

Trabalho 4 - Perfilagem de código

Neste trabalho, será utilizado um programa em C que implementa a solução de um sistema de equações lineares pelo método iterativo de Jacobi. De forma resumida, esse método resolve sistemas de equações lineares da forma $Ax = b$, ao decompor a matriz A em uma matriz diagonal D e uma matriz R com os elementos fora da diagonal. Então, a solução é atualizada iterativamente, recalculando cada variável x_i usando apenas os valores da iteração anterior e os elementos de D e R . Por fim, o processo continua até que a diferença entre iterações sucessivas seja suficientemente pequena, garantindo assim a convergência para a solução correta.

Fonte do programa: <https://github.com/UoB-HPC/intro-hpc-jacobi/blob/master/jacobi.c>

Exercício 1: compile e execute o programa para pelo menos 3 tamanhos de matrizes e escolha 1 tamanho para prosseguir.

Para iniciar, o código foi compilado usando o GNU Compiler Collection.

- `gcc -o jacobi jacobi.c -lm`

Ao utilizar o programa no terminal, é possível passar o argumento `-n` para definir o tamanho da matriz a ser resolvida. Vamos testar alguns tamanhos e analisar as saídas.

- `./jacobi -n 500`

```
Matrix size:          500x500
Maximum iterations:   20000
Convergence threshold: 0.000100

Solution error = 0.024952
Iterations      = 1465
Total runtime   = 0.706413 seconds
Solver runtime  = 0.698515 seconds
```

- `./jacobi -n 1000`

```
Matrix size:          1000×1000
Maximum iterations:   20000
Convergence threshold: 0.000100

Solution error = 0.050047
Iterations     = 2957
Total runtime  = 5.626274 seconds
Solver runtime = 5.604598 seconds
```

- `./jacobi -n 1500`

```
Matrix size:          1500×1500
Maximum iterations:   20000
Convergence threshold: 0.000100

Solution error = 0.074989
Iterations     = 4207
Total runtime  = 26.511689 seconds
Solver runtime = 26.464425 seconds
```

- `./jacobi -n 2000`

```
Matrix size:          2000×2000
Maximum iterations:   20000
Convergence threshold: 0.000100

Solution error = 0.099977
Iterations     = 5479
Total runtime  = 63.756843 seconds
Solver runtime = 63.670389 seconds
```

- `./jacobi -n 3000`

```
Matrix size:          3000x3000
Maximum iterations:    20000
Convergence threshold: 0.000100

Solution error = 0.149981
Iterations     = 7691
Total runtime  = 180.492047 seconds
Solver runtime = 180.348173 seconds
```

- `./jacobi -n 4000`

```
Matrix size:          4000x4000
Maximum iterations:    20000
Convergence threshold: 0.000100

Solution error = 0.199909
Iterations     = 10040
Total runtime  = 490.671564 seconds
Solver runtime = 490.368515 seconds
```

Vamos prosseguir com a matriz de tamanho 4000x4000. O tempo total de execução foi de 490.671564 segundos ≈ **8 minutos**.

Exercício 2: recompile e execute o programa com otimização automática.

Ao utilizar o GCC, é possível definir o nível de otimização que o compilador deve aplicar ao código fonte durante o processo de compilação ao usar o parâmetro `-O`. Nesta etapa, vamos usar a flag `-O3`, que representa o maior nível de otimização (mais agressivo) suportado pelo compilador.

- `gcc -O3 -o jacobi_otim_auto jacobi.c -lm`

Vamos calcular novamente a solução para uma matrix 4000x4000.

- `./jacobi_otim_auto -n 4000`

```
Matrix size:          4000x4000
Maximum iterations:   20000
Convergence threshold: 0.000100

Solution error = 0.199909
Iterations      = 10040
Total runtime   = 331.244648 seconds
Solver runtime  = 330.975378 seconds
```

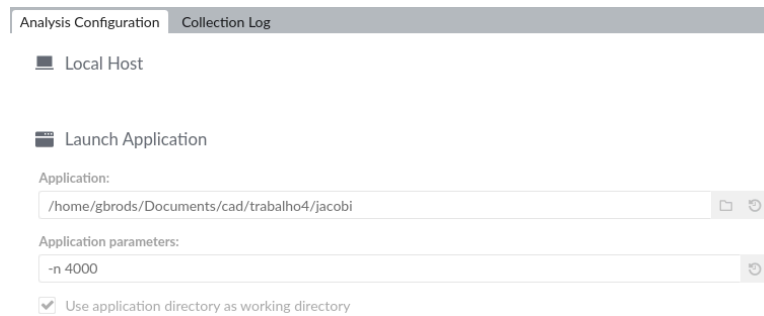
O tempo total de execução foi de 331.244648 segundos \approx **5,5 minutos**.

