OpenMP: New Features

3.0 and beyond

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

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

SGI
- Merged, needed commonality across products

Cray
- ISV - needed larger market

KAI
- was tired of recoding for SMPs. Urged vendors to standardize.

Wrote a rough draft straw man SMP API

Other vendors invited to join

DEC

HP

IBM

Intel

ASCI

OpenMP

1997
OpenMP Release History

1997
OpenMP Fortran 1.0

1999
OpenMP Fortran 1.1

2000
OpenMP Fortran 2.0

1998
OpenMP C/C++ 1.0

2002
OpenMP C/C++ 2.0

2005
OpenMP 2.5

2008
OpenMP 3.0

2011
OpenMP 3.1

Tasking, runtime control over loop schedules, explicit control over nested parallel regions, refined control over resources.

A single specification for Fortran, C and C++

Expanded atomics, refined tasking, and more control over nested parallel regions
Outline

- Tasks (OpenMP 3.0)
  - The OpenMP Memory model (flush)
- Atomics (OpenMP 3.1)
- Recapitulation
Consider simple list traversal

• Given what we’ve covered about OpenMP, how would you process this loop in Parallel?

```c
p=head;
while (p) {
    process(p);
    p = p->next;
}
```

• Remember, the loop worksharing construct only works with loops for which the number of loop iterations can be represented by a closed-form expression at compiler time. While loops are not covered.
while (p != NULL) {
   p = p->next;
   count++;
}
parr = (*node) malloc(count * sizeof(struct node));
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++)
      process(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

This is really ugly! There has got to be a better way
OpenMP needed a more flexible way to define units of work: Tasks

- Tasks are independent units of work.
- Tasks are composed of:
  - **code** to execute
  - **data** environment
  - **internal control variables** (ICV)
- Threads perform the work of each task.
- The runtime system decides when tasks are executed
  - Tasks may be deferred
  - Tasks may be executed immediately
Task Construct – Explicit Tasks

1. Create a team of threads.

```c
#pragma omp parallel
{
    #pragma omp single
    {
        node * p = head;
        while (p) {
            #pragma omp task firstprivate(p)
            process(p);
            p = p->next;
        }
    }
}
```

2. One thread executes the `single` construct

   … other threads wait at the implied barrier at the end of the single construct

3. The “single” thread creates a task with its own value for the pointer p

4. Threads waiting at the barrier execute tasks.

   Execution moves beyond the barrier once all the tasks are complete
Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

```c
#pragma omp parallel
{
    #pragma omp single
    { //block 1
        node * p = head;
        while (p) { // block 2
            #pragma omp task
            process(p);
            p = p->next; //block 3
        }
    }
}
```
When are tasks guaranteed to complete

- Tasks are guaranteed to be complete at thread barriers:
  
  ```cpp
  #pragma omp barrier
  ```

- or task barriers
  
  ```cpp
  #pragma omp taskwait
  ```

```cpp
#pragma omp parallel
{
    #pragma omp task
    foo();
    #pragma omp barrier
    #pragma omp single
    {
        #pragma omp task
        bar();
    }
}
```

- Multiple foo tasks created here – one for each thread
- All foo tasks guaranteed to be completed here
- One bar task created here
- bar task guaranteed to be completed here
Data Scoping with tasks: Fibonacci example.

int fib ( int n )
{
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task
    x = fib(n-1);
    #pragma omp task
    y = fib(n-2);
    #pragma omp taskwait
    return x+y
}
Data Scoping with tasks: Fibonacci example.

```c
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;
}
```

n is private in both tasks

x & y are shared

**Good solution**
we need both values to compute the sum
Data Scoping with tasks: List Traversal example

List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
    for(e=ml->first;e;e=e->next)
#pragma omp task
    process(e);
}
Data Scoping with tasks: List Traversal example

List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
    for(e=ml->first; e; e=e->next)
    #pragma omp task firstprivate(e)
        process(e);
}
A real example: Symmetric rank-k update

\[ C_{10} \quad C_{11} \]

\[ C \]

Add \( A_1 A_T^0 \)

Add \( A_0 A_T^0 \)

Note: the iteration sweeps through \( C \) and \( A \), creating a new block of rows to be updated with new parts of \( A \). These updates are completely independent.

