Loop Parallelism (LLP)

- Almost all HPC codes use iterative constructs; i.e., loops
  - In contrast to general recursion

- OpenMP focuses on parallelizing loops
  - Particularly useful when starting from an existing program
  - Major restructuring is impractical/unnecessary

- Goal of exploiting LLP is to evolve sequential programs into parallel programs
  - Through transformations that leave the program semantics unchanged

- LLP works well for shared memory machines
  - Limited scalability beyond shared memory
General Approach for Loop Parallelism

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

Find the Hotspots

- By code inspection
- By using performance analysis tools
Parallel Loops

• for (i = 0; i < n; i++) {
}

• for (i = 1; i < n; i++) {
    A[i] = A[i-1] + C[i-1]; /* S1 */
    B[i] = B[i-1] + A[i]; /* S2 */
}

A[0] = A[0] + B[0];
for (i = 0; i < n-1; i++) {
    B[i+1] = C[i] + D[i]; /* S2 */
    A[i+1] = A[i+1] + B[i+1]; /* S1 */
}
B[n] = C[n-1] + D[n-1];
**Discussion**

- What is the difference between parallel loops and vectorization?

```c
for (i = 0; i < n; i++) {
}
```

- More than one operation in parallel
- Multiple threads vs. SIMD
- Need explicit programmer synchronization

- Multiple parallel programming models can be used in one application

---

**What is OpenMP?**

- OpenMP is a pragma based API
  - Extension to C/C++ and FORTRAN

- Designed for shared memory programming

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

- Prior to 1997, vendors all had their own proprietary shared memory programming commands
  - Programs were not portable
  - Developers were unhappy

- 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
  - E.g. GCC 4.2

- 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
Comments

• Multiple threads of control
  - One per processor
  - One master thread

• Synchronous/Asynchronous
  - Implicit synchronization after each parallel region
  - Some synchronization is programmer-specified

• Some variables are local (private), some global
  - Every thread has its own copy of private variables
  - Global variables are shared
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(none) 
    num_threads(NUM_PROCS) . . . << var info >> . . .
    for (i=0; i < NUM_PROCS; i++)
    {
        /* Parallel Work Here */
    }
    ... 
}
```

Important!!

- Simpler than java threads for this basic for example
- But harder for less structured parallelism (like webservers) until OpenMP 3.0
- Just “attaches” to the following for loop & runs it in parallel
First OpenMP Example

For-loop with independent iterations

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

For-loop parallelized using an OpenMP pragma

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

% cc -xopenmp source.c
% setenv OMP_NUM_THREADS 4
% a.out

gcc source.c -fopenmp

Privatizing Variables

- Critical to performance!
- OpenMP pragmas:
  - Designed to make parallelizing code easier
  - Makes copies of “private” variables automatically
    - And performs some automatic initialization, too
  - Must specify shared/private per-variable in parallel region
    - 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(none)\ 
    shared(n,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 --*/
(implies barrier)
```

Loop Level Parallelism with OMP

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

```c
for (i=0; i<n; ++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];
}
```
**Nested Loop Parallelism**

```c
#pragma omp parallel for
for(int y=0; y<25; ++y)
{
    #pragma omp parallel for
    for(int x=0; x<80; ++x)
        tick(x, y);
}
```

---

**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

```c
#pragma omp sections
{
    #pragma omp section
    { taskA(); }
    #pragma omp section
    { taskB(); }
}
```
Sections Example

```c
#pragma omp parallel default(none)
    shared(n,a,b,c,d) private(i)
{
    #pragma omp sections nowait
    {
        #pragma omp section
        for (i=0; i<n-1; i++)
            b[i] = (a[i] + a[i+1])/2;
        #pragma omp section
        for (i=0; i<n; i++)
            d[i] = 1.0/c[i];
    } /*-- End of sections --*/
} /*-- End of parallel region --*/
```

Multiple Part Parallel Regions

- 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 for
    . . . Loop here . . .
    #pragma omp single
    . . . Serial portion here . . .
    #pragma omp sections
    . . . Sections here . . .
}
```
“if” Clause

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

