Arquitetura de Computadores 2 - Trabalho Final - Programação Paralela

*GIOVANNA ALBURQUERQUE, LUCIANO PEREIRA E RYAN MATHEUS*

*15 de dezembro de 2022*

*CPU de cada membro*

Giovanna: **Intel Core i5-7300HQ**

Luciano: **Intel Core i3-7100**

Ryan: **Intel Core i7-5500U**

1. CODIGO ORIGINAL

Abaixo o código original, contendo 3 loops, o primeiro usando loop para preencher 4 arrays com operação de multiplicação com double e os outros dois loops fazem o papel de realizar multiplicações entre valores dos arrays:

1. #include <chrono>
2. #include <iostream>
3. // GRUPO : LUCIANO PEREIRA, GIOVANNA ALBURQUERQUE, RYAN MATHEUS
4. static long NUM = 10000;
5. double step;
6. using namespace std;
7. int main () {
8. double a[NUM], b[NUM], c[NUM], d[NUM]; // arrays
9. int i,j; // loop counters
10. auto start = std::chrono::*high\_resolution\_clock*::now();
12. for (i=0; i < NUM; i++) {
13. a[i] = i \* 1.0;
14. b[i] = i \* 2.0;
15. c[i] = i \* 3.0;
16. d[i] = i \* 4.0;
17. }
18. for (i=0; i < NUM-2; i++) {
19. for (j=0; j < NUM-2; j++) {
20. a[i] = b[i] + c[i] \* d[j];
21. }
22. }
23. for (i=0; i < NUM; i++) {
24. d[i] = a[i] \* a[i];
25. }
26. auto finish = std::chrono::*high\_resolution\_clock*::now();
27. auto duration = std::chrono::duration\_cast<std::chrono::*microseconds*>(finish - start).count();
28. float time = duration / 1000000.0;
29. // print time
30. cout << "Time: " << time << " seconds" << endl;
31. }

Tabela de tempo de execução

(Desconsiderado a primeira execução devido ao cache miss e o consequente tempo discrepante aos demais testes)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Giovanna | Luciano | Ryan |
| N = 10000 | 0.274 s | 0.229677 s | 0.3566 s |
| N = 30000 | 2.34548 | 2.03315 s |  |
| N = 50000 | 6.34386 | 5.42051 s |  |

1. OTIMIZAÇÃO DE MEMÓRIA

Para otimizar o código foi feito a substituição de variáveis de ponto flutuante para inteiros bem como o uso de apenas uma matriz ao invés de 4 arrays

#include <chrono>

#include <iostream>

// GRUPO : LUCIANO PEREIRA, GIOVANNA ALBURQUERQUE, RYAN MATHEUS

static long NUM = 10000;

double step;

using namespace std;

int main () {

    //double a[NUM], b[NUM], c[NUM], d[NUM];

    int effarray[NUM][4];

    int i,j, temp;

    auto start = std::chrono::*high\_resolution\_clock*::now();

    for (i=0; i < NUM; i++) {

        temp = i \* 1;

        effarray[i][0] = temp;

        effarray[i][1] = temp \* 2;

        effarray[i][2] = temp \* 3;

        effarray[i][3] = temp \* 4;

    }

    for (i=0; i < NUM; i++) {

        for (j=0; j < NUM; j++) {

            effarray[i][0] = effarray[i][1] + effarray[j][2];

        }

    }

    for (i=0; i < NUM; i++) {

        effarray[i][3] = effarray[i][0] \* effarray[i][0];

    }

    auto finish = std::chrono::*high\_resolution\_clock*::now();

    auto duration = std::chrono::duration\_cast<std::chrono::*microseconds*>(finish - start).count();

    float time = duration / 1000000.0;

    // print time

    cout << "Time: " << time << " segundos" << endl;

}

Tabela de tempo de execução

(Desconsiderado a primeira execução devido ao cache miss e o consequente tempo discrepante aos demais testes)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Giovanna | Luciano | Ryan |
| N = 10000 | 0.22633 s | 0.191 s | 0.32685 s |
| N = 30000 | 2.23434 | 1.88837 s |  |
| N = 50000 | 5.98658 | 5.18491 s |  |

