Parallel design of Merge Sort algorithm

Marcelo Q. Pinto[1] and José V. Silva[1]

Abstract: Parallel computing helps in performing large computations by dividing the workload between more than one processor, all of which work through the computation at the same time. In this work, we present a parallel design of Merge Sort algorithm with almost null communication between processes. The results showed some scalability for big vetores (like 10 million integers), allowing gains greater than 5 for a 20-core machine and gains close to 3 for 6-core and 4-core machines.

Index Terms: Parallel Computing; High Performance; Concurrent execution.

1 Introdução

Computação paralela é uma área que tem impacto na otimização de algoritmos sequenciais, vinculando-se ao princípio de que é possível obter um programa mais eficiente, principalmente em análise de complexidade de tempo, utilizando várias linhas de execução. Podemos atingir este objetivo de várias formas, como por exemplo utilizando paralelismo de dados, paralelismo funcional ou em *pipeline*. Neste artigo vamos implementar paralelismo de dados, uma vez que vamos aplicar o mesmo algoritmo a várias partes do *dataset* em simultâneo.

Este trabalho visa a otimização de um algoritmo sequencial obtendo uma versão paralela eficiente, utilizando OpenMP. Propusemos a paralelização do algoritmo *Merge Sort* porque achamos importante a escolha de algoritmos diferentes entre os grupos de trabalho, incrementando a aprendizagem do coletivo, e porque o algoritmo *Merge Sort* é um algoritmo sequencial bastante eficiente, e por isso interessante.

Os resultados mostraram alguma escalabilidade para grandes vetores, como por exemplo 10 milhões de inteiros, permitindo ganhos de 5 numa máquina de 20 cores, e ganhos perto de 3 para máquinas de 4 e 6 cores.

2 Algoritmo Sequencial

2.1 Visão geral

O *Merge Sort* é um algoritmo de ordenação do tipo divisão e conquista (divide and conquer). O *vetor* inicial é dividido em elementos únicos (divisão) e consequentemente cada 2 conjuntos de elementos são ordenados recursivamente até se atingir o tamanho inicial do vetor (conquista).

Este algoritmo é um dos algoritmos de ordenação mais eficiente descrito na literatura, com complexidade O(n*log(n)).

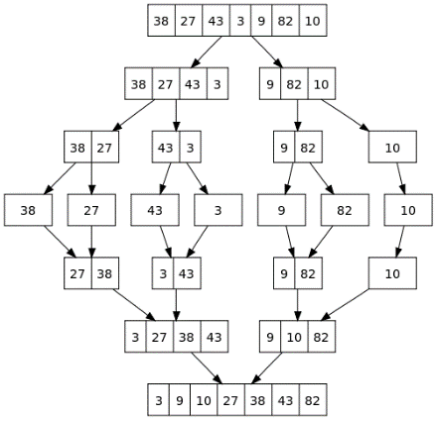

Fig. 1. Visão geral do algoritmo *Merge Sort*

2.2 *In-Place* vs *Out-of-Place Merge Sort*

Em algoritmos de ordenação, a versão *in-place* é normalmente vista como a ideal, uma vez que não requer (ou requer pouco) espaço extra além do vetor inicial, ao contrário da versão *out-of-place* que requer signitivamente mais espaço extra. No entanto, tal facto não é verificado no contexto do algoritmo *Merge Sort*: para melhorar a complexidade de espaço da versão *out-of-place* - O(n), uma vez que necessita como espaço extra, o mesmo do vetor inicial – e que possui uma complexidade de tempo O(n*log(n)), podemos utilizar a versão *in-place* do algoritmo *Merge Sort*, no entanto iremos ter complexidade de tempo de cerca de $O(n^2)$. Ora vejamos, enquanto que na versão *in-place* temos n trocas (mas copiando diretamente para um array) e log(n) níveis recursivos, totalizando numa complexidade de tempo O(n*log(n)), na versão *in-place*, apesar de a estrutura recursiva permanecer a mesma (correspondente à divisão), e por isso mantermos a parte da complexidade correspondente a lg(n), o movimento gerado pelos dados pode ser extremamente superior, uma vez que ao invés de necessitarmos de n movimentos de dados para a estrutura auxiliar (versão *out-of-place*), necessitamos de 3 operações para cada troca, utilizando uma variável exterior (temp=a, a=b e b=temp). Este

