SMM (2)

OpenMP Programming Model

- Reading for next time
  - OpenMP tutorial: look through secns 3-5 plus secn 6 up to exercise 1
Topics

• OpenMP shared-memory parallel programming model
  – loop-level parallel programming

• Characterizing performance
  – performance measurement of a simple program
  – how to monitor and present program performance
  – general barriers to performance in parallel computation
Shared-memory programming models

- **Work-Time programming model**
  - sequential programming language + `forall`
    - PRAM execution
      - synchronous
      - scheduling implicit (via Brent’s theorem)
    - W-T cost model (work and steps)

- **Loop-level parallel programming model**
  - sequential programming language + directives to mark `for` loop as “forall”
    - shared-memory multiprocessor execution
      - asynchronous execution of loop iterations by multiple threads in a single address space
        - must avoid dependence on synchronous execution model
      - scheduling of work across threads is controlled via directives
        - implemented by the compiler and run-time systems
    - cost model depends on underlying shared memory architecture
      - can be difficult to quantify
      - but some general principles apply
Example shared-memory machine

Phaedra

- 10 compute cores per socket, total 20 cores
- Single shared physical address space
- 64 GB memory per socket, 128 GB total shared memory
- Cache-coherence protocol for performance
OpenMP

- **OpenMP**
  - parallelization directives for mainstream performance-oriented sequential programming languages
    - C/C++, Fortran (88, 90/95)
  - directives are written as comments in the program text
    - ignored by non-OpenMP compilers
    - honored by OpenMP-compliant compilers in “OpenMP” mode
  - directives specify
    - parallel execution
      - create multiple threads, generally each thread runs on a separate core in a CC-NUMA machine
    - partitioning of variables
      - a variable is either shared between threads OR each thread maintains a private copy
    - work scheduling in loops
      - partitioning of loop iterations across threads

- **C/C++ binding of OpenMP**
  - form of directives
    - #pragma omp . . .
OpenMP parallel execution of loops

...  
printf(“Start.\n”);  
for (i = 1; i < N-1; i++) {
    b[i] = (a[i-1] + a[i] + a[i+1]) / 3;
}
printf(“Done.\n”);
...

- Can different iterations of this loop be executed simultaneously?  
  - for different values of \( i \), the body of the loop can be executed simultaneously

- Suppose we have \( n \) iterations and \( p \) threads?  
  - we have to *partition* the iteration space across the threads
OpenMP directives to control partitioning

```c
... 
printf(“Start.\n”);

#pragma omp parallel for shared(a,b) private(i)
for (i = 1; i < N-1; i++) {
    b[i] = (a[i-1] + a[i] + a[i+1]) / 3;
}
printf(“Done.\n”);
... 
```

- The **parallel** directive indicates the next *statement* should be executed by all threads.
- The **for** directive indicates the work in the loop body should be partitioned across threads.
- The **shared** directive indicates that arrays `a` and `b` are shared by all threads.
- The **private** directive indicates `i` has a separate instance in each thread.
- The last two directives would be inferred by the OpenMP compiler.
OpenMP components

- **Directives**
  - specify parallel vs sequential regions
  - specify shared vs private variables in parallel regions
  - specify work sharing: distribution of loop iterations over threads
  - specify synchronization and serialization of threads

- **Run-time library**
  - obtain parallel processing resources
  - control dynamic aspects of work sharing

- **Environment variables**
  - external to program
  - specification of resources available for a particular execution
    - enables a single compiled program to run using differing numbers of processors
C/OpenMP concepts: parallel region

Fork-join model
- master thread forks a team of threads on entry to block
  - variables in scope within the block are
    - shared among all threads
      » if declared outside of the parallel region
      » if explicitly declared shared in the directive
    - private to (replicated in) each thread
      » if declared within the parallel region
      » if explicitly declared private in the directive
      » if variable is a loop index variable in a loop within the region
  - the team of threads has dynamic lifetime to end of block
    - statements are executed by all threads
  - the end of block is a barrier synchronization that joins all threads
    - only master thread proceeds thereafter

#pragma omp parallel shared(...) private(...)  
<single entry, single exit block>
C/OpenMP concepts: work sharing

```c
#pragma omp for schedule(…)
for (<var> = <lb>; <var> <op> <ub>; <incr-expr>)
    <loop body>
```

