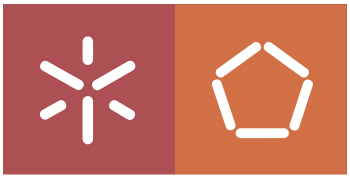


Universidade do Minho
Escola de Engenharia

Mestrado em Engenharia Informática

Computação Paralela

OpenMP

PARADIGMA DE MEMÓRIA PARTILHADA

Bruno Carvalho, a83851

Joana Afonso Gomes, a84912

5 de janeiro de 2022

Conteúdo

Lista de Figuras

1 INTRODUÇÃO

Foi-nos proposto neste trabalho, no âmbito da UC de **Computação Paralela**, paralelizar um algoritmo, implementando também inicialmente a sua versão sequencial, e analisando todas as vertentes merecedoras perante este estudo.

De entre as opções disponível selecionamos a **3ª opção**, o algoritmo ***quicksort***: um algoritmo de ordenação do estilo *divide-and-conquer*, desenvolvido por Tony Hoare em 1959, e geralmente reconhecido como sendo um algoritmo de ordenação bastante rápido e eficiente [1].

Para o seu desenvolvimento foi, por isso, necessário desenhar e delinear atentamente a versão paralela, experimentando e analisando os resultados com diversos *inputs* e medições.

Começa-se por fazer uma análise do código do algoritmo, mais especificamente olhando para as suas versões sequenciais e paralelas. De seguida, são apresentados os resultados de vários testes de desempenho, juntamente com a sua interpretação. Por fim, analisa-se quais as limitações desta abordagem e que conclusões se podem retirar.

2 MAIN

```
if (PARALLEL) {
    #pragma omp parallel num_threads(ctr)
    #pragma omp single
    {
        int cur_t = omp_get_num_threads();
        if (CUTOFF != -1) {
            int frac = n / cur_t;
            CUTOFF = (frac > 5000) ? 5000 : frac;
        }
        p_quicksort(arr, 0, n - 1);
        #pragma omp taskwait
    }
} else s_quicksort(arr, 0, n - 1);
```

Listing 1: Corpo principal do programa

A parte principal da função *main* pode ser vista na Listagem 1. Começa-se por verificar se o utilizador pediu ao programa para correr a versão paralela ou

sequencial. Se o programa estiver a ser executado na versão paralela, o número de *threads* é definido conforme a indicação do utilizador (o maior número de *threads* disponível é assumido por omissão). É também indicado que apenas uma *thread* desta equipa deve executar o bloco seguinte, de forma a garantir uma estrutura de tarefas correta. De seguida, é calculado o ponto de quebra do paralelismo como sendo menor valor entre a divisão por igual entre o tamanho dos dados e o número de *threads*, e 5000 elementos, salvo indicação contrária por parte do utilizador. A avaliação do desempenho resultante desta escolha pode ser vista na Secção 5. A execução do programa continua com a chamada da função de ordenação paralela, e, por fim, com uma cláusula de espera até que todas as *tasks* geradas pela função tenham sido terminadas, de forma a garantir um conjunto de dados devidamente ordenado.

A execução do programa pode ser configurada com as seguintes opções:

- **-seq**: Executar em modo sequencial.
- **-p**: Executar em modo paralelo (ativo por omissão).
- **-s [tamanho]**: Indica o tamanho do conjunto de dados.
- **-t [quantidade]** Indica o número de *threads* a lançar.
- **–loop-threads**: Cria um ciclo de execução que utiliza desde uma ao valor máximo de *threads* disponíveis.
- **–loop-size**: Cria um ciclo de execução que utiliza um conjunto de dados proporcional ao número de *threads*, sendo estas aumentadas de uma até ao valor máximo disponível.
- **–no-cutoff**: Executar sem quebra de paralelismo.

O conjunto de dados a ordenar é inicializado com valores aleatórios gerados pela função `rand()` da *stdlib* do C.

