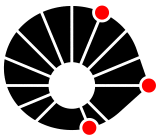

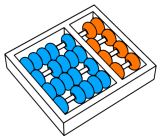

Análise da Otimização de Loops Do-across

E. Silva *G. Araújo*

Relatório Técnico - IC-PFG-23-17
Projeto Final de Graduação
2023 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Análise da Otimização de Laços Do-across

Emerson José Bezerra da Silva Guido Costa Souza de Araújo

Resumo

Este trabalho é o relatório de um Projeto de Conclusão de Curso tratando o problema da paralelização de loops com dependências do tipo do-across. Uma proposta de melhoramento foi levantada e analisada, tendo como base o posicionamento da seção de execução sequencial, e essa proposta foi comparada com o comportamento atual de um compilador, em testes manuais. Os resultados da comparação foram um baixo potencial de otimização, com ganhos médios observados de $1,33\%$ usando esta estratégia e um desempenho mais que razoável das técnicas atualmente utilizadas para tratar destes tipos de loops.

1 Introdução

Desde o surgimento do primeiro computador operacional, uma das questões centrais da computação tem sido o ganho de desempenho. Até bastante recentemente na história, esse ganho foi realizado utilizando técnicas sequenciais, i.e, permitindo que as mesmas operações fossem executadas de maneira mais veloz, porém, diante de um retorno decrescente dessas técnicas, houve uma migração para técnicas paralelas, que permitem a execução de múltiplas operações simultâneas. Esse novo campo de otimização trazia dificuldades maiores do que aquelas associadas à otimização de execução sequencial, porém.

A principal dificuldade associada a este problema é uma limitação fundamental do paralelismo: a execução paralela se presta melhor a certos problemas do que a outros, e os ganhos das técnicas de otimização com paralelismo são desiguais entre as classes de problemas. A classe de maior ganho é a dos problemas trivialmente paralelos, em que o problema é facilmente dividido em várias instâncias que, por serem independentes, podem ser facilmente paralelizadas. Uma classe mais genérica e possivelmente mais comum é a dos problemas que possuem dependências de dados, isto é, uma operação depende do resultado de uma operação anterior, e portanto a segunda operação não pode ser realizada em paralelo com a primeira.

Um exemplo da primeira classe de problema é a soma de duas matrizes, em que cada célula pode ser somada e armazenada de maneira independente de todas as outras células. Já da segunda classe problema, a resolução de um sistema linear por um método iterativo é um problema de paralelização limitada, já que a aproximação X_{n+1} depende do cálculo de X_n.

Outra dificuldade associada ao uso de paralelismo é a necessidade de declarar explicitamente o seu uso. Como diversos problemas na computação, o custo dessa dificuldade pode ser reduzido por uma camada de abstração. Nesse caso, a abstração poderá ser realizada

através de uma API que forneça um modo de descrever as operações paralelas sem levar em conta as capacidades e os modelos de paralelismo do sistema que irá executar o código. Uma API assim é o OpenMP, que possui compatibilidade com a maioria dos compiladores.

Na interseção dessas duas questões, temos a questão do modo de expressar dependências de dados e aproveitar o paralelismo que seja possível apesar delas. No caso específico do OpenMP, que será considerado, essas dependências podem ser expressas através da diretiva ordered, que expressa que parte de uma construção deve ser executada de forma sequencial e outra parte pode ser paralelizada.

Diante disso, esse trabalho busca tratar o comportamento e as possibilidade de otimização de uma construção assim, considerando o caso de um loop do-across [7] que possua uma parte paralelizável e uma parte necessariamente sequencial. Como, então, um compilador se comporta diante da diretiva ordered? Como ele se comporta se uma parte não necessariamente sequencial do código for posta depois da diretiva ordered? Quanto de performance se perde nesses casos, ou, quanta performance poderia ser ganha removendo essa execução sequencial desnecessária? Essas são as perguntas que, através de benchmarks com diferentes posições para o ordered e análise do comportamento de códigos compilados com OpenMP, esse trabalho tentará resolver.

O ambiente de testes usará um simulador para RISC-V e uma instalação Linux mínima, além de testes retirados do conjunto de benchmarks Polybench.

Foi percebido então que, mesmo com a análise do código gerado indicando custo extremamente baixo de sincronização por iteração individual, loops particularmente aninhados ainda não seriam paralelizáveis com do-across, por conta do atraso combinado de todas as iterações poder superar qualquer ganho de performance com a própria paralelização. Com esses testes, o trabalho chegou no resultado de que as possibilidades do otimização com a técnica considerada são bastante variáveis em diferentes casos de uso, porém geralmente baixas, com uma média de $1,33\%$ e uma máxima de $10,53\%$ em relação ao código básico do loop com ordered.

2 Conceitos

2.1 Paralelismo

A paralelização é uma técnica que permite a execução mais rápida de programas através da duplicação de estruturas para permitir execução simultânea de múltiplos cálculos. O desenvolvimento de hardware para paralelização tem como vantagem sua simplicidade, dado não ter impactos tão fortes quanto a técnica até então dominante para o avanço do hardware, o aumento da frequência, no consumo e na complexidade da construção de um circuito. Porém, o paralelismo apresenta o problema de não ser, em grande parte das suas formas, transparente para o desenvolvedor. O código escrito para execução sequencial raramente pode ser automaticamente executado de modo paralelo e, dadas certas condições, um problema não pode sequer ser escrito de forma paralela, por conta das suas dependências, e assim o paralelismo introduz três complexidades: seus ganhos são desiguais entre diferentes problemas, dificultando medir o impacto de uma nova técnica para uma classe de problemas, ele exige, em geral, código escrito explicitamente para explorar o paralelismo e, por

fim, código paralelo correto exige consideração das dependências envolvidas.

