

PARALELIZAÇÃO DO QUICKSORT USANDO C+OPENMP

Silvio do Lago Pereira¹, Luiz Tsutomu Akamine², Lucio Nunes de Lira³

¹ Prof. Dr. do Departamento de Tecnologia da Informação – FATEC-SP

² Prof. Esp. do Departamento de Tecnologia da Informação – FATEC-SP

³ Aux. de Doc. do Departamento de Tecnologia da Informação – FATEC-SP

slago@fatecsp.br, lakamine@fatecsp.br, lucio.nunes@fatecsp.br

Resumo

Quicksort é um dos algoritmos de ordenação mais eficientes que existem. Por ser baseado na estratégia de *divisão e conquista*, ele pode ser facilmente paralelizado; porém, suas versões paralelas simples raramente têm bom desempenho. Em geral, para se beneficiar de paralelismo, esse algoritmo requer grandes modificações, o que torna seu código excessivamente complexo e dificulta seu uso. Assim, o objetivo deste trabalho é apresentar uma versão paralela simples e eficiente desse algoritmo. Para avaliar a eficiência dessa versão proposta, foram comparados os tempos de execução de várias versões seriais e paralelas do *Quicksort*, para ordenar vetores aleatórios de vários tamanhos. Os resultados comparativos mostraram que a versão paralela proposta é, de fato, bastante eficiente.

1. Introdução

Dado um vetor v , com n itens em ordem arbitrária, a *ordenação* é uma operação que permuta os itens de v , de forma que se tenha $v_0 \leq v_1 \leq v_2 \leq \dots \leq v_{n-1}$. Essa operação é fundamental em diversas aplicações práticas em computação e há vários algoritmos que a implementam de modo eficiente. Entre eles, *Quicksort* é um dos mais rápidos [1].

Há várias versões paralelas de *Quicksort* na literatura; porém, em geral, elas são pouco eficientes ou, então, são muito complexas e exigem máquinas mais potentes [2-4].

Neste artigo, o objetivo é apresentar uma versão paralela de *Quicksort* que seja simples, que possa ser executada em um *notebook* com poucos processadores e que, apesar disso, apresente um bom desempenho.

O restante deste artigo está organizado do seguinte modo: a Seção 2 introduz fundamentos de programação paralela em C e *OpenMP*; a Seção 3 descreve as versões seriais e paralelas do *Quicksort* propostas neste trabalho; a Seção 4 descreve o método para escolha do *cutoff* que maximiza o *speedup* das versões paralelas do *Quicksort*; a Seção 5 apresenta e discute os resultados empíricos obtidos com os algoritmos implementados; e, finalmente, a Seção 6 apresenta as conclusões finais do trabalho.

2. Programação Paralela em C+OpenMP

OpenMP (*Open Multi-Processing*) é uma interface de programação que suporta processamento paralelo, com memória compartilhada, em múltiplas plataformas [5,6]. Em C [7], essa interface está disponível no arquivo `omp.h` e pode ser acessada por meio de diretivas `#pragma omp`, que são inseridas no código-fonte dos programas. Tais diretivas permitem ao programador informar ao compilador que partes do código-fonte devem ser executadas em paralelo, bem como de que forma essas partes devem ser distribuídas entre os *threads* disponíveis.

Um *thread* é uma entidade capaz de executar um trecho de código, de forma independente. Ao executar um

programa, o sistema operacional cria um processo correspondente e aloca alguns recursos necessários para sua execução (e.g., espaço de memória e registradores). Se múltiplos *threads* devem colaborar para executar um processo, então eles precisam compartilhar tais recursos. Além dos recursos compartilhados, cada *thread* precisa ter seu próprio contador de programa (PC), que indica a instrução a ser executada a cada instante, bem como uma área de memória privada para salvar seus dados específicos, incluindo variáveis locais, registradores e pilha.

A interface `omp.h` possui diversos recursos úteis para paralelizar vários tipos de algoritmos. Nas duas próximas subseções, são apresentados apenas os recursos usados na paralelização do *Quicksort*, proposta neste trabalho.

2.1. Definição de Região Paralela

Uma *região paralela* é um bloco de código que deve ser executado por vários *threads*, simultaneamente.

