NBODy Parallel OpenMp

Jonathan W. Guimarães[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brasil

jonathanwguimaraes@gmail.com

1. Estratégia de paralelismo

A estratégia de paralelismo utilizada foi a de executar o mínimo de alterações no código original para executar o paralelismo. Primeiro foi criada uma otientação a objetos que facilitasse os testes, e versões paralelas das funções hounde ouvessem sido identificados trechos que poderiam ser paralelizáveis. As funções detectadas e replicadas foram *InitParticles*, *ComputeForces* e *ComputeNewPos*, sendo criadas então as suas respectivas versões paralelas, *ParallelInitParticles*, *ParallelComputeForces* e *ParallelComputeNewPos*.

2. Kernels

As funções *ParallelInitParticles* e *ParallelComputeNewPos*. Como podemos ver nas Figuras 01 e 02, a estratégia de paralelismo utilizada foi simplesmente adicionar a pré diretiva de compilação pragma omp parallel for, setando o número de threads vindo do arquivo de configuração para o teste em questão.

```
void NBody::ParallelInitParticles( Particle particles[], ParticleV pv[], int npart )
{
    int i;
    #pragma omp parallel for num_threads(num_threads)
        for (i=0; i<npart; i++) {
            particles[i].x    = this->Random();
            particles[i].y    = this->Random();
            particles[i].z    = this->Random();
            particles[i].mass = 1.0;
            pv[i].xold    = particles[i].x;
            pv[i].yold    = particles[i].y;
            pv[i].zold    = particles[i].z;
            pv[i].fx      = 0;
            pv[i].fy      = 0;
            pv[i].fz      = 0;
        }
}
```

Já a função ParallelComputeForces, na figura 03, possuía laço duplo teve uma estratégia de diretivas particular a ela. Foi necessário fazer um collapse(2) para que o for fosse de profundidade 2, ou seja, englobasse os dois laços aninhados.
A solução vitoriosa precisou levar em conta uma redução de soma em cima do valor max_f. Isso acelerou bastante o desempenho em relação às outras formas de tentativas de paralelismo.Esse kernel também levou em consideração o número de testes de cada rotina definida pelo arquivo de configuração.

```
double NBody::ParallelComputeNewPos(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 num_threads(num_threads)
  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;
}
```

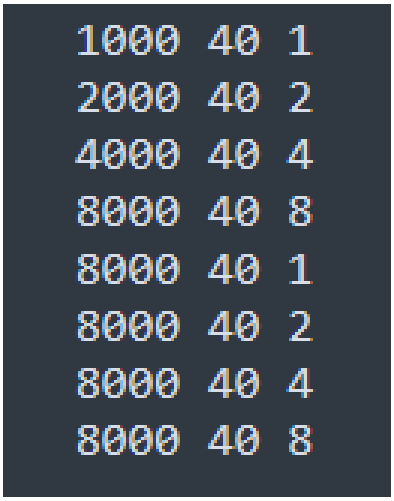
```
double NBody::ParallelComputeForces( Particle myparticles[], Particle others[], ParticleV pv[], int npart )
{
  double max_f;
  int i;
  max_f = 0.0;
#pragma omp parallel for reduction(+ : max_f) collapse(2) num_threads(num_threads)      expected unqualified

  for (i=0; i<npart; i++) {
        int j;
        double xi, yi, mi, rx, ry, mj, r, fx, fy, rmin;                                  unused va
        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;
}
```

3. Metodologia e Detalhes do Experimento

Para execução dos testes foi criada uma bateria de testes unitários, tendo como referência o arquivo de entrada *nbody.in*. Cada uma das linhas é composta de 3 números, significando respectivamente o número de partículas daquele teste, o número de interações entre as partículas e o número de testes que aquele teste irá utilzar para sua execução. O tamanho da entrada foi variando conforme o número de threads de cada teste. Foram criadas configurações de testes para 2,4,6 e 8 threads. Cada um dos testes foi repetido 20 vezes.
O ambiente utilizado foi : Windows 10, 64 bits, processador Intel i5 11th gen, com 8 núcleos. Foi utilizado compilador msvc2019 x64 para a compilação, com as tags de otimização ativadas.

Foi computada a média de cada uma das baterias de testes, com o cálculo da média de cada um deles. No fim da execução, temos também a média total dos tempos paralelo e sequencial. É importante salientar que para cada novo teste com diferente entrada e número de threads também foi executada a sua versão em série, capturando esse tempo para o cálculo de speedup, seguindo a lei de Amdahl. O arquivo de configuração utilizado se encontra juntamente aos códigos.
Podemos ver na Figura 04 o arquivo de entrada nbody.in, mostrando o número de partículas, interações e threads de cada bateria de testes.
As primeiras quatro linhas do arquivo de configuração eram parte de um teste de escalabilidade fraca, e as últimas quatro, parte de um teste de escalabilidade forte.



