Parallel Tuning

Beth Richardson
NCSA Consultant and Trainer
bethr@ncsa.uiuc.edu

Outline for the Workshop

1. Port your code
2. Tune the code to improve scalar performance
3. Parallelize the code
4. Tune the code to improve parallel performance

We will cover 4. In this lecture.
Limitations on Parallel Speedup

- The code is I/O bound
- There is network saturation
- The problem size is fixed
- The problem size is too small
- There is too much scalar code
- The computations are order dependent
- Need to change to a parallel algorithm
- Too much parallel overhead
- There is load imbalance
- There are memory bottlenecks

What is Parallel Overhead

Parallel overhead is the processing time spent in:

- creating threads
- spin/blocking threads
- starting/ending parallel regions
- synchronization at the end of parallel regions
- extra code added for parallelization
**Measuring Parallel Overhead**

**Step One**
- Time the serial run

**Step Two**
- Parallelize the code and time it using only one thread

**Step Three**
- Take the difference of the two timings.
- It gives a rough underestimate of parallel overhead.

---

**Reducing Parallel Overhead**

Don’t parallelize ALL the loops

Don’t parallelize small loops

- Need about 1000 floating point operations or 500 statements in the loop
- Use the "if" modifier in the c$doacross directive
  
  c$doacross if(n > 500), local(...), share(...)  
  do i=1,n  
  ... body of loop ...  
  enddo
Reducing Parallel Overhead (Cont.)

Use task parallelism instead of data parallelism
doesn’t generate as much parallel overhead
often more code runs in parallel
Don’t use more threads than you need

Measuring Load Balance

Profile the parallelized code using software tools like pixie and prof
the output shows the number of cpu cycles consumed by each thread

Compare the cycle counts
the master thread (thread 0) always uses more cycles than the slave threads
if the counts are vastly different this indicates load imbalance problems
Improving Load Balance

Try changing the way that loop iterations are allocated to threads
   by changing the schedule type
   by changing the chunk size

Loop Scheduling Types

On the Power Challenge Array and Origin 2000 there are four different schedule types
   SIMPLE  INTERLEAVE  DYNAMIC  GSS
If you don’t specify a schedule type, then the SIMPLE schedule type will be used
Simple Schedule Type

How to Specify

At runtime
setenv MP_SCHEDTYPE SIMPLE

At compile time
c$doacross mp_schedtype=simple

Simple Schedule Type (Cont.)

How it Works

This schedule type allocates 20 iterations on 4 threads as:

```
11111111112
iteration no. 12345678901234567890
--------------------------------------------------------
thread no. 0000011111222223333
```
Interleave Schedule Type

How to Specify

At runtime
setenv MP_SCHEDTYPE INTERLEAVE

At compile time
c$doacross mp_schedtype=interleave

Interleave Schedule Type (Cont.)

How it Works

This schedule type allocates 20 iterations on 4 threads as:

<table>
<thead>
<tr>
<th>Iteration no.</th>
<th>12345678901234567890</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread no.</td>
<td>01230123012301230123</td>
</tr>
</tbody>
</table>
Interleave Schedule Type (Cont.)

Why it’s used
- It’s used when some of the iterations do more work than others
- With “interleave”, iterations are allocated in a round-robin fashion to the threads

Dynamic Schedule Type

How to Specify
At runtime
  setenv MP_SCHEDTYPE DYNAMIC
At compile time
  c$doacross mp_schedtype=dynamic

How it works
- The iterations are dynamically allocated to threads at runtime
Dynamic Schedule Type (Cont.)

Why It’s Used

It’s useful when you don’t know the iteration count or work pattern ahead of time.
Each thread is given a chunk of iterations. When a thread finishes its work, it goes into a critical section where it’s given another chunk to work on.

Positives and Negatives

This keeps all the threads busy
But at a much higher overhead cost

Guided Self Scheduling

This is dynamic scheduling that starts with large chunks of iterations and ends with small chunks of iterations
Specify it at runtime by using
setenv MP_SCHEDTYPE GSS
The word "chunk" refers to a grouping of iterations. Chunk size means how many iterations are in the grouping. The INTERLEAVE and DYNAMIC schedule types can be used with a chunk size. Specify the chunk size using

```
setenv CHUNK n
```

If no chunk size is specified, a chunk size of 1 will be used.

---

Example of using a chunk size:

```
setenv MP_SCHEDTYPE INTERLEAVE
setenv CHUNK 2
```

Then 20 iterations are allocated on 4 threads as:

<table>
<thead>
<tr>
<th>iteration no.</th>
<th>thread no.</th>
</tr>
</thead>
<tbody>
<tr>
<td>111111111112</td>
<td>00112233001122330011</td>
</tr>
<tr>
<td>12345678901234567890</td>
<td>111111111112</td>
</tr>
<tr>
<td>111111111112</td>
<td>00112233001122330011</td>
</tr>
<tr>
<td>12345678901234567890</td>
<td>111111111112</td>
</tr>
</tbody>
</table>

---

University of Illinois at Urbana-Champaign
Try changing the chunk size.
Chunk size divides loop into chunks of n iterations and distributes chunks to procs in a round-robin fashion
c$dir loop_parallel (chunk_size=4)