while ( FLA_Obj_length( CTL ) < FLA_Obj_length( C ) ) {
    b = min( FLA_Obj_length( CBR ), nb_alg );

    FLA_Repard_2x2_to_3x3( CTL, /**/ CTR, &C00, /**/ &C01, &C02,
                            /**/ /**/ &C10, /**/ &C11, &C12,
                            CBL, /**/ CBR, &C20, /**/ &C21, &C22,
                            b, b, FLA_BR );
    FLA_Repard_2x1_to_3x1( AT, &A0,
                            /* */ /* */
                            &A1,
                            AB, &A2, b, FLA_BOTTOM );

    /***************************************************/
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE, ONE, A0, A1, ONE, C10 );
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, ONE, A1, ONE, C11 );

    /***************************************************/
    FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR, C00, C01, /**/ C02,
                               C10, C11, /**/ C12,
                               /**/ /**/ &CBL, /**/ &CBR, C20, C21, /**/ C22,
                               FLA_TL );
    FLA_Cont_with_3x1_to_2x1( &AT, A0,
                               /* */ /* */
                               &A1,
                               AB, A2, FLA_TOP );
}
#pragma omp parallel
{
  #pragma omp single
  {
    while ( FLA_Obj_length( CTL ) < FLA_Obj_length( C ) ){
      b = min( FLA_Obj_length( CBR ), nb_alg );

      FLA_Repart_2x2_to_3x3( CTL, /*/ CTR, &C00, /*/ &C01, &C02,
                      /*/ */ &C10, /*/ &C11, &C12,
                      CBL, /*/ CBR, &C20, /*/ &C21, &C22,
                      b, b, FLA_BR );

      FLA_Repart_2x1_to_3x1( AT, &A0,
                       /* */ &A1,
                       AB, &A2, b, FLA_BOTTOM );
    }  // end of task
  }  // end of parallel region

  #pragma omp task firstprivate(A0, A1, C10, C11)
  {
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE, ONE, A0, A1, ONE, C10 );
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, ONE, A1, ONE, C11 );
  } /* end task */
} /* end task */

FLA_Cont_with_3x3_to_2x2( &CTL, /*/ &CTR, C00, C01, /*/ C02,
                      /*/ */ &C10, C11, /*/ &C12,
                      /*/ */ &CBL, /*/ &CBR, C20, C21, /*/ &C22,
                      FLA_TL );

FLA_Cont_with_3x1_to_2x1( &AT, A0,
                     A1,
                     */* */ &AB, A2, FLA_TOP );

} // end of task-queue
} // end of parallel region
Note: the above graphs is for the most naïve way of marching through the matrices. By picking blocks dynamically, much faster ramp-up can be achieved.
Outline

- Tasks (OpenMP 3.0)
- The OpenMP Memory model (flush)
- Atomics (OpenMP 3.1)
- Recapitulation
OpenMP memory model

- OpenMP supports a shared memory model.
- All threads share an address space … but what actually see at a given point in time may be complicated:

A memory model is defined in terms of:
- Coherence: Behavior of the memory system when a single address is accessed by multiple threads.
- Consistency: Orderings of reads, writes, or synchronizations (RWS) with various addresses and by multiple threads.
OpenMP Memory Model: Basic Terms

Source code

Program order

Compile order

Executable code

Code order

RW’s in any semantically equivalent order

private view

thread

memory

Commit order
Consistency: Memory Access Re-ordering

• Re-ordering:
  – Compiler re-orders program order to the code order
  – Machine re-orders code order to the memory commit order

• At a given point in time, the “private view” seen by a thread may be different from the view in shared memory.

• Consistency Models define constraints on the orders of Reads (R), Writes (W) and Synchronizations (S)
  – … i.e. how do the values “seen” by a thread change as you change how ops follow (→) other ops.
  – Possibilities include:
Consistency

• Sequential Consistency:
  – In a multi-processor, ops (R, W, S) are sequentially consistent if:
    – They remain in program order for each processor.
    – They are seen to be in the same overall order by each of the other processors.
    – Program order = code order = commit order

• Relaxed consistency:
  – Remove some of the ordering constraints for memory ops (R, W, S).
OpenMP and Relaxed Consistency

• OpenMP defines consistency as a variant of **weak consistency**:
  – S ops must be in sequential order across threads.
  – Can not reorder S ops with R or W ops on the same addresses on the same thread
    – Weak consistency guarantees
      S→W, S→R , R→S, W→S, S→S

• The Synchronization operation relevant to this discussion is flush.
Flush

• Defines a sequence point at which a thread is guaranteed to see a consistent view of memory with respect to the “flush set”.