- **Work sharing**
  - only has meaning inside a parallel region
  - the *iteration space* is distributed among the threads
    - several different scheduling strategies available
  - the loop construct must follow some restrictions
    - <var> has a signed integer type
    - <lb>, <ub>, <incr-expr> must be loop invariant
    - <op>, <incr-expr> restricted to simple relational and arithmetic operations
  - implicit barrier at completion of loop
#include <stdio.h>
#include <omp.h>
define N 50000000
#define NITER 100

double a[N], b[N];
main ()
{
  double t1, t2, td;
  int i, t, max_threads, niter;

  max_threads = omp_get_max_threads();
  printf("Initializing: N = %d, max # threads = %d\n", N, max_threads);

  /*
   * initialize arrays
   */
  for (i = 0; i < N; i++){
    a[i] = 0.0;
    b[i] = 0.0;
  }
  a[0] = b[0] = 1.0;
Shared Memory Multiprocessing (2)

Program, contd. (V1)

/*
 * time iterations
 */
t1 = omp_get_wtime();
for (t = 0; t < NITER; t = t + 2){

    #pragma omp parallel for private(i)
    for (i = 1; i < N-1; i++)
        b[i] = (a[i-1] + a[i] + a[i+1]) / 3.0;

    #pragma omp parallel for private(i)
    for (i = 1; i < N-1; i++)
        a[i] = (b[i-1] + b[i] + b[i+1]) / 3.0;
}

    t2 = omp_get_wtime();
td = t2 - t1;
    printf("Time per element = %6.1f ns\n", td * 1E9 / (NITER * N));
}
/\*
  * time iterations
  */
  
t1 = omp_get_wtime();

#pragma omp parallel private(i,t)
for (t = 0; t < NITER; t = t + 2){

  #pragma omp for
  for (i = 1; i < N-1; i++)
    b[i] = (a[i-1] + a[i] + a[i+1]) / 3.0;

  #pragma omp for
  for (i = 1; i < N-1; i++)
    a[i] = (b[i-1] + b[i] + b[i+1]) / 3.0;

}

t2 = omp_get_wtime();
td = t2 - t1;
printf("Time per element = %6.1f ns\n", td * 1E9 / (NITER * N));
}
```c
#include <stdio.h>
#include <omp.h>
define N 50000000
#define NITER 100

double a[N], b[N];

main ()
{
    double t1, t2, td;
    int i, t, max_threads, niter;

    max_threads = omp_get_max_threads();
    printf("Initializing:  N = %d, max # threads = %d\n", N, max_threads);

    #pragma omp parallel private(i, t)
    { // start parallel region

        /*
        * initialize arrays
        */
        #pragma omp for
        for (i = 1; i < N; i++){
            a[i] = 0.0;
            b[i] = 0.0;
        }

        #pragma omp master
        a[0] = b[0] = 1.0;
    }
} // end parallel region
```

Complete program (V3 – page and cache affinity)
/*
 * time iterations
 */
#pragma omp master
t1 = omp_getwtime();

for (t = 0; t < NITER; t = t + 2){

#pragma omp for
for (i = 1; i < N-1; i++)
    b[i] = (a[i-1] + a[i] + a[i+1]) / 3.0;

#pragma omp for
for (i = 1; i < N-1; i++)
    a[i] = (b[i-1] + b[i] + b[i+1]) / 3.0;
}
} // end parallel region

t2 = omp_get_wtime();
td = t2 - t1;
printf("Time per element = %6.1f ns\n", td * 1E9 / (NITER * N));
}
Effect of caches

- Time to update one element in *sequential execution*
  - \( b[i] = (a[i-1] + a[i] + a[i+1]) / 3.0; \)
  - depends on where the elements are found
    - registers, L1 cache, L2 cache, main memory
How to present scaling of parallel programs?

- **Independent variables**
  - either
    - number of processors $p$
    - problem size $n$

- **Dependent variable (choose)**
  - Time (secs)
  - Rate (opns/sec)
  - Speedup $S = \frac{T_1}{T_p}$
  - Efficiency $E = \frac{T_1}{pT_p}$

- **Horizontal axis**
  - independent variable ($n$ or $p$)

- **Vertical axis**
  - Dependent variable (e.g. time per element)
  - May show multiple curves (e.g. different values of $n$)
Time

- Shortest time is our true goal
  - But hard to judge improvements because values get very small at large $p$
Execution rate (MFLOP / second)

- Shows work per time
  - easier to judge scaling
  - highest detail at large $n$, $p$
  - how to measure MFLOPS?