• [1] *Department of Informatics, University of Minho, Braga, Portugal. E-mail: mqspinto@gmail.com, a75280@alunos.uminho.pt*

processo leva-nos, no limite a uma complexidade de $O(n^2)$, e no caso médio a cerca de metade. Apesar disso, revela-se ainda mais eficiente que algoritmos como *bubble sort* ou *insertion sort*, teoricamente com a mesma complexidade. Assim, decidimos escolher a versão *out-of-place*, optando pela versão sequencial mais eficiente deste algoritmo.

3 Versão Paralela

3.1 Desenho

Partindo da versão sequencial mais otimizada do algoritmo *Merge Sort* encontrada na nossa pesquisa começamos por observar quais as zonas que poderiamos paralelizar e quais é que não poderíamos, e que poderiam dar origem a *data races*.

A zona chave paralelizável que encontramos neste algoritmo foi a recursividade utilizada pelo mesmo para dividir o vetor em metades até chegar a conjuntos de 1 elemento, e depois juntar os conjuntos, 2 a 2, ainda nas funções recursivas chamadas, até se atingir a totalidade do vetor, acabando na primeira instância da função recursiva utilizada. Assim, encontramos uma zona que permite a distribuição em várias tarefas, e que é chamada recursivamente (logo existirá divisão constante de tarefas) e consequentemente a reunião das 2 metades do vetor já ordenadas independentemente e recursivamente até à sua totalidade.

Outro aspeto são os acessos à memória, pois o *Merge Sort* é um algoritmo que consome alguma memória visto que a divisão em vários tamanhos ocupa espaço e devemos minimizar. Daí que tenhamos de ter em atenção aos tamanhos dos inputs pois se forem demasiado pequenos poderão ficar armazenados na *cache* L1 que é bastante rápida observando quase pouco ganho. Contudo também não podemos escolher tamanhos bastante grandes porque podem ficar armazenados em memória RAM que por si só traz custos não desejados. Na fase de experimentação explicaremos com mais detalhe este aspeto.

3.2 Implementação

Tendo em conta o desenho do nosso algoritmo vamos falar da sua implementação. Como dito anteriormente procedemos à paralelização da divisão do vetor, e para tal, optamos pelo uso da primitiva do *OpenMP task*. Esta primitiva permite criar uma pool de tasks para serem executadas pelas *thread*s disponíveis. Utilizamos esta primitiva sem parâmetros indicando que todas as variáveis da *task* são partilhadas pelas *threads*. Uma vantagem desta primitiva é aquando do *merge*, pois ao declararmos que cada metade do vetor fica distribuída em *x threads* não podemos juntar sem que a ordenação esteja feita. Assim, antes da chamada da função merge declaramos um *taskwait*, esperando pelo término das tasks a executar. Deste modo o *Merge Sort* sofreu modificações, nomeadamente nas duas chamadas recursivas do mesmo, passando a serem executadas cada uma por tasks.

Outra modificação foi considerar o número de *thread*s: a cada chamada recursiva diminuímos em metade o número inicial de *thread*s, para que quando tenhamos apenas uma *thread* possamos correr a versão sequencial, evitanto assim sobrecarga de tasks desnecessária, uma vez que todas as *thread*s já estarão a ser utilizadas. Além disso, isto permite que o tempo de execução melhore, uma vez que não queremos tarefas muito pequenas, aumentando o overload de paralelização sem trazer beneficios.

Outro aspeto implementado foi a declaração da zona paralela. Como queremos apenas observar a paralelização do algoritmo de *Merge Sort* declaramos como zona paralela a chamada da mesma usando omp parallel e de seguida utilizar o omp single para que as tasks criadas no *Merge Sort* fossem geradas apenas uma vez.

