UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ARTHUR LONGONI OLIVEIRA

Impacto da granularidade de tarefas em um programa OpenMP recursivo - Mergesort

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Dr. Nicolas Maillard
Orientador

Porto Alegre, Fevereiro de 2024

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões Mendes
Vice-Reitora: Profª. Patrícia Pranke
Pró-Reitora de Graduação: Profª. Cíntia Inês Boll
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Ciência da Computação: Prof. Marcelo Walter
Bibliotecário-Chefe do Instituto de Informática: Alexsander Borges Ribeiro

AGRADECIMENTOS

Gostaria de agradecer a todos os amigos ex-alunos da UFRGS que saindo formados ou não seguiram torcendo por mim e me convenceram a seguir o curso apesar das dificuldades encontradas ao longo dos anos, aos amigos que seguem na graduação, principalmente ao calouro vindo de transferência interna que acabou por me indicar a principal referência bibliográfica deste trabalho e sem a qual o desenvolvimento não teria sido possível. Agradeço também aos familiares que me apoiaram, aos amigos próximos, aqueles que sempre pude recorrer em momentos de aflição e ao meu orientador que conseguiu tornar a experiência de fazer este trabalho uma enriquecedora e divertida investigação.

RESUMO

Este trabalho mostra o processo de paralelização de um clássico algoritmo de ordenação, explicitando as razões para a escolha do mesmo para a realização das análises e os principais fatores que impactam em seu desempenho ao executar de forma paralela juntamente com os maiores obstáculos encontrados para a obtenção de uma boa otimização com redução significativa de tempo de execução. O conteúdo aqui visto também pode ser usado em futuras reflexões para direcionar abordagens de paralelização e otimização com o uso de OpenMP tasks a programas geralmente vistos com uso apenas sequencial ou programas cujas versões paralelas já são conhecidas porém ainda possuem espaço para novas otimizações.

Palavras-chave: OpenMP, computação paralela, sistemas operacionais, mergesort, arquitetura multi-core, compiladores, multi-threading, multi-tasking.

Task granularity impact in a recursive OpenMP program - Mergesort

ABSTRACT

This work shows the parallelization process of a classic sorting algorithm, expliciting the reasons for its choice to the realization of the analyses and the key factors which impact on its performance when executing in parallel along with the biggest obstacles found to the obtention of a good optimization with significative runtime reduction. The content seen here can also be used in future reflections to direct parallelization and optimization approaches with the use of OpenMP tasks to programs usually seen only in sequential versions or programs which parallel versions are already known but still got room for further optimizations.

Keywords: OpenMP, parallel computing, operating systems, mergesort, multi-core architecture, compilers, multi-threading, multi-tasking.

LISTA DE FIGURAS

Figura 3.1 – Resultados iniciais: tempo de execução com 8 tasks por thread..........................16
Figura 3.2 – Comparação entre tempos de execução ERAD vs mergesort sendo testado........16
Figura 4.1 – Pseudo-código de merge paralelo 17
Figura 4.2 – Ilustração de funcionamento da busca binária no merge paralelo........................17
Figura 6.1 – Gráfico de desempenho do mergesort pós paralelização do merge step..............23
Figura 6.2 – Gráfico comparativo com o GCC..26
Figura 6.3 – Gráfico comparativo com o NVC..27
Figura 6.4 – Gráfico comparativo com o ICX...28

LISTA DE ABREVIATURAS E SIGLAS

CPU	Central Processment Unit
ERAD	Escola Regional de Alto Desempenho
GPU	Graphics Processment Unit
OMP	Open Multi-Processing
UFRGS	Universidade Federal do Rio Grande do Sul

SUMÁRIO

1 INTRODUÇÃO....9
1.1 OpenMP....9
1.2 Multi-Core....10
1.3 O Problema da Otimização de Desempenho com OpenMP Tasks....10
1.4 Mergesort....11
1.5 Como Abordar Programas Paralelos Recursivos?....11
2 CONTEXTO CIENTÍFICO....12
2.1 OpenMP Pragmas....12
2.2 OpenMP Tasks....12
2.3 Pragmas e Cláusulas OpenMP Utilizadas....13
2.4 Exemplo Didático: Fibonacci com OpenMP Tasks....14
2.5 O Que Muda na Compilação?....14
2.6 Hardware Utilizado....15
2.7 Bloco Base de Chamada Dentro da main()....15
3 PARALELIZANDO O MERGESORT COM OPENMP TASKS....17
3.1 Ideia Inicial....17
3.2 Análise dos Primeiros Resultados....18
4 PARALELIZANDO O MERGE....20
4.1 Uso da Busca Binária na Paralelização do Merge....21
4.2 Mergesort com Merge Paralelo....22
5 FATORES DE IMPACTO OBSERVADOS....24
5.1 Impacto do Compilador....24
5.2 Impacto da Parada da Recursão....24
5.3 Impacto da Geração de Tasks....25
5.4 Granularidade: Número de Tarefas por Thread....25
6 RESULTADOS PÓS-OTIMIZAÇÃO....27
6.1 Escalabilidade....27
7 CONCLUSÃO....31
REFERÊNCIAS....32
APÊNDICE A: USO DO CHATGPT....33
APÊNDICE B: DEMAIS CÓDIGOS....34

1 INTRODUÇÃO

OpenMP já existe desde o final dos anos 90 e se tornou popular entre programadores interessados em computação paralela, oferecendo diversos recursos para paralelizar programas porém nem todos esses recursos possuem a mesma fama e portanto acabam sendo sub-utilizados além de terem seu uso menos compreendido até mesmo por usuários mais avançados. O recurso das OpenMP tasks por exemplo oferece um nível além das threads, permitindo uma quantidade mais dinâmica de fluxos de execução, especialmente aplicável a programas recursivos.

No entanto, o fine-tuning de programas que utilizam métodos menos conhecidos de paralelismo acaba sendo dificultado pois agora temos novos recursos à disposição que precisam ser controlados corretamente para atingir um bom funcionamento. Este trabalho visa mostrar esta experiência e como o caso foi superado, tomando como base o mergesort, algoritmo recursivo que não apenas foi paralelizado como acabou sendo otimizado além do esperado. O objetivo principal é fazer o leitor entender que antes de otimizar parâmetros sensíveis em uma aplicação (como o uso da memória cache com padding) é possível atingir bons ganhos de desempenho ao se fazer mudanças em parâmetros mais simples (como o compilador sendo usado e o número de threads e tasks).

Nos próximo capítulo veremos um aprofundamento em OpenMP, necessário para a compreensão do resto do trabalho, nos capítulos seguintes o detalhamento do processo de paralelização do mergesort, seguido das dificuldades encontradas, soluções empregadas e os principais impactos causados por essas mudanças. Por fim, um comparativo entre como começamos e o que alcançamos em termos de desempenho.

