

Lecture 14

Performance

Programming with OpenMP

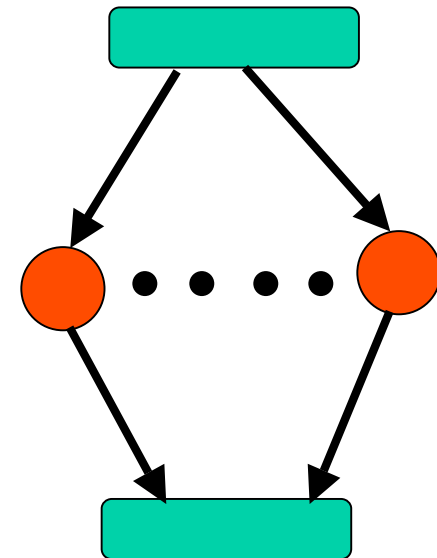
Announcements

- Sluggish front end nodes?
- Advice from NCSA
- Login nodes are honest[1-4].ncsa.uiuc.edu
- Login to specific node to target one that's not overloaded
- Don't rely on the round robin assignment

OpenMP programming

- Simpler interface than pthreads
- Parallelization handled via annotations
 - Loop decomposition
 - Serial sections
 - Barriers
- A standard
 - <http://www.openmp.org>
- Parallel loop:

```
#pragma omp parallel
{
  #pragma omp for private(i) shared(n)
  for(i=0; i < n; i++)
    work(i);
}
```



Documentation

- Documentation at software.intel.com/en-us/articles/
 - [getting-started-with-openmp](#)
 - [more-work-sharing-with-openmp](#)
 - [advanced-openmp-programming](#)
 - [32-openmp-traps-for-c-developers](#)
- Tutorial on OpenMP, with code examples
 - <https://computing.llnl.gov/tutorials/openMP/exercise.html>
- More updates on the Abe Document in the OpenMP section
<http://www.cse.ucsd.edu/classes/fa08/cse260/Abe.html#openmp>

Programming model

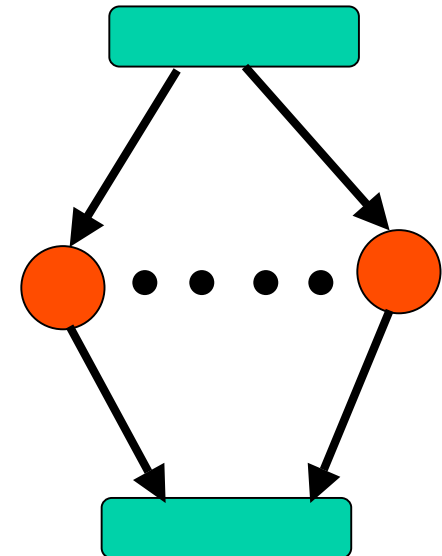
- Start with a single root thread
- Fan out a set of concurrently executing threads
- Threads may or may not execute on different processors, and might be interleaved
- Scheduling behavior specified separately

```
#pragma omp parallel // Begin parallel construct
```

```
{
```

```
    STATEMENTS
```

```
} // End of Parallel Construct; disband the team
```

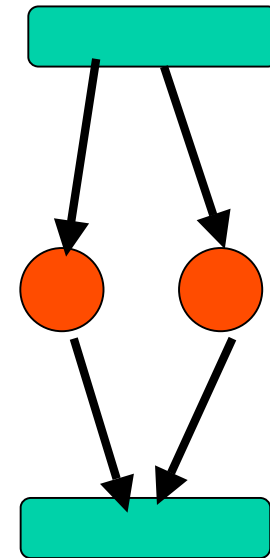


Parallel Sections

```
#pragma omp parallel // Begin a parallel construct
{ // form a team
  // Each team member executes the same code
  #pragma omp sections // Begin work sharing
  {
    #pragma omp section // A unit of work
    {x = x + 1;}

    #pragma omp section // Another unit
    {x = x + 1;}

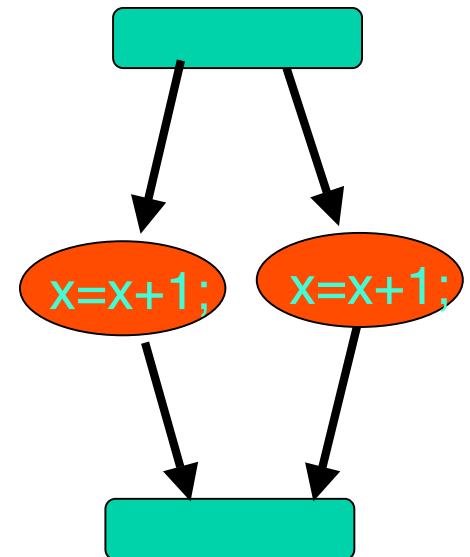
  } // Wait until both units complete
} // End of Parallel Construct; disband team
// continue serial execution
```