3.3 Otimização

Uma otimização realizada à nossa versão inicial foi acrescentar o *Merge Sort* sequencial quando já temos todas as *thread*s a trabalhar. Assim diminuíamos o overload da paralelização quando esta já não teria efeito.

Assim, o parâmetro *thread*s permite-nos estimar quando é que o número de *thread*s termina e assim começar a utilizar a versão sequencial. Este parâmetro é setado da seguinte forma: na primeira chamada recursiva queremos que sejam dadas metade das *thread*s, por isso temos *thread*s/2 e na segunda chamada da mesma função mandamos as restantes, corresponde a fazer *thread*s-(*thread*s/2). Assim, temos log(*thread*) níveis até esgotarmos as *thread*s e começarmos a executar a versão sequencial.

Outra forma de otimização equivalente e testada (e que oferecia tempos equivalentes), foi implementar aquando do número de *thread*s for igual a 2 uma paralelização com duas chamadas sequenciais, uma para cada *thread*. Após alguns testes concluímos que os tempos andavam próximos da versão que apresentamos.

Assim, verificamos que a nossa versão não tem comunicação entre processos, e, portanto, neste ponto não oferece qualquer problema de paralelização. No entanto, pode existir má gestão de balanceamento de carga, problema inerente a qualquer paralelização de algoritmos de ordenação, uma vez que é impossível dividir os dados com conhecimento do número de trocas que haverá nas diferentes partes do *dataset*.

4 Variáveis de experimentação

Decidiu-se utilizar 3 vetores de inteiros, de diferentes tamanhos, para realizar testes, sendo eles de tamanhos:

- 80KB (10 mil inteiros), ocupando memória inferior à *cache* L2 de uma máquina comum;
- 400KB (50 mil inteiros), ocupando memória inferior à *cache* L3 de uma máquina comum, mas superior à *cache* L2 de uma máquina comum;
- 80MB (10 milhões de inteiros), ocupando memória superior a qualquer *cache* L3 que iremos utilizar.

Os vetores foram gerados em ordem decrescente, uma vez que desta forma garantimos que o número de trocas é

sempre igual em vetores do mesmo tamanho e sempre proporcional em vetores de tamanho diferente. Ao contrário da geração de inteiros aleatórios, esta técnica enibe qualquer alteração de valores de tempo por fruto de mais ou menos trocas, uma vez que os algoritmos de ordenação, e em particular o *Merge Sort*, são muito suscetíveis ao número de trocas a efetuar aquando da análise da complexidade de tempo. Garantimos também que a geração dos vetores, e qualquer output não é tido em conta na contagem do tempo de execução do algoritmo. Evitando assim um mau balanceamento de carga.

As máquinas utilizadas para testes são:

- Um nó do cluster (número 652-2) do departamento de informática da Universidade do Minho, denominado SeARCH, contendo, esse nó, 2 processadores com 10 *cores* cada, tendo L2=256KB e L3=20MB;
- Um desktop com processador AMD Ryzen 1600, de arquitetura Zen, seis *cores* e doze *threads*, *hexacore* com 512KB de *cache* L2 e 16MB de *cache* L3;
- Um laptop com processador Intel Pentium N3700, arquitetura Braswell, *quadcore* e quatro *threads*, com dual L2=512KB sem L3.

O método de contagem utilizado foi a diretiva *omp_get_wtime()*. Foram colocadas duas variáveis, *start* como inicial e *stop* como final, a delimitar o algoritmo, e no final é imprimida a diferença entre stop e start obtendo-se assim os tempos de experimentação.

Todas as versões dos testes foram compiladas utilizando o gcc com a otimização -O3 e com suporte para ISO C99 (-std=c99) chamando a diretiva -fopenmp, para correr o OpenMP e com a livraria Math conetada (-lm).