• The flush set is:
  – “all thread visible variables” for a flush construct without an argument list.
  – a list of variables when the “flush(list)” construct is used.

• The action of Flush is to guarantee that:
  – All R,W ops that overlap the flush set and occur prior to the flush complete before the flush executes
  – All R,W ops that overlap the flush set and occur after the flush don’t execute until after the flush.
  – Flushes with overlapping flush sets can not be reordered.

Memory ops: R = Read,  W = write,  S = synchronization
Synchronization: flush example

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

```c
double A;
A = compute();
flush(A);  // flush to memory to make sure other threads can pick up the right value
```

Note: OpenMP’s flush is analogous to a fence in other shared memory API’s.
What is the Big Deal with Flush?

• Compilers routinely reorder instructions implementing a program
  – This helps better exploit the functional units, keep machine busy, hide memory latencies, etc.

• Compiler generally cannot move instructions:
  – past a barrier
  – past a flush on all variables

• But it can move them past a flush with a list of variables so long as those variables are not accessed

• Keeping track of consistency when flushes are used can be confusing … especially if “flush(list)” is used.

Note: the flush operation does not actually synchronize different threads. It just ensures that a thread’s values are made consistent with main memory.
Example: prod_cons.c

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

```c
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.
Pair wise synchronization in OpenMP

- OpenMP lacks synchronization constructs that work between pairs of threads.
- When this is needed you have to build it yourself.
- Pair wise synchronization
  - Use a shared flag variable
  - Reader spins waiting for the new flag value
  - Use flushes to force updates to and from memory
Example: producer consumer

```c
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);
        }
        #pragma omp flush
    }
}
```

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
Outline

• Tasks (OpenMP 3.0)
• The OpenMP Memory model (flush)
• Atomics (OpenMP 3.1)
• Recapitulation
This program only works since we don’t really care about the value of flag … all we care is that the flag no longer equals zero.

Why is there a problem communicating the actual value of flag? Doesn’t the flush assure the flag value is cleanly communicated?
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);
        }
    }
}
Remember the Atomic construct?

• The original OpenMP atomic was too restrictive …. For example it didn’t include a simple atomic store.

Additional forms of atomic were added in OpenMP 3.1. We will discuss these later.
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:

  \[ \begin{align*}
  v &= x++; & v &= ++x; & v &= x--; & v &= -x; & v &= x \text{ binop expr};
  \end{align*} \]

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

  \[ \begin{align*}
  \{ v &= x; \quad x \text{ binop = expr}; \} & \{ x \quad \text{binop = expr}; \quad v &= x; \} \\
  \{ v = x; \quad x = x \text{ binop expr}; \} & \{ X = x \text{ binop expr}; \quad v &= x; \} \\
  \{ v = x; \quad x++; \} & \{ v = x; \quad ++x; \} \\
  \{ ++x; \quad v = x; \} & \{ x++; \quad v = x; \} \\
  \{ v = x; \quad x--; \} & \{ v = x; \quad --x; \} \\
  \{ --x; \quad v = x; \} & \{ x--; \quad v = x; \}
  \end{align*} \]

The capture semantics in atomic were added to map onto common hardware supported atomic ops and to support modern lock free algorithms.
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 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 can not conflict.
Outline

• Tasks (OpenMP 3.0)
• The OpenMP Memory model (flush)
• Atomics (OpenMP 3.1)
• Recapitulation
If you become overwhelmed during this course …

• Come back to this slide and remind yourself … things are not as bad as they seem.

**Parallel programming is easy**

• So all you need to do is:
  - **Pick** your language.
    - I suggest sticking to industry standards and open source so you can move around between hardware platforms:
      - pthreads
      - OpenMP
      - OpenCL
      - MPI
      - TBB
  - **Learn** the key 7 patterns
    - SPMD
    - Kernel Parallelism
    - Fork/join
    - Actors
  - **Master** the few patterns common to your platform and application domain … for example, most application programmers just use these three patterns
    - SPMD
    - Kernel Parallelism
    - Loop Parallelism
SPMD: Single Program Multiple Data

• Run the same program on P processing elements where P can be arbitrarily large.
• Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.

This pattern is very general and has been used to support most (if not all) the algorithm strategy patterns.

MPI programs almost always use this pattern … it is probably the most commonly used pattern in the history of parallel programming.
OpenMP Pi program: SPMD pattern