1.1 OpenMP

OpenMP é uma API que surgiu com o intuito de facilitar a programação paralela aos que visavam extrair mais desempenho de seus códigos e também para que os mais experientes pudessem aumentar a eficiência de seus programas já paralelos, oferecendo diversas formas de explorar os recursos de hardware disponíveis e de estruturar os fluxos de execução, tanto em C/C++ quanto em Fortran, sendo recomendada para adeptos do desenvolvimento voltado a HPC (High Performance Computing) com foco em CPU e também em arquiteturas heterogêneas. Um dos recursos mais usados neste trabalho foi a OpenMP task, indicada para

paralelizar algoritmos recursivos acompanhada de diretivas (também chamadas de “pragmas” pelos desenvolvedores que as usam) que auxiliam na forma como as OMP tasks são geradas, sincronizadas e como operam, sendo mais detalhadas nos próximos capítulos.

1.2 Multi-Core

Desde 2003 vimos os esforços de fabricantes de CPUs em fornecer maior paralelismo aos consumidores em razão do aumento de frequência de clock ser perigoso para a época, começando com o surgimento do Pentium 4 HT que mesmo tendo apenas um núcleo já nos permitia ter duas threads, o que com o tempo aumentou até atualmente termos CPUs de propósito geral com até 24 núcleos como no caso das CPUs mais recentes Core i9 da Intel, com o possível paralelismo de 48 threads.

Antigamente os usuários tinham CPUs operando apenas sobre uma tarefa por vez, porém pela rapidez das operações tinha-se apenas a impressão de que eram executadas simultaneamente mas com o surgimento de hyper-threading e CPUs multi-core passamos a ter execuções realmente simultâneas.

Infelizmente, mesmo com todo o potencial de paralelismo que temos em hardware, programadores que utilizam de fato paralelismo no momento de escrever código ainda parecem ser minoria.

1.3 O Problema da Otimização de Desempenho com OpenMP Tasks

Mesmo com o OpenMP facilitando a programação paralela ainda temos algumas barreiras a enfrentar mas só as enxergamos após testar os programas paralelos e compará-los com suas versões sequenciais para ter a certeza de que realmente estamos aumentando o desempenho, correndo o risco de estarmos na verdade diminuindo-o. No caso das OpenMP tasks, a primeira característica mais visível é o problema do controle de geração de tasks: Quantas tasks gerar, quando parar de gerar mais tasks, como e quando gerá-las. Tasks demais podem acabar reduzindo o desempenho pois acabam operando sobre volumes muito pequenos de dados enquanto o uso de poucas tasks pode reduzir nosso paralelismo e consequentemente o desempenho.

1.4 Mergesort

O algoritmo escolhido para este trabalho foi o mergesort em razão de diversas características do algoritmo auxiliarem na didática das futuras elucidações e na prática para todo o desenvolvimento e os testes feitos ao longo dele. A escolha se deu por ser um algoritmo simples, recursivo e clássico, amplamente conhecido por alunos de graduação e demais acadêmicos da área de computação, além de também ter boa complexidade e distribuição homogênea de carga entre suas chamadas recursivas, algo que algoritmos como o quicksort não possuem, apesar do bom desempenho.

1.5 Como Abordar Programas Paralelos Recursivos?

A otimização de um programa paralelo recebe um nível de dificuldade a mais no momento em que decidimos adicionar recursividade. Para ter certeza de que obtemos um programa paralelo recursivo de desempenho adequado dependemos intrinsecamente de muitas execuções de testes com medições de tempo e comparações entre execuções sequenciais e paralelas, não só para ter certeza de que as versões paralelas são mais rápidas mas também para medir quão mais rápidas essas versões são e que abordagens melhoram ou pioram o desempenho, sendo inclusive a escolha do compilador utilizado um fator de alto impacto. Além destes fatores é preciso tomar cuidado com o controle da recursão, que pode acabar tornando a call stack um problema a mais e acabou por ser um dos desafios encontrados ao longo do desenvolvimento deste trabalho.

2 CONTEXTO CIENTÍFICO

Este capítulo visa resumir os recursos de software utilizados neste trabalho, focando em OpenMP, seus itens mais característicos para uso em programação paralela com tasks e como estes recursos interagem entre si.

2.1 OpenMP Pragmas

Pragmas aparecem diversas vezes ao longo nós códigos elaborados ao longo da investigação e são usados para passar informações ao compilador para que gere código otimizado pra OpenMP de acordo com as restrições e informações que passarmos. A funcionalidade dos pragmas é ampla, servindo para especificar se variáveis devem ser compartilhadas ou privadas entre os fluxos paralelos de execução, quantos fluxos devem ser gerados e sob que condições devemos seguir gerando-os ou parar de gerá-los, além dos usos em sincronização ao final de execuções de trechos do programa.

Sabendo que os pragmas são passados ao compilador para que este mude sua forma de gerar código para atender o programador e visando maior precisão na análise do comportamento do merge sort paralelizado com tasks, pareceu promissor executar os testes do algoritmo com seu programa gerado a partir de 3 compiladores diferentes: GCC (amplamente usado e conhecido pelos estudantes da graduação), NVC (NVIDIA) e ICX (Intel, recente substituto do ICC). O compilador Clang foi cogitado para uso neste trabalho porém acabou se mostrando um tanto complexo no momento de instalá-lo e configurá-lo para uso.

2.2 OpenMP Tasks

Tasks em OpenMP são elementos mais usados em casos onde é requerido mais dinamismo em comparação a outros construtos como as OpenMP sections e loops. As OpenMP tasks são recomendadas para programas OpenMP que utilizam chamadas recursivas e oferecem a vantagem de não precisarem de scheduling especificado em código, deixando este trabalho a cargo do sistema de runtime.

2.3 Pragmas e Cláusulas OpenMP Utilizadas

#pragma omp parallel for: Pragma que precede um laço for a ser paralelizado entre as threads disponíveis.

#pragma omp parallel: Pragma básico que define o bloco de código a seguir como trecho paralelizado.

#pragma omp task: Pragma básico para instanciar uma task que executará o bloco de código a seguir.

#pragma omp single: Pragma que garante que a execução do bloco a seguir será feita por apenas uam thread.

#pragma omp taskwait: Pragma que funciona como mecanismo de barreira para sincronização da execução de tasks.

if: Cláusula para controlar spawn condicional de tasks, testada neste trabalho porém substituída pelo uso de ifs originais da linguagem C para garantir consistência.

final: Cláusula responsável por definir um ponto de parada no spawn de tasks, indicada para casos onde se usa recursividade porém o uso da cláusula ainda não confiável[4]. Também substituída neste trabalho por ifs da linguagem C.

untied: Cláusula utilizada na instanciação de uma task que a permite ser retomada por qualquer thread disponível ao invés de apenas a thread que a suspendeu, que é o modo padrão de operação.

num_threads(x): Cláusula que define que x threads serão usadas no bloco de código a seguir.

Vale também citar a função omp_get_wtime(), utilizada nas medições de tempo pois funcionava de forma precisa em comparação com outros métodos que acabavam por retornar o somatório de tempos gastos em CPU, resultando em uma medição de tempo maior do que o próprio tempo de execução em razão dos trechos paralelos.

