memory affinity

- In quad,cache mode, no special setting is needed to use MCDRAM
- In quad,flat mode, using quad,flat as an example
  - NUMA node 1 is MCDRAM
- Enforced memory mapping to MCDRAM
  - If using >16 GB, malloc will fail
  - Use “**numactl -m 1 ./myapp**” as the executable (instead of “./myapp”)
  - Or add **“--mem_bind=map_mem:1**” as an “srun” flag
- Preferred memory mapping to MCDRAM
  - If using >16 GB, malloc will spill to DDR
  - Use “**numactl -p 1 ./myapp**” as the executable (instead of “./myapp”)
  - Or add **“--mem_bind=preferred:map_mem:1**” as an “srun” flag

Cori: sample job script to run on KNL quad,cache nodes

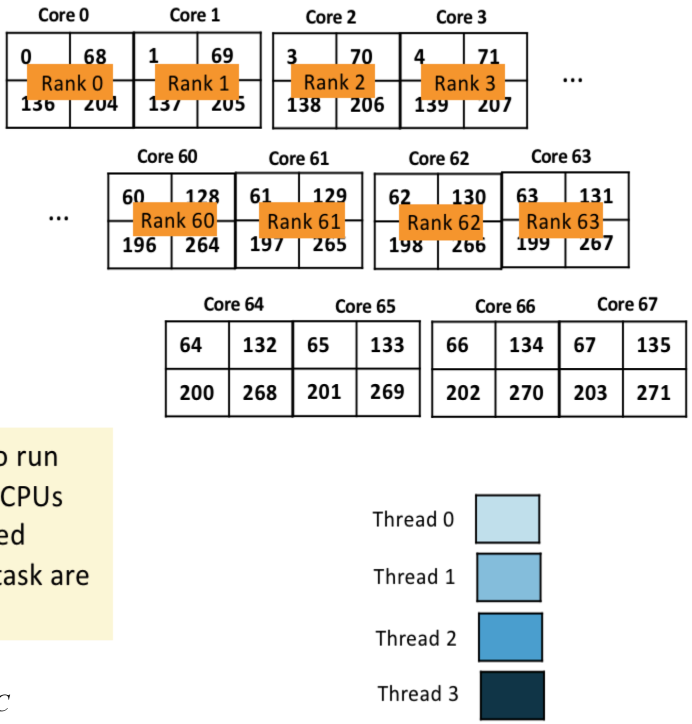
Sample Job script (MPI+OpenMP)

```
#!/bin/bash -l
#SBATCH -N 2
#SBATCH -q regular
#SBATCH -t 1:00:00
#SBATCH -L SCRATCH
#SBATCH -C knl,quad,cache

export OMP_NUM_THREADS=4
srun -n 128 -c 4 --cpu_bind=cores ./a.out
```

This job script requests 2 KNL quad, cache nodes to run 128 MPI tasks, 64 MPI tasks per node, allocating 4 CPUs per task, and binds each task to the 4 CPUs allocated within each core. The 4 OpenMP threads per MPI task are bound to the 4 CPUs in the core.

Courtesy of Zhengji Zhao, NERSC

Process and thread affinity



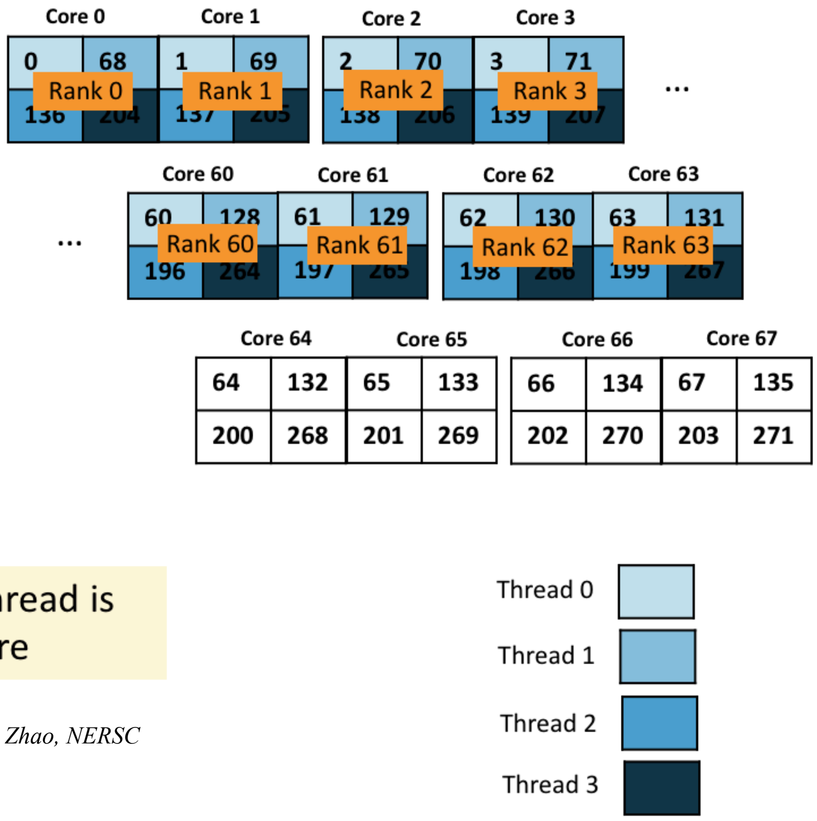
Cori: sample job script to run on KNL quad,cache nodes

Sample Job script (MPI+OpenMP)

```
#!/bin/bash -l
#SBATCH -N 2
#SBATCH -q regular
#SBATCH -t 1:00:00
#SBATCH -L SCRATCH
#SBATCH -C knl,quad,cache

export OMP_PROC_BIND=true
export OMP_PLACES=threads
export OMP_NUM_THREADS=4
srun -n 128 -c 4 --cpu_bind=cores ./a.out
```

With the above two OpenMP envs, each thread is now pinned to a single CPU within each core

Process and thread affinity



Courtesy of Zhengji Zhao, NERSC

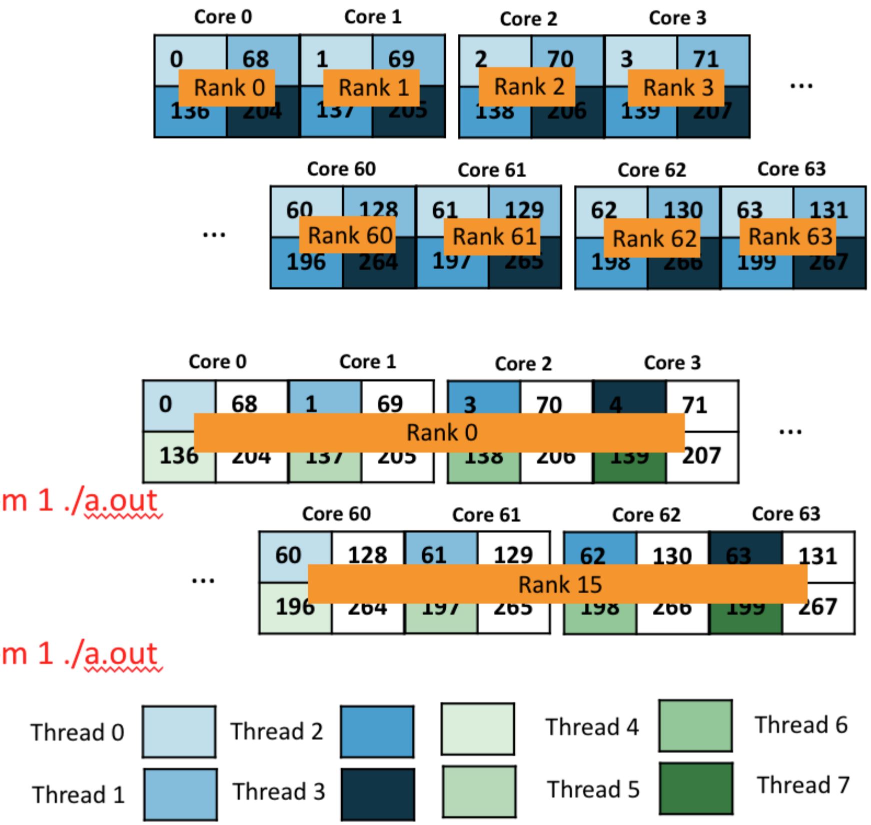
Cori: sample job script to run on KNL quad,flat nodes

Sample Job script (MPI+OpenMP)

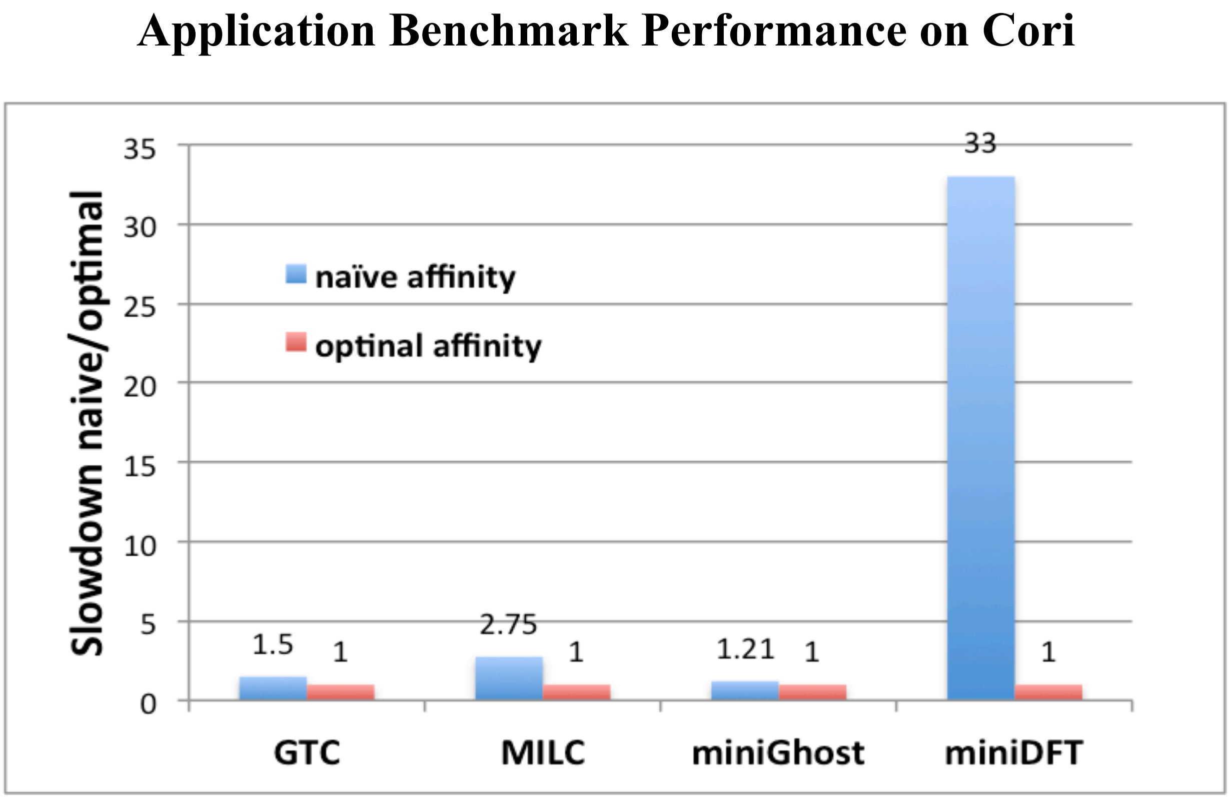
```
#!/bin/bash -l
#SBATCH -N 2
#SBATCH -q regular
#SBATCH -t 1:00:00
#SBATCH -C knl,quad,flat

export OMP_PROC_BIND=true
export OMP_PLACES=threads

export OMP_NUM_THREADS=4
srun -n 128 -c 4 --cpu_bind=cores numactl -m 1 ./a.out

export OMP_NUM_THREADS=8
srun -n 32 -c 16 --cpu_bind=cores numactl -m 1 ./a.out
```

Adapted from Zhengji Zhao, NERSC

Process and thread affinity



Naïve vs. optimal affinity



Exercise: hybrid MPI/OpenMP affinity choices

- Hybrid MPI/OpenMP code: xthi.c
- Nested OpenMP code: xthi-nested.c
- Sample output:

```
% srun -n 2 -c 32 --cpu_bind=cores ./xthi |sort -k4n,6n
 Hello from rank 0, thread 0, on nid00019. (core affinity = 0)
 Hello from rank 0, thread 1, on nid00019. (core affinity = 2)
  Hello from rank 0, thread 2, on nid00019. (core affinity = 4) …
```

- Experiment with different OMP_NUM_THREADS, OMP_PROC_BIND, and OMP_PLACES settings to check thread affinity on different compute node architectures (for example, Cori Haswell and KNL).
- Try different compilers. Compile your own or use our pre-built binaries: check-hybrid.cori.intel, check-hybrid.cori.gnu
  - Compare OMP_PROC_BIND=spread vs OMP_PROC_BIND=true for gnu compiler

Upcoming in OpenMP 5.0: Display Affinity

- Berkeley Lab proposed to have the display affinity support.
  - Accepted into TR6 for OpenMP 5.0
- Two runtime environment variables
  - OMP_DISPLAY_AFFINITY
  - OMP_DISPLAY_AFFINITY_FORMAT
- Runtime APIs to get/set the thread affinity info
  - omp_get_display_affinity, omp_set_display_affinity
  - omp_get_affinity_format, omp_set_affinity_format
  - omp_display_affinity
  - omp_capture_affinity # write into buffer

OMP_AFFINITY_FORMAT fields

Short Name	Long name	Meaning
L	thread_level	from omp_get_level()
n	thread_num	from omp_get_thread_num()
a	thread_affinity	the numerical identifiers of the processors the current thread is binding to, in the format of a comma separated list of OpenMP thread places or a range of thread places described with non-negative numbers as those used in OMP_PLACES.
h	host	host or node name
p	process_id	process id used by the implementation (such as the process id for the MPI process)
N	num_threads	from omp_get_num_threads()
A	ancestor_tnum	from omp_get_ancestor_thread_num(). One level up only.

Sample display using format "Thread Affinity: %0.5L, %.10n, %.20a”:

```
Thread Affinity:00001,          0,              0-3
Thread Affinity:00001,          1,              4-7
```

Outline

- The common core: a quick review
- OpenMP Tasks
- The divide and conquer pattern
- Task group, task loops, and more
- Threadprivate
- The other workshare constructs
- Do across loops
- The OpenMP Memory model
- Point to point synchronization, atomic, and locks
- NUMA systems
- Thread affinity
- Hybrid MPI/OpenMP
- More about process and thread affinity
- A quick survey of the rest of OpenMP

An Outline of OpenMP

- Parallel Construct and associated clauses
  - Create a team of threads
- Data Environment
  - Controlling how data is shared
- Worksharing constructs:
  - Splitting up work among a team of threads
- SIMD constructs
  - Explicit vectorization
- Task Constructs
  - Create and manage explicit tasks
- Device Constructs
  - Offloading work to GPUs, many-core CPUs, and other attached devices

- Synchronization (and the master construct)
  - Add order constraints to your parallel program
- Cancelation
  - Ending work in a controlled manner
- User Defined Reductions
  - Generalizing the reduction concept
- Combined and composite constructs
  - Type less and occasionally new semantics
- Runtime libraries and environment variables

An Outline of OpenMP

- Parallel Construct and associated clauses
  - Create a team of threads
- Data Environment
  - Controlling how data is shared
- Worksharing constructs:
  - Splitting up work among a team of threads
- SIMD constructs
  - Explicit vectorization
- Task Constructs
  - Create and manage explicit tasks
- Device Constructs
  - Offloading work to GPUs, many-core CPUs, and other attached devices

- Synchronization (and the master construct)
  - Add order constraints to your parallel program
- Cancelation
  - Ending work in a controlled manner
- User Defined Reductions
  - Generalizing the reduction concept
- Combined and composite constructs
  - Type less and occasionally new semantics
- Runtime libraries and environment variables

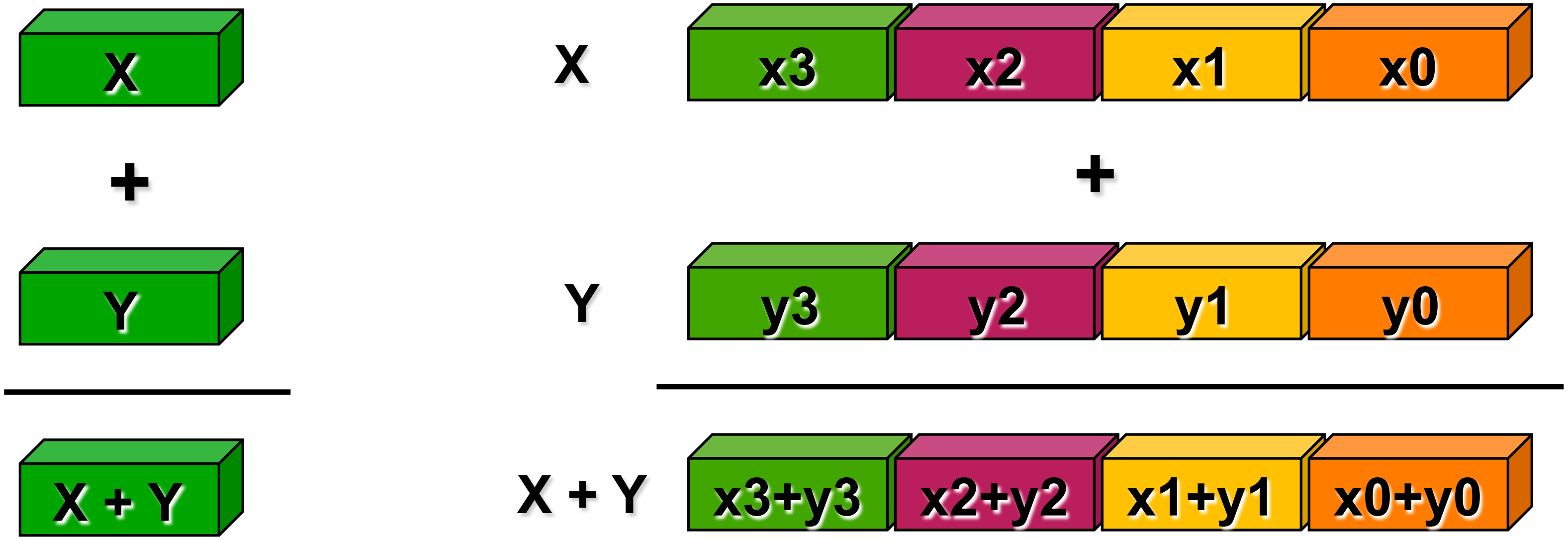
We covered the key points for most of OpenMP but we've said little or nothing about four topics: SIMD, Devices, Cancelation and User Defined Reductions

The Rest of OpenMP

- SIMD
- Devices
- Cancelation
- User Defined Reductions

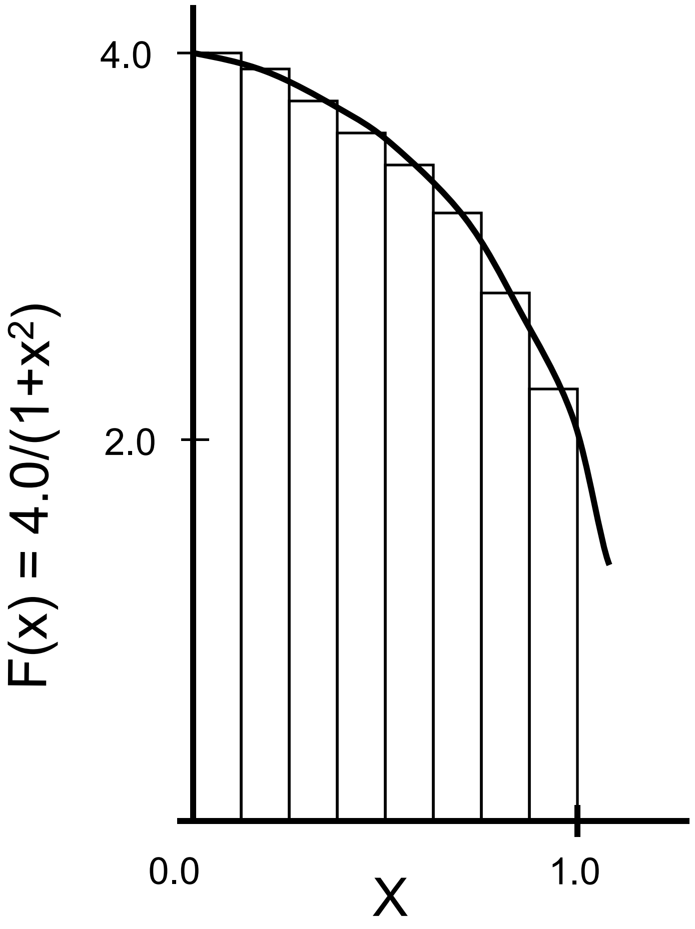
Vector SIMD (single instruction, multiple data)

- A functional unit typically associated with a CPU core takes a single stream of instructions that are applied in parallel to the elements of values in special vector registers.
  - SSE, 128 bits. 2 DP or 4 SP
  - AVX, 256 bits, 4 DP or 8 SP (Haswell)
  - AVX-512, 512 bits, 8 DP or 16 SP (KNL)
- Vector instructions usually generated by the compiler “automatically” from loops
- Best performance may require explicit coding.



Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

Example Problem:
Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\, dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial PI program

Compile as (O3 no-vec), 0.0052 secs
Compile as (O3 autovec), 0.0023 secs

```
static long num_steps = 8388608;
float step;
int main ()
{           int i;      float x, pi, sum = 0.0;

            step = 1.0/(float) num_steps;

            for (i=0;i< num_steps; i++){
                      x = (i+0.5f)*step;
                      sum = sum + 4.0f/(1.0f+x*x);
            }
            pi = step * sum;
}
```

Note that literals (such as 4.0, 1.0 and 0.5) are explicitly declared as floats. This is very important when trying to get code to vectorize … mixing types can kill vectorization.

Normally, I'd use double types throughout to minimize roundoff errors especially on the accumulation into sum. But to maximize impact of vectorization for these exercise, we'll use float types.

Pi Program: Explicit Vectorization with intriniscs (SSE)

```
 float pi_sse(int  num_steps)
{  float scalar_one =1.0, scalar_zero = 0.0,  ival, scalar_four =4.0, step, pi, vsum[4];
   step = 1.0/(float) num_steps;

   __m128 ramp   = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);
   __m128 one     = _mm_load1_ps(&scalar_one);
   __m128 four    = _mm_load1_ps(&scalar_four);
   __m128 vstep  = _mm_load1_ps(&step);
   __m128 sum    = _mm_load1_ps(&scalar_zero);
   __m128 xvec;   __m128 denom;  __m128 eye;

  for (int i=0;i< num_steps; i=i+4){          // unroll loop 4 times
     ival       = (float)i;                    // and assume num_steps%4 = 0
     eye       = _mm_load1_ps(&ival);
     xvec     = _mm_mul_ps(_mm_add_ps(eye,ramp),vstep);
     denom  = _mm_add_ps(_mm_mul_ps(xvec,xvec),one);
     sum     = _mm_add_ps(_mm_div_ps(four,denom),sum);
   }
   _mm_store_ps(&vsum[0],sum);
   pi = step * (vsum[0]+vsum[1]+vsum[2]+vsum[3]);
  return pi;
}
```

O3 (no-vec), 0.0052 secs
O3 (autovec), 0.0023 secs
SSE intrinsics, 0.00168 secs

Explicit Vectorization PI program

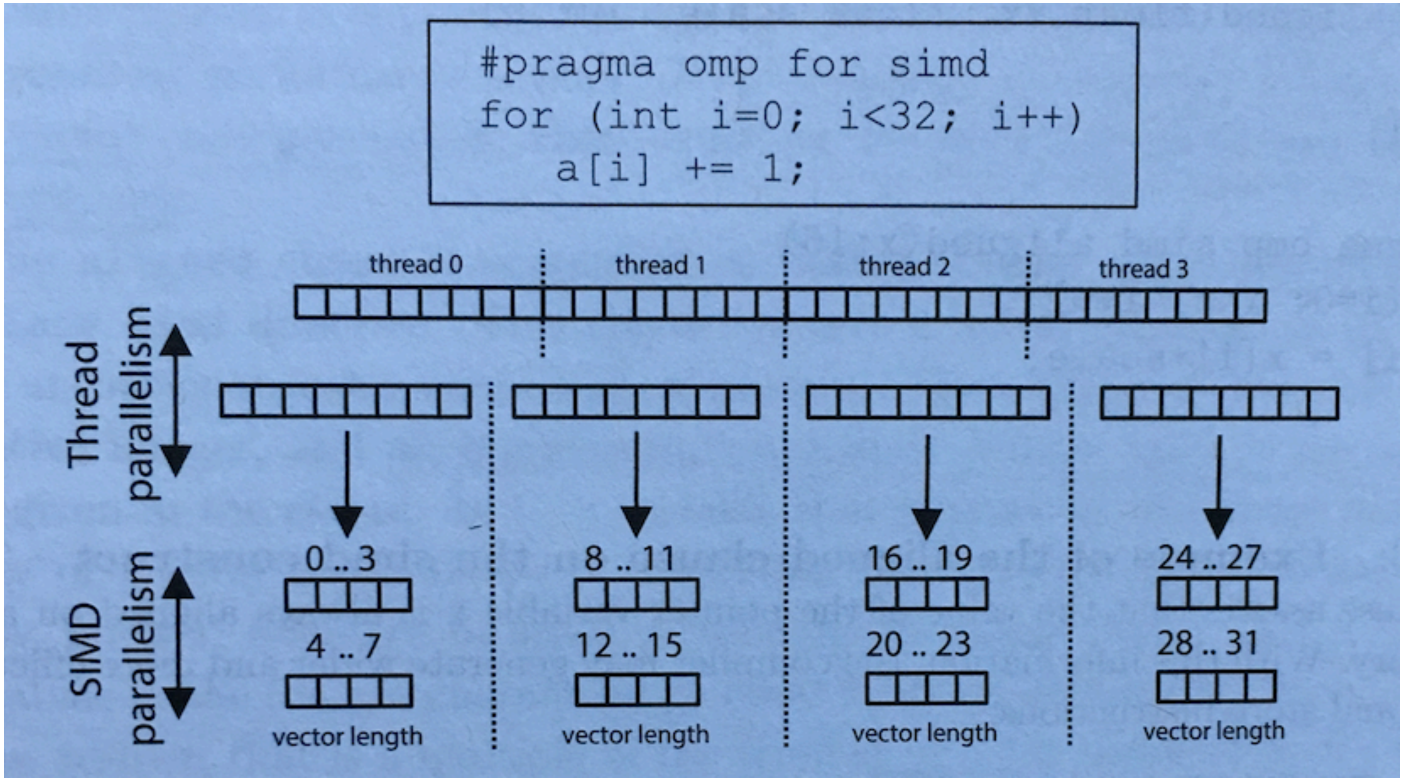
```
static long num_steps = 100000;
float step;
int main ()
{           int i;     float x, pi, sum = 0.0;

            step = 1.0/(float) num_steps;
            #pragma omp for simd reduction(+:sum)
            for (i=0;i< num_steps; i++){
                      x = (i+0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

Explicit Vectorization PI program

You can combine with parallel for to get threads and SIMD

```
static long num_steps = 100000;
float step;
int main ()
{           int i;      float x, pi, sum = 0.0;

            step = 1.0/(float) num_steps;
            #pragma omp parallel for simd reduction(+:sum)
            for (i=0;i< num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

Illustration of combining thread and SIMD parallelism



```
#pragma omp for simd
for (int i=0; i<32; i++)
   a[i] += 1;
```

Figure from "Using OpenMP - the Next Step" book by Ruud van de Pas et.al.

Pi Program: Vector intriniscs plus OpenMP

```
float pi_sse(int  num_steps)
{  float scalar_one =1.0, scalar_zero = 0.0,  ival, scalar_four =4.0, step, pi, vsum[4];
  float local_sum[NTHREADS];   // set NTHREADS elsewhere, often to num of cores
  step = 1.0/(float) num_steps;  pi = 0.0;
 #pragma omp parallel
 {    int i, ID=omp_get_thread_num();
      __m128 ramp   = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);
      __m128 one    = _mm_load1_ps(&scalar_one);
      __m128 four   = _mm_load1_ps(&scalar_four);
      __m128 vstep  = _mm_load1_ps(&step);
      __m128 sum    = _mm_load1_ps(&scalar_zero);
      __m128 xvec;   __m128 denom;  __m128 eye;
     #pragma omp for
     for (int i=0;i< num_steps; i=i+4){
        ival      = (float)i;
        eye       = _mm_load1_ps(&ival);
        xvec      = _mm_mul_ps(_mm_add_ps(eye,ramp),vstep);
        denom     = _mm_add_ps(_mm_mul_ps(xvec,xvec),one);
        sum       = _mm_add_ps(_mm_div_ps(four,denom),sum);
     }
    _mm_store_ps(&vsum[0],sum);
    local_sum[ID] = step * (vsum[0]+vsum[1]+vsum[2]+vsum[3]);
 }
  for(int k = 0; k<NUM_THREADS;k++) pi+=local_sum[k];
 return pi;
}
```

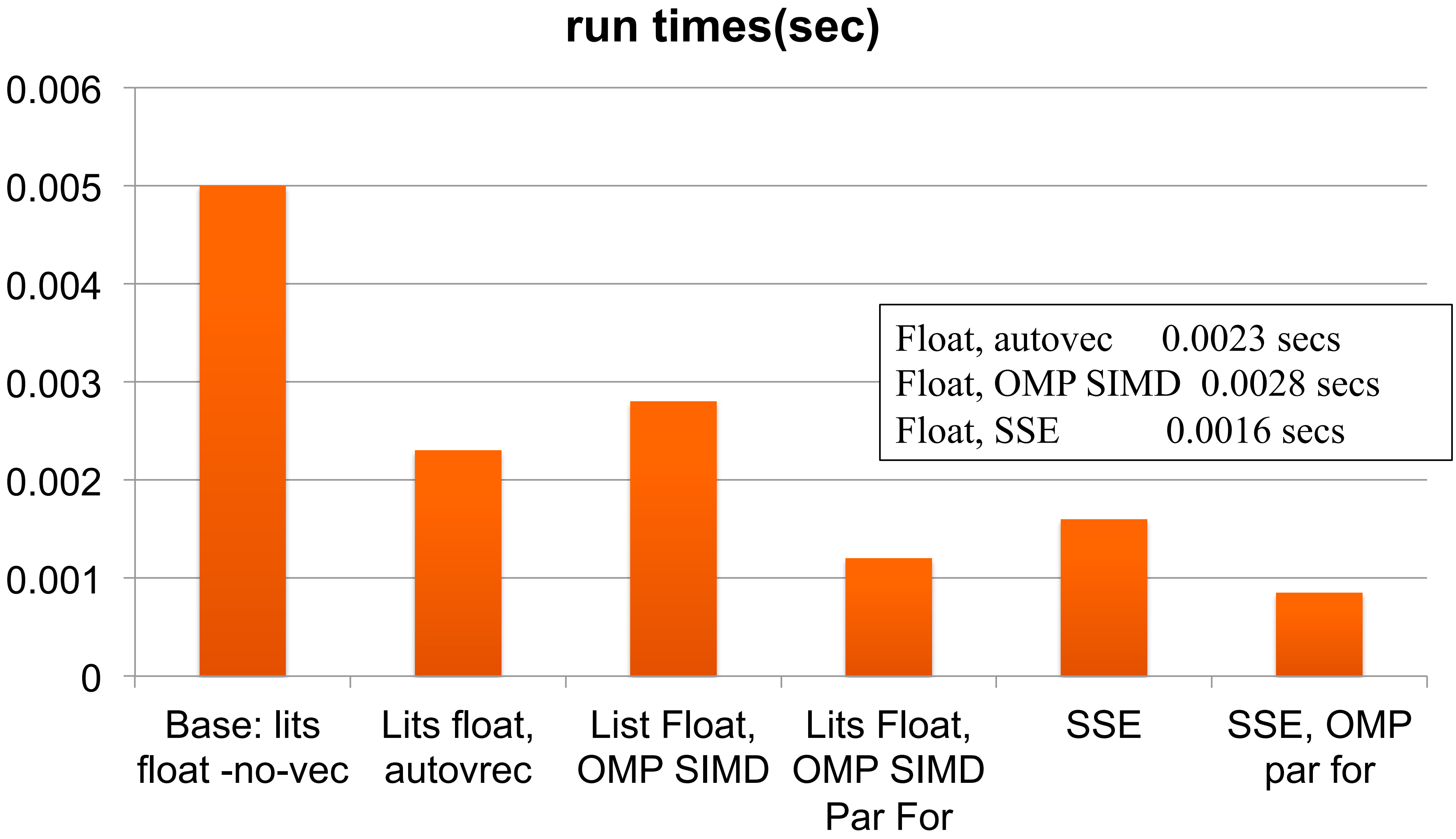
To parallelize with OpenMP:

1. Promote local_sum to an array to there is a variable private to each thread but available after the parallel region
2. Add parallel region and declare vector registers inside the parallel region so each thread has their own copy.
3. Add workshop loop (for) construct
4. Add local sums after the parallel region to create the final value for pi

PI program Results:

4194304 steps

Times in Seconds (50 runs, min time reported)



Float, autovec	0.0023 secs
Float, OMP SIMD	0.0028 secs
Float, SSE	0.0016 secs

- Intel Core i7, 2.2 Ghz, 8 GM 1600 MHz DDR3, Apple MacBook Air OS X 10.10.5.
- Intel(R) C Intel(R) 64 Compiler XE for applications running on Intel(R) 64, Version 15.0.3.187 Build 20150408

SIMD construct for explicit vectorization

- #pragma omp simd [clause[[,] clause], …]
- Where common clauses are:

safelen(length)	Max # of concurrent iterations without breaking a dependence
simdlen(length)	preferred length of SIMD registers
linear(list[: linear-step])	Variables linear relation with iteration number ($x_i = x_{orig} + I * \text{linear-step}$)
aligned(list[: alignment])	list items have given alignment

- Plus the usual private, firstprivate, reduction, and collapse.
- The SIMD construct Applies to a loop in standard form.
- Can be combined with the for construct

SIMD example

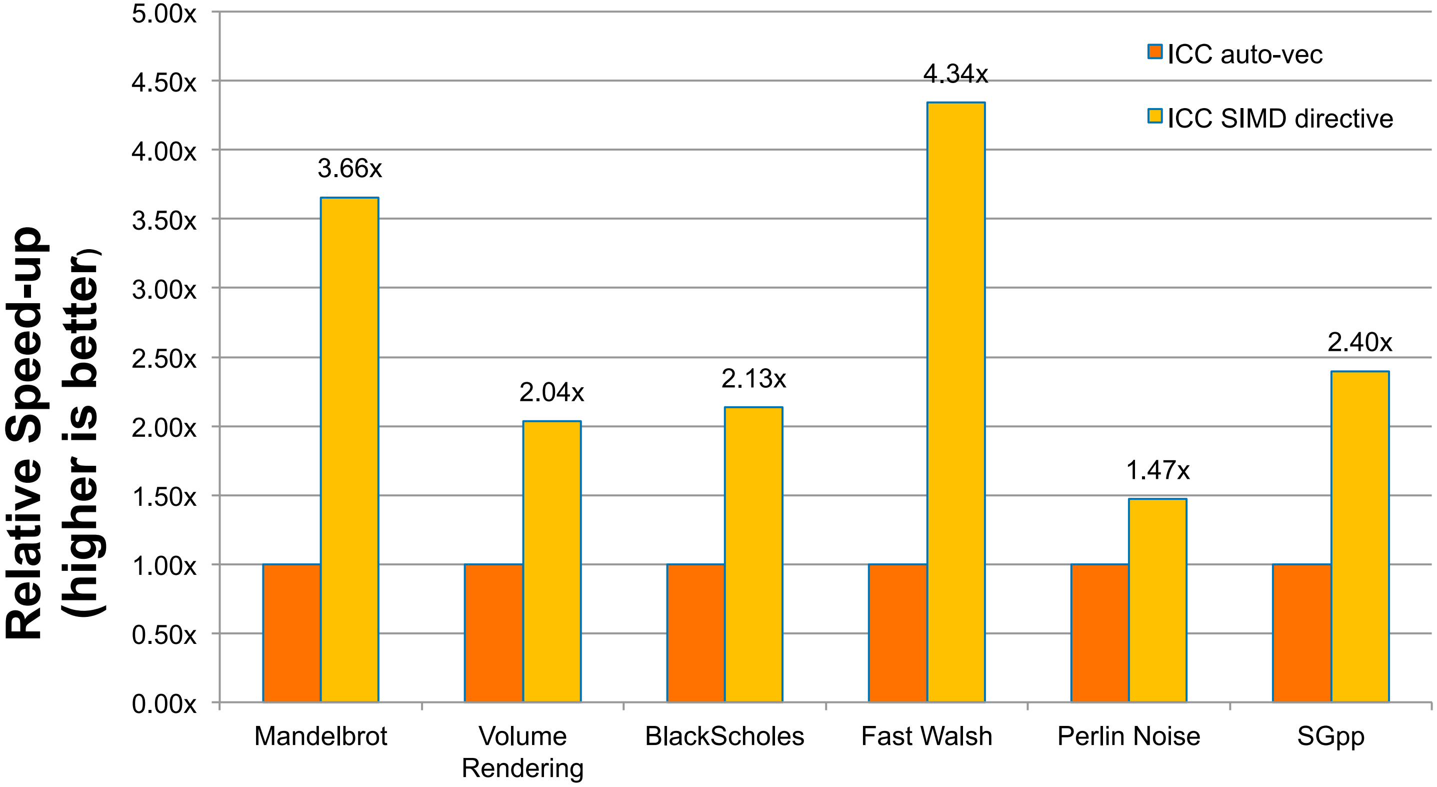
- Explicit control lets you “force” vectorization in cases where the system might not otherwise use the vector units.

```
void work( float *b, int n, int m )
{
    int i;
    #pragma omp simd safelen(16)
     for (i = m; i < n; i++)
             b[i] = b[i-m] - 1.0f;
}
```

As long as the variable m is less than or equal to 16, this program will work correctly

Explicit Vectorization – Performance Impact

Explicit Vectorization looks better when you move to more complex problems.