Num aspecto do problema relevante para este trabalho, o primeiro ponto oferece grande espaço para a aplicação de compiladores para elaboração para código paralelo, considerando a possibilidade de reduzirem a dificuldade do uso das tecnologias pelo programador, abstraindo tanto quanto possível a complexidade da paralelização no funcionamento interno do compilador. O segundo ponto, porém, apresenta a principal dificuldade envolvida na compilação de códigos destinados a serem executados de modo paralelo: a análise das dependências pode ser complexa, significando impacto nos tempos de compilação, e, por precisar ser conservadora para garantir a corretude, pode não explorar completamente as possibilidades da paralelização oferecidas por determinado algoritmo. Por conta disso, o problema da paralelização geralmente não é resolvido de forma totalmente transparente aos programadores.

2.2 OpenMP

Uma ferramenta utilizada para escrever código que, ao ser compilado, será executado de maneira paralela é o OpenMP [9]. Focado especialmente no paralelismo baseado em multiprocessadores, ele fornece uma API em que o programador, a partir de anotações em linhas de código e estruturas de controle, pode definir de que forma elas serão paralelizadas. Assim, com as informações extras fornecidas pelo programador, o compilador pode produzir código paralelo com necessidade bem menor de análise, dado que certas conclusões sobre o código são fornecidas pelo programador, que assume então responsabilidade pela sua corretude. A implementação do OpenMP na linguagem C, que será considerada no restante desse trabalho, utiliza pragmas, diretivas que não fazem parte do código-fonte mas que fornecem instruções para como o processador deve tratar aquele código, para essas anotações. Sendo que pragmas são tratados como comentários quando não podem ser interpretados pelo compilador, acontecerá que um programa escrito para ser paralelizado pelo OpenMP, ao ser compilado na ausência deste se tornará um programa sequencial perfeitamente funcional. Um exemplo de código escrito com as funções do OpenMP que serão necessárias para a consideração desse trabalho é dado abaixo:

```
int main(int argc, char **argv)
{
    int in[1000000];
    int out[1000000];

    in[0] = out[0] = 0;

    #pragma omp parallel for
    for (int i = 1; i < 1000000; i++) {
        in[i] = i * i;
        #pragma omp ordered
        out[i] = out[i - 1] + in[i];
    }
```

```
    return 0;
}
```

No fragmento de código acima, é possível ver a instrução que indica uma seção a ser paraleliza, expressa com *#pragma omp parallel*, além de uma especificação da natureza da seção a ser paralelizada, no caso do fragmento acima for. Além disso, para o trecho de código que depende das execuções anteriores, é utilizada uma segunda instrução para o OpenMP, na forma de *#pragma omp ordered*. Com esses dois comandos, o seguinte resultado é produzido: a primeira linha do for pode ser executada em tantas instâncias paralelas quanto a máquina-destino permitir, enquanto que a segunda linha fica em espera em relação à iteração anterior do mesmo loop, evitando que a dependência em relação ao valor $out[i-1]$, produzido anteriormente, faça com que o código produza resultados errados.

2.3 Loops do-across

O código apresentado no exemplo acima não foi arbitrário, mas apresenta um exemplo trivialmente simples da classe de problemas paralelos a ser tratada nesse trabalho. Dada uma estrutura de repetição, por exemplo o for acima, cada iteração pode ser parametrizada por um valor, no caso de um loop for por uma série de números, convenientemente a própria variável de iteração, possui uma parte de execução independente e uma parte cujos resultados dependem das iterações anteriores.

Essa parte sequencial gera a necessidade de comunicação entre as diversas iterações para sincronização. Isso signfica que, antes da sua parte sequencial, a thread executando uma iteração precisa entrar em um estado de espera até que o resultado da thread anterior esteja disponível e, após produzir seu próprio resultado, precisa informar sua conclusão para as threads seguintes que possam estar esperando por esse resultado. Em geral, então, o funcionamento de um loop DOACROSS seria como descrito no pseudocódigo abaixo:

```
para index de 1 ate N
    parte paralela
    aguardar index - 1
    parte sequencial
    informar conclusão de index
fimpara
```

De modo mais geral, porém, e isso será relevante para as otimizações aqui discutidas, se pode ter mais de uma parte paralela, como por exemplo:

```
para index de 1 ate N
    primeira parte paralela
    aguardar index - 1
    parte sequencial
    informar conclusão de index
    segunda parte paralela
fimpara
```

É importante destacar que a segunda parte paralela, apesar de estar escrita depois da parte sequencial, não pode depender de nenhum dos seus resultados, dado que isso tornaria ela,

também, sequencial. Assim, a primeira forma do loop pode ser tomada como a forma geral e canônica de um loop DOACROSS, sendo que os loops escritos na segunda forma são redutíveis à primeira forma, apresentando apenas, porém, maior dificuldade para o compilador os identificar, devido à forma como foram escritos pelo programador.

Além disso, é preciso ressaltar que a dependência entre a parte sequencial e a parte paralela é uma dependência de dados real é necessária, por definição, para a caracterização de um loop DOACROSS. Portanto, qualquer solução para o problema que seja capaz de produzir código correto precisará garantir a execução da parte sequencial de $index = 3$ depois da parte paralela de $index = 3$ e depois da parte sequencial de $index = 2$.

2.4 gem5

Para a realização dos testes na arquitetura RISC-V, um simulador foi utilizado. Dadas as preocupações com performance não precisarem de precisão absoluta na simulação da arquitetura do processador, um simulador no nível de sistema foi escolhido no lugar de um simulador com precisão de ciclos. Entre esses simuladores, foi escolhido o gem5 por possuir documentação razoável, grandes possibilidades de personalização e por ser um projeto aberto.

