Programming with OpenMP*

Agenda

- What is OpenMP?
- Parallel regions
- Data-Sharing Attribute Clauses
- Worksharing
- OpenMP 3.0 Tasks
- Synchronization
- Runtime functions/environment variables
- Optional Advanced topics

What Is OpenMP?

- Portable, shared-memory threading API
  - Fortran, C, and C++
  - Multi-vendor support for both Linux and Windows
- Stand … arallelism
- Supp … sm
- Comb … e in single sourc …
- Stand … iler-directed threading experience

http://www.openmp.org

Current spec is OpenMP 3.0

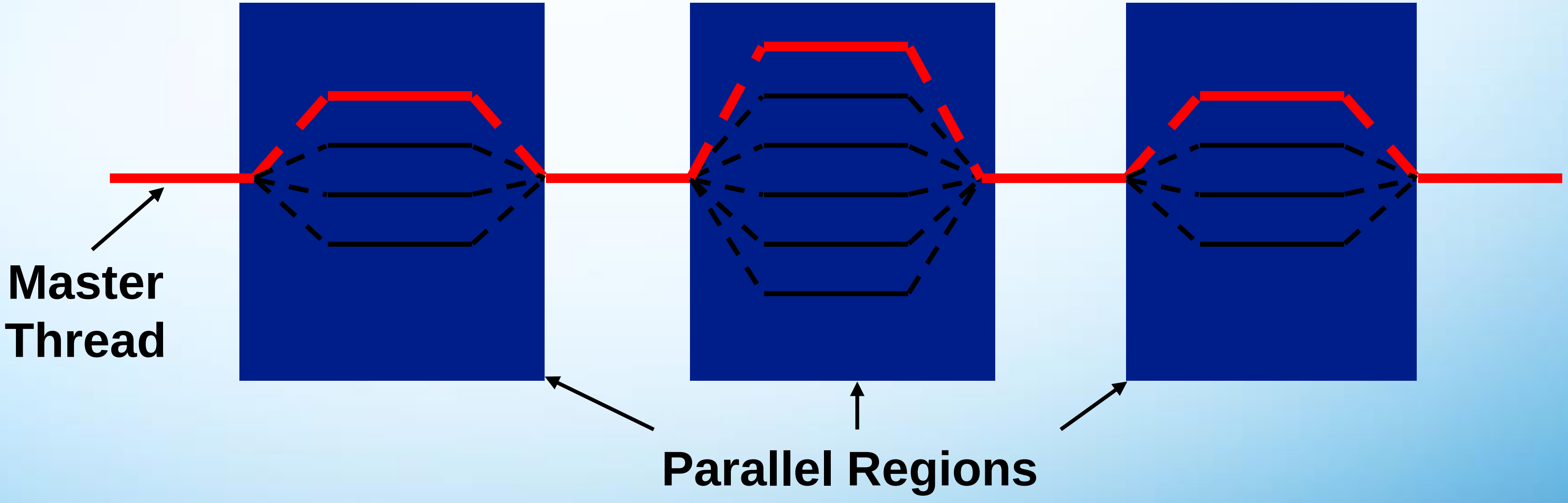
318 Pages

(combined C/C++ and Fortran)

Programming Model

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed
- Parallelism is added incrementally: that is, the sequential program evolves into a parallel program



A Few Details to Get Started

- Compiler option /Qopenmp in Windows
    or –openmp in Linux
- Most of the constructs in OpenMP are compiler directives or pragmas
    - For C and C++, the pragmas take the form:

      ```
      #pragma omp construct [clause [clause]…]
      ```

    - For Fortran, the directives take one of the forms:

      ```
      C$OMP construct [clause [clause]…]
      !$OMP construct [clause [clause]…]
      *$OMP construct [clause [clause]…]
      ```

- Header file or Fortran module

  ```
  #include "omp.h"
  use omp_lib
  ```

Agenda

- What is OpenMP?
- Parallel regions
- Data-Sharing Attribute Clauses
- Worksharing
- OpenMP 3.0 Tasks
- Synchronization
- Runtime functions/environment variables
- Intel® Parallel Debugger Extension
- Optional Advanced topics

Parallel Region & Structured Blocks (C/C++)

- Most OpenMP constructs apply to structured blocks
  - Structured block: a block with one point of entry at the top and one point of exit at the bottom
  - The only "branches" allowed are STOP statements in Fortran and exit() in C/C++

A structured block	Not a structured block

`#pragma omp parallel`
`{`
`  int id = omp_get_thread_num();`
 	
`more: res[id] = do_big_job (id);`
 	
`  if (conv (res[id])) goto more;`
`}`
`printf ("All done\n");`
	`if (go_now()) goto more;`
	`#pragma omp parallel`
	`{`
	`  int id = omp_get_thread_num();`
	`more:  res[id] = do_big_job(id);`
	`  if (conv (res[id])) goto done;`
	`  goto more;`
	`}`
	`done: if (!really_done()) goto more;`

Agenda

- What is OpenMP?
- Parallel regions
- Data-Sharing Attribute Clauses
- Worksharing
- OpenMP 3.0 Tasks
- Synchronization
- Runtime functions/environment variables
- Intel® Parallel Debugger Extension
- Optional Advanced topics

Data-Sharing Attribute Clauses

shared	declares one or more list items to be shared by tasks generated by a parallel or task construct (Default).
private	declares one or more list items to be private to a task.
default	allows the user to control the data-sharing attributes of variables that are referenced in a parallel or task construct, and whose data-sharing attributes are implicitly determined
firstprivate	declares one or more list items to be private to a task, and initializes each of them with the value that the corresponding original item has when the construct is encountered.
lastprivate	declares one or more list items to be private to an implicit task, and causes the corresponding original list item to be updated after the end of the region.
reduction	specifies an operator and one or more list items. For each list item, a private copy is created in each implicit task, and is initialized appropriately for the operator. After the end of the region, the original list item is updated with the values of the private copies using the specified operator.

The Private Clause

- Reproduces the variable for each task
  - Variables are un-initialized; C++ object is default constructed
  - Any value external to the parallel region is undefined

```
void* work(float* c, int N) {
 float x, y; int i;
 #pragma omp parallel for private(x,y)
   for(i=0; i<N; i++) {
         x = a[i]; y = b[i];
         c[i] = x + y;
   }
}
```

Firstprivate Clause

- Variables initialized from shared variable
- C++ objects are copy-constructed

```
incr=0;
#pragma omp parallel for firstprivate(incr)
for (I=0;I<=MAX;I++) {
        if ((I%2)==0) incr++;
        A[I]=incr;
}
```

Lastprivate Clause

- Variables update shared variable using value from last iteration
- C++ objects are updated as if by assignment

```
void sq2(int n,
      double *lastterm)
{
 double x; int i;
 #pragma omp parallel
 #pragma omp for lastprivate(x)
 for (i = 0; i < n; i++){
   x = a[i]*a[i] + b[i]*b[i];
   b[i] = sqrt(x);
 }
 *lastterm = x;
}
```

OpenMP* Reduction Clause

```
reduction (op : list)
```

- The variables in "*list*" must be shared in the enclosing parallel region
- Inside parallel or work-sharing construct:
  - A PRIVATE copy of each list variable is created and initialized depending on the "op"
  - These copies are updated locally by threads
  - At end of construct, local copies are combined through "op" into a single value and combined with the value in the original SHARED variable

Activity 1 – Hello Worlds

- Modify the “Hello, Worlds” serial code to run multithreaded using OpenMP*

Agenda

- What is OpenMP?
- Parallel regions
- Data-Sharing Attribute Clauses
- Worksharing
- OpenMP 3.0 Tasks
- Synchronization
- Runtime functions/environment variables
- Optional Advanced topics

Worksharing

- **Worksharing** is the general term used in OpenMP to describe distribution of work across threads.

- Three examples of worksharing in OpenMP are:
- loop(omp for) construct
- omp sections construct
- omp task construct

Automatically divides work among threads

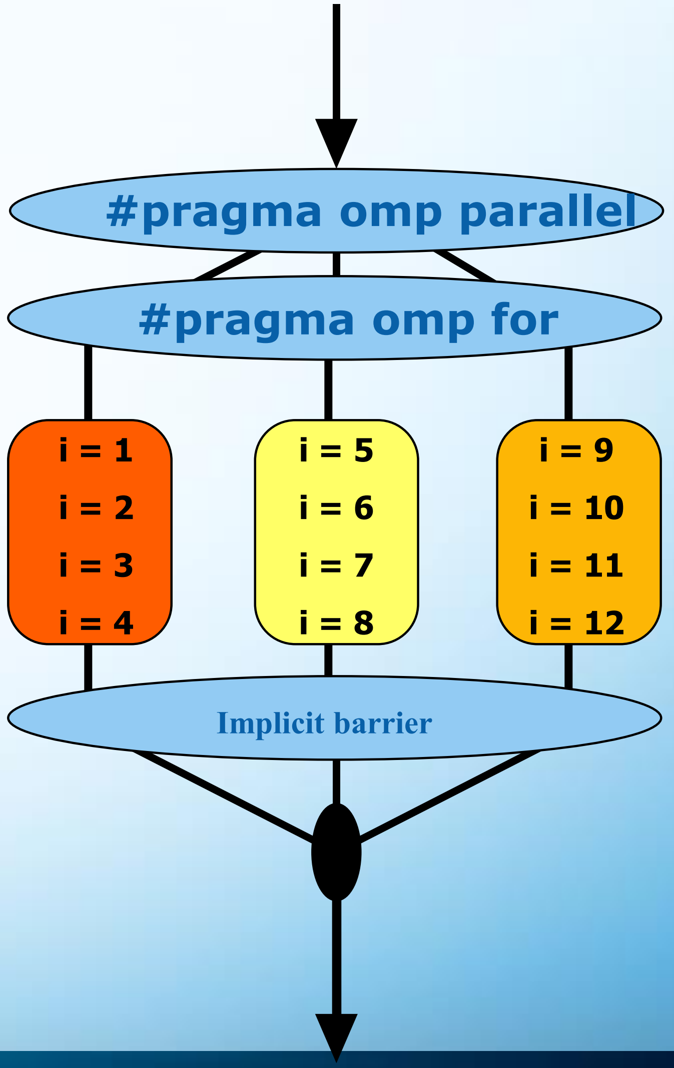
Agenda

- What is OpenMP?
- Parallel regions
- Data-Sharing Attribute Clauses
- Worksharing – Parallel For
- OpenMP 3.0 Tasks
- Synchronization
- Runtime functions/environment variables
- Optional Advanced topics

omp for Construct

```
// assume N=12
#pragma omp parallel
#pragma omp for
  for(i = 1, i < N+1, i++)
    c[i] = a[i] + b[i];
```

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct



Combining constructs

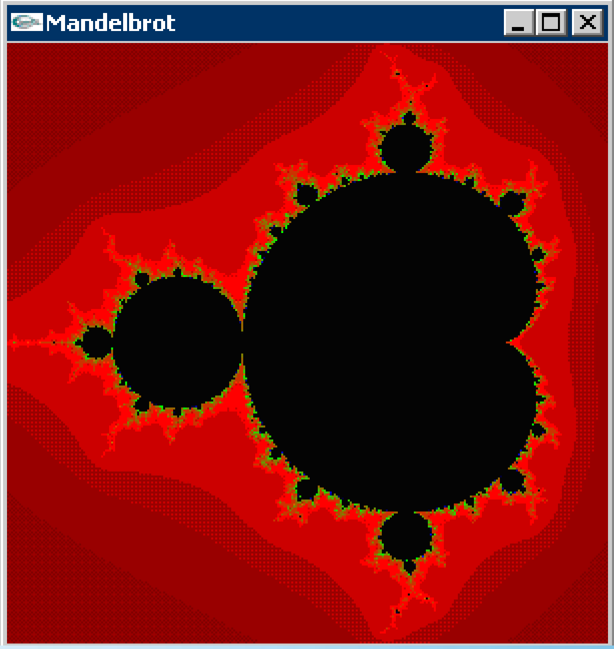
- These two code segments are equivalent

```
#pragma omp parallel
{
   #pragma omp for
   for (i=0;i< MAX; i++) {
         res[i] = huge();
   }
}
```

```
#pragma omp parallel for
   for (i=0;i< MAX; i++) {
         res[i] = huge();
   }
```

Activity 2 – Parallel Mandelbrot

Objective: create a parallel version of Mandelbrot. Modify the code to add OpenMP worksharing clauses to parallelize the computation of Mandelbrot.



