Porte para *OpenMP* do Algoritmo *Streamcluster*

Angelo Leite Medeiros de Góes
20200000545
Jhonat Heberson Avelino de Souza
20200000680
Angelo Marcelino Cordeiro
20190152879
Maurício Thiago Ferreira de Lima
20180155222

I. Metodologia usada para reescrever o código em OpenMP

O porte do código foi conduzido através de três etapas. Primeiramente foram explicitadas, através da perfilagem realizada no primeiro relatório, as funções que apresentavam gargalo à aplicação. Dentre estas, o escopo foi ainda mais estreito a partir da análise de funções auto-vetorizadas pelo próprio compilador (g++ [1]) ao usar a *flag* de otimização nível 3 (-O3), como discutido no segundo relatório.

Em sequência foi trabalhada uma versão "limpa" da implementação inicial, o "streamcluster.cpp", com o objetivo de serialização do programa. Nesse estágio foram removidas todas as implementações de paralelismo pelo PARSEC [2], sendo elas "*threads building blocks*" (TBB) da Intel [3] e *pthreads*. Excluindo-se estas linhas foi obtida uma redução de aproximadamente 42% do tamanho total da aplicação, promovendo uma maior clareza na lógica de programação. Com o afastamento do uso de *pthreads*, a implementação de barreiras [4] pelo PARSEC "parsec_barrier.cpp" não seria mais compilado ou usado novamente.

Por fim foi feita uma nova perfilagem para confirmar os gargalos observados inicialmente e que fosse possível definir os laços de maior interesse a serem finalmente portados para OpenMP [5]. No caso, a função "*pgain*" foi confirmada como função mais exaustiva da aplicação, chegando ocupar praticamente 100% do tempo de execução, excluindo-se o, agora removido, *overhead* imposto pelas barreiras de *threads*. Esta é a função responsável pelo cálculo do custo de abertura de novos centroides.

Foram então temporizados todos os laços não vetorizados dentro da função através do método "*omp_ get_wtime*". Isto teve como objetivo descobrir onde se endereçava maior parte do gargalo. Com isso foi identificado o laço apresentado na Figura 1, que através de estudo foi observado deter 70% do tempo de "*pgain*". Sabendo disso, a paralelização deste trecho (considerando uma *speedup* linear deste) se espera obter um *speedup* total de 1.35 para o programa como um todo, considerando 2 *threads* (*nproc* = 2).

```
for ( i = 0; i < points->num; i++ ) {
  float x_cost = dist(points->p[i], points->p[x], points->dim) * points->p[i].weight;
  float current_cost = points->p[i].cost;

  if ( x_cost < current_cost ) {
    switch_membership[i] = 1;
    cost_of_opening_x += x_cost - current_cost;

  } else {
    int assign = points->p[i].assign;
    lower[center_table[assign]] += current_cost - x_cost;
  }
}
```

Fig. 1. Laço mais custoso

As *flags* de vetorização indicaram que a dificuldade se encontrava no controle de fluxo dentro do laço. Isto foi superado ao se mover a parte custosa para fora, criando um novo laço, desta vez paralelizável, e acumulando os resultados em um vetor a ser usado posteriormente. O resultado pode ser observado na Figura 2.

```
float* x_cost_arr = (float*)malloc(points->num*sizeof(float));
#pragma omp parallel num_threads(2)
  #pragma omp for
    for ( i = 0; i < points->num; i++ ) {
      x_cost_arr[i] =  dist(points->p[i], points->p[x], points->dim);
      x_cost_arr[i] *= points->p[i].weight;
    }

for ( i = 0; i < points->num; i++ ) {
  float current_cost = points->p[i].cost;

  if ( x_cost_arr[i] < current_cost ) {
    switch_membership[i] = 1;
    cost_of_opening_x += x_cost_arr[i] - current_cost;

  } else {
    int assign = points->p[i].assign;
    lower[center_table[assign]] += current_cost - x_cost_arr[i];
  }
}
```

Fig. 2. Laço otimizado

Mais tarde, estudando o fluxo de execução geral, ficou evidente que na aplicação de *pthreads* a paralelização ocorria de forma "global". Isto pois as *threads* eram criadas a partir de "*localsearch*", função chamada no início do programa, com todas as *threads* modificando apenas seus espaços de memória indexados por seus indexes de processo "*pid*", passados como argumento para todas as funções auxiliares. O uso de barreiras também era abundante por esse motivo, já que o paralelismo "abria" logo no início e "fechava" só no término da execução, precisando ser sincronizado ao longo do caminho.