2.4 Exemplo Didático: Fibonacci com OpenMP Tasks

O exemplo a seguir mostra um uso da cláusula "shared", que explicita itens a serem compartilhados entre tasks criadas pela mesma região geradora, porém não utilizado ao longo do trabalho e portanto não citado anteriormente.

```
#include <stdio.h>

#include <stdlib.h>
#include <omp.h>

int fibonacci(int n) {
  int i, j;
  if (n<2) return n;
  else{
    #pragma omp task shared(i)
    i=fibonacci(n-1);
    #pragma omp task shared(j)
    j=fibonacci(n-2);
    #pragma omp taskwait
    return i+j;
  }
}

int main(int argc, char **argv){
  int n, r;
  n = atoi(argv[1]);
  #pragma omp parallel
    #pragma omp single
     r = fibonacci(n);
  printf("Resultado: %d\n", r);
  return 0;
}
```

2.5 O Que Muda na Compilação?

Para compilar utilizando OpenMP precisamos além do include, alterar levemente a linha de comando utilizada, adicionando também no caso deste trabalho a biblioteca de matemática caso quiséssemos utilizar logaritmos para facilitar o acesso de índices em alguns arrays, levando em consideração que a maioria dos dados gira em torno de potências de 2. Utilizando o arquivo .c do código anterior como exemplo a seguir temos:

GCC: gcc fibonacci.c -o fib_test -fopenmp

NVC: nvc fibonacci.c -o fib_test -mp

ICX: icx fibonacci.c -o fib_test -qopenmp

É importante ressaltar que para uma comparação justa não foram utilizadas flags de otimização nem de debugging nos 3 casos.

Cada compilador emprega versões diferentes de OpenMP[1] e algumas funcionalidades podem ser afetadas, a seguir temos as versões correspondentes a cada um:

GCC: versão 11.4.0 – Suporte total a OpenMP 4.5, parcial a OpenMP 5.0.

NVC: versão 23.5 – Suporte total a OpenMP 3.1, parcial a OpenMP 5.1.

ICX: versão 2023.1.0 – Suporte total a OpenMP 4.5, parcial a OpenMP 5.0.

2.6 Hardware Utilizado

A Máquina usada em todos os testes possuía um processador de 8 núcleos AMD Ryzen 7 3700X, memória RAM DDR4 com latência de CAS 15 e frequência de operação de 2400MHz. O sistema operacional era o Ubuntu, versão 22.04.3.

Cogitou-se utilizar máquinas com mais núcleos para ter testes com maiores níveis de paralelismo porém isso dificultaria a reprodução dos experimentos por uma grande parte dos alunos que lessem este trabalho.

2.7 Bloco Base de Chamada Dentro da main()

A chamada básica do mergesort paralelo se mantém a mesma ao longo de todas as mudanças feitas para ganhos de desempenho. Estas mudanças se concentram no código da própria função de mergesort e na função auxiliar de merge. A seguir a forma padrão como o mergesort paralelo é chamado:

```
#pragma omp parallel num_threads(threads)
{
 #pragma omp single
  mergesort(array, 0, SIZE-1, SIZE/(threads*tasks));
 #pragma omp taskwait
}
```

Onde SIZE é o tamanho do array original, tasks o número de tasks “final” por thread e threads o número de threads a serem usadas na execução.

3 PARALELIZANDO O MERGESORT COM OPENMP TASKS

Utilizar OpenMP inicialmente pareceu uma tarefa fácil mas o uso de pragmas que lidam com tasks é um pouco mais complexo do que pragmas para lidar com laços ou sections e o entendimento de seu uso é algo ainda ambíguo mesmo após a leitura da documentação, por exemplo o funcionamento da cláusula if no spawn de tasks que parece seguir gerando tasks independentemente da avaliação, resultando em perda de performance como se não tivesse sido usado.

3.1 Ideia Inicial

Tomando como base o mergesort com OpenMP tasks visto da ERAD[2] 2021 e partindo da idéia de que podemos ter mais do que uma task a ser executada por cada thread, precisávamos pensar em um ponto de partida no número de tasks por thread para então dividir o array em partes iguais entre as tasks e definir um ponto de parada para o spawn delas. Tendo em vista que cada chamada do algoritmo gera duas chamadas recursivas que potencialmente criarão duas tasks, chegamos ao cálculo do número total de tasks em uma execução: $2^{n+1} - 2$, onde n é o último nível da árvore de chamadas recursivas onde queremos ter o spawn de tasks, sendo 0 o primeiro nível.

O parâmetro "tasks por thread" tem a finalidade de indicar quantas tasks por thread temos executando de forma "final" e foi usado ao longo deste trabalho na limitação do spawn de tasks, portanto levamos em conta para este valor apenas o número de tasks no último nível de spawn da árvore. Após testes iniciais notou-se que uma boa quantidade de tasks por thread seria entre 4 e 8 (com o desempenho caindo ao sair deste intervalo), tendo um if que controla quando gerar tasks e quando seguir a execução recursiva sem spawn de tasks novas, com a avaliação da expressão que se pode traduzir como:

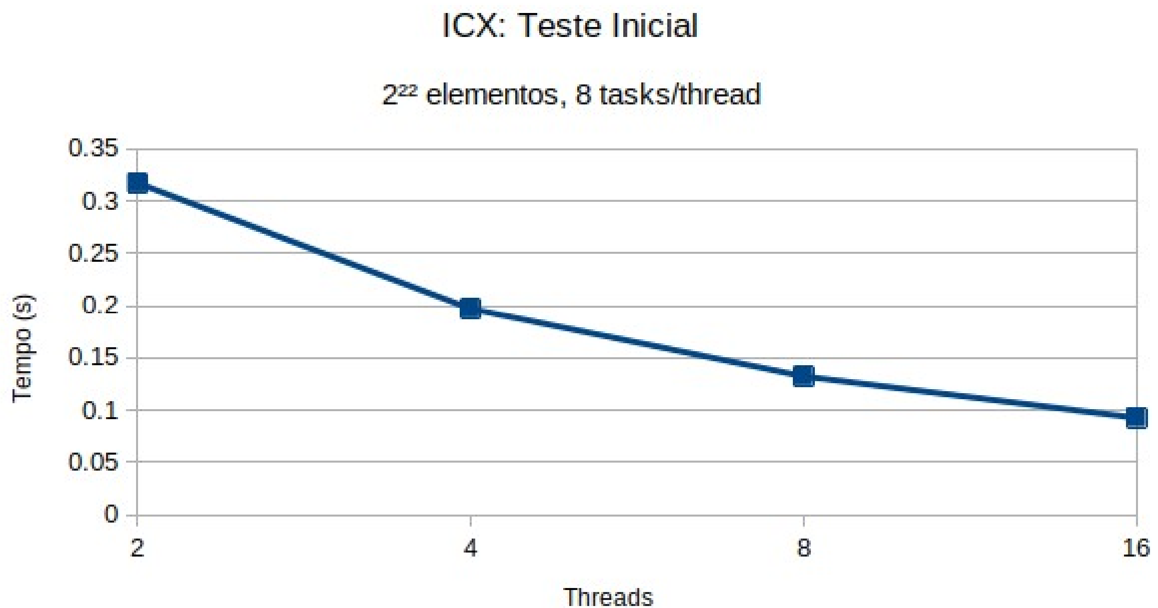
current_array_size > original_array_size / (tasks_per_thread * number_of_threads)

Onde:

- current_array_size = tamanho do array recebido pela chamada atual.
- original_array_size = tamanho do array de entrada no início da execução.

