Relatório Programação Paralela

Guilherme Carbonari Boneti - GRR20196478

Parte principal (Kernel) do algoritmo sequencial

A parte principal do algoritmo é o seguinte laço *while*:

```
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);
}
```

Esse laço executa *cnt* vezes, onde *cnt* é o número de etapas de tempo a serem simuladas pelo algoritmo. O laço cria uma variável do tipo *double* chamada *max_f* que recebe o resultado da função *ComputeForces()*. Essa função calcula as forças gravitacionais, posições e velocidade das partículas da simulação. Em seguida utiliza a função *ComputeNewPos()* que calcula as mudanças nos dados para a próxima etapa com base nos valores anteriores de posição, velocidade e força das partículas. O retorno da função é acumulado na variável *sim_t* que armazena o tempo da simulação. A cada iteração, a função *ComputeForces()* depende dos dados calculados pela função *ComputeNewPos()* da iteração anterior, portanto há dependência entre as iterações.

Já a função *ComputeForces(),*

```
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;
}
```

percorre todas as partículas do sistema, calculando a influência das outras partículas do sistema em cada partícula individualmente. As variáveis *xi*, *yi*, *mi*, *mj*, *rx*, *ry* e *rmin* são recalculadas a cada iteração, enquanto *fx, fy* e *r* são acumuladas a cada iteração.

Por fim, a parte principal da função *ComputeNewPos()*,

```
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;
}
```

percorre cada partícula calculando seus novos valores. As variáveis *xi* e *yi* são recalculadas a cada iteração, portanto não há dependência entre as iterações.

Estratégia de paralelização utilizada

As funções *InitParticles()*, *ComputeForces()* e *ComputeNewPos()* foram escolhidas para a paralelização, enquanto o laço *while* da função *main()* não foi paralelizado por haver dependência de dados entre as iterações e exigir uma execução sequencial.
Na função *InitParticles()*, paralelizei o laço *for*, deixando cada thread calcular certa região dos vetores *pv* e *particles* e utilizei a declaração *schedule(auto)* para deixar a biblioteca dividir as tarefas entre as threads como achar melhor.
Já na função *ComputeForces()*, apliquei a paralelização nos laços externo e interno da função. Privatizei as variáveis *xi, yi, rx, ry, mj, r, fx, fy, rmin* para evitar condições de corrida. Além disso apliquei um *reduction* nas variáveis *fx* e *fy* para garantir que elas sejam privadas a cada thread e que possam juntar os resultados parciais nas variáveis globais.
Finalmente, na função *ComputeNewPos()*, paralelizei o laço for que percorria as particulas. Utilizei, de forma similar, a estratégia de privatizar as variáveis *xi* e *yi* que são acessadas por todas a threads. Assim cada thread calculou um pedaço dos vetores *particles* e *pv*. Também utilizei a declaração *schedule(auto)* como feito anteriormente.

Descrição da metodologia dos experimentos

Utilizei para medir os tempos em pedaços do código a função *omp_get_wtime().* Cada métrica de tempo calculada corresponde à média de 20 execuções do algoritmo. O desvio padrão correspondente a cada média também foi calculado.
A versão do SO utilizada foi **Linux Mint 19.2 Cinnamon**.
O Kernel do Linux utilizado é **4.15.0-65-generic**.
O processador da máquina utilizada nos experimentos é um **Intel Core i7-8565U**, com 12GB de memória RAM.
Para a compilação foi utilizado o gcc com a flag -O3.

Porcentagem de tempo na região não paralelizável

Para obter a porcentagem de tempo que o algoritmo passa em trechos que não serão paralelizados, medi 20 vezes o tempo que o algoritmo gasta em cada um dos trechos que serão paralelizados e o tempo total gasto no algoritmo. A média de tempo nas funções *InitParticles()*, *ComputeForces()*, *ComputeNewPos()* e na função *main()*, para tamanhos de entrada N=10.000, 14.500 e 20.000, assim como o desvio padrão da amostra são apresentados na tabelas abaixo.

MÉDIA DO TEMPO SEQUENCIAL (10 ETAPAS)				
	INIT	MAIN	FORCES	NEW
N=10.000	0,00138865	4,4262915	4,3928553	0,00036865
N=14.500	0,002165	9,254871	9,211648	0,000661
N=20.000	0,00279385	17,6011965	17,5427466	0,0009724

DESVIO PADRÃO DO TEMPO SEQUENCIAL (10 ETAPAS)				
	INIT	MAIN	FORCES	NEW
N=10.000	0,0005426936	0,0719872962	0,0738983593	0,0000330776
N=14.500	0,0012819209	1,607006621	1,607040202	0,0001461063
N=20.000	0,0008631306	0,1588507445	0,1612941995	0,0000560407

Com base nesses dados, obtive a porcentagem de tempo que o algoritmo passa em trechos que não serão paralelizados:

PORCENTAGEM DE TEMPO EM TRECHOS QUE NÃO SERÃO PARALELIZADOS (10 ETAPAS)	
N=10.000	0,76%
N=14.500	0,47%
N=20.000	0,33%

Speedup teórico pela Lei de Amdahl

Utilizando as porcentagens do gráfico anterior, calculei o speedup máximo referente a cada entrada de tamanho 10.000, 14.500 e 20.000, com 2, 4, 8 e infinitos processadores.