3 Justificativa

Com o que foi apresentado, os loops do-across apresentam um problema de paralelismo que pode ser estudado, principalmente no que diz respeito à eficiência da sua implementação. O tratamento mais eficiente das dependências pode reduzir o tempo de espera desnecessária, em que a parte do trabalho do loop que pode ser paralelizada espera por uma iteração anterior ao invés de prosseguir sua execução, gerando tempo ocioso. A analogia com a questão das dependências entre instruções num processador com execução fora de ordem, problema já consideravelmente trabalhado pelo algoritmo de Tomasulo e com implementação em hardware bem conhecida, indica que a possibilidade de uma extensão em hardware do tratamento de dependências seria algo a ser estudado. Além disso, a existência de recursos no OpenMP para o tratamento de loops com uma parte dependente abre questões também sobre a forma como esse problema é tratado por software atualmente.

4 Objetivos

Esse trabalho então se propõe a considerar quão grande é o impacto do tempo de comunicação entre as threads para o tempo de execução de um loop do-across, dado que esse tempo poderia ser reduzido por uma hipotética implementação em hardware do tratamento das dependências, e como um compilador específico disponível para uso amplo, o GCC, trata essas dependências na fase de otimização. O trabalho então será composto de alguma análise de código da implementação do OpenMP e de alguns testes realizados com programas compilados pelo GCC para medição da eficiência das otimizações aplicadas por ele. Todos esses testes serão restritos à arquitetura de processador RISC-V, que por ser aberta

e comparativamente simples do ponto de vista estrutural, está melhor posicionada para alguma possível extensão de hardware que mostre possibilidades de ganhos de performance.

5 Desenvolvimento do Trabalho

5.1 Estrutura de um loop ordered

O primeiro passo a ser feito é considerar o código gerado ao compilar um loop contendo uma dependência do-across. Um exemplo simples será usado, produzindo dois vetores: um contém os quadrados dos números de 0 a 10 (podendo, portanto, ser gerado em paralelo) e outro contém a soma desses quadrados (dependendo, portanto, da iteração anterior). Para uma última parte, um terceiro vetor será criado somente com os valores da variável de iteração, para mostrar o comportamento do ordered para trechos independentes postos depois do ordered. O código para isso, em C, é dado abaixo:

```
int main() {
    int x[11];
    int y[11];
    int z[11];
    x[0] = y[0] = z[11] = 0;
    #pragma omp parallel for ordered
    for (int i = 1; i <= 10; i++){
        x[i] = i * i;
        #pragma omp ordered
        y[i] = y[i-1] + x[i];
        z[i] = i;
    }
}
```

Do ponto de vista da performance, esse não é um código particularmente útil para se analisar. Ele permite, porém, visualizar a forma como é implementado o loop paralelo pelo openMP. Com a opção de otimização -O3, o resultado da compilação desse código em assembly será

```
main._omp_fn.0:
        (...)
        call    GOMP_loop_ordered_static_start
        (...)
.L4:
        (...)
.L3:
        (...) #x[i] = i * i
        call    GOMP_ordered_start
        (...) #y[i] = y[i-1] + x[i];
        call    GOMP_ordered_end
```

```
        (...) #z[i] = i;
        call    GOMP_loop_ordered_static_next
        (...)
.L2:
        call    GOMP_loop_end_nowait
        (...)
main:
        #inicialização do programa
        (...)
        call    GOMP_parallel
        #finalização do programa
        (...)
        jr      ra
```

Em resumo do que acontece, o loop é transformado em uma função, o início da função envolve uma função estática *GOMP_loop_ordered_static_start*, que é responsável pela atribuição das iterações às threads, portanto não se relaciona particularmente com a análise a ser feita, assim como a função de finalização do loop no final da função gerada, *GOMP_loop_end_nowait*, chamada logo antes do retorno da função. As partes relevantes para o entendimento do funcionamento do ordered seriam então três funções, presentes no corpo de cada iteração do loop: as funções *GOMP_ordered_start*, *GOMP_ordered_end* e *GOMP_loop_ordered_static_next*. A primeira função inicia a zona de exclusão associada ao loop, e não retorna controle para o código do loop antes da execução anterior do loop estar finalizada. A segunda função marca o final da região de exclusão. A terceira função, executada apenas no fim do loop, serve para autorizar a próxima iteração a entrar na zona marcada como ordenada. O código utilizado para implementar essas funções está disponível e pode ser comentado:

```
gomp_ordered_sync (void)
{
  struct gomp_thread *thr = gomp_thread ();
  struct gomp_team *team = thr->ts.team;
  struct gomp_work_share *ws = thr->ts.work_share;

  /* Work share constructs can be orphaned.  But this clearly means that
     we are the only thread, and so we automatically own the section.  */
  if (team == NULL || team->nthreads == 1)
    return;

  /* ??? I believe it to be safe to access this data without taking the
     ws->lock.  The only presumed race condition is with the previous
     thread on the queue incrementing ordered_cur such that it points
     to us, concurrently with our check below.  But our team_id is
     already present in the queue, and the other thread will always
     post to our release semaphore.  So the two cases are that we will
     either win the race an momentarily block on the semaphore, or lose
```

```
        the race and find the semaphore already unlocked and so not block.
        Either way we get correct results.  */

      if (ws->ordered_owner != thr->ts.team_id)
        {
          gomp_sem_wait (team->ordered_release[thr->ts.team_id]);
          ws->ordered_owner = thr->ts.team_id;
        }
    }

    GOMP_ordered_start (void)
    {
      gomp_ordered_sync ();
    }
```

A função *GOMP_ordered_start* é simplesmente uma chamada à função *gomp_ordered_sync*, também mostrada acima. Duas coisas podem ser percebidas nela, primeiro alguma lógica própria para evitar uma trava eterna da função, depois, uma lógica bastante padrão de espera por um semáforo e então tomada de posse de uma trava. Como o tratamento canônico de regiões de exclusão é justamente o uso de semáforos, não existe muito o que possa ser feito quanto a essa função, e a parte responsável por evitar a espera eterna serviria para evitar um único condicional e dependeria de alterações profundas na lógica como as threads são identificadas, não se oferecendo, portanto, como uma estratégia particularmente eficiente para ganhar performance para loops assim.