The schedule clause

The schedule clause affects how loop iterations are mapped onto threads

schedule(static [,chunk])

- Blocks of iterations of size "chunk" to threads
- Round robin distribution
- Low overhead, may cause load imbalance

schedule(dynamic[,chunk])

- Threads grab "chunk" iterations
- When done with iterations, thread requests next set
- Higher threading overhead, can reduce load imbalance

schedule(guided[,chunk])

- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than "chunk"

schedule(runtime)

- Scheduling is deferred until run time, and the schedule and chunk size are taken from the *run-sched-var ICV*

Assigning Loop Iterations in OpenMP*: Which Schedule to Use

Schedule Clause	When To Use
STATIC	Predictable and similar work per iteration
DYNAMIC	Unpredictable, variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead
RUNTIME	Modify schedule at run-time via environment variable

Schedule Clause Example

```
#pragma omp parallel for schedule (static, 8)
   for( int i = start; i <= end; i += 2 )
   {
     if ( TestForPrime(i) )  gPrimesFound++;
   }
```

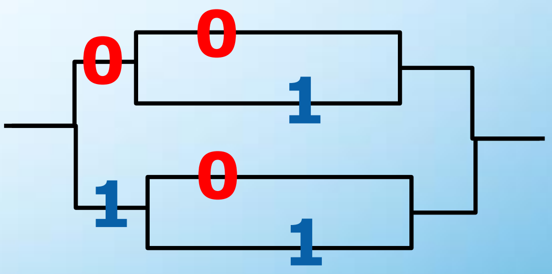
Iterations are divided into chunks of 8

- If start = 3, then first chunk is **i**={3,5,7,9,11,13,15,17}

Nested Parallelism

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
int n;
omp_set_nested(1);
#pragma omp parallel private(n)
{
n=omp_get_thread_num();
#pragma omp parallel
printf("thread number %d - nested thread number %d\n", n, omp_get_thread_num());
}
}
```

```
thread number 0 - nested thread number 0
thread number 0 - nested thread number 1
thread number 1 - nested thread number 0
thread number 1 - nested thread number 1
```



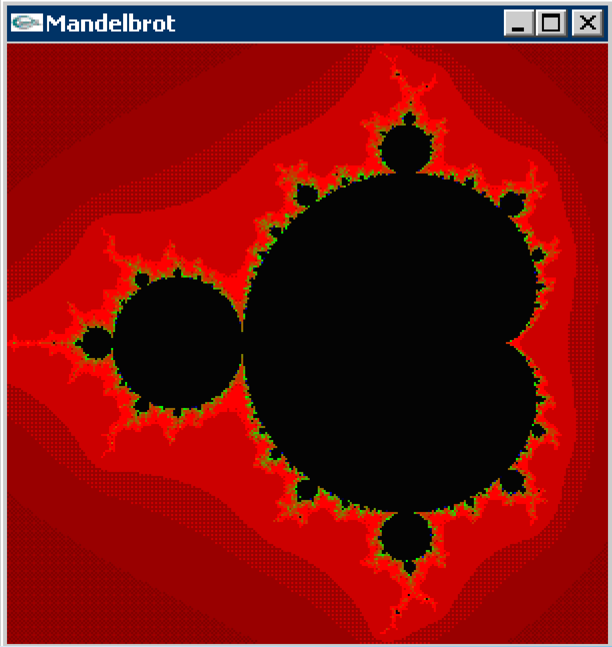
The collapse clause

Collapse is new concept in OpenMP 3.0. May be used to specify how many loops are associated with the loop construct.

```
#pragma omp parallel for collapse(2)
    for( i = 0; i < N; ++i )
    for( k = 0; k < N; ++k )
      for( j = 0; j < N; ++j )
                foo();
```

Activity 3 –Mandelbrot Scheduling

Objective: create a parallel version of mandelbrot. That uses OpenMP dynamic scheduling

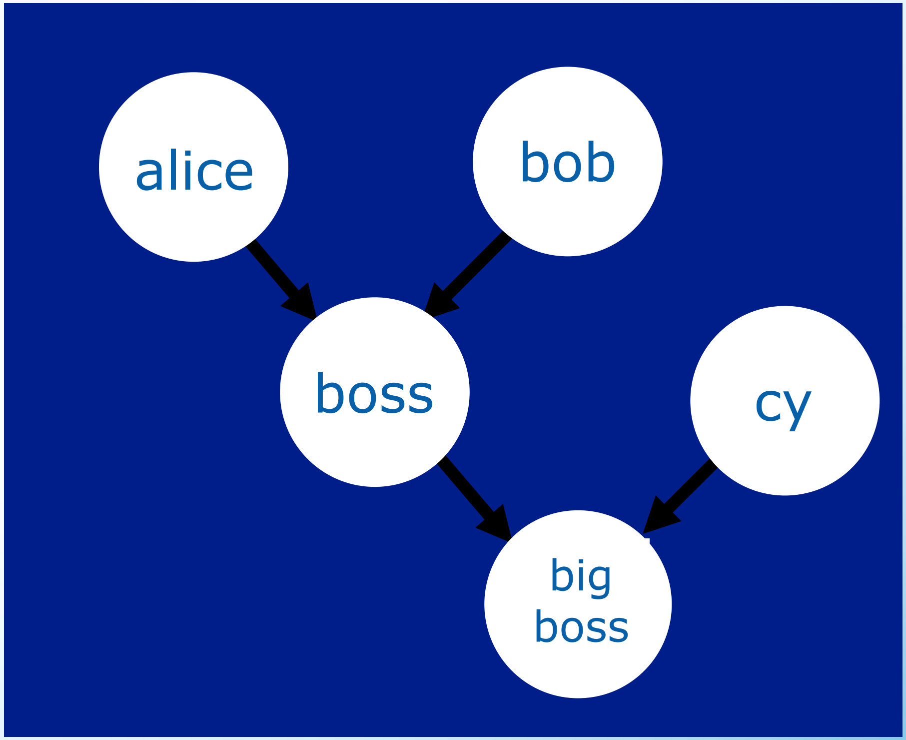


Agenda

- What is OpenMP?
- Parallel regions
- Data-Sharing Attribute Clauses
- Worksharing – Parallel Sections
- OpenMP 3.0 Tasks
- Synchronization
- Runtime functions/environment variables
- Optional Advanced topics

Task Decomposition

```
a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf ("%6.2f\n",
     bigboss(s,c));
```



alice,bob, and cy can be computed in parallel

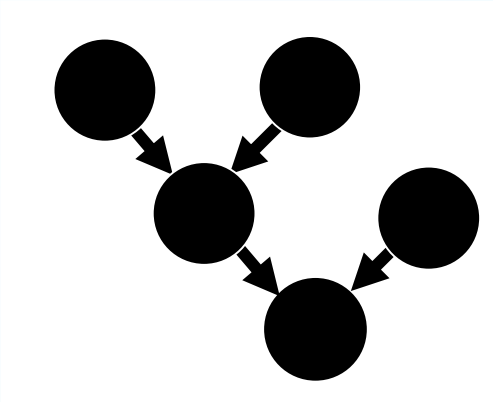
omp sections

- `#pragma omp sections`
- Must be inside a parallel region
- Precedes a code block containing of *N* blocks of code that may be executed concurrently by *N* threads
- Encompasses each omp section

- `#pragma omp section`
- Precedes each block of code within the encompassing block described above
- May be omitted for first parallel section after the parallel sections pragma
- Enclosed program segments are distributed for parallel execution among available threads

Functional Level Parallelism w sections

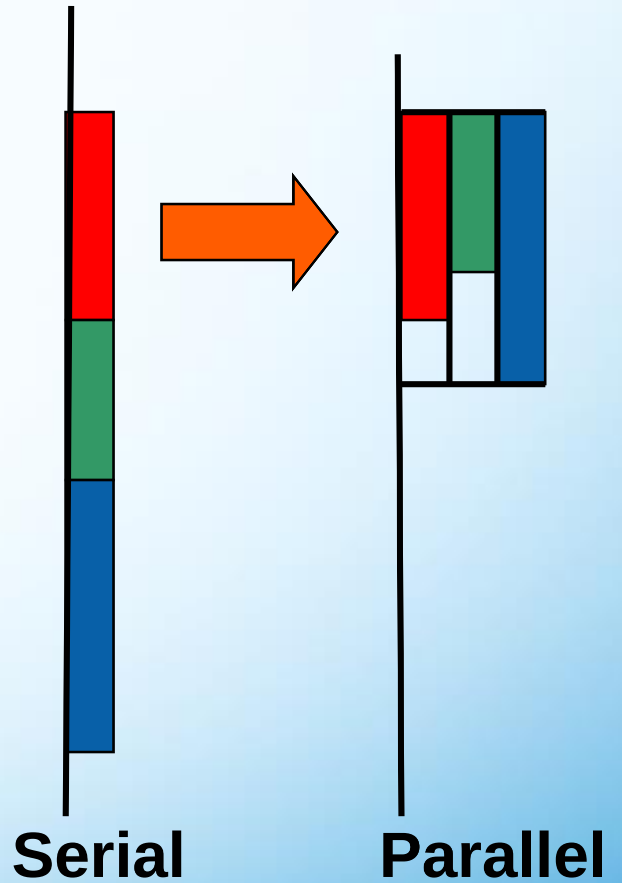
```
#pragma omp parallel sections
{
#pragma omp section  /* Optional */
   a = alice();
#pragma omp section
   b = bob();
#pragma omp section
   c = cy();
}

s = boss(a, b);
printf ("%6.2f\n",
        bigboss(s,c));
```



Advantage of Parallel Sections

- Independent sections of code can execute concurrently – reduce execution time

