Tarefa 13: Avaliando o Impacto da Afinidade de Threads

Otimizando a Interação entre Software e Hardware

Werbert Arles de Souza Barradas

Universidade Federal do Rio Grande do Norte (UFRN)
Disciplina de Programação Paralela - DCA3703

03 de outubro de 2025

Agenda

1. Teoria da Afinidade de Threads

2. Cláusulas e Variáveis de Controle

O Que é Afinidade de Thread?

Definição

Afinidade de Thread (ou *Thread Affinity*) é o processo de "prender" ou "amarrar" (*binding/pinning*) uma thread de software a um recurso de hardware específico, como um núcleo de CPU.

O Problema: Migração de Threads

Por padrão, o Sistema Operacional pode mover uma thread entre núcleos livremente. Para computação de alto desempenho (HPC), isso é prejudicial por dois motivos principais:

- **Perda de Localidade de Cache:** Se uma thread muda de núcleo, os dados em sua cache L1/L2 são perdidos, forçando uma busca lenta na memória RAM.
- **Ignora a Arquitetura NUMA:** Em sistemas com múltiplos processadores (soquetes), mover uma thread para outro soquete a afasta de sua memória local, aumentando a latência de acesso.

Por Que a Afinidade Importa?

A Solução: Controlar o Posicionamento

Ao fixar uma thread em um núcleo específico, garantimos que ela se beneficie da arquitetura de hardware subjacente. Isso resulta em:

- **Reutilização de Cache:** A thread permanece no mesmo núcleo, permitindo que os mesmos dados sejam acessados rapidamente a partir da cache, o que maximiza a performance.
- **Otimização para NUMA:** As threads podem ser distribuídas de forma inteligente entre os soquetes, aproveitando ao máximo a largura de banda de memória de todo o sistema e minimizando a latência.

Como Controlar a Afinidade em OpenMP

Existem duas maneiras principais de definir a política de afinidade, com diferentes níveis de flexibilidade.

Método 1: Variáveis de Ambiente (Flexível)

É a forma mais prática para experimentação. A variável é definida no terminal **antes** de executar o programa. **Principal Variável:** `OMP_PROC_BIND`

Método 2: Diretiva no Código (Rígido)

A política é "gravada" no código-fonte. Exige uma nova compilação para cada mudança de política. **Principal Cláusula:** `#PRAGMA OMP PARALLEL proc_bind(...)`

EX: master, close, spread

As Políticas de OMP_PROC_BIND

false

O que faz: Desliga a afinidade. Threads podem migrar pelos nucleos de CPU. **Hipótese:** Desempenho potencialmente inferior devido a perdas de cache.

spread

O que faz: Espalha as threads uniformemente pelos recursos. **Hipótese: Provavelmente a melhor política para o NPAD**, pois utiliza todos os soquetes e nós de memória NUMA.

close

O que faz: Agrupa as threads em núcleos fisicamente próximos. **Hipótese:** Pode causar contenção em sistemas NUMA, pois concentra o uso da memória em um só soquete.

master

O que faz: "Empilha"todas as threads no mesmo local da thread mestre. **Hipótese: Provavelmente o pior desempenho**. A contenção por recursos de um só núcleo anulará o paralelismo.

Controle Fino com OMP_PLACES

Definindo os "Lugares" para as Threads

A variável OMP_PLACES permite definir explicitamente quais recursos de hardware são considerados "lugares" onde as threads podem ser alocadas pela política do OMP_PROC_BIND.

- `export OMP_PLACES=sockets`
  - Cada "lugar" é um soquete de CPU. Com OMP_PROC_BIND=spread, o OpenMP tentará colocar o mesmo número de threads em cada soquete.
- `export OMP_PLACES=cores`
  - Cada "lugar" é um núcleo físico. Esta é a configuração mais comum para garantir que cada thread tenha seu próprio núcleo.

Plano de Ação para a Tarefa 13

Objetivo

Avaliar como a escalabilidade do código de Navier-Stokes otimizado muda ao utilizar as diversas políticas de afinidade de threads no nó de computação do NPAD.

Metodologia

1. Utilizar o código `navier_stokes_paralelo_otm2.c` como base.
2. Gerar um conjunto de heatmaps de escalabilidade para cada uma das principais políticas de `OMP_PROC_BIND` (`close`, `spread`, `master`).
3. Comparar os heatmaps resultantes, com foco nos gráficos de **Eficiência** e **Escalabilidade Forte**.
4. Elaborar uma conclusão justificando qual política foi a melhor (e a pior) para este problema, conectando os resultados com a teoria de arquitetura de computadores (Cache e NUMA).

