CS 475/575 -- Spring Quarter 2015

Project #2

OpenMP: N-body Problem -- Coarse vs Fine and Static vs Dynamic

100 Points

Due: April 27

This page was last updated: March 29, 2015

Introduction

This project involves a rumble between static scheduling vs. dynamic, and coarse-grained parallelism vs. fine-grained.

The problem that we are solving is an "N-Body Problem", in which a group of planetary masses are swarming around by being mutually attracted to each other. As all bodies are attracted to all other bodies, this is potentiially an O(N^2) problem, and thus would be ripe for parallelism.

Requirements

- Use OpenMP for this. Use at least 100 bodies. Take at least 200 time steps.
- Use a variety of different numbers of threads. At least use 1, 2, and 4. You can also use more if you'd like.
- In the code below, "coarse-grained parallelism" means putting the OpenMP `#pragma omp parallel for` before the *i* for-loop. "fine-

grained parallelism" means putting it before the *j* for-loop.

- When you do the fine-grained parallelism, don't forget that the variables `fx, fy, fz` need to undergo a reduction-add.

- You can control static vs. dynamic scheduling by adding a clause to the end of the `#pragma omp parallel for` line. Use either `schedule(static)` or `schedule(dynamic)`.

- Don't worry about the scheduling chunksize. Let it default to 1. (Joe Parallel tried a few combinations and it didn't seem to make any difference. This makes sense since the passes through the j-loop are equally time-sonsuming.)

- Record the data in units of something that gets larger as speed increases. Joe Parallel used "MegaBodies Compared Per Second" (`(float)(NUMBODIES*NUMBODIES*NUMSTEPS)/(time1-time0)/1000000.`), but you can use anything that makes sense.

- Your commentary write-up (turned in as a PDF file) should include:

  1. Tell what machine you ran this on
  2. Create a table with your results.
  3. Draw a graph. The X axis will be the number of threads. The Y axis will be the performance in whatever units you sensibly choose. On the same graph, plot 4 curves:
     1. coarse+static
     2. coarse+dynamic
     3. fine+static
     4. fine+dynamic
  4. What patterns are you seeing in the speeds?
  5. Why do you think it is behaving this way?

The Skeleton Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

// constants:

const double G                  = 6.67300e-11;  // m^3 / ( kg s^2 )
const double EARTH_MASS         = 5.9742e24;    // kg
const double EARTH_DIAMETER     = 12756000.32;  // meters
const double TIMESTEP           =   1.0;        // secs
```

```c
#define NUMBODIES        100
#define NUMSTEPS         200
#ifndef NUMT
#define NUMT      4
#endif

struct body
{
        float mass;
        float x, y, z;          // position
        float vx, vy, vz;       // velocity
        float fx, fy, fz;       // forces
        float xnew, ynew, znew;
        float vxnew, vynew, vznew;
};

typedef struct body Body;


Body    Bodies[NUMBODIES];


// function prototypes:

float           GetDistanceSquared( Body *, Body * );
float           GetUnitVector( Body *, Body *, float *, float *, float * );
float           Ranf( float, float );
int             Ranf( int, int );


int
main( int argc, char *argv[ ] )
{
#ifndef _OPENMP
        fprintf( stderr, "OpenMP is not available\n" );
        return 1;
#endif

        omp_set_num_threads( NUMT );
        int numProcessors = omp_get_num_procs( );
        fprintf( stderr, "Have %d processors.\n", numProcessors );


        for( int i = 0; i < NUMBODIES; i++ )
        {
                Bodies[i].mass = EARTH_MASS  * Ranf( 0.5f, 10.f );
```