```c
#include <omp.h>
void main (int argc, char *argv[])
{
    int i, pi=0.0, step, sum = 0.0;
    step = 1.0/(double) num_steps ;
#pragma omp parallel firstprivate(sum) private(x, i)
{
    int id = omp_get_thread_num();
    int numprocs = omp_get_num_threads();
    int step1 = id *num_steps/numprocs ;
    int stepN = (id+1)*num_steps/numprocs;
    if (stepN != num_steps) stepN = num_steps;
    for (i=step1; i<stepN; i++)
    {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
#pragma omp critical
    pi += sum *step ;
}
```
Loop parallelism

• Collections of tasks are defined as iterations of one or more loops.
• Loop iterations are divided between a collection of processing elements to compute tasks in parallel.

```
#pragma omp parallel for shared(Results) schedule(dynamic)
for(i=0;i<N;i++){
    Do_work(i, Results);
}
```

This design pattern is heavily used with data parallel design patterns. OpenMP programmers commonly use this pattern.
OpenMP PI Program: 
Loop level parallelism pattern

#include <omp.h>
static long num_steps = 100000;       double step;
#define NUM_THREADS 2
void main ()
{
    int i;       double x, pi, sum =0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel for private(x) reduction (+:sum)
    for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    pi = sum * step;
}
Fork-join

- **Use when:**
  - Target platform has a shared address space
  - Dynamic task parallelism

- **Particularly useful when you have a serial program to transform incrementally into a parallel program**

- **Solution:**
  1. A computation begins and ends as a single thread.
  2. When concurrent tasks are desired, additional threads are forked.
  3. The thread carries out the indicated task,
  4. The set of threads recombine (join)

*Cilk and OpenMP make heavy use of this pattern.*
#include <stdio.h>
#include <pthread.h>
#define NSTEPS 10000000
#define NTHRS 4
double gStep=0.0, gPi=0.0;
pthread_mutex_t gLock;
void *Func(void *pArg)
{
    int i, ID = *((int *)pArg);
    double partialSum = 0.0, x;
    for(i=ID;i<NSTEPS;i+=NTHRS)
    {
        x = (i + 0.5f) * gStep;
        partialSum +=
            4.0f/(1.0f+x*x);
    }
    pthread_mutex_lock(&gLock);
    gPi += partialSum * gStep;
    pthread_mutex_unlock(&gLock);
    return 0;
}

int main()
{
    pthread_t thrds[NTHRS];
    int tNum[NTHRS], i;
    pthread_mutex_init(&gLock,NULL);
    gStep = 1.0 / NSTEPS;
    for (i = 0; i < NTHRS; ++i )
    {
        tRank[i] = i;
        pthread_create(&thrds[i],NULL,
                       Func,(void)&tRank[i]);
    }
    for (i = 0; i < NTHRS; ++i )
    {
        pthread_join(thrds[i], NULL);
    }
    pthread_mutex_destroy(&gLock);
    return 0;
}
Divide and Conquer Pattern

• Use when:
  – A problem includes a method to divide into subproblems and a way to recombine solutions of subproblems into a global solution.

• Solution
  – Define a split operation
  – Continue to split the problem until subproblems are small enough to solve directly.
  – Recombine solutions to subproblems to solve original global problem.

• Note:
  – Computing may occur at each phase (split, leaves, recombine).
Divide and conquer

- Split the problem into smaller sub-problems. Continue until the sub-problems can be solved directly.

3 Options:
- Do work as you split into sub-problems.
- Do work only at the leaves.
- Do work as you recombine.
Program: OpenMP tasks (divide and conquer pattern)

```c
#include <omp.h>
static long num_steps = 100000000;
#define MIN_BLK 10000000
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

• Original Serial pi program with 100000000 steps ran in 1.83 seconds.

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.
Kernel Parallelism

- Kernel Parallelism:
  - Implement data parallel problems:
    - Define an abstract index space that appropriately spans the problem domain.
    - Data structures in the problem are aligned to this index space.
    - Tasks (e.g. work-items in OpenCL or “threads” in CUDA) operate on these data structures for each point in the index space.

