Trabalho Programação Distribuída e Paralela
Problema N-Body

Cleiber Gustavo Soares Rodrigues Jr.
Jordi Pujol Ricarte

O problema

- Problema da maratona de programação paralela de 2015;
- Calcular a interação gravitacional entre partículas em um determinado número de iterações.
- Problema altamente paralelizável.

N-Body

In physics, the N-Body problem consists in simulating the gravitational interaction between *N* particles (bodies) in a system and predicting how the system would evolve in a time frame. In this application the initial position and mass for each particle is randomly generated. The application will compute gravitational forces, positions and velocity of each particle in each time step of the simulation.

We are not interested in finding out different algorithms for computing the N-Body simulation. We just want to focus on obtaining a parallel version of the given code. Therefore, it is not allowed to change the computation method used in this problem.

O algoritmo

- O algoritmo se consiste em duas etapas: Criação e Simulação.
- Durante a criação, geramos aleatoriamente N partículas com uma posição XYZ e uma massa;
- Simulação em 2 etapas, repetidas M vezes:
  - Cálculo das forças;
  - Atualização das posições.

O algoritmo - Cálculo das forças

- Para cada partícula, calculamos a interação desta com as demais;
- Não atualizamos os valores de posição, e não lemos valores de força;
- Retornamos a maior força para controle.

```
double ComputeForces( Particle myparticles[], Particle others[], ParticleV pv[], int npart )
{
  double max_f;
  int i;
  max_f = 0.0;
  for (i=0; i<npart; i++) {
    int j;
    double xi, yi, mi, rx, ry, mj, r, fx, fy, rmin;
    rmin = 100.0;
    xi   = myparticles[i].x;
    yi   = myparticles[i].y;
    fx   = 0.0;
    fy   = 0.0;
    for (j=0; j<npart; j++) {
      rx = xi - others[j].x;
      ry = yi - others[j].y;
      mj = others[j].mass;
      r  = rx * rx + ry * ry;
      /* ignore overlap and same particle */
      if (r == 0.0) continue;
      if (r < rmin) rmin = r;
      r  = r * sqrt(r);
      fx -= mj * rx / r;
      fy -= mj * ry / r;
    }
    pv[i].fx += fx;
    pv[i].fy += fy;
    fx = sqrt(fx*fx + fy*fy)/rmin;
    if (fx > max_f) max_f = fx;
  }
  return max_f;
}
```

O algoritmo - Atualização das Posições

- As posições são atualizadas de acordo com as forças anteriormente calculadas;
- Cada iteração utiliza apenas os valores de uma partícula.

```
double ComputeNewPos( Particle particles[], ParticleV pv[], int npart, double max_f)
{
  int i;
  double a0, a1, a2;
  static double dt_old = 0.001, dt = 0.001;
  double dt_new;
  a0   = 2.0 / (dt * (dt + dt_old));
  a2   = 2.0 / (dt_old * (dt + dt_old));
  a1   = -(a0 + a2);
  for (i=0; i<npart; i++) {
    double xi, yi;
    xi             = particles[i].x;
    yi             = particles[i].y;
    particles[i].x = (pv[i].fx - a1 * xi - a2 * pv[i].xold) / a0;
    particles[i].y = (pv[i].fy - a1 * yi - a2 * pv[i].yold) / a0;
    pv[i].xold     = xi;
    pv[i].yold     = yi;
    pv[i].fx       = 0;
    pv[i].fy       = 0;
  }
  dt_new = 1.0/sqrt(max_f);
  /* Set a minimum: */
  if (dt_new < 1.0e-6) dt_new = 1.0e-6;
  /* Modify time step */
  if (dt_new < dt) {
    dt_old = dt;
    dt     = dt_new;
  }
  else if (dt_new > 4.0 * dt) {
    dt_old = dt;
    dt    *= 2.0;
  }
  return dt_old;
}
```

O algoritmo - Iteração principal

```
/* Generate the initial values */
    InitParticles( particles, pv, npart);
    sim_t = 0.0;

    while (cnt--) {
      double max_f;
      /* Compute forces (2D only) */
      max_f = ComputeForces( particles, particles, pv, npart );
      /* Once we have the forces, we compute the changes in position */
      sim_t += ComputeNewPos( particles, pv, npart, max_f);
    }
    for (i=0; i<npart; i++)
      fprintf(stdout,"%.5lf %.5lf %.5lf\n", particles[i].x, particles[i].y, particles[i].z);
    return 0;
```

Paralelizando o problema

- O problema tem dois pontos principais para paralelizar:
  - O cálculo da força;
  - A atualização das posições;
- Em cada uma dessas etapas, as partículas podem ter seus valores calculados individualmente;
- Escolhemos uma decomposição de Domínio em Blocos unidimensionais;
- Apenas um ponto de comunicação entre Threads.