Análise de Desempenho: Afinidade `close`

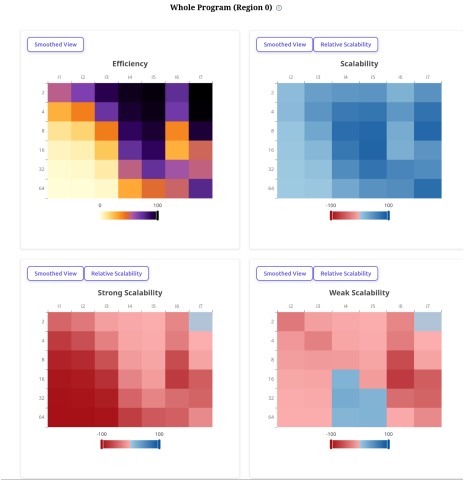

Figura: OMP PROC BIND close

Análise de Desempenho: Afinidade `close`

Principais Observações

- **Eficiência Máxima:** A eficiência é mais alta com um número baixo de núcleos (2-8), onde a política `close` maximiza o uso da cache L3 em um único soquete.
- **Limite de Escalabilidade Forte:** O gráfico *Strong Scalability* (predominantemente vermelho) evidencia a Lei de Amdahl: o ganho de desempenho diminui drasticamente após 16 núcleos.
- **Escalabilidade Fraca:** Geralmente baixa (tons de rosa), indicando que o custo de comunicação/sincronização cresce com o número de núcleos.
- **Conclusão:** A afinidade `close` é ideal para explorar a localidade de dados com poucas threads. No entanto, o algoritmo em si possui gargalos que impedem a escalabilidade para um número massivo de cores (32-64).

Análise de Desempenho: Afinidade `master`

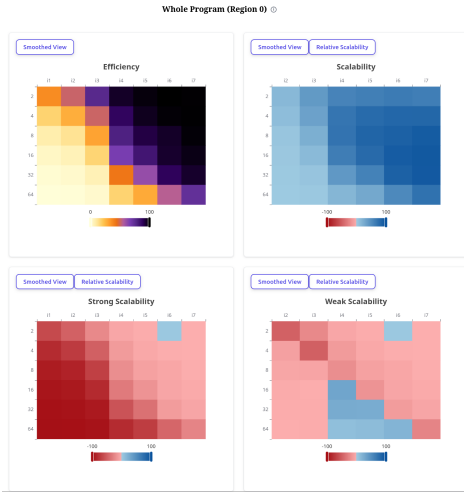

Figura: OMP PROC BIND master

Análise de Desempenho: Afinidade `master`

Principais Observações

- **Similaridade com `close`:** Para até 64 núcleos, o desempenho é quase idêntico ao da política `close`, pois ambas as estratégias confinam as threads a um único soquete, otimizando o uso da cache.
- **Gargalo Artificial:** A política `master` impede o uso do segundo soquete. Ao usar 64 threads, o nó fica subutilizado, e a performance é severamente limitada pela capacidade de um único processador (64 núcleos).
- **Teto de Desempenho:** A escalabilidade forte (gráfico vermelho) estagna completamente após 32 núcleos. Adicionar mais threads não traz benefício, apenas overhead de gerenciamento.
- **Conclusão:** A afinidade `master` é inadequada para este problema ao escalar para o nó inteiro. Ela é menos flexível e apresenta um desempenho inferior à política `close` quando se utilizam todos os núcleos da máquina.

Análise de Desempenho: Afinidade spread

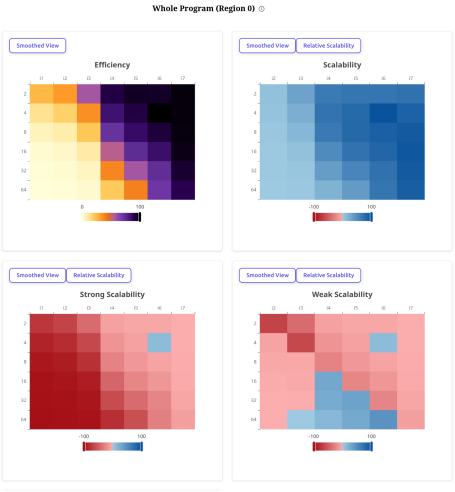

Figura: OMP PROC BIND spread

Análise de Desempenho: Afinidade `spread`

Principais Observações

- **Impacto da Arquitetura:** A política `spread` distribui as threads entre os diferentes complexos de núcleos (CCDs) do processador, maximizando a distância e a latência de comunicação entre elas.
- **Pior Eficiência:** A quebra da localidade da cache L3 resulta na pior eficiência entre todas as políticas. O desempenho só é razoável nos maiores problemas (i6, i7), onde o cômputo intensivo consegue mascarar a latência.
- **Escalabilidade Prejudicada:** A escalabilidade forte (gráfico vermelho) é extremamente ruim, confirmando que o custo de comunicação entre threads distantes supera os benefícios da paralelização.
- **Conclusão:** No contexto de um único soquete, `spread` é a estratégia menos indicada para esta aplicação. Ela serve como um excelente contraexemplo que valida a importância da afinidade `close` para códigos com alta dependência de dados compartilhados.