Já a função *GOMP_loop_ordered_static_next*

```
void
gomp_ordered_next (void)
{
  struct gomp_thread *thr = gomp_thread ();
  struct gomp_team *team = thr->ts.team;
  struct gomp_work_share *ws = thr->ts.work_share;
  unsigned index, next_id;

  /* Work share constructs can be orphaned.  */
  if (team == NULL || team->nthreads == 1)
    return;

  /* We're no longer the owner.  */
  ws->ordered_owner = -1;

  /* If there's only one thread in the queue, that must be us.  */
  if (ws->ordered_num_used == 1)
    {
      /* We have a similar situation as in gomp_ordered_first
```

```
        where we need to post to our own release semaphore.  */
      gomp_sem_post (team->ordered_release[thr->ts.team_id]);
      return;
    }

  /* If the queue is entirely full, then we move ourself to the end of
     the queue merely by incrementing ordered_cur.  Only if it's not
     full do we have to write our id.  */
  if (ws->ordered_num_used < team->nthreads)
    {
      index = ws->ordered_cur + ws->ordered_num_used;
      if (index >= team->nthreads)
        index -= team->nthreads;
      ws->ordered_team_ids[index] = thr->ts.team_id;
    }

  index = ws->ordered_cur + 1;
  if (index == team->nthreads)
    index = 0;
  ws->ordered_cur = index;

  next_id = ws->ordered_team_ids[index];
  gomp_sem_post (team->ordered_release[next_id]);
}
```

A função que passa o controle para a próxima iteração também não é particularmente complexa e trabalha com o conceito de semáforos e com o modo de identificação das threads envolvidas na exclusão mútua. Existe novamente um possível espaço para otimização simplificando o sistema de identificação das threads, mas os ganhos potenciais não parecem particularmente elevados, dado que o código de tratamento dos casos especiais não é de forma alguma complexo, e as mudanças necessárias para conseguir qualquer ganho seriam bem mais estruturais, alterando a forma básica de funcionamento do OpenMP, de modo que essa alternativa não parece atraente.

Por fim, a função *GOMP_ordered_end* tem como código

```
void
GOMP_ordered_end (void)
{
}
```

Isso significa que o marco final da seção reservada é meramente simbólico, e não existe qualquer diferença na forma como a linha que define $z[i]$, e pode ser executada de forma completamente independente, é tratada, ela é executada exatamente da mesma forma que a linha que realmente precisa ser ordenada, acima dela. Para esse código de testes extremamente simples, o impacto da diferença não é alto, para um código consideravelmente mais complexo essa diferença poderia ser significativa.

5.2 Performance da sincronização

Além dos códigos envolvidos, para análise do potencial ganho de performance envolvido numa alteração da forma do tratamento das dependências, é preciso entender quanto tempo é gasto nas funções acima, que são responsáveis pela sincronização entre as diferentes threads ordered geradas pelo OpenMP. Esse tempo dará um limite superior para os potenciais ganhos com qualquer otimização do processo de sincronização/tratamento de dependências, já que, mesmo que o tempo final seja reduzido a zero, não é possível ganhar mais performance do que simplesmente removendo o tempo envolvido no processo na versão atual.

Os resultados dos testes com a eficiência dessa função foram

Resultados obtidos com o Gem5 [1] alterado a classe do processador Risc-V utilizado na simulação. Para a execução dos kernel, foi aplicado uma versão bootavel do linux compilador a partir do Cross-compiler gerado pelo RISC-V GNU Compiler Toolchain [8]

- Atomic: ∼32 ciclos
- In-order: 200∼220 ciclos
- Out-of-order: 4∼6 ciclos

Em todas as classes, foi observado um padrão: o número de ciclos necessários para realizar a primeira sincronização é superior as demais, no caso do processador fora de ordem, era necessário entre 600 a 3000 ciclos. No entanto, esse valor pode ser amortizado, caso o kernel execução realize grandes quantidades de iterações.

E esses resultados indicam que o código já é suficiente otimizado, principalmente considerando a amortização do custo e portanto E esses resultados não parecem justificar o desenvolvimento de uma solução em hardware usando algo como uma adaptação da técnica de Tomasulo [6], uma técnica para resolução em hardware de dependências de dados normalmente usada para paralelismo em nível de instrução, para tratar as dependências de dados. Os ganhos potenciais seriam muito pequenos, mesmo no melhor dos casos, para justificar uma extensão do processador para lidar com esse tipo de loop.

5.3 Comportamento do GCC diante de outros loops ordered

Para todos os testes, a configuração utilizada para o simulador foram quatro núcleos, do tipo de processador 'atomic-simple-CPU', com $256MB$ de RAM DDR3, utilizando, para as restantes opções, o modo padrão de simulação RISC-V oferecido pelo gem5. Além disso, a simulação foi realizada usando o Berkeley Boot loader, um supervisor do ambiente de execução, uma imagem pré-compilada do Linux para RISC-V e uma imagem de disco para armazenar os programas compilados juntamente com um script de inicialização para invocá-los. Esses componentes de software estão todos disponíveis num repositório sobre a simulação usando gem5, juntamente com tutoriais [3] para a produção desses artefatos a partir de código fonte.