Houve uma redução de 159.426916 segundos \approx 2,6 minutos no tempo de execução, se comparado à compilação sem otimização automática. A melhora foi de aproximadamente 32%.

$$\frac{490.671564 - 331.244648}{490.671564} \times 100 \cong 32\%$$

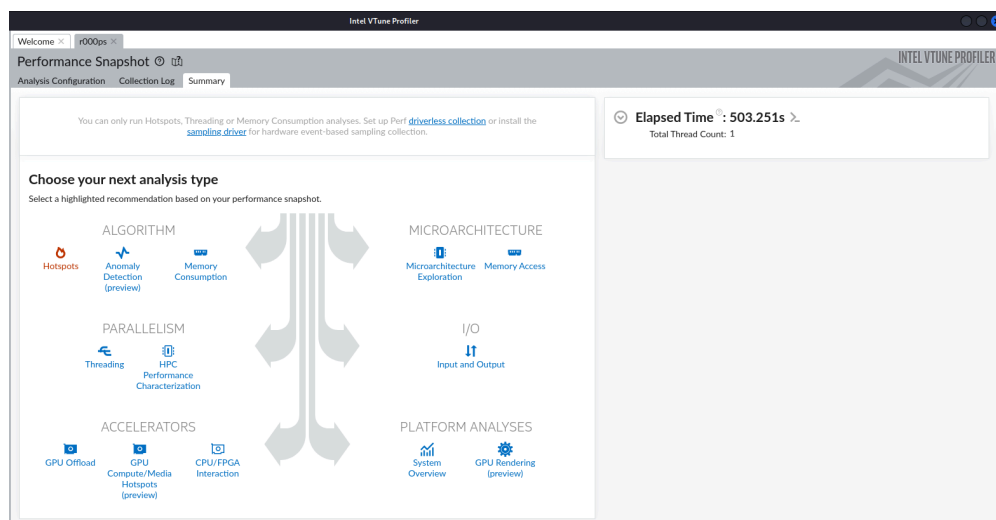
Exercício 3: execute a perfilagem do programa usando o gprof ou o Intel VTune.

Vamos executar a perfilagem do programa usando o Intel VTune, através da interface gráfica. Foi inserida a localização do programa jacobi compilado normalmente com o gcc e passado o parâmetro -n 4000.

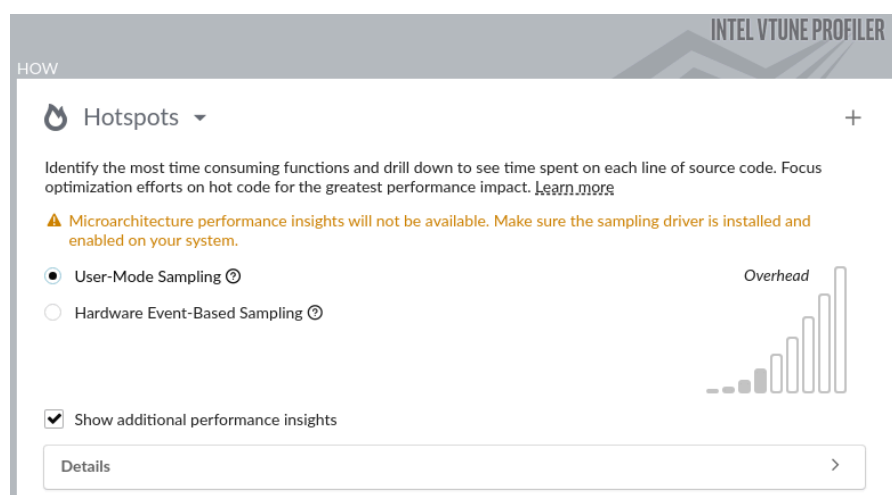


The screenshot shows the 'Analysis Configuration' tab of the Intel VTune Profiler. It includes a 'Local Host' selection, a 'Launch Application' button, an 'Application' field with the path '/home/gbrods/Documents/cad/trabalho4/jacobi', an 'Application parameters' field with '-n 4000', and a checked checkbox for 'Use application directory as working directory'.