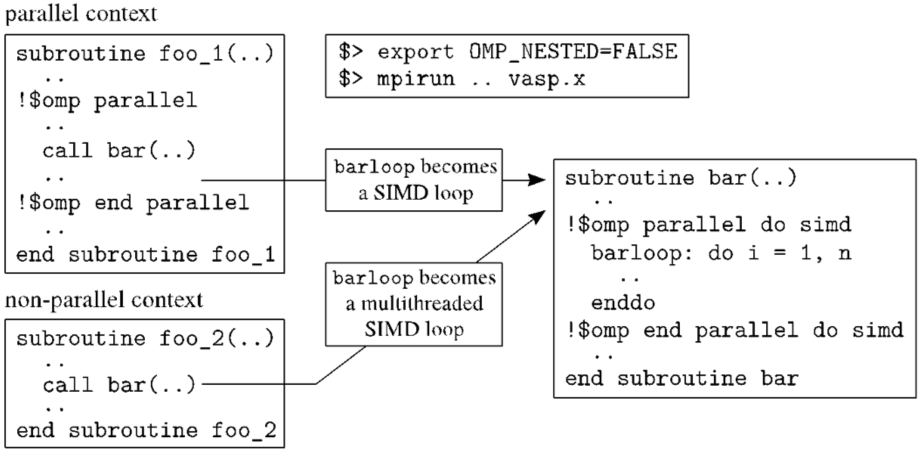
Source: M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell, “Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP”, pages 59-72, Rome, Italy, June 2012. LNCS 7312.

VASP: SIMD Vectorization

SIMD constructs:

- Loop vectorization via !$omp simd
- Function vectorization via !$omp declare simd
- Both could be extended: simdlen(x), aligned(varlist[:alignment]), uniform(varlist)
- Used schedule(simd:static) or schedule(simd:static,x) to match the chunk size with the SIMD width

Leveraging OpenMP parallelization with SIMD via !$omp parallel do simd



parallel context

```
subroutine foo_1(..)
  ..
!$omp parallel
  ..
  call bar(..)
  ..
!$omp end parallel
  ..
end subroutine foo_1
```

non-parallel context

```
subroutine foo_2(..)
  ..
  call bar(..)
  ..
end subroutine foo_2
```

```
$> export OMP_NESTED=FALSE
$> mpirun .. vasp.x
```

barloop becomes a SIMD loop
barloop becomes a multithreaded SIMD loop

```
subroutine bar(..)
  ..
!$omp parallel do simd
  barloop: do i = 1, n
    ..
  enddo
!$omp end parallel do simd
  ..
end subroutine bar
```

Courtesy of Zhengji Zhao, NERSC

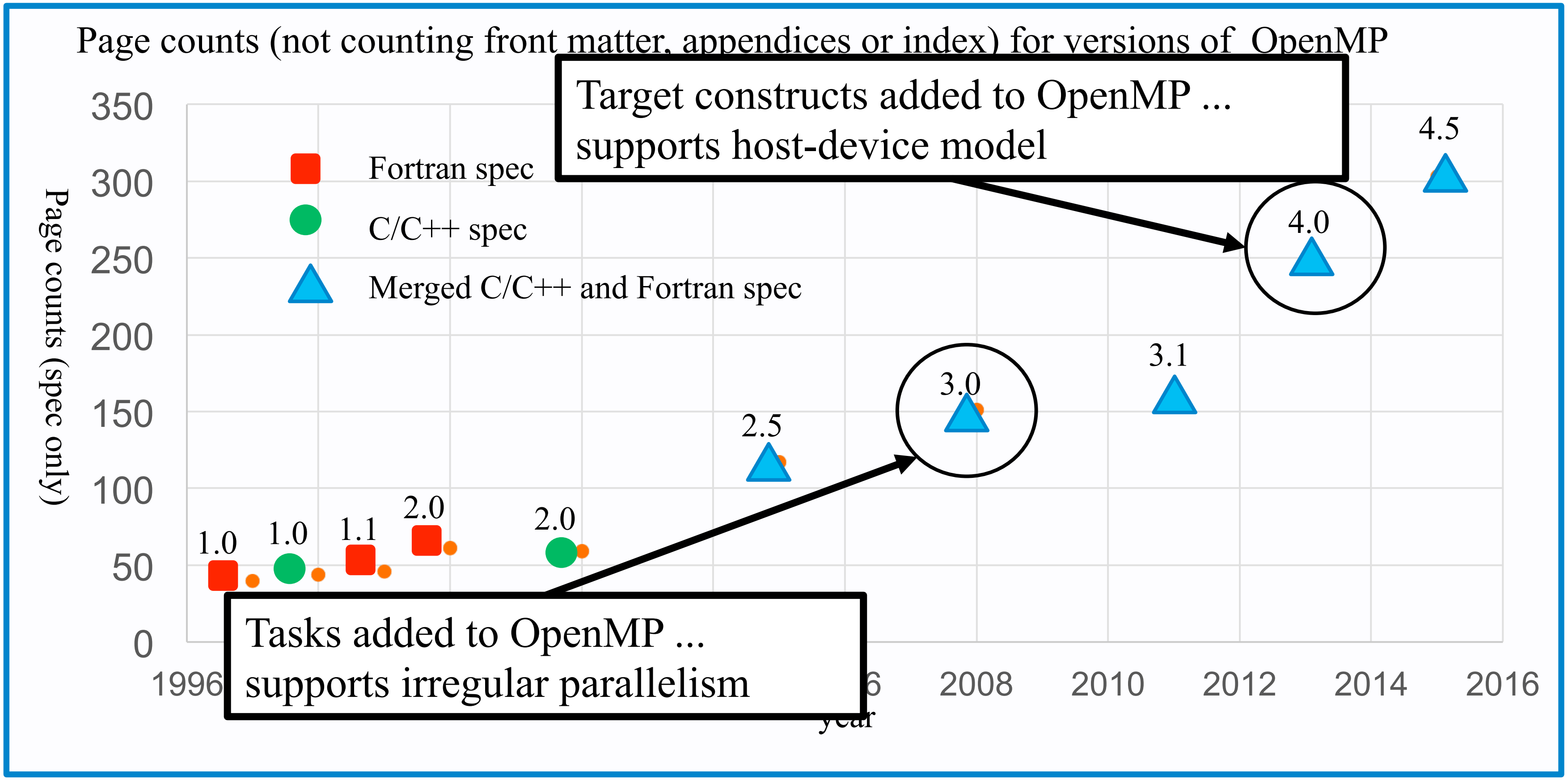
The Rest of OpenMP

- SIMD
- Devices
- Cancelation
- User Defined Reductions

The growth of complexity in OpenMP

- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science.

- The complexity has grown considerably over the years!



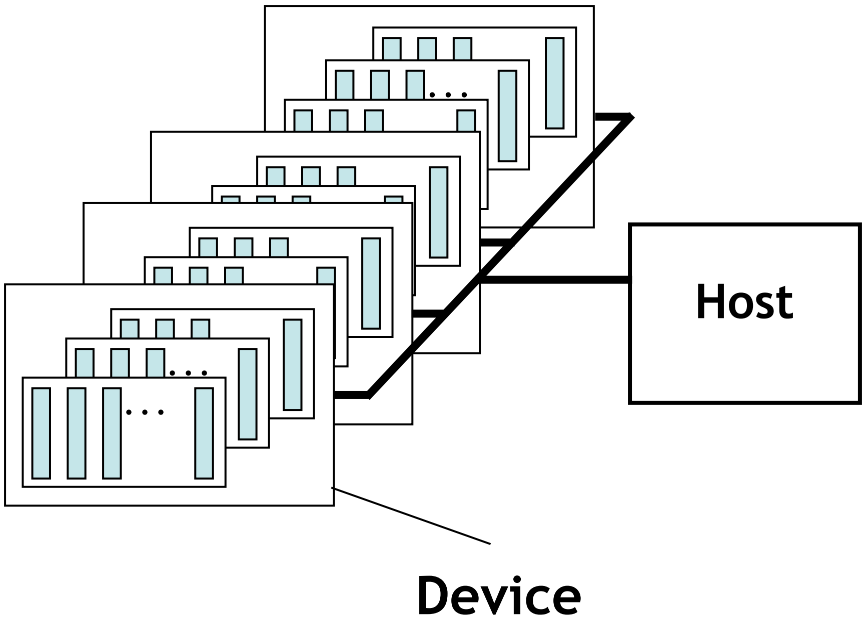
Target constructs added to OpenMP ... supports host-device model

Tasks added to OpenMP ... supports irregular parallelism

The complexity of the full spec is overwhelming, so we focus on the 19 constructs most OpenMP programmers restrict themselves to … the so called "OpenMP Common Core"

The OpenMP device programming model

- OpenMP uses a host/device model
  - The host is where the initial thread of the program begins execution
  - Zero or more devices are connected to the host



```
#include <omp.h>
#include <stdio.h>
int main()
{
   printf(“There are %d devices\n”,
               omp_get_num_devices());
}
```

OpenMP with target devices

- The target construct offloads execution to a device.

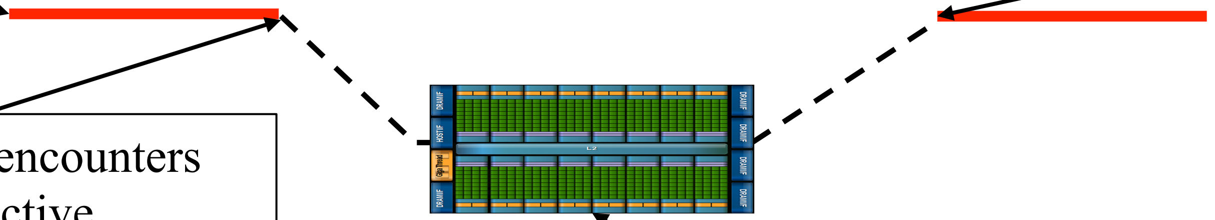
```
#pragma omp target
{….}  // a structured block of code
```

1. Program begins. Launches **Initial thread** running on the **host device**.

2. Implicit parallel region surrounds entire program

3. Initial task begins execution

10. Initial task on host continues once execution associated with the target region completes



4. Initial thread encounters the target directive.

5. Initial task generates a target task which is a mergable, included task

6. Target task launches target region on the device

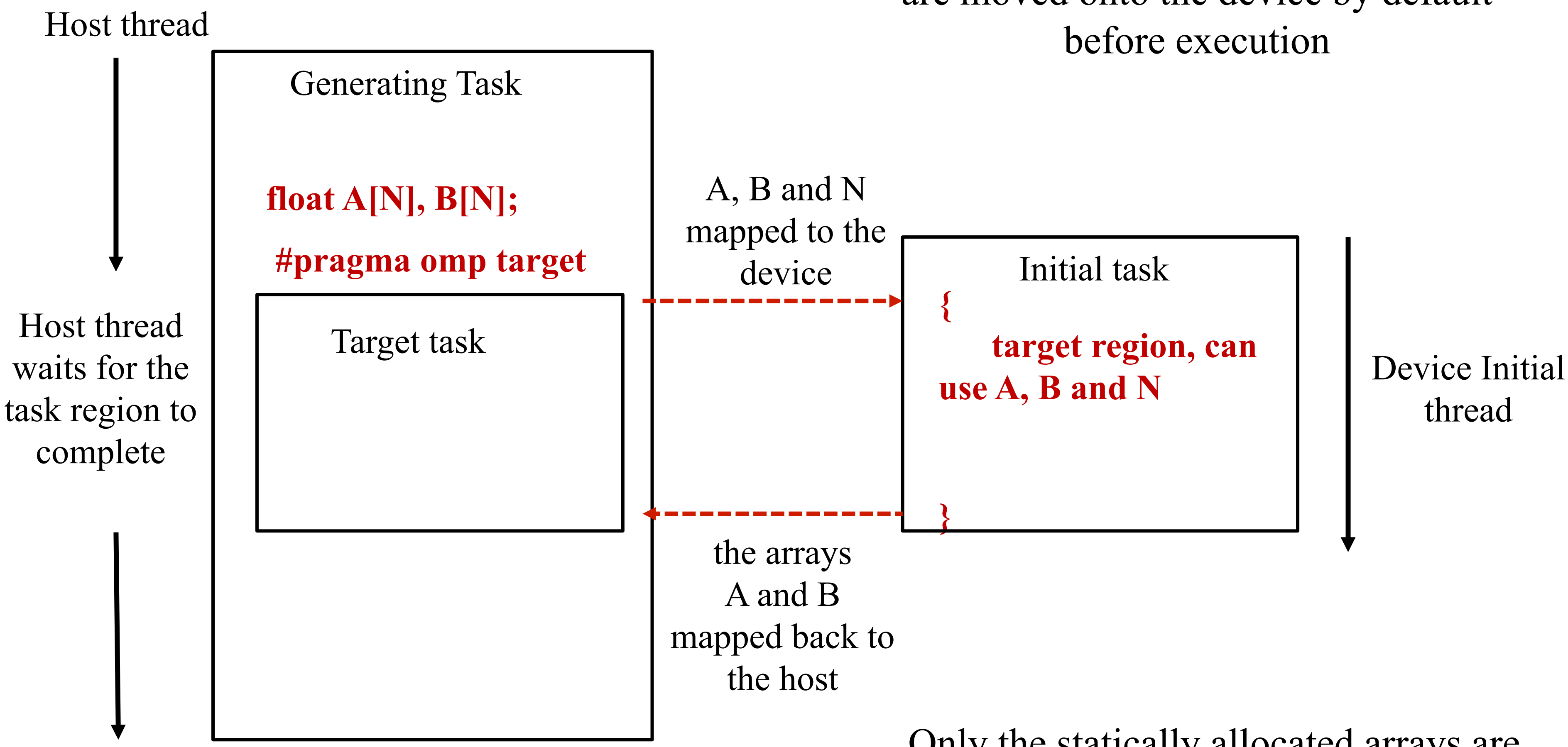
7. A new initial thread runs on the device.

8. Implicit parallel region surrounds device program

9. Initial task executes code in the target region.

The target data environment

Scalars and statically allocated arrays are moved onto the device by default before execution



Host thread

Generating Task

float A[N], B[N];

#pragma omp target

Target task

Host thread waits for the task region to complete

A, B and N mapped to the device

Initial task

{

target region, can use A, B and N

}

the arrays A and B mapped back to the host

Device Initial thread

Only the statically allocated arrays are moved back to the host after the target region completes

Based on figure 6.4 in **Using OpenMP – The Next Step** by van der Pas, Stotzer and Terboven, MIT Press, 2017

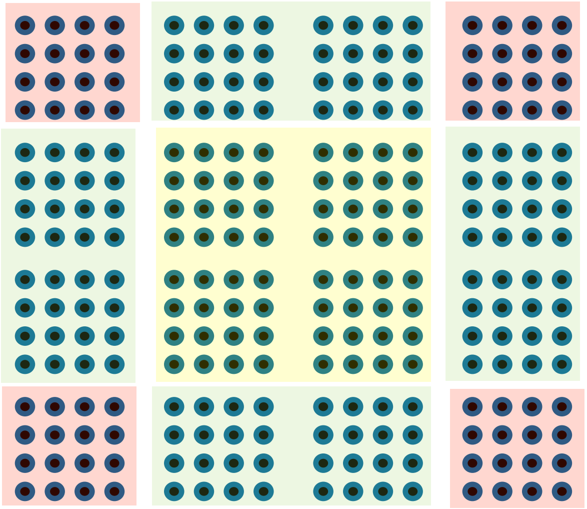
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

1. Turn source code into a scalar work-item

```
extern void reduce(   __local  float*,   __global float*);

__kernel void pi(  const int niters, float  step_size,
         __local  float*  l_sums,  __global float*  p_sums)
{
   int n_wrk_items  = get_local_size(0);
   int loc_id       = get_local_id(0);
   int grp_id    = get_group_id(0);
   float x, accum = 0.0f;     int i,istart,iend;

   istart =   (grp_id * n_wrk_items    + loc_id) * niters;
   iend   = istart+niters;

   for(i= istart; i<iend; i++){
       x = (i+0.5f)*step_size;    accum += 4.0f/(1.0f+x*x); }

   l_sums[local_id] = accum;
   barrier(CLK_LOCAL_MEM_FENCE);
   reduce(l_sums, p_sums);
}
```

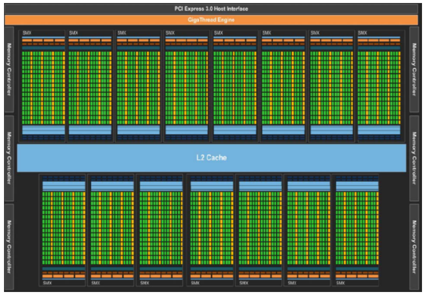
This is OpenCL kernel code … the sort of code the OpenMP compiler generates on your behalf

2. Map work-items onto an an N dim index space.



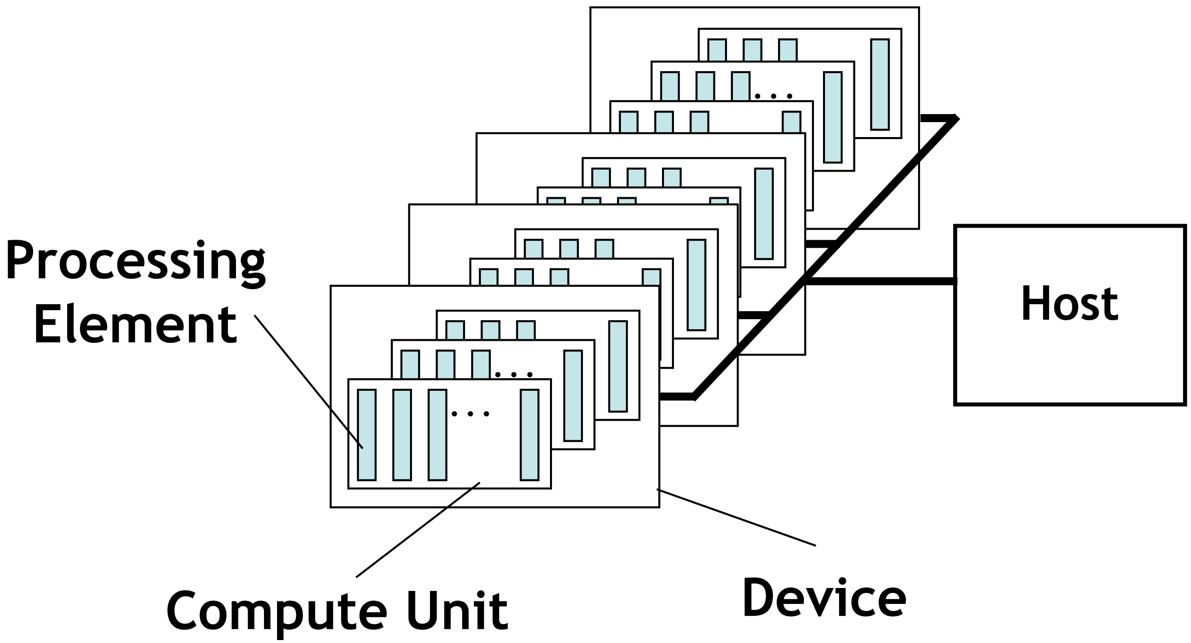
3. Map data structures onto the same index space