Critical Sections

- Only one thread at a time may run the code in a critical section
- *Mutual exclusion* to implement critical sections

```
#pragma omp parallel // Begin a parallel construct
{
  #pragma omp sections // Begin worksharing
  { //
    #pragma omp critical // Critical section
    {x = x + 1;}
    #pragma omp critical // Another critical section
    {x = x + 1;}
    ... // More Replicated Code
  }
  #pragma omp barrier // Wait for all members to arrive
  } // Wait until both units of work complete
}
```



Parallel for loop

```
#pragma omp parallel private(i) shared(n)
{
  #pragma omp for                for i = istart to end
  for(i=0; i < n; i++)
    work(i);
}
```

- **static workload assignments**
- Each thread gets a unique range of indices based on its identity
- The calls to `work(i)` take the same time to compute

Shorthand

```
#pragma omp parallel private(i) shared(n) for  
for(i=0; i < n; i++)  
    work(i);
```

Implementing the Game of Life with OpenMP

(See \$(PUB)/Examples/Life_omp)

Parallelization

```
#pragma omp parallel for shared(maxIter,population,gmesh,pmesh, nx,ny)
private(i,j)
for(i=1;i<nx-1;i++){
    for(j=1;j<ny-1;j++) {
        int nn = gmesh[i+1][j]+gmesh[i-1][j]+gmesh[i][j+1]+gmesh[i][j-1] +
            gmesh[i+1][j+1]+gmesh[i-1][j-1]+gmesh[i-1][j+1]+
            gmesh[i+1][j-1];
        pmesh[i][j] = gmesh[i][j] ? (nn == 2 || nn == 3) : (nn == 3);
    }
}
```

NT=1	2	4	8
Sp	1.95	3.89	5.98
1.10	0.565	0.283	0.219
1.10	0.582	0.283	0.184
1.10	0.584	0.305	0.208
1.10	0.585	0.283	0.338

Nested parallelization

```
#pragma omp parallel for shared(maxIter,population,gmesh,pmesh, nx,ny)
private(i,j) schedule(static)
for(i=1;i<nx-1;i++){

#pragma omp parallel for shared(maxIter,population,gmesh,pmesh, nx,ny)
private(i,j) schedule(static)
for(j=1;j<ny-1;j++) {
    int nn = gmesh[i+1][j]+gmesh[i-1][j]+gmesh[i][j+1]+gmesh[i][j-1] +
            gmesh[i+1][j+1]+gmesh[i-1][j-1]+gmesh[i-1][j+1]+
            gmesh[i+1][j-1];
    pmesh[i][j] = gmesh[i][j] ? (nn == 2 || nn == 3) : (nn == 3);
}
} Setenv OMP_NESTED 1
```

Moving between parallel and serial execution

```
#pragma omp parallel for shared(maxIter,population,gmesh,pmesh, nx,ny)
private(i,j) schedule(static)
for(i=1;i<nx-1;i++){
}
```

X=2

```
#pragma omp parallel for shared(maxIter,population,gmesh,pmesh, nx,ny)
private(i,j) schedule(static)
for(i=1;i<nx-1;i++){
}
```

How does it work under the hood?

- When the master thread encounters a parallel construct, it creates a team of threads
- The enclosed program statements enclosed executed in parallel by all team members, including procedure calls.
- Statements enclosed lexically within a construct define the static extent of the construct
- When we reach the end of the scope the team of threads synchronize, the team is dissolved, and only the master thread continues execution. The other threads in the team enter a wait state.
- Thread teams can be created and dissolved many times during program execution.

www.ncsa.uiuc.edu/UserInfo/Resources/Software/Intel/Compilers/10.0/main_cls/mergedProjects/optaps_cls/whskin_homepage.htm