Conclusões Finais

Conclusão Principal: O Trade-off entre Latência e Largura de Banda

A política de afinidade depende do gargalo da aplicação, que muda com a escala do problema e o número de núcleos. A análise revela um *trade-off* claro entre otimizar para baixa latência de comunicação ou para alta largura de banda de memória.

Análise Comparativa das Estratégias

- `close` **(Otimizada para Latência):** É a melhor estratégia para **escalabilidade forte** e problemas de tamanho pequeno a médio. Ao agrupar as threads, maximiza o uso da cache L3 e minimiza o custo de comunicação entre elas.
- `spread` **(Otimizada para Largura de Banda):** Mostra uma vantagem em cenários de **escalabilidade fraca** com muitos núcleos e problemas massivos. Ao espalhar as threads, distribui melhor o acesso à RAM, utilizando toda a largura de banda de memória do processador.
- `master` (Redundante): Comportou-se como a `close` neste cenário.

navier_stokes_paralelo_master.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

// NT agora é uma constante interna.
#define NT 2000

// Funções auxiliares agora recebem as dimensões como parâmetros
double** allocate_grid(int nx, int ny) {
    double *data = (double*)malloc(nx * ny * sizeof(double));
    double **array = (double**)malloc(nx * sizeof(double*));
    for (int i = 0; i < nx; i++) {
        array[i] = &(data[i * ny]);
    }
    return array;
}

void free_grid(double** array) {
    free(array[0]);
    free(array);
}

int main(int argc, char *argv[]) {
    // Agora esperamos o tamanho da grade (NX) como argumento
    if (argc != 2) {
        fprintf(stderr, "Uso: %s <TAMANHO_DA_GRADE>\n", argv[0]);
        fprintf(stderr, "Exemplo: %s 512\n", argv[0]);
        return 1;
    }
    int NX = atoi(argv[1]);
    int NY = NX; // NY será sempre igual a NX

    // Alocação de memória usa as variáveis NX e NY
    double **u = allocate_grid(NX, NY);
    double **v = allocate_grid(NX, NY);
    double **u_new = allocate_grid(NX, NY);
    double **v_new = allocate_grid(NX, NY);

    // Inicialização usa as variáveis NX e NY
    #pragma omp parallel for
    for (int i = 0; i < NX; i++) {
        for (int j = 0; j < NY; j++) {
            u[i][j] = 1.0; v[i][j] = 0.0;
            double dx = i - NX/2.0, dy = j - NY/2.0;
            double dist = sqrt(dx*dx + dy*dy);
            if (dist < (NX / 25.0)) { // Condição inicial relativa ao tamanho
da grade
                u[i][j] += 2.0 * exp(-dist*dist/(NX/5.0));
                v[i][j] += 1.5 * exp(-dist*dist/(NX/5.0));
```

```
50              }
51          }
52      }
53
54      double start_time = omp_get_wtime();
55
56      #pragma omp parallel proc_bind(master)
57      {
58          for (int step = 0; step < NT; step++) {
59
60              #pragma omp for collapse(2) schedule(static)
61              for (int i = 1; i < NX-1; i++) {
62                  for (int j = 1; j < NY-1; j++) {
63                      double d2u_dx2 = (u[i+1][j] - 2.0*u[i][j] + u[i-1][j]);
64                      double d2u_dy2 = (u[i][j+1] - 2.0*u[i][j] + u[i][j-1]);
65                      double d2v_dx2 = (v[i+1][j] - 2.0*v[i][j] + v[i-1][j]);
66                      double d2v_dy2 = (v[i][j+1] - 2.0*v[i][j] + v[i-1][j]);
67
68                      u_new[i][j] = u[i][j] + (0.001 * 0.01) * (d2u_dx2 +
    d2u_dy2); // DT e NU podem ser fixados
69                      v_new[i][j] = v[i][j] + (0.001 * 0.01) * (d2v_dx2 +
    d2v_dy2);
70                  }
71              }
72
73              #pragma omp for
74              for (int i = 0; i < NX; i++) {
75                  u_new[i][0] = u_new[i][NY-2];
76                  u_new[i][NY-1] = u_new[i][1];
77                  v_new[i][0] = v_new[i][NY-2];
78                  v_new[i][NY-1] = v_new[i][1];
79              }
80
81              #pragma omp for
82              for (int j = 0; j < NY; j++) {
83                  u_new[0][j] = u_new[NX-2][j];
84                  u_new[NX-1][j] = u_new[1][j];
85                  v_new[0][j] = v_new[NX-2][j];
86                  v_new[NX-1][j] = v_new[1][j];
87              }
88
89              #pragma omp single
90              {
91                  double **temp_u = u;
92                  double **temp_v = v;
93                  u = u_new;
94                  v = v_new;
95                  u_new = temp_u;
96                  v_new = temp_v;
97              }
98          }
99      }
100
```