5 Análise dos resultados

5.1 Valores teóricos

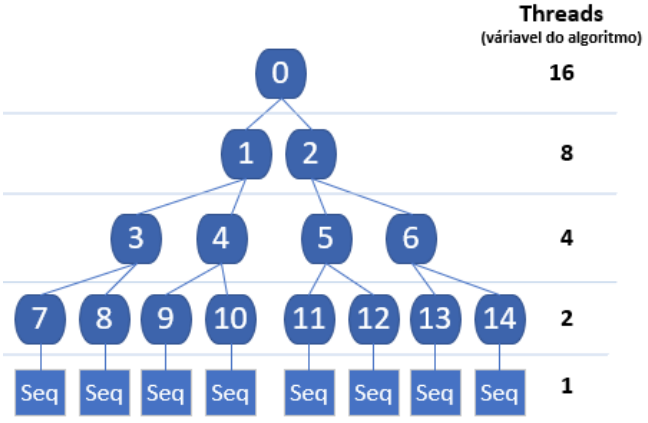

Fig. 2. Chamada de Threads em recurvidade

O algoritmo em análise não permite ganhos ideais, uma vez que não permite a execução paralela do total número de threads: como podemos observar na Fig. 2. a thread 0 efetua 2 chamadas (thread 1 e 2) e só continua a sua execução aquando do término destas. O mesmo processo acontece nos níveis de recursividade seguintes, não permitindo que 2 níveis de recursividade se efetuem em paralelo (primeiro as threads filhas terão de terminar para a thread mãe continuar a sua execução).

Assim, verificamos que o número máximo de threads a correr em simultâneo são as do nível para o qual não existem mais threads disponíveis (penúltimo nível da figura), limitando logo o ganho a metade do ideal para um algoritmo comum, uma vez que o último nível possui metade das threads iniciais. Este nível é alcançado a partir de um parâmetro threads setado com o número inicial de threads e que se divide em 2 à medida que acontece um nível de recursividade. Quando este parâmetro é 1, as threads irão executar a versão sequencial, evitanto overload desnecessário com a versão paralela.

Uma vez que decorre um processo de divisão em threads, o algoritmo sofrerá também um atraso proporcional aos níveis chamados (log(threads)) e ao tamanho do vetor de input (uma vez que terá de copiar dados). Assim, a versão paralela terá tempo de execução (TPara) proporcional à equação:

em que TSeq é a versão sequencial e recebe um parâmetro

$$TPara = TSeq\left(\frac{tamanhoArray}{\frac{threads}{2}}\right) \times \frac{threads}{2} + T$$

tamanho de array: como esta versão é chamada threads/2 vezes existe a multiplicação descrita e o tamanho do array foi dividido recursivamente, equivalendo a dividir por threads/2. O T representa todos os níveis recursivos antes de ser chamada a versão sequencial. Exemplo: Para 16 threads e um *input* inicial de 8 mil inteiros existem 4 níveis de recursividade, os quais demoram T tempo, assim temos já 7 threads paradas e consequentemente vamos utilizar 8 threads em simultâneo (que são threads/2) a correr a versão sequencial cada uma com um input de mil inteiros. Estas threads vão correr a versão sequencial porque existe um parâmetro Threads no algoritmo que é dividido por 2 em cada nível e ao chegar a 1 é executada a versão sequencial. O tempo final será o tempo de cada thread multiplicado pelas threads que estão a correr a versão sequencial (que são threads/2 ou neste caso 8) somando com o tempo das threads que dividiram o algoritmo até esse nível e que depois juntaram os vetores ordenandos das threads inferiores (neste caso 8 arrays de mil inteiros).

Assim, apesar de termos pouca precisão na variável T, conseguimos prever que os melhores ganhos serão para o dobro de threads ideias num algoritmo comum, uma vez que, em paralelo, conseguimos correr, no máximo e em grande parte do algoritmo, metade das threads em paralelo.