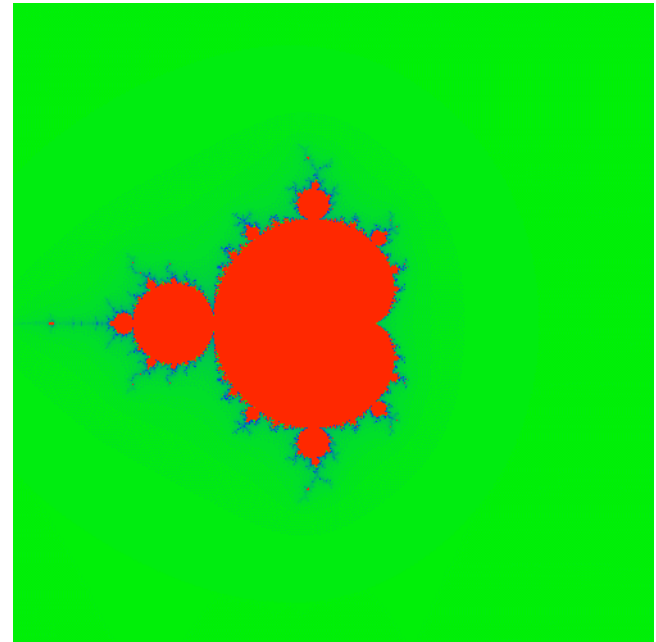
Reductions in OpenMP

- OpenMP uses a local accumulator, which it then accumulates globally when the loop is over

```
#pragma omp parallel reduction(+:sum)  
for (int i=i0; i< i1; i++)  
    sum += x[i];
```

Dynamic workload assignment

- Static decomposition assumes that all the iterations take the same time
- In some applications, this condition doesn't hold
- We can't use static decomposition
- Motivating application: Mandelbrot set computation
- Named after B. Mandelbrot



A quick review of complex numbers

- We define $\mathbf{i} = \sqrt{-1}$
- A *complex number* $z = x + \mathbf{i} y$
 - x is called the *real part*
 - y is called the *imaginary part*
- We associate each complex number with a point in the x - y plane
- The magnitude of a complex number is the same as vector length: $|z| = \sqrt{x^2 + y^2}$
- $z^2 = (x+\mathbf{i}y)(x+\mathbf{i}y) = (x^2 - y^2) + 2xy\mathbf{i}$

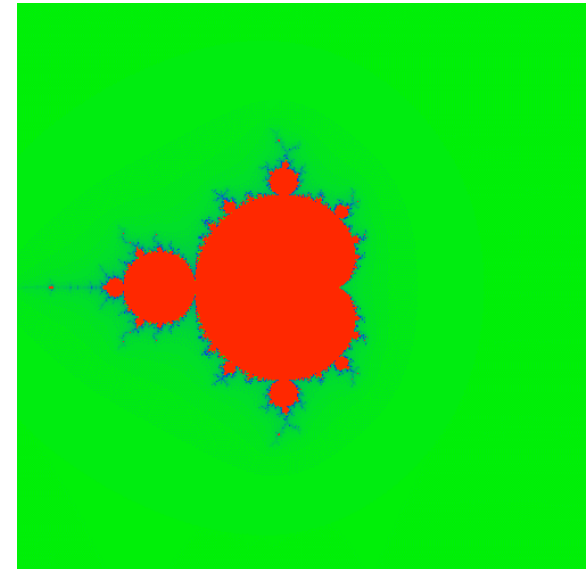
The Mandelbrot set

- For which points c in the complex plane does the following iteration remain bounded?

$$z_{k+1} = z_k^2 + c$$

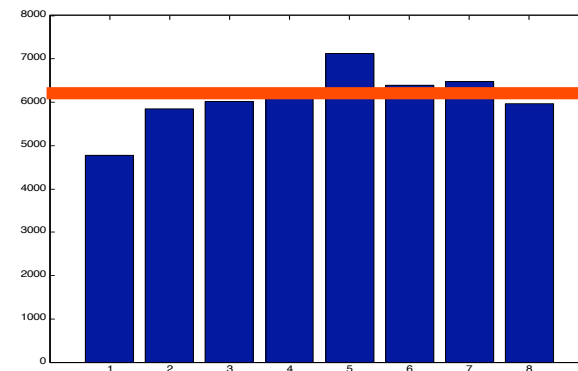
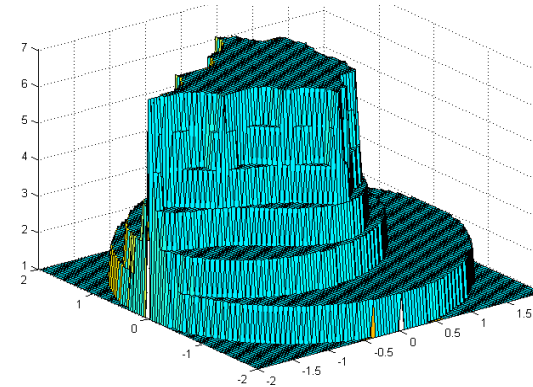
c is a complex number, $z_0 = 0$