```
101     double end_time = omp_get_wtime();
102     printf("%.6f\n", end_time - start_time);
103 
104     free_grid(u); free_grid(v); free_grid(u_new); free_grid(v_new);
105 
106     return 0;
107 }
108 
```

navier_stokes_paralelo_close.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

// NT agora é uma constante interna.
#define NT 2000

// Funções auxiliares agora recebem as dimensões como parâmetros
double** allocate_grid(int nx, int ny) {
    double *data = (double*)malloc(nx * ny * sizeof(double));
    double **array = (double**)malloc(nx * sizeof(double*));
    for (int i = 0; i < nx; i++) {
        array[i] = &(data[i * ny]);
    }
    return array;
}

void free_grid(double** array) {
    free(array[0]);
    free(array);
}

int main(int argc, char *argv[]) {
    // Agora esperamos o tamanho da grade (NX) como argumento
    if (argc != 2) {
        fprintf(stderr, "Uso: %s <TAMANHO_DA_GRADE>\n", argv[0]);
        fprintf(stderr, "Exemplo: %s 512\n", argv[0]);
        return 1;
    }
    int NX = atoi(argv[1]);
    int NY = NX; // NY será sempre igual a NX

    // Alocação de memória usa as variáveis NX e NY
    double **u = allocate_grid(NX, NY);
    double **v = allocate_grid(NX, NY);
    double **u_new = allocate_grid(NX, NY);
    double **v_new = allocate_grid(NX, NY);

    // Inicialização usa as variáveis NX e NY
    #pragma omp parallel for
    for (int i = 0; i < NX; i++) {
        for (int j = 0; j < NY; j++) {
            u[i][j] = 1.0; v[i][j] = 0.0;
            double dx = i - NX/2.0, dy = j - NY/2.0;
            double dist = sqrt(dx*dx + dy*dy);
            if (dist < (NX / 25.0)) { // Condição inicial relativa ao tamanho
da grade
                u[i][j] += 2.0 * exp(-dist*dist/(NX/5.0));
                v[i][j] += 1.5 * exp(-dist*dist/(NX/5.0));
```

```
            }
        }
    }

    double start_time = omp_get_wtime();

    #pragma omp parallel proc_bind(close)
    {
        for (int step = 0; step < NT; step++) {

            #pragma omp for collapse(2) schedule(static)
            for (int i = 1; i < NX-1; i++) {
                for (int j = 1; j < NY-1; j++) {
                    double d2u_dx2 = (u[i+1][j] - 2.0*u[i][j] + u[i-1][j]);
                    double d2u_dy2 = (u[i][j+1] - 2.0*u[i][j] + u[i][j-1]);
                    double d2v_dx2 = (v[i+1][j] - 2.0*v[i][j] + v[i-1][j]);
                    double d2v_dy2 = (v[i][j+1] - 2.0*v[i][j] + v[i-1][j]);

                    u_new[i][j] = u[i][j] + (0.001 * 0.01) * (d2u_dx2 +
d2u_dy2); // DT e NU podem ser fixados
                    v_new[i][j] = v[i][j] + (0.001 * 0.01) * (d2v_dx2 +
d2v_dy2);
                }
            }

            #pragma omp for
            for (int i = 0; i < NX; i++) {
                u_new[i][0] = u_new[i][NY-2];
                u_new[i][NY-1] = u_new[i][1];
                v_new[i][0] = v_new[i][NY-2];
                v_new[i][NY-1] = v_new[i][1];
            }

            #pragma omp for
            for (int j = 0; j < NY; j++) {
                u_new[0][j] = u_new[NX-2][j];
                u_new[NX-1][j] = u_new[1][j];
                v_new[0][j] = v_new[NX-2][j];
                v_new[NX-1][j] = v_new[1][j];
            }

            #pragma omp single
            {
                double **temp_u = u;
                double **temp_v = v;
                u = u_new;
                v = v_new;
                u_new = temp_u;
                v_new = temp_v;
            }
        }
    }