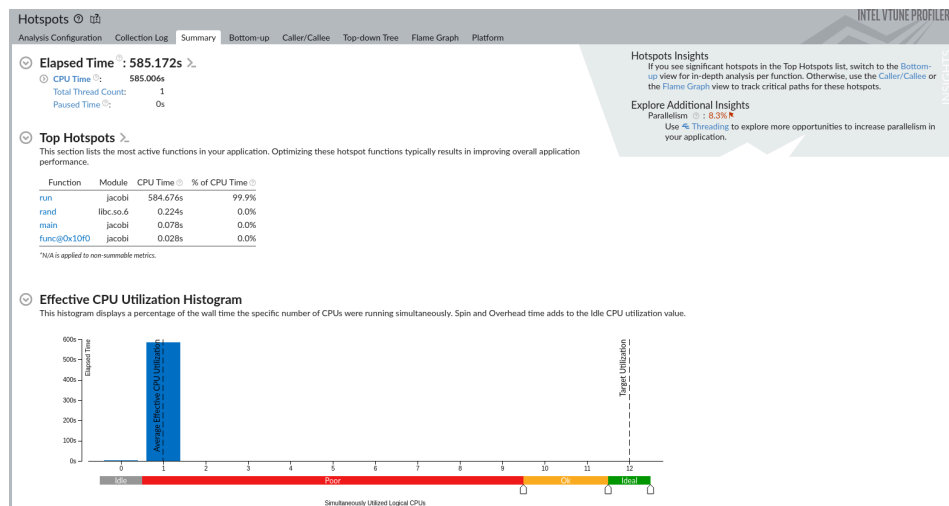
Após execução, temos a seguinte tela:



Vamos executar a análise de Hotspots (em vermelho) para entender a eficiência do código.



A análise gerou os seguintes resultados:



Exercício 4: analise o resultado da perfilagem.

Em destaque, as funções mais ativas do programa são:

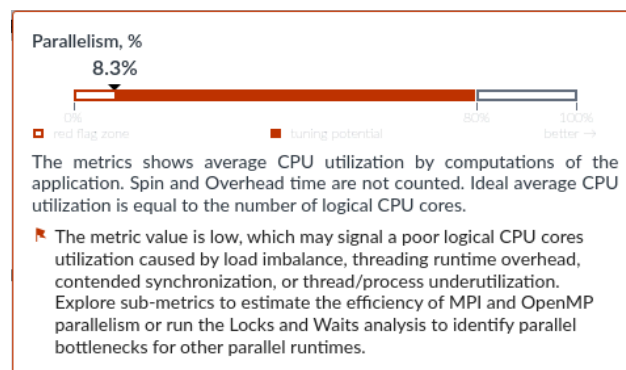
Function	Module	CPU Time	% of CPU Time
run	jacobi	584.676s	99.9%
rand	libc.so.6	0.224s	0.0%
main	jacobi	0.078s	0.0%
func@0x10f0	jacobi	0.028s	0.0%

*N/A is applied to non-summable metrics.

Foi possível identificar que:

- A função run no módulo jacobi é responsável por praticamente todo o tempo de CPU consumido. Isso indica que essa função é o principal ponto de interesse para otimizações, pois qualquer melhoria nesta função terá um impacto significativo no desempenho geral do programa.
- As outras três funções quase não consomem tempo de CPU, não sendo um ponto de preocupação para otimização.

Outro insight importante: foi detectado um baixo nível de paralelismo. Isso significa que, em média, apenas 8.3% da capacidade de processamento disponível está sendo usada eficientemente.



Exercício 5: de posse dos resultados de sua análise, tente 1 ação de otimização.

Vamos analisar a função run:

```
// Run the Jacobi solver
// Returns the number of iterations performed
int run(double *A, double *b, double *x, double *xtmp)
{
    int itr;
    int row, col;
    double dot;
    double diff;
    double sqdiff;
    double *ptrtmp;

    // Loop until converged or maximum iterations reached
    itr = 0;
    do
    {
        // Perform Jacobi iteration
        for (row = 0; row < N; row++)
        {
            dot = 0.0;
            for (col = 0; col < N; col++)
            {
                if (row != col)
                    dot += A[row + col*N] * x[col];
            }
            xtmp[row] = (b[row] - dot) / A[row + row*N];
        }

        // Swap pointers
        ptrtmp = x;
        x      = xtmp;
        xtmp   = ptrtmp;

        // Check for convergence
        sqdiff = 0.0;
        for (row = 0; row < N; row++)
        {
            diff = xtmp[row] - x[row];
            sqdiff += diff * diff;
        }

        itr++;
    } while ((itr < MAX_ITERATIONS) && (sqrt(sqdiff) >
CONVERGENCE_THRESHOLD));

    return itr;
}
```

Uma ação de otimização que podemos realizar diz respeito ao modo como os elementos da matriz A são acessados. No código original, a matriz A é armazenada em ordem de linha principal (row-major order), de modo que os elementos da matriz são armazenados linha por linha na memória. Veja:

```
for (col = 0; col < N; col++)
{
    if (row != col)
        dot += A[row + col*N] * x[col]; // row-major order
}
```

Linhas 59 a 63

Já no código modificado, podemos fazer com que a matriz A seja armazenada em ordem de coluna principal (column-major order), de modo que os elementos da matriz são armazenados coluna por coluna na memória. Veja as modificações:

```
for (col = 0; col < N; col++)
{
    if (row != col)
        dot += A[row*N + col] * x[col]; // modificado - column-major order
}
```

Modificação na linha 62

```
for (int col = 0; col < N; col++)
{
    double value = rand()/(double)RAND_MAX;
    A[row*N + col] = value; // modificado
    rowsum += value;
}
```

Modificação na linha 111

```
for (int col = 0; col < N; col++)
{
    tmp += A[row*N + col] * x[col]; // modificado
}
```

Modificação na linha 131

Vamos compilar novamente o programa e verificar o tempo total de execução.