3 VERSÃO SEQUENCIAL

```
void s_quicksort(int* arr, long lo, long hi) {
    long i = lo, j = hi;
    int pi = arr[(lo + hi) / 2];

    // Partition.
    do {
        while(arr[i] < pi) i++;
        while(arr[j] > pi) j--;
        if(i <= j) {
            int temp = arr[i];
            arr[i++] = arr[j];
            arr[j--] = temp;
        }
    } while(i <= j);

    if (lo < j) s_quicksort(arr, lo, j);
    if (i < hi) s_quicksort(arr, i, hi);
}
```

Listing 2: Versão sequencial

Na Listagem 2, encontra-se a versão sequencial do *quicksort*, utilizada como ponto de partida para este trabalho. Não foram feitas grandes modificações em comparação com aquilo que é o algoritmo comum, uma vez que é um algoritmo já bastante estudado e otimizado.

4 Versão Paralela

```
void p_quicksort(int* arr, long lo, long hi) {
    if (CUTOFF > 0 && hi - lo < CUTOFF)
        s_quicksort(arr, lo, hi);
    else {
        // Partition.
        long i = lo, j = hi;
        int pi = arr[(lo + hi) / 2];
        do {
            while(arr[i] < pi) i++;
            while(arr[j] > pi) j--;
            if(i <= j) {
                int temp = arr[i];
                arr[i++] = arr[j];
                arr[j--] = temp;
```

```
15              }
16          } while(i <= j);
17
18          if (lo < j)
19              #pragma omp task
20              p_quicksort(arr, lo, j);
21          if (i < hi)
22              #pragma omp task
23              p_quicksort(arr, i, hi);
24      }
25  }
```

Listing 3: Versão paralela

Na Listagem 3 podemos ver a versão paralela do *quicksort*. Começa-se por comparar a amplitude de valores com que a *thread* terá que trabalhar com um `CUTOFF`, ou quebra, de paralelismo. Se for inferior ao valor definido para o `CUTOFF`, a *thread* passará para a versão sequencial do programa. Caso a amplitude de valores seja superior, a *thread* prossegue na versão paralela, em que faz o algoritmo de particionamento normal e, de seguida, cria duas *tasks* a serem colocadas na *taskpool*: uma que continuará a recursividade do *quicksort* para os valores inferiores ao *pivot*, e outra que o fará para os valores superiores.

Na Figura 1 encontra-se uma representação da execução do algoritmo, assim como uma indicação da criação de *tasks* do OpenMP. Podemos observar, desde já, que é possível que exista um mau balanceamento de carga, na medida em que o trabalho de cada *thread*, na fase de partição dos dados, não é garantidamente igual para todas. Um exemplo disto encontra-se logo no fim da primeira iteração e início da segunda (segunda linha a contar de cima), em que são criadas duas tarefas, mas que a da esquerda tem mais dois elementos para percorrer que a da direita. Uma possível solução seria fazer com que a escolha do *pivot* tentasse aproximar este à mediana dos valores a serem ordenados, de forma a equilibrar a carga de trabalho atribuída a cada par de tarefas criado, mas tal mecanismo não foi implementado na solução aqui alvo de discussão.

Vemos também que, num cenário perfeito, cada *thread* iria criar duas *tasks* com igual carga de trabalho, e cada uma seria executada por uma *thread*, dando origem a quatro tarefas, e assim sucessivamente. Desta forma, fazendo s a duração da primeira iteração (a parte sequencial do algoritmo), t o número de *threads* e y a duração do algoritmo, temos que

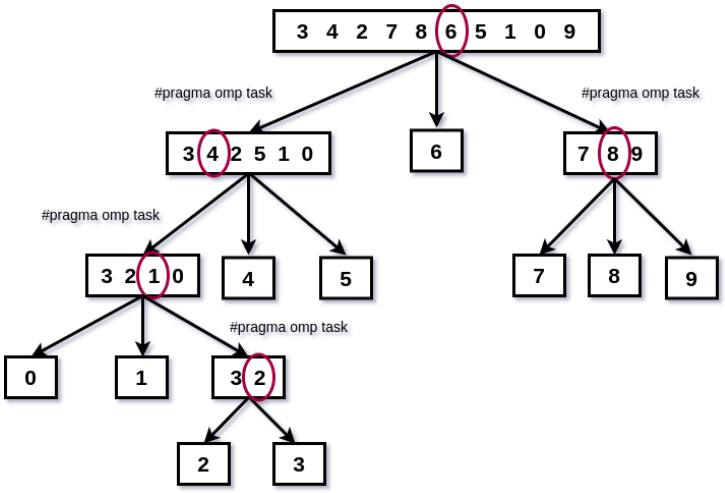


Figura 1: Diagrama de execução do algoritmo

$$y = s + \sum_{t=2}^{+\infty} \frac{s}{t}$$

o que nos diz que a duração deste algoritmo é limitada inferiormente pelo desempenho da sua secção sequencial, e que esta tem um impacto em todas as iterações seguintes. De notar, também, que a criação de tarefas é um processo iterativo, não sendo possível utilizar de imediato o elevado número de *threads* à nossa disposição.