Parallel performance

- MFLOP / second
- Number of processors
- $n = 10,000,000$
- $n = 1,000,000$
Speedup

- Speedup of run time relative to single processor \((t_1 / t_p)\)
  - How to define \(t_1\)?
    - run time of parallel algorithm at \(p = 1\)?
    - run time of best serial algorithm?
  - Superlinear speedup?
OpenMP: scheduling loop iterations

- Scheduling a loop with \( n \) iterations using \( p \) threads
  - The unit of scheduling is a chunk of \( k \) iterations
  - \( T_i \) means iteration(s) executed by thread \( i \)

- \( \text{schedule}(\text{static}, k) \)
  - Chunks mapped to threads in
    at entry to loop
  - default \( k = n/p \)

- \( \text{schedule}(\text{dynamic}, k) \)
  - chunks handed out consecutively
    to ready threads
  - default \( k = 1 \)

- \( \text{schedule}(\text{guided}, k) \)
  - size \( d \) chunk handed to
    first available thread
  - \( d \) decreases exponentially
    from \( n/p \) down to \( k \):
    \[ d_{i+1} = (1-1/p)d_i \] where \( d_0 = n/p \)
  - default \( k = 1 \)
Varying scheduling strategy: diffusion problem

Speedup by schedule type
(n = 10,000,000)

Number of processors vs. speedup for different scheduling strategies:
- p
- static
- guided
- dynamic,32
- dynamic
Causes of poor parallel performance

Possible suspects:

- Low computational intensity
  - Performance limited by memory performance

- Poor cache behavior
  - access pattern has poor locality
  - access pattern is poorly matched to CC-NUMA

- Sequential overhead
  - Amdahl’s law
    - fraction f serial work limits speedup to $1/f$

- Load imbalance
  - Unequal distribution of work, or
  - Unequal thread progress on equal work
    - busy machine, uncooperative OS
    - CC-NUMA issues

- Bad luck
  - Insufficient sampling - show timing variation on plots!
Cache-related mysteries

Execution rate

MFLOP/second vs. number of processors

- Red line: $n = 10,000,000$
- Blue line: $n = 1,000,000$

Execution rate for different $n$ values.
Cache-related mysteries: speedup

Parallel speedup
(single parallel region)

![Graph showing parallel speedup with lines for n = 1,000,000 and n = 10,000,000. The x-axis represents the number of processors, and the y-axis represents speedup. There are two lines: one for each value of n, with the line for n = 10,000,000 being more steep than the one for n = 1,000,000.]
OpenMP on CC-NUMA

- Performance guidelines
  - shared data structures
    - use cache-line spatial locality
      - linear access patterns (read and write)
      - structs with components grouped by access
    - don’t mix reads and writes to same data on different processors
      - use phased updates
    - avoid false sharing
      - unrelated values sharing a cache line updated by multiple threads
  - make sure data structures are physically distributed across memory
    - by parallel initialization
      » artifact of page placement policy under e.g. Linux
    - by explicit placement directives and page allocation policies
OpenMP on CC-(N)UMA

- Other guidelines
  - Enlarge parallel region
    - to retain processor – data affinity
    - to avoid overhead of repeated entry to parallel region in an inner loop
  - Use appropriate work distribution schedule
    - static, else
    - guided, else
    - dynamic with large chunksize
    - runtime-specified schedule involves relatively small overhead
  - Don’t use too many processors
    - OS scheduling of threads behaves erratically when machine is oversubscribed
    - be aware of dynamic thread adjustment (OMP_DYNAMIC)
Reductions and critical statements

- A **reduction loop** does not have independent iterations

  ```c
  for (i=0; i<n; i++) {
      sum = sum + a[i];
  }
  ```

- The loop may be parallelized by inserting a **critical section**
  - The **critical directive** serializes a single statement or block

    ```c
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        #pragma omp critical
        sum = sum + a[i];
    }
    ```
  - But this is a poor strategy!

- A reduction loop can be identified using a **reduction directive**

  ```c
  #pragma omp parallel for reduction(+: sum)
  for (i=0; i<n; i++) {
      sum = sum + a[i];
  }
  ```
Implementation of reduction directive

- A better implementation of the reduction loop
  
  ```c
  sum = 0;
  #pragma omp parallel
  {
    int i, local_sum = 0;
    #pragma omp for
    for (i=0; i<n; i++) {
      local_sum = local_sum + a[i];
    }
    #pragma omp critical
    sum = sum + local_sum;
  }
  
  – reduces number of critical operations from $n$ to $p$
  ```

- other reduction strategies
  
  - serialization: master thread sequentially combines local_sum values
  - tree-based reduction
  - hybrid strategy

  OpenMP compiler should generate code that selects optimal strategy at run time