- This approach was popularized for graphics applications where the index space mapped onto the pixels in an image.

- In the last ~10 years, It’s been extended to General Purpose GPU (GPGPU) programming for heterogeneous platforms.

Note: This is basically a fine grained extreme form of the SPMD pattern.
OpenCL: An N-dim. domain of work-items

• Define an N-dimensioned index space that is “best” for your algorithm
  – Global Dimensions: 1024 x 1024 (whole problem space)
  – Local Dimensions: 128 x 128 (work group … executes together)

Synchronization between work-items possible only within workgroups: barriers and memory fences

Cannot synchronize outside of a workgroup
OpenCL PI Program:
Kernel parallelism pattern (host code not shown)

```c
__kernel void pi(     const int niters,                const float step_size,
                __local float* local_sums,  __global float* partial_sums)
{
    int num_wrk_items = get_local_size(0), local_id = get_local_id(0);
    int group_id = get_group_id(0), i, istart, iend;
    float x, sum, accum = 0.0f;
    istart = (group_id * num_wrk_items + local_id) * niters;
    iend = istart+niters;
    for(i= istart; i<iend; i++){
        x = (i+0.5f)*step_size;
        accum += 4.0f/(1.0f+x*x);
    }
    local_sums[local_id] = accum;
    barrier(CLK_LOCAL_MEM_FENCE);
    if (local_id == 0){
        sum = 0.0f;
        for(i=0; i<num_wrk_items;i++){
            sum += local_sums[i];
        }
        partial_sums[group_id] = sum;
    }
}
```

Geometric decomposition to define work for each OpenCL work-item.

Local sum per work-item saved in a local array (shared inside workgroup)

One work item combines work from all the work-items in the group

Store results from this work-group in the globally visible buffer. Finish the sum on the host.
Vector Parallelism

- Definition: A single instruction stream is applied to multiple data elements.
  - One program text
  - One instruction counter
  - Distinct data streams per PE
SSE intrinsics PI Program:
Vector parallelism pattern

#include "xmmintrin.h"
float pi_sse_double(int num_steps) {
    int i;
    double step, pi;
    double scalar_one = 1.0,
    double scalar_zero = 0.0;
    double ival, scalar_four = 4.0;
    double vsum[2];
    step = 1.0/(double) num_steps;
    __m128d xvec;
    __m128d denom;
    __m128d eye;
    __m128d ramp = _mm_setr_pd(0.5, 1.5);
    __m128d one = _mm_load1_pd(&scalar_one);
    __m128d four = _mm_load1_pd(&scalar_four);
    __m128d vsum = _mm_load1_pd(&vsum[0]);
    __m128d sum = _mm_load1_pd(&scalar_zero);

    for (i=0;i< num_steps; i=i+2){
        ival = (double)i;
        eye = _mm_load1_pd(&ival);
        xvec = _mm_mul_pd(
            _mm_add_pd(eye,ramp),vstep);
        denom = _mm_add_pd(
            _mm_mul_pd(xvec,xvec),one);
        sum = _mm_add_pd(
            _mm_div_pd(four,denom),sum);
    }
    __m_store_pd(&vsum[0],sum);
    pi = step * (vsum[0]+vsum[1]);
    return (float)pi;
}
If you become overwhelmed during this course ...

- Come back to this slide and remind yourself ... things are not as bad as they seem

**Parallel programming is easy**

- **So all you need to do is:**
  - **Pick** your language.
    - I suggest sticking to industry standards and open source so you can move around between hardware platforms:
      - pthreads
      - OpenMP
      - OpenCL
      - MPI
      - TBB
  - **Learn** the key 7 patterns
    - SPMD
    - Kernel Parallelism
    - Fork/join
    - Actors
    - **Vector Parallelism**
    - **Loop Parallelism**
    - **Work Pile**
  - **Master** the few patterns common to your platform and application domain ... for example, most application programmers just use these three patterns
    - SPMD
    - Kernel Parallelism
    - Loop Parallelism
OpenMP summary

• We have covered most of OpenMP … enough so you can start writing real parallel applications with OpenMP.
• We have discussed the most common patterns with OpenMP as well …. 
  **Loop level parallelism, fork/join, divide and conquer**
• The next step is up to you … write lot’s of code!!!