A medida de tempo utilizada foi produzida pelo próprio código do Polybench, usando a opção de compilação $-DPOLYBENCH_TIME$.

5.4 Atax

O primeiro benchmark utilizado, para ser usado como referência por alguns dos testes futuros, é o teste atax, extraído do conjunto de testes Polybench [5], transposição de matrizes e multiplicação vetorial, executado na sua forma original, de modo totalmente sequencial. Esse teste foi escolhido por algumas características, que podem ser percebidas no seu loop principal, que permitem testar os limites da otimização realizada, sendo, de certa forma, o melhor caso possível para a otimização de um loop de execução sequencial. A maior parte do trabalho realizado pelo benchmark se encontra, também, em um único loop, simplicidade que apresenta vantagens para a análise. O loop que realiza a maior parte das operações envolvidas na programa é dado abaixo, na sua versão original:

```
static void kernel_atax(int m, int n,
                 DATA_TYPE POLYBENCH_2D(A,M,N,m,n),
                 DATA_TYPE POLYBENCH_1D(x,N,n),
                 DATA_TYPE POLYBENCH_1D(y,N,n),
                 DATA_TYPE POLYBENCH_1D(tmp,M,m))
{
  int i, j;

#pragma scop
  for (i = 0; i < _PB_N; i++)
    y[i] = 0;
  for (i = 0; i < _PB_M; i++)
    {
      tmp[i] = SCALAR_VAL(0.0);
      for (j = 0; j < _PB_N; j++)
            tmp[i] = tmp[i] + A[i][j] * x[j];
      for (j = 0; j < _PB_N; j++)
            y[j] = y[j] + A[i][j] * tmp[i];
    }
#pragma endscop
}
```

Notar que o loop mais externo, iterando a variável i, não possui quaisquer dependências entre iterações e, além disso, compõe a absoluta maior parte do trabalho realizado pelo kernel. Fora dele, existe apenas a inicialização do vetor y em um loop separado, acima.

Dessa forma, esse loop está na categoria dos chamados 'vergonhosamente paralelos' e não existe nenhuma dependência do-across nele, além de não existir nenhum efeito colateral da ordem de execução, permitindo, no caso ideal, reduzir uma execução ordenada dele a uma desordenada e paralela, por otimização, sem violar nenhum dos pressupostos associados à compilação de código em C numa otimização mais alta.

A execução do loop nessa forma será o primeiro benchmark realizado. O segundo benchmark envolve a realização de uma alteração no loop a partir de pragmas do OpenMP, de modo a forçar a execução sequencial dele. Essa execução é perfeitamente desnecessária e

não tem efeitos observáveis no programa, e para instruir o compilador a realizá-la, o código será modificado para

```
#pragma omp parallel for ordered
for (i = 0; i < _PB_M; i++)
{
  #pragma omp ordered
  tmp[i] = SCALAR_VAL(0.0);
  for (j = 0; j < _PB_N; j++)
        tmp[i] = tmp[i] + A[i][j] * x[j];
  for (j = 0; j < _PB_N; j++)
        y[j] = y[j] + A[i][j] * tmp[i];
}
```

Instruindo cada thread do loop a só ser executada após a finalização da thread anterior, o que produziria a máxima ineficiência na execução paralela, caso a instrução fornecida pelo pragma seja ingenuamente executada.

Um caso mais razoável, para um terceiro tempo de execução a ser medido, seria restringir a seção ordenada para não ocupar todo o código. Marcando apenas o final do código, nesse caso o último loop interno, permitiria paralelização das partes anteriores do loop externo, e portanto um ganho de velocidade em relação à execução completamente sequencial. O código para isso será

```
#pragma omp parallel for ordered
for (i = 0; i < _PB_M; i++)
{
  tmp[i] = SCALAR_VAL(0.0);
  for (j = 0; j < _PB_N; j++)
        tmp[i] = tmp[i] + A[i][j] * x[j];
  #pragma omp ordered
  for (j = 0; j < _PB_N; j++)
        y[j] = y[j] + A[i][j] * tmp[i];
}
```

Por fim, também para efeitos de comparação, o código pode ser executado de forma totalmente paralela, e será obtido o caso ótimo da paralelização desse problema em específico, utilizando o máximo possível dos recursos do processador para a execução. O código para isso envolverá simplesmente a remoção das clausulas ordered, produzindo

```
#pragma omp parallel for
for (i = 0; i < _PB_M; i++)
{
  tmp[i] = SCALAR_VAL(0.0);
  for (j = 0; j < _PB_N; j++)
        tmp[i] = tmp[i] + A[i][j] * x[j];
```

```
      for (j = 0; j < _PB_N; j++)
            y[j] = y[j] + A[i][j] * tmp[i];
    }
```

5.5 Deriche

Em contraste com os testes realizado até agora, um programa de benchmark mais complexo e com condições mais difíceis de otimização será também utilizado: o teste marcado como Deriche implementa o algoritmo de detecção de bordas de Deriche [5] na sua forma mais genérica, permitindo também suavização de imagens. Esse teste tem duas diferenças para o atax, sendo elas: ele é composto por um número maior de loops independentes e, em segundo lugar, a execução de uma iteração de um loop é quase completamente dependente da iteração, existindo então pouco que posso ser feito para paralelizar o loop.