- tasks_per_thread = número de tasks para cada thread usada na execução. Importante ressaltar que isso não impede que uma thread execute mais ou menos tasks do que este número, ele apenas serve para definir o total de tasks a serem executadas no último nível com spawn de tasks ocorrendo.
- number_of_threads = número de tasks a serem usadas na execução.

Figura 3.1 – Resultados iniciais: tempo de execução com 8 tasks por thread



Fonte: Autor

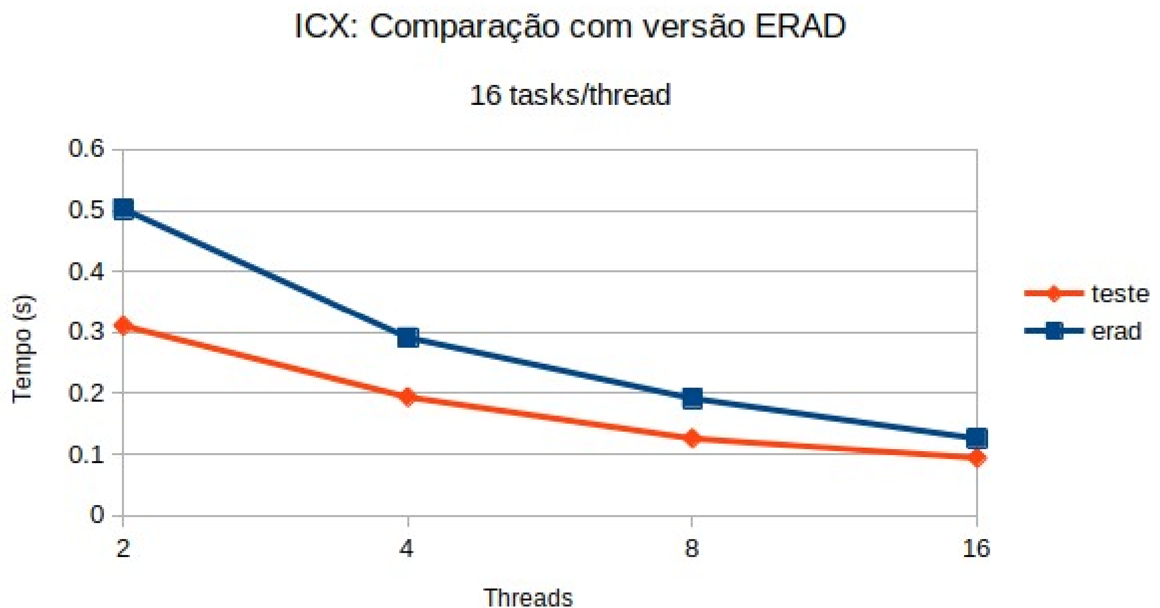
Os melhores tempos de execução acompanhados de menor variância entre execuções foram obtidos desde o início com o uso do compilador ICX, acima vemos a média de tempo de execução utilizando entre duas e dezesseis threads, com 8 tasks para cada thread em todas as execuções. Mais tarde descobriríamos que ainda é possível aumentar o número de tasks por thread para 16 sem perder desempenho.

3.2 Análise dos Primeiros Resultados

Surgiu um novo desafio ao notar que o desempenho com aumento no número de threads não crescia como esperado e após maior investigação foi percebido que a etapa de merge do algoritmo funciona de forma sequencial mesmo quando paralelizamos chamadas de merge sort, sendo assim necessário focar na paralelização do merge antes de voltar às análises de desempenho do algoritmo. Ainda antes da paralelização do merge foi possível notar após diversas experimentações a diferença no desempenho ao usar ifs/elses de C ao invés das

cláusulas if de OpenMP e utilizando uma mesma função de merge para tornar a comparação justa, foi possível notar que o desempenho do código desenvolvido até então já superava o da versão presente na ERAD 2021.

Figura 3.2 – Comparação entre tempos de execução ERAD vs mergesort sendo testado



Fonte: Autor

Ao fazer testes comparativos entre a versão inicial do mergesort paralelo deste trabalho e a versão apresentada na ERAD 2021 notou-se que utilizar um if/else da linguagem C ao invés de uma cláusula if dentro do pragma responsável por gerar tarefas acabou por melhorar o desempenho e reduzir o tempo de execução mais do que o esperado. Ainda assim, esperava-se obter um desempenho ainda melhor após a paralelização do merge.

4 PARALELIZANDO O MERGE

Paralelizar a função de merge não é trivial mesmo sabendo que existe mais de uma forma de fazê-lo. Cormen[3] mostrou didaticaente em forma de pseudo-código a função de merge paralelizada com o auxílio de busca binária, utilizando-a neste caso na função FIND-SPLIT-POINT, que busca o elemento x no array A entre os índices p2 e r2:

Figura 4.1 – Pseudo-código de merge paralelo

```
P-MERGE(A, p, q, r)
 1  let B[p : r] be a new array             // allocate scratch array
 2  P-MERGE-AUX(A, p, q, q + 1, r, B, p)    // merge from A into B
 3  parallel for i = p to r                 // copy B back to A in parallel
 4      A[i] = B[i]

P-MERGE-AUX(A, p1, r1, p2, r2, B, p3)
 1  if p1 > r1 and p2 > r2                  // are both subarrays empty?
 2      return
 3  if r1 − p1 < r2 − p2                    // second subarray bigger?
 4      exchange p1 with p2                 // swap subarray roles
 5      exchange r1 with r2
 6  q1 = ⌊(p1 + r1)/2⌋                      // midpoint of A[p1 : r1]
 7  x = A[q1]                               // median of A[p1 : r1] is pivot x
 8  q2 = FIND-SPLIT-POINT(A, p2, r2, x)     // split A[p2 : r2] around x
 9  q3 = p3 + (q1 − p1) + (q2 − p2)         // where x belongs in B ...
10  B[q3] = x                               // ... put it there
11  // Recursively merge A[p1 : q1 − 1] and A[p2 : q2 − 1] into B[p3 : q3 − 1].
12  spawn P-MERGE-AUX(A, p1, q1 − 1, p2, q2 − 1, B, p3)
13  // Recursively merge A[q1 + 1 : r1] and A[q2 : r2] into B[q3 + 1 : r3].
14  spawn P-MERGE-AUX(A, q1 + 1, r1, q2, r2, B, q3 + 1)
15  sync                                    // wait for spawns
```

