Tarefa 18: Análise de Desempenho OpenMP

Comparação de Adição de Vetores: Sequencial vs. Paralelo

Werbert Arles de Souza Barradas

DCA3703 - Programação Paralela
UFRN - Universidade Federal do Rio Grande do Norte

Outubro de 2025

Introdução: O Problema

O objetivo deste trabalho é analisar o ganho de desempenho obtido ao paralelizar um código de adição de vetores utilizando as diretivas OpenMP para CPUs multi-core.

Foram comparadas duas implementações:

1. **Sequencial:** Uma versão padrão em C que executa em um único núcleo (‘vadd.c‘).
2. **Paralela:** Uma versão que utiliza a diretiva `#pragma omp parallel for` para distribuir o trabalho entre múltiplos núcleos da CPU (‘vadd_par.c‘).

Objetivo: Quantificar o *speedup* e avaliar a eficiência da paralelização em um ambiente de computação de alto desempenho.

Implementações: Sequencial vs. Paralela

1. Versão Sequencial (‘vadd.c‘)

O laço de adição é executado em uma única thread, sem paralelismo.

```
1                // add two vectors
2                for (int i=0; i<N; i++){
3                   c[i] = a[i] + b[i];
4                }
```

2. Versão Paralela (‘vadd_par.c‘)

A diretiva OpenMP distribui as iterações do laço entre as threads disponíveis.

```
1                // add two vectors
2                #pragma omp parallel for
3                for (int i=0; i<N; i++){
4                   c[i] = a[i] + b[i];
5                }
```

Passo a Passo da Execução

1. **Compilação da Versão Sequencial:**
   O executável `./vadd` foi gerado usando o `makefile` padrão.

```
make vadd
```

2. **Compilação da Versão Paralela:**
   O executável `./vadd_par` foi gerado manualmente, ativando o suporte OpenMP.

```
gcc -fopenmp -O3 vadd_par.c -o vadd_par
```

3. **Submissão dos Jobs ao SLURM:**
   Cada versão foi submetida com seu próprio script, solicitando os recursos apropriados. A versão paralela solicitou 64 CPUs.

```
sbatch submit_vadd      # Executa ./vadd (sequencial)
sbatch script_18.sh     # Executa ./vadd_par (paralelo)
```

Problemas Encontrados e Soluções Apresentadas

Durante o desenvolvimento e execução da tarefa, alguns desafios surgiram.

Problema 1: Alteração do makefile original

Descrição: O código inicial em ‘makefile‘ criava o executavel de varios codigos disponiveis no tutorial, (`openmp-tutorial`), causando uma falha de compilação do "vadd.c".

Solução: Fiz a analise e a "limpeza"domake file para que executasse apenas o vadd.c e o vadd.o.

Problema 2: Compilação da versão paralela do vadd.c(vadd_par.c) com Makefile Padrão

Descrição: O `makefile` existente não possuía uma regra para compilar o arquivo ‘vadd_par.c‘, apenas o ‘vadd.c‘.

Solução: Em vez de modificar o `makefile`, a compilação da versão paralela foi feita diretamente pela linha de comando, (gcc -O3 -fopenmp -o vadd_par vadd_par.c -lm).

Problemas Encontrados e Soluções Apresentadas

Problema 3: Disponibilidade da GPU no NPAD

Descrição: Ao verificar as partições disponiveis no NPAD para execução do codigo serial verifique que somente a partição gpu-8-h100 estava disponivel

Solução: Fiz uma alteração do script submit_vadd e alterei a partição gpu-4-a10 para gpu-8-h100.

Configuração do Benchmark

O experimento foi conduzido em um nó do cluster NPAD para uma comparação direta.

Parâmetros do Problema:

- **Tamanho do Vetor (N):** 10^7 elementos.

Versões Comparadas:

- **Sequencial:** Executável `./vadd` rodando em 1 thread de CPU.
- **Paralelo:** Executável `./vadd_par` rodando com 64 threads de CPU (partição `amd-512`).

Métrica Coletada:

- **Tempo de Cômputo (s):** Tempo gasto exclusivamente no laço de adição dos vetores.

Resultados Reais do Benchmark

Tabela: Tempo de cômputo (s) para N = 10^7.

Versão	Threads	Tempo (s)
Sequencial	1	0.003
Paralelo	64	0.050

Observação

A versão paralela foi **16.7x mais LENTA** que a versão sequencial.

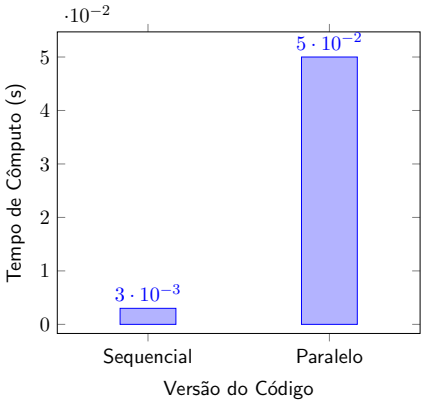

Figura: Comparativo gráfico dos tempos de execução.

Análise dos Resultados

1. Versão Paralela: Alto Custo de Overhead

O tempo de computação de 0.050s foi dominado pelo **overhead** da paralelização.

- O problema (N=10^7) era muito pequeno para 64 threads.
- O custo de criar, gerenciar e sincronizar 64 threads foi maior que o benefício de dividir o trabalho.
- É o análogo a contratar 64 pessoas para uma tarefa que uma só faria em segundos.