• `#pragma omp parallel`
• `#pragma omp for`
• `#pragma omp critical`
• `#pragma omp atomic`
• `#pragma omp barrier`
• Data environment clauses
  – `private (variable_list)`
  – `firstprivate (variable_list)`
  – `lastprivate (variable_list)`
  – `reduction(+:variable_list)`
• Tasks (remember … private data is made firstprivate by default)
  – `pragma omp task`
  – `pragma omp taskwait`
• `#pragma threadprivate(variable_list)`

Where variable_list is a comma separated list of variables

Put this on a line right after you define the variables in question
Backup

• References
  • Threadprivate Data and random numbers
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 compunity and help define the future of OpenMP
Books about OpenMP

An excellent book about using OpenMP ... though out of date (OpenMP 2.5)

A book about how to “think parallel” with examples in OpenMP, MPI and Java
Background references

A general reference that puts languages such as OpenMP in perspective (by Sottile, Mattson, and Rasmussen)

An excellent introduction and overview of multithreaded programming (by Clay Breshears)
The OpenMP reference card
A two page summary of all the OpenMP constructs ... don’t write OpenMP code without it.

OpenMP Papers


• Bentz J., Kendall R., “Parallelization of General Matrix Multiply Routines Using OpenMP”, Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 1, 2005


OpenMP Papers (continued)

• Jost G., Labarta J., Gimenez J., What Multilevel Parallel Programs do when you are not watching: a Performance analysis case study comparing MPI/OpenMP, MLP, and Nested OpenMP, Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 29, 2005


OpenMP Papers (continued)


Backup

• References

• Threadprivate Data and random numbers
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 thread private example (C)

Use threadprivate to create a counter for each thread.

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

int increment_counter()
{
    counter++;        
    return (counter);
}
```
Data Copying: Copyin

You initialize threadprivate data using a copyin clause.

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

C 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
```
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.

```c
#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);
    }
}
```
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 $\pi$ with a digital dart board:

  ● Throw darts at the circle/square.
  ● Chance of falling in circle is proportional to ratio of areas:
    \[ A_c = r^2 \times \pi \]
    \[ A_s = (2 \times r) \times (2 \times r) = 4 \times r^2 \]
    \[ P = \frac{A_c}{A_s} = \frac{\pi}{4} \]
  ● Compute $\pi$ by randomly choosing points, count the fraction that falls in the circle, compute pi.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\pi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2.8</td>
</tr>
<tr>
<td>100</td>
<td>3.16</td>
</tr>
<tr>
<td>1000</td>
<td>3.148</td>
</tr>
</tbody>
</table>
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 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).
Parallel Programmers love Monte Carlo algorithms

```c
#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 square 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("%d trials, pi is %f \n",num_trials, pi);
}
```

Embarrassingly parallel: the parallelism is so easy its embarrassing.
Add two lines and you have a parallel program.
Linear Congruential Generator (LCG)

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

```plaintext
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);
}
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.
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 + ADDEND)% PMOD;
    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

• 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!
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 thread 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

```c
#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 of pseudo random numbers

Select type of RNG and set seed

Fill buff with BLOCK pseudo rand. nums, uniformly distributed with values between low 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.

```c
VSLStreamStatePtr stream;
#pragma omp threadprivate(stream)
...
vsINewStream(&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 i\textsuperscript{th} number in the sequence
  - Stride through sequence, stride length = number of threads.
- Result … the same sequence of values regardless of the number of threads.

```c
#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

<table>
<thead>
<tr>
<th>Steps</th>
<th>One thread</th>
<th>2 threads</th>
<th>4 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>3.156</td>
<td>3.156</td>
<td>3.156</td>
</tr>
<tr>
<td>10000</td>
<td>3.1168</td>
<td>3.1168</td>
<td>3.1168</td>
</tr>
<tr>
<td>100000</td>
<td>3.13964</td>
<td>3.13964</td>
<td>3.13964</td>
</tr>
<tr>
<td>1000000</td>
<td>3.140348</td>
<td>3.140348</td>
<td>3.140348</td>
</tr>
<tr>
<td>10000000</td>
<td>3.141658</td>
<td>3.141658</td>
<td>3.141658</td>
</tr>
</tbody>
</table>