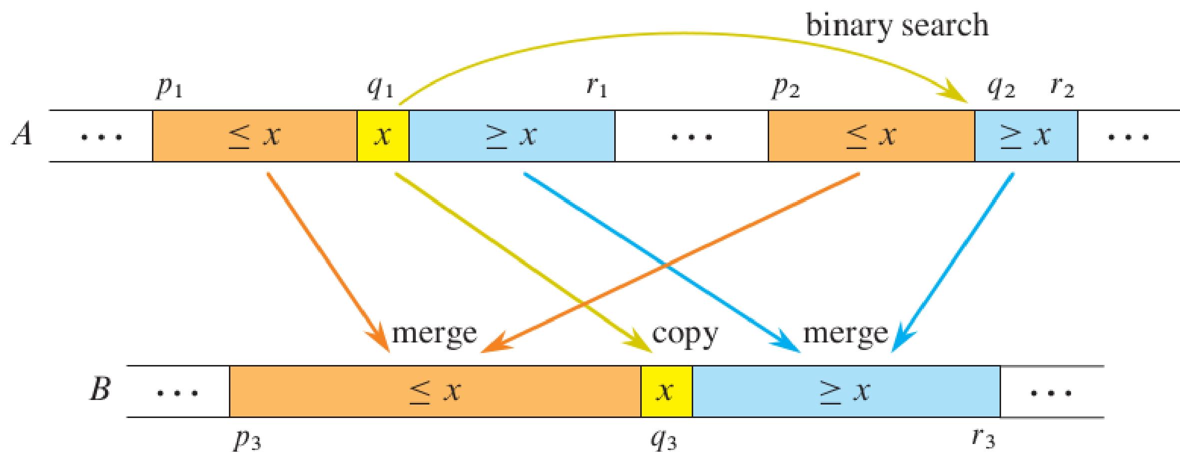
Fonte: Cormen(2022)

Convenientemente a notação usada pelo autor neste caso utilizou o termo “spawn”, como já estávamos nos referindo sobre o momento da criação de tasks e temos o ponto de taskwait em “sync”. Outro detalhe importante a ser destacado é que a função compara o tamanho dos subarrays para garantir que o pivô sempre será retirado do maior subarray. A função mais complexa a ser adaptada para código C é sem dúvida P-MERGE-AUX, por ter uma grande

quantidade de parâmetros em sua chamada e por ter o código mais longo em comparação com as demais funções, lidando constantemente com os índices dos arrays.

A imagem a seguir facilita o entendimento do uso da busca binária na paralelização do merge.

Figura 4.2 – Ilustração de funcionamento da busca binária no merge paralelo



Fonte: Cormen(2022)

4.1 Uso da Busca Binária na Paralelização do Merge

- Temos dois subarrays **A** e **B**, onde **B** armazenará o resultado das ordenações.
- No subarray **A** temos dois subarrays já ordenados que chamaremos de **D** e **E**.
- **D** vai do índice **p1** ao **r1**.
- **E** vai do índice **p2** ao **r2**.
- **q1** é o índice do ponto médio no subarray **D**.
- **x** é o elemento localizado neste índice.
- **p3** é o primeiro indice do array original na primeira chamada e passa a ser o índice de início dos demais subarrays em chamadas futuras
- A posição **q3** é definida pela posição **p3** somada ao tamanho dos dois subarrays indicados na imagem contendo apenas elementos menores ou iguais a **x**.

Dadas estas definições, usamos busca binária para buscar **x** no subarray **E** e utilizaremos o índice onde **x** seria inserido para dividir o subarray **E** em duas partes.

Após estas divisões, faremos ao mesmo tempo o merge das partes menores que **x** a esquerda no subarray **B** e das partes maiores à direita de forma paralela.

Para saber em que índice devemos colocar **x** no array **B** precisamos apenas da variável **p3** que tem como propósito indicar o índice onde este array ordenado deve ficar, assim sabemos que **x** deve ser colocado a partir do índice p3 logo após a soma dos tamanhos dos subarrays menores que ele em **A**.

4.2 Mergesort com Merge Paralelo

Após as devidas adaptações para converter o pseudo-código em código C:

Split Point:

```
int split(int arr[], int p, int r, int target){
 int low = p;
 int high = r + 1;
 while (low < high) {
  int mid = (low + high) / 2;
  if (target <= arr[mid])
   high = mid;
  eles
   low = mid + 1;
 }
 return low;
}
```

Merge Paralelo:

```
void p_merge(int arr[], int p, int q, int r, int task_size){
 int* barr = (int*)malloc(r * sizeof(int));
 p_merge_aux(arr, p, q, q+1, r, barr, p, task_size);
 #pragma omp parallel for
 for (int i=p; i<=r; i++)
  arr[i] = barr[i];
 free(barr);
}
```

Mergesort:

```
void mergesort(int arr[], int p, int r, int task_size){
 if (p >= r) return;
 int q = (p + r) / 2;
 int size = r - p + 1;
 if(size > task_size){
  #pragma omp task untied
   mergesort(arr, p, q, task_size);
  #pragma omp task untied
   mergesort(arr, q + 1, r, task_size);
  #pragma omp taskwait
  p_merge(arr, p, q, r, task_size);
 }else{
  mergesort(arr, p, q, task_size);
  mergesort(arr, q + 1, r, task_size);
  merge(arr, p, q, r);
 }
}
```

Auxiliar de merge para a parelelização:

```
void p_merge_aux(int arr[], int left_start, int left_end, int right_start,
int right_end, int barr[], int b_start, int task_size){
    int left_size = left_end - left_start;
    int right_size = right_end - right_start;
    if (left_size < right_size){
     int x = left_end;
     left_end = right_end;
     right_end = x;
     x = left_start;
     left_start = right_start;
     right_start = x;
     x = left_size;
     left_size = right_size;
     right_size = x;
    }
    if(task_size > left_size + right_size + 1){
     int j = b_start;
     for(int i=left_start;i<=left_end;i++){
      barr[j] = arr[i];
      j++;
     }
     for(int i=right_start;i<=right_end;i++){
      barr[j] = arr[i];
      j++;
     }
       merge(barr, b_start, b_start + left_size, b_start + left_size +
right_size + 1);
     return;
    }
    int midpoint = (left_end + left_start) / 2;
    int pivot = arr[midpoint];
    int s_point = split(arr, right_start, right_end, pivot);
    int t = b_start + (midpoint - left_start) + (s_point - right_start);
    barr[t] = pivot;
    #pragma omp task untied
      p_merge_aux(arr, left_start, midpoint-1, right_start, s_point-1, barr,
b_start, task_size);
    #pragma omp task untied
      p_merge_aux(arr, midpoint+1, left_end, s_point, right_end, barr, t+1,
task_size);
    #pragma omp taskwait
   }
```

A chamada de merge() presente nesta função refere-se ao merge sequencial, feita quando o array se torna pequeno o suficiente para que paremos de criar tasks.

5 FATORES DE IMPACTO OBSERVADOS

Este capítulo visa detalhar de forma mais aprofundada os principais parâmetros que ao variar impactaram mais intensamente no desempenho ao longo dos diversos testes feitos.

5.1 Impacto do Compilador

Ao testarmos o mesmo programa compilado pelo GCC, NVC e ICX notamos através do desvio padrão dos tempos de execução em amostras com 100 execuções (feitas com cada compilador e utilizando a função omp_get_wtime() para medir apenas o tempo de ordenação de um array pré-alocado com inteiros aleatórios) que o compilador GCC gerava programas de comportamento imprevisível com tempo de execução variando de forma extrema a cada momento além de gerar programas sempre mais lentos que as versões geradas pelo NVC e pelo ICX, sendo assim verificado como não confiável para ser levado em conta durante as outras investigações.

O compilador NVC mostrou não ser apropriado para lidar com OpenMP tasks no contexto deste trabalho mesmo tendo baixa variância de tempo de execução e apesar de ao menos vermos coerência deste parâmetro entre todas as execuções, visto que o tempo sempre aumentava conforme adicionássemos mais tasks até alcançarmos 16 por thread, onde o desempenho mostra uma leve melhora. Nos restou apenas supor que em razão deste compilador ser da NVIDIA sua aplicação é mais indicada para processamento massivamente paralelo ao invés de uma CPU com apenas 8 cores porém os testes com GPU e sistemas com múltiplas CPUs fugiriam o escopo deste trabalho.