O código do kernel da execução desse benchmark, com seus vários loops, está a seguir:

```
    static
void kernel_deriche(int w, int h, DATA_TYPE alpha,
       DATA_TYPE POLYBENCH_2D(imgIn, W, H, w, h),
       DATA_TYPE POLYBENCH_2D(imgOut, W, H, w, h),
       DATA_TYPE POLYBENCH_2D(y1, W, H, w, h),
       DATA_TYPE POLYBENCH_2D(y2, W, H, w, h)) {
    //declaração de variáveis
    (...)

#pragma scop
   //inicialização de variáveis
   (...)

   for (i=0; i<_PB_W; i++) {
        //inicialização de variáveis
        (...)
        for (j=0; j<_PB_H; j++) {
            y1[i][j] = a1*imgIn[i][j] + a2*xm1 + b1*ym1 + b2*ym2;
            xm1 = imgIn[i][j];
            ym2 = ym1;
            ym1 = y1[i][j];
        }
    }

    for (i=0; i<_PB_W; i++) {
        //inicialização de variáveis
        (...)
        for (j=_PB_H-1; j>=0; j--) {
            y2[i][j] = a3*xp1 + a4*xp2 + b1*yp1 + b2*yp2;
            xp2 = xp1;
```

```
            xp1 = imgIn[i][j];
            yp2 = yp1;
            yp1 = y2[i][j];
        }
    }

    for (i=0; i<_PB_W; i++)
        for (j=0; j<_PB_H; j++) {
            imgOut[i][j] = c1 * (y1[i][j] + y2[i][j]);
        }

    for (j=0; j<_PB_H; j++) {
        //inicialização de variáveis
        (...)
        for (i=0; i<_PB_W; i++) {
            y1[i][j] = a5*imgOut[i][j] + a6*tm1 + b1*ym1 + b2*ym2;
            tm1 = imgOut[i][j];
            ym2 = ym1;
            ym1 = y1 [i][j];
        }
    }


    for (j=0; j<_PB_H; j++) {
        //inicialização de variáveis
        (...)
        for (i=_PB_W-1; i>=0; i--) {
            y2[i][j] = a7*tp1 + a8*tp2 + b1*yp1 + b2*yp2;
            tp2 = tp1;
            tp1 = imgOut[i][j];
            yp2 = yp1;
            yp1 = y2[i][j];
        }
    }

    for (i=0; i<_PB_W; i++)
        for (j=0; j<_PB_H; j++)
            imgOut[i][j] = c2*(y1[i][j] + y2[i][j]);

#pragma endscop
}
```

Para a versão em execução paralela ordenada, os loops internos foram tornados sequenciais, ficando assim no formato:

```
for (...) { //loop externo, inalterado
    //inicialização de variáveis
    (...)
    #pragma omp parallel for ordered
    for (...) { //loop interno
        #pragma omp ordered
        //corpo do loop
        (...)
    }
}
```

Notar que, dada a existência real de dependências, não existe um modo melhor de paralelizar o código ou definir sua seção ordenada.

5.6 Gramschmidt

Esse benchmark realiza uma decomposição de matrizes QR, tendo como entrada uma matriz A e como saída um par de matrizes Q e R tais que $A = QR$. O caso é análogo ao do atax tendo sido tirado do mesmo conjunto de testes que ele, [5], e portanto serão descritas só as posições dos pragmas (em comentários) para a realização dos mesmos quatro testes (sequencial, ordered, ordered com seção ordenada maior e completamente paralelo).

```
//loop a ser testado
for (k = 0; k < _PB_N; k++)
  {
    nrm = SCALAR_VAL(0.0);
    for (i = 0; i < _PB_M; i++)
      nrm += A[i][k] * A[i][k];
    R[k][k] = SQRT_FUN(nrm);
    for (i = 0; i < _PB_M; i++)
      Q[i][k] = A[i][k] / R[k][k];
    for (j = k + 1; j < _PB_N; j++)
      {
        R[k][j] = SCALAR_VAL(0.0);
 //ordered (seção maior)
        for (i = 0; i < _PB_M; i++)
          R[k][j] += Q[i][k] * A[i][j];
  //ordered (seção menor)
        for (i = 0; i < _PB_M; i++)
          A[i][j] = A[i][j] - Q[i][k] * R[k][j];
      }
  }
```

5.7 Seidel-2d

Calcula um stencil, transformação sequencial de um vetor com base em um padrão definido, baseado no método de Gauss-Seidel. O caso é análogo ao do Deriche e eles foram extraídos do mesmo conjunto de testes, [5], e portanto só o código será apresentado, com os comentários indicando o loop que foi executado sequencialmente para testar o custo de sincronização.

```
#pragma omp parallel for ordered
for (t = 0; t <= _PB_TSTEPS - 1; t++)
#pragma omp ordered
for (i = 1; i<= _PB_N - 2; i++)
  for (j = 1; j <= _PB_N - 2; j++)
    A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]
               + A[i][j-1] + A[i][j] + A[i][j+1]
               + A[i+1][j-1] +
 A[i+1][j] + A[i+1][j+1])/SCALAR_VAL(9.0);
```

5.8 Rotate

Esse teste, o primeiro do pacote Starbench [4], possui uma real dependência do-across associada ao output presente no loop principal. Ele rotaciona uma imagem em ppm por um ângulo fornecido via linha de comando. A imagem de teste utilizada foi tirada do site Download Sample Files [2], sendo utilizada a imagem de resolução $1920x1080$ e, dado o ângulo influenciar no tempo de processamento, uma média de cinco ângulos foi usada, 45°, 60°, 90°, 120° e 180°. O código do loop utilizado é dado abaixo:

```
#pragma omp parallel for ordered
        for(int j = 0; j < target_w; j++) {
            //inicialização de variáveis
            (...)

            //coleta de samples
            (...)
                //ordered (seção mais longa)
                        #pragma omp ordered
                /* Get colors for samples */
                for(int k = 0; k < 4; k++) {
                    colors[k] = input.getPixelAt(samples[k][0], samples[k][1]);
                }



                Pixel final;
                //ordered (seção mais curta)
```

```
            #pragma omp ordered
            filter(colors, &final, &origin_pix);

            /* Write output */
            output.setPixelAt(j, i, &final);
        } else {
            /* Pixel is not in source image, write black color */

            Pixel final = {0,0,0};
            //ordered (fixo nos dois caminhos)
            #pragma omp ordered
            output.setPixelAt(j, i, &final);
        }
    }
```

A invocação foi feita com
./rotimg.ppmimg − out.ppm$angle
Para cada um dos ângulos dados acima.

