Introduction to OpenMP

www.openmp.org

Motivation

- Parallel machines are abundant
  - Servers are 2-8 way SMPs and more
  - Upcoming processors are multicore – parallel programming is beneficial and actually necessary to get high performance
- Multithreading is the natural programming model for SMP
  - All processors share the same memory
  - Threads in a process see the same address space
  - Lots of shared-memory algorithms defined
- Multithreading is (correctly) perceived to be hard!
  - Lots of expertise necessary
  - Deadlocks and race conditions
  - Non-deterministic behavior makes it hard to debug

Motivation 2

- Parallelize the following code using threads:

```
for (i=0; i<n; i++) {
    sum = sum+sqrt(sin(data[i]));
}
```

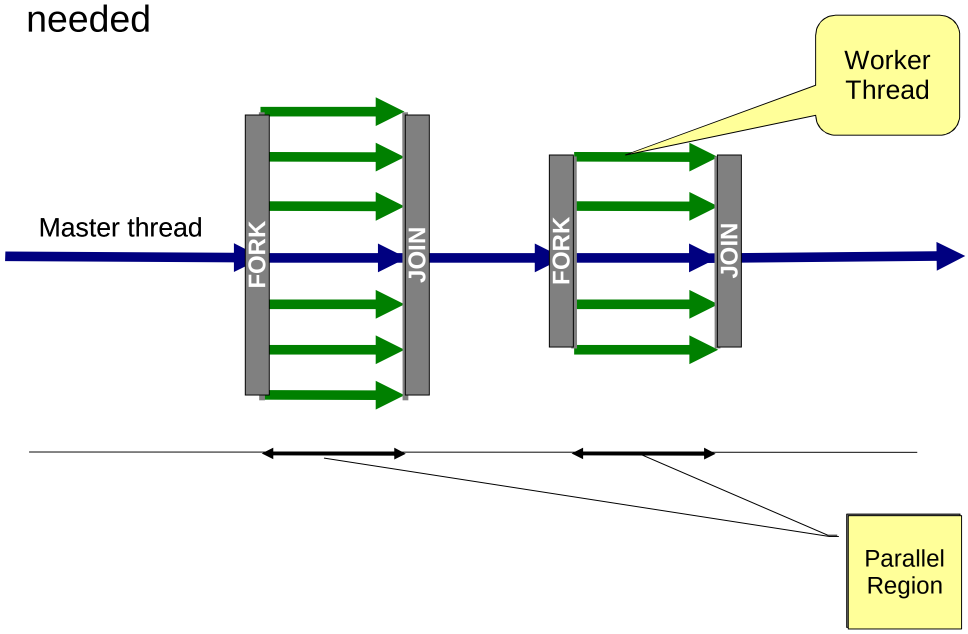
- A lot of work to do a simple thing
- Different threading APIs:
  - Windows: CreateThread
  - UNIX: pthread_create
- Problems with the code:
  - Need mutex to protect the accesses to sum
  - Different code for serial and parallel version
  - No built-in tuning (# of processors)

Motivation: OpenMP

- A language extension that introduces parallelization constructs into the language
- Parallelization is orthogonal to the functionality
  - If the compiler does not recognize the OpenMP directives, the code remains functional (albeit single-threaded)
- Based on shared-memory multithreaded programming
- Includes constructs for parallel programming: critical sections, atomic access, variable privatization, barriers etc.
- Industry standard
  - Supported by Intel, Microsoft, Sun, IBM, HP, etc.
    Some behavior is implementation-dependent
  - Intel compiler available for Windows and Linux

OpenMP execution model

Fork and Join: Master thread spawns a team of threads as needed



OpenMP memory model

- Shared memory model
  - Threads communicate by accessing shared variables
- The sharing is defined syntactically
  - Any variable that is seen by two or more threads is shared
  - Any variable that is seen by one thread only is private
- Race conditions possible
  - Use synchronization to protect from conflicts
  - Change how data is stored to minimize the synchronization

OpenMP syntax

- Most of the constructs of OpenMP are pragmas
  - #pragma omp construct [clause [clause] …]
    (FORTRAN: !$OMP, not covered here)
  - An OpenMP construct applies to a *structural block* (one entry point, one exit point)
- Categories of OpenMP constructs
  - Parallel regions
  - Work sharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
- In addition:
  - Several omp_<something> function calls
  - Several OMP_<something> environment variables