4. Run on hardware designed around the same SIMT execution model



Third Party names are the property of their owners

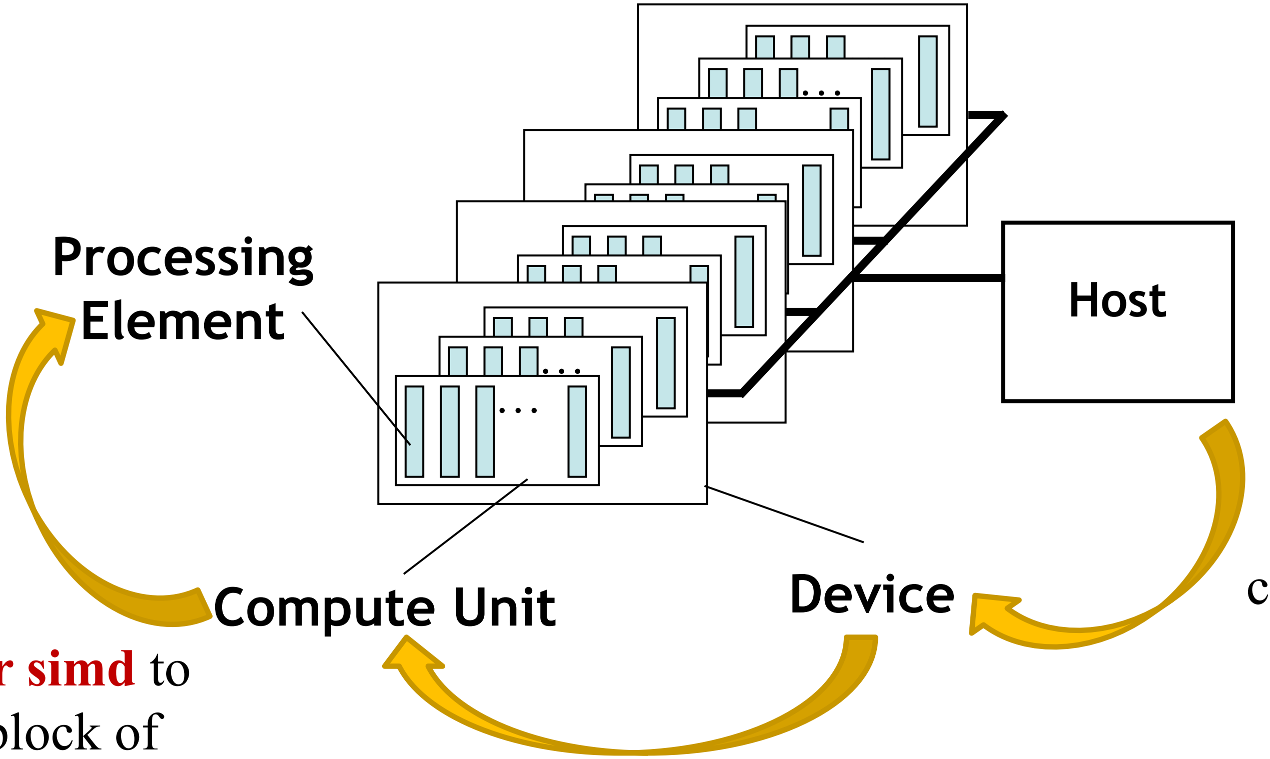
A Generic Host/Device Platform Model



- One ***Host*** and one or more ***Devices***
  - Each Device is composed of one or more ***Compute Units***
  - Each Compute Unit is divided into one or more ***Processing Elements***
- Memory divided into ***host memory*** and ***device memory***

Third party names are the property of their owners.

Our host/device Platform Model and OpenMP



Target construct to get onto a device

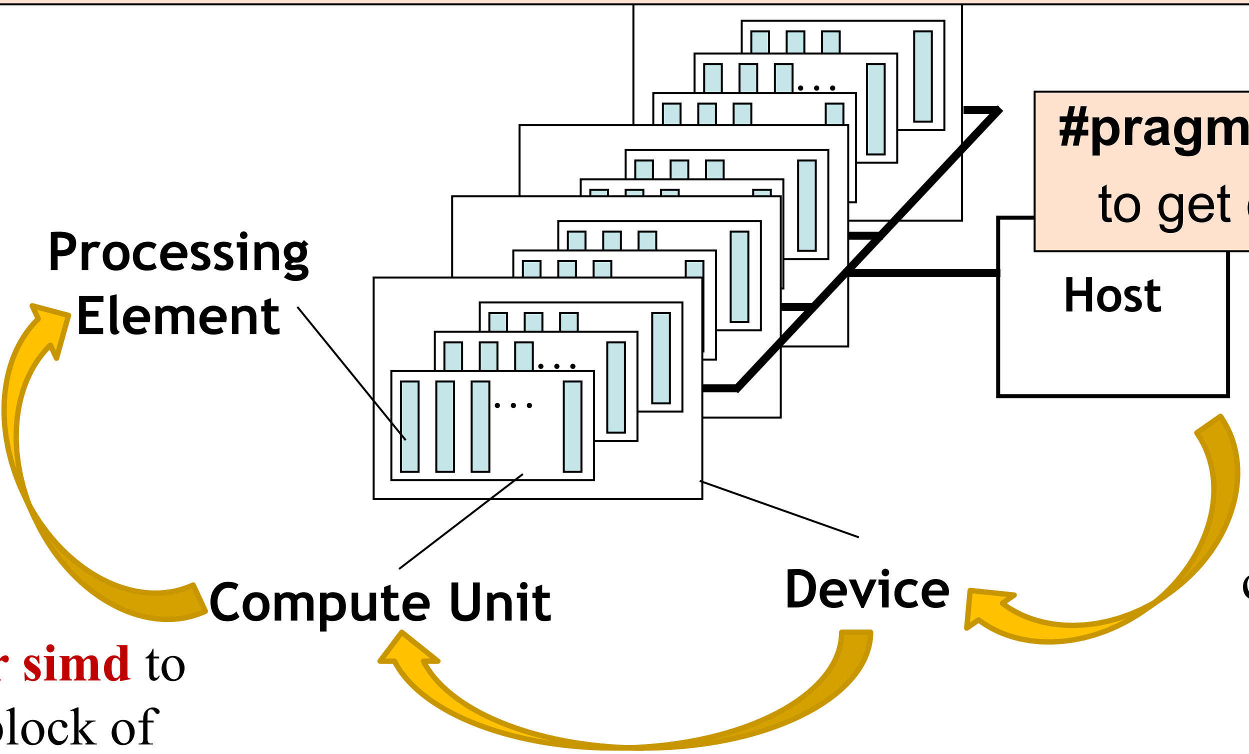
Parallel for simd to run each block of loop iterations on the processing elements

Teams construct to create a league of teams with one team of threads on each compute unit.

Distribute clause to assign blocks of loop iterations to teams.

Our host/device Platform Model and OpenMP

Typical usage ... let the compiler do what's best for the device:



#pragma omp target

to get on the device

Target construct to get onto a device

Parallel for simd to run each block of loop iterations on

Teams construct to create a league of

#pragma omp teams distribute parallel for simd

to assign work to the device processing elements

Distribute clause to assign blocks of loop iterations to teams.

Our running example: a Jacobi solver

- This program uses a Jacobi iterative method to solve a system of linear equations (Ax= b).
- Here is the basic idea behind the method.
  - Rewrite the matrix A as a Lower Triangular (L), upper triangular (U) and diagonal matrix (D):

$$Ax = (L + D + U)x = b$$

  - Carry out the multiplication and rearrange:

$$Dx = b - (L+U)x \quad \text{-->} \quad x = (b-(L+U)x)/D$$

  - Continue in an iterative manner until the error is small enough

x_new = (b-(L+U)x_old)/D

Jacobi Solver (serial)

```
<<< allocate and initialize the matrix A and >>>
<<< vectors x1, x2 and b                     >>>
while((conv > TOL) && (iters<MAX_ITERS))
  {
    iters++;
    xnew = iters % s ? x2 : x1;
    xold   = iters % s ? x1 : x2;

    for (i=0; i<Ndim; i++){
       xnew[i] = (TYPE) 0.0;
       for (j=0; j<Ndim;j++){
          if(i!=j)
            xnew[i]+= A[i*Ndim + j]*xold[j];
       }
       xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

```
    //
    // test convergence
    //
    conv = 0.0;
    for (i=0; i<Ndim; i++){
       tmp  = xnew[i]-xold[i];
       conv += tmp*tmp;
    }
    conv = sqrt((double)conv);

 } // end while loop
```

Jacobi Solver (Par Targ, 1/2)

```
while((conv > TOL) && (iters<MAX_ITERS))
 {
   iters++;
   xnew = iters % 2 ? x2 : x1;
   xold   = iters % 2 ? x1 : x2;
#pragma omp target map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                     map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
    #pragma omp teams distribute parallel for simd private(i,j)
   for (i=0; i<Ndim; i++){
      xnew[i] = (TYPE) 0.0;
      for (j=0; j<Ndim;j++){
         if(i!=j)
           xnew[i]+= A[i*Ndim + j]*xold[j];
      }
      xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
   }
```

Jacobi Solver (Par Targ, 2/2)

```
    //
    // test convergence
    //
    conv = 0.0;
 #pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                   map(to:Ndim) map(tofrom:conv)
       #pragma omp teams distribute parallel for simd  \
                   private(i,tmp) reduction(+:conv)
    for (i=0; i<Ndim; i++){
       tmp  = xnew[i]-xold[i];
       conv += tmp*tmp;
    }
    conv = sqrt((double)conv);

 } \\ end while loop
```

Jacobi Solver (Par Targ, 2/2)

```
    //
    // test convergence
    //
    conv = 0.0;
 #pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                      map(to:Ndim) map(tofrom:conv)
   #pragma omp teams distribute parallel for simd  \
                      private(i,tmp) reduction(+:conv)
    for (i=0; i<Ndim; i++){
       tmp  = xnew[i]-xold[i];
       conv += tmp*tmp;
    }
    conv = sqrt((double)conv);

 } \\ end while loop
```

This worked but the performance was awful. Why?

System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs

Cray® XC40™ Supercomputer running Cray® Compiling Environment 8.5.3.
Intel® Xeon ® CPU E5-2697 v2 @ 2.70GHz with 32 GB DDR3. NVIDIA® Tesla® K20X, 6GB.

Third party names are the property of their owners.

Data movement dominates!!!

```
while((conv > TOLERANCE) && (iters<MAX_ITERS))
  { iters++;
    xnew = iters % s ? x2 : x1;
    xold   = iters % s ? x1 : x2;

    #pragma omp target map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                       map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
     #pragma omp teams distribute parallel for simd private(i,j)
     for (i=0; i<Ndim; i++){
         xnew[i] = (TYPE) 0.0;
         for (j=0; j<Ndim;j++){
             if(i!=j)
               xnew[i]+= A[i*Ndim + j]*xold[j];
         }
         xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
     }
  // test convergence
    conv = 0.0;
    #pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                           map(to:Ndim) map(tofrom:conv)
      #pragma omp teams distribute parallel for private(i,tmp) reduction(+:conv)
      for (i=0; i<Ndim; i++){
          tmp  = xnew[i]-xold[i];
          conv += tmp*tmp;
      }
    conv = sqrt((double)conv);
  }
```

Typically over 4000 iterations!

For each iteration, copy to device (3*Ndim+Ndim^2)*sizeof(TYPE) bytes

For each iteration, copy from device 2*Ndim*sizeof(TYPE) bytes

For each iteration, copy to device 2*Ndim*sizeof(TYPE) bytes

Target data directive

- The **target data** construct creates a target data region.
- You use the map clauses for explicit data management

```
#pragma omp target data map(to: A,B) map(from: C)
 {….}  // a structured block of code
```

- Data copied into the device data environment at the beginning of the directive and at the end
- Inside the **target data** region, multiple **target** regions can work with the single data region

```
#pragma omp target data map(to: A,B) map(from: C)
{
    #pragma omp target
         {do lots of stuff with A, B and C}
    {do something on the host}
    #pragma omp target
         {do lots of stuff with A, B, and C}
}
```

Target update directive

- You can update data between target regions with the target update directive.

```
#pragma omp target data map(to: A,B) map(from: C)
{
    #pragma omp target
        {do lots of stuff with A, B and C}

    #pragma omp update from(A)

    host_do_something_with(A)

    #pragma omp update to(A)

    #pragma omp target
        {do lots of stuff with A, B, and C}
}
```

Copy A on the device to A on the host.

Copy A on the host to A on the device.

Target update details

- #pragma omp target update clause[[[,]clause]...]new-line
- creates a target task to handle data movement between the host and a device

- clause is either motion-clause or one of the following:
  - if(scalar-expression)
  - device(integer-expression)
  - nowait
  - depend (dependence-type : list)
- the motion-clause is one of the following:
  - to(list)
  - from(list)
- This directive generates a target task.
- nowait and depend apply to the target task running on the host.

Jacobi Solver (Par Targ Data, 1/2)

```
  #pragma omp target data map(tofrom:x1[0:Ndim],x2[0:Ndim]) \
                    map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
while((conv > TOL) && (iters<MAX_ITERS))
  {  iters++;
     // alternate x vectors.
    xnew  = iters % 2 ? x2 : x1;
    xold   =  iters % 2 ? x1 : x2;

#pragma omp target
      #pragma omp teams distribute parallel for simd private(j)
    for (i=0; i<Ndim; i++){
       xnew[i] = (TYPE) 0.0;
       for (j=0; j<Ndim;j++){
          if(i!=j)
            xnew[i]+= A[i*Ndim + j]*xold[j];
       }
       xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

Jacobi Solver (Par Targ Data, 2/2)

```
   //
   // test convergence
   //
 conv = 0.0;
#pragma omp target map(tofrom: conv)
{
#pragma omp teams distribute parallel for simd  \
                           private(tmp) reduction(+:conv)
   for (i=0; i<Ndim; i++){
      tmp  = xnew[i]-xold[i];
      conv += tmp*tmp;
   }
} // end target region
 conv = sqrt((double)conv);
} // end while loop
```

System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs
	Above plus target data region	18.37 secs

Third party names are the property of their owners.

The Rest of OpenMP

- SIMD
- Devices
- Cancelation
- User Defined Reductions

Cancellation

- Sometimes you want an OpenMP construct to shut down gracefully
  - Error condition that prevent threads from continuing
  - The work is done.

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

      res=do_some_work();

      if ( res == DONE ){
         #pragma omp cancel parallel
      }

      do_more_work();

      #pragma omp cancelation point
   }
}
```

The thread that encounters this pragma signals cancellation and ends execution of parallel region

Threads check for cancellation signal and end their execution of the parallel region

- Cancellation: parallel, taskgroup, sections and worksharing loops
- OMP_CANCELLATION environment variable must be set to true to enable cancelation.

The Rest of OpenMP

- SIMD
- Devices
- Cancelation
- User Defined Reductions

User Defined Reductions

- What if you need a reduction in OpenMP, but the standard built in reductions do not cover your needs?
- OpenMP added a capability for user defined reductions.
- The declare reduction directive

```
#pragma omp declare reduction (reduction_identifier : typename :  \
          combiner) [initializer-clause]
```

name	Description
Reduction_identifier	A C/C++ identifier. May be one of the existing, predefined reduction operators
typename	The name of a type or list of types if the reduction applies at different instances to different types
combiner	A function or expression for pairwise combination of results from threads using variables: omp_orig: value of "original variable" from scope prior to the reduction omp_priv: value used to initialize private, reduction variables omp_in, omp_out: variables from each thread with result in omp_out
Initializer-clause	2 forms: omp_priv = initializer or omp_priv = function(argument_list)

UDR Example

```
#define N       128
int main()
{
   int *a;
   int result = INT_MAX;
   // create and initialize the array a

   // declare the user defined reduction
   #pragma omp declare reduction (my_abs_min : int :   \
        omp_out = abs(omp_in) < omp_out ? abs(omp_in) : abs (omp_out)) \
        initializer (omp_priv = INT_MAX)

