Universidade Federal do Rio Grande do Norte - UFRN
Departamento de Engenharia da Computação e Automação - DCA
DCA3703 - Programação Paralela

Tarefa 2: Pipeline e vetorização

Discente: Quelita Míriam
Docente: Samuel Xavier de Souza

I. INTRODUÇÃO

O avanço dos processadores modernos permitiu a exploração do paralelismo ao nível de instrução (ILP - *Instruction-Level Parallelism*), uma técnica que busca aumentar o desempenho da execução ao processar múltiplas instruções simultaneamente. Entre as principais abordagens utilizadas para maximizar o ILP estão o *pipelining* e a vetorização.

O *pipelining* é uma técnica na qual diferentes estágios da execução de instruções ocorrem simultaneamente, reduzindo o tempo necessário para processar um conjunto de operações. Isso permite que enquanto uma instrução está sendo buscada na memória, outra pode estar sendo decodificada e uma terceira pode estar sendo executada, otimizando o fluxo de dados dentro da CPU.

Já a vetorização ocorre quando o compilador reorganiza operações para que múltiplos dados sejam processados ao mesmo tempo por meio de registradores SIMD (*Single Instruction, Multiple Data*). Essa abordagem é especialmente eficaz para cálculos em laços, onde operações repetitivas podem ser substituídas por instruções mais eficientes que trabalham com blocos de dados simultaneamente.

Neste contexto, este relatório apresenta a implementação e análise de três laços para investigar os efeitos do ILP, considerando diferentes níveis de otimização do compilador GCC (-O0, -O2 e -O3). O objetivo é entender como a estrutura do código e as dependências entre as iterações afetam o desempenho da execução. Além disso, será analisado o impacto da variação na quantidade de variáveis utilizadas na soma paralelizada, permitindo uma compreensão mais aprofundada sobre os limites e benefícios da exploração do ILP.

II. METODOLOGIA

Abaixo encontra-se a implementação do código[1] em C para a tarefa de investigar os efeitos do paralelismo ao nível de instrução (ILP) através da implementação de três laços:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define SIZE 100000000

void initialize_array(int *arr) {
    for (int i = 0; i < SIZE; i++) {
        arr[i] = (i % 10) + 2;
    }
}

long long sum_sequential(int *arr) {
    long long sum = 0;
    for (int i = 0; i < SIZE; i++) {
        sum += arr[i];
    }
    return sum;
}
```

[1] Fonte do código base: GitHub - <https://github.com/quelita2/programacao-paralela/blob/main/topico01/tarefa02/src/main.c>. Acessado em 03 abril 2025.

```
long long sum_parallel(int *arr) {
    long long sum1 = 0, sum2 = 0, sum3 = 0, sum4 = 0;
    for (int i = 0; i < SIZE; i += 4) {
        sum1 += arr[i];
        sum2 += arr[i + 1];
        sum3 += arr[i + 2];
        sum3 += arr[i + 3];
    }
    return sum1 + sum2 + sum3 + sum4;
}

void measure_execution_time(void (*func)(int *), int *arr, const char *label) {
    struct timeval start, end;
    gettimeofday(&start, NULL);
    func(arr);
    gettimeofday(&end, NULL);
    double elapsed_time = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1e6;
    printf("%s: Tempo = %f segundos\n", label, elapsed_time);
}

long long measure_sum_execution_time(long long (*func)(int *), int *arr, const char *label) {
    struct timeval start, end;
    gettimeofday(&start, NULL);
    long long result = func(arr);
    gettimeofday(&end, NULL);
    double elapsed_time = (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1e6;
    printf("%s: Resultado = %lld, Tempo = %f segundos\n", label, result, elapsed_time);
    return result;
}

int main() {
    int *arr = (int *)malloc(SIZE * sizeof(int));
    if (!arr) {
        printf("Erro ao alocar memória!\n");
        return 1;
    }
    measure_execution_time(initialize_array, arr, "\nInicialização do array");

measure_sum_execution_time(sum_sequential, arr, "Soma sequencial");

measure_sum_execution_time(sum_parallel, arr, "Soma paralelizada");

    free(arr);
    return 0;
}
```

Primeiramente, a inicialização de um vetor de 100.000.000 elementos foi preenchido com valores calculados a partir do índice. Em seguida, para a soma sequencial, a soma dos elementos foi realizada de forma acumulativa, introduzindo dependência entre as iterações. Enquanto, para a soma paralela, a dependência foi quebrada utilizando 4 variáveis para realizar somas parciais, explorando o paralelismo ao nível de instrução.

As execuções foram realizadas com diferentes níveis de otimização do compilador GCC:

- **Sem otimização (-O0)**: Código traduzido diretamente para máquina sem otimização.
- **Otimização intermediária (-O2)**: Melhorias de desempenho sem alteração agressiva da estrutura do código.
- **Otimização agressiva (-O3)**: Vetorização e desenrolamento de loops para maximizar desempenho.

Os comandos de compilação utilizados foram:

```
gcc -O0 main.c -o testeO0 && ./testeO0
gcc -O2 main.c -o testeO2 && ./testeO2
gcc -O3 main.c -o testeO3 && ./testeO3
```

Os tempos de execução foram medidos utilizando "*gettimeofday()*", e os resultados foram comparados para avaliar o impacto das otimizações na eficiência do código.

III. RESULTADOS E DISCUSSÕES

Os testes revelaram diferenças significativas no desempenho das diferentes abordagens e níveis de otimização:

Tabela 1: Resultados de Compilação do código fonte

	Tempo de Inicialização do Array	Soma Sequencial	Soma Paralelizada
O0	0.660122 s	0.292072 s	0.145322 s
O2	0.320234 s	0.064576 s	0.052346 s
O3	0.232178 s	0.050861 s	0.052385 s

O tempo de inicialização diminui à medida que o nível de otimização aumenta. Isso ocorre porque as otimizações melhoram o acesso à memória e a eficiência do loop de inicialização.

- Com -O0 (sem otimizações): A soma paralelizada foi mais rápida que a soma sequencial. Isso confirma que, sem otimizações, a abordagem acumulativa sofre mais com dependências entre iterações, enquanto a paralelização manual reduz essas dependências.
- Com -O2: O tempo da soma sequencial já melhora bastante em relação a -O0, mas ainda não supera a paralelizada, sugerindo que o compilador faz algumas otimizações, mas sem vetorizar completamente o código.
- Com -O3: O tempo da soma sequencial e da soma paralelizada ficam praticamente iguais, o que indica que o compilador aplicou otimizações agressivas, incluindo vetorizações e unroll loops. Ele conseguiu eliminar a desvantagem da soma sequencial ao otimizar a execução.

A partir dos tempos é possível notar que com -O0, a soma paralela apresentou melhor desempenho do que a soma sequencial, pois a falta de otimização manteve as dependências da soma acumulativa. Nesse tipo de compilação, o código é traduzido quase diretamente, sem otimizações de registradores e sem eliminação de dependências. Assim, a soma com múltiplas variáveis é mais rápida porque reduz dependências entre iterações, permitindo mais paralelismo ao nível de instrução.

Enquanto com -O3 a situação se inverteu, com a soma sequencial sendo mais eficiente. Isso ocorreu porque o compilador vetoriza e desenrola loops, realizando otimizações automáticas que tornam a abordagem acumulativa tão eficiente quanto a paralela.

Ademais, constatou-se que a utilização de muitas variáveis na soma paralela pode não garantir um desempenho melhor, pois o processador tem um número limitado de registradores, e ao aumentar a quantidade de variáveis, pode ocorrer *spilling*, ou seja, os valores precisam ser armazenados temporariamente na memória em vez de permanecerem em registradores, aumentando a latência. Além disso, a eficiência da execução depende da otimização aplicada pelo compilador e do acesso eficiente à memória cache, podendo ocorrer bottlenecks no

barramento de memória ao processar muitas variáveis simultaneamente.

IV. CONCLUSÃO

Com base nos experimentos realizados, conclui-se que o nível de otimização do compilador tem um impacto significativo no desempenho dos diferentes estilos de soma. A otimização progressiva de -O0 para -O3 resultou em melhorias expressivas na execução, especialmente na soma sequencial, que inicialmente era penalizada por dependências entre iterações, mas se tornou tão eficiente quanto a soma paralelizada com otimizações mais agressivas. Além disso, observou-se que aumentar o número de variáveis na abordagem paralelizada não implica necessariamente em um melhor desempenho, devido às limitações de registradores e possíveis gargalos de memória. Os resultados demonstram a importância de compreender como o compilador otimiza código e como diferentes abordagens impactam a eficiência computacional, evidenciando que estratégias manuais de paralelismo nem sempre superam otimizações automáticas avançadas.

V. REFERÊNCIAS

GHIRARDELLO, G. **O que é processamento paralelo e quais os tipos?** Disponível em: <https://blog.botcity.dev/pt-br/2023/12/05/processamento-paralelo/>.

HIGA, P. **O que é pipeline no processador? Entenda as vantagens da segmentação de instruções** – Tecnoblog. Disponível em: <https://tecnoblog.net/responde/o-que-e-pipeline-processador/>.

TANENBAUM, Andrew S.; AUSTIN, Todd. **Organização Estruturada de Computadores**. 6. ed. São Paulo: Pearson, 2013.