T2 - 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

A estratégia de paralelização utilizada foi baseada no loop while principal do programa. Inicialmente, criei dois tipos derivados de dados do MPI, MPI_Particle e MPI_ParticleV, que correspondem às estruturas de dados Particle e ParticleV. Após isso, calculei o número de partículas a serem divididas entre cada processo e utilizei a função MPI_Bcast() para todos os processos terem acesso a essa variável. Criei dois subvetores chamados sub_part e sub_pv, para armazenar parte das partículas e utilizei novamente a função MPI_Bcast() para enviar os vetores particles e pv a todos os processos. Por fim novamente utilizei MPI_Bcast() para enviar a variável cnt que indica o numero de simulações a todos os processos. Após isso, dentro do laço, utilizei MPI_Scatter para dividir as particulas de particles e pv entre sub_part e sub_pv, respectivamente. calculei parte da função ComputeForces() em cada processo e utilizei MPI_AllReduce para alcançar a variável max_f global. Por fim, calculei a função ComputeNewPos() e uni os resultados parciais dos subvetores nos vetores globais com MPI_Gather.

Descrição da metodologia dos experimentos

Utilizei para medir os tempos em pedaços do código a função *MPI_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 Tina base: Ubuntu 18.04 bionic**.
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, 4 núcleos físicos e 8 CPUs.
A versão do gcc utilizada foi gcc 7.4.0 (Ubuntu 7.4.0-1ubuntu1~18.04.1).
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 no trecho que não foi paralelizado e o tempo total gasto no trecho paralelizado. A média de tempo em ambos os trechos 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 NA REGIÃO SEQUENCIAL (10 ETAPAS)		
	INIT	MAIN
N=10.000	0,001324444444	4,5461
N=20.000	0,002894	17,18944444
N=30.000	0,004579	39,903

PORCENTAGEM DO DESVIO PADRÃO DO TEMPO NA REGIÃO SEQUENCIAL (10 ETAPAS)		
	INIT	MAIN
N=10.000	4,98%	1,83%
N=20.000	2,98%	0,92%
N=30.000	2,80%	0,24%

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	
N=10.000	0,03%
N=20.000	0,02%
N=30.000	0,01%

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	1,000145689
P=4	1,00021855
P=∞	1,000291421

SPEEDUP MÁXIMO (10 ETAPAS) => N=20.000	
P=2	1,000084187
P=4	1,000126285
P=∞	1,000168387

SPEEDUP MÁXIMO (10 ETAPAS) => N=30.000	
P=2	1,00005738
P=4	1,000086072
P=∞	1,000114766

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 e 4 CPUs, variando a entrada de N=10.000, 20.000 e 30.000, 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 CPU	2 CPUs	4 CPUs
N=10.000	4,5461	2,5966992	1,1968
N=20.000	17,18944444	8,912	4,665
N=30.000	39,903	20,872	12,666

PORCENTAGEM DO DESVIO PADRÃO DO TEMPO NA REGIÃO PARALELA (10 ETAPAS)			
	1 CPU	2 CPUs	4 CPUs
N=10.000	1,83%	0,92%	1,42%
N=20.000	0,92%	1,10%	0,19%
N=30.000	0,24%	1,55%	1,83%

Utilizando essas tabelas e as tabelas de tempo do algoritmo sequencial apresentadas anteriormente, calculei as métricas de Speedup e Eficiência, para 1, 2, 4 CPUs e para N=10.000, 20.000 e 30.000:

SPEEDUP (10 ETAPAS)			
	1 CPU	2 CPUs	4 CPUs
N=10.000	1	1,75072261	3,798546123
N=20.000	1	1,928797626	3,684768369
N=30.000	1	1,911795707	3,546933333

EFICIÊNCIA (10 ETAPAS)			
	1 CPU	2 CPUs	4 CPUs
N=10.000	1	0,8753613048	0,9496365307
N=20.000	1	0,9643988128	0,9211920922
N=30.000	1	0,9558978536	0,8867333333

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 (10 ETAPAS)	
N=10.000	99,97%
N=20.000	99,98%
N=30.000	99,99%

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 CPUs obtemos um valor próximo de 2 para todas as entradas e o mesmo segue para 4 CPUs, quando obtemos um valor próximo de 4. Da mesma forma, a eficiência obtida segue esse padrão, uma eficiência alta para 2 e 4 CPUs. 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 é fortemente escalável, pois não diminui muito rapidamente a eficiência conforme o número de CPUs aumenta, como ilustra essa linha da tabela de eficiência apresentada anteriormente:

EFICIÊNCIA (10 ETAPAS)			
	1 CPU	2 CPUs	4 CPUs
N=20.000	1	0,9643988128	0,9211920922

Ainda, com base nos dados coletados, observamos que ao aumentar N, propositalmente de 10.000 para 20.000 a 30.000, de forma que o tempo de execução dobrasse a cada N, obtivemos um leve declínio de eficiência para cada número de CPUs fixo, conforme o exemplo abaixo:

	4 CPUs
N=10.000	0,9496365307
N=14.500	0,9211920922
N=20.000	0,8867333333

Também, ao analisar as diagonais, vemos que a eficiência tem um pequeno declínio ao dobrar o tempo de execução e o número de CPUs, conforme a tabela abaixo:

EFICIÊNCIA (10 ETAPAS)			
	1 CPU	2 CPUs	4 CPUs
N=10.000	1	0,8753613048	0,9496365307
N=20.000	1	0,9643988128	0,9211920922
N=30.000	1	0,9558978536	0,8867333333

Assim, não podemos afirmar que o algoritmo possui escalabilidade fraca, pois ao usar mais CPUs conforme N aumenta, o tempo de execução total diminui. 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 CPUs de 1 para 2 e para 4. Portanto, o algoritmo também é fracamente escalável.