Embedded computing for scientific and industrial imaging applications

Lecture 16 - Fine grain vs coarse grain. Demo of OpenMP

Outline

- Fine grain vs. coarse grain parallelism
- Manually splitting loops between threads
- Examples with bugs

References:

- https://computing.llnl.gov/tutorials/openMP/

Fine vs. coarse grain parallelism

Fine grain: Parallelize at the level of individual loops, splitting work for each loop between threads.

Coarse grain: Split problem up into large pieces and have each thread deal with one piece.

May need to synchronize or share information at some points.

Fine vs. coarse grain parallelism

Fine grain: Parallelize at the level of individual loops, splitting work for each loop between threads.

Coarse grain: Split problem up into large pieces and have each thread deal with one piece.

May need to synchronize or share information at some points.
More similar to what must be done in MPI.

Domain Decomposition: Splitting up a problem on a large domain (e.g. three dimensional grid) into pieces that are handled separated (with suitable coupling).

Solution of independent ODEs by Euler's method

Solve $u_i'(t) = c_i u_i(t)$ for $t \geq 0$

with initial condition $u_i(0) = \eta_i$. Decoupled system of ODEs for $i=1, 2, ..., n$

Solution of independent ODEs by Euler's method

Solve $u_i'(t) = c_i u_i(t)$ for $t \geq 0$

with initial condition $u_i(0) = \eta_i$. Decoupled system of ODEs for $i=1, 2, \ldots, n$

Exact solution: $u_i(t) = e^{c_i t}\eta_i$

Euler method: $u_i(t + \Delta t) \approx u_i(t) + \Delta t c_i u_i(t) = (1 + c_i \Delta t) u_i(t)$

Implement this for large number of time steps for large n.

For each i time stepping can't be easily made parallel.

But for large n, this problem is embarrassingly parallel:

Problem for each i is completely decoupled from problem for any other i. Could solve them all simultaneously with no communication needed.

Fine grain solution with parallel for loops

```
#pragma omp parallel for
for (i=0; i < n; i++)
    u[i] = eta[i];
for (m=0; m < nsteps; m++)
{
    #pragma omp parallel for
    for (i=0; i < n; i++)
        u[i] = (1.0 + dt * c[i]) * u[i];
}
```

Note that threads are forked `nsteps+1` times.

Requires shared memory:

    don’t know which thread will handle each `i`.

Fine grain solution with parallel for loops

Might try to fork threads only once via: Wrong!

```
#pragma omp parallel private(m)
{
     #pragma omp for
     for(i=0; i < n; i++)
          u[i] = eta[i];

     for(m=0; m < nsteps; m++)
     {
          #pragme omp for
          for(i=0; i < n; i++)
               u[i] = (1.d0 + dt * c[i]) * u[i]
     }
}
```

Error: the loop on `m` will be done independently by each thread.

(Actually works in this case but not good coding.)

Fine grain solution with parallel for loops

Can rearrange loops:

```
#pragma omp parallel private(m)
{
     #pragma omp for
     for(i=0; i < n; i++)
          u[i] = eta[i];

     #pragma omp for
     for(m=0; m < nsteps; m++)
     {
          for(i=0; i < n; i++)
               u[i] = (1.d0 + dt * c[i]) * u[i]
     }
}
```

Only works because ODEs are decoupled — can take all time steps on $u_1(t)$ without interacting with $u_2(t)$, for example.

Coarse grain solution of ODEs

Split up $i = 1, 2, \ldots, n$ into `nthreads` disjoint sets.

A set goes from `i=istart` to `i=iend-1`

These private values are different for each thread.

Each thread handles 1 set for the entire problem.

```
#pragma omp parallel private(istart,iend,i,m)
{
     istart = ??
     iend = ??
     for(i=istart; i < iend; i++)
          u[i] = eta[i];
     for(m=0; m < nsteps; m++)
          for(i=istart; i < iend; i++)
               u[i] = (1.d0 + dt * c[i]) * u[i];
}
```

Threads are forked only once,

Each thread only needs subset of data.

Setting `istart` and `iend`

Example: If `n=100` and `nthreads = 2`, we would want:

Thread 0: `istart=0` and `iend=49`,

Thread 1: `istart=50` and `iend=99`.