O compilador ICX acabou por gerar os programas com o melhor desempenho, tendo o comportamento previsto de redução de tempo de execução conforme aumentamos o número de OpenMP tasks até o limite de 16 por thread.

5.2 Impacto da Parada da Recursão

Em um merge sort sequencial clássico possuímos dois pontos de chamadas recursivas: o início da função e o ponto de chamada da função auxiliar merge.

No caso do merge sort paralelo acabamos tendo a necessidade de ajustar um ponto da recursão para que o algoritmo siga de forma iterativa, assim evitamos chamadas recursivas com poucos elementos e que prejudicariam o desempenho do programa. Como mencionado na ERAD 2021, existem otimizações do mergesort que envolveriam o uso de outros algoritmos de ordenação chamados a partir de certo ponto para evitar o acúmulo de chamadas recursivas com sub-arrays muito pequenos porém o emprego de técnicas desse tipo já seria aplicável ao mergesort sequencial e fugiria o escopo de ambos os trabalhos. Foram feitos diversos testes adiantando e postergando o ponto de iteratização da função até alcançarmos um ponto ótimo.

Houve um caso especial onde antecipar a parada da recursão no merge paralelo acaba melhorando o desempenho ao se usar entre duas e quatro OpenMP tasks por thread.

5.3 Impacto da Geração de Tasks

Utilizamos a geração de tasks tanto nas chamadas da função principal quanto nas chamadas de merge, sendo assim é muito importante que limitemos quantas tasks iremos gerar ao longo da execução para garantir paralelismo e evitar sobrecarga do sistema no gerenciamento. O principal fator de controle para o spawn de tasks foi o parâmetro “tasks por thread”, utilizado na função principal e no merge.

5.4 Granularidade: Número de Tarefas por Thread

Para identificar o número ideal de tasks OpenMP por thread, fixamos o número de threads a serem usadas e testamos diversas execuções variando apenas a quantidade de tasks. A hipótese inicial de que o número ideal seria entre 4 e 8 tasks por thread se mostrou próxima do obtido, porém o melhor caso acabou sendo ao utilizarmos no ICX o valor de 16 tasks por thread. Chegou a ser feito um teste com 32 tasks por thread e apesar de surpreendentemente o desempenho se manter parecido ao caso com 16 ocorreu uma leve piora.

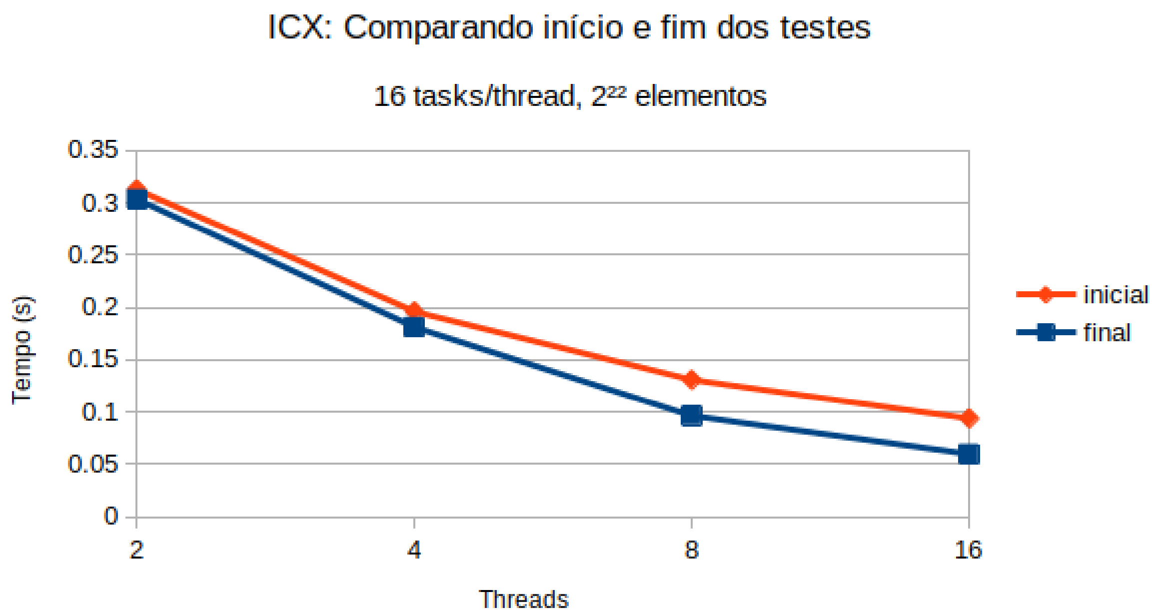
Houve também o teste onde gerávamos tasks apenas em um ponto específico das chamadas recursivas mas este caso também se mostrou pior em termos de desempenho quando comparado com outras execuções pois antecipar este ponto acabaria por gerar menos tasks do que queremos e postergá-lo não apenas geraria tasks demais para o volume de dados

de entrada como também faria o programa gastar muito tempo atuando de forma sequencial antes de começar sua parte paralela.

6 RESULTADOS PÓS-OTIMIZAÇÃO

Teste comparativo utilizando versões compiladas pelo ICX usando merge paralelo e merge apenas sequencial mostrou que o uso do novo merge melhora em pelo menos 33% o tempo de execução.