   #pragma omp parallel for reduction(my_abs_min:result)
   for (int i=0; i<N; i++){
      if (abs(a[i] < result))
          result = abs(a[i]);
   }
   printf("result = %d \n",result);
}
```

Conclusion

- That’s it … you’ve now gone beyond the common core.
- At this point, you should be able to grab some OpenMP books and a copy of the specification, and run with OpenMP on your own

OpenMP organizations

- OpenMP architecture review board URL, the "owner" of the OpenMP specification:

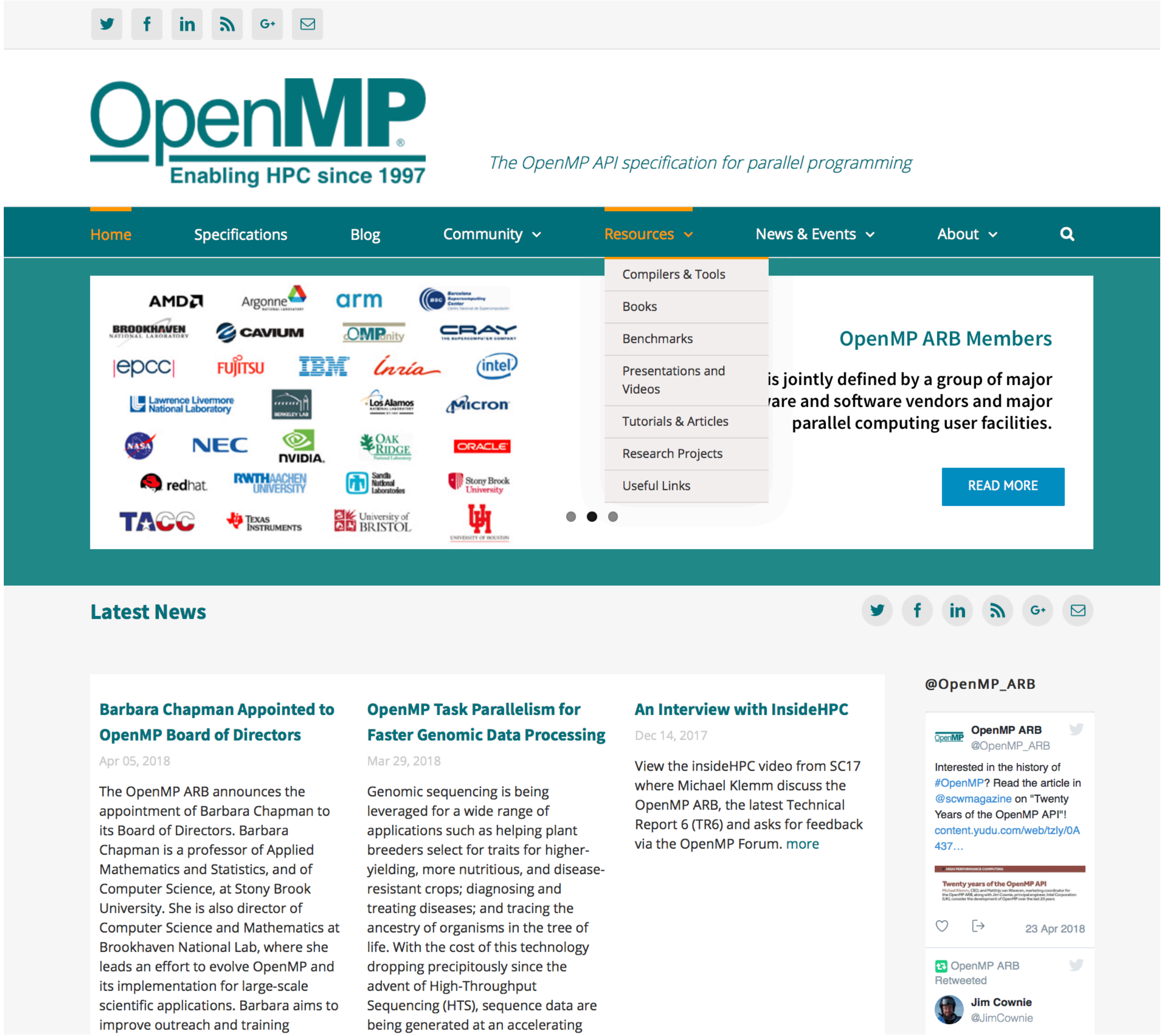
  www.openmp.org

- OpenMP User's Group (cOMPunity) URL:

  www.compunity.org

Get involved, join the ARB and cOMPunity and help define the future of OpenMP

http://www.openmp.org



The OpenMP API specification for parallel programming

OpenMP ARB Members

is jointly defined by a group of major
are and software vendors and major
parallel computing user facilities.

Latest News

Barbara Chapman Appointed to OpenMP Board of Directors

Apr 05, 2018

The OpenMP ARB announces the appointment of Barbara Chapman to its Board of Directors. Barbara Chapman is a professor of Applied Mathematics and Statistics, and of Computer Science, at Stony Brook University. She is also director of Computer Science and Mathematics at Brookhaven National Lab, where she leads an effort to evolve OpenMP and its implementation for large-scale scientific applications. Barbara aims to improve outreach and training

OpenMP Task Parallelism for Faster Genomic Data Processing

Mar 29, 2018

Genomic sequencing is being leveraged for a wide range of applications such as helping plant breeders select for traits for higher-yielding, more nutritious, and disease-resistant crops; diagnosing and treating diseases; and tracing the ancestry of organisms in the tree of life. With the cost of this technology dropping precipitously since the advent of High-Throughput Sequencing (HTS), sequence data are being generated at an accelerating

An Interview with InsideHPC

Dec 14, 2017

View the insideHPC video from SC17 where Michael Klemm discuss the OpenMP ARB, the latest Technical Report 6 (TR6) and asks for feedback via the OpenMP Forum. more

@OpenMP_ARB

OpenMP ARB
@OpenMP_ARB

Interested in the history of #OpenMP? Read the article in @scwmagazine on "Twenty Years of the OpenMP API"! content.yudu.com/web/tzly/0A437…

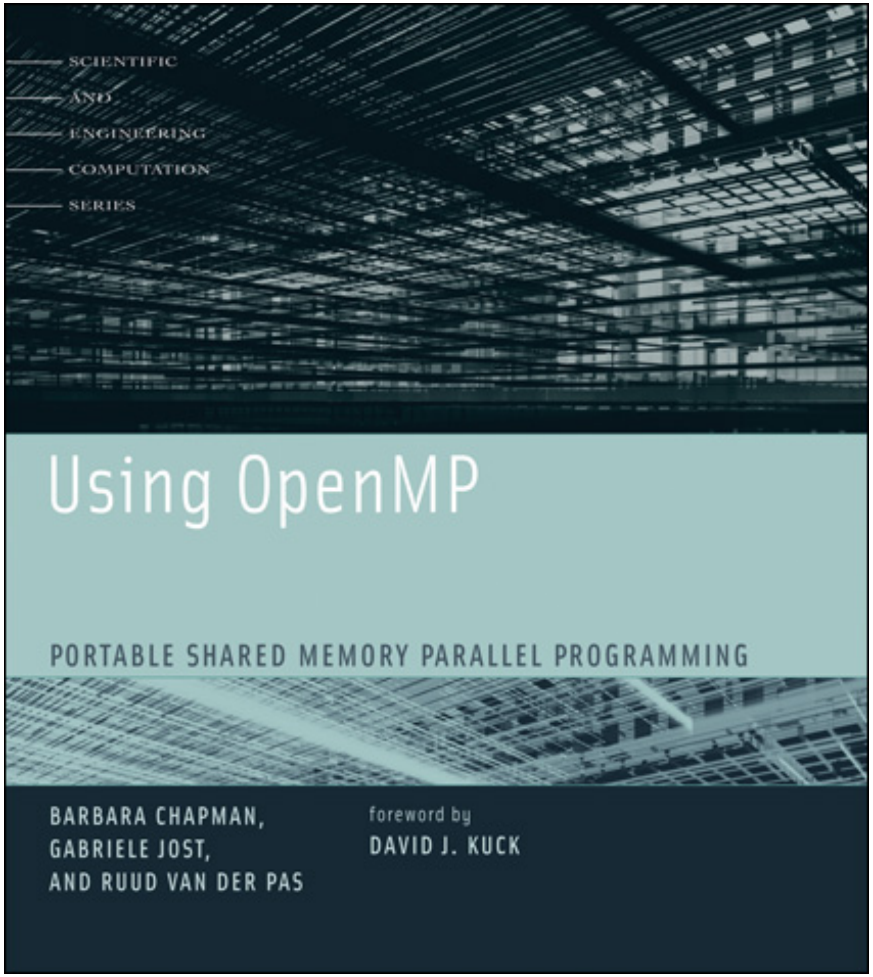
Twenty years of the OpenMP API

23 Apr 2018

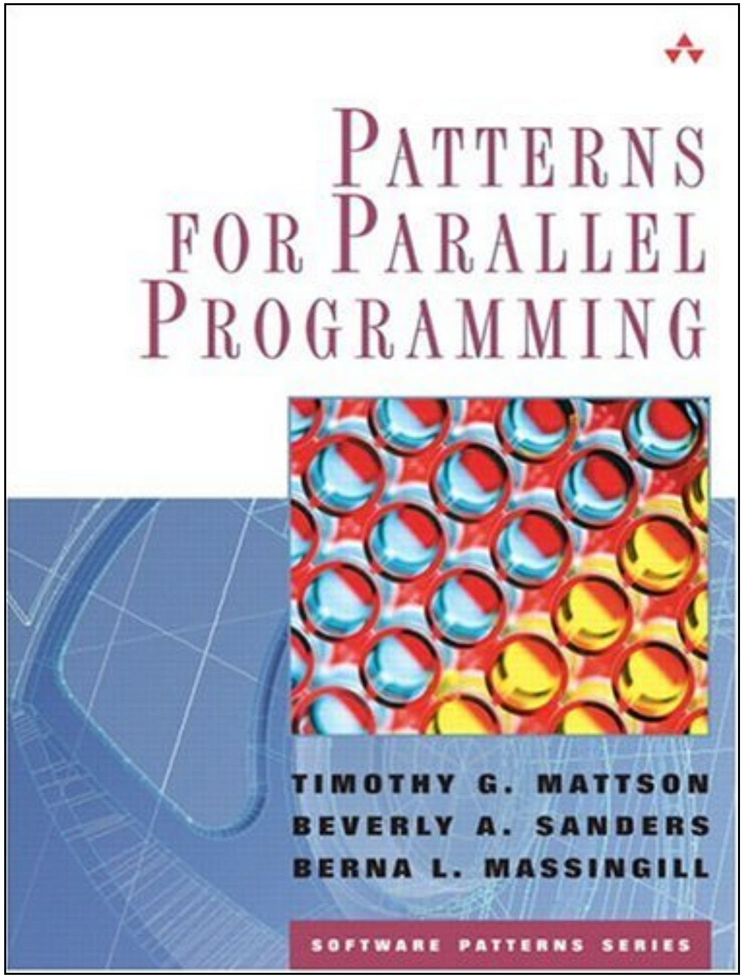
OpenMP ARB Retweeted

Jim Cownie
@JimCownie

Books about OpenMP



- A book about OpenMP by a team of authors at the forefront of OpenMP's evolution.



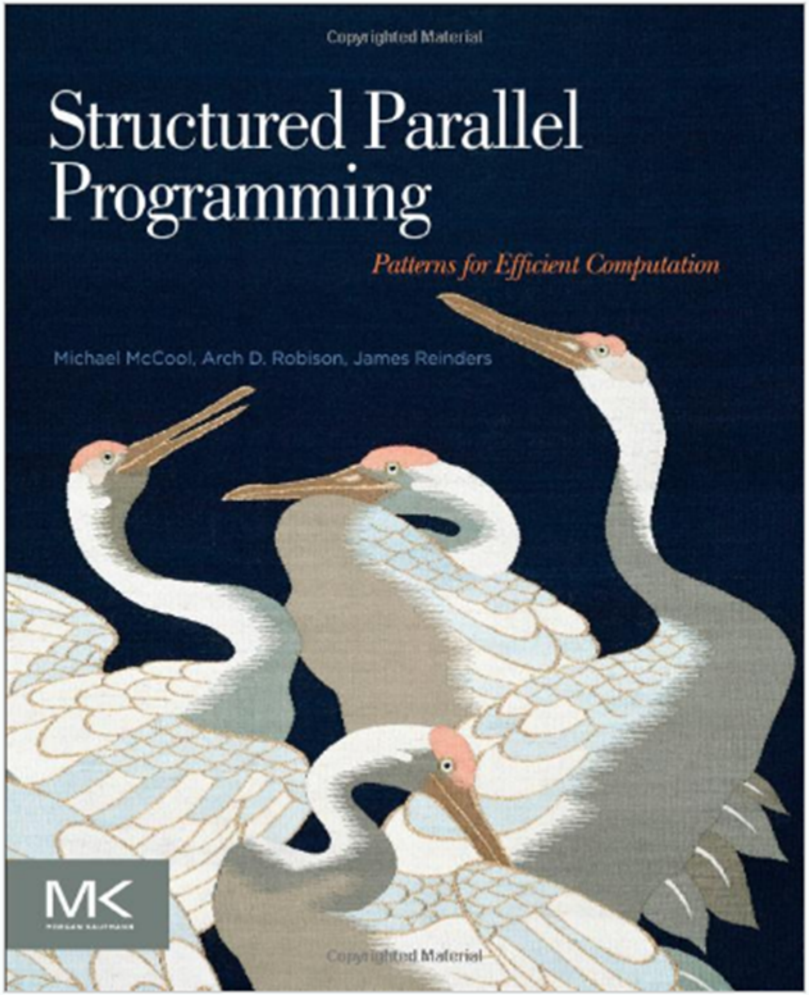
- A book about how to "think parallel" with examples in OpenMP, MPI and java

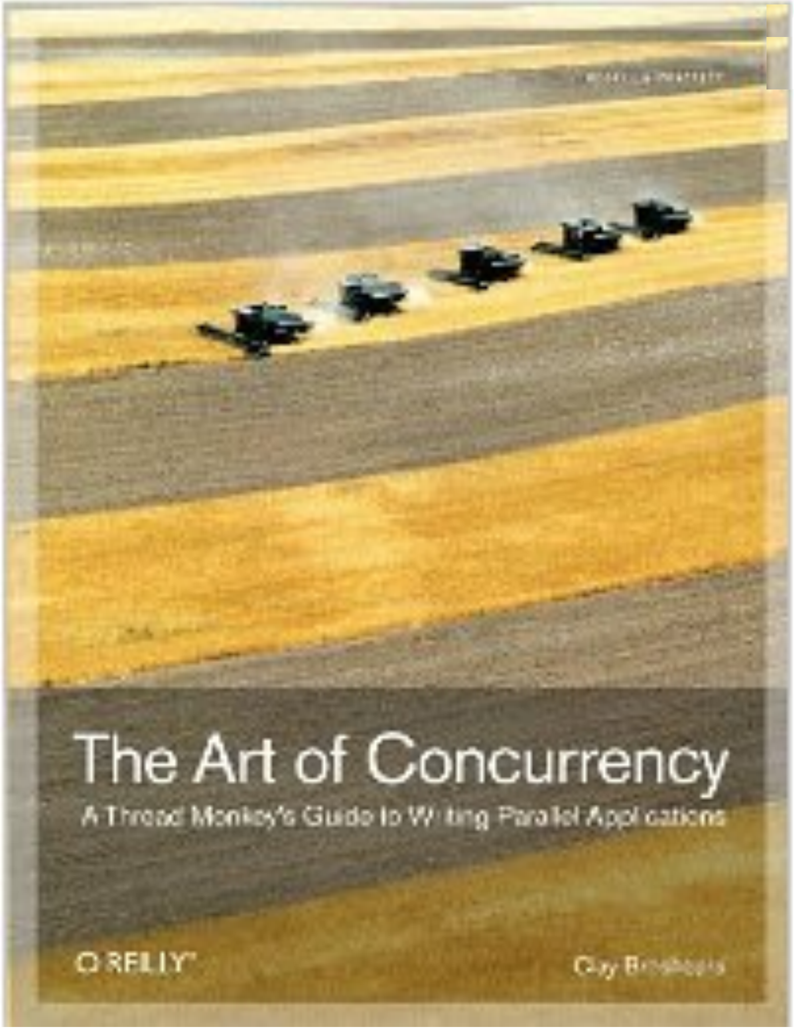
Resources:

A great new book that covers OpenMP features beyond OpenMP 2.5

SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES
USING OPENMP – THE NEXT STEP
Affinity, Accelerators, Tasking, and SIMD
Ruud van der Pas, Eric Stotzer, and Christian Terboven
USING OPENMP – THE NEXT STEP

Background references


Copyrighted Material
Structured Parallel Programming
Patterns for Efficient Computation
Michael McCool, Arch D. Robison, James Reinders
MK
Copyrighted Material



A great book that explores key patterns with Cilk, TBB, OpenCL, and OpenMP (by McCool, Robison, and Reinders)

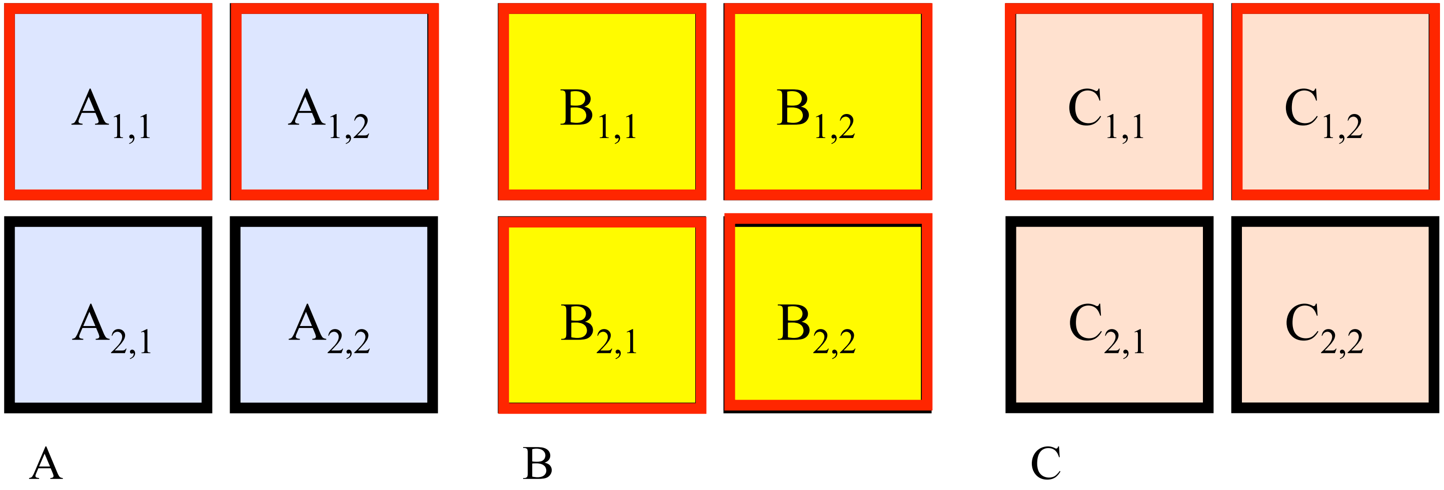
An excellent introduction and overview of multithreaded programming in general (by Clay Breshears)

Extra content

- Additional examples
  - Divide and conquer: recursive matrix multiplication
  - Task Dependencies: 1D Stencil
  - Flow Graph Analyzer and task dependencies
- Notes on Parallel random number generation

Recursive matrix multiplication

- Quarter each input matrix and output matrix
- Treat each submatrix as a single element and multiply
- 8 submatrix multiplications, 4 additions



$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$$

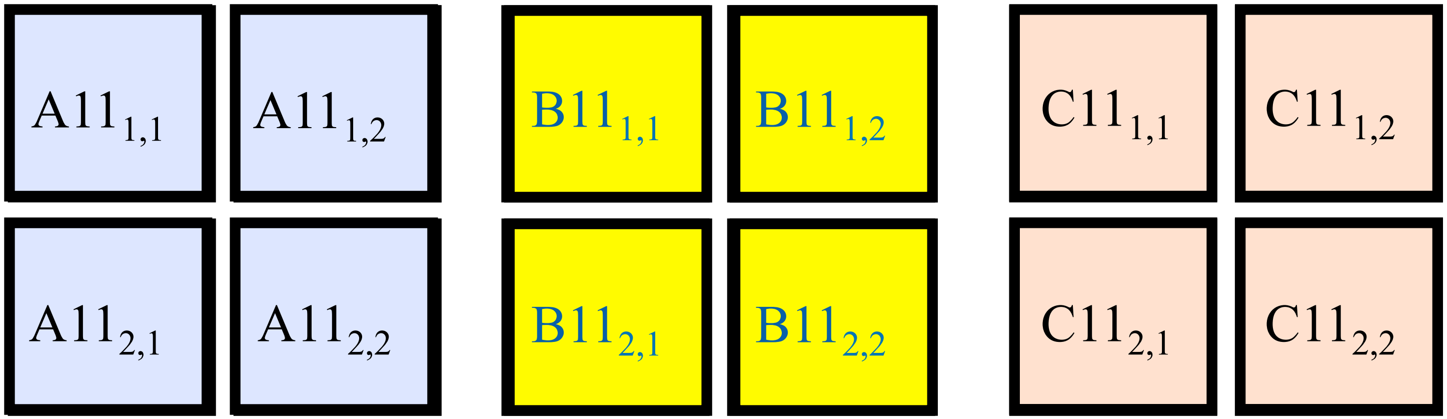
$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$$

$$C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$$

$$C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$$

Recursive matrix multiplication
How to multiply submatrices?

- Use the same routine that is computing the full matrix multiplication
  - Quarter each input submatrix and output submatrix
  - Treat each sub-submatrix as a single element and multiply



$A11_{1,1}$ $A11_{1,2}$

$A11_{2,1}$ $A11_{2,2}$

$B11_{1,1}$ $B11_{1,2}$

$B11_{2,1}$ $B11_{2,2}$

$C11_{1,1}$ $C11_{1,2}$

$C11_{2,1}$ $C11_{2,2}$

$\mathbf{A}_{1,1}$ $\mathbf{B}_{1,1}$ $C_{1,1}$

$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$$

$$C11_{1,1} = A11_{1,1} \cdot B11_{1,1} + A11_{1,2} \cdot B11_{2,1} + A12_{1,1} \cdot B21_{1,1} + A12_{1,2} \cdot B21_{2,1}$$

Recursive matrix multiplication
Recursively multiply submatrices

$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} \qquad C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$$

$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \qquad C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$$

- Need range of indices to define each submatrix to be used

```
void matmultrec(int mf, int ml, int nf, int nl, int pf, int pl,
                double **A, double **B, double **C)
{// Dimensions: A[mf..ml][pf..pl]  B[pf..pl][nf..nl]  C[mf..ml][nf..nl]