If `nthreads` divides `n` evenly...

```
points_per_thread = n / nthreads;
#pragma omp parallel private(thread_num, istart, iend, i)
{
     thread_num = 0; //needed in serial mode
     thread_num = omp_get_thread_num();
     istart = thread_num * points_per_thread;
     iend = (thread_num+1) * points_per_thread;
     for(i=istart; i < iend; i++)
     {
          // work on thread’s part of array
     }
}
```

Setting `istart` and `iend` more generally

Example: If `n=101` and `nthreads = 2`, we would want:

  Thread 0: `istart=0` and `iend=50`,

  Thread 1: `istart=51` and `iend=100`.

If `nthreads` might not divide `n` evenly…

```
points_per_thread = (n + nthreads - 1) / nthreads;
#pragma omp parallel private(thread_num, istart, iend, i)
{
     thread_num = 0; //needed in serial mode
     thread_num = omp_get_thread_num();
     istart = thread_num * points_per_thread;
     iend = min((thread_num+1)*points_per_thread, n);
     for(i=istart; i < iend; i++)
     {
          // work on thread's part of array
     }
}
```

Example: Normalizing a vector

Given a vector (1-dimensional array) x,
Compute the normalized vector $x/\|x\|_1$, with $\|x\|_1 = \sum_{i=1}^{n} |x_i|$

Fine-grain: Using `parallel for` loops.

```
norm = 0.0;
#pragma omp parallel for reduction(+ : norm)
for(i=0; i < n; i++)
{
     norm = norm + fabs(x[i]);
}
#pragma omp parallel for
for(i=0; i < n; i++)
     x[i] = x[i] / norm;
```

Note: Must finish computing `norm` before using for any `x[i]`,
so we are using the implicit barrier after the first loop.

Example: Normalizing a vector

Another fine-grain approach, forking threads only once:

from `$CSE6000/codes/OpenMP/Examples/normalize1/normalize1.c`

```
norm = 0.0;
#pragma omp parallel private(i)
{
    #pragma omp for reduction(+ : norm)
    for(i=0; i < n; i++)
        norm = norm + fabs(x[i]);

    #pragma omp barrier //not needed (implicit)
    #pramga omp for
    for(i=0; i < n; i++)
        x[i] = x[i] / norm;
}
```

Example: Normalizing a vector

Compute the normalized vector $x/\|x\|_1$, with $\|x\|_1 = \sum_{i=1}^{n} |x_i|$

Coarse grain version:

Assign blocks of i values to each thread. Threads must:

- Compute thread's contribution to $\|x\|_1$,

$$\text{norm\_thread} = \sum_{\text{istart}}^{\text{iend}} |x_i|\,,$$

- Collaborate to compute total value $\|x\|_1$:

$$\|x\|_1 = \sum_{\text{threads}} \text{norm\_thread}$$

- Loop over `i = istart, iend` to divide x_i by $\|x\|_1$.

Example: Normalizing a vector

from $CSE6000/codes/OpenMP/Examples/normalize2/normalize2.c

```
norm = 0.0;
#pragma omp parallel private(i,norm_thread, istart,iend,thread_num)
{
     thread_num = omp_get_thread_num();
     istart = thread_num * points_per_thread;
     iend = min((thread_num+1) * points_per_thread, n);
     norm_thread = 0.0;
     for(i=istart; i < iend; i++)
          norm_thread = norm_thread + fabs(x[i]);
     //update global norm with value from each thread:
     #pragma omp critical
          norm = norm + norm_thread;
     #pragma omp barrier // needed here
     for(i=istart; i < iend; i++)
          y[i] = x[i] / norm;
}
```

Example: Normalizing a vector — parallel block

```
norm_thread = 0.0;
for(i=istart; i < iend; i++)
    norm_thread = norm_thread + fabs(x[i]);

//update global norm with value from each thread:
#pragma omp critical
    norm = norm + norm_thread;

#pragma omp barrier // needed here
for(i=istart; i < iend; i++)
    y[i] = x[i] / norm;
```

Normalizing a vector — possible bugs

1. Not declaring proper variables private
2. Setting `norm = 0.0;` inside parallel block.
   Ok if it's in a `omp single` block. Otherwise second thread might set to zero after first thread has updated by `norm_thread`.
3. Not using `omp critical` block to update global `norm`.
   Data race.
4. Not having a `barrier` between updating norm and using it.
   First thread may use `norm` before other threads have added their contributions.

None of these bugs would give compile or runtime errors!
Just wrong results (sometimes).