- When $c=0$, all points lay within a unit disk : $|z| \leq 1$
- If $|z| \geq 2$, the iteration is guaranteed to diverge to ∞
- Plot the rate at which points in a given region diverge
- Stop the iterations when $|z_{k+1}| \geq 2$ or k reaches some limit
- Plot k at each position



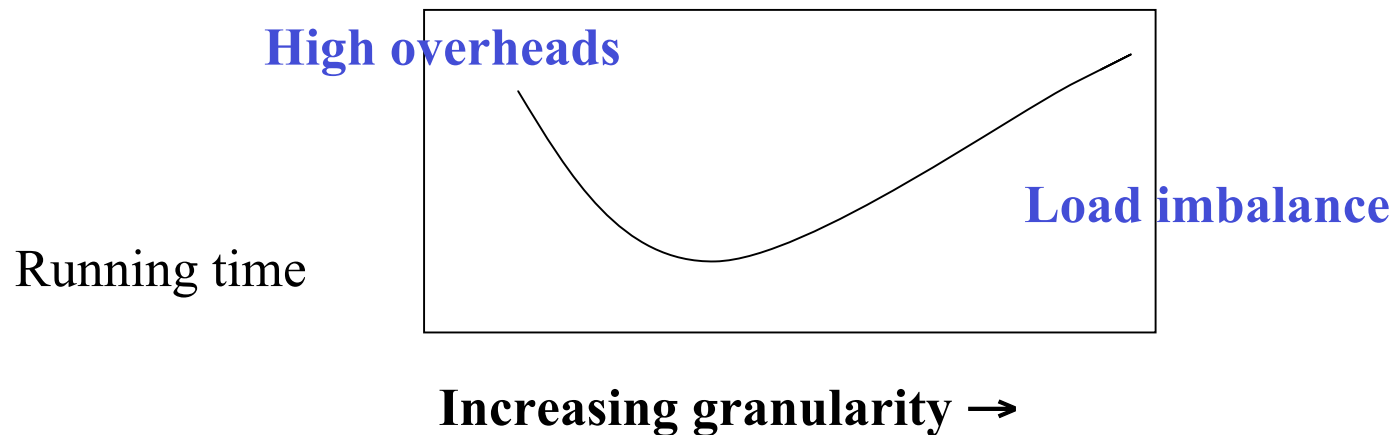
A load balancing problem

- Some points iterate **longer** than others
- If we use a uniform static decomposition, some processors finish later than others
- We have a *load imbalance*



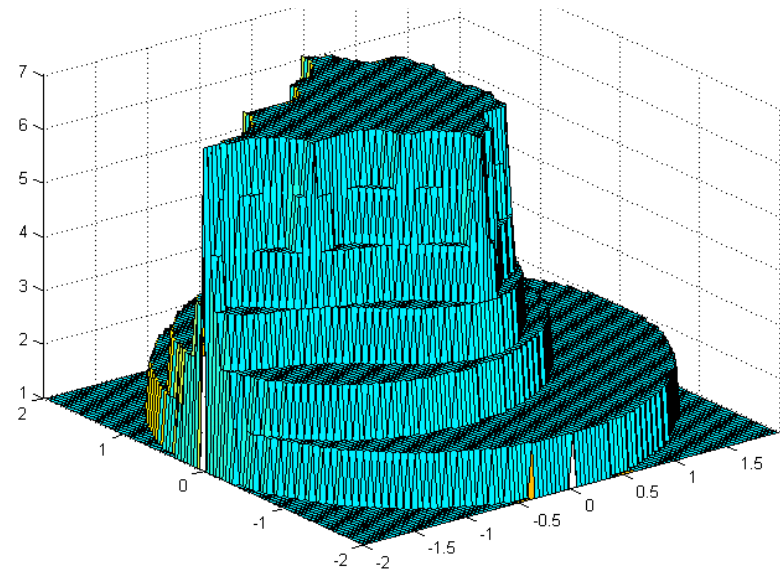
Dynamic shceduling

- OpenMP provides a **dynamic** scheduling option
- Processors “schedule” themselves
- Sample a shared counter or work queue to obtain work
- User tunes work granularity (“chunk size”) to trade off the overhead of sampling the queue against increased load imbalance