5.9 Rot-cc

Esse benchmark, o segundo do conjunto de testes do Starbench [4], combina o benchmark acima com um benchmark de conversão de cores, de RGB para YUV. O loop de rotação será alterado da mesma forma, por estar presente também nesse benchmark. Os dois loops da conversão de cores serão tratados da seguinte forma:

```
#pragma omp parallel for ordered
for(int j = 0; j < width; j++) {
            R=input->getPixelAt(j,i).r;
            G=input->getPixelAt(j,i).g;
            B=input->getPixelAt(j,i).b;
            //ordered (seção longa)
            #pragma omp ordered
            p.r = round(0.256788*R+0.504129*G+0.097906*B) + 16;
            p.g = round(-0.148223*R-0.290993*G+0.439216*B) + 128;
            p.b = round(0.439216*R-0.367788*G-0.071427*B) + 128;
            //ordered (seção curta)
            #pragma omp ordered
            output->setPixelAt(j,i,&p);
        }
```

E

```
#pragma omp parallel for ordered
for(int i = 0; i < height; i++) {
```

```
        //ordered (seção longa)
        #pragma omp ordered
        for(int j = 0; j < width; j++) {
            writebuf[3*j] = output->getPixelAt(j,i).r;
            writebuf[3*j+1] = output->getPixelAt(j,i).g;
            writebuf[3*j+2] = output->getPixelAt(j,i).b;
        }
        //ordered (seção curta)
        #pragma omp ordered
        fwrite(writebuf, 1, 3*width, fp);
    }
```

A invocação do programa foi feita da mesma forma que a anterior.

5.10 rgbyuv

Esse benchmark, o último do conjunto de testes Starbench [4], realiza apenas a conversão de cores, e demanda, além da foto de entrada, um número definido para a contagem de iterações, assim, ele é invocado com

$rgbyuv - iimg - c20$

Para o número escolhido de iterações (20). O loop utilizado foi

```
#pragma omp parallel for ordered
  for(int j = 0; j < args->pixels; j++) {
    R = *in++;
    G = *in++;
    B = *in++;

    //ordered (seção longa)
    #pragma omp ordered
    Y = round(0.256788*R+0.504129*G+0.097906*B) + 16;
    U = round(-0.148223*R-0.290993*G+0.439216*B) + 128;
    V = round(0.439216*R-0.367788*G-0.071427*B) + 128;

    //ordered (seção curta)
    *pY++ = Y;
    *pU++ = U;
    *pV++ = V;
  }
```

6 Resultados

6.1 Atax

O primeiro teste com o atax, sua execução sequencial, produziu o resultado $0,380369$ segundos, na média das execuções. A execução com ordered, porém, produziu $0,300691$ segundos, novamente na média das iterações, indicando um ganho de performance de $20,95\%$ em cima da execução sequencial. Aproximar o ordered do final do loop não produziu grandes variações, produzindo como tempo médio $0,300468$ segundos, um ganho de 21% de performance em cima do código sequencial e um resultado extremamente próximo do ganho produzido pela execução ordered com o código inteiro como zona de exclusão. Por fim, a execução paralela sem ordem teve um tempo médio de $0,095225$ segundos entre as iterações, ou seja, um ganho de $74,97\%$ em cima da execução sequencial.

Os resultados do teste realmente paralelizável indicam que algum tipo de processamento está acontecendo durante o período de espera pela iteração anterior, dado o ganho de performance, algo que não acontecia com o exemplo trivial analisado pela estrutura do ordered. Esse processamento não é todo o processamento que poderia ser feito, dado que, diante da inexistência de uma dependência real o ganho potencial de performance é quase perfeitamente proporcional ao número de núcleos, mas é considerável. Esse efeito é visto também pela falta de impacto da mudança de posição do ordered.

6.2 Deriche

O tempo do teste sequencial foi $4,632895$ segundos em média, enquanto que para o teste utilizando ordered em cada um dos loops com a zona de exclusão abrangendo todo o loop, uma média de $11,429663$ segundos foi atingida, significando que o programa foi $146,71\%$ mais lento do que a versão sequencial.

É possível aqui perceber que nenhuma otimização foi realizada quanto à inutilidade em paralelizar o loop completamente sequencial, e o efeito geral da alteração foi apenas o custo da criação das threads e da sincronização, que foi multiplicado pelo posicionamento da paralelização no loop interno, produzindo uma quantidade de ineficiência pela posição incorreta do ordered.

6.3 Gramschmidt

Os resultados produzidos foram: $65,0383s$ para a execução sequencial, $61,4651s$ para a execução ordenada com seção menor, $68,6922s$ para a execução ordenada com seção maior e $28,4585s$ para a execução completamente paralela. Isso indica que, nesse teste, o loop não representava tanto da execução do programa quanto no teste atax, e que o impacto do aumento da seção ordenada aqui teve um impacto consideravelmente maior do que naquele teste, chegando a reverter os ganhos pela execução ordenada.

6.4 Seidel-2d

Os resultados foram uma média de $170,4276s$ para a versão sequencial e $170,4288s$ para a versão com ordered, indicando que, nesse caso, os custos de sincronização foram completamente desprezíveis. Esse caso foi bastante diferente do teste no benchmark Deriche, em que o impacto foi considerável. O número de repetições do loop, que naquele caso foi maior por se tratar de um loop mais profundamente aninhado, pode explicar em parte a diferença observada.

