Heterogeneous Computing using openMP
lecture 2

F21DP Distributed and Parallel Technology

Sven-Bodo Scholz
int main()
{
    int i; double x, pi, sum[10] = 0.0; int num_t;
    step = 1.0/(double)num_steps;
    omp_set_num_threads(10);
    #pragma omp parallel
    {
        int i, id, num_threads;
        double x;
        id = omp_get_thread_num();
        num_threads = omp_get_num_threads();
        if(id==0) num_t = num_threads;
        for(i = id; i<num_steps; i = i + num_threads )
        {
            x = (i+.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0; i<num_t; i++)
    {
        pi += sum[i] * step;
    }
}
Synchronisation in openMP

- **#pragma omp barrier**
  - synchronises across all threads

- **#pragma omp critical {}**
  - ensures mutual exclusion

- **#pragma omp atomic {}**
  - tries to use an atomic operation, i.e. only works for very simple operations. If not possible, it turns into omp critical
Using Critical Sections

int main()
{
    int i; double x, pi, sum[10]=0.0; int num_t;
    step = 1.0/(double)num_steps;
   omp_set_num_threads(10);
#pragma omp parallel
    { int i, id, num_threads;
        double x;
        id = omp_get_thread_num();
        num_threads = omp_get_num_threads();
        if( id==0) num_t = num_threads;
        for( i = id; i<num_steps; i = i + num_threads ){
            x = (i+0.5) * step;
            #pragma omp critical
                {sum[id] += 4.0/(1.0+x*x);}   
        }
    }
    for(i=0; i< num_t; i++)
        pi += sum[i] * step;
}
Going Atomic

```c
int main()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double)num_steps;
    omp_set_num_threads(10);
    #pragma omp parallel
    { int i, id, num_threads;
        double x;
        id = omp_get_thread_num();
        num_threads = omp_get_num_threads();

        for( i = id; i < num_steps; i = i + num_threads ) {
            x = (i+0.5) * step;
            x = 4.0/(1.0+x*x);
            #pragma omp atomic
            { sum += 4.0/(1.0+x*x); }
        }
    }
    pi = sum * step;
}
```
Concurrent Loops

• OpenMP provides a loop pattern!!
  – Requires: No data dependencies (reads/write or write/write pairs) between iterations!
• Preprocessor calculates loop bounds for each thread directly from serial source
• Scheduling no longer hand-coded

```c
#pragma omp parallel
{
  #pragma omp for
  for( i=0; i < 20; i++ )
  {
    printf(“Hello World!”);
  }
}
```
Loop Scheduling

• schedule clause determines how loop iterations are divided among the thread team
  – **static([chunk])** divides iterations statically between threads
    • Each thread receives [chunk] iterations, rounding as necessary to account for all iterations
    • Default [chunk] is \( \text{ceil}( \# \text{ iterations} / \# \text{ threads} ) \)
  – **dynamic([chunk])** allocates [chunk] iterations per thread, allocating an additional [chunk] iterations when a thread finishes
    • Forms a logical work queue, consisting of all loop iterations
    • Default [chunk] is 1
  – **guided([chunk])** allocates dynamically, but [chunk] is exponentially reduced with each allocation
Loop Scheduling II

• schedule clause determines how loop iterations are divided among the thread team
  – **auto** leaves the choice to the compiler
  – **runtime** enables dynamic control through
    • Environment variable **OMP_SCHEDULE** *type*
    • Library function **omp_set_schedule**(*type*)
Using **OMP FOR**

```c
int main()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double)num_steps;
    omp_set_num_threads(10);
    #pragma omp parallel
    { int i, id, num_threads;
      double x;
      id = omp_get_thread_num();
      #pragma omp for schedule(static)
      for(i=0; i<num_steps; i += num_threads )
      {
        x = (i+0.5) * step;
        x = 4.0/(1.0+x*x);
        #pragma omp atomic
        { sum += x; }
      }
    }
    pi = sum * step;
}
```
Reductions

• Typical pattern:

```c
double vect[N];
double accu = neutral;

for( i=0; i<N; i++) {
    accu = accu \text{op big}_\text{load}(i);
}
```

• Examples: sum / product / mean / ...
double vect[N];
double accu = neutral;

#pragma omp parallel for reduction( op: accu)
for( i=0; i<N; i++) {
  accu = accu op big_load(i);
}

Reductions
Using \textit{Reduction}

int main()
{
  int i; double x, pi, sum = 0.0;
  step = 1.0/(double)num_steps;
  \texttt{omp_set_num_threads(10);}
  \#pragma omp parallel
  {
    double x;
    \#pragma omp for static(1) reduction(+:sum)
    for( i = 0; i < num_steps; ++i )
      x = (i + 0.5) * step;
      sum += 4.0/(1.0 + x*x);
    \#pragma omp atomic
        \{ sum += x; \}
  }
  pi = sum * step;
}
Private vs Shared

• Default behaviour:
  – outside the parallel region: shared
  – inside the parallel region: private

• Explicit control:
  – clause \texttt{shared( var, ...)}
  – clause \texttt{private( var, ...)}
Example:

```
int x=1, y=1, z=1, q=1;

#pragma parallel private(y,z) shared(q)
{
  x=2;
  y=2;
  q=z;
}
```

What values of x, y, z and q do we have after the parallel region?

x==2
y==1
z==1
q==<undefined>
Private vs Shared ctnd.

• Default behaviour:
  – outside the parallel region: shared
  – inside the parallel region: private

• Explicit control:
  – clause \texttt{shared( var, ...)}
  – clause \texttt{private( var, ...)}
  – clause \texttt{firstprivate( var, ...)}
Example:

```c
int x=1, y=1, z=1, q=1;

#pragma parallel private(y) shared(q) firstprivate(z)
{
    x=2;
    y=2;
    q=z;
}
```

What values of x, y, z and q do we have *after* the parallel region?

- x==2
- y==1
- z==1
- q==1
Summary

• The most common pattern can easily be achieved
• There is much more (->www.openmp.org)
  – Tasks
  – Sections
  – Vector-instructions
  – Teams
  – …

Check it out 😊