1. OTIMIZAÇÃO NA COMPILAÇÃO

Sendo usado o código otimizado em memoria acima, foram compilados 3 arquivos usando a seguinte argumentação:

* g++ -O1 '.\projeto com otimização de memoria.cpp' -o projetoO1
* g++ -O2 '.\projeto com otimização de memoria.cpp' -o projetoO2
* g++ -O3 '.\projeto com otimização de memoria.cpp' -o projetoO3

A seguir será apresentado os valores de tempo para cada grau de otimização (O1, O2, O3)

Tabela de tempo de execução com N = 10000

|  |  |  |  |
| --- | --- | --- | --- |
|  | Giovanna | Luciano | Ryan |
| O1 | 0.034288 | 0.056989 |  |
| O2 | 0.000021 | 0.061028 |  |
| O3 | 0.000019 | 0.060991 |  |

Tabela de tempo de execução com N = 30000

|  |  |  |  |
| --- | --- | --- | --- |
|  | Giovanna | Luciano | Ryan |
| O1 | 0.310695 | 0.53168 |  |
| O2 | 0.000113 | 0.52517 |  |
| O3 | 0.000087 | 0.514894 |  |

Tabela de tempo de execução com N = 50000

|  |  |  |  |
| --- | --- | --- | --- |
|  | Giovanna | Luciano | Ryan |
| O1 | 0.827625 | 1.49145 |  |
| O2 | 0.000256 | 1.48029 |  |
| O3 | 0.000138 | 1.45955 |  |

Conclusão

Nota-se uma grande melhoria ao aplicar uma otimização, mas além do primeiro nível não se nota uma mudança significativa

1. OTIMIZAÇÃO DE PARALELIZAÇÃO

Sendo usado o código otimizado em memoria acima com novas adições abaixo, envolvendo o OPENMP:

#include <stdio.h>

#include <omp.h>

#include <chrono>

#include <iostream>

// GRUPO : LUCIANO PEREIRA, GIOVANNA ALBURQUERQUE, RYAN MATHEUS

static long NUM = 10000;

using namespace std;

int main()

{

    int effarray[NUM][4];

    int i, j, temp;

    auto start = std::chrono::*high\_resolution\_clock*::now();

omp\_set\_num\_threads(4);

#pragma *omp* *parallel* *default*(*none*) *shared*(*NUM*, *effarray*, *temp*) *private*(*i*, *j*)

    {

#pragma *omp* *for* *nowait*

        for (i=0; i < NUM; i++) {

        temp = i \* 1;

        effarray[i][0] = temp;

        effarray[i][1] = temp \* 2;

        effarray[i][2] = temp \* 3;

        effarray[i][3] = temp \* 4;

    }

#pragma *omp* *for* *nowait*

        for (i=0; i < NUM; i++) {

            for (j=0; j < NUM; j++) {

                effarray[i][0] = effarray[i][1] + effarray[j][2];

        }

    }

#pragma *omp* *for* *nowait*

        for (i=0; i < NUM; i++) {

            effarray[i][3] = effarray[i][0] \* effarray[i][0];

        }

    }

    auto finish = std::chrono::*high\_resolution\_clock*::now();

    auto duration = std::chrono::duration\_cast<std::chrono::*microseconds*>(finish - start).count();

    float time = duration / 1000000.0;

    cout << "Time: " << time << " seconds" << endl;

}

Tabela de tempo de execução com N = 10000

|  |  |  |  |
| --- | --- | --- | --- |
|  | Giovanna | Luciano | Ryan |
| 1 Thread | 0.2411 | 0.202995 | 0.290995 |
| 2 Threads | 0.12525 | 0.125032 | 0.161659 |
| 3 Threads | 0.095394 | 0.104511 | 0.158534 |
| 4 Threads | 0.109656 | 0.098022 | 0.143941 |
| 5 Threads | 0.0904 | 0.103873 |  |
| 6 Threads | 0.094464 | 0.105993 |  |
| 7 Threads | 0.071045 | 0.102997 |  |
| 8 Threads | 0.07224 | 0.102996 |  |