```
		Bodies[i].x = EARTH_DIAMETER * Ranf( -100.f, 100.f );
		Bodies[i].y = EARTH_DIAMETER * Ranf( -100.f, 100.f );
		Bodies[i].z = EARTH_DIAMETER * Ranf( -100.f, 100.f );
		Bodies[i].vx = Ranf( -100.f, 100.f );;
		Bodies[i].vy = Ranf( -100.f, 100.f );;
		Bodies[i].vz = Ranf( -100.f, 100.f );;
	};

	double time0 = omp_get_wtime( );

	for( int t = 0; t < NUMSTEPS; t++ )
	{
		for( int i = 0; i < NUMBODIES; i++ )
		{
			float fx = 0.;
			float fy = 0.;
			float fz = 0.;
			Body *bi = &Bodies[i];
			for( int j = 0; j < NUMBODIES; j++ )
			{
				if( j == i )	continue;

				Body *bj = &Bodies[j];

				float rsqd = GetDistanceSquared( bi, bj );
				if( rsqd > 0. )
				{
					float f = G * bi->mass * bj->mass / rsqd;
					float ux, uy, uz;
					GetUnitVector( bi, bj,   &ux, &uy, &uz );
					fx += f * ux;
					fy += f * uy;
					fz += f * uz;
				}
			}

			float ax = fx / Bodies[i].mass;
			float ay = fy / Bodies[i].mass;
			float az = fz / Bodies[i].mass;

			Bodies[i].xnew = Bodies[i].x + Bodies[i].vx*TIMESTEP + 0.5*ax*TIMESTEP*TIMESTEP;
			Bodies[i].ynew = Bodies[i].y + Bodies[i].vy*TIMESTEP + 0.5*ay*TIMESTEP*TIMESTEP;
			Bodies[i].znew = Bodies[i].z + Bodies[i].vz*TIMESTEP + 0.5*az*TIMESTEP*TIMESTEP;

			Bodies[i].vxnew = Bodies[i].vx + ax*TIMESTEP;
			Bodies[i].vynew = Bodies[i].vy + ay*TIMESTEP;
```

```
			Bodies[i].vznew = Bodies[i].vz + az*TIMESTEP;
		}

		// setup the state for the next animation step:

		for( int i = 0; i < NUMBODIES; i++ )
		{
			Bodies[i].x = Bodies[i].xnew;
			Bodies[i].y = Bodies[i].ynew;
			Bodies[i].z = Bodies[i].znew;
			Bodies[i].vx = Bodies[i].vxnew;
			Bodies[i].vy = Bodies[i].vynew;
			Bodies[i].vz = Bodies[i].vznew;
		}


	}  // t

	double time1 = omp_get_wtime( );

	// print performance here:::

	return 0;
}


float
GetDistanceSquared( Body *bi, Body *bj )
{
	float dx = bi->x - bj->x;
	float dy = bi->y - bj->y;
	float dz = bi->z - bj->z;
	return dx*dx + dy*dy + dz*dz;
}


float
GetUnitVector( Body *from, Body *to, float *ux, float *uy, float *uz )
{
	float dx = to->x - from->x;
	float dy = to->y - from->y;
	float dz = to->z - from->z;

	float d = sqrt( dx*dx + dy*dy + dz*dz );
	if( d > 0. )
	{
```

```
                dx /= d;
                dy /= d;
                dz /= d;
        }

        *ux = dx;
        *uy = dy;
        *uz = dz;

        return d;
}


float
Ranf( float low, float high )
{
        float r = (float) rand();               // 0 - RAND_MAX

        return(   low  +  r * ( high - low ) / (float)RAND_MAX   );
}


int
Ranf( int ilow, int ihigh )
{
        float low = (float)ilow;
        float high = (float)ihigh + 0.9999f;

        return (int)(  Ranf(low,high) );
}
```

Where Did This Project Come From?

This project was inspired by the colliding galaxies scene from the IMAX movie *Cosmic Voyage*. It involved a 165GB dataset and thousands of hours of computer time to simulate. You can see this scene by going to: http://www.youtube.com/watch?v=Jrrm4F2IJMc .

(Don't worry about trying to make a real animation out of this assignment. We would probably need to pay much closer attention to the program's parameters to make this happen correctly.)

Grading:

Feature	Points
Table of Results	30
Graph of Results	30
Commentary	40
Potential Total	**100**