5 Análise de Desempenho

A análise do desempenho do algoritmo está dividida em três categorias, consoante o ponto em que ocorre a quebra de paralelismo: sem quebra, quebra consoante o número de *threads* de execução, e quebra aos cinco mil elementos. Em cada um destes casos, será analisada também a forma com que o algoritmo evolui em termos de escalabilidade forte e fraca. No que toca à escalabilidade forte, o conjunto de dados manteve-se nos $5 * 10^7$ elementos, aumentando-se progressivamente o número de *threads* de uma até quarenta. Para a análise de escalabilidade fraca, o número de *threads* foi na mesma aumentada de uma até quarenta, e o tamanho do conjunto de dados estava definido como sendo o

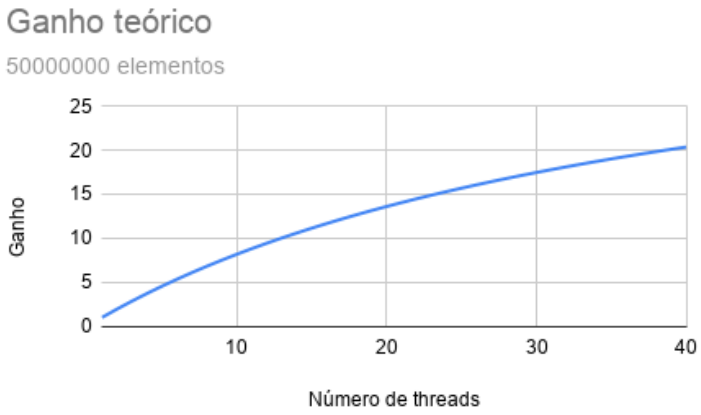


Figura 2: Ganho teórico calculado a partir da Lei de Amdahl

número de *threads* em execução multiplicado por 10^6.

Uma análise do algoritmo na sua forma sequencial, através do cálculo da média do tempo que a primeira iteração (que é a fase sequencial do algoritmo na versão paralela) leva, a partir de cinquenta observações, permite-nos fazer estimativas acerca do potencial que o pararelismo traz. A partir da Lei de Amdahl, podemos calcular o ganho máximo do paralelismo para este algoritmo, tendo em conta a sua fração de execução sequencial. Na Figura 2 encontra-se o gráfico de

$$S_p = \frac{1}{f + \frac{1-f}{p}}$$

em que f é a média da fração sequencial do algoritmo (2.474%) e p é o número de *threads*. Isto indica-nos qual o maior ganho possível, em teoria, para cada número de *threads* em execução. No limite, este seria

$$\lim_{p \to +\infty} \frac{1}{0.02474 + \frac{1-0.02474}{p}} \approx 40.42$$

O cálculo do ganho das diferentes versões do algoritmo paralelo que serão estudadas é feito em comparação com a duração média da versão sequencial para o mesmo número de elementos, sendo esta 6.892 segundos.

Os gráficos das secções seguintes foram compilados com dados retirados da execução do algoritmo no SeARCH, salvo indicação em contrário.

5.1 Sem quebra de paralelismo

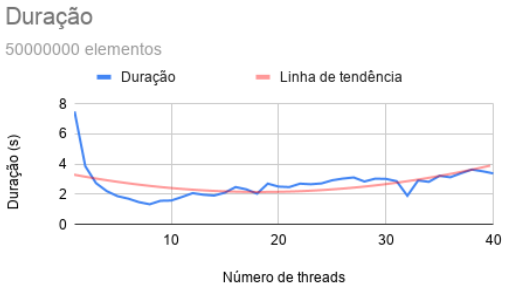
No gráfico da Figura 3a encontra-se representada a duração da execução do algoritmo sem quebra de paralelismo. Podemos ver a partir da linha a vermelho, representativa da tendência dos dados, que a duração do algoritmo começa por cair, mas volta a subir a partir das dez *theads*, provavelmente devido ao trabalho extra que vem com a criação e gestão destas, assim como das tarefas do OpenMP. Uma vez que não existe quebra de paralelismo, cada tarefa só deixa de criar outras duas quando tem a seu cargo menos de três elementos, o que significa que a árvore de tarefas terá uma profundidade bastante elevada, causando aqui uma perda de desempenho. Por outras palavras, a granularidade do paralelismo aparenta ser demasiado fina.