Conclui-se a partir dos dados acima expostos que quando o numero de threads ultrapassa o número de CPUs de uma máquina a otimização se torna irrelevante

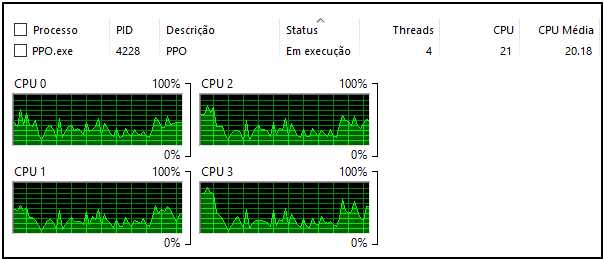
Tabela de tempo de execução com N = 30000

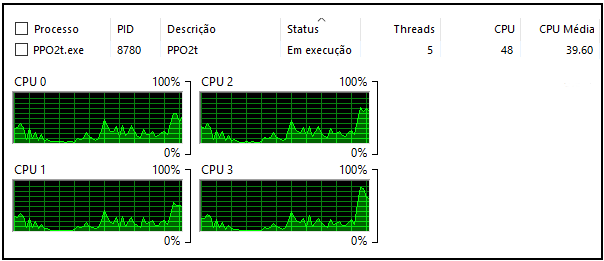
|  |  |  |  |
| --- | --- | --- | --- |
|  | Giovanna | Luciano | Ryan |
| 1 Thread | 2.30488 | 1.93231 | 2.57518 |
| 2 Threads | 1.22573 | 1.12519 | 1.64952 |
| 3 Threads | 0.878431 | 0.91599 | 1.58673 |
| 4 Threads | 0.7526 | 0.80757 | 1.43625 |

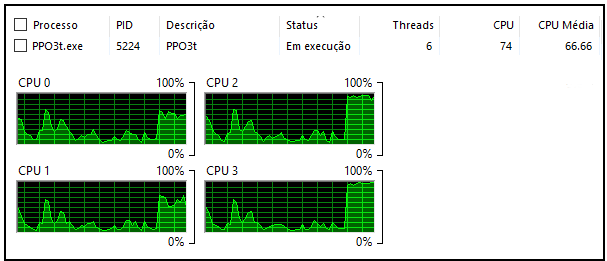
Tabela de tempo de execução com N = 50000

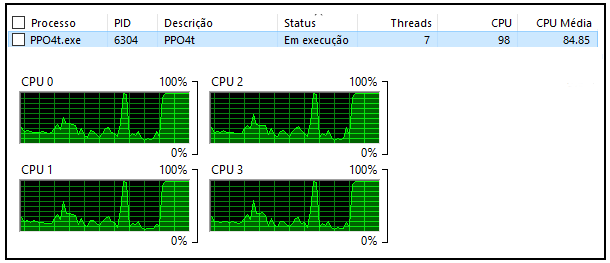
|  |  |  |  |
| --- | --- | --- | --- |
|  | Giovanna | Luciano | Ryan |
| 1 Thread | 6.73661 | 4.91612 |  |
| 2 Threads | 4.16616 | 3.03005 |  |
| 3 Threads | 2.7001 | 2.37024 |  |
| 4 Threads | 2.10147 | 2.2135 |  |

Uso nas CPUs (N=120000)

Uma Thread no Código

Duas Threads no Código

Três Threads no Código

Quatro Threads no Código

Conclusões

Após analise das imagens consta-se que à medida que o OPENMP inicializa mais threads, mais consome-se a CPU em geral, em 4 threads chega perto de alcançar os 100% de uso, enquanto uma thread em média só consome 20%, com isso podemos entender um dos motivos pelo qual a execução se torna mais rápida

1. COMPARAÇÃO