Loop-Level Parallelism and OpenMP

CS149
Lecture 14

Announcements

• PA4 due Thursday, Feb 25

Loop Parallelism

- Overwhelming majority of scientific/engineering applications are expressed in terms of iterative constructs, that is, loops
  - Focus on parallelizing loops
- Particular useful approach if starting from an existing program
  - Major restructuring is impractical/unecessary
- Goal of this pattern is to evolve the sequential program into a parallel program
  - Through transformations that leave the program semantics unchanged
- This pattern works well for shared address spaces
- Recall Amdahl's law and its requirement to minimize a program's serial fraction
  - Using Loop Parallelism exclusively may limit scalability

General Approach for Loop Parallelism

1. Find the hotspots
2. Eliminate loop-carried dependencies
3. Parallelize the loops
4. Optimize the loop schedule
Parallel Loops

• for (i = 0, i < n, i++) {
}

• for (i = 0, i < n, i++) {
  \[A[i+1] = A[i] + C[i];\]
  \[B[i+1] = B[i] + A[i+1];\]
}

• for (i = 2, i < n, i++) {
  \[C[i] = A[i-2] + B[i];\]
}

Find the Hotspots

• By code inspection

• By using performance analysis tools

What is OpenMP?

• OpenMP is a pragma based API that provides a simple extension to C/C++ and FORTRAN

• It is exclusively designed for shared memory programming

• OpenMP is a very simple interface to threads based programming
  - Compiler directives
  - Environment variables
  - Run time routines

OpenMP: Where did it come from?

• Prior to 1997, vendors all had their own proprietary shared memory programming commands

• Programs were not portable from one SMP to another

• Developers were calling for some kind of portability

• ANSI X3H5 (1994) proposal tried to formalize a shared memory standard – but ultimately failed

• OpenMP (1997) worked because the vendors got behind it and there was new growth in the shared memory arena
Advantages of OpenMP

- De-facto standard
- An OpenMP program is portable
  - Supported by a large number of compilers
- Requires little programming effort
- Allows the program to be parallelized incrementally
- Maps naturally onto a multicore architecture:
  - Lightweight
  - Each OpenMP thread in the program can be executed by a hardware thread

First OpenMP Example

For-loop with independent iterations

```c
for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```

For-loop parallelized using an OpenMP pragma

```c
#pragma omp parallel for
shared(a, b, c)
pprivate(i)
for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```

OpenMP Execution Model

OpenMP Directives

- C: directives are case sensitive
  - Syntax: #pragma omp directive [clause [clause] ...]
- Continuation: use \ in pragma
OpenMP Example Implementation

```c
#include <omp.h>

main(){
  #pragma omp parallel for default(private)
  num_threads(NUM_PROCS) . . . << var info >> . . .
  for (i=0; i < NUM_PROCS; i++)
  {
    /* Parallel Work Here */
  }
  /* End of OpenMP parallel region */
}
```

Privatizing Variables

- Critical to performance!
- OpenMP pragmas:
  - Designed to make parallelizing sequential code easier
  - Makes copies of "private" variables automatically
    - And performs some automatic initialization, too
  - Must specify shared/private per-variable in parallel
    - private: Uninitialized private data
      - Private variables are undefined on entry and exit of the parallel region
    - shared: All-shared data
    - threadprivate: "static" private for use across several parallel regions

Firstprivate/Lastprivate Clauses

- `firstprivate (list)`
  - All variables in the list are initialized with the value the original object had before entering the parallel region
- `lastprivate(list)`
  - The thread that executes the last iteration or section in sequential order updates the value of the objects in the list

Example Private Variables

```c
main()
{
  A = 10;
  #pragma omp parallel
  {  
    #pragma omp for private(i) firstprivate(A) lastprivate(B)...
    for (i=0; i<n; i++)
    {
      ....
      B = A + i;  /* A undefined, unless declared firstprivate */
      ....
    }
    C = B;  /* B undefined, unless declared lastprivate */
  }
  /* End of OpenMP parallel region */
}
```
for directive Example

```c
#pragma omp parallel default(mone) 
    shared(a,b,c,d) private(i)
{
    #pragma omp for nowait
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
    for (i=0; i<n; i++)
        d[i] = 1.0/c[i];
}  
/*-- End of parallel region --*/

Prof. Aiken/Olukotun    CS 149 Lecture 14
```

Loop Level Parallelism with OMP

- Consider the single precision vector add-multiply operation \(Y = aX + \overline{Y}\) ("SAXPY")

```c
for (i=0; i<n-1; i++)
    Y[i] += a*X[i];

#pragma omp parallel for 
    private(i) shared(X, Y, n, a)
for (i=0; i<n; i++)
    Y[i] += a*X[i];
```

OpenMP Sections

- Parallel threads can also do different things with sections
  - Use instead of for in the pragma, and no attached loop
  - Contains several section blocks, one per thread

- You can also have a “multi-part” parallel region
  - Allows easy alternation of serial & parallel parts
  - Doesn’t require re-specifying # of threads, etc.

```c
#pragma omp parallel 
{ 
    #pragma omp section 
        taskA(); 
    #pragma omp section 
        taskB(); 
}  
/*-- End of parallel region --*/
```

Sections Example

```c
#pragma omp parallel default(mone) 
    shared(a,b,c,d) private(i)
{
    #pragma omp sections nowait
    { 
        #pragma omp section 
            b[i] = (a[i] + a[i+1])/2;
        #pragma omp section 
            d[i] = 1.0/c[i];
    }  
/*-- End of sections --*/

Prof. Aiken/Olukotun    CS 149 Lecture 14
```
"if" Clause