Da mesma forma, podemos ver na Figura 3b que o ganho de desempenho aumenta até perto de cinco, com cerca de dez *threads*, começando depois a cair. Comparando com os valores calculados para o ganho teórico, estes encontram-se muito abaixo.

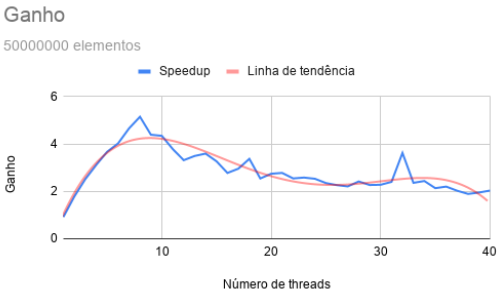
Por fim, na Figura 3c está representada a fração de tempo de execução que corresponde à primeira iteração do *quicksort*, que é a parte sequencial do algoritmo. Nela, podemos ver que a sua média se mantém entre os cinco e os dez por cento.

O gráfico representativo da escalabilidade fraca do algoritmo pode ser visto na Figura 4. Nele, vemos que a duração da execução quando o tamanho dos dados é diretamente proporcional ao número de *threads* disponíveis não é constante, o que seria o pretendido. A explicação dada anteriormente pode aplicar-se também aqui, visto que a granularidade do paralelismo parece ser demasiado fina, o que causa uma degradação do desempenho notável quando se aumenta o número de *threads*. O facto de o número de elementos a tratar aumentar implica também um aumento da duração da secção sequencial do algoritmo, o que faz aumentar o tempo geral de execução.

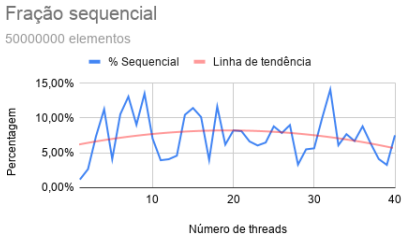
Na Figura 5 encontra-se o resultado da execução do comando *perf* no algoritmo, num computador portátil. Vemos que, como não existe quebra de paralelismo, toda a parte de ordenação é passada na versão paralela, designada