Figura 6.1 – Gráfico de desempenho do mergesort pós paralelização do merge step



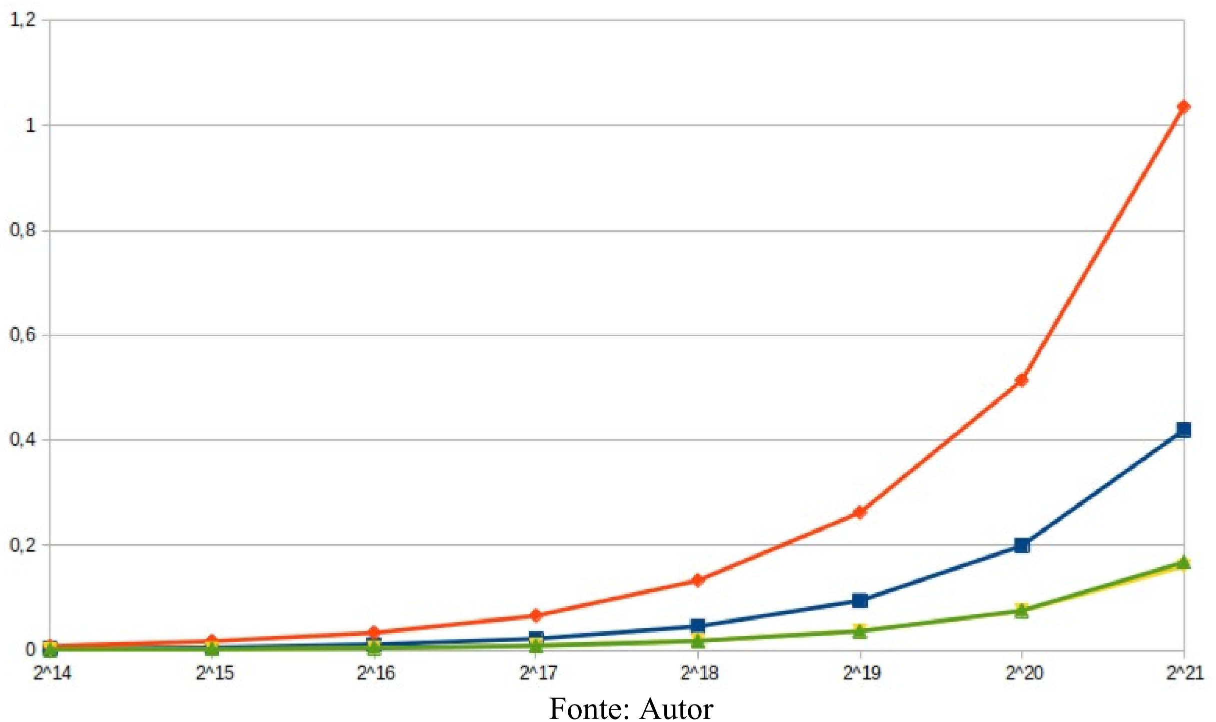
Fonte: Autor

6.1 Escalabilidade

A seguir temos um gráfico para cada compilador mostrando como as diversas versões do mergesort se comportam conforme aumentamos o tamanho do problema. É importante ressaltar que apesar de não ter obtido o melhor desempenho, o compilador NVC pareceu lidar melhor com os pragmas que aparentemente não funcionaram como esperado nos outros dois compiladores e que no momento da geração destes gráficos o merge paralelo ainda não estava tão otimizado quanto no caso exibido na figura 6.1. Os gráficos também ressaltam uma grande diferença na escalabilidade do mergesort ao utilizarmos qualquer versão paralela no lugar de uma sequencial. No eixo Y dos gráficos temos o tempo de execução em segundos e no eixo X o tamanho do array, medido em potências de 2. As curvas verde e amarela representam respectivamente os mergesorts com chamadas sequenciais e paralelas enquanto a

curva azul mostra resultados para a versão puramente sequencial e a curva laranja relacionada à versão da ERAD.

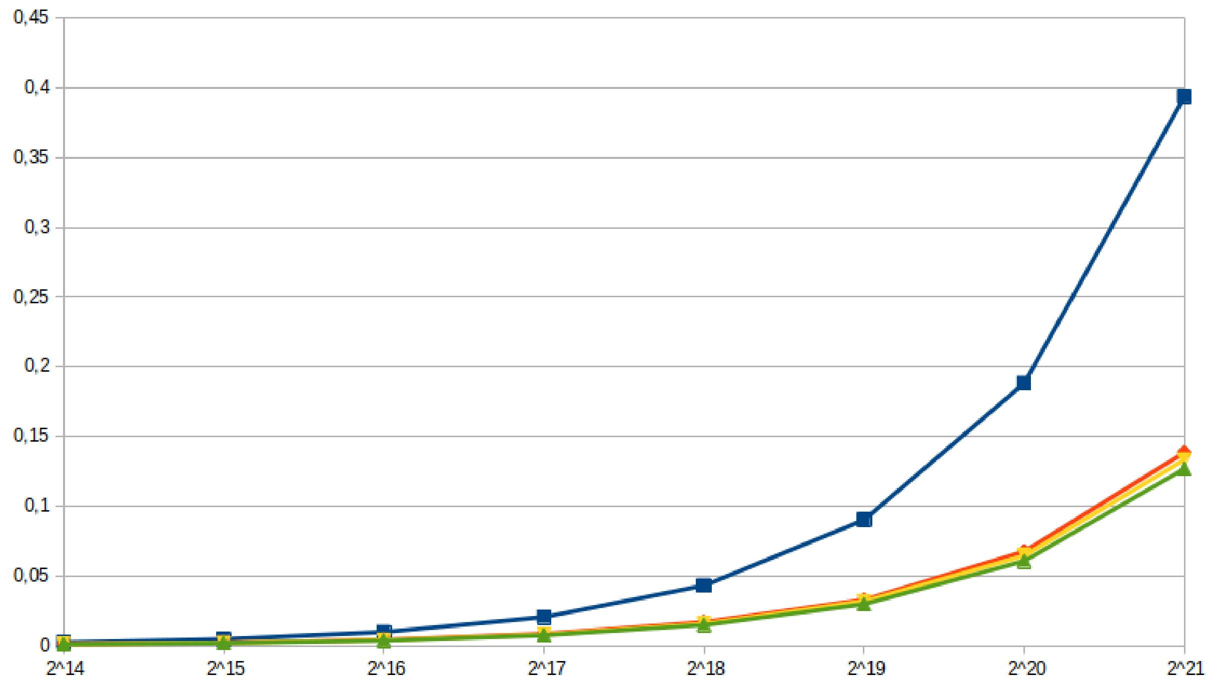
Figura 6.2 – Gráfico comparativo com o GCC



Fonte: Autor

As maiores surpresas neste caso foram ver que a versão da ERAD (em laranja) acaba tendo desempenho pior do que a versão puramente sequencial (em azul) e as versões de mergesort paralelo com merges sequencial e paralelizado ficaram com desempenho quase idêntico.

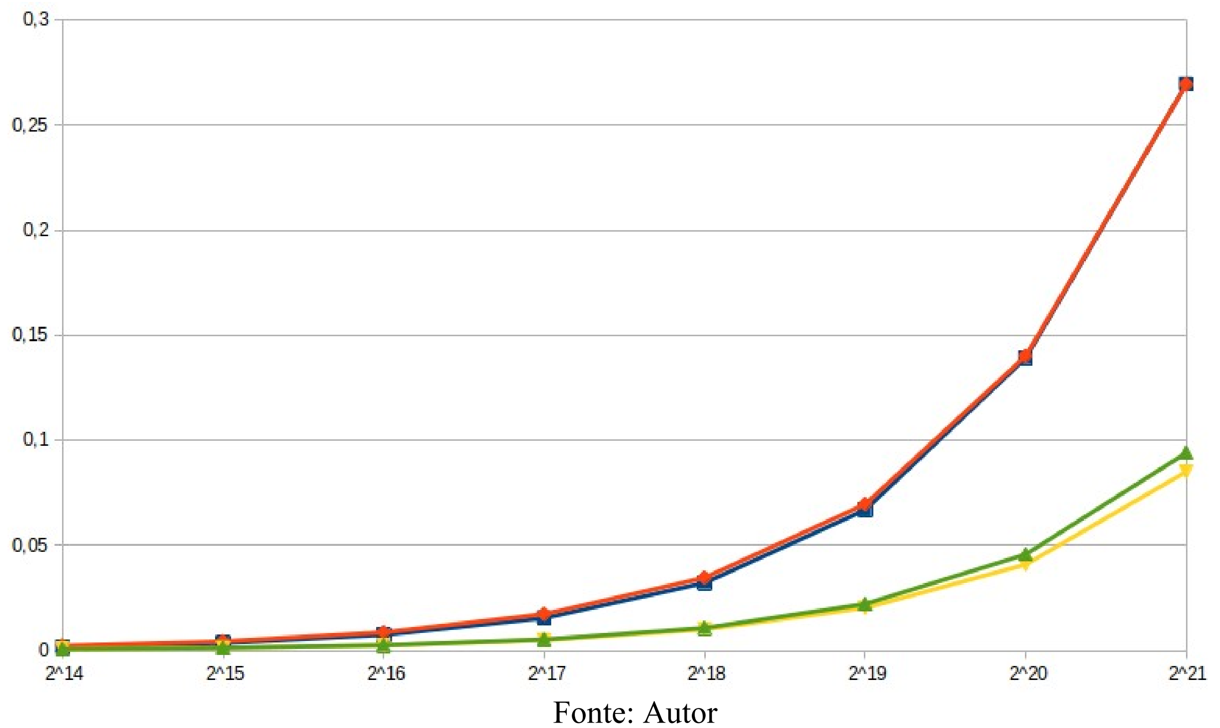
Figura 6.3 – Gráfico comparativo com o NVC