O código paralelizado

```
double ComputeForces( Particle myparticles[], Particle others[], ParticleV pv[], int npart )
{
  double max_f;
  int i;
  max_f = 0.0;

  #pragma omp parallel for reduction(max:max_f) schedule(static)
  for (i=0; i<npart; i++) {
    int j;
    double xi, yi, mi, rx, ry, mj, r, fx, fy, rmin;
    rmin = 100.0;
```

O código paralelizado

```
double ComputeNewPos( Particle particles[], ParticleV pv[], int npart, double max_f)
{
  int i;
  double a0, a1, a2;
  static double dt_old = 0.001, dt = 0.001;
  double dt_new;
  a0   = 2.0 / (dt * (dt + dt_old));
  a2   = 2.0 / (dt_old * (dt + dt_old));
  a1   = -(a0 + a2);

  #pragma omp parallel for schedule(static)
  for (i=0; i<npart; i++) {
    double xi, yi;
    xi             = particles[i].x;
```

A solução da IA

- Utilizamos o Bard para gerar um código paralelo;
- Utilizamos a sentença: “Paralelizar esse código em C usando OpenMP onde você acredita ser seguro fazê-lo”;
- O código gerado foi basicamente o mesmo, mas sem definir o escalonamento das threads;
- Tivemos que fazer alterações no código para poder compilar essa solução.

A solução da IA

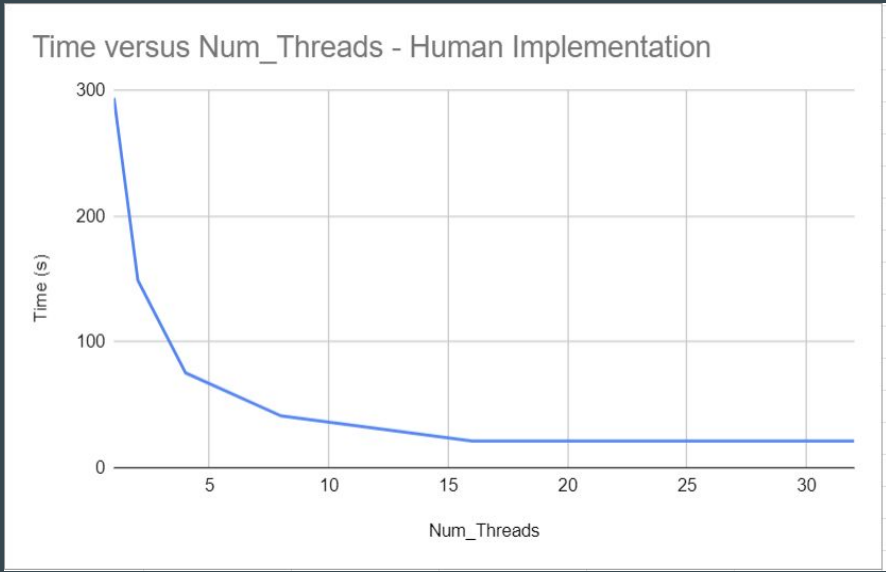
```
#pragma omp parallel for private(rx, ry, mj, r, fx, fy)
for (i=0; i<npart; i++) {
    int j;
    double xi, yi, mi, rx, ry, mj, r, fx, fy, rmin;
    rmin = 100.0;
    xi   = myparticles[i].x;
    yi   = myparticles[i].y;
    fx   = 0.0;
```

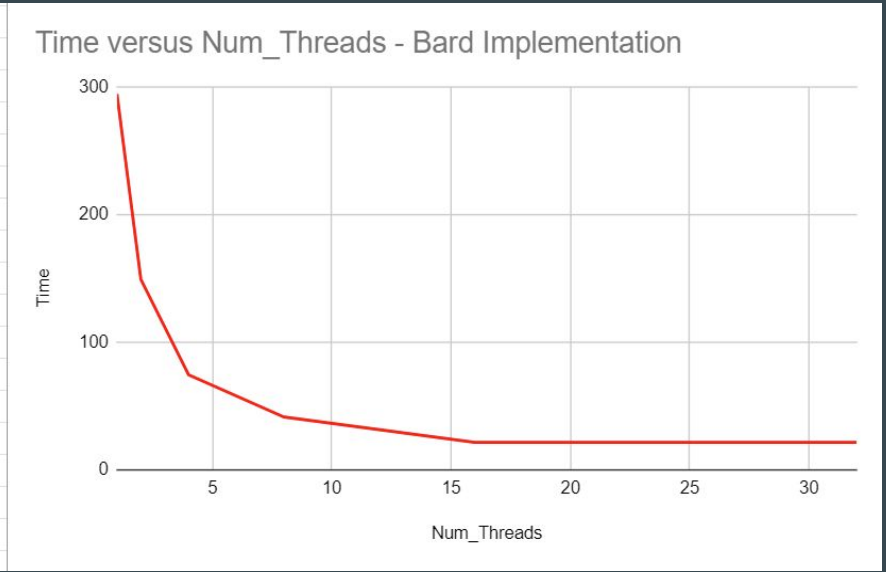
```
#pragma omp parallel for private(xi, yi)
for (i=0; i<npart; i++) {
    double xi, yi;
    xi              = particles[i].x;
    yi              = particles[i].y;
```