OpenMP: extents

- Static (lexical) extent
  Defines all the locations immediately visible in the lexical scope of a statement
- Dynamic extent
  Defines all the locations reachable dynamically from a statement
  - For example, the code of functions called from a parallelized region is in the region's dynamic extent
  - Some OpenMP directives may need to appear within the dynamic extent, and not directly in the parallelized code (think of a called function that needs to perform a critical section).
- Directives that appear in the dynamic extent (without enclosing lexical extent) are called *orphaned*.

OpenMP: Parallel Regions

```
double D[1000];
#pragma omp parallel
{
  int i; double sum = 0;
  for (i=0; i<1000; i++) sum += D[i];
  printf("Thread %d computes %f\n",
      omp_thread_num(), sum);
}
```

- Executes the same code several times (as many as there are threads)
  - How many threads do we have?
    omp_set_num_threads(n)
  - What is the use of repeating the same work several times in parallel?
    Can use omp_thread_num() to distribute the work between threads.
- D is shared between the threads, i and sum are private

OpenMP: Work Sharing Constructs 1

```
answer1 = long_computation_1();
answer2 = long_computation_2();
if (answer1 != answer2) { … }
```

- How to parallelize?
  - These are just two independent computations!

```
#pragma omp sections
{
  #pragma omp section
  answer1 = long_computation_1();
  #pragma omp section
  answer2 = long_computation_2();
}
if (answer1 != answer2) { … }
```

OpenMP: Work Sharing Constructs 2

Sequential code

```
for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
```

(Semi) manual parallelization

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  int Nthr = omp_get_num_threads();
  int istart = id*N/Nthr, iend = (id+1)*N/Nthr;
  for (int i=istart; i<iend; i++) { a[i]=b[i]+c[i]; }
}
```

Automatic parallelization of the for loop

```
#pragma omp parallel
#pragma omp for schedule(static)
{
  for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }
}
```

Notes on #parallel for

- Only simple kinds of for loops are supported
  - One signed integer variable in the loop.
  - Initialization: var=init
  - Comparison: var op last, op: <, >, <=, >=
  - Increment: var++, var--, var+=incr, var-=incr, etc.
  - All of init, last, incr must be loop invariant

- Can combine the parallel and work sharing directives:
  `#pragma omp parallel for …`

Problems of #parallel for

- Load balancing
  - If all the iterations execute at the same speed, the processors are used optimally
  - If some iterations are faster than others, some processors may get idle, reducing the speedup
  - We don't always know the distribution of work, may need to re-distribute dynamically
- Granularity
  - Thread creation and synchronization takes time
  - Assigning work to threads on per-iteration resolution may take more time than the execution itself!
  - Need to coalesce the work to coarse chunks to overcome the threading overhead
- Trade-off between load balancing and granularity!

Schedule: controlling work distribution

- schedule(static [, chunksize])
  - Default: chunks of approximately equivalent size, one to each thread
  - If more chunks than threads: assigned in round-robin to the threads
  - What if we want to use chunks of different sizes?
- schedule(dynamic [, chunksize])
  - Threads receive chunk assignments dynamically
  - Default chunk size = 1
- schedule(guided [, chunksize])
  - Start with large chunks
  - Threads receive chunks dynamically. Chunk size reduces exponentially, down to chunksize

Controlling Granularity

- #pragma omp parallel if (expression)
  - Can be used to disable parallelization in some cases (when the input is determined to be too small to be beneficially multithreaded)
- #pragma omp num_threads (expression)
  - Control the number of threads used for this parallel region

OpenMP: Data Environment

- Shared Memory programming model
  - Most variables (including locals) are shared by default

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

  - Global variables are shared
- Some variables can be private
  - Automatic variables inside the statement block
  - Automatic variables in the called functions
  - Variables can be explicitly declared as private.
    In that case, a local copy is created for each thread

Overriding storage attributes

- private:
  - A copy of the variable is created for each thread
  - There is no connection between the original variable and the private copies
  - Can achieve the same using variables inside { }