(a) Duração em segundos



(b) Ganho



(c) Fração de tempo da execução sequencial

Figura 3: Escalabilidade forte do algoritmo sem quebra de paralelismo

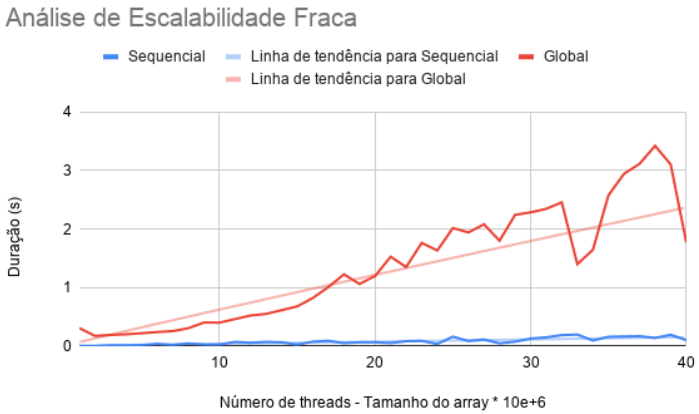


Figura 4: Escalabilidade fraca do algoritmo sem quebra de paralelismo

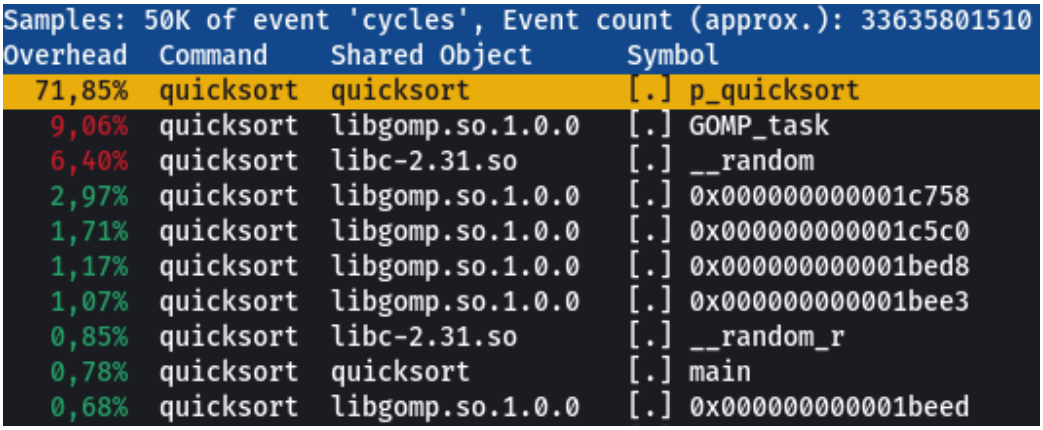


Figura 5: Perfil de execução do algoritmo sem quebra de paralelismo

por *p_quicksort*.

5.2 Quebra de paralelismo definida pelo número de *threads*

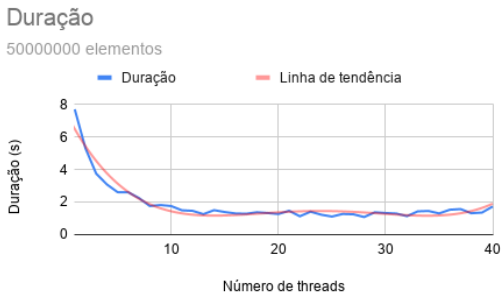
Aqui, estuda-se o caso em que se define a quebra de paralelismo tendo em conta o número de *threads*, sendo este determinado por

$$cut_{par} = \frac{n}{t}$$

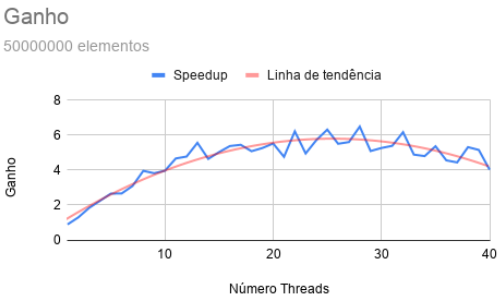
em que n é o número de elementos a ordenar e t o número de *threads*.

No gráfico da Figura 6a podemos ver como evolui o tempo de execução do algoritmo com o aumento de número de *threads*. Inicialmente, esta aproxima-se dos oito segundos, caindo até cerca de um segundo numa trajetória logarítmica. De notar que já não acontece de o tempo de execução voltar a subir consideravelmente a partir de um determinado número de *threads*, pelo menos até quarenta, como verificado no caso anterior. Tal parece indicar que esta abordagem para a quebra de paralelismo, que implica uma granularidade mais grossa, possui uma melhor escalabilidade.

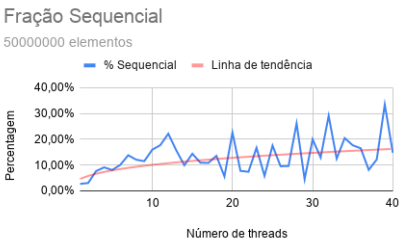
Na Figura 6b, encontra-se um gráfico que apresenta a evolução do ganho do algoritmo. Vemos que a versão paralela chega a ser seis vezes mais rápida que a versão sequencial, mas há um ligeiro declínio a partir das trinta *threads*. Apesar de o ganho ser maior que o observado na secção anterior, e de manter uma certa



(a) Duração em segundos



(b) Ganho



(c) Fração de tempo da execução sequencial

Figura 6: Escalabilidade forte do algoritmo com quebra de paralelismo em função do número de *threads*

estabilidade ao atingir o máximo, continua a ser muito inferior ao valor teórico calculado.

No que toca à fração de tempo passada em execução sequencial, vemos, por observação do gráfico da Figura 6c, que a média deste parece rondar os quinze por cento.

A escalabilidade fraca do algoritmo pode ser analisada a partir do gráfico da Figura 7. Nele vemos que, mais uma vez, o tempo de execução não se mantém constante quando o aumento de carga é diretamente proporcional ao número de *threads*. As justificações indicadas na secção anterior aplica-se também aqui.