Por exemplo, no programa da Figura 1, a diretiva `#pragma omp parallel` cria um grupo de *threads* para executar o bloco de instruções subsequente no código, a função `omp_get_num_threads()` informa o tamanho do grupo criado e a função `omp_get_thread_num()` informa o número do *thread* que está executando o bloco.

```
1 #include <stdio.h>
2 #include <omp.h>
3 int main(void) {
4     int shared = 1;
5     #pragma omp parallel
6     {
7         int group = omp_get_num_threads();
8         int thread = omp_get_thread_num();
9         printf("%d/%d: Hello world!\n", thread, group);
10        shared *= 2;
11    }
12    printf("shared = %d\n", shared);
13    return 0;
14 }
```

Figura 1 – “Hello world” paralelizado com *OpenMP*.

A execução do programa da Figura 1 produz a saída na Figura 2 (exceto pela variação causada pela serialização das saídas paralelas, para exibição em vídeo).

```
2/4: Hello world!
0/4: Hello world!
1/4: Hello world!
3/4: Hello world!
shared = 16
```

Figura 2 – Resultado da execução do “Hello world” paralelo.

Note que apenas a variável `shared`, declarada fora da região paralela, é compartilhada pelos *threads*. Como a máquina usada na execução do programa possui 2 núcleos, cada um deles composto por 2 processadores lógicos, o grupo criado contém 4 *threads*. Ademais, uma barreira é inserida automaticamente no final da região paralela, de modo que a execução da primeira instrução fora dessa região (aquela que exibe o valor de `shared`) deve aguardar até que a execução de todos os *threads* do grupo termine.

2.2. Definição de Tarefas Paralelas

Uma *tarefa* é uma instância específica de um código executável, juntamente com seu ambiente de dados. Uma tarefa pode ser executada por qualquer *thread* do grupo responsável pela execução da região paralela na qual ela é disparada, em paralelo com esta região. A execução de uma tarefa pode ser imediata (se houver no grupo algum *thread* ocioso) ou pode ser adiada até um momento posterior (neste caso, a tarefa deve aguardar numa fila de escalonamento de tarefas, gerenciada pelo sistema).

A diretiva `#pragma omp single nowait` indica que apenas um *thread* do grupo criado pela diretiva `#pragma omp parallel` (chamado *master thread*) deve executar a região paralela e que os demais *threads* do grupo não precisam esperar o término da execução deste *thread*.

A diretiva `#pragma omp task`, usada dentro de uma região paralela, indica que o bloco de código subsequente é uma tarefa. Ela é útil no caso de *paralelismo irregular*, onde tarefas têm diferentes tempos de execução, bem como no caso de *paralelismo aninhado*, onde tarefas disparam recursivamente novas tarefas. Em programas com paralelismo aninhado, a chamada `omp_set_nested(1)`, feita por `main()`, indica que as tarefas podem ser aninhadas.

A Figura 3 mostra um programa com paralelismo irregular, que calcula o *Fibonacci* de uma série de números, e sua saída é exibida na Figura 4. Note que, como executar `f(35)` demora muito, o *thread* 0 recebe apenas essa tarefa.

```
1 #include <stdio.h>
2 #include <omp.h>
3 int f(int n) { return n<3 ? 1 : f(n-1) + f(n-2); }
4 int main(void) {
5     int v[6] = {35,25,20,27,22,21};
6     #pragma omp parallel
7     #pragma omp single nowait
8     {
9         for(int i=0; i<6; i++)
10             #pragma omp task
11             {
12                 int t = omp_get_thread_num();
13                 int n = v[i];
14                 double s = omp_get_wtime(); // start
15                 int r = f(n);
16                 double e = omp_get_wtime(); // end
17                 printf("[%d, %fs] f(%d)=%d\n",t,e-s,n,r);
18             }
19     }
20     return 0;
21 }
```

Figura 3 – Paralelização com tarefas para calcular *Fibonacci*.

```
[2, 0.000055s] f(20)=6765
[1, 0.000612s] f(25)=75025
[2, 0.000188s] f(22)=17711
[3, 0.001544s] f(27)=196418
[1, 0.000100s] f(21)=10946
[0, 0.038156s] f(35)=9227465
```

Figura 4 – Distribuição irregular de tarefas entre os *threads*.

2.3. Paralelização, Speedup e Speeddown

A paralelização de um algoritmo visa reduzir seu tempo de execução. Portanto, comparar os tempos de execução de suas versões serial e paralela é essencial para analisar a *eficiência da paralelização* (i.e., o desempenho da versão paralela em relação àquele da versão serial).