2. Versão Sequencial: Alta Otimização

O excelente tempo de 0.003s deve-se a dois fatores:

- **Hardware Potente:** O código rodou em um único núcleo de uma CPU de servidor moderna e de alta performance.
- **Auto-Vetorização (SIMD):** Com a flag `-O3`, o compilador `gcc` otimizou o laço para usar instruções que realizam múltiplas somas em um único ciclo de clock.

Conclusão

- O experimento demonstrou um conceito fundamental: **a paralelização tem um custo (overhead)**. Se o problema não for grande o suficiente, esse custo pode anular ou até reverter os ganhos de desempenho.
- A performance de um código sequencial simples pode ser extraordinária em hardware moderno devido a otimizações agressivas do compilador (como SIMD), não devendo ser subestimada.
- Para que o OpenMP demonstre sua vantagem neste problema, seria necessário aumentar significativamente o tamanho do vetor (N) para que o tempo de computação se torne muito maior que o overhead de gerenciamento das threads.

vadd.c

```c
#include <stdio.h>
#include <omp.h>
#define N 10000000
#define TOL  0.0000001
//
//  This is a simple program to add two vectors
//  and verify the results.
//
//  History: Written by Tim Mattson, November 2017
//
int main()
{

    float a[N], b[N], c[N], res[N];
    int err=0;

    double init_time, compute_time, test_time;
    init_time    = -omp_get_wtime();

   // fill the arrays
   for (int i=0; i<N; i++){
      a[i] = (float)i;
      b[i] = 2.0*(float)i;
      c[i] = 0.0;
      res[i] = i + 2*i;
   }

   init_time    +=  omp_get_wtime();
   compute_time  = -omp_get_wtime();

   // add two vectors
   for (int i=0; i<N; i++){
      c[i] = a[i] + b[i];
   }

   compute_time +=  omp_get_wtime();
   test_time     = -omp_get_wtime();

   // test results
   for(int i=0;i<N;i++){
      float val = c[i] - res[i];
      val = val*val;
      if(val>TOL) err++;
   }

   test_time    +=  omp_get_wtime();

   printf(" vectors added with %d errors\n",err);

   printf("Init time:    %.3fs\n", init_time);
```

```
51      printf("Compute time: %.3fs\n", compute_time);
52      printf("Test time:    %.3fs\n", test_time);
53      printf("Total time:   %.3fs\n", init_time + compute_time + test_time);
54      return 0;
55  }
56
```

vadd_par.c

```c
#include <stdio.h>
#include <omp.h>
#define N 10000000
#define TOL  0.0000001
//
//  This is a simple program to add two vectors
//  and verify the results.
//
//  History: Written by Tim Mattson, November 2017
//
int main()
{

    float a[N], b[N], c[N], res[N];
    int err=0;

    double init_time, compute_time, test_time;
    init_time    = -omp_get_wtime();

   // fill the arrays
   #pragma omp parallel for
   for (int i=0; i<N; i++){
      a[i] = (float)i;
      b[i] = 2.0*(float)i;
      c[i] = 0.0;
      res[i] = i + 2*i;
   }

   init_time    +=  omp_get_wtime();
   compute_time  = -omp_get_wtime();

   // add two vectors
   #pragma omp parallel for
   for (int i=0; i<N; i++){
      c[i] = a[i] + b[i];
   }

   compute_time +=  omp_get_wtime();
   test_time     = -omp_get_wtime();

   // test results
   #pragma omp parallel for reduction(+:err)
   for(int i=0;i<N;i++){
      float val = c[i] - res[i];
      val = val*val;
      if(val>TOL) err++;
   }

   test_time    +=  omp_get_wtime();

```

```
51      printf(" vectors added with %d errors\n",err);
52
53      printf("Init time:    %.3fs\n", init_time);
54      printf("Compute time: %.3fs\n", compute_time);
55      printf("Test time:    %.3fs\n", test_time);
56      printf("Total time:   %.3fs\n", init_time + compute_time + test_time);
57      return 0;
58  }
59
```

makefile

```
#
#  USAGE:
#     make          ... to build the program
#     make test     ... to run the default test case
#

include make.def

EXES= vadd$(EXE)

JAC_OBJS  = jac_solv.$(OBJ) mm_utils.$(OBJ)

all: $(EXES)

vadd$(EXE): vadd.$(OBJ)
    $(CLINKER) $(OPTFLAGS) -o vadd$(EXE) vadd.$(OBJ) $(LIBS)

test: $(EXES)
    for i in $(EXES); do \
            $(PRE)$$i; \
        done

clean:
    $(RM) $(EXES) *.$(OBJ) *.ptx *.cub

.SUFFIXES:
.SUFFIXES: .c .cpp .$(OBJ)

.c.$(OBJ):
    $(CC) $(CFLAGS) -c $<

.cpp.$(OBJ):
    $(CC) $(CFLAGS) -c $<

```

submit_vadd

```
#!/bin/bash
#SBATCH --partition gpu-8-h100
#SBATCH --nodes 1
#SBATCH --time 00:02:00
#SBATCH --job-name vadd
#SBATCH --output vadd-%j.out

cd $SLURM_SUBMIT_DIR

ulimit -s unlimited

./vadd

```

script_18.sh

```
#!/bin/bash
#SBATCH --job-name=vadd_omp
#SBATCH --partition=amd-512
#SBATCH --time=0-0:1

# --- Configuração de Recursos para OpenMP ---
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=64

echo "Job OpenMP iniciado em: $(date)"

# Define o número de threads OpenMP para ser igual ao número de CPUs que o SLURM alocou
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

echo "Executando com $OMP_NUM_THREADS threads..."

# --- Programa a ser executado ---
# Não usamos mpirun ou srun para programas OpenMP simples
./vadd_par

echo "Job concluído em: $(date)"

```