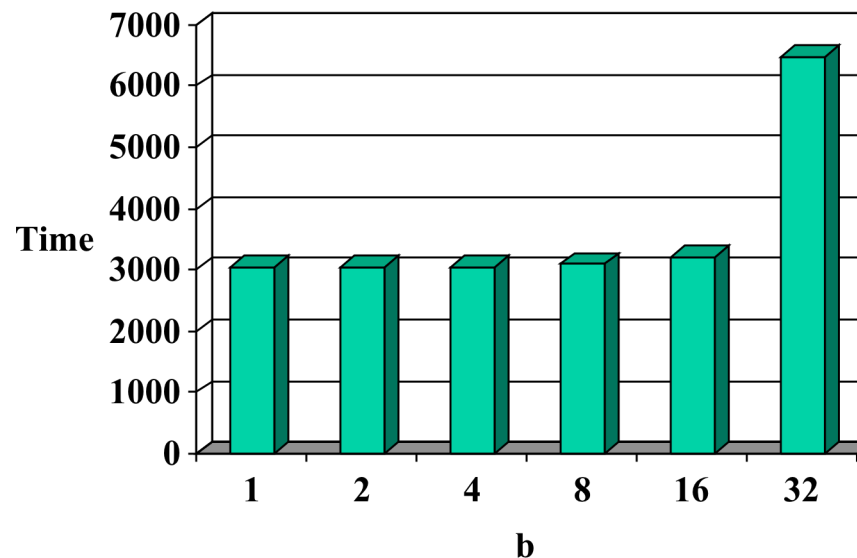
Granularity tradeoff

- We need to choose an appropriate task granularity
- The finest granularity: each point is a separate task
- Coarsest granularity: one block per processor
- What is the tradeoff?



Simulations

- Let's simulate the workload distribution
- Assumptions:
 - Work distribution is instantaneous (zero cost)
 - Running time is proportional to k
- Vary the chunk size b
- Optimal running time on 16 processors: 3044

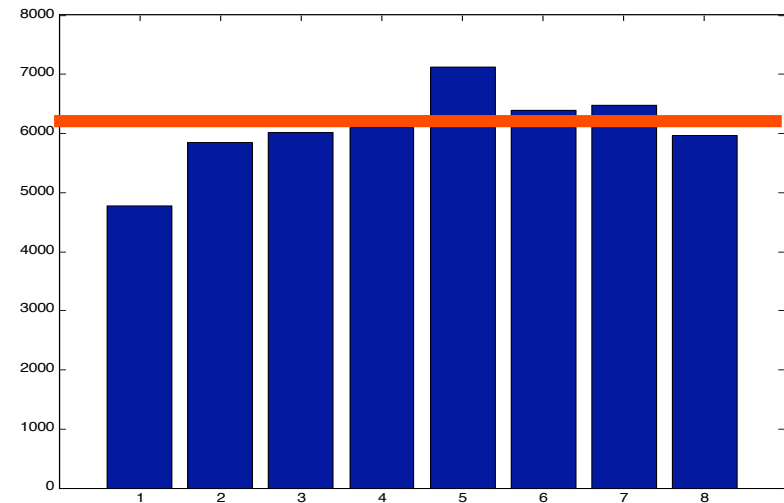


Limits to performance

- What happened when the chunk size was too large?
- Consider $P=8$
- The running times on the processors:

4771	5855	6018	6101
7129	6390	6470	5964

- The optimal running time is 6088



How to specify dynamic scheduling

```
#pragma omp parallel private(i,j) shared(unew, u, n)
  #pragma omp for schedule(dynamic)
  for(i=0; i < n; i++)
    for(j=0; j < n; j++) {
      unew[i,j] = Work(u[i,j])
    }
```

```
setenv OMP_DYNAMIC 1
```

How does self scheduling work?

```
while (getNextChunk(&mymin, &mymax ))
    for i = mymin to mymax
        work(i);
    end for
end while
```

```
SelfScheduler S(n,P,CSize);
```

```
Boolean getNextChunk(int * mymin, int * mymax ){
```

```
    $omp critical
```

```
    {
```

```
    k = S.counter += S.ChunkSize;
```

```
    }
```

```
    mymin = k;
```

```
    mymax = k + S.chunkSize;
```

```
}
```

Guided self-scheduling

- Observation: we need many more pieces of work than there are processors (say 10 times) at the time we are sampling the counter
- Adjust or “guide” task granularity in proportion to the amount of remaining work
- For example, $g = N_{\text{Remain}}/10P$
- When there’s lots of work to do, we hand out large chunks. Toward the end, we hand out smaller chunks
- See Polychronopoulos, C.D., and Kuck, D.J. “Guided self-scheduling: a practical scheduling scheme for parallel supercomputers.” *IEEE Trans, on Computers*, Vol. C-36, No. 12, Dec. 1987, pp. 1425-1439.