OpenMP example with shared exit criterion

Solve $u_i'(t) = c_i u_i(t)$ for $t \geq 0$

with initial condition $u_i(0) = \eta_i$

Exact solution: $u_i(t) = e^{c_i t}\eta_i$

Euler method: $u_i(t + \Delta t) \approx u_i(t) + \Delta t c_i u_i(t) = (1 + c_i \Delta t) u_i(t)$

New wrinkle: Stop time stepping when any of the $u_i(t)$ values exceeds 100.

(Will certainly happen as long as $c_j > 0$ for some j.)

OpenMP example with shared exit criterion

Stop time stepping when any of the $u_i(t)$ values exceeds 100.

Idea:

Each time step, compute `umax` $=$ maximum value of u_i over all i and exit the time-stepping if `umax > 100`.

u_i

OpenMP example with shared exit criterion

Stop time stepping when any of the $u_i(t)$ values exceeds 100.

Idea:

Each time step, compute `umax` $=$ maximum value of u_i over all i and exit the time-stepping if `umax > 100`.

Each thread has a private variable `umax_thread` for the maximum value of u_i for its values of `i`. Updated for each `i`.

OpenMP example with shared exit criterion

Stop time stepping when any of the $u_i(t)$ values exceeds 100.

Idea:

Each time step, compute `umax` $=$ maximum value of u_i over all i and exit the time-stepping if `umax > 100`.

Each thread has a private variable `umax_thread` for the maximum value of u_i for its values of `i`. Updated for each `i`.

Each thread updates shared `umax` based on its `umax_thread`.

    This needs to be done in critical section.

OpenMP example with shared exit criterion

Stop time stepping when any of the $u_i(t)$ values exceeds 100.

Idea:

Each time step, compute `umax` $=$ maximum value of u_i over all i and exit the time-stepping if `umax > 100`.

Each thread has a private variable `umax_thread` for the maximum value of u_i for its values of `i`. Updated for each `i`.

Each thread updates shared `umax` based on its `umax_thread`.

  This needs to be done in critical section.

Also need two barriers to make sure all threads are in synch at certain points.

Study code in `$CSE6000/codes/OpenMP/Examples/umax1/umax1.c`.

OpenMP example with shared exit criterion

```
#pragma omp parallel private(i, m, umax_thread, istart, iend, thread_num)
{
        thread_num = omp_get_thread_num();
        istart = thread_num * points_per_thread;
        iend = min((thread_num+1) * points_per_thread, n);
        for(m=0; m < nsteps; m++)
        {
                umax_thread = 0.0;
                #pragma omp single
                        umax = 0.0;
                for(i=istart; i < iend; i++)
                {
                        u[i] = (1.0 + c[i] * dt) * u[i];
                        umax_thread = max(umax_thread, u[i]);
                }
                #pragma omp critical
                        umax = max(umax, umax_thread);
                #pragma omp barrier
                if (umax > 100)
                        break;
                #pragma omp barrier
        }
}
```

do loop in parallel block:

```
for(m=0; m < nsteps; m++)
{
     umax_thread = 0.0;
     #pragma omp single
          umax = 0.0;
     for(i=istart; i < iend; i++)
     {
          u[i] = (1.0 + c[i] * dt) * u[i];
          umax_thread = max(umax_thread, u[i]);
     }
     #pragma omp critical
          umax = max(umax, umax_thread);
     #pragma omp barrier
     if (umax > 100)
          break;
     #pragma omp barrier
}
```

OpenMP example with shared exit criterion

- If there were no barriers, the following could happen:
  Thread 0 executes critical section first, setting umax to 0.5.
  Thread 0 checks if umax > 100. False, starts next iteration.
  Thread 1 executes critical section, updating umax to 110.
  Thread 1 checks if umax > 100. True, so it exits.
  Thread 0 next sets umax to 0.4.
  Thread 0 might never reach umax > 100. Runs forever.

OpenMP example with shared exit criterion

- If there were no barriers, the following could happen:
  Thread 0 executes critical section first, setting umax to 0.5.
  Thread 0 checks if umax > 100. False, starts next iteration.
  Thread 1 executes critical section, updating umax to 110.
  Thread 1 checks if umax > 100. True, so it exits.
  Thread 0 next sets umax to 0.4.
  Thread 0 might never reach umax > 100. Runs forever.
- With only first barrier, the following could happen:
  umax < 100 in iteration m.
  Thread 1 checks if umax > 100. Go to iteration m + 1.
  Thread 1 does iteration on i and sets umax > 100,
  Stops at first barrier.
  Thread 0 (iteration m) checks if umax > 100. True, Exits.
  Thread 0 never reaches first barrier again, code hangs.