O desempenho de um algoritmo paralelo é dado por uma métrica chamada *speedup*. Sejam T_s e T_p os tempos de execução serial e paralela, respectivamente, de um algoritmo, numa máquina com p processadores. Então, o

speedup obtido com a paralelização é T_s / T_p . Assim, por exemplo, um *speedup* igual a 2 indica que a versão paralela executa duas vezes mais rapidamente que a versão serial; enquanto um *speedup* igual a 1 indica que ambas as versões têm o mesmo desempenho. Alternativamente, um *speedup* inferior a 1 pode ser chamado de *speeddown*.

O *speeddown* é causado pela proliferação excessiva de tarefas. Neste caso, o sistema tem uma sobrecarga para gerenciar tarefas (i.e., criar, sincronizar e destruir), que consome mais tempo que aquele necessário para executá-las.

2.4. Granularidade, workload, overhead e Cutoff

Granularidade refere-se ao tamanho das tarefas criadas durante a execução de um algoritmo paralelo, variando de fina (*fine-grain*) até grossa (*coarse-grain*).

A granularidade fina facilita o balanceamento da carga de trabalho (*workload balance*), pois permite distribuir as tarefas entre os *threads* de forma equilibrada (Figura 5a); porém, ela aumenta o tempo necessário para gerenciar as tarefas (*task overhead*). Inversamente, a granularidade grossa diminui o *task overhead*, mas dificulta o *workload balance* (Figura 5b). Assim, a eficiência do algoritmo com paralelismo irregular depende muito da granularidade.

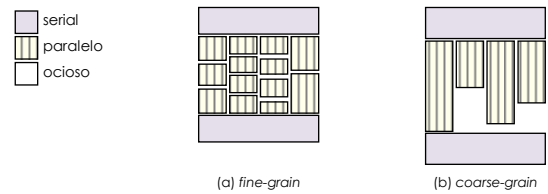


Figura 5 – Granularidade fina versus granularidade grossa.

Em algoritmos com paralelismo irregular e aninhado, um mecanismo de *cutoff* pode ser usado para controlar a granularidade (definindo um tamanho *mínimo* de tarefa, a partir do qual novas tarefas não possam mais ser disparadas recursivamente). Na prática, porém, é difícil definir precisamente um *cutoff* que garanta o perfeito equilíbrio entre *task overhead* e *workload balance*.

3. O Algoritmo Quicksort

Quicksort é um algoritmo de ordenação [1], baseado na estratégia de divisão e conquista. A divisão é feita pela operação de *partição* e a conquista é feita com *recursão*.

Dado um vetor $v[start .. end]$, a operação de *partição* escolhe um item *pivot*, permuta os itens de v , e devolve um índice *cut* que divide v em duas partes, $v[start .. cut]$ e $v[cut+1 .. end]$, tais que $x \leq pivot$, para todo $x \in v[start .. cut]$, e $x \geq pivot$, para todo $x \in v[cut+1 .. end]$. A implementação dessa operação em C é apresentada na Figura 6.

```
1 int partition(int v[], int start, int end) {
2     int pivot = v[(start + end)/2];
3     int cut = end+1;
4     start--;
5     while( start < cut ) {
6         do cut--; while( v[cut] > pivot );
7         do start++; while( v[start] < pivot );
8         if( start < cut ) swap(v, start, cut);
9     }
10    return cut;
11 }
```

Figura 6 – Operação de partição, usada pelo *Quicksort*.

Após a partição de v , a ordenação de cada uma de suas partes é um problema *independente*. Então, para ordenar v completamente, basta ordenar recursivamente suas partes.

3.1. Quicksort Padrão Serial

A versão serial do *Quicksort* padrão, implementada em C, é apresentada na Figura 7.

```
1 void ssqs(int v[], int start, int end) {
2     if( start >= end ) return;
3     int cut = partition(v, start, end);
4     ssqs(v, start, cut);
5     ssqs(v, cut+1, end);
6 }
7 void SSqs(int v[], int n) {
8     ssqs(v, 0, n-1);
9 }
```

Figura 7 – Serial Standard Quicksort.

A função *SSqs()* – *Serial Standard Quicksort* – é apenas um *wrapper* para a função *ssqs()*. O melhor caso para a função *ssqs()*, ocorre quando a operação *partition()* sempre divide o vetor em duas partes do mesmo tamanho. Neste caso, a complexidade de tempo da função *SSqs()* é $O(n \lg n)$, como mostra a Figura 8.

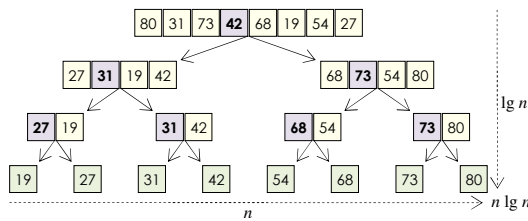


Figura 8 – No melhor caso, o algoritmo *Quicksort* é $O(n \lg n)$.

