Heterogeneous Computing using openMP
lecture 1

F21DP Distributed and Parallel Technology

Sven-Bodo Scholz
Introduction to OpenMP

• What is OpenMP?
  – Open specification for Multi-Processing
  – “Standard” API for defining multi-threaded shared-memory programs
  – openmp.org – Talks, examples, forums, etc.

• High-level API
  – Preprocessor (compiler) directives ( ~ 80% )
  – Library Calls ( ~ 19% )
  – Environment Variables ( ~ 1% )
A Programmer’s View of OpenMP

• OpenMP is a portable, threaded, shared-memory programming specification with “light” syntax
  – Exact behavior depends on OpenMP implementation!
  – Requires compiler support (C or Fortran)

• OpenMP will:
  – Allow a programmer to separate a program into serial regions and parallel regions
  – Hide stack management
  – Provide synchronisation constructs

• OpenMP will not:
  – Parallelize automatically
  – Guarantee speedup
  – Provide freedom from data races
int main() {

    // Do this part in parallel

    printf( "Hello, World!\n" );

    return 0;
}
#include <omp.h>

int main() {

    omp_set_num_threads(16);

    // Do this part in parallel
    #pragma omp parallel
    {
        printf( "Hello, World!\n" );
    }

    return 0;
}

compile with –fopenmp    !!!!
Programming Model
Computation of π

\[ F(x) = \frac{4.0}{1 + x^2} \]

\[ \int_{0}^{1} \frac{4.0}{1 + x^2} \]
int num_steps = 10000;
double step;

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

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

```
#include<omp.h>
int num_steps = 10000;
double step;

int main()
{
    int i; double x, pi, sum=0.0;
    step = 1.0/(double)num_steps;
    #pragma omp_set_num_threads( 10);
    #pragma omp parallel
    {
        for( i=0; i<num_steps; i++)
        {
            x = (i+0.5) * step;
            sum += 4.0/(1.0+x*x);
        }
    }

    pi = step * sum;
    ...
}
```
Exposing Concurrency

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

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

    pi = 4.0*sum;
    printf("The value of pi is %f\n", pi);
}
```

race condition!!!
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_threads; i = i + num_threads ){
      x = (i+0.5) * step;
      sum [id] += 4.0/(1.0+x*x);
    }
  }
  for(i=0; i<num_threads; i++)
    pi += sum[i] * step;
}
int main()
{
    int i; double x, pi, sum[10] = 0.0; int num_t;
    step = 1.0/(double)num_steps;
    #pragma 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;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0; i<num_t; i++)
        pi += sum[i] * step;
}
False Sharing!

\[
\text{sum [0]} \quad += \quad \frac{4.0}{1.0 + x^2};
\]

\[
\text{sum [num_t - 1]} \quad += \quad \frac{4.0}{1.0 + x^2};
\]

Cache invalidation on every single write!!!
Summary so far

• Introduction of concurrency is easy!
• Pitfalls:
  – scope of variables (global/local)
  – race conditions
  – false sharing

=> constructs introduced so far are too rudimentary!!