Fonte: Autor

Neste caso o tempo de execução cai drasticamente em comparação com os resultados obtidos ao usar o GCC e ocorre o esperado ao vermos o pior tempo ser o da versão puramente sequencial. Temos porém a versão de merge paralelo com desempenho pior do que a versão com merge sequencial, ficando com desempenho próximo ao da versão da erad, indicando que ao utilizar o compilador NVC temos um funcionamento correto ou pelo menos mais próximo do esperado ao utilizarmos a cláusula “if” no pragma responsável por gerar tasks.

Figura 6.4 – Gráfico comparativo com o ICX



Fonte: Autor

Neste caso obtivemos os melhores tempos de execução, chegando a no máximo aproximadamente 0.27 segundos para as versões sequencial/ERAD e menos de 0.1 segundo para as versões com merge sequencial/paralelo e vemos a versão de merge paralelo obtendo o melhor desempenho apesar de ainda não estar otimizada o suficiente para ganhar da versão de merge sequencial por uma margem maior.

7 CONCLUSÃO

Foi possível perceber que utilizar tasks em OpenMP não é uma tarefa difícil. Adiciona-se um nível de complexidade no momento em que começamos a inserir cláusulas no pragma que ocasiona o spawn de tasks porém o verdadeiro desafio é conseguir controlar o spawn de tasks e atingir um ponto ideal para garantir o máximo de paralelismo sem excesso de tasks que causem perda de desempenho.

Assim como usar tasks, paralelizar um mergesort também é fácil e a verdadeira dificuldade está em como paralelizar o merge. A abordagem de Cormen(2022) apesar de eficiente ainda deixa espaço para uma pergunta: E se dividíssemos o array em mais de duas partes para que o merge pudesse ser feito por 4 ou mais tasks? Além disso, cada setup de número de threads juntamente a um número de tasks por thread pode acabar tendo um ponto ideal diferente tanto para a parada do spawn de novas tasks quanto para a parada das chamadas recursivas, mostrando que o tuning de algoritmos desse tipo dificilmente terá uma fórmula generalizada.

A economia de cerca de 33% no tempo de execução com 16 threads mostra que mesmo sendo uma árdua tarefa, o trabalho de otimizar algoritmos paralelos é fortemente compensatório e merece ser estudado mas em diversos casos é possível que simplesmente trocar o compilador cause um impacto positivo ainda maior no desempenho.

Diversas abordagens de otimização (como utilizar outros algoritmos de ordenação em subarrays menores) não foram utilizadas neste trabalho apenas em razão da restritividade do escopo que visava analisar apenas o uso de tasks e paralelização do merge, deixando claro que haveria espaço para mais otimizações no algoritmo e novas reduções no tempo de execução.

REFERÊNCIAS

OPENMP Compilers & Tools. **OpenMP.org**, 2024. Disponível em: <https://www.openmp.org/resources/openmp-compilers-tools/>. Acesso em: 21/02/2024.

NESI, L. L.; MILETTO, M. C.; PINTO, V. G.; SCHNORR, L. M. Desenvolvimento de Aplicações Baseadas em Tarefas com OpenMP Tasks. **Minicursos da XXI Escola Regional de Alto Desempenho da Região Sul**, Porto Alegre, p. 129-150, 2021.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. **Introduction to Algorithms**. 4Th Edition. Cambridge, Massachusetts: The MIT Press, 2022.

TERBOVEN C.; KLEMM M.; VAN DER PAS R.; STOTZER E.; DE SUPINSKI B. R.; MATEO S., OpenMP Tasking. **Advanced OpenMP Tutorial**, p. 13-52, set. 2018. Disponível em: <http://bit.ly/201809tutorial/>. Acesso em: 7 fev. 2024.

APÊNDICE A – USO DO CHATGPT

Atualmente sugerido por muitos estudantes em final de curso, o uso do ChatGPT para geração de código neste caso se mostrou extremamente ineficiente, sendo inclusive difícil para a ferramenta reconhecer pedidos para a geração de um merge paralelo, confundindo constantemente a função de merge com o próprio merge sort apesar de oferecer um mergesort sequencial eficiente.

APÊNDICE B – DEMAIS CÓDIGOS

Merge sequencial utilizado como comparativo:

```
void merge(int arr[], int l, int m, int r) {

 int i, j, k;
 int n1 = m - l + 1;
 int n2 = r – m;
 int *L = (int*)malloc(n1 * sizeof(int));
 int *R = (int*)malloc(n2 * sizeof(int));
 for (i = 0; i < n1; i++)
  L[i] = arr[l + i];
 for (j = 0; j < n2; j++)
  R[j] = arr[m + 1 + j];
 i = 0;
 j = 0;
 k = l;
 while (i < n1 && j < n2) {
  if (L[i] <= R[j]) {
   arr[k] = L[i];
   i++;
  } else {
   arr[k] = R[j];
   j++;
  }
  k++;
 }
 while (i < n1) {
  arr[k] = L[i];
  i++;
  k++;
 }
 while (j < n2) {
  arr[k] = R[j];
  j++;
  k++;
 }
 free(L);
 free(R);
}
```

Mergesort single task:

```
void sinMergesort(int arr[], int p, int r, int task_size){
 if (p >= r) return;
 int q = (p + r) / 2;
 if(r - p >= task_size){
  #pragma omp task
   sinMergesort(arr, p, q, task_size);
   sinMergesort(arr, q + 1, r, task_size);
  #pragma omp taskwait
 }else{
  sinMergesort(arr, p, q, task_size);
  sinMergesort(arr, q + 1, r, task_size);
 }
 merge(arr, p, q, r);
}
```

Mergesort paralelo com merge sequencial:

```
void mergesort(int arr[], int p, int r, int task_size){
 if (p >= r) return;
 int q = (p + r) / 2;
 int size = r - p + 1;
 if(size > task_size){
  #pragma omp task untied
   mergesort(arr, p, q, task_size);
  #pragma omp task untied
   mergesort(arr, q + 1, r, task_size);
  #pragma omp taskwait
 }else{
  mergesort(arr, p, q, task_size);
  mergesort(arr, q + 1, r, task_size);
 }
 merge(arr, p, q, r);
}
```