```
int i;
#pragma omp parallel for private(i)
for (i=0; i<n; i++) { … }
```

- firstprivate:
  - Same, but the initial value of the variable is copied from the main copy
- lastprivate:
  - Same, but the last value of the variable is copied to the main copy

```
int idx=1;
int x = 10;
#pragma omp parallel for \
  firsprivate(x) lastprivate(idx)
for (i=0; i<n; i++) {
   if (data[i]==x) idx = i;
}
```

Threadprivate

- Similar to private, but defined per variable
  - Declaration immediately after variable definition. Must be visible in all translation units.
  - Persistent between parallel sections
  - Can be initialized from the master's copy with #pragma omp copyin
  - More efficient than private, but a global variable!
- Example:

```
int data[100];
#pragma omp threadprivate(data)
…
#pragma omp parallel for copyin(data)
for (……)
```

Reduction

```
for (j=0; j<N; j++) {
  sum = sum+a[j]*b[j];
}
```

- How to parallelize this code?
  - sum is not private, but accessing it atomically is too expensive
  - Have a private copy of sum in each thread, then add them up
- Use the reduction clause!
  #pragma omp parallel for reduction(+: sum)
  - Any associative operator can be used: +, -, ||, |, *, etc.
  - The private value is initialized automatically (to 0, 1, ~0 …)

OpenMP Synchronization

```
X = 0;
#pragma omp parallel
X = X+1;
```

- What should the result be (assuming 2 threads)?
  - 2 is the expected answer
  - But can be 1 with unfortunate interleaving
- OpenMP assumes that the programmer knows what (s)he is doing
  - Regions of code that are marked to run in parallel are independent
  - If access collisions are possible, it is the programmer's responsibility to insert protection

Synchronization Mechanisms

- Many of the existing mechanisms for shared programming
  - Single/Master execution
  - Critical sections, Atomic updates
  - Ordered
  - Barriers
  - Nowait (turn synchronization off!)
  - Flush (memory subsystem synchronization)
  - Reduction (already seen)

Single/Master

- #pragma omp single
  - Only one of the threads will execute the following block of code
  - The rest will wait for it to complete
  - Good for non-thread-safe regions of code (such as I/O)
  - Must be used in a parallel region
  - Applicable to parallel for sections
- #pragma omp master
  - The following block of code will be executed by the master thread
  - No synchronization involved
  - Applicable only to parallel sections

Example:

```
#pragma omp parallel
{
  do_preprocessing();
  #pragma omp single
  read_input();
  #pragma omp master
  notify_input_consumed();

  do_processing();
}
```

Critical Sections

- #pragma omp critical [name]
  - Standard critical section functionality
- Critical sections are global in the program
  - Can be used to protect a single resource in different functions
- Critical sections are identified by the name
  - All the unnamed critical sections are mutually exclusive throughout the program
  - All the critical sections having the same name are mutually exclusive between themselves

Atomic execution

- Critical sections on the cheap
  - Protects a single variable update
  - Can be much more efficient (a dedicated assembly instruction on some architectures)
- #pragma omp atomic
  update_statement
- Update statement is one of: var= var op expr, var op= expr, var++, var--.
  - The variable must be a scalar
  - The operation op is one of: +, -, *, /, ^, &, |, <<, >>
  - The evaluation of expr is not atomic!

Ordered

- #pragma omp ordered
  statement
  - Executes the statement in the sequential order of iterations
- Example:

```
#pragma omp parallel for
for (j=0; j<N; j++) {
   int result = heavy_computation(j);
   #pragma omp ordered
   printf("computation(%d) = %d\n", j, result);
}
```

Barrier synchronization

- #pragma omp barrier
- Performs a barrier synchronization between all the threads in a team *at the given point*.
- Example:

```
#pragma omp parallel
{
  int result = heavy_computation_part1();
  #pragma omp atomic
  sum += result;
  #pragma omp barrier
  heavy_computation_part2(sum);
}
```

OpenMP Runtime System

- Each #pragma omp parallel creates a team of threads, which exist as long as the following block executes
  - #pragma omp for and #pragma omp section must be placed dynamically within a #pragma omp parallel.
  - Optimization: If there are several #pragma omp for and/or #pragma omp section within the same parallel, the threads will not be destroyed and created again
- Problem: a #pragma omp for is not permitted within a dynamic extent of another #pragma omp for
  - Must include the inner #pragma omp for within its own #pragma omp parallel
  - Nested parallelism?
  - The effect is implementation-dependent (will it create a new set of threads?)

Controlling OpenMP behavior

- omp_set_dynamic(int)/omp_get_dynamic()
  - Allows the implementation to adjust the number of threads dynamically
- omp_set_num_threads(int)/omp_get_num_threads()
  - Control the number of threads used for parallelization (maximum in case of dynamic adjustment)
  - Must be called from sequential code
  - Also can be set by OMP_NUM_THREADS environment variable
- omp_get_num_procs()
  - How many processors are currently available?
- omp_get_thread_num()
- omp_set_nested(int)/omp_get_nested()
  - Enable nested parallelism
- omp_in_parallel()
  - Am I currently running in parallel mode?
- omp_get_wtime()
  - A portable way to compute wall clock time

Explicit locking

- Can be used to pass lock variables around (unlike critical sections!)
- Can be used to implement more involved synchronization constructs
- Functions:
  - omp_init_lock(), omp_destroy_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock()
  - The usual semantics
- Use #pragma omp flush to synchronize memory

A Complete Example

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define NRA 62
#define NCA 15
#define NCB 7
int main() {
  int i,j,k,chunk=10;
  double a[NRA][NCA], b[NCA][NCB], c[NRA][NCB];
  #pragma omp parallel shared(a,b,c)
     private(tid,i,j,k)
  {
    /* Initialize */
    #pragma omp for schedule(static,chunk)
    for (i=0;i<NRA; i++)
      for (j=0;j<NCA; j++) a[i][j] = i+j;
    #pragma omp for schedule(static,chunk)
    for (i=0;i<NCA; i++)
      for (j=0;j<NCB; j++) b[i][j] = i*j;
    #pragma omp for schedule(static,chunk)
    for (i=0;i<NRA; i++)
      for (j=0;j<NCB; j++) c[i][j] = 0;
    #pragma omp for schedule(static,chunk)
    for (i=0; i<NRA; i++)
    {
      for (j=0; j<NCB; j++)
        for (k=0; k<NCA; k++)
          c[i][j] += a[i][k] * b[k][j];
    }
  } /* End of parallel section */

  /* Print the results … */
}
    }
```

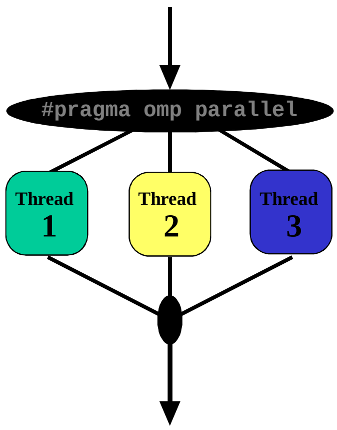
Conclusions

- Parallel computing is good today and indispensible tomorrow
  - Most upcoming processors are multicore
- OpenMP: A framework for code parallelization
  - Available for C++ and FORTRAN
  - Based on a standard
  - Implementations from a wide selection of vendors
- Easy to use
  - Write (and debug!) code first, parallelize later
  - Parallelization can be incremental
  - Parallelization can be turned off at runtime or compile time
  - Code is still correct for a serial machine

OpenMP: Tutorial

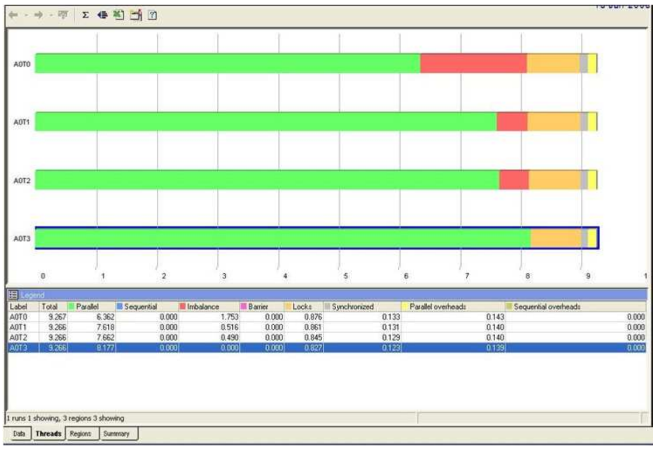
- Most constructs in OpenMP* are compiler directives or pragmas.
  - For C and C++, the pragmas take the form:

    #pragma omp *construct [clause [clause]…]*
- Main construct:
  #pragma omp parallel
  - Defines a *parallel region* over structured block of code
  - Threads are created as **'parallel'** pragma is crossed
  - Threads block at end of region



OpenMP: Methodology

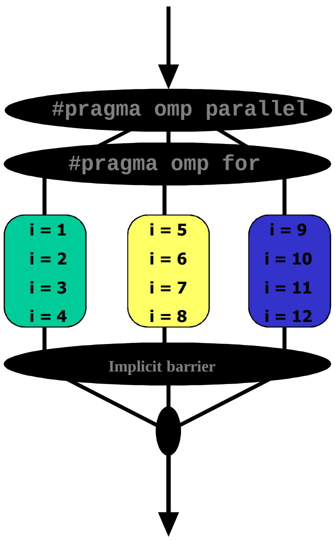
- Parallelization with OpenMP is an optimization process. Proceed with care:
  - Start with a working program, then add parallelization (OpenMP helps greatly with this)
  - Measure the changes after every step. Remember Amdahl's law.
  - Use the profiler tools available



Work-sharing: the for loop

```
#pragma omp parallel
#pragma omp for
   for(i = 1; i < 13; 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



OpenMP Data Model

- OpenMP uses a shared-memory programming model
  - Most variables are shared by default.
  - Global variables are shared among threads
    - C/C++: File scope variables, static
- But, not everything is shared...
  - Stack variables in functions called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE
  - Loop index variables are private (with exceptions)
    - C/C+: The first loop index variable in nested loops following a **`#pragma omp for`**

#pragma omp private modifier

- Reproduces the variable for each thread
  - 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;
     }
}
```

- If initialization is necessary, use firstprivate(x) modifier

#pragma omp shared modifier

- Notify the compiler that the variable is shared

```
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's the problem here?

Shared modifier cont'd

- Protect shared variables from data races

```
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;
}
```

- Another option: use #pragma omp atomic
  - Can protect only a single assignment
  - Generally faster than critical

#pragma omp reduction

- Syntax: #pragma omp 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

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

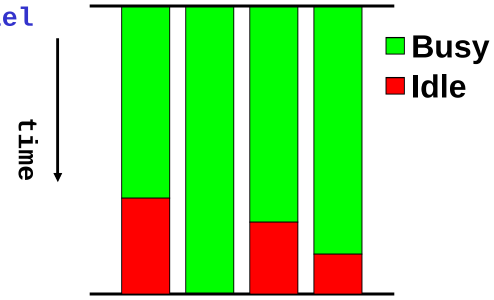
Performance Issues

- Idle threads do no useful work
- Divide work among threads as evenly as possible
  - Threads should finish parallel tasks at same time
- Synchronization may be necessary
  - Minimize time waiting for protected resources
- Parallelization Granularity may be too low

Load Imbalance

- Unequal work loads lead to idle threads and wasted time.
  - Need to distribute the work as evenly as possible!

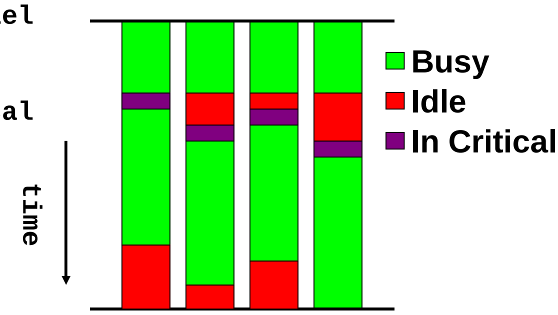
```
#pragma omp parallel
{