// C11 += A11*B11
 matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A,B,C);
// C11 += A12*B21
 matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A,B,C);
   . . .
}
```

- Also need stopping criteria for recursion

Recursive matrix multiplication

- Could be executed in parallel as 4 tasks
  - Each task executes the two calls for the same output submatrix of C
- However, the same number of multiplication operations needed

```
#define THRESHOLD 32768   // product size below which simple matmult code is called

void matmultrec(int mf, int ml, int nf, int nl, int pf, int pl,
                double **A, double **B, double **C)

// Dimensions: A[mf..ml][pf..pl]    B[pf..pl][nf..nl]   C[mf..ml][nf..nl]

{
   if ((ml-mf)*(nl-nf)*(pl-pf) < THRESHOLD)
      matmult (mf, ml, nf, nl, pf, pl, A, B, C);
   else
   {
#pragma omp task firstprivate(mf,ml,nf,nl,pf,pl)
{
      matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A, B, C);  // C11 += A11*B11
      matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A, B, C);  // C11 += A12*B21
}
#pragma omp task firstprivate(mf,ml,nf,nl,pf,pl)
{
      matmultrec(mf, mf+(ml-mf)/2, nf+(nl-nf)/2, nl, pf, pf+(pl-pf)/2, A, B, C);  // C12 += A11*B12
      matmultrec(mf, mf+(ml-mf)/2, nf+(nl-nf)/2, nl, pf+(pl-pf)/2, pl, A, B, C);  // C12 += A12*B22
}
#pragma omp task firstprivate(mf,ml,nf,nl,pf,pl)
{
     matmultrec(mf+(ml-mf)/2, ml, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A, B, C);  // C21 += A21*B11
     matmultrec(mf+(ml-mf)/2, ml, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A, B, C);  // C21 += A22*B21
}
#pragma omp task firstprivate(mf,ml,nf,nl,pf,pl)
{
     matmultrec(mf+(ml-mf)/2, ml, nf+(nl-nf)/2, nl, pf, pf+(pl-pf)/2, A, B, C);  // C22 += A21*B12
     matmultrec(mf+(ml-mf)/2, ml, nf+(nl-nf)/2, nl, pf+(pl-pf)/2, pl, A, B, C);  // C22 += A22*B22
}
#pragma omp taskwait

   }
}
```

Extra content

- Additional examples
  - Divide and conquer: recursive matrix multiplication
  - Task Dependencies: 1D Stencil
  - Flow Graph Analyzer and task dependencies
- Notes on Parallel random number generation

1D Stencil Example

The heat equation:

```
double k = 0.5; // heat transfer coefficient
double dt = 1.; // time step
double dx = 1.; // grid spacing

double heat(double left, double mid, double right)
{
    return mid+(k*dt/dx*dx)*(left-2*mid+right);
}
```

1D Stencil Example

Application of the heat equation to a 1D array

```
void heat_part( int size, double* next,
                double* left,
                double *mid, double *right)
{
    next[0] = heat(left[size-1], mid[0], mid[1]);

    for (int i = 1; i < size-1; ++i)
      next[i] = heat(mid[i-1], mid[i], mid[i+1]);

      next[size-1] = heat(mid[size-2], mid[size-1],
                        right[0]);
}
```

1D Stencil Example

Dividing the work into partitions of the array

```
for (int i = 0; i < np; ++i) {
    heat_part( nx, &next[i*nx],
                   &current[idx(i-1, np)*nx],
                   &current[i*nx],
                   &current[idx(i+1, np)*nx]);
}

//idx does the wrapping here
int idx(int i, int size)
{
    return (i < 0) ? (i + size) % size : i % size;
}
```

1D Stencil Example

Reads and writes need to be done on separate arrays

```
U[0] = malloc(np*nx * sizeof(double));
U[1] = malloc(np*nx * sizeof(double));

double* current = U[0];
double* next    = U[1];
```

1D Stencil Example

Each iteration alternates between arrays

```
for(int t = 0; t < nt; t++) {
    for (int i = 0; i < np; ++i) {
        heat_part( nx, &next[i*nx],
                   &current[idx(i-1, np)*nx],
                   &current[i*nx],
                   &current[idx(i+1, np)*nx]);
    }
    current = U[(t+1) % 2];
    next    = U[ t    % 2];
}
```

1D Stencil Example

Because of the partitioning, one task directive is needed

```
for(int t = 0; t < nt; t++) {
    for (int i = 0; i < np; ++i) {
#pragma omp task depend(out: next[i*nx]) \
            depend(in: current[idx(i-1, np)*nx],\
            current[i*nx], current[idx(i+1, np)*nx])
        heat_part( nx, &next[i*nx],
                   &current[idx(i-1, np)*nx],
                   &current[i*nx],
                   &current[idx(i+1, np)*nx]);
    }
    current = U[(t+1) % 2];
    next    = U[ t    % 2];
}
#pragma omp taskwait
```

Extra content

- Additional examples
  - Divide and conquer: recursive matrix multiplication
  - Task Dependencies: 1D Stencil
  - Flow Graph Analyzer and task dependencies
- Notes on Parallel random number generation

OMPT and Flow Graph Analyzer: Visualization and Analysis of OpenMP Task Dependencies

Vishakha Agrawal
Intel Corp

May 2 2018

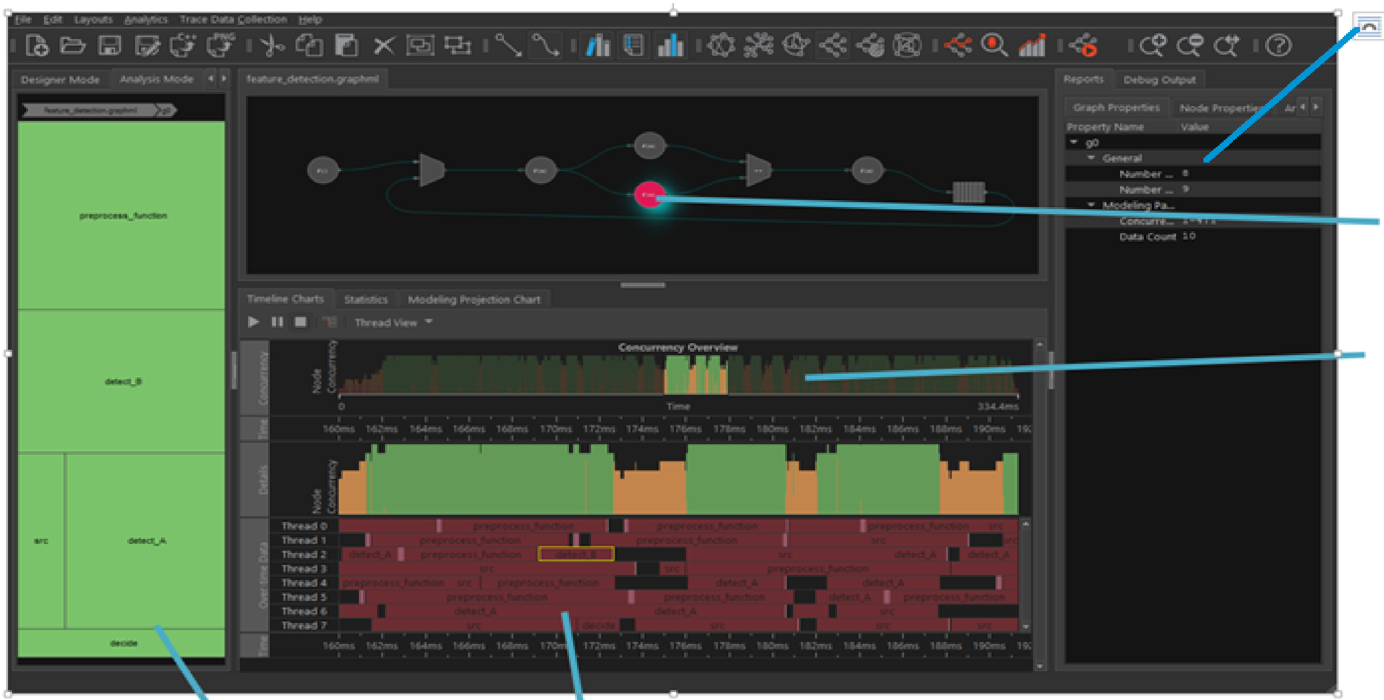
AGENDA

- Introduction to Flow Graph Analyzer
- Original code showing OpenMP inner loop
- FGA screenshot where you can see additional dependency
- New code with perf fix
- Chart/graph showing performance gain.

Flow Graph Analyzer (FGA) :

Released as Tech. Preview in Intel Parallel Studio for TBB

- A visualization tool that supports the analysis and design of parallel applications that use computational graphs



It provides details about graph properties, node properties and useful debuging info for tool.

Selection on the timeline highlights nodes that were executed at that point in time.

The concurrency histogram shows the parallelism achieved by the graph over time. You can interact with this chart by zooming in to a region of time, for ex. during the low concurrency.

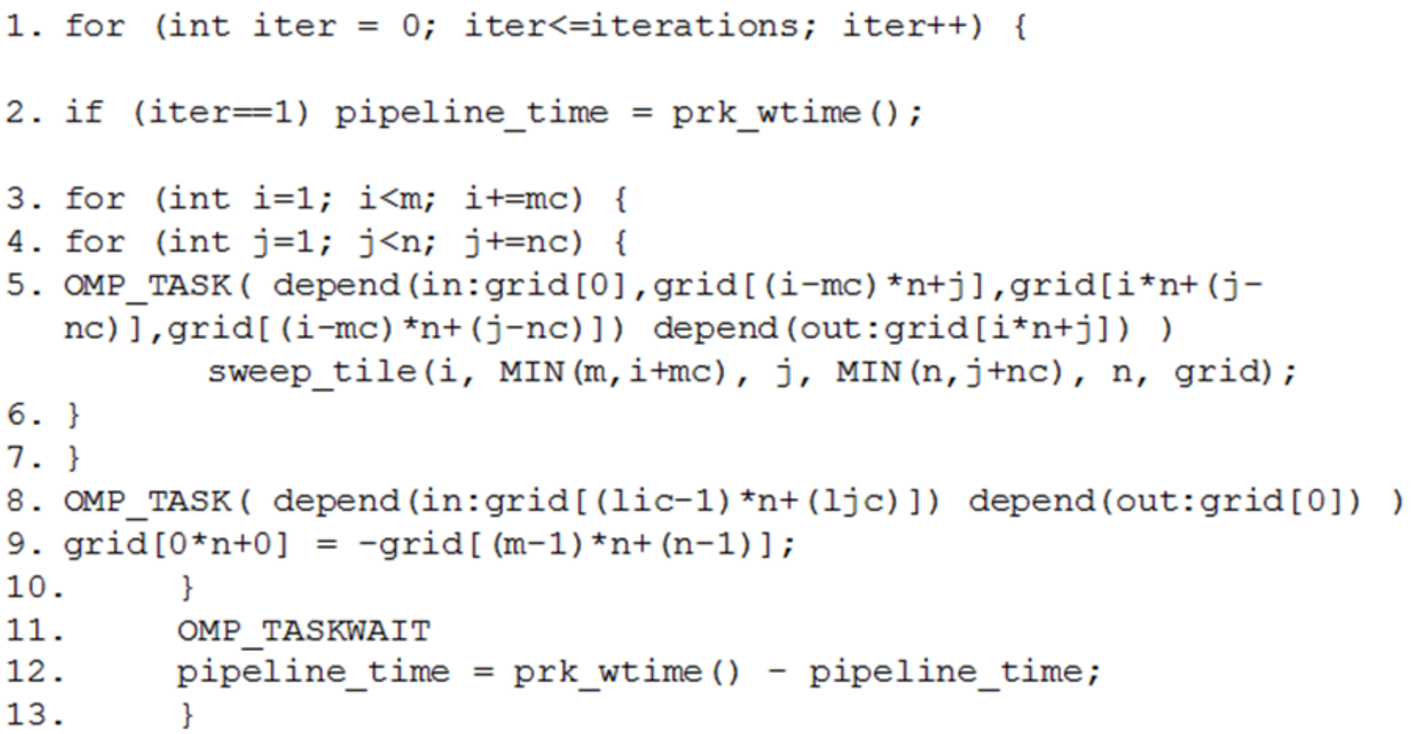
The concurrency histogram remains at the intial zoom level, and the zoomed in region is displayed below it.

Treemap view gives you the general health of the graph's performance along with the ability to dive to the node level.

The per-thread task view shows the tasks executed by each thread along with the task durations.

A Proof of Concept system for OpenMP tasks with depend clauses

Original code showing OpenMP inner loop



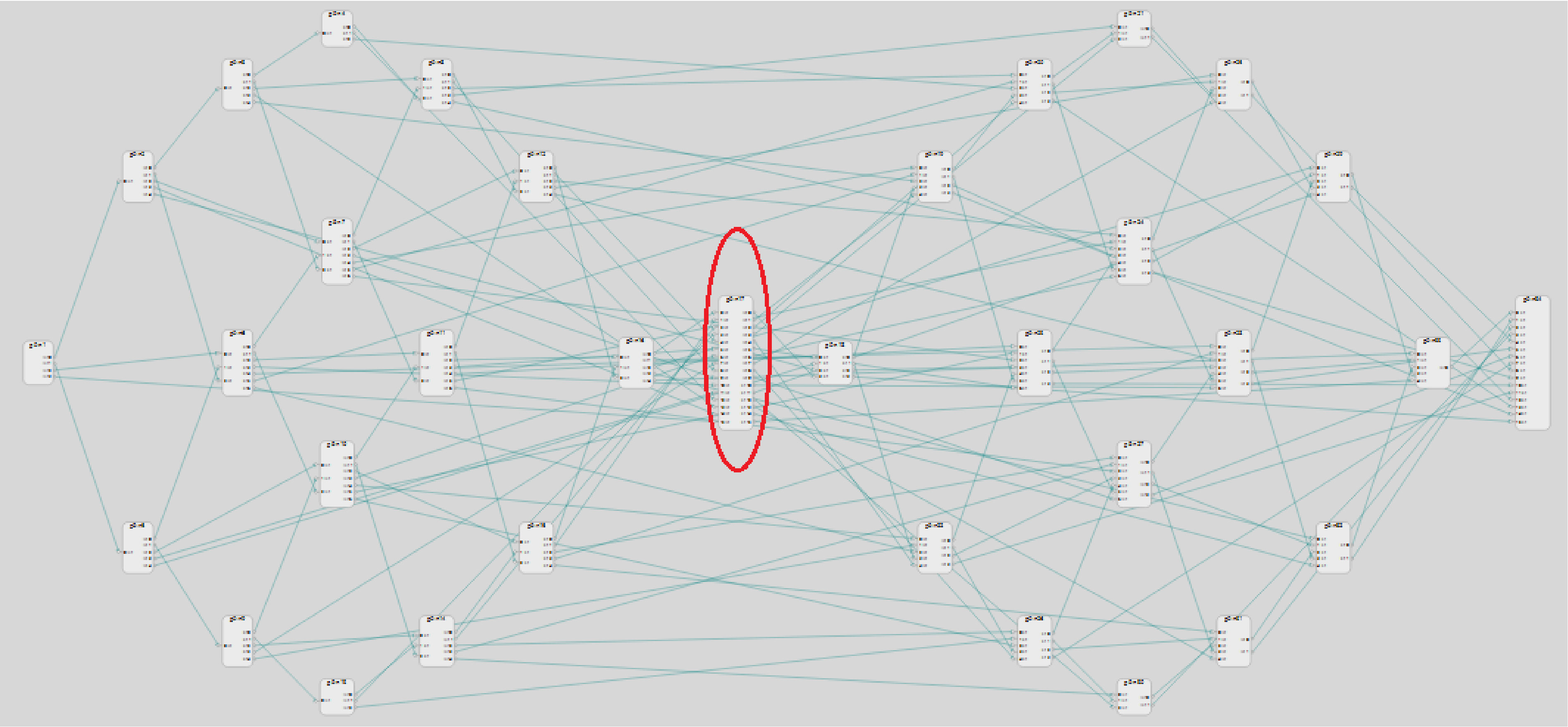
```
for (int iter = 0; iter<=iterations; iter++) {

if (iter==1) pipeline_time = prk_wtime();