Resultados - 25000 Particulas

Human Results			Bard Results			Diff	
Num_Threads	Time(s) - Human		Num_Threads	Time(s) - Bard		Num_Threads	Time(s) - Bard
1	294,258515		1	294,2975133		1	0,01%
2	149,1043118		2	149,1000492		2	0,00%
4	75,35806875		4	74,200492		4	-1,54%
8	41,1234882		8	41,0772326		8	-0,11%
16	21,185521		16	21,1741354		16	-0,05%
32	21,2211775		32	21,222427		32	0,01%

Resultados - 25000 Particulas

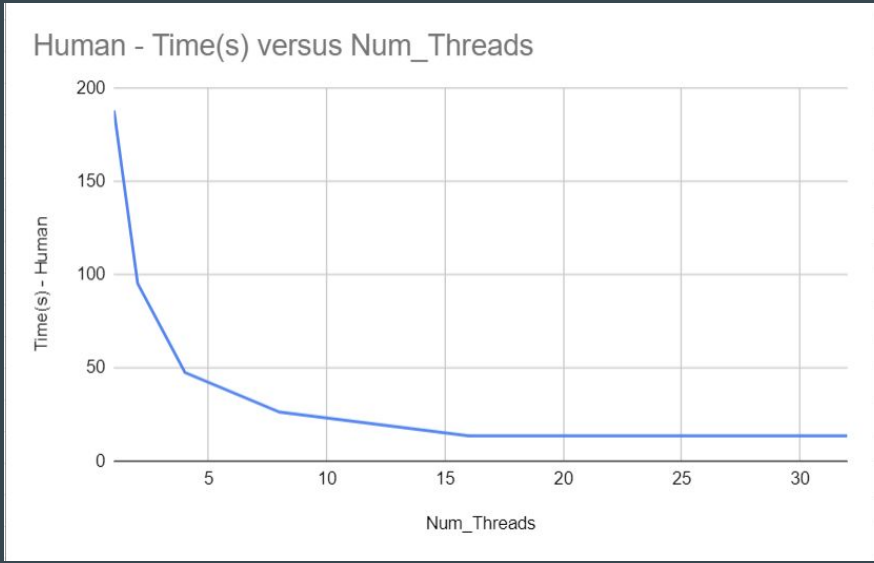


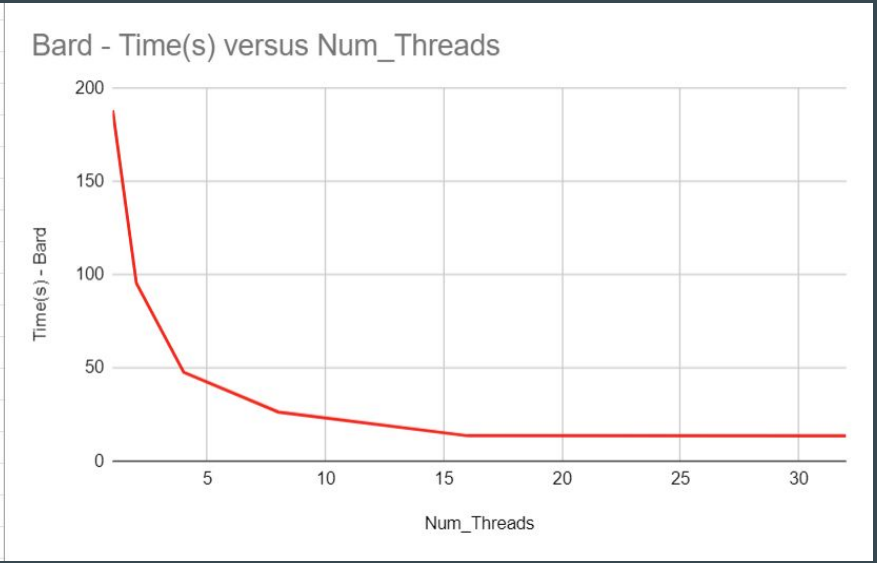


Resultados - 20000 Particulas

Human Results			Bard Results			Diff	
Num_Threads	Human - Time(s)		Num_Threads	Time(s) - Bard		Num_Threads	Bard - Time(s)
1	188,3331377		1	188,367645		1	0,02%
2	95,351094		2	95,4734834		2	0,13%
4	47,5778618		4	47,6651574		4	0,18%
8	26,24028125		8	26,2966912		8	0,21%
16	13,5625345		16	13,5894985		16	0,20%
32	13,5800188		32	13,5801086		32	0,00%