#pragma omp for
  for( ; ; ){

  }

}
```



Synchronization

- Lost time waiting for locks
  - Prefer to use structures that are as lock-free as possible!
  - Use parallelization granularity which is as large as possible.

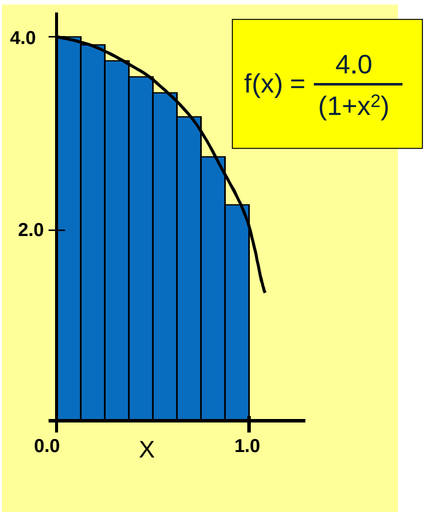
```
#pragma omp parallel
{

#pragma omp critical
  {
    ...
  }
  ...
}
```



Minimizing Synchronization Overhead

- Heap contention
  - Allocation from heap causes implicit synchronization
  - Allocate on stack or use thread local storage
- Atomic updates versus critical sections
  - Some global data updates can use atomic operations (Interlocked family)
  - Use atomic updates whenever possible
- Critical Sections versus mutual exclusion
  - Critical Section objects reside in user space
  - Use CRITICAL SECTION objects when visibility across process boundaries is not required
  - Introduces lesser overhead
  - Has a spin-wait variant that is useful for some applications

Example: Parallel Numerical Integration




```
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);
}
```

Computing Pi through integration

```
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?

Computing Pi through integration

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

void main()
{  int i;
   double x, sum = 0.0;

   step = 1.0/(double) num_steps;
   #pragma omp parallel for \
      private(x) 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;
   printf("Pi = %f\n",pi);
}
```

i is private since it is the loop variable

Assigning iterations

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

`schedule(dynamic[,chunk])`

- Threads grab "chunk" iterations
- When done with iterations, thread requests next set

`schedule(guided[,chunk])`

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

When to use
Predictable and similar work per iteration Small iteration size
Unpredictable, highly variable work per iteration Large iteration size
Special case of dynamic to reduce scheduling overhead

Example: What schedule to use?

- The function TestForPrime (usually) takes little time
  - But can take long, if the number is a prime indeed

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

- Solution: use dynamic, but with chunks

Getting rid of loop dependency

```
for (I=1; I<N; I++)
  a[I] = a[I-1] + heavy_func(I);
```

Transform to:

```
#pragma omp parallel for
for (I=1; I<N; I++)
   a[I] = heavy_func(I);
/* serial, but fast! */
for (I=1; I<N; I++)
   a[I] += a[I-1];
```

Compiler support for OpenMP

General Optimization Flags

Mac*/Linux*	Windows*	
-O0	/Od	Disables optimizations
-g	/Zi	Creates symbols
-O1	/O1	Optimize for Binary Size: Server Code
-O2	/O2	Optimizes for speed (default)
-O3	/O3	Optimize for Data Cache: Loopy Floating Point Code

OpenMP Compiler Switches

- Usage:
  - OpenMP switches: **`-openmp`** : **`/Qopenmp`**
  - OpenMP reports: **`-openmp-report`** : **`/Qopenmp-report`**

```
#pragma omp parallel for
 for (i=0;i<MAX;i++)
    A[i]= c*A[i] + B[i];
```

Intel's OpenMP Extensions

- Workqueuing extension
  - Create Queue of tasks…Works on…
    - Recursive functions
    - Linked lists, etc.
- Non-standard!!!

```
#pragma intel omp parallel taskq shared(p)
{
   while (p != NULL) {
#pragma intel omp task captureprivate(p)
   do_work1(p);
      p = p->next;
   }
}
```

Auto-Parallelization

- Auto-parallelization: Automatic threading of loops without having to manually insert OpenMP* directives.
  - Compiler can identify "easy" candidates for parallelization, but large applications are difficult to analyze.

Mac*/Linux*	Windows*
-parallel	/Qparallel
-par_report[n]	/Qpar_report[n]

- Also, use parallel libraries, example: Intel's MKL