```
#pragma omp parallel sections
{
   #pragma omp section
   phase1();
   #pragma omp section
   phase2();
   #pragma omp section
   phase3();
}
```



Agenda

- What is OpenMP?
- Parallel regions
- Data-Sharing Attribute Clauses
- Worksharing
- OpenMP 3.0 Tasks
- Synchronization
- Runtime functions/environment variables
- Optional Advanced topics

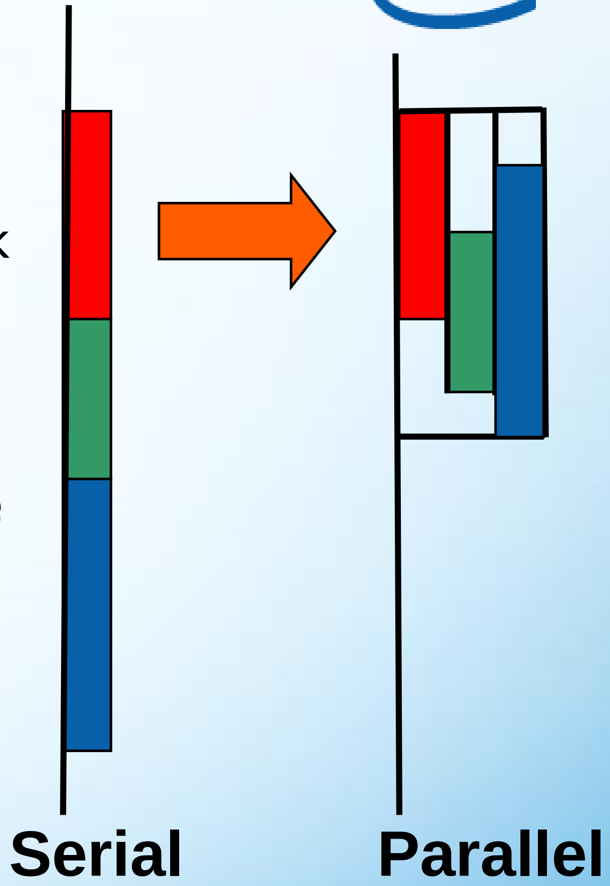
New Addition to OpenMP

- Tasks – Main change for OpenMP 3.0
- Allows parallelization of irregular problems
    - unbounded loops
    - recursive algorithms
    - producer/consumer

What are tasks?

- Tasks are independent units of work
- Threads are assigned to perform the work of each task
  - Tasks may be deferred
- Tasks may be executed immediately
- The runtime system decides which of the above

  - Tasks are composed of:
    - **code** to execute
    - **data** environment
    - **internal control variables** (ICV)



Simple Task Example

```
#pragma omp parallel
// assume 8 threads
{
  #pragma omp single private(p)
  {
    …
    while (p) {
    #pragma omp task
      {
        processwork(p);
      }
      p = p->next;
    }
  }
}
```

A pool of 8 threads is created here

One thread gets to execute the while loop

The single “while loop” thread creates a task for each instance of processwork()

Task Construct – Explicit Task View

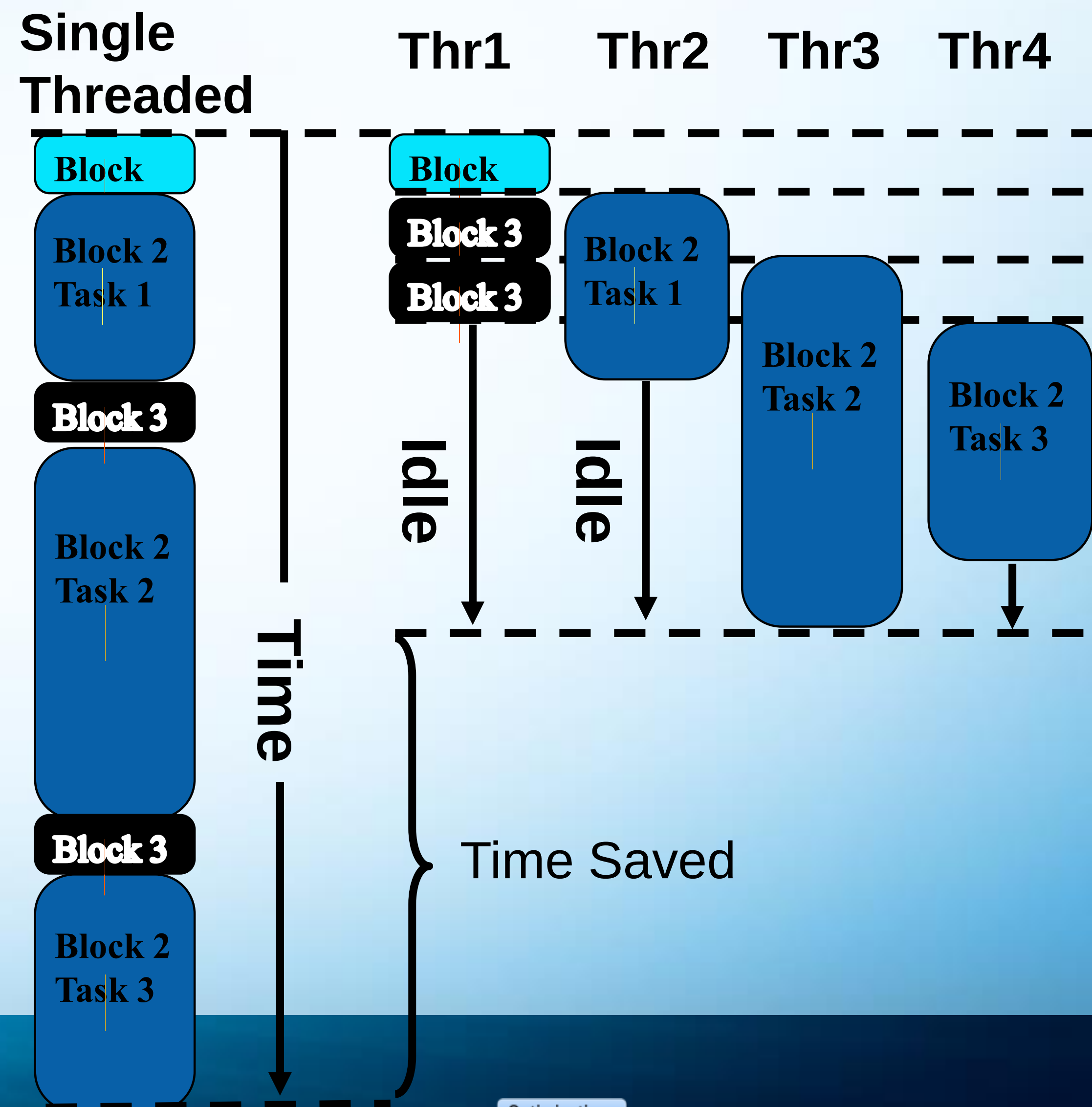
- A team of threads is created at the omp parallel construct
- A single thread is chosen to execute the while loop – lets call this thread “L”
- Thread L operates the while loop, creates tasks, and fetches next pointers
- Each time L crosses the omp task construct it generates a new task and has a thread assigned to it
- Each task runs in its own thread
- All tasks complete at the barrier at the end of the parallel region’s single construct

```
#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
    }
  }
}
```

Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

```
#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
    }
  }
}
```



Activity 4 – Parallel Fibonacci

Objective:

1. create a parallel version of Fibonacci sample. That uses OpenMP tasks;
2. try to find balance between serial and parallel recursion parts.

38

When are tasks gauranteed to be complete?

Tasks are gauranteed to be complete:

- At thread or task barriers
- At the directive: **#pragma omp barrier**
- At the directive: **#pragma omp taskwait**

Task Completion Example

```
#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

Agenda

- What is OpenMP?
- Parallel regions
- Data-Sharing Attribute Clauses
- Worksharing
- OpenMP 3.0 Tasks
- Synchronization
- Runtime functions/environment variables
- Optional Advanced topics

Example: Dot Product

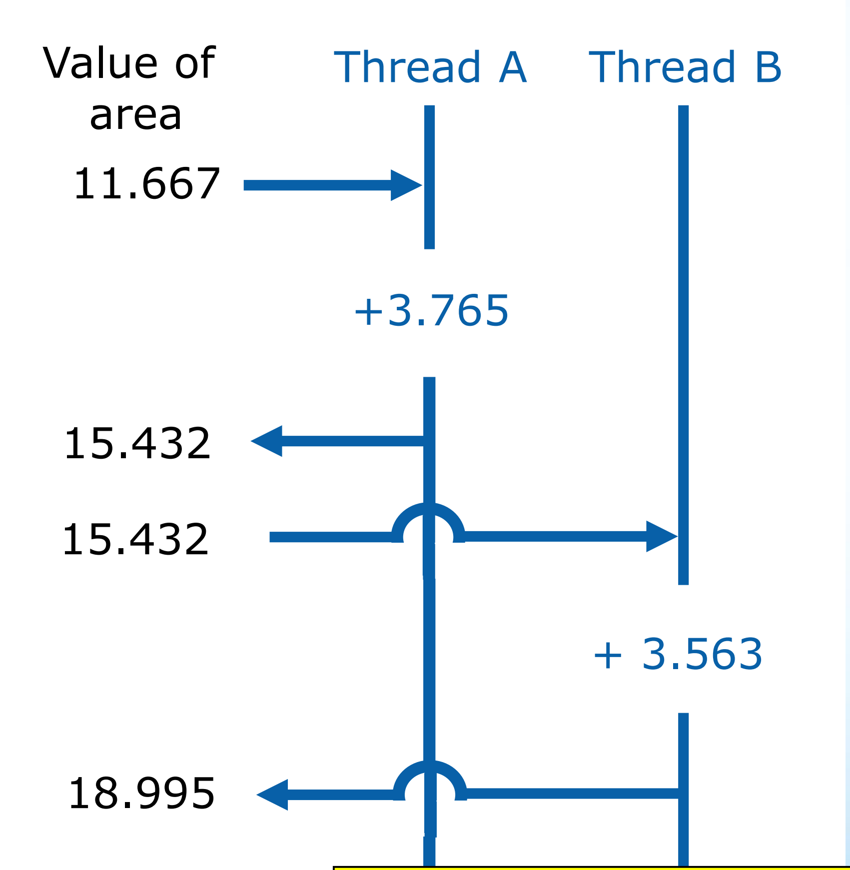
```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
  return sum;
}
```

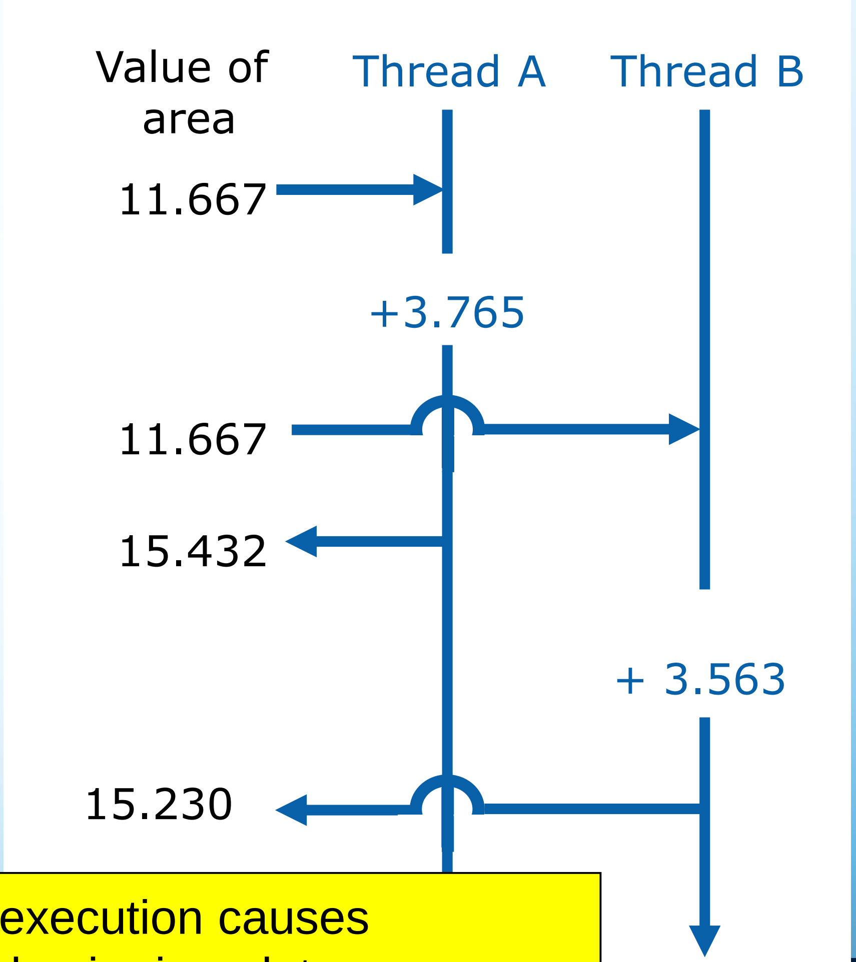
What is Wrong?