Por outro lado, o pior caso ocorre quando a operação *partition()* sempre divide o vetor em uma parte contendo apenas um item e outra parte contendo todos os demais itens. Neste caso, a complexidade de tempo da função *SSqs()* é $O(n^2)$, como mostra a Figura 9.

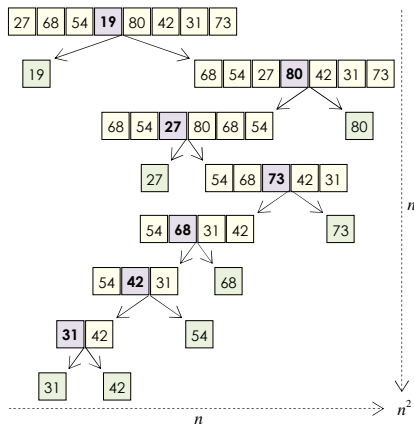


Figura 9 – No pior caso, o algoritmo *Quicksort* é $O(n^2)$.

No caso médio, com vetores aleatórios, a complexidade de tempo da função *SSqs()* é $O(n \lg n)$ [1].

3.2. Quicksort Padrão Paralelo

A versão paralela do *Quicksort* padrão, em C, é apresentada na Figura 10. Nesta implementação, a função *PSqs()* – *Parallel Standard Quicksort* – é apenas um *wrapper* para a função *psqs()*. Para controlar a granularidade das tarefas aninhadas, essa função recebe como parâmetro um valor de *cutoff*. Quando o tamanho de uma tarefa (i.e., o tamanho do vetor a ser ordenado pela tarefa) é igual ou inferior ao *cutoff*, a função deixa de disparar novas tarefas paralelas e resolve o problema usando diretamente a versão padrão serial (Figura 7).

```
1 void psqs(int v[], int start, int end, int cutoff) {
2     if( end-start+1 <= cutoff )
3         ssqs(v, start, end);
4     else {
5         int cut = partition(v, start, end);
6         #pragma omp task
7         psqs(v, start, cut, cutoff);
8         #pragma omp task
9         psqs(v, cut+1, end, cutoff);
10    }
11 }
12 void PSqs(int v[], int n, int cutoff) {
13     #pragma omp parallel
14     #pragma omp single nowait
15     psqs(v, 0, n-1, cutoff);
16 }
```

Figura 10 – Parallel Standard Quicksort.

A função *FPSqs()* – *Fine-grained Parallel Standard Quicksort* – na Figura 11, é um *wrapper* para a função *psqs()*, que define um *cutoff* igual a 1. Assim, as tarefas paralelas são recursivamente disparadas, até que os vetores a serem ordenados por elas tenham apenas um item.

```
1 void FPSqs(int v[], int n) {
2     #pragma omp parallel
3     #pragma omp single nowait
4     psqs(v, 0, n-1, 1);
5 }
```

Figura 11 – Fine-grained Parallel Standard Quicksort.

A função *CPSqs()* – *Coarse-grained Parallel Standard Quicksort* – na Figura 12, é um *wrapper* para a função *psqs()*, que define um *cutoff* igual a $n \gg 1$ (i.e., $n/2$). Assim, tarefas paralelas são recursivamente disparadas, até que os vetores a serem ordenados tenham no máximo a metade do tamanho do vetor original.

```
1 void CPSqs(int v[], int n) {
2     #pragma omp parallel
3     #pragma omp single nowait
4     psqs(v, 0, n-1, n>>1);
5 }
```

Figura 12 – Coarse-grained Parallel Standard Quicksort.

3.3. Quicksort Otimizado Serial

Um problema com o *Quicksort* padrão é que, no pior caso, ele cria uma pilha de tamanho proporcional a n , ou seja, tem complexidade de espaço $O(n)$. Então, para n muito grande, sua execução pode causar um erro de *stack overflow*. Uma forma de garantir complexidade de espaço $O(\lg n)$, no pior caso, consiste em particionar o vetor a ser ordenado, resolver a parte *menor* de forma recursiva e a outra de forma iterativa. Assim, como a parte menor pode ter no máximo $n/2$ itens, as chamadas recursivas podem atingir uma profundidade máxima da ordem de $\lg n$. A Figura 13 mostra a versão serial otimizada do *Quicksort*.

```
1 void soqs(int v[], int start, int end) {
2     while( start < end ) {
3         int cut = partition(v, start, end);
4         if( cut-start+1 <= end-cut ) {
5             soqs(v, start, cut);
6             start = cut + 1;
7         }
8         else {
9             soqs(v, cut+1, end);
10            end = cut;
11        }
12    }
13 }
14 void SOqs(int v[], int n) {
15     soqs(v, 0, n-1);
16 }
```

Figura 13 – Serial Optimized Quicksort.