- **gcc -o jacobi_otim_perfilagem jacobi_otim_perfilagem.c -lm**
- **./jacobi_otim_perfilagem -n 4000**

```
Matrix size:          4000x4000
Maximum iterations:   20000
Convergence threshold: 0.000100

Solution error = 0.199909
Iterations      = 10040
Total runtime   = 309.472181 seconds
Solver runtime  = 309.220353 seconds
```


O tempo melhorou em relação ao código inicial. Vamos compilar novamente com otimização automática e verificar o tempo total de execução.

- `gcc -O3 -o jacobi_otim_perfilagem jacobi_otim_perfilagem.c -lm`
- `./jacobi_otim_perfilagem -n 4000`

```
Matrix size:          4000×4000
Maximum iterations:   20000
Convergence threshold: 0.000100

Solution error = 0.199909
Iterations      = 10040
Total runtime   = 92.799536 seconds
Solver runtime  = 92.549603 seconds
```

O tempo de execução diminuiu drasticamente.

Link do código otimizado:

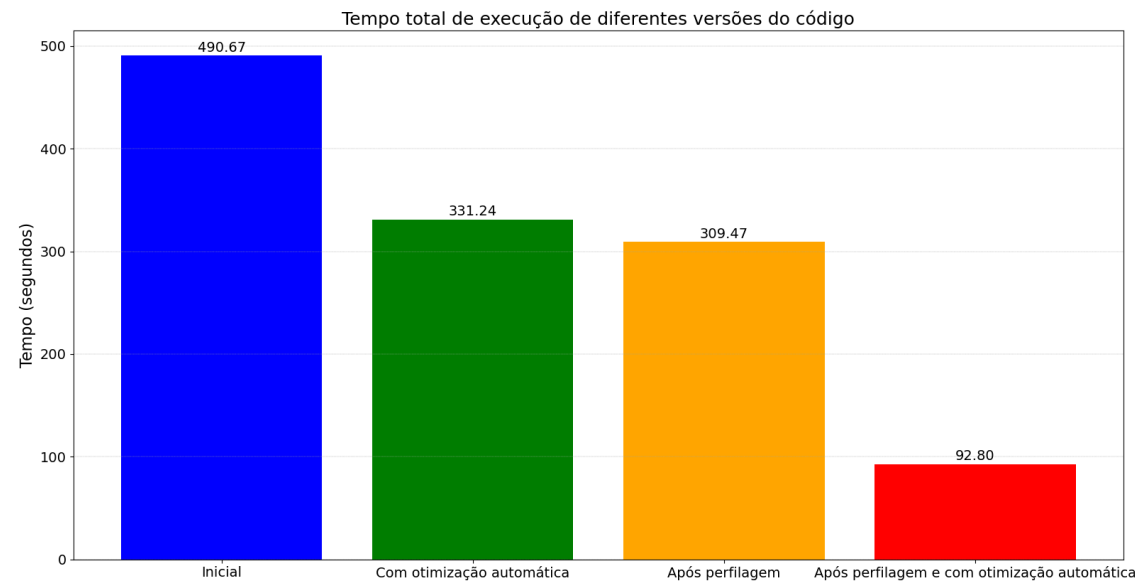
https://github.com/gbrods/comp-alto-desempenho/blob/main/trabalho4/jacobi_otim_perfilagem.c

Exercício 6: compare os tempos das versões (inicial, com otimização automática, e após perfilagem).

A seguir, vamos comparar os tempos das diferentes versões do código.

Versão do código	Inicial (matriz 400x400)	Otimização automática (gcc -O3)	Após perfilagem (column-major order)	Após perfilagem e com otimização automática
Tempo total de execução (s)	490.671564	331.244648	309.472181	92.799536

Para melhor visualização, foi feito o seguinte gráfico:



Link do código para gerar o gráfico:

<https://github.com/gbrods/comp-alto-desempenho/blob/main/trabalho4/grafico.py>

Portanto, é possível verificar que as ações de otimização alcançaram bons resultados e melhoraram significativamente o tempo total de execução do programa.