Na Figura 8 podemos ver o perfil de execução do algoritmo; neste caso, com quebra de paralelismo. Já são utilizadas tanto a versão paralela como a sequencial, com prevalência para esta última (75.21% contra 9.50%). Esta grande disparidade pode ser vista como um indicador de que a quebra de paralelismo

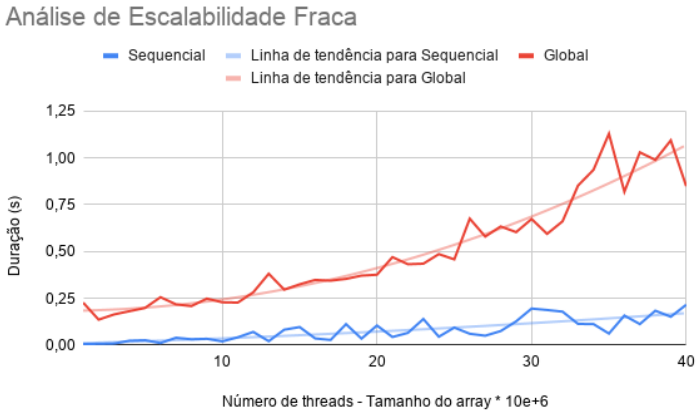


Figura 7: Escalabilidade fraca do algoritmo com quebra de paralelismo em função do número de *threads*

definida aqui é demasiado elevada, estando, portanto, a desperdiçar potencial.

5.3 Quebra de paralelismo aos 5000 elementos

Por fim, definiu-se que a quebra de paralelismo deveria ocorrer aos cinco mil elementos. Este valor mostrou ser um bom ponto para quebrar o paralelismo, pela realização de várias experiências com valores distintos.

Através da observação do gráfico da Figura 9a, vemos que o tempo de execução do algoritmo começa a rondar os oito segundos, caindo numa trajetória logarítmica até menos de um segundo. Não se verifica uma subida notável nos tempos de execução a partir de um determinado número de *threads*, mantendo-se este estável a partir das dez *threads* até às quarenta. A granularidade do paralelismo parece, aqui, mais adequada.

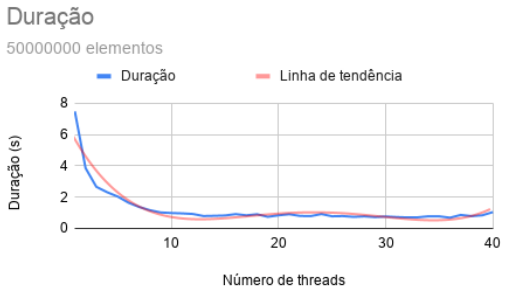
O ganho de desempenho pode ser visto no gráfico da Figura 9b. O algoritmo paralelo chega a ser dez vezes mais rápido que a versão sequencial, o que é o maior ganho registado até agora. No entanto, é apenas um quarto do calculado pela Lei de Amdahl.

A fração da execução sequencial encontra-se entre os vinte e os trinta por cento, por observação do gráfico da Figura 9c.

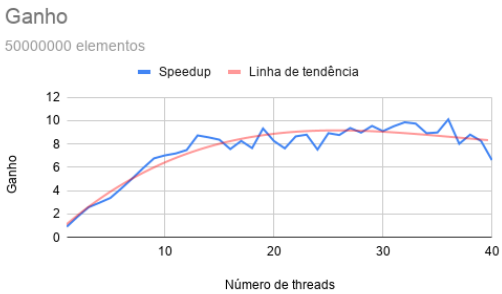
Na Figura 10 está o gráfico que mostra a evolução do algoritmo em termos

```
Samples: 43K of event 'cycles', Event count (approx.): 29613665090
Overhead  Command    Shared Object      Symbol
  75,21%  quicksort  quicksort          [.] s_quicksort
   9,50%  quicksort  quicksort          [.] p_quicksort
   7,35%  quicksort  libc-2.31.so       [.] __random
   2,90%  quicksort  libgomp.so.1.0.0   [.] 0x000000000001c758
   1,94%  quicksort  libgomp.so.1.0.0   [.] 0x000000000001c5c0
   0,98%  quicksort  libc-2.31.so       [.] __random_r
   0,88%  quicksort  quicksort          [.] main
   0,34%  quicksort  libc-2.31.so       [.] rand
```

