Resolução – Exercício Programação 2 (Programação Paralela e Distribuída)

Dezembro - 2022 / Leticia Bossatto Marchezi – 791003

Introdução

Este trabalho consiste no teste de desempenho da resolução da equação de transferência de calor de Laplace usando o método de Jacob, aplicando programação paralela com a biblioteca Pthreads e analisando o desempenho para diferentes números de cores utilizados.

O processo para execução dos testes pode ser descrito em 4 partes:

1) Código em C em que será realizada a solução da equação;

2) Arquivo makefile para compilação do código em C++;

3) Arquivo de definição do container Singularity;

4) Script para execução dos jobs no cluster HPC.

Resolução:

1. Código em C++ utilizando threads e barreiras
2. Definição do container

```
Bootstrap: docker
From: ufscar/ubuntu_mpich:latest

%help
        Paralelização da solução da equação de transferência de calor de Laplace
 pelo método de Jacobi usando Pthreads

%files
        ./app/. /opt

%post
        echo "Compiling programs..."
        cd /opt && make
        cp laplace_pth_it laplace_seq_it /usr/bin/

%runscript
        exec $@
```

Figure 1: Configuração do hardware do Cluster HPC

3. Script do job

```
#!/bin/bash
#SBATCH -J laplace                    # Job name
#SBATCH -p fast                      # Job partition
#SBATCH -n 1                         # Number of processes
#SBATCH -t 01:30:00                  # Run time (hh:mm:ss)
#SBATCH --cpus-per-task=40           # Number of CPUs per process
#SBATCH --output=%x.%j.out           # Name of stdout output file
#SBATCH --error=%x.%j.err            # Name of stderr output file


echo "*** SEQUENTIAL LAPLACE EQUATION GRID 1000X1000 ***"
srun singularity run container.sif laplace_seq_it 1000
echo " "
echo "*** PTHREAD LAPLACE EQUATION grid 1000x1000 ***"
for i in 1 2 5 10 20 40
do
        echo "*** Parallel algorithm with $i threads ***"
        srun singularity run container.sif laplace_pth_it 1000 $i
        echo " "
done
```

Figure 2: Script a ser executado no cluster

A plataforma a ser utilizada para executar o job é o cluster HPC da UFSCar, que possui as seguintes configurações:

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list: 0-3
Thread(s) per core:  1
Core(s) per socket:  4
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               63
Model name:          Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz
Stepping:            0
CPU MHz:             2297.339
BogoMIPS:            4594.67
Hypervisor vendor:   VMware
Virtualization type: full
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            25600K
NUMA node0 CPU(s):   0-3
```

Figure 3: Configuração do hardware do Cluster HPC

Na tabela acima, é possível observar que cada core possui 1 thread. Dessa forma, a progressão da quantidade de cores é de razão 1:1 em relação ao número de threads. Além disso, o servidor possui 4 CPUs e seu modelo é Intel Xeon CPU ES-2650 v3, possuindo clock rate de 2.30GHz.

Execução

Resultados dos testes no cluster HPC da UFSCar

Resolução:

Após logar no servidor HPC, inserir os programas em c, os arquivos do container buildado e do job, o job é submetido para execução na fila.

Os testes foram realizados com um grid de tamanho 1000x1000 para o algoritmo sequencial e paralelo, variando a quantidade de cores(threads) entre 1, 2, 5, 10, 20 e 40.

O output do job é o seguinte:

```
*** SEQUENTIAL LAPLACE EQUATION GRID 1000X1000 ***
Jacobi relaxation calculation: 1000 x 1000 grid

Kernel executed in 25.794345 seconds with 3001 iterations

*** PTHREAD LAPLACE EQUATION grid 1000x1000 ***
*** Parallel algorithm with 1 threads ***
Kernel executed in 25.737099 seconds with 3000 iterations and 1 threads