4. Resultados

Tendo como base o arquivo da figura acima, temos abaixo respectivamente o resultado dos testes das 20 execuções de cada um dos testes.
As linhas de 01 a 04 representam um teste de escalabilidade fraca, enquanto as linhas de 05 a 08, um teste de escalabilidade forte.

4.1. Linha 01

Número de partículas : 1000
Número de interações : 40
número de Threads 1

Média após 20x	**LINHA 01**
Algoritmo em Série	0.39945
Algoritmo Paralelo	0.38905
Speedup	1.0267

4.2. Linha 02

Número de partículas : 2000
Número de interações : 40
Número de Threads 2

Média após 20x	**LINHA 02**
Algoritmo em Série	1.6016
Algoritmo Paralelo	1.0123
Speedup	1.5822

4.3. Linha 03

Número de partículas : 4000
Número de interações : 40
Número de Threads 4

Média após 20x	**LINHA 03**
Algoritmo em Série	6.8182
Algoritmo Paralelo	2.9104
Speedup	2.3427

4.4. Linha 04

Número de partículas : 8000
Número de interações : 40
número de Threads 8

Média após 20x	**LINHA 04**
Algoritmo em Série	25.503
Algoritmo Paralelo	9.3383
Speedup	2.731

4.5. Linha 05

Número de partículas : 8000
Número de interações : 40
número de Threads 1

Média após 20x	**LINHA 05**
Algoritmo em Série	38.444
Algoritmo Paralelo	38.253
Speedup	1.005

4.6. Linha 06

Número de partículas : 8000
Número de interações : 40
Número de Threads 2

Média após 20x	**LINHA 06**
Algoritmo em Série	33.81
Algoritmo Paralelo	25.51
Speedup	1.3253

4.7. Linha 07

Número de partículas : 8000
Número de interações : 40
Número de Threads 4

4.8. Linha 08

Número de partículas : 8000
Número de interações : 40
número de Threads 8

Média após 20x	**LINHA 07**
Algoritmo em Série	370.81
Algoritmo Paralelo	18.603
Speedup	19.933

Média após 20x	**LINHA 08**
Algoritmo em Série	51.507
Algoritmo Paralelo	13.719
Speedup	3.7545

5. Análise e Conclusão

Observando os testes das linhas 01 a 04, podemos perceber claramente que o speedup aumenta, quando o número de threads utilizados aumenta, mesmo que os dados de entrada também cresçam. Isso significa que o algoritmo paralelo é eficiente e possui escalabilidade fraca, dado que o aumento do número de entradas variou proporcionalmente ao de processos.

Para afirmarmos que o mesmo possui escalabilidade forte, é necessário acompanharmos os testes das linhas 05 a 08, onde o valor de entrada foi aumentado consideravelmente e o número de threads atuantes foi incrementado aos poucos. Os resultados desses testes mostraram que o algoritmo é fortemente escalável, dado que o tempo de execução e o speedup mostram um aumento de performance em relação à mesma entrada quando aumentamos as threads. Um leitor atencioso perceberá a diveregência do valor do *speedup* dos testes da configuração da linha 07 para os demais, 19,933 de aumento de performance. Isso não significa que o algoritmo paralelo com 4 threads seja melhor que o com 8 threads, ainda mais nessa grandeza de quase 20 vezes. Se compararmos a média das execuções paralelas da linha 07 com a 08, perceberemos que o algoritmo paralelo com 8 threads ainda é mais eficiente que o com 4 : 13.719 contra 18.603. O que percebemos, na verdade é que por algum motivo (provavelmente uma excessiva troca de contextos) o algoritmo serial do nbody nessa execução foi terrivelmente lento. A fim de esclarecer se isso foi um caso isolado ou se faz parte do comportamento do algoritmo, seriam necessários mais testes, com mais variações de entradas e quantidades de threads, além de executar mais vezes essa mesma bateria de testes, analisando e comparando os resultados.

Podemos evidenciar também nesse trabalho como é possível, através das diretivas certas de pré-compilação, acelerar a execução de um algoritmo tendo em vista o mínimo de alterações no código : apenas entendendo o contexto do problema e raciocinando sobre qual a melhor diretiva do opencv para aquele problema específico e como ela se relaciona com a parte paralelizável do código. Antes de chegar na solução vitoriosa, foram testadas várias combinações de diretivas de pré-compilação que não obtiveram performance semelhante à solução final.