Race Condition

- A *race condition* is nondeterministic behavior caused by the times at which two or more threads access a shared variable
- For example, suppose both Thread A and Thread B are executing the statement
- `area += 4.0 / (1.0 + x*x);`

Two Timings





Order of thread execution causes
non determinant behavior in a data race

Protect Shared Data

- Must protect access to shared, modifiable data

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
#pragma omp critical
     sum += a[i] * b[i];
   }
  return sum;
}
```

OpenMP* Critical Construct

- `#pragma omp critical [(lock_name)]`

- Defines a critical region on a structured block

Threads wait their turn –at a time, only one calls **consum()** thereby protecting RES from race conditions

Naming the critical construct **RES_lock** is optional

```
float RES;
#pragma omp parallel
{ float B;
#pragma omp for
 for(int i=0; i<niters; i++){
   B = big_job(i);
#pragma omp critical (RES_lock)
   consum (B, RES);
 }
}
```

Good Practice – Name all critical sections

OpenMP* Reduction Clause

```
reduction (op : list)
```

- The variables in "*list*" must be shared in the enclosing parallel region
- Inside parallel or work-sharing construct:
  - A PRIVATE copy of each list variable is created and initialized depending on the "op"
  - These copies are updated locally by threads
  - At end of construct, local copies are combined through "op" into a single value and combined with the value in the original SHARED variable

Reduction Example

```
#include <stdio.h>
#include "omp.h"
int main(int argc, char *argv[])
{
 int count = 10;
 #pragma omp parallel reduction (+: count)
   { int num;
     num = omp_get_thread_num();
     count++;
     printf("count is equal : %d on thread %d \n", count, num);
   }
 printf("count at the end: %d\n", count);
}
```

```
count is equal : 1 on thread 0
count is equal : 1 on thread 1
count at the end: 12
```

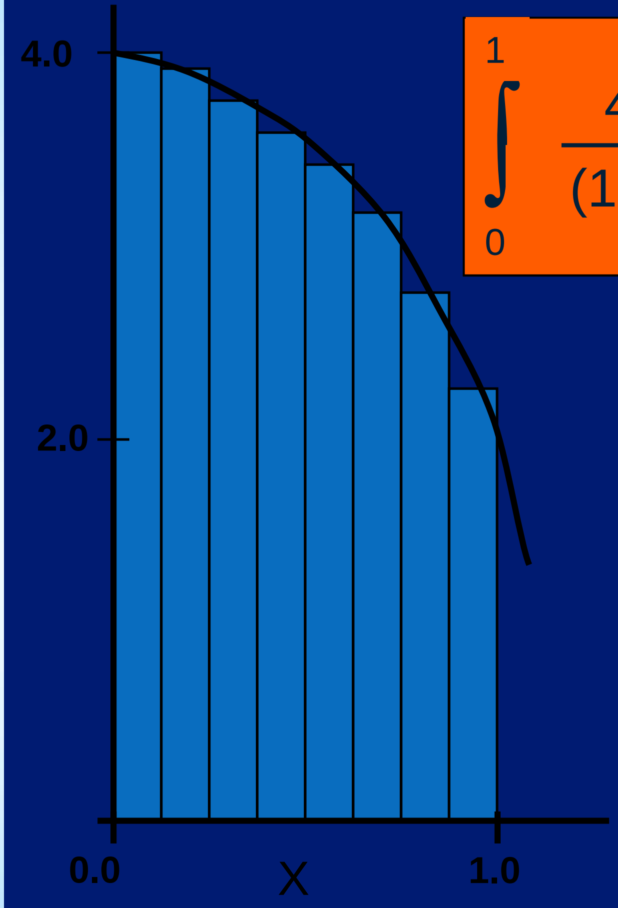
C/C++ Reduction Operations

- A range of associative operands can be used with reduction
- Initial values are the ones that make sense mathematically

Operand	Initial Value
+	0
*	1
-	0
^	0

Operand	Initial Value
&	~0
|	0
&&	1
||	0

Numerical Integration Example



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

```
static long num_steps=100000;
double step, pi;

void main()
{  int i;
   double x, 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;
   printf("Pi = %f\n",pi);
}
```

Activity 5 - Computing Pi

```
static long num_steps=100000;
double step, pi;

void main()
{  int i;
   double x, 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;
   printf(“Pi = %f\n”,pi);
}
```

- Parallelize the numerical integration code using OpenMP
- What variables can be shared?
- What variables need to be private?
- What variables should be set up for reductions?

Single Construct

- Denotes block of code to be executed by only one thread
  - Implementation defined
- Implicit barrier at end

```
#pragma omp parallel
{
  DoManyThings();
#pragma omp single
  {
   ExchangeBoundaries();
  }  // threads wait here for single
  DoManyMoreThings();
}
```

Master Construct

- Denotes block of code to be executed only by the master thread
- No implicit barrier at end

```
#pragma omp parallel
{
  DoManyThings();
#pragma omp master
  {          // if not master skip to next stmt
   ExchangeBoundaries();
  }
  DoManyMoreThings();
}
```

Implicit Barriers

- Several OpenMP* constructs have implicit barriers
  - Parallel – necessary barrier – cannot be removed
  - for
  - single
- Unnecessary barriers hurt performance and can be removed with the nowait clause
  - The nowait clause is applicable to:
    - For clause
    - Single clause

Nowait Clause

```
#pragma omp for nowait
  for(...)
    {...};
```

```
#pragma single nowait
{ [...] }
```

- Use when threads unnecessarily wait between independent computations

```
#pragma omp for schedule(dynamic,1) nowait
 for(int i=0; i<n; i++)
   a[i] = bigFunc1(i);

#pragma omp for schedule(dynamic,1)
 for(int j=0; j<m; j++)
   b[j] = bigFunc2(j);
```

Barrier Construct

- Explicit barrier synchronization
- Each thread waits until all threads arrive

```
#pragma omp parallel shared (A, B, C)
{
        DoSomeWork(A,B);
        printf("Processed A into B\n");
#pragma omp barrier
        DoSomeWork(B,C);
        printf("Processed B into C\n");
}
```

Atomic Construct

- Special case of a critical section
- Applies only to simple update of memory location

```
#pragma omp parallel for shared(x, y, index, n)
  for (i = 0; i < n; i++) {
    #pragma omp atomic
     x[index[i]] += work1(i);
    y[i] += work2(i);
  }
```

Agenda

- What is OpenMP?
- Parallel regions
- Data-Sharing Attribute Clauses
- Worksharing
- OpenMP 3.0 Tasks
- Synchronization
- Runtime functions/environment variables
- Optional Advanced topics

Environment Variables

- Set the default number of threads

  `OMP_NUM_THREADS` *integer*

- Set the default scheduling protocol

  `OMP_SCHEDULE "schedule[, chunk_size]"`

- Enable dynamic thread adjustment

  `OMP_DYNAMIC [TRUE|FALSE]`

- Enable nested parallelism

  `OMP_NESTED [TRUE|FALSE]`

20+ Library Routines

- Runtime environment routines:
  - Modify/check the number of threads

    `omp_[set|get]_num_threads()`

    `omp_get_thread_num()`

In this course, focuses on the directives only approach to OpenMP – which makes incremental parallelism easy

  - Explicit locks

    `omp_[set|unset]_lock()`

  - And many more...

Library Routines

- To fix the number of threads used in a program
  - Set the number of threads
  - Then save the number returned

```
#include <omp.h>

void main ()
{
   int num_threads;
   omp_set_num_threads (omp_num_procs ());

   #pragma omp parallel
   {
      int id = omp_get_thread_num ();

      #pragma omp single
      num_threads = omp_get_num_threads ();