- if (scalar expression)
  - Only execute in parallel if expression evaluates to true
  - Otherwise, execute serially

Performance without if clause

Locks and Critical Sections in OpenMP

- The OpenMP critical directive is just a lock, but completely different syntax
  - #pragma omp critical [(name)] {<code-block>}
- #pragma omp ordered {<code-block>}
- #pragma omp master {<code-block>}

Reductions

- One of the banes of parallelism is a reduction in dimensionality
  - Go from N dimensions to N-1, N-2, ... 0
  - Dot products are the most common example
    - \( a[i] = a[i] + b[j] \times c[j] \)
- Single output, associative reduction
  - Combine to P elements
  - Do as much of the reduction in parallel as possible
  - Do final step in serial (small P) or in a parallel tree (large P)
- Single output, non-associative reduction
  - It's serial, so try to overlap parts of tasks
  - Good place to apply dataflow/pipeline parallelism

Reductions in OpenMP

- Reductions are so common that OpenMP provides support for them
- May add reduction clause to parallel for pragma
- Specify reduction operation and reduction variable
- OpenMP takes care of storing partial results in private variables and combining partial results after the loop
- The reduction clause has this syntax: reduction (op: <variable>)
- Operators
  - + Sum
  - * Product
  - &, |, ^ Bitwise and, or, exclusive or
  - &\&, || Logical and, or
Loop Parallelized Pi Computation

```c
#include <omp.h>
static long num_steps = 1000000; double step;
#define NUM_THREADS 8
void main(void)
{
    int i; double x, pi, 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;
}
```

- Notice that we haven’t changed any lines of code, only added 4 lines
- Compare to CUDA

Static Task Decompositions

- Many applications decompose into tasks easily
  - Fixed-size tasks
  - Known number of tasks
  - Both are important!

Static Partitioning with OpenMP

- OpenMP offers simple options for loops
  - `schedule(static, size)` distributes `size` iterations/CPU
    - Simple and clear
    - Nesting works in some environments
      - Works under Solaris
      - Usually use entire rows/columns of multi-D arrays
    - Can get stuck if you (`# iterations`)/(`size * n_procs`) not an integer
      - Some "extra" processors during last batch of blocks
    - This covers most common cases

Dividing Up the Loop

- Easy to allocate to processors
  - Fork off `n_procs` looping pthreads or use a parallel for
  - Allocate by:
    - Loop iteration (many tasks)
    - Chunks of loop iterations (medium)
    - 1/n_procs iterations/processor (fewest)
    - Decide allocation based on algorithm and architecture
      - Does it have a "natural" chunk size?
      - Does it have a particular communication pattern between iterations?
      - How expensive is communication?

Regular Arrays
- Fixed
- Irregular Data Structures

Profs. Aiken/Olukotun    CS 149 Lecture 14
Static Partitioning Comparison
16 iterations, 4 threads

<table>
<thead>
<tr>
<th>STATIC</th>
<th>THREAD 0</th>
<th>THREAD 1</th>
<th>THREAD 2</th>
<th>THREAD 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>No chunks</td>
<td>0 1 2 3</td>
<td>4 5 6 7</td>
<td>8 9 10 11</td>
<td>12 13 14 15</td>
</tr>
<tr>
<td>STATIC</td>
<td>THREAD 0</td>
<td>THREAD 1</td>
<td>THREAD 2</td>
<td>THREAD 3</td>
</tr>
<tr>
<td>size=1</td>
<td>0 4 8 12 1 5 9 13 2 6 10 14 3 7 11 15</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Problems with Static Partitioning

- Sometimes static task partitioning just won’t work:
  - Unknown number of tasks
  - Dependent upon a complex data structure (e.g. trees)
  - Tasks generated dynamically, as we work
  - Unknown size of tasks
  - Data-dependent execution time
  - Need to balance among processors at runtime

Dynamic Tasking with OpenMP

- OpenMP is a mixed bag
  - `schedule(dynamic, size)` is a dynamic equivalent to the `static` directive
    - Master passes off values of iterations to the workers of size size
    - Automatically handles dynamic tasking of simple loops
  - Otherwise must make your own
    - Includes many commonly used cases, unlike `static`
    - Just like pthreads, except must be lock-only

OpenMP Guided Scheduling

- `schedule(guided, size)`
- Guided scheduling is a compromise to reduce scheduling overhead
- Iteration space is divided up into exponentially decreasing chunks
- Final size is usually 1, unless set by the programmer
- Chunks of work are dynamically obtained
- Works quite well provided work per iteration is constant – if unknown dynamic is better
OpenMP Scheduling

500 iterations on 4 threads

Tasking in OpenMP 3.0

- Tasking allows parallelization of units of work that are dynamically generated
- Provides flexible model for irregular parallelism
- #pragma omp task [clause [{,clause}...]]
  - structured-block
- Task Synchronization
  - C/C++: #pragma omp taskwait
  - Current task suspends execution until all children tasks, generated within the current task up to this point, are complete

Fibonacci Example

- Default for local variables is firstprivate

```c
int fib ( int n )
{
  int x,y;
  if ( n < 2 ) return n;
  #pragma omp task shared(x)
  x = fib(n-1);
  #pragma omp task shared(y)
  y = fib(n-2);
  #pragma omp taskwait
  return x+y;;
}
```

OpenMP Summary

- OpenMP provides a simple programming model
  - Loops or sections
  - Incremental parallelism
- Targeted at shared memory systems
  - Won’t scale easily to large machines
- Compilers with OpenMP 2.5 support are widely available
- OpenMP 3.0 supports tasking
  - Supports irregular parallelism