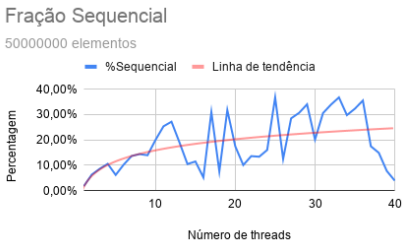
Figura 8: Perfil de execução do algoritmo com quebra de paralelismo em função do número de *threads*



(a) Duração em segundos



(b) Ganho



(c) Fração de tempo da execução sequencial

Figura 9: Escalabilidade forte do algoritmo com quebra de paralelismo aos 5000 elementos

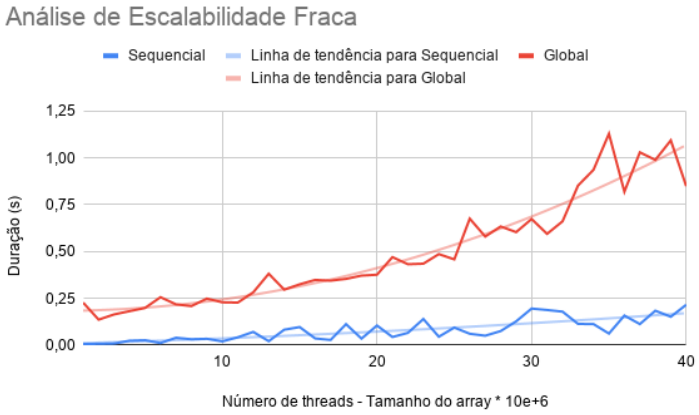


Figura 10: Escalabilidade fraca do algoritmo com quebra de paralelismo aos 5000 elementos

de escalabilidade fraca. Mais uma vez, o tempo de execução não se mantém constante, como seria desejável, e pelas razões indicadas anteriormente.

Por fim, a Figura 11 mostra o perfil de execução desta versão do algoritmo. Aqui, vemos que as execuções das secções sequencial e paralela são as mais próximas até agora, o que trouxe benefícios ao desempenho do algoritmo.

5.4 Limitações do desempenho

Até agora, ficou claro que o algoritmo desenvolvido não atinge o desempenho que foi calculado no início desta secção. No entanto, existem várias razões para tal, e muitas têm a ver com a natureza do algoritmo de ordenação em questão.

Em primeiro lugar, e como pudemos ver pelas análises de escalabilidade fraca, a duração da parte sequencial do algoritmo aumenta com o aumento do número de elementos a ordenar. Tal acontece porque a primeira divisão do conjunto de dados ocorre sequencialmente, sendo necessário percorrer todos os elementos com apenas uma *thread*. De seguida, a forma como o conjunto de dados é dividido não é necessariamente igual. Aliás, na maioria dos casos, será entregue a uma *thread* uma carga de trabalho diferente do que será entregue a outra, após uma iteração do algoritmo. Este desbalanceamento de carga penaliza o desempenho (uma possível solução para isto foi discutida na introdução do

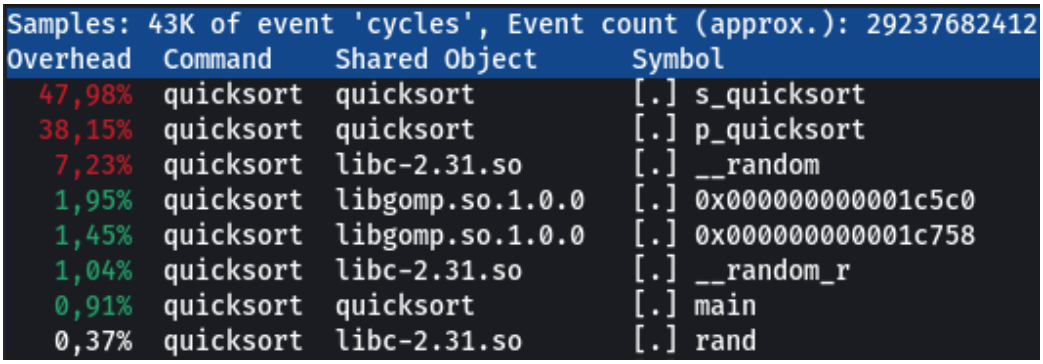


Figura 11: Perfil de execução do algoritmo com quebra de paralelismo aos 5000 elementos