      do_lots_of_stuff (id);
   }
}
```

Request as many threads as you have processors.

Protect this operation because memory stores are not atomic

Agenda

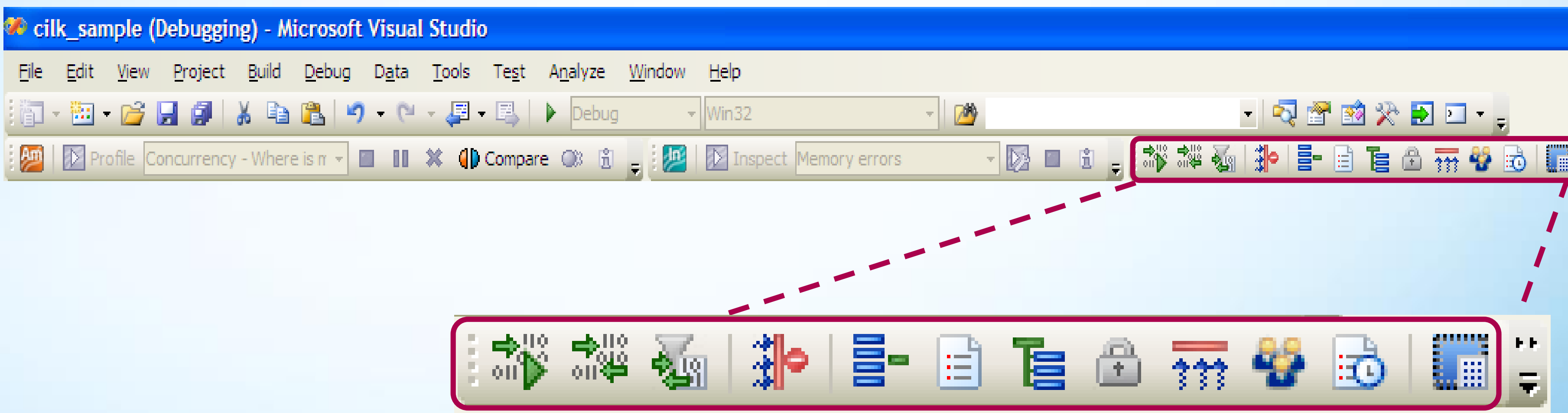
- What is OpenMP?
- Parallel regions
- Data-Sharing Attribute Clauses
- Worksharing
- OpenMP 3.0 Tasks
- Synchronization
- Runtime functions/environment variables
- Optional Advanced topics
  - Intel® Parallel Debugger Extension
  - Advanced Concepts

Key Features

- Thread Shared Data Event Detection
  - Break on Thread Shared Data Access (read/write)
- Re-entrant Function Detection
- SIMD SSE Registers Window
- Enhanced OpenMP* Support
  - Serialize OpenMP threaded application execution on the fly
  - Insight into thread groups, barriers, locks, wait lists etc.

Debugger Extensions in Microsoft* VS

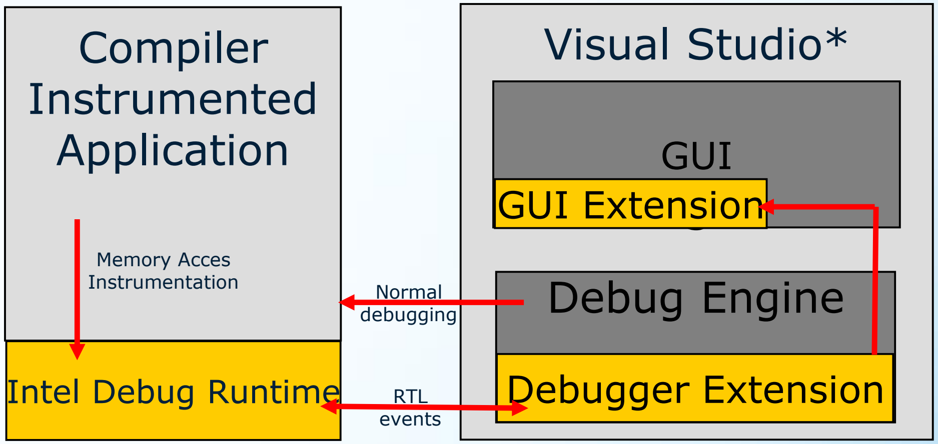
The Intel® Debugger Extensions is a plug-in to Visual Studio 2005/2008/2010 and add features to the Microsoft* debugger.



The parallel debugger extensions has similar buttons as the Linux version of IDB. The functionality is identical.

Shared Data Events Detection

- Shared data access is a major problem in multi- threaded applications
  - Can cause hard to diagnose intermittent program failure
  - Tool support is required for detection



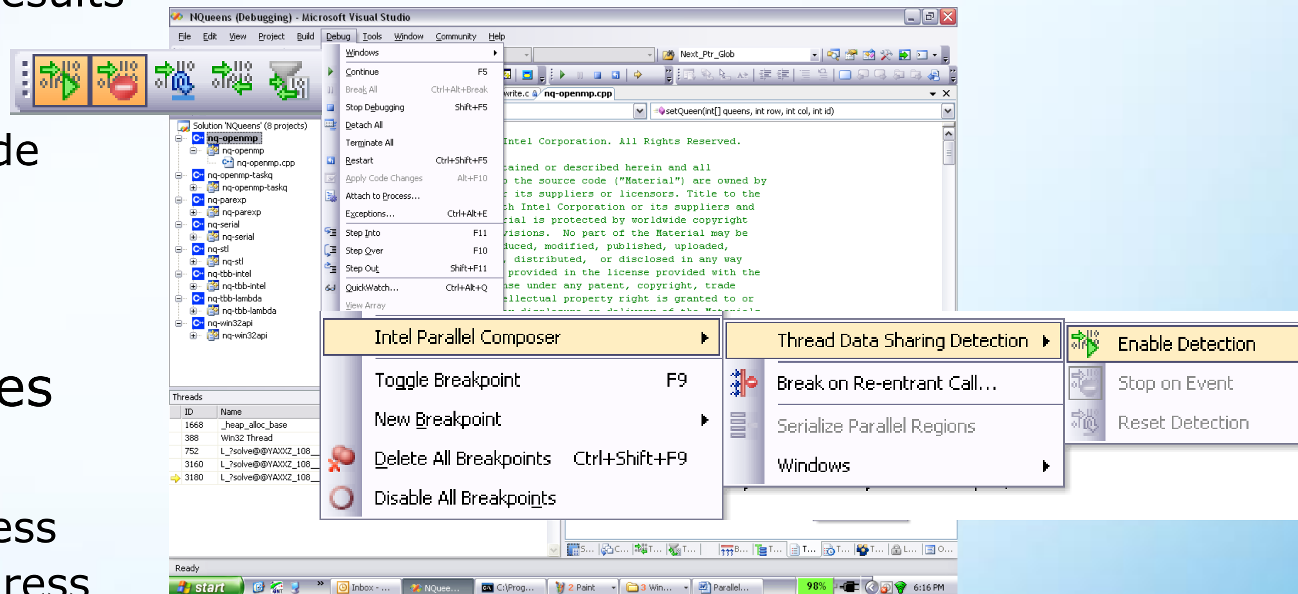
Technology built on:

- Code & debug info Instrumentation by Intel compiler (/Qopenmp /debug:parallel)
- Debug runtime library that collects data access traces and triggers debugger tool (libiomp5md.dll, pdbx.dll)
- Debugger Extension in Visual Studio Debug Engine collects and exports information