Resultados - 20000 Particulas

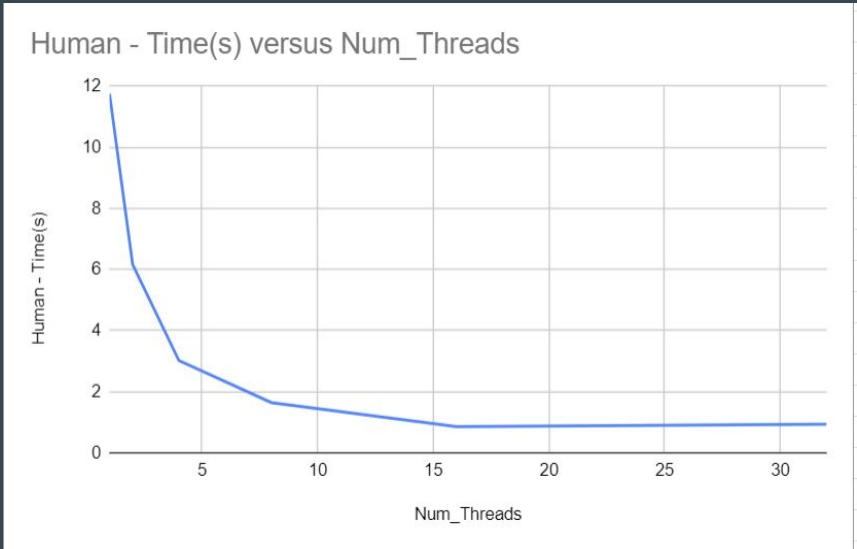


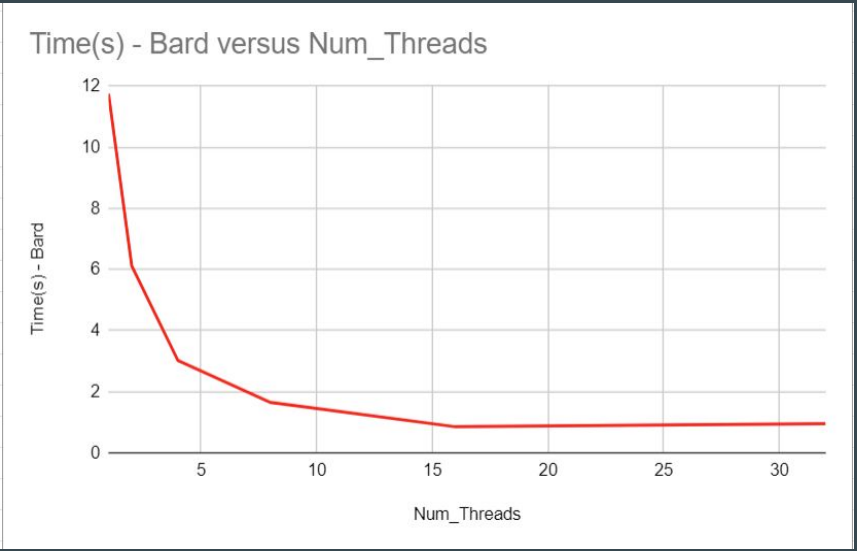


Resultados - 5000 Particulas

Human Results			Bard Results			Diff	
Num_Threads	Human - Time(s)		Num_Threads	Time(s) - Bard		Num_Threads	Bard - Time(s)
1	11,750217		1	11,7499294		1	0,00%
2	6,1639618		2	6,119575		2	-0,72%
4	3,0176188		4	3,0170008		4	-0,02%
8	1,6418198		8	1,6447046		8	0,18%
16	0,85098525		16	0,8506886		16	-0,03%
32	0,9334082		32	0,9493668		32	1,71%

Resultados - 5000 Particulas





Conclusões

- Temos um speedup significativo utilizando paralelismo;
- O N-Body é um problema simples e facilmente paralelizável;
  - Torna-o um problema ideal para ser paralelizado com auxílio de uma IA;
  - Mesmo assim, é importante que o programador entenda o que está sendo gerado.
- Para problemas simples, o compilador consegue fazer uma boa escolha do escalonamento.

Obrigado!