algoritmo paralelo, na Secção 4). Outra causa para o desempenho obtido é o facto de, mesmo tendo disponíveis quarenta *threads*, não ser possível tirar partido de todas logo desde o início, uma vez que o algoritmo começa com um conjunto de dados, que é partido em dois, e assim sucessivamente. Neste caso, são necessárias seis iterações do algoritmo para que estejam quarenta *threads* em execução ($\log_2 40 \approx 5.3$). Por fim, existe toda a sobrecarga que é consequência da gestão de várias tarefas e *threads*, assim como limitações nos acessos à memória e à *cache*, uma vez que este algoritmo baseia-se fortemente na troca de posições de elementos em memória partilhada.

6 Conclusões

Após a comparação de três níveis distintos de granularidade do paralelismo, conclui-se que este último estudado é o que oferece melhor desempenho. Apesar de apenas chegar a um quarto daquele que é calculado usando a Lei de Amdahl, é claramente superior às outras alternativas.

Um paralelismo com granularidade muito fina, como aquele estudado em primeiro lugar (sem quebra de paralelismo), sofre pela sobrecarga que a manutenção de tarefas e *threads* traz, especialmente notável quando a carga de trabalho de cada *thread* é reduzida, o que acontece nas últimas iterações do algoritmo.

Um paralelismo com granularidade muito grossa, como aquele estudado

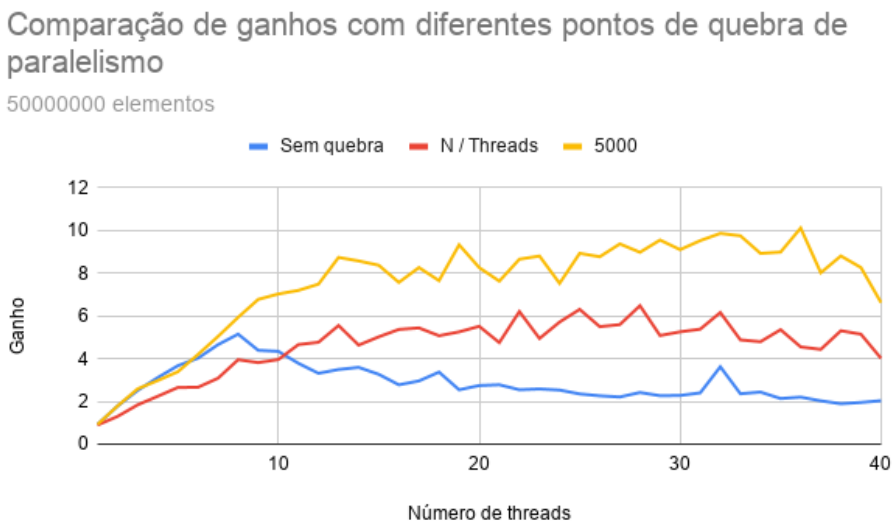


Figura 12: Comparação dos ganhos de desempenho entre as várias versões paralelas do algoritmo

quando a quebra do paralelismo era definida em função do número de *threads*, desperdiçava o potencial que este poderia trazer, uma vez que o algoritmo passava para a versão sequencial demasiado cedo (com demasiados elementos por ordenar).

O melhor caso acabou por se revelar como sendo aquele em que o algoritmo explorava o paralelismo com granularidade fina, uma vez que passava à versão sequencial quando se tratava de um número bastante reduzido de elementos comparativamente ao número total, mas não exageradamente. Assim, encontrou-se um equilíbrio entre o benefício de tratar os dados em paralelo, e a sobrecarga consequente da gestão de várias tarefas e *threads*.

Na Figura 12 encontra-se um resumo destes dados, em que se comparam os ganhos registados para as três versões estudadas. A azul encontra-se representado a curva de ganho da versão sem quebra de paralelismo. A curva a vermelho representa a versão cuja quebra era calculada como sendo o número de elementos a dividir pelo número de *threads* em execução. Por fim, a curva a amarelo representa o ganho da versão cuja quebra de paralelismo acontecia quando cada *thread* tinha cinco mil elementos com que trabalhar. Fica, então, clara a forma como estas três versões se comparam.

Referências

[1] https://en.wikipedia.org/wiki/Quicksort