- GUI integration provides parallelism views and user interactivity

Shared Data Events Detection – contd.

Data sharing detection is part of overall debug process

- Breakpoint model (stop on detection)
- GUI extensions show results & link to source



- Filter capabilities to hide false positives

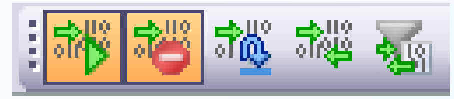
New powerful data breakpoint types

- Stop when 2nd thread accesses specific address
- Stop on read from address

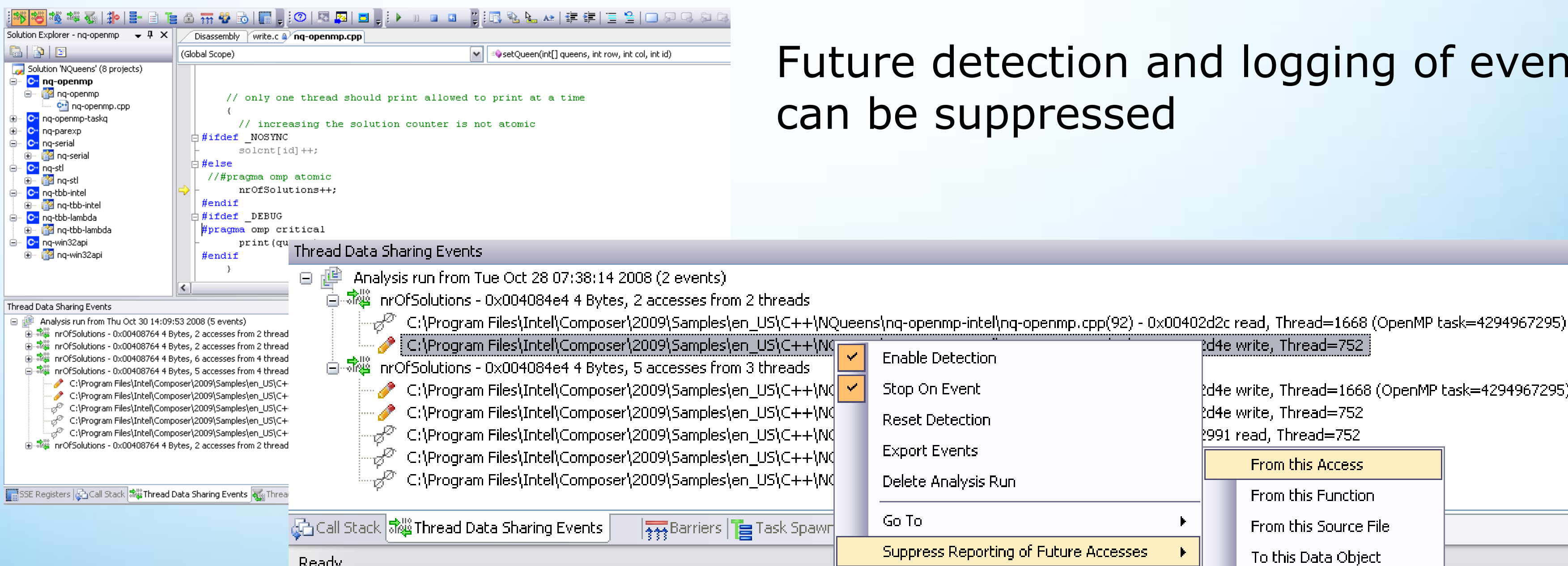
Key User Benefit:
A simplified feature to detect shared data accesses from multiple threads

Shared Data Events Detection - Usage

Enabling Shared Data Events Detection automatically enables "Stop On Event"



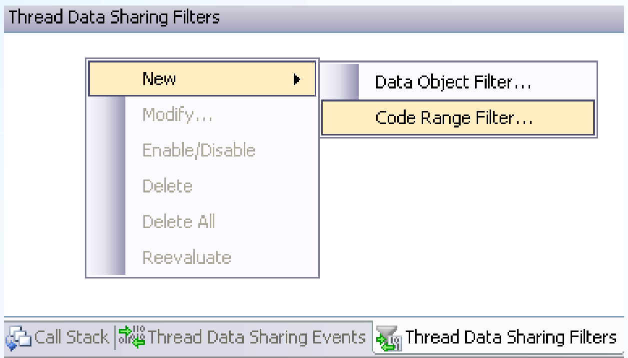
Future detection and logging of events can be suppressed



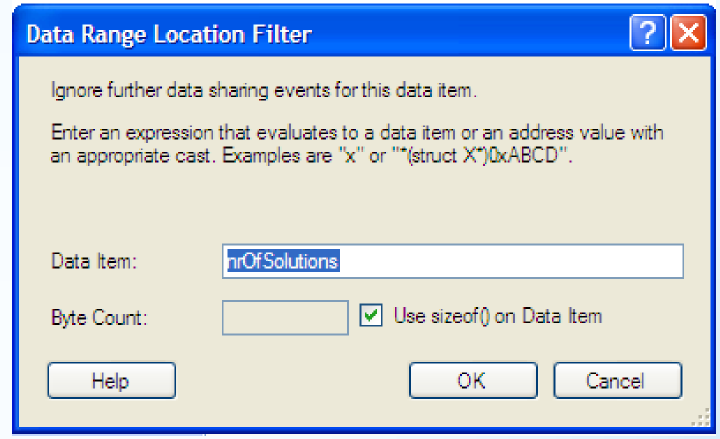
Shared Data Events Detection - Filtering

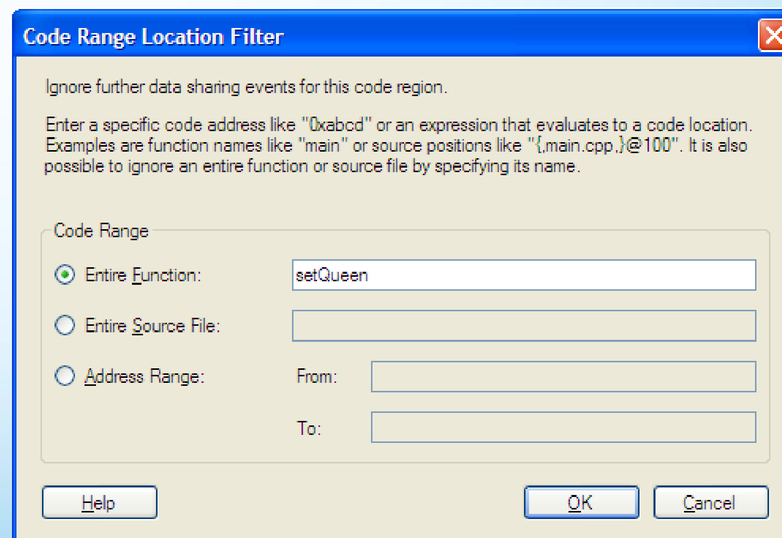
Data sharing detection is selective

- Data Filter
  - Specific data items and variables can be excluded



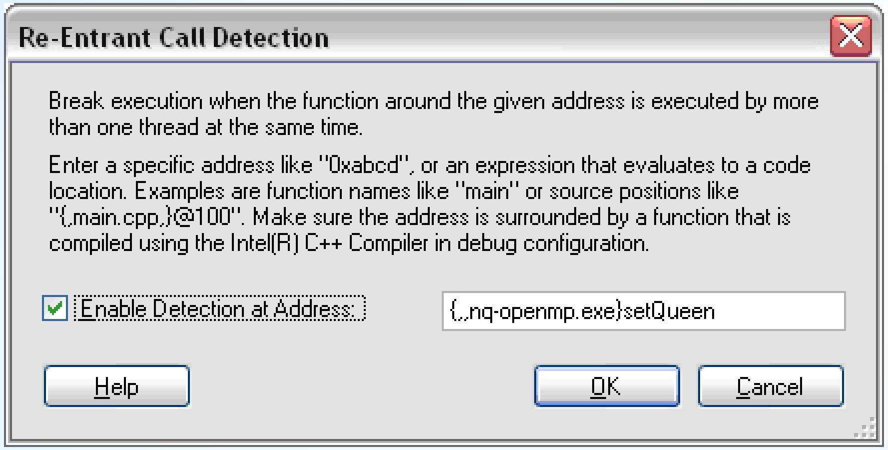
- Code Filter
  - Functions can be excluded
  - Source files can be excluded
  - Address ranges can be excluded

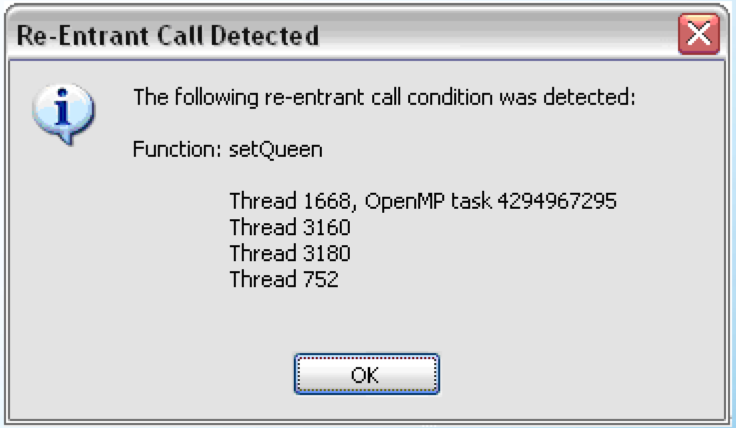




Re-Entrant Call Detection

Automatically halts execution when a function is executed by more than one thread at any given point in time.

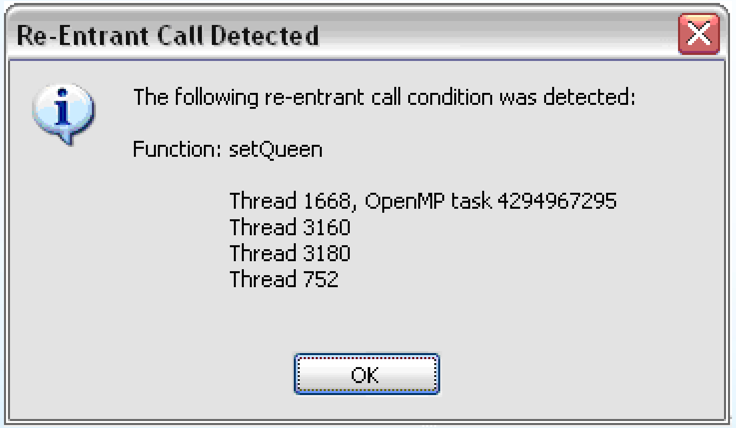


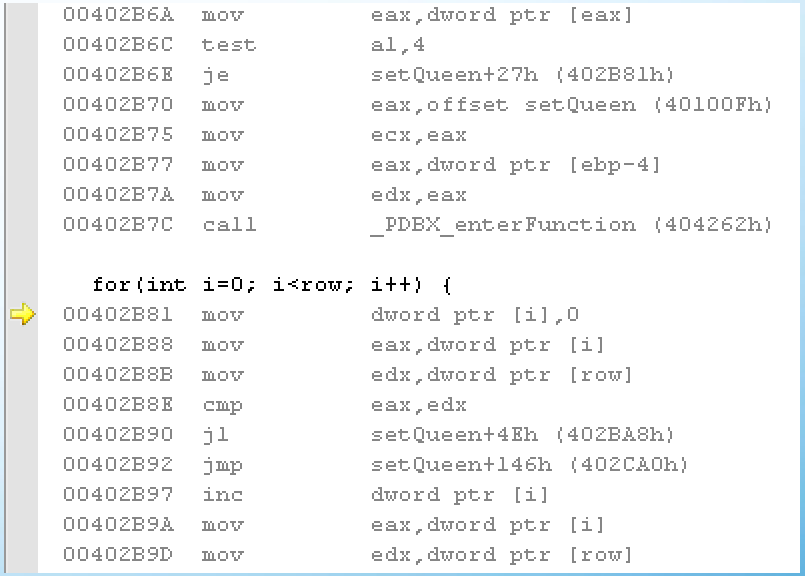


Allows to identify reentrancy requirements/problems for multi-threaded applications

Re-Entrant Call Detection

- After execution is halted and user clicks ok the program counter will point exactly to where the problem occurred.