for (int i=1; i<m; i+=mc) {
for (int j=1; j<n; j+=nc) {
OMP_TASK( depend(in:grid[0],grid[(i-mc)*n+j],grid[i*n+(j-
nc)],grid[(i-mc)*n+(j-nc)]) depend(out:grid[i*n+j]) )
     sweep_tile(i, MIN(m,i+mc), j, MIN(n,j+nc), n, grid);
}
}
OMP_TASK( depend(in:grid[(lic-1)*n+(ljc)]) depend(out:grid[0]) )
grid[0*n+0] = -grid[(m-1)*n+(n-1)];
    }
    OMP_TASKWAIT
    pipeline_time = prk_wtime() - pipeline_time;
    }
```

- There are two performance improvement opportunities:

1. Between each iteration there is a node that uses depend to create a barrier
2. There are depend items that can be removed due to transitivity

The dependence graph displayed in FGA

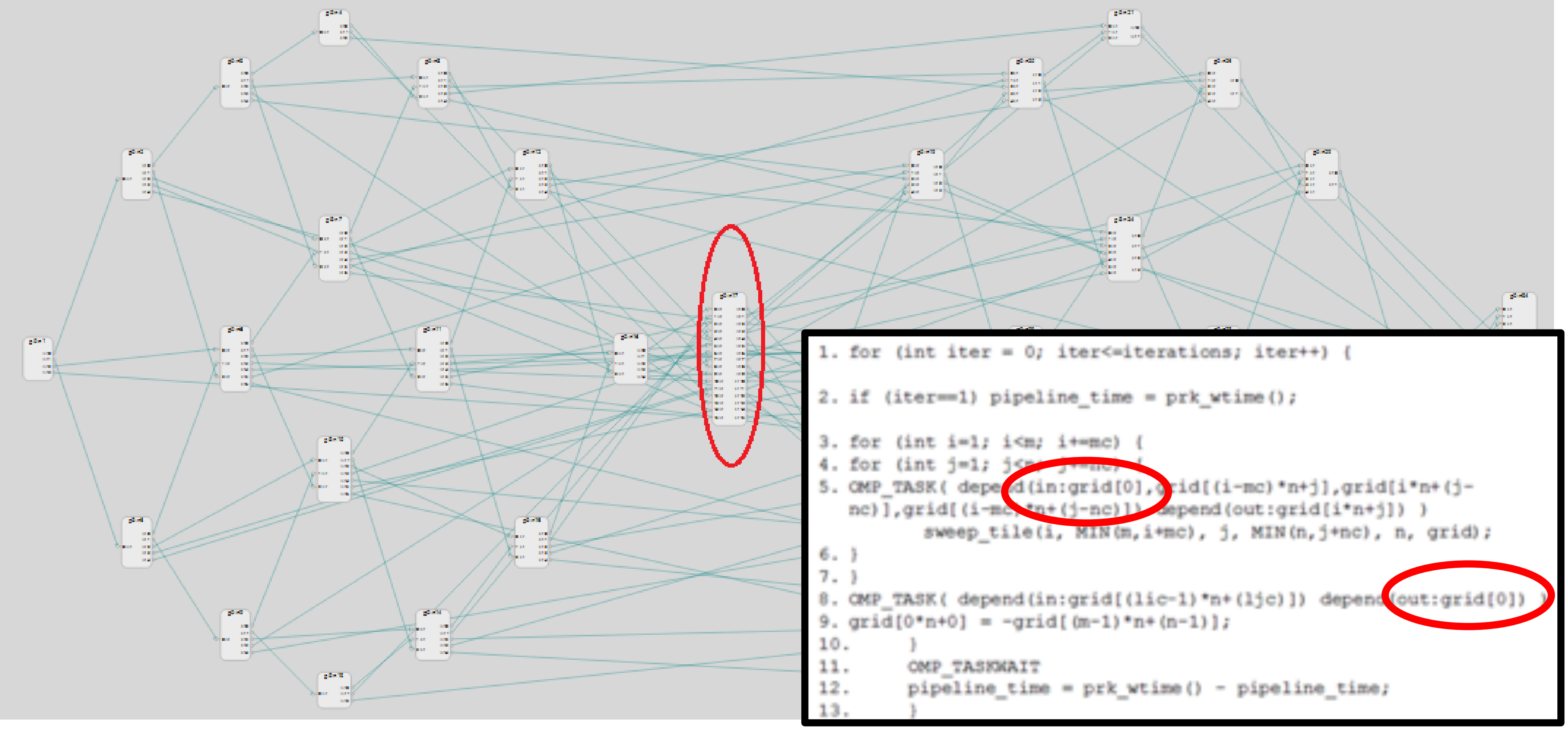
Optimization 1: the barrier node



Note; All tasks pass through this one choke point …. Basically meaning we are using depend clauses to create the equivalent of a taskwait

The dependence graph displayed in FGA

Optimization 1: the barrier node

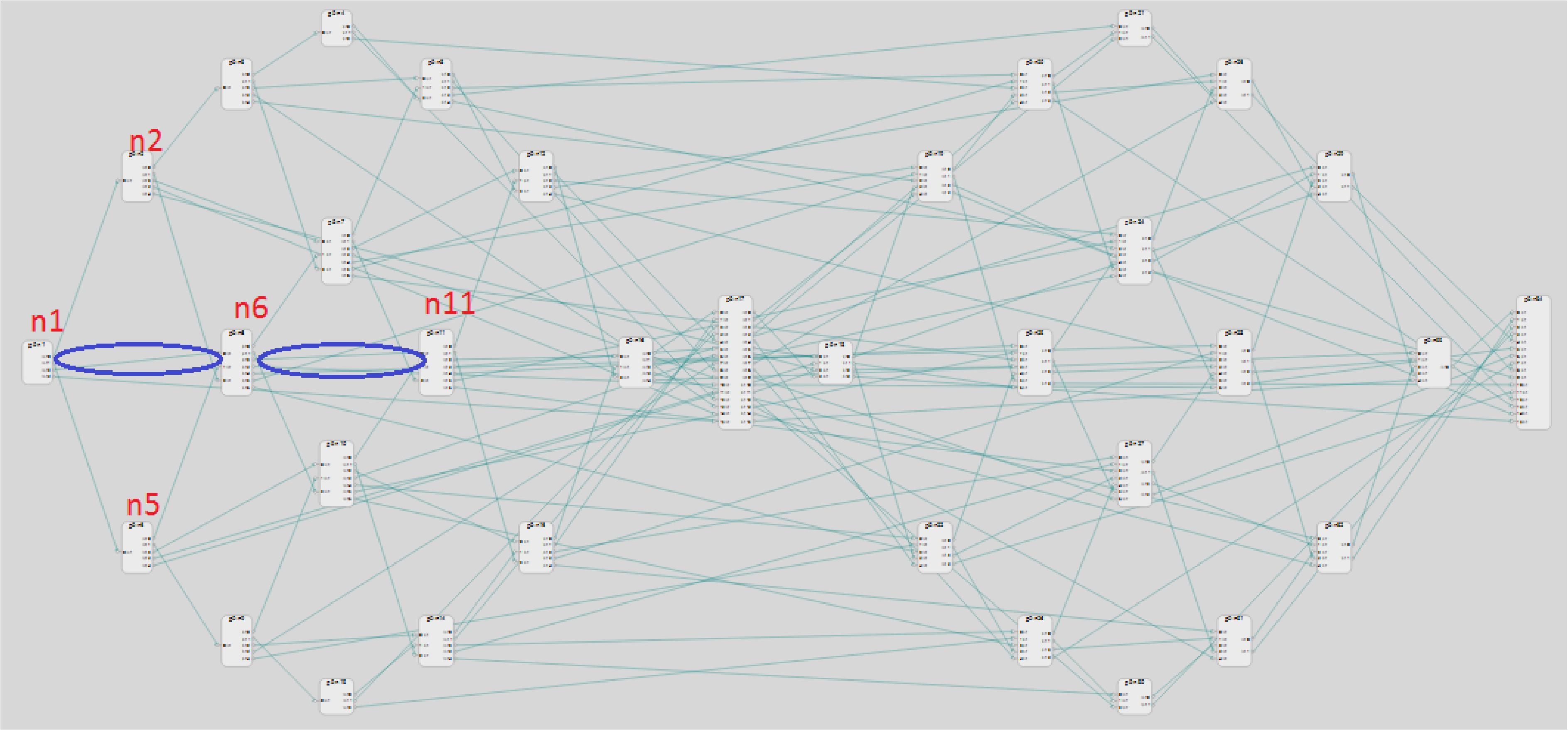


```
for (int iter = 0; iter<=iterations; iter++) {

if (iter==1) pipeline_time = prk_wtime();

for (int i=1; i<m; i+=mc) {
for (int j=1; j<n; j+=nc) {
OMP_TASK( depend(in:grid[0],grid[(i-mc)*n+j],grid[i*n+(j-
nc)],grid[(i-mc)*n+(j-nc)]) depend(out:grid[i*n+j]) )
          sweep_tile(i, MIN(m,i+mc), j, MIN(n,j+nc), n, grid);
}
}
OMP_TASK( depend(in:grid[(lic-1)*n+(ljc)]) depend(out:grid[0]) )
grid[0*n+0] = -grid[(m-1)*n+(n-1)];
        }
        OMP_TASKWAIT
        pipeline_time = prk_wtime() - pipeline_time;
        }
```

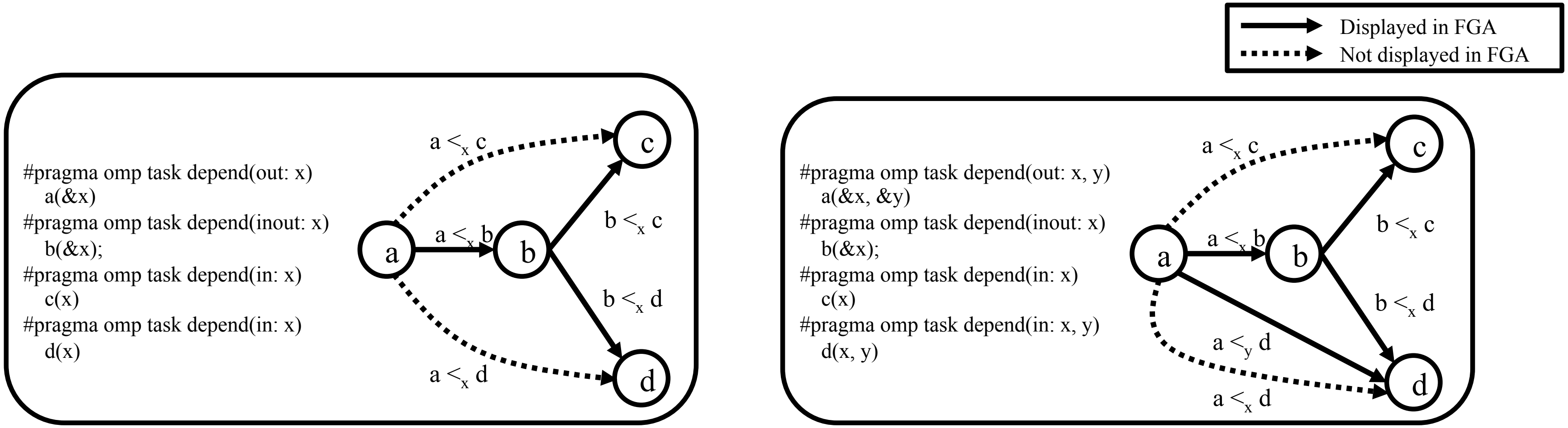
Note; All tasks pass through this one choke point …. Basically meaning we are using depend clauses to create the equivalent of a taskwait

The dependence graph displayed in FGA

Optimization 2: removal of transitive dependences



Transitive OpenMP Dependences in FGA

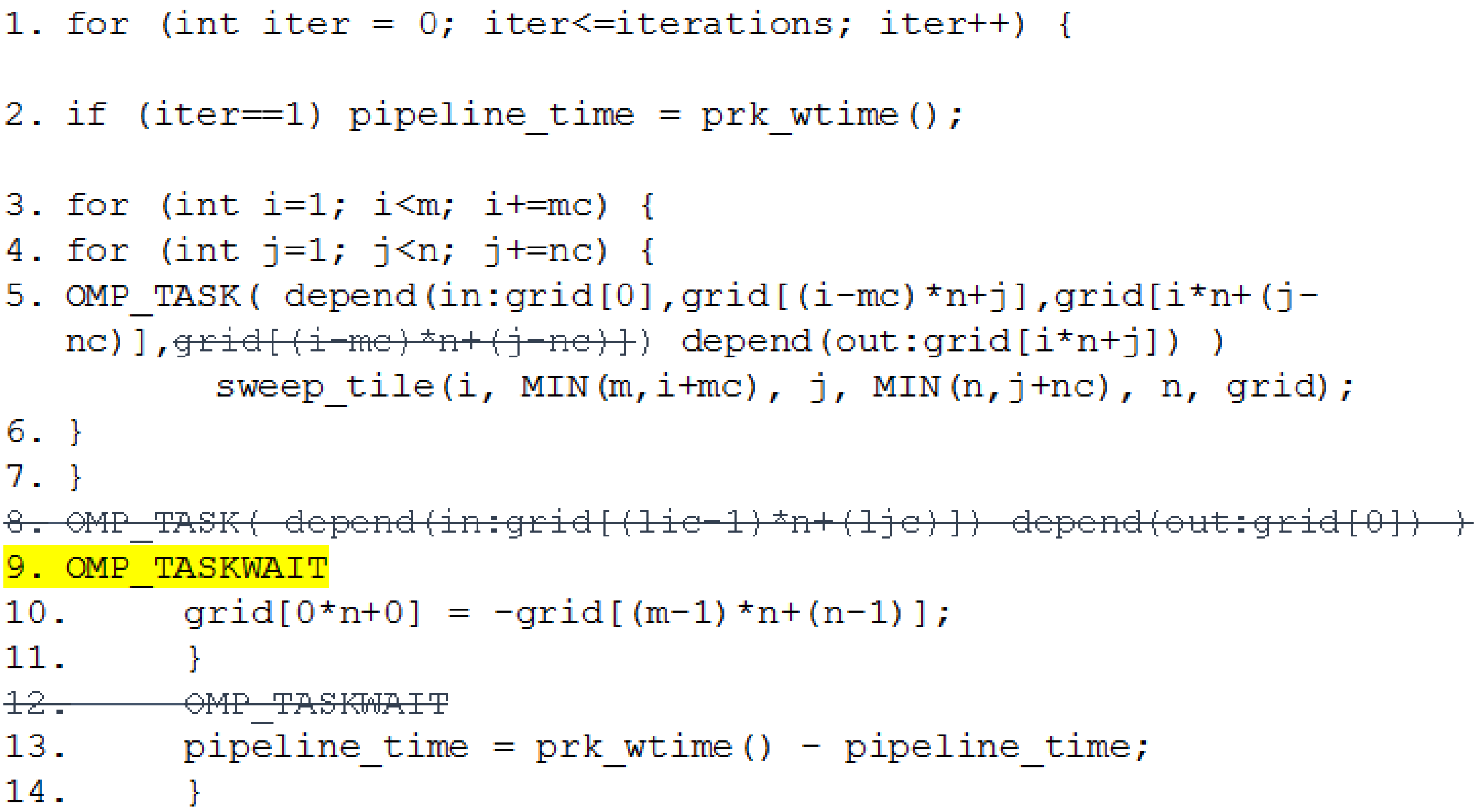


```
#pragma omp task depend(out: x)
   a(&x)
#pragma omp task depend(inout: x)
   b(&x);
#pragma omp task depend(in: x)
   c(x)
#pragma omp task depend(in: x)
   d(x)
```

```
#pragma omp task depend(out: x, y)
   a(&x, &y)
#pragma omp task depend(inout: x)
   b(&x);
#pragma omp task depend(in: x)
   c(x)
#pragma omp task depend(in: x, y)
   d(x, y)
```

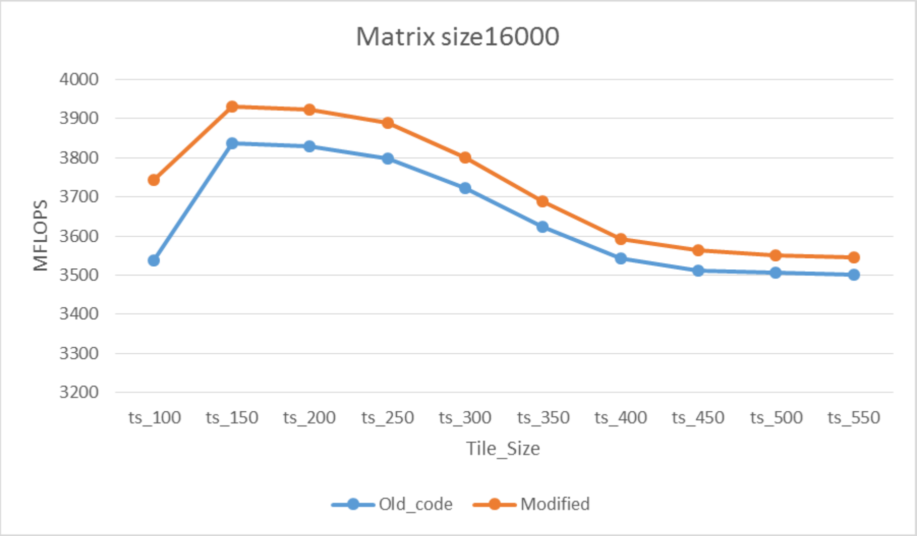
- $a <_x b$ means a must execute before b due to a dependence on location x
- As shown in the left figure, FGA does NOT display $a <_x d$ since $a <_x b$ and $b <_x d$
- But in the right figure, there is no transitive dependence due to y, so FGA does show $a <_y d$
- However, the ordering is enforced due to the x dependence, and therefore it is legal to remove y from depend clause for d in the source code to reduce book-keeping
  - Whether this is good idea or not (maintainability or readability) is left to developers

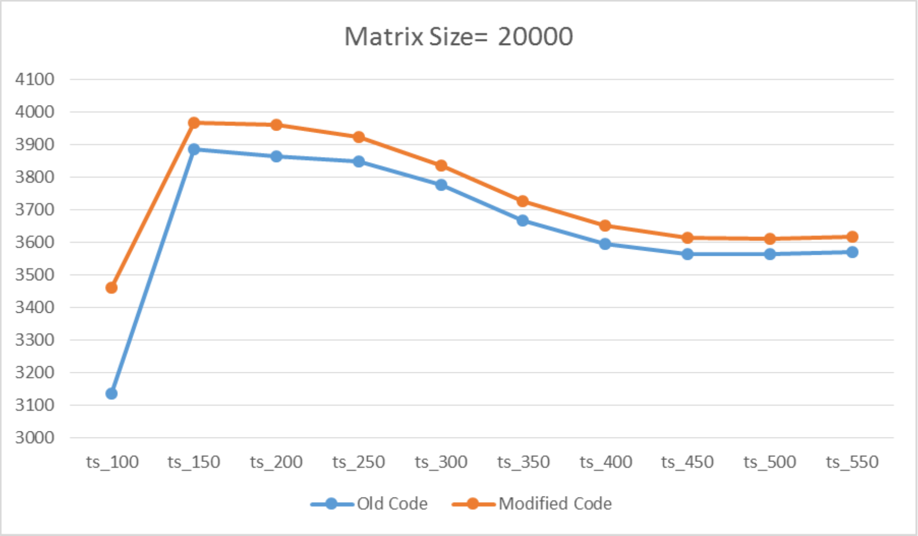
Modified code with both optimizations:



```
for (int iter = 0; iter<=iterations; iter++) {

if (iter==1) pipeline_time = prk_wtime();

for (int i=1; i<m; i+=mc) {
for (int j=1; j<n; j+=nc) {
OMP_TASK( depend(in:grid[0],grid[(i-mc)*n+j],grid[i*n+(j-
nc)],~~grid[(i-mc)*n+(j-nc)]~~) depend(out:grid[i*n+j]) )
         sweep_tile(i, MIN(m,i+mc), j, MIN(n,j+nc), n, grid);
}
}
~~OMP_TASK( depend(in:grid[(lic-1)*n+(ljc)]) depend(out:grid[0]) )~~
OMP_TASKWAIT
        grid[0*n+0] = -grid[(m-1)*n+(n-1)];
        }
        ~~OMP_TASKWAIT~~
        pipeline_time = prk_wtime() - pipeline_time;
        }
```

Performance Charts:





There is roughly a constant 100 MFLOPS improvement

Configuration Info:
Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz
Compiler Intel 18.0, Arch: intel64
KMP_HW_SUBSET=1S,18C,1T
KMP_AFFINITY=granularity=fine,compact

Legal Disclaimer & Optimization Notice

- The benchmark results reported above may need to be revised as additional testing is conducted. The results depend on the specific platform configurations and workloads utilized in the testing, and may not be applicable to any particular user's components, computer system or workloads. The results are not necessarily representative of other benchmarks and other benchmark results may show greater or lesser impact from mitigations.
- 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.
- Copyright © 2018, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

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

Extra content

- Additional examples
  - Divide and conquer: recursive matrix multiplication
  - Task Dependencies: 1D Stencil
  - Flow Graph Analyzer and task dependencies
- Notes on Parallel random number generation

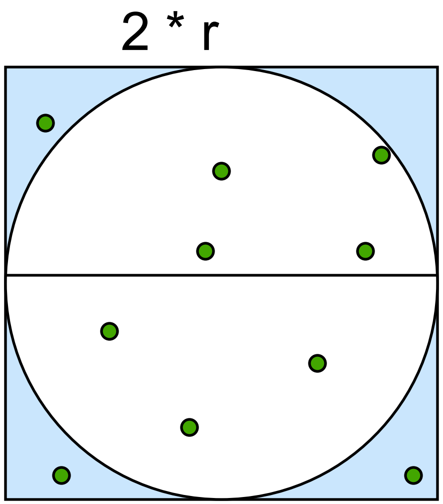
Computers and random numbers

- We use "dice" to make random numbers:
  - Given previous values, you cannot predict the next value.
  - There are no patterns in the series … and it goes on forever.
- Computers are deterministic machines … set an initial state, run a sequence of predefined instructions, and you get a deterministic answer
  - By design, computers are not random and cannot produce random numbers.
- However, with some very clever programming, we can make "pseudo random" numbers that are as random as you need them to be … but only if you are very careful.
- Why do I care? Random numbers drive statistical methods used in countless applications:
  - Sample a large space of alternatives to find statistically good answers (Monte Carlo methods).

Monte Carlo Calculations

Using Random numbers to solve tough problems

- Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
- Example: Computing π with a digital dart board:



N= 10	π = 2.8
N=100	π = 3.16
N= 1000	π = 3.148

- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:

  $A_c = r^2 * \pi$

  $A_s = (2*r) * (2*r) = 4 * r^2$

  $P = A_c/A_s = \pi /4$
- Compute π by randomly choosing points, count the fraction that falls in the circle, compute pi.

Parallel Programmers love Monte Carlo algorithms

Embarrassingly parallel: the parallelism is so easy its embarrassing.

Add two lines and you have a parallel program.