6.5 Rotate

Os resultados foram: média de $9,3088s$, para a versão sequencial, $8,6784s$ para a versão ordenada com seção mais curta, e média de $8,6826s$, para a versão ordenada com seção mais longa. O impacto aqui observado então foi virtualmente nulo, sendo que a diferença de menos de 1% pode ser apenas uma variação aleatória.

6.6 Rot-cc

Os resultados foram: média de $14,216s$, para a versão sequencial, $15,3862s$ para a versão ordenada com seção mais curta, e média de $15,9924s$, para a versão ordenada com seção mais longa. É possível notar então aqui um impacto real, mas pequeno, da mudança na seção ordenada, um resultado interessante em comparação com o teste similar rotate, que apresentou ganhos consideravelmente menores.

6.7 rgbyuv

E os resultados foram: $32,8036s$ para a versão sequencial, $44,3976s$ para a versão com seção sequencial mais curta e $41,1912s$ para a versão com seção sequencial mais longa. Uma possível explicação para isso é que a seção ordenada se tornando unificada, alguma otimização foi possibilitada que o compilador não conseguia provar com uma chamada de função separando as suas partes, resultando num código que é executado mais rápido quando sua seção ordenada se torna maior.

6.8 Compilação dos resultados

Benchmark	Sequencial	Ordered (curto)	Ordered (longo)	Paralelo
Atax	0,380369s	0,300468s	0,300691s	0,095225s
Deriche	4,632895s	11,429663s	–	–
Gramschmidt	65,0383s	61,4651s	68,6922s	28,4585s
Seidel-2d	170,4276s	170,4288s	–	–
Rotate	9,3088s	8,6784s	8,6826s	–
Rot-cc	14,216s	15,3862s	15,9924s	–
Rgbyuv	32,8036s	44,3976s	41,1912s	–

Benchmark	Diferença (sequencial/ordered)	Diferença (ordered longo/curto)
Atax	-21%	-0,07%
Deriche	+146,7%	–
Gramschmidt	-5,5%	-10,53%
Seidel-2d	$\approx 0\%$	–
Rotate	9,3088s	-0,05%
Rot-cc	14,216s	-3,8%
Rgbyuv	32,8036s	+7,8%

7 Conclusão

Os resultados mostram uma média de $-1,33\%$ de diferença de desempenho entre a versão com um setor aumentado depois do ordered e a versão com setor mais curto, com uma mínima de $-0,05\%$ e uma máxima de $-10,53\%$, indicando alta variação nos resultados. Em um dos casos, alguma outra otimização possibilitada pela continuidade das seções mostrou uma perda de performance em reduzir a seção ordered da execução, e a magnitude dessa perda foi $7,8\%$. Considerando o melhor caso, em que o ordered era completamente indiferente para os resultados finais, observamos uma possibilidade de ganho de até $3,8\%$, mas em outro caso não houve impacto considerável e ainda em um terceiro o ordered mais curto impossibilitou uma otimização e teve impacto negativo.

A mais provável razão para os ganhos não serem consideráveis com essa técnica tem a ver com o tamanho dos loops típicos em casos de uso reais. O ganho por iteração, com loops contendo poucas instruções, não é grande o suficiente para ter um efeito considerável no tempo total da execução, resultando em ganhos baixos que podem mesmo ser negados por outros efeitos, como aconteceu no último teste.

Por fim, a magnitude desses ganhos indica baixo potencial para otimizações. Um trabalho posterior poderia considerar a dificuldade de implementação de otimizações nessa área, principalmente seu custo computacional, para considerar sua validade diante dos ganhos potenciais apresentados. A implementação traria também vantagens, ao permitir análise da performance numa gama maior de situações do que aquela possibilitada pela análise manual.

Além disso, outra possibilidade posterior de trabalho seria uma análise sobre a forma do código atualmente produzido, para melhor entender os comportamentos descritos nesse trabalho e descrever eles de forma mais aprofundada que a sua caracterização em tempo, buscando principalmente entender possíveis otimizações realizadas.

Referências

[1] gem5. 2023. Disponível em: https://www.gem5.org/. Acesso em: 9 dec. 2023

[2] Download Sample Files. 2023. Disponível em: https://www.dwsamplefiles.com/download-ppm-sample-files/. Acesso em: 9 dec. 2023

[3] YUEN, PETER. gem5 RISC-V Full System Linux Guide. 2021. Disponível em: https://github.com/ppeetteerrs/gem5-RISC-V-FS-Linux. Acesso em: 9 dec. 2023

[4] ANDERSCH, Michael; CHI, Chi Ching; JUURLINK, Ben. Starbench parallel benchmark suite. 2013. Disponível em: https://www.tu.berlin/aes/forschung/projekte/completed-projects/starbench-parallel-benchmark-suite. Acesso em: 9 dec. 2023

[5] POUCHET, Louis-Noel. 2012. Disponível em: https://web.cs.ucla.edu/ pouchet/-software/polybench/. Acesso em: 9 dec. 2023

[6] SHEN, John Paul; LIPASTI, Mikko. Modern Processor Design: Fundamentals of Superscalar Processors. 2013. Waveland Press, Inc.

[7] PANKRATIUS, Victor; Adl-Tabatabai ALD-TABATABAI, Ali-Reza; TICHY, Walter. Fundamentals of Multicore Software Development. 2012. CRC Press

[8] Risc-V GNU Compiler Toolchain. 2023. Disponível em: https://github.com/riscv-collab/riscv-gnu-toolchain. Acesso em: 9 dec. 2023

[9] OpenMP. 2021. Disponível em: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf. Acesso em: 9 dec. 2023