A abordagem apresentada pelo grupo para paralelismo com OpenMP é mais simplificada, limpa e local, já que se instanciam *threads* pelas diretivas "*parallel*" e "*for*" só para os laços mais custosos, da função mais exaustiva presente "*pgain*". Num primeiro porte, também não foi observada necessidade de implementação de barreiras.

II. Conclusões

Após implementação e realização de testes de *benchmarking*, foi observada uma redução de aproximadamente 30% (1 - 57.4/81.6), próximo do ideal como esperado, para 2 *threads*. Os tempos podem ser observados no gráfico da Figura 3. Houve um aumento no tempo para 3 e 4 *threads*, esse comportamento é esperado já que o hardware atual é limitado a apenas 2 *threads* físicas e a troca de contexto adiciona *overhead* que piora o desempenho.

É conclusivo que após o porte para OpenMP obtiveram-se resultados positivos que melhoraram consideravelmente o desempenho da função "*pgain*", e do programa como um todo, refletido nos testes de temporização realizados.

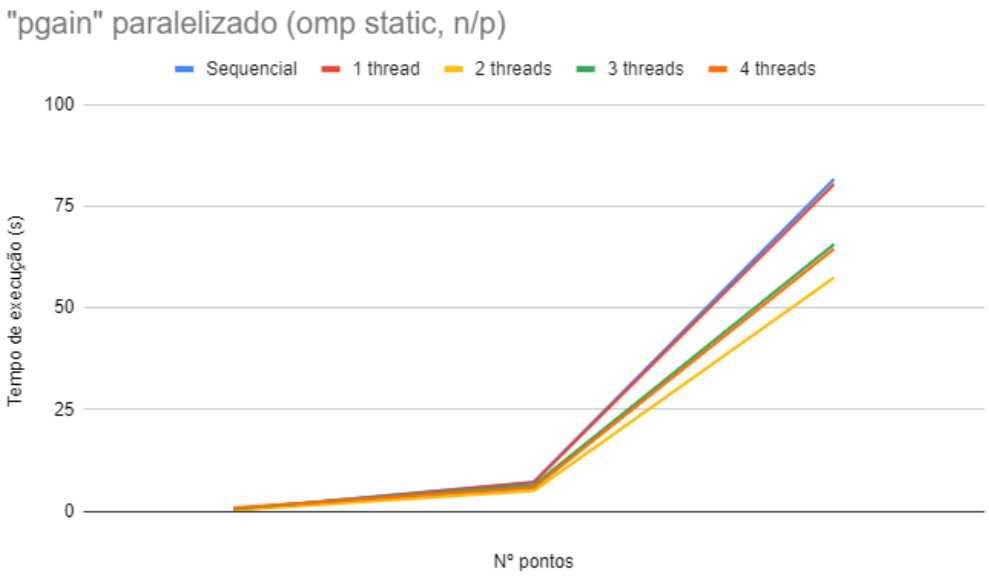

Fig. 3. Tempos de execução para 1000, 10000 e 100000 pontos

REFERÊNCIAS

[1] GCC, "Gcc, the gnu compiler collection," 2021. [Online]. Available: https://gcc.gnu.org/
[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," Princeton University, Tech. Rep. TR-811-08, January 2008.
[3] Intel, "Advanced hpc threading: Intel® oneapi threading building blocks," 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onetbb.html#gs.8qqi13
[4] *Introduction to barriers (pthread_barrier)*. YouTube, Dec 2020. [Online]. Available: https://www.youtube.com/watch?v=_P-HYxHsVPc
[5] OpenMP, Apr 2021. [Online]. Available: https://www.openmp.org/