```
#include “omp.h”
static long num_trials = 10000;
int main ()
{
  long i;     long Ncirc = 0;      double pi, x, y;
  double r = 1.0;   // radius of circle. Side of squrare is 2*r
  seed(0,-r, r);  // The circle and square are centered at the origin
  #pragma omp parallel for private (x, y) reduction (+:Ncirc)
  for(i=0;i<num_trials; i++)
  {
    x = random();        y = random();
    if ( x*x + y*y) <= r*r)   Ncirc++;
   }

   pi = 4.0 * ((double)Ncirc/(double)num_trials);
   printf("\n %d trials, pi is %f \n",num_trials, pi);
}
```

Linear Congruential Generator (LCG)

- LCG: Easy to write, cheap to compute, portable, OK quality

```
random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
random_last = random_next;
```

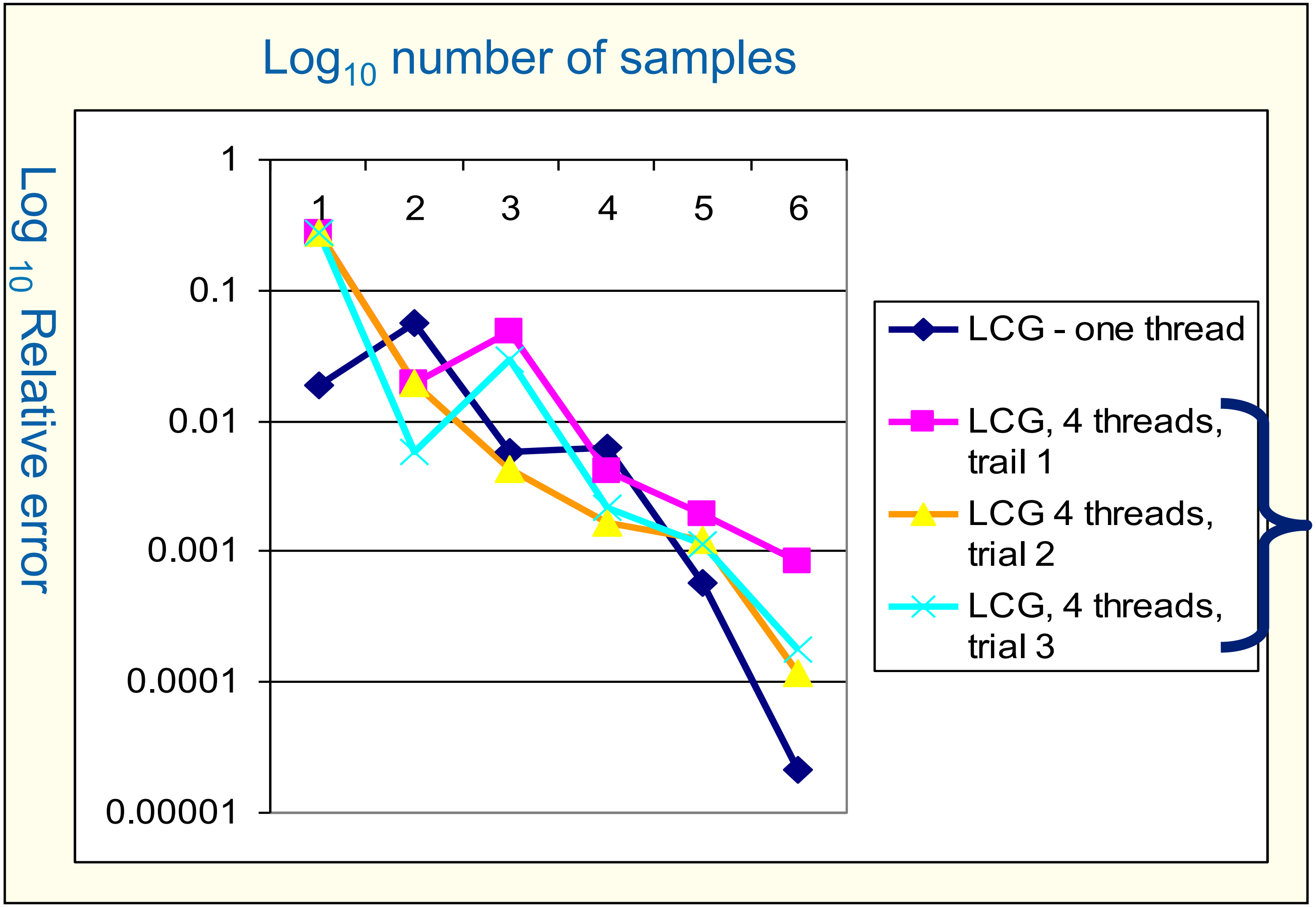
- If you pick the multiplier and addend correctly, LCG has a period of PMOD.
- Picking good LCG parameters is complicated, so look it up (Numerical Recipes is a good source). I used the following:
    - MULTIPLIER = 1366
    - ADDEND = 150889
    - PMOD = 714025

LCG code

```
static long MULTIPLIER  = 1366;
static long ADDEND      = 150889;
static long PMOD        = 714025;
long random_last = 0;
double random ()
{
    long random_next;

    random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
    random_last = random_next;

   return  ((double)random_next/(double)PMOD);
}
```

Seed the pseudo random sequence by setting random_last

Running the PI_MC program with LCG generator



Run the same program the same way and get different answers!

That is not acceptable!

Issue: my LCG generator is not threadsafe

Program written using the Intel C/C++ compiler (10.0.659.2005) in Microsoft Visual studio 2005 (8.0.50727.42) and running on a dual-core laptop (Intel T2400 @ 1.83 Ghz with 2 GB RAM) running Microsoft Windows XP.

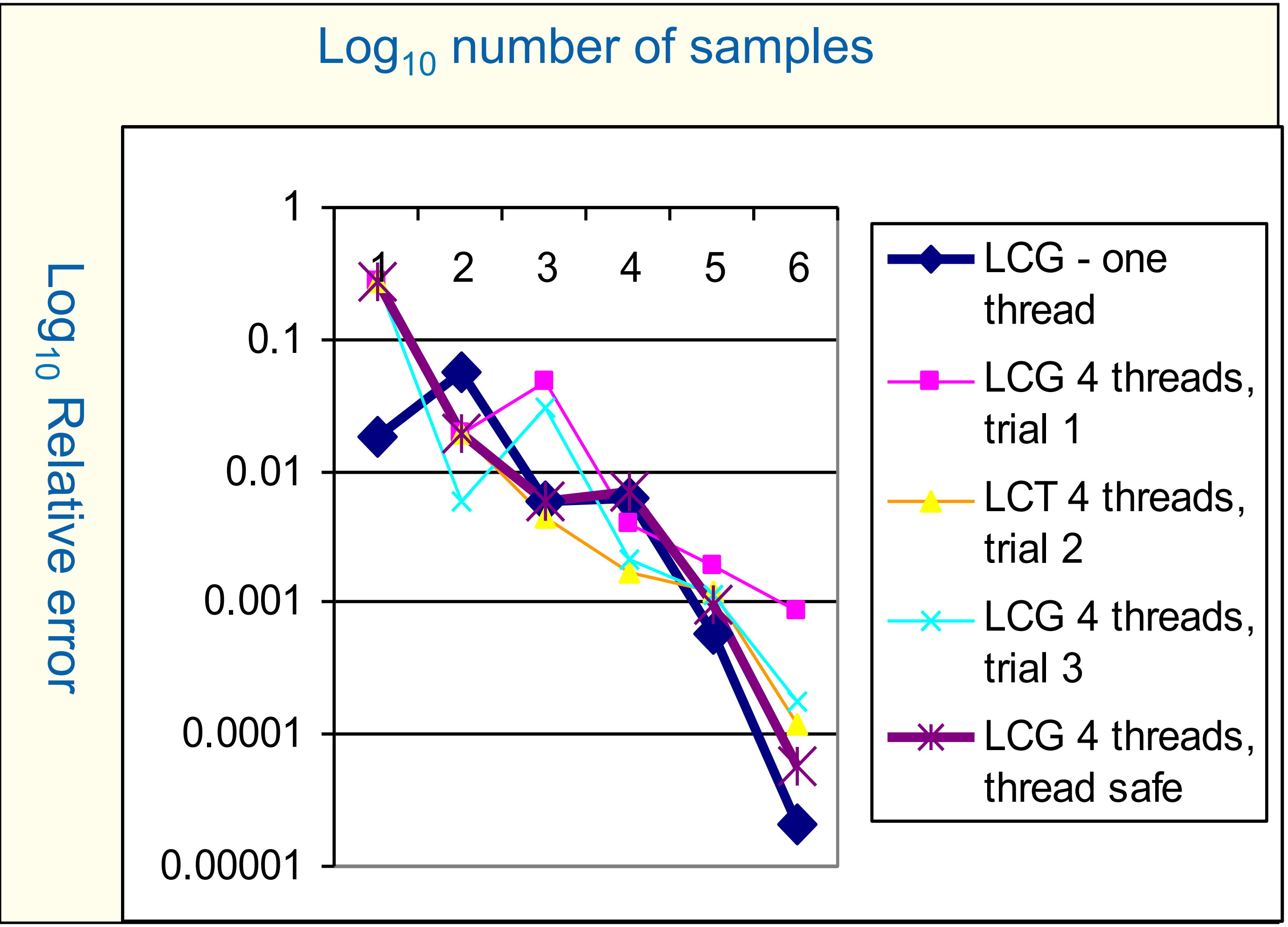
LCG code: threadsafe version

```
static long MULTIPLIER  = 1366;
static long ADDEND      = 150889;
static long PMOD        = 714025;
long random_last = 0;
#pragma omp threadprivate(random_last)
double random ()
{
    long random_next;

    random_next = (MULTIPLIER  * random_last + AD
    random_last = random_next;

   return  ((double)random_next/(double)PMOD);
}
```

random_last carries state between random number computations,

To make the generator threadsafe, make random_last threadprivate so each thread has its own copy.

Thread safe random number generators



Thread safe version gives the same answer each time you run the program.

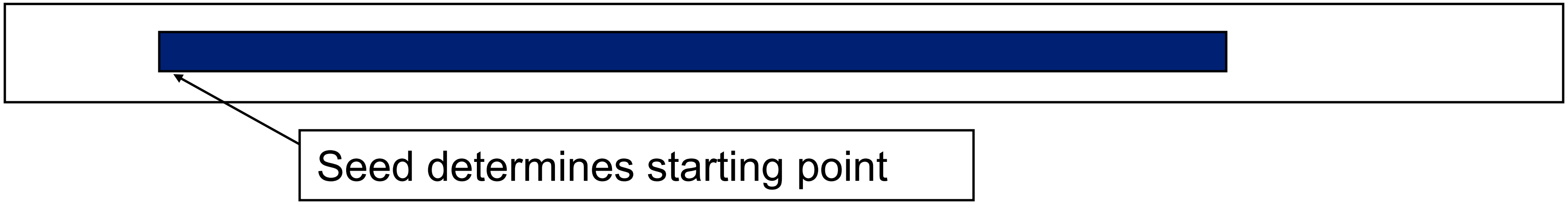
But for large number of samples, its quality is lower than the one thread result!

Why?

Pseudo Random Sequences

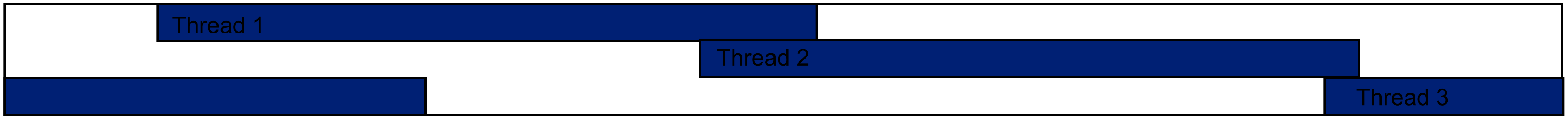
- Random number Generators (RNGs) define a sequence of pseudo-random numbers of length equal to the period of the RNG

- In a typical problem, you grab a subsequence of the RNG range



Seed determines starting point

- Grab arbitrary seeds and you may generate overlapping sequences
  - E.g. three sequences … last one wraps at the end of the RNG period.



- Overlapping sequences = over-sampling and bad statistics … lower quality or even wrong answers!

Parallel random number generators

- Multiple threads cooperate to generate and use random numbers.
- Solutions:
  - Replicate and Pray
  - Give each thread a separate, independent generator
  - Have one thread generate all the numbers.
  - Leapfrog … deal out sequence values “round robin” as if dealing a deck of cards.
  - Block method … pick your seed so each threads gets a distinct contiguous block.
- Other than “replicate and pray”, these are difficult to implement. Be smart … buy a math library that does it right.

If done right, can generate the same sequence regardless of the number of threads …

Nice for debugging, but not really needed scientifically.

Intel’s Math kernel Library supports all of these methods.

MKL Random number generators (RNG)

- MKL includes several families of RNGs in its vector statistics library.
- Specialized to efficiently generate vectors of random numbers

```
#define BLOCK 100
double  buff[BLOCK];
VSLStreamStatePtr stream;

vslNewStream(&ran_stream, VSL_BRNG_WH, (int)seed_val);

vdRngUniform (VSL_METHOD_DUNIFORM_STD, stream,
                    BLOCK, buff, low, hi)

vslDeleteStream( &stream );
```

Initialize a stream or pseudo random numbers

Select type of RNG and set seed

Fill buff with BLOCK pseudo rand. nums, uniformly distributed with values between lo and hi.

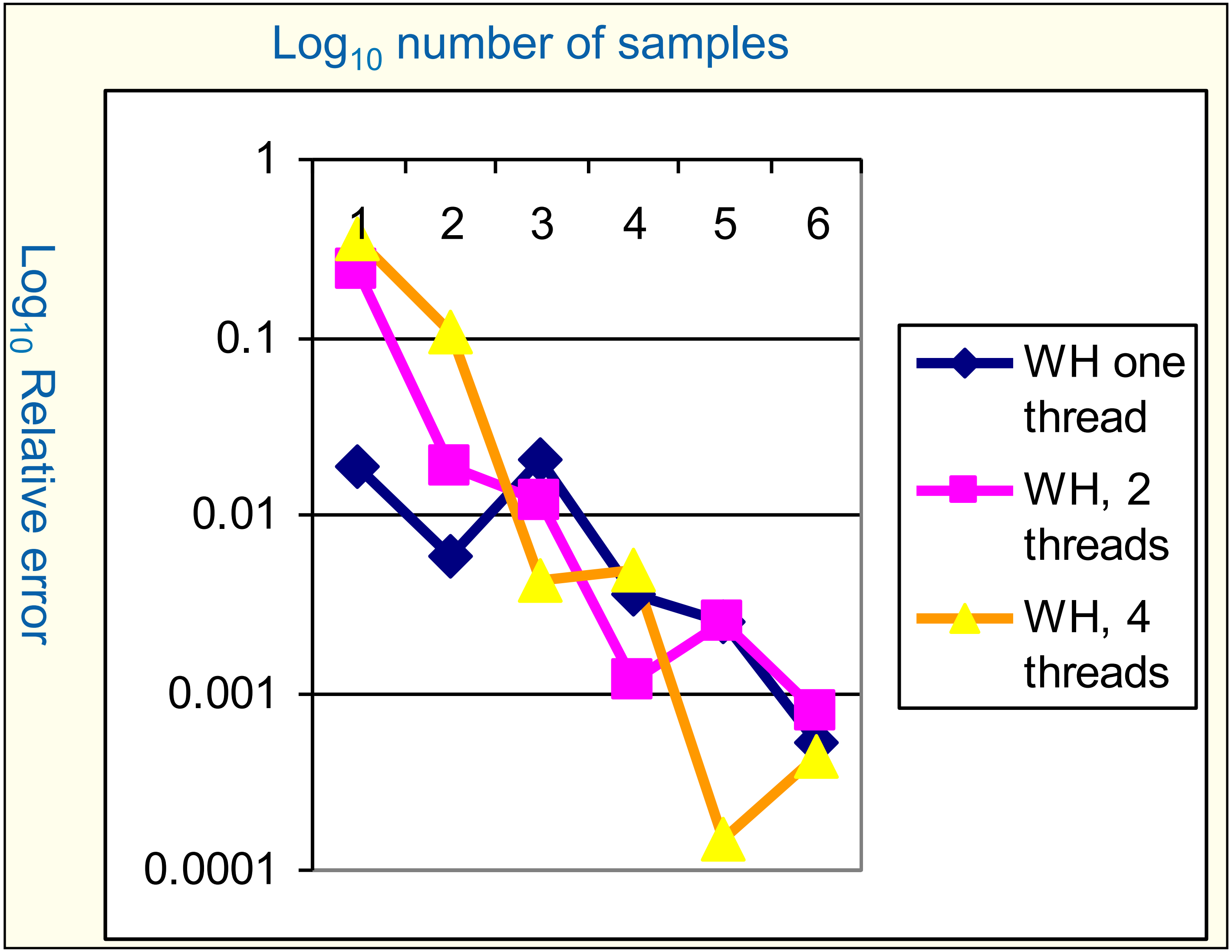
Delete the stream when you are done

Wichmann-Hill generators (WH)

- WH is a family of 273 parameter sets each defining a non-overlapping and independent RNG.
- Easy to use, just make each stream threadprivate and initiate RNG stream so each thread gets a unique WG RNG.

```
VSLStreamStatePtr stream;

#pragma omp threadprivate(stream)

                              …

vslNewStream(&ran_stream, VSL_BRNG_WH+Thrd_ID, (int)seed);
```

Independent Generator for each thread



Notice that once you get beyond the high error, small sample count range, adding threads doesn’t decrease quality of random sampling.

Leap Frog method

- Interleave samples in the sequence of pseudo random numbers:
  - Thread i starts at the ith number in the sequence
  - Stride through sequence, stride length = number of threads.
- Result … the same sequence of values regardless of the number of threads.

```
#pragma omp single
{   nthreads = omp_get_num_threads();
     iseed = PMOD/MULTIPLIER;     // just pick a seed
     pseed[0] = iseed;
     mult_n = MULTIPLIER;
     for (i = 1; i < nthreads; ++i)
    {
        iseed = (unsigned long long)((MULTIPLIER * iseed) % PMOD);
        pseed[i] = iseed;
        mult_n = (mult_n * MULTIPLIER) % PMOD;
    }

 }
 random_last = (unsigned long long) pseed[id];
```

One thread computes offsets and strided multiplier

LCG with Addend = 0 just to keep things simple

Each thread stores offset starting point into its threadprivate “last random” value

Same sequence with many threads.

- We can use the leapfrog method to generate the same answer for any number of threads

Steps	One thread	2 threads	4 threads
1000	3.156	3.156	3.156
10000	3.1168	3.1168	3.1168
100000	3.13964	3.13964	3.13964
1000000	3.140348	3.140348	3.140348
10000000	3.141658	3.141658	3.141658

Used the MKL library with two generator streams per computation: one for the x values (WH) and one for the y values (WH+1). Also used the leapfrog method to deal out iterations among threads.