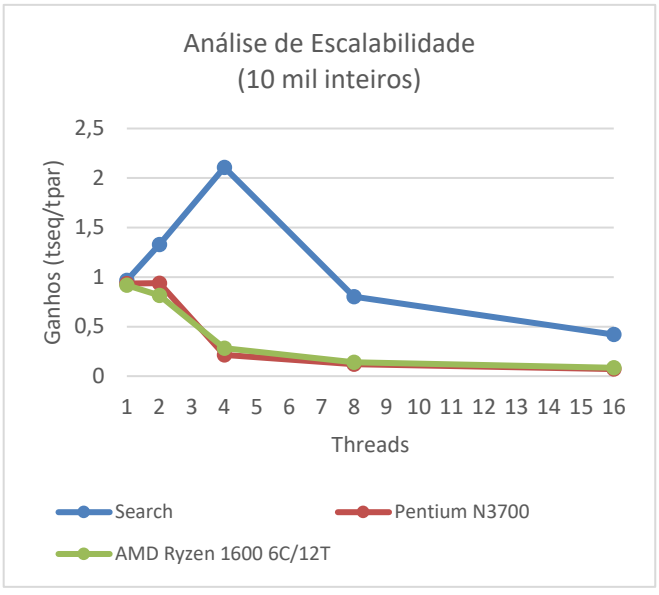

Fig. 3. Análise da escalibilidade – 10 mil inteiros

5.2 Vetor de 10.000 inteiros

Na Fig. 2 observamos a curva de *speedups* correspondente ao vetor de 10 mil inteiros. Podemos observar que, neste caso, o algoritmo consegue escalar até 4 *threads* (com pouco mais de 2 de ganho) numa boa máquina como o SeARCH, mas nas outras 2 máquinas não é escalável. O overload provocado pela versão paralela não justifica a utilização da mesma para este tamanho de vetor (10 mil inteiros), uma vez que estamos a lidar com apenas 80 KB de dados. Estes dados não cabem na *cache* L1 de qualquer uma das máquinas, mas cabem na *cache* L2 de todas. Como para este *dataset* se obtivemos cerca de 0.00064 segundos para a versão sequencial, não seria expectável uma melhora ao paralelizar.

Contudo seria de esperar que nas nossas máquinas que houvesse um ganho em vez de um speeddown com a utilização de 4 *threads* pois, ambas apresentam 4 cores físicos. Tal não aconteceu porque a comunicação entre cores e memória *cache* não é otimizada para situações de paralelismo entre *threads*. Já no SeARCH, observamos um ganho de pouco mais de metade do ganho ideal, para 4 *threads*, reforçando a ideia que, para 10 mil inteiros, a paralelização não é fácil. Este resultado deve-se ao facto de o SeARCH estar otimizado para paralelização, oferecendo menos *overload*, e ao facto de não ser muito bom com versões sequenciais.

De realçar que o vetor inicial é dividido em sub-vetores e que quando estes subvetores possuem menos de 32KB conseguem ser armazenados na *cache* L1, melhorando a escalabilidade até ao número máximo de processadores. No entanto, para este caso, este fator traz pouca influência uma vez que o conjunto inicial é pequeno.

Aquando de *threads*>4 no caso do SeARCH e no caso de qualquer paralelização nas outras máquinas, todas sofrem um overload de paralelização que não compensa o tempo que a versão sequencial demora.

5.3 Vetor de 50 mil inteiros

Quanto ao teste de *speedup* com 50 mil inteiros, que podemos observar na Fig. 3., podemos observar novamente um cenário semelhante ao vetor de 10 mil inteiros. Neste caso, o SeARCH não possui *cache* L2 suficiente para armazenar todos os dados (400KB para uma *cache* de 256KB), no entanto as outras duas máquinas possuem uma *cache* de 512KB.