*** Parallel algorithm with 2 threads ***
Kernel executed in 12.711964 seconds with 3000 iterations and 2 threads

*** Parallel algorithm with 5 threads ***
Kernel executed in 5.238354 seconds with 3000 iterations and 5 threads

*** Parallel algorithm with 10 threads ***
Kernel executed in 3.291886 seconds with 3000 iterations and 10 threads

*** Parallel algorithm with 20 threads ***
Kernel executed in 2.465989 seconds with 3000 iterations and 20 threads

*** Parallel algorithm with 40 threads ***
Kernel executed in 2.500394 seconds with 3000 iterations and 40 threads
```

Figure 4: Script do job

A partir do tempo sequencial e do tempo de execução paralelizado, é possível calcular a taxa de speed up com a seguinte fórmula:

$$Speedup = \frac{T_{sequential}}{T_{parallel}} \tag{2.1}$$

E a taxa de eficiência com a seguinte fórmula:

$$Eficiency = \frac{Speedup}{N_{Threads}} \tag{2.2}$$

Tabela de tempo de execução, speedup e eficiência para o algoritmo com PThreads

Threads	Execution Time	Speed Up	Efficiency
1	25.737099	1.002224	1.002224
2	12.711964	2.029139	1.014570
5	5.238354	4.924132	0.984826
10	3.291886	7.835735	0.783573
20	2.465989	10.460041	0.523002
40	2.500394	10.316112	0.257903

Figure 5: Tabela de speedup e eficiência

Gráfico de Speed Up pela quantidade de Threads:

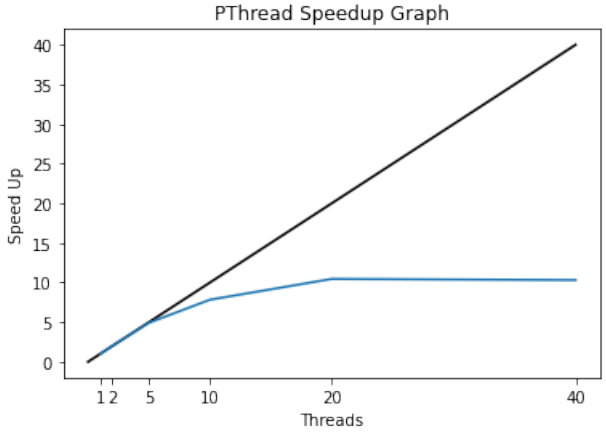

Figure 6: Gráfico PThread

Discussão dos resultados

Análise dos resultados

Resolução:

Após a paralelização do algoritmo de resolução da equação de transferência de calor, é possível observar que para um grid de dimensão 1000x1000 o tempo de execução foi otimizado de 25,7 segundos para 2,5 no melhor caso. Assim, a progressão de uso de cores entre 1 e 20 apresenta crescimento no speed-up devido ao processo de paralelização.

Entretanto, a alocação e gerenciamento das threads pode se tornar um processo custoso para o processador. Isso significa que nem sempre é proveitoso aumentar a quantidade de threads neste processo, por isso é necessário fazer o estudo do speed-up e eficiência da escalabilidade de um algoritmo.

No caso desenvolvido nesse exercício há uma queda de speed-up entre a execução com 20 threads para 40. Uma explicação possível é o custo de tempo para alocar quantidades superiores de threads e gerenciá-las, de forma que a vantagem do uso de paralelização não compensa tal custo, prejudicando o tempo total de execução. Neste algoritmo também são utilizadas barreiras para sincronização das threads, que pode provocar o desaceleramento da paralelização devido à necessidade de espera de sincronização entre as threads.

Além disso, o gráfico de speedup do algoritmo apresenta crescimento ideal entre 1 a 5 threads, entretanto se distancia da reta ideal a partir de 10 threads, concluindo que tal código apresenta baixa escalabilidade pelo insuficiente ganho de desempenho ao alocar mais recursos.

□