```
00402B6A  mov          eax,dword ptr [eax]
00402B6C  test         al,4
00402B6E  je           setQueen+27h (402B81h)
00402B70  mov          eax,offset setQueen (40100Fh)
00402B75  mov          ecx,eax
00402B77  mov          eax,dword ptr [ebp-4]
00402B7A  mov          edx,eax
00402B7C  call         _PDBX_enterFunction (404262h)

  for(int i=0; i<row; i++) {
00402B81  mov          dword ptr [i],0
00402B88  mov          eax,dword ptr [i]
00402B8B  mov          edx,dword ptr [row]
00402B8E  cmp          eax,edx
00402B90  jl           setQueen+4Eh (402BA8h)
00402B92  jmp          setQueen+146h (402CA0h)
00402B97  inc          dword ptr [i]
00402B9A  mov          eax,dword ptr [i]
00402B9D  mov          edx,dword ptr [row]
```

SIMD SSE Debugging Window

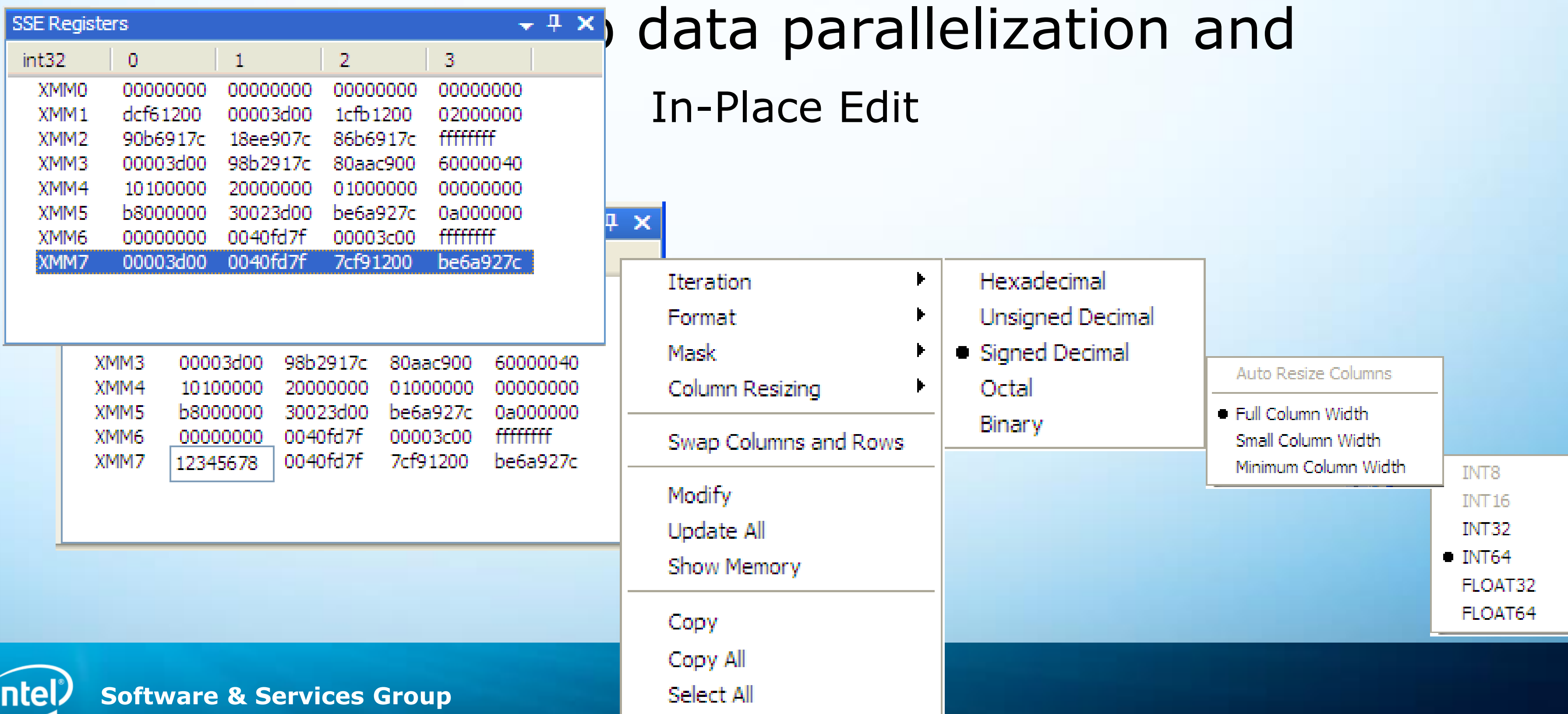
SSE Registers Window

SSE Registers display of variables used for SIMD operations

Free Formatting for flexible representation of data

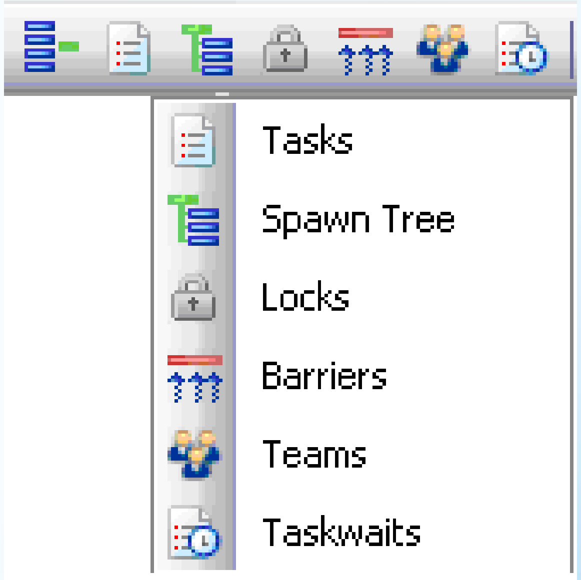
…o data parallelization and

In-Place Edit



Enhanced OpenMP* Debugging Support

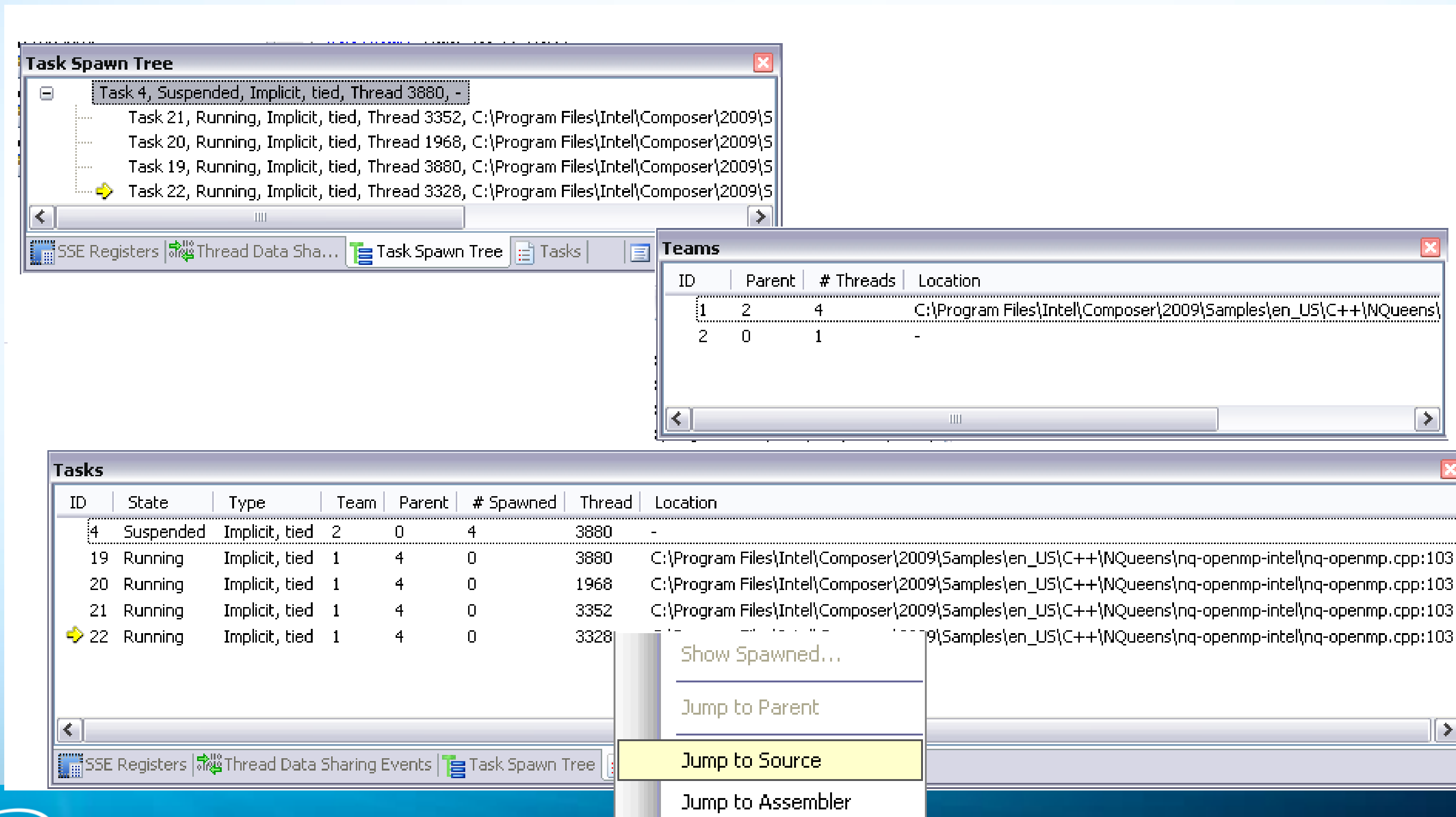
- Dedicated OpenMP runtime object - information Windows
  - OpenMP Task and Spawn Tree lists
  - Barrier and Lock information
  - Task Wait lists
  - Thread Team worker lists
- Serialize Parallel Regions
  - Change number of parallel threads dynamically during runtime to 1 or N (all)
  - Verify code correctness for serial execution vs. Parallel execution
  - Identify whether a runtime issue is really parallelism related



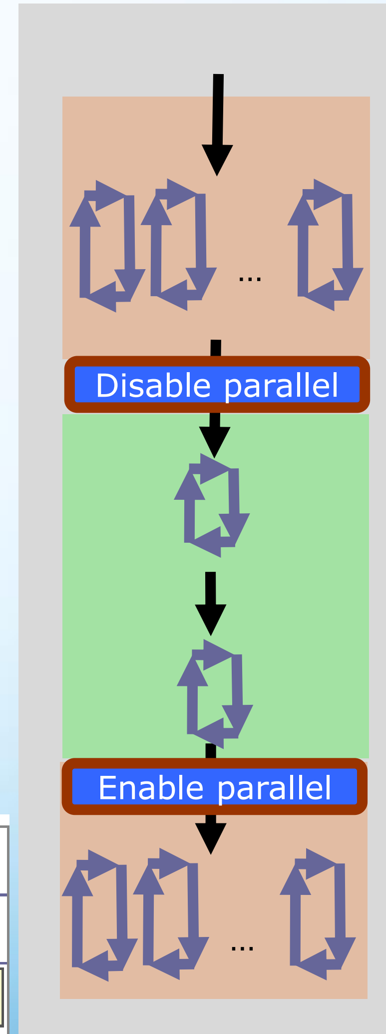
User benefit:
Detailed execution state information for OpenMP applications (deadlock detection). Influences execution behavior without recompile!

OpenMP* Information Windows

- Tasks, Spawn Trees, Teams, Barriers, Locks
- (Task ID reporting incorrect)



Serialize Parallel Regions



Problem:

Parallel loop computes a wrong result. Is it a concurrency or algorithm issue ?

Parallel Debug Support

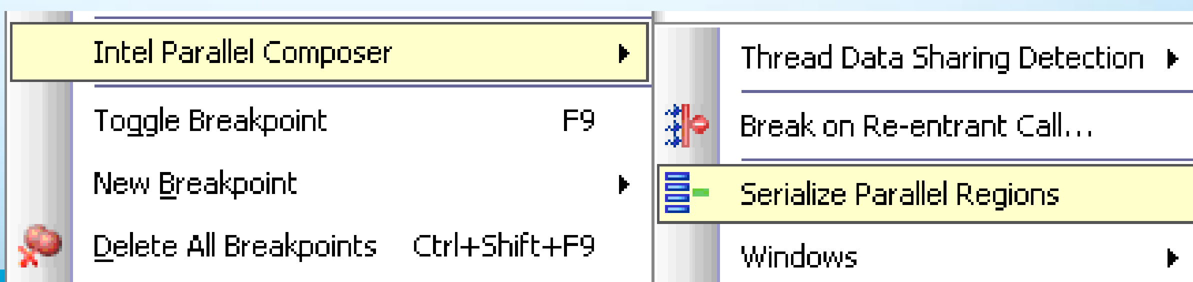
Runtime access to the OpenMP num_thread property

Set to 1 for serial execution of next parallel block

User Benefit

Verification of a algorithm “on-the-fly” without slowing down the entire application to serial execution

On demand serial debugging without recompile/restart

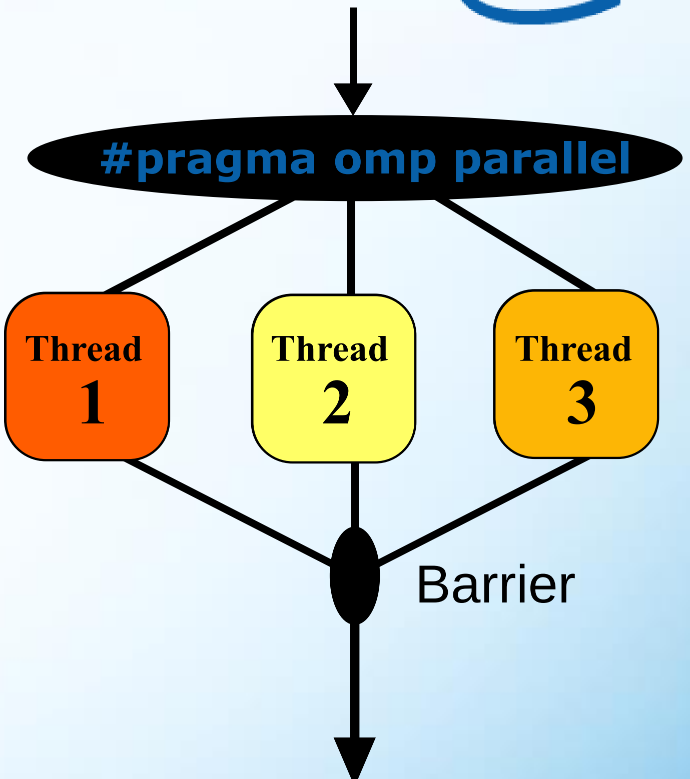


Agenda

- What is OpenMP?
- Parallel regions
- Data-Sharing Attribute Clauses
- Worksharing
- OpenMP 3.0 Tasks
- Synchronization
- Runtime functions/environment variables
- Optional Advanced topics
  - Intel® Parallel Debugger Extension
  - Advanced Concepts

Parallel Construct – Implicit Task View

- Tasks are created in OpenMP even without an explicit task directive.
- Lets look at how tasks are created implicitly for the code snippet below
  - Thread encountering parallel construct packages up a set of *implicit* tasks
  - Team of threads is created.
  - Each thread in team is assigned to one of the tasks (and *tied* to it).
  - Barrier holds original master thread until all implicit tasks are finished.