Verificamos que a escalabilidade de todas as máquinas,

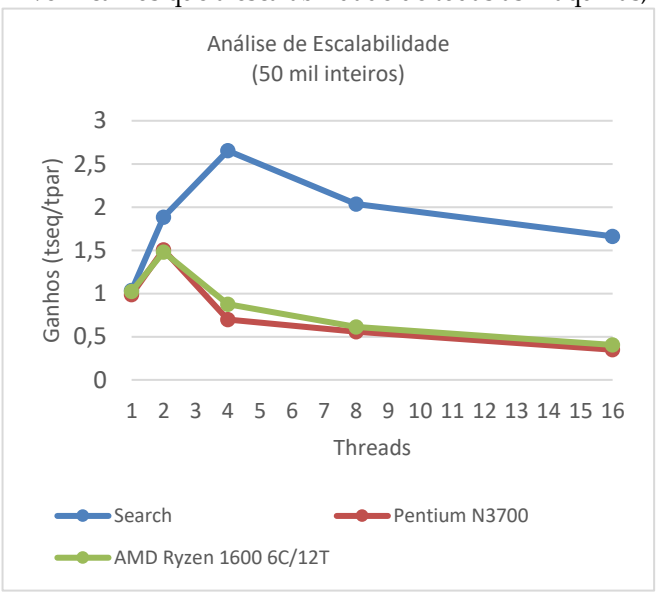

Fig. 4. Análise da escalibilidade – 50 mil inteiros

para este input melhora consideravelmente: tendo em conta 4 *threads* temos valores como 0.7, 0.9 e 2.7, ao qual para 10 mil inteiros, tínhamos respetivamente valores como 0.4, 0.6 e 2.2. Assim, podemos concluir que o overload provocado pela paralelização passa muito mais despercebido para este caso uma vez que o algoritmo ordena 5x mais dados.

De salientar que os ganhos no SeARCH não são tão grandes como as outras máquinas em relação ao vetor de 10 mil inteiros, isto porque o SeARCH não consegue armazenar os 50 mil inteiros na *cache* L2 (400KB > 256KB) e recorre à cache L3 (20 megas), ao passo que as outras 2 máquinas conseguem armazenas na *cache* L2 (mas como já explicado, não possuem mecanismos ótimos de paralelização como o SeARCH.) No entanto o SeARCH consegue armazenar na *cache* L2 metade do vetor (200KB) e por isso após a primeira chamada recursiva de tasks, o SeARCH consegue lidar com os sub-vetores utilizando a *cache* L2.

De realçar que o vetor inicial é dividido em sub-vetores e que quando estes subvetores possuem menos de 32KB conseguem ser armazenados na *cache* L1, melhorando a escalabilidade até ao número máximo de processadores de cada máquina (nº das *caches* L1). No entanto, para este caso, os fatores já falados têm maior peso nos tempos demonstrados.

Aquando o número de *threads*>4 no caso do SeARCH e *threads*>2 no caso das outras máquinas, todas sofrem um

overload de paralelização que não compensa o tempo que a versão sequencial demora.

5.4 Vetor de 10 milhões inteiros

Na Fig. 4 observamos a curva de *speedups* correspondente ao vetor de 10 milhões de inteiros. Este vetor não é suportado por qualquer *cache* nem conjunto de *caches* entre processadores (como um todo), uma vez que ocupa 80MB. Apesar de os acessos à memória serem obrigatórios, estes restringem-se aos primeiros níveis de recursividade, uma vez que para o SeARCH, por exemplo, em 2 níveis de recursividade já temos 4 vetores de 20MB cada, ao passo que 1 deles cabe na *cache* L3. Para mais 7 chamadas recursivas obtemos 128 vetores de 156KB e a partir daí a *cache* L2 será utilizada nos 20 processadores (156KB<256KB). Assim é facilmente explicada a grande escalabilidade deste algoritmo para grandes vetores.

No entanto, o ponto mais eficiente de escalabilidade será antes (para *threads*=32 e para *threads*=16 no caso do Pentium N3700) uma vez que há um jogo entre overload de paralelização e divisão eficiente do vetor em *thread*s. Uma vez que o número de processadores é limitado, o jogo acabará com o overload a dominar, uma vez que um grande número de *thread*s também será insuportável. No entanto, uma vez que o `if(threads==1)` obriga à utilização da versão sequencial para `threads=1` o overload é largamente diminuído e por isso esta versão prevê-se estável, não piorando muito.