```
#pragma omp parallel if (a > threshold) \
  shared(a, x, y) private(i)
{
  #pragma omp for
  for (i=0; i<n; i++)
    x[i] = y[i];
}/*-- End of parallel region --*/
```

Performance without if clause

LOCKS IN OPENMP

<table>
<thead>
<tr>
<th>Lock Task</th>
<th>OpenMP function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock Object Type</td>
<td>omp_lock_t</td>
</tr>
<tr>
<td>Initialize New Lock</td>
<td>omp_init_lock</td>
</tr>
<tr>
<td>Destroy Lock</td>
<td>omp_destroy_lock</td>
</tr>
<tr>
<td>Blocking Lock Acquire</td>
<td>omp_set_lock</td>
</tr>
<tr>
<td>Lock Release</td>
<td>omp_unset_lock</td>
</tr>
<tr>
<td>Non-blocking Lock Acquire</td>
<td>omp_test_lock</td>
</tr>
</tbody>
</table>
OpenMP Synchronization

- **OpenMP** provides for a few useful "common cases"
- **barrier** implements an arbitrary barrier
  - A barrier anywhere in one line
  - Note that many other primitives *implicitly* add barriers, too

- **ordered locks and sequences a block**
  - Acts like a lock around a code block
  - Forces loop iterations to run block in "loop iteration" order
  - Only one allowed per loop

- **single/master** force only one thread to execute a block
  - Acts like a lock
  - Only allows one thread to run the critical code
  - Good for computing a common, global value or handling I/O

Programming Model - Synchronization

- **OpenMP Critical Sections**
  - Named or unnamed
  - No *explicit* locks / mutexes
  
```c
#pragma omp critical
{
    /* Critical code here */
}
```

- **Barrier directives**

```c
#pragma omp barrier
```

- **Explicit Lock functions**
  - When all else fails - may require *flush* directive

```c
omp_set_lock( lock l );
/* Code goes here */
omp_unset_lock( lock l );
```

- **Single-thread regions within parallel regions**
  - **master, single** directives

```c
#pragma omp single
{
    /* Only executed once */
}
```
Reductions

• A common pattern is a reduction in dimensionality
  - Go from N dimensions to N-1, N-2, ... 0
  - Dot products are a 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 last step serially (small P) or as 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:
  \[ \text{reduction} \ (<\text{op}> :<\text{variable}>) \]
• Operators
  - \(+\) \quad \text{Sum}
  - \(*\) \quad \text{Product}
  - \&, |, ^ \quad \text{Bitwise and, or, exclusive or}
  - \&\&, || \quad \text{Logical and, or}
Example: Numerical Integration

- We know mathematically that
  \[ \pi = \int_0^1 \frac{4.0}{1 + x^2} \, dx \]
  \[ \sum_{i=0}^N F(x_i)\Delta x \approx \pi \]
  - We can approximate the integral as a sum of rectangles:

Sequential Pi Computation

```cpp
static long num_steps = 100000;
double step;

void 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 = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```
Loop Parallelized Pi Computation

```c
#include <omp.h>
static long num_steps = 1000000; double step;
#define NUM_THREADS 8

void main(){
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #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 MPI

Static Task Decompositions

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

Regular Arrays

Fixed Irregular Data Structures
Dividing Up the Loop

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

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 \((# \text{iterations})/(\text{size} \cdot n_{\text{procs}})\) not integral
      - Some "extra" processors during last batch of blocks
    - This covers most common cases
### Static Partitioning Comparison

16 iterations, 4 threads

<table>
<thead>
<tr>
<th>STATIC No chunksize</th>
<th>THREAD 0</th>
<th>THREAD 1</th>
<th>THREAD 2</th>
<th>THREAD 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STATIC size=1</th>
<th>THREAD 0</th>
<th>THREAD 1</th>
<th>THREAD 2</th>
<th>THREAD 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 4 8 12</td>
<td>1 5 9 13</td>
<td>2 6 10 14</td>
<td>3 7 11 15</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. tree)
    - Tasks generated dynamically, as we work
  - Unknown size of tasks
    - Data-dependent execution time (e.g. number of edges per node in a graph)
    - 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`

---

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

Thread ID

Iteration Number

0 50 100 150 200 250 300 350 400 450 500

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
  - Easy to create false sharing

• Compilers with OpenMP 2.5 support are widely available

• OpenMP 3.0 supports tasking
  - Supports irregular parallelism