SPEEDUP MÁXIMO (10 ETAPAS) => N=10.000	
P=2	1,985005273
P=4	3,911360758
P=8	7,598221325
P=∞	132,3802196

SPEEDUP MÁXIMO (10 ETAPAS) => N=14.500	
P=2	1,990702826
P=4	3,944730802
P=8	7,746742849
P=∞	214,1191264

SPEEDUP MÁXIMO (10 ETAPAS) => N=20.000	
P=2	1,9933804
P=4	3,960543584
P=8	7,818260325
P=∞	301,1330473

Speedup e Eficiência

Para obter as métricas de Speedup e Eficiência, medi 20 vezes o tempo do algoritmo paralelo e calculei sua média e desvio padrão, para 1, 2, 4 e 8 Threads, variando a entrada de N=10.000, 14.500 e 20.000, de forma que o algoritmo dobrasse aproximadamente seu tempo de execução a cada N. As métricas obtidas se encontram nas tabelas abaixo:

MÉDIA DO TEMPO PARALELO (10 ETAPAS)				
	1 THREAD	2 THREADS	4 THREADS	8 THREADS
N=10.000	4,4262915	2,5966992	1,33818	1,1694463
N=14.500	9,254871	4,943727	2,608585	2,343562
N=20.000	17,6011965	9,451189	4,950237	4,43194475

DESVIO PADRÃO DO TEMPO PARALELO (10 ETAPAS)				
	1 THREAD	2 THREADS	4 THREADS	8 THREADS
N=10.000	0,07198729627	0,239644594	0,6495790668	0,06182566897
N=14.500	1,607006621	0,7449649014	1,244933427	1,384278786
N=20.000	0,1588507445	0,4769227025	1,725915977	0,1840291041

Utilizando essas tabelas e as tabelas de tempo do algoritmo sequencial apresentadas anteriormente, calculei as métricas de Speedup e Eficiência, para 1, 2, 4 e 8 Threads e para N=10.000, 14.500 e 20.000:

SPEEDUP (10 ETAPAS)				
	1 THREAD	2 THREADS	4 THREADS	8 THREADS
N=10.000	1	1,704583842	3,307695153	3,784946346
N=14.500	1	1,872043299	3,547851038	3,949061727
N=20.000	1	1,862326158	3,555627034	3,971438611

EFICIÊNCIA (10 ETAPAS)				
	1 THREAD	2 THREADS	4 THREADS	8 THREADS
N=10.000	1	0,8522919212	0,8269237883	0,4731182932
N=14.500	1	0,9360216493	0,8869627595	0,4936327159
N=20.000	1	0,9311630791	0,8889067584	0,4964298263

Análise dos resultados

Vemos que o algoritmo possui uma porcentagem de região paralelizável muito grande, como mostra a tabela abaixo:

PORCENTAGEM DE TEMPO EM TRECHOS QUE SERÃO PARALELIZADOS	
N=10.000	99,28%
N=14.500	99,56%
N=20.000	99,69%

Dessa forma, a estimativa de speedup máximo pela lei de Amdahl beira o speedup superlinear, em que a razão entre o tempo de execução sequencial e o tempo de execução em paralelo com ‘p’ processadores é maior que p. Assim, ao analisar o speedup real obtido, vemos que com 2 threads obtemos um valor próximo de 2 para todas as entradas e o mesmo segue para 4 threads, quando obtemos um valor próximo de 4, o que é muito bom, porém ao aumentar o número de threads para 8, o speedup cresce muito lentamente e não se aproxima do máximo estimado pela lei de Amdahl 8 threads, chegando a 3,97 com N=20.000. Da mesma forma a eficiência obtida segue esse padrão, uma eficiência alta para 2 e 4 threads mas que diminui rapidamente ao aumentar o número de threads. Esse cenário poderia ser alterado ao aumentar N ou o número de etapas da simulação.

Escalabilidade

Podemos observar, pela tabela de speedup que o algoritmo não é fortemente escalável, pois diminui muito rapidamente a eficiência conforme o número de threads aumenta, como ilustra essa linha da tabela de eficiência apresentada anteriormente:

	1 THREAD	2 THREADS	4 THREADS	8 THREADS
N=10.000	1	0,8522919212	0,8269237883	0,4731182932

De fato, até 4 threads o algoritmo mantém a escalabilidade mas ao aumentar para 8 threads o algoritmo diminui a escalabilidade bruscamente, portanto não é possível afirmar que ele é fortemente escalável.

Ainda, com base nos dados coletados, observamos que ao aumentar N, propositalmente de 10.000 para 14.500 a 20.000, de forma que o tempo de execução dobrasse a cada N, obtivemos um leve aumento de eficiência para cada número de threads fixo, conforme o exemplo abaixo:

	4 THREADS
N=10.000	0,8269237883
N=14.500	0,8869627595
N=20.000	0,8889067584

Porém, ao analisar as diagonais, vemos que a eficiência não é mantida ao dobrar o tempo de execução e o número de threads, conforme a tabela abaixo:

	1 THREAD	2 THREADS	4 THREADS	8 THREADS
N=10.000	1	0,8522919212	0,8269237883	0,4731182932
N=14.500	1	0,9360216493	0,8869627595	0,4936327159
N=20.000	1	0,9311630791	0,8889067584	0,4964298263

Assim, também não podemos afirmar que o algoritmo possui escalabilidade fraca, pois ao usar mais threads conforme N aumenta, o tempo de execução total não permanece fixo. Essa característica já pode ser notada na tabela de tempo de execução do algoritmo paralelo, onde o tempo de execução caia pela metade ao aumentar as threads de 1 para 2 e para 4, porém ao aumentar para 8, o tempo de execução diminuiu apenas levemente. Portanto, o algoritmo não é escalável.
Esse resultado nos mostra que utilizar o algoritmo paralelo com até 4 threads pode valer a pena, pois possui uma alta eficiência e o tempo de execução é reduzido em até 90%, porém com 8 threads pode não utilizar bem os recursos disponíveis.