Para este teste decidimos explorar a capacidade total de cores que temos nas máquinas onde realizamos as experiências. Assim sendo exploramos 6 cores por ser a capacidade física total do AMD Ryzen, 10 por ser a capacidade física de apenas uma *processing unit* do nó SeARCH, 20 por ser a capacidade física das duas *processing units* do nó e por fim testamos a hipótese de *hyperthreading* com 40 *thread*s. Além disso quisemos avançar com o número de *thread*s até acharmos que começa a estabilizar. Neste teste conseguimos observar melhor porque é que o SeARCH escala melhor que todas as outras máquinas. Uma vez que este input sai fora da *cache* L3 terá que aceder à memória física. Este acesso à memória física é bastante penoso para as nossas máquinas pois não têm memória cuja largura banda é bastante grande, podendo distribuir mais rapidamente os segmentos do vetor e juntá-los mais rapidamente. Além disso possui suporte para *quad-channel* que é bastante rápido comparado com o *dual channel* utilizado na máquina AMD e ainda mais rápido comparado com o *single channel* do Pentium.

Observando os gráficos dos *speedups* utilizados (versão sequencial/versão paralela) verificamos que, tal como esperado nos valores teóricos, os melhores ganhos ganhos acontecem para o dobro de threads necessárias para utilizar todos os cores, uma vez que este algoritmo só consegue ter em paralelo metade das threads.

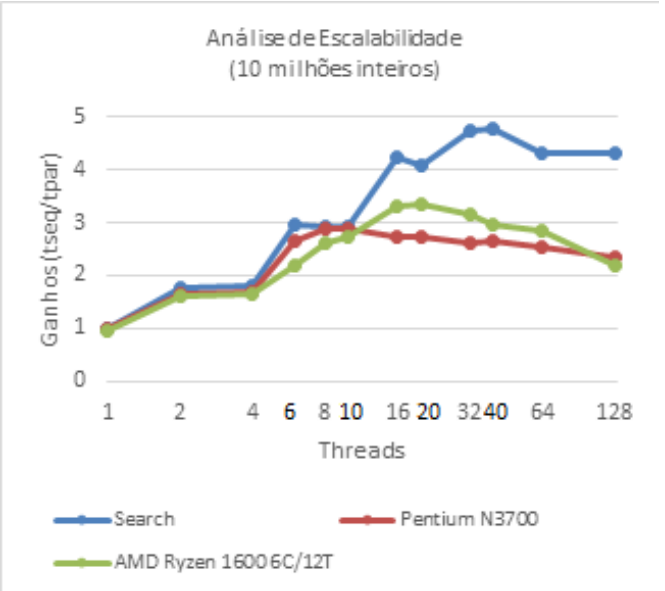

Fig. 5. Análise da escalibilidade – 10 milhões de inteiros (escala logaritmica)

6 Conclusão

Esta versão do algoritmo *Merge Sort* apresenta resultados médios em termos de escalabilidade para grandes inputs, apesar de longe dos ganhos ideais. Não é adequeada para pequenos inputs, sendo preferencial, nesse caso, a versão sequencial.

Mostrou-se, claramente, que a solução apresentada, conciliando a paralelização com a versão sequencial, aquando do número de *thread*s se esgotar, permite melhorar a versão sequencial correndo normalmente num único core (e que é um dos algoritmos de ordenação mais eficientes descritos na literatura).

Agradecimentos

Agradecemos aos professores que lecionam a cadeira de Computação Paralela, pelos conhecimentos que nos permitiram adquirir.

Referências

[1] A. Davidson, D. Tarjan, M. Garland, J. Owens. “Efficient Parallel Merge Sort for Fixed and Variable Length Keys”, Innovative Parallel Computing (InPar), 2012.

[2] M. Albutiu, A. Kemper, T. Newmann. “Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems”, Proceedings of the VLDB Endowment - Volume 5 - Issue 10, June 2012.

[3] R. Nicole, T. Dobravec, “Comparison of parallel sorting algorithms", Cornell University Library, November 2015.

[4] G. Blelloch, C. Leiserson, B. Maggs. “A comparison of sorting algorithms for the connection machine CM-2”, 91 Proceedings of the third annual ACM symposium on Parallel algorithms and architectures - Pages 3-16, 1991.