```
#pragma omp
par
  {
     int mydata;
     code...
  }
```

Task Construct

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

where *clause* **can be one of:**

if (*expression***)**
untied
shared (*list***)**
private (*list***)**
firstprivate (*list***)**
default(shared | none)

Tied & Untied Tasks

- Tied Tasks:
  - A tied task gets a thread assigned to it at its first execution and the same thread services the task for its lifetime
  - A thread executing a tied task, can be suspended, and sent off to execute some other task, but eventually, the same thread will return to resume execution of its original tied task
  - Tasks are tied unless explicitly declared untied

- Untied Tasks:
  - An united task has no long term association with any given thread. Any thread not otherwise occupied is free to execute an untied task. The thread assigned to execute an untied task may only change at a "task scheduling point".
  - An untied task is created by appending “untied” to the task clause
  - Example: #pragma omp task untied

Task switching

- **task switching** The act of a *thread* switching from the execution of one *task* to another *task.*
- The purpose of task switching is distribute threads among the unassigned tasks in the team to avoid piling up long queues of unassigned tasks
- Task switching, for tied tasks, can only occur at task scheduling points located within the following constructs
  - encountered **task** *constructs*
  - encountered **taskwait** *constructs*
  - encountered **barrier** *directives*
  - implicit **barrier** *regions*
  - at the end of the *tied task region*
- Untied tasks have implementation dependent scheduling points

Task switching example

The thread executing the "for loop" , AKA the generating task, generates many tasks in a short time so...

The SINGLE generating task will have to suspend for a while when "task pool" fills up

- Task switching is invoked to start draining the "pool"
- When "pool" is sufficiently drained – then the single task can being generating more tasks again

```
int exp;  //exp either T or F;
#pragma omp single
{
   for (i=0; i<ONEZILLION; i++)
    #pragma omp task
       process(item[i]);
   }
}
```

Optional foil - OpenMP* API

- Get the thread number within a team

```
int omp_get_thread_num(void);
```

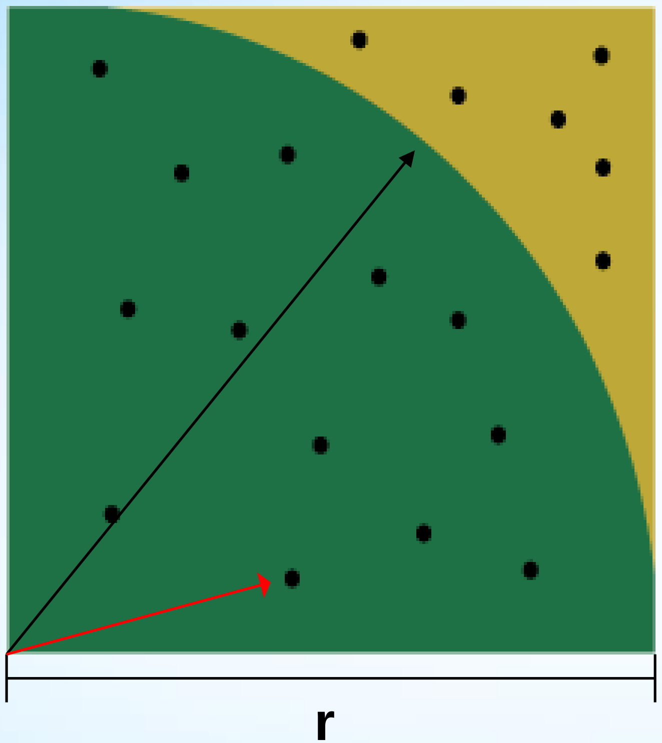
- Get the number of threads in a team

```
int omp_get_num_threads(void);
```

- Usually not needed for OpenMP codes
  - Can lead to code not being serially consistent
  - Does have specific uses (debugging)
  - Must include a header file

```
#include <omp.h>
```

Optional foil - Monte Carlo Pi



$$\frac{\#\text{ of darts hitting circle}}{\#\text{ of darts in square}} = \frac{\frac{1}{4}\pi r^2}{r^2}$$

$$\pi = 4\,\frac{\#\text{ of darts hitting circle}}{\#\text{ of darts in square}}$$

```
loop 1 to MAX
     x.coor=(random#)
     y.coor=(random#)
     dist=sqrt(x^2 + y^2)
     if (dist <= 1)
      hits=hits+1
pi = 4 * hits/MAX
```

Optional foil - Making Monte Carlo's Parallel

```
hits = 0
call SEED48(1)
DO I = 1, max
    x = DRAND48()
    y = DRAND48()
    IF (SQRT(x*x + y*y) .LT. 1) THEN
        hits = hits+1
    ENDIF
END DO
pi = REAL(hits)/REAL(max) * 4.0
```

What is the challenge here?

Optional Activity 6: Computing Pi

- Use the Intel® Math Kernel Library (Intel® MKL) VSL:
  - Intel MKL's VSL (Vector Statistics Libraries)
  - VSL creates an array, rather than a single random number
  - VSL can have multiple seeds (one for each thread)
- Objective:
  - Use basic OpenMP* syntax to make Pi parallel
  - Choose the best code to divide the task up
  - Categorize properly all variables

OpenMP* 4.0 Specification

Released July 2013

- http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
- A document of examples is expected to release soon

Changes from 3.1 to 4.0 (Appendix E.1):

- *Places and thread affinity: 2.5.2, 4.5*
- *SIMD extensions: 2.8*
- *Device / Accelerator: 2.9*
- *Combined simd, parallel, target, teams, distribute constructs: 2.10.4 - 2.10.13*
- *Taskgroup and dependent tasks: 2.12.5, 2.11*
- *Sequentially consistent atomics: 2.12.6*
- *Error handling: 2.13*
- *User-defined reductions: 2.15*
- *Fortran 2003 support*

Intel® C/C++ and Fortran Compilers

Main SIMD and Offloading features are supported in 14.0 compilers

- *Places and thread affinity (14.0)*
- *Main features of SIMD extensions (14.0)*
- *Main features of Device / Accelerator extensions (14.0)*
- *Combined simd, parallel, target, teams, distribute constructs (15.0)*
- *Taskgroup and dependent tasks (15.0)*
- *Sequentially consistent atomics (14.0)*
- *Error handling (15.0)*
- *User-defined reductions (TBD)*
- *Fortran 2003 support (15.0)*

SIMD Extensions for Loops

#pragma omp simd *[clause[[,] clause] …] new-line*

for-loops

where the *clause* is one the following:

safelen(*length*)

linear(*list[:linear-step]*)

aligned(*list[:alignment]*)

private(*list*)

lastprivate(*list*)

reduction(*operator:list*)

collapse(*n*)

!$omp simd *[clause[[,] clause …]*

do-loops

[**!$omp end simd**]

A SIMD-enabled Function Example

Callee Function

```
#pragma omp declare simd uniform(a) linear(i:1)
simdlen(4)
void foo(int *a, int i){
        std::cout<<a[i]<<"\n";
}
```

Caller Loop

```
#pragma omp simd safelen(4)
for(int i = 0; i < n; i++)
        foo(a, i);
#pragma omp simd safelen(4)
for(int i = 0; i < n; i++){
        k = b[i];  // k is not linear
        foo(a, k);
}
```

Vector Report

testmain.cc(14): (col. 13) remark: SIMD LOOP WAS VECTORIZED

testmain.cc(21): (col. 9) remark: No suitable vector variant of function '_Z3fooPii' found

testmain.cc(18): (col. 1) remark: SIMD LOOP WAS VECTORIZED

header.cc(3): (col. 24) remark: FUNCTION WAS VECTORIZED

Target Extensions for Coprocessors

C/C++

#pragma omp target *[clause[[,] clause],...] new-line*

Clauses: device(*scalar-integer-expression***)**
map(alloc | to | from | tofrom: *list***)**
if(*scalar-expr***)**

#pragma omp target data *[clause[[,] clause],...] new-line*
structured-block
Clauses: device(*scalar-integer-expression***)**
map(alloc | to | from | tofrom: *list***)**
if(*scalar-expr***)**

#pragma omp target update *[clause[[,] clause],...] new-line*

Clauses: to(*list* **)**
from(*list* **)**
device(*integer-expression* **)**
if(*scalar-expression* **)**

#pragma omp declare target *new-line*
[function-variable-definition-or-declaration]
#pragma omp end declare target *new-line*

Teams and Distribute Constructs

C/C++ syntax

#pragma omp teams *[clause[[,] clause],...] new-line*
structured-block

Clauses: num_teams(*integer-expression* **)**
thread_limit(*integer-expression* **)**
default(shared | none)
private(*list* **)**
firstprivate(*list* **)**
shared(*list* **)**
reduction(*operator* **:** *list* **)**

If specified, a **teams** construct must be contained within a **target** construct. That **target** construct must contain no statements or directives outside of the **teams** construct.
distribute, **parallel**, **parallel loop**, **parallel sections**, and **parallel workshare** are the only OpenMP constructs that can be closely nested in the **teams** region.

#pragma omp distribute *[clause[[,] clause],...] new-line*
for-loops
Clauses: private(*list* **)**
firstprivate(*list* **)**
collapse(*n* **)**
dist_schedule(*kind[, chunk_size]* **)**

A **distribute** construct must be closely nested in a **teams** region.

Target Extensions for Coprocessors

Fortran Syntax

!$omp target *[clause[[,] clause],...] new-line*
structured-block
!$omp end target

Clauses: **device(***scalar-integer-expression***)**
map(alloc | to | from | tofrom: *list***)**
if(*scalar-expr***)**

!$omp target data *[clause[[,] clause],...] new-line*
structured-block
!$omp end target data

Clauses: **device(***scalar-integer-expression***)**
map(alloc | to | from | tofrom: *list***)**
if(*scalar-expr***)**

!$omp target update *[clause[[,] clause],...] new-line*

Clauses: **to(** *list* **)**
from(*list* **)**
device(*integer-expression* **)**
if(*scalar-expression* **)**

!$omp declare target [(list)] *new-line*
list : *subroutine-function-variable*

Teams and Distribute Constructs

Fortran Syntax

!$omp teams *[clause[[,] clause],...] new-line*
structured-block
!$omp end teams

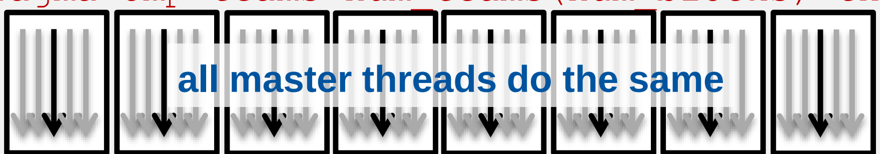
Clauses: **num_teams(** *integer-expression* **)**
thread_limit(*integer-expression* **)**
default(shared | none)
private(*list* **)**
firstprivate(*list* **)**
shared(*list* **)**
reduction(*operator* **:** *list* **)**

!$omp distribute *[clause[[,] clause],...] new-line*
do-loops
*[***!$omp end distribute]**
Clauses: **private(** *list* **)**
firstprivate(*list* **)**
collapse(*n* **)**
dist_schedule(*kind[, chunk_size]* **)**

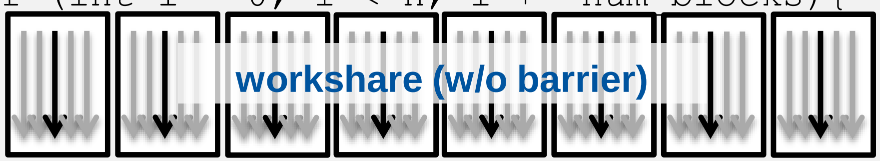
A **distribute** construct must be closely nested in a **teams** region.

Offloading: SAXPY Example

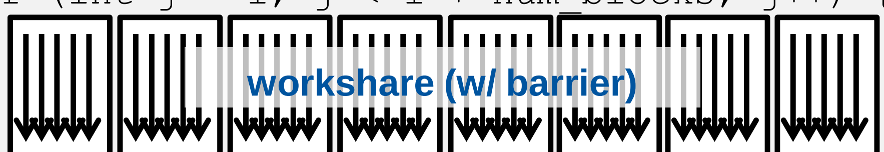
```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars n, a, b & initialize x, y

  // Run SAXPY TWICE
#pragma omp target data map(to:x[0:n])
{
#pragma omp target map(tofrom:y)
#pragma omp teams num_teams(num_blocks) thread_limit(bsize)
```




```
#pragma omp distribute
  for (int i = 0; i < n; i += num_blocks){
```




```
#pragma omp parallel for
    for (int j = i; j < i + num_blocks; j++) {
```




```
      y[j] = a*x[j] + y[j];
  } }
} free(x); free(y); return 0; }
```

Optimization Notice

Optimization Notice

Intel compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel Compiler User and Reference Guides” under “Compiler Options." Many library routines that are part of Intel compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel compilers, associated libraries and associated development tools 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 Intel Streaming SIMD Extensions 2 (Intel SSE2), Intel Streaming SIMD Extensions 3 (Intel SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel 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.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20110228

Legal Disclaimer

http://intel.com/software/products