```

```
101     double end_time = omp_get_wtime();
102     printf("%.6f\n", end_time - start_time);
103 
104     free_grid(u); free_grid(v); free_grid(u_new); free_grid(v_new);
105 
106     return 0;
107 }
108 
```

navier_stokes_paralelo_spread.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

// NT agora é uma constante interna.
#define NT 2000

// Funções auxiliares agora recebem as dimensões como parâmetros
double** allocate_grid(int nx, int ny) {
    double *data = (double*)malloc(nx * ny * sizeof(double));
    double **array = (double**)malloc(nx * sizeof(double*));
    for (int i = 0; i < nx; i++) {
        array[i] = &(data[i * ny]);
    }
    return array;
}

void free_grid(double** array) {
    free(array[0]);
    free(array);
}

int main(int argc, char *argv[]) {
    // Agora esperamos o tamanho da grade (NX) como argumento
    if (argc != 2) {
        fprintf(stderr, "Uso: %s <TAMANHO_DA_GRADE>\n", argv[0]);
        fprintf(stderr, "Exemplo: %s 512\n", argv[0]);
        return 1;
    }
    int NX = atoi(argv[1]);
    int NY = NX; // NY será sempre igual a NX

    // Alocação de memória usa as variáveis NX e NY
    double **u = allocate_grid(NX, NY);
    double **v = allocate_grid(NX, NY);
    double **u_new = allocate_grid(NX, NY);
    double **v_new = allocate_grid(NX, NY);

    // Inicialização usa as variáveis NX e NY
    #pragma omp parallel for
    for (int i = 0; i < NX; i++) {
        for (int j = 0; j < NY; j++) {
            u[i][j] = 1.0; v[i][j] = 0.0;
            double dx = i - NX/2.0, dy = j - NY/2.0;
            double dist = sqrt(dx*dx + dy*dy);
            if (dist < (NX / 25.0)) { // Condição inicial relativa ao tamanho
da grade
                u[i][j] += 2.0 * exp(-dist*dist/(NX/5.0));
                v[i][j] += 1.5 * exp(-dist*dist/(NX/5.0));
```

```
            }
        }
    }

    double start_time = omp_get_wtime();

    #pragma omp parallel proc_bind(spread)
    {
        for (int step = 0; step < NT; step++) {

            #pragma omp for collapse(2) schedule(static)
            for (int i = 1; i < NX-1; i++) {
                for (int j = 1; j < NY-1; j++) {
                    double d2u_dx2 = (u[i+1][j] - 2.0*u[i][j] + u[i-1][j]);
                    double d2u_dy2 = (u[i][j+1] - 2.0*u[i][j] + u[i][j-1]);
                    double d2v_dx2 = (v[i+1][j] - 2.0*v[i][j] + v[i-1][j]);
                    double d2v_dy2 = (v[i][j+1] - 2.0*v[i][j] + v[i-1][j]);

                    u_new[i][j] = u[i][j] + (0.001 * 0.01) * (d2u_dx2 +
d2u_dy2); // DT e NU podem ser fixados
                    v_new[i][j] = v[i][j] + (0.001 * 0.01) * (d2v_dx2 +
d2v_dy2);
                }
            }

            #pragma omp for
            for (int i = 0; i < NX; i++) {
                u_new[i][0] = u_new[i][NY-2];
                u_new[i][NY-1] = u_new[i][1];
                v_new[i][0] = v_new[i][NY-2];
                v_new[i][NY-1] = v_new[i][1];
            }

            #pragma omp for
            for (int j = 0; j < NY; j++) {
                u_new[0][j] = u_new[NX-2][j];
                u_new[NX-1][j] = u_new[1][j];
                v_new[0][j] = v_new[NX-2][j];
                v_new[NX-1][j] = v_new[1][j];
            }

            #pragma omp single
            {
                double **temp_u = u;
                double **temp_v = v;
                u = u_new;
                v = v_new;
                u_new = temp_u;
                v_new = temp_v;
            }
        }
    }

```

```
101     double end_time = omp_get_wtime();
102     printf("%.6f\n", end_time - start_time);
103 
104     free_grid(u); free_grid(v); free_grid(u_new); free_grid(v_new);
105 
106     return 0;
107 }
108 
```