3.4. Quicksort Otimizado Paralelo

A Figura 14 apresenta a versão paralela do *Quicksort* otimizado. A função `POqs()` – *Parallel Optimized Quicksort* – é um *wrapper* para a função `poqs()`. Analogamente à versão padrão, a versão otimizada também recebe como parâmetro um valor de *cutoff*. Quando o tamanho de uma tarefa é igual ou inferior ao *cutoff*, a função deixa de disparar novas tarefas paralelas e resolve o problema usando diretamente a versão otimizada serial (Figura 13).

```

1 void poqs(int v[], int start, int end, int cutoff) {
2     while( end-start+1 > cutoff ) {
3         int cut = partition(v, start, end);
4         if( cut-start+1 <= end-cut ) {
5             #pragma omp task
6             poqs(v, start, cut, cutoff);
7             start = cut + 1;
8         }
9         else {
10            #pragma omp task
11            poqs(v, cut+1, end, cutoff);
12            end = cut;
13        }
14    }
15    sqqs(v, start, end);
16 }
17 void POqs(int v[], int n, int cutoff) {
18     #pragma omp parallel
19     #pragma omp single nowait
20     poqs(v, 0, n-1, cutoff);
21 }

```

Figura 14 – Parallel Optimized Quicksort.

A função `FPOqs()` – *Fine-grained Parallel Optimized Quicksort* – na Figura 15, é um *wrapper* para a função `poqs()`, que define um *cutoff* igual a 1.

```

1 void FPOqs(int v[], int n) {
2     #pragma omp parallel
3     #pragma omp single nowait
4     poqs(v, 0, n-1, 1);
5 }

```

Figura 15 – Fine-grained Parallel Optimized Quicksort.

A função `CPOqs()` – *Coarse-grained Parallel Optimized Quicksort* – na Figura 16, é um *wrapper* para a função `poqs()`, que define um *cutoff* igual a $n \gg 1$.

```

1 void CPOqs(int v[], int n) {
2     #pragma omp parallel
3     #pragma omp single nowait
4     poqs(v, 0, n-1, n>>1);
5 }

```

Figura 16 – Coarse-grained Parallel Optimized Quicksort.

3.5. Quicksort da Biblioteca Padrão de C

Sendo um dos algoritmos de ordenação mais eficientes que existem, o *Quicksort* faz parte da biblioteca padrão em C (`stdlib.h`). Ele é implementado de forma serial pela função `qsort()`. Por ser uma função de biblioteca, `qsort()` foi implementada para ordenar vetores de qualquer tipo de dados, em ordem crescente ou decrescente.

Para ter a generalidade necessária, a função `qsort()` precisa receber 4 parâmetros: um ponteiro para o vetor a ser ordenado, o tamanho desse vetor, o tamanho dos itens desse vetor (em *bytes*) e um ponteiro para a função que será usada para comparar os itens durante a ordenação. Tal função deve devolver um valor *nulo*, quando os itens comparados forem iguais; *positivo*, quando o primeiro item for maior que o segundo; ou *negativo*, quando o primeiro item for menor que o segundo.

A função `CSLqs()` – *C Standard Library Quicksort* – na Figura 17, é um *wrapper* para a função `qsort()` que ordena um vetor de inteiros, em ordem crescente. Essa função foi usada para obtenção de resultados empíricos que são apresentados na Subseção 5.4.

```

1 int ascending(const void *a, const void *b) {
2     int x = *((int *) a);
3     int y = *((int *) b);
4     return (x>y) - (x<y);
5 }
6 void CSLqs(int v[], int n) {
7     qsort(v, n, sizeof(int), ascending);
8 }

```

Figura 17 – C Standard Library Quicksort.

4. Escolha do Melhor Cutoff

Como explicado na Subseção 2.4, a eficiência de um algoritmo com paralelismo irregular e aninhado depende muito do valor de *cutoff* escolhido, pois este definirá a granularidade das tarefas e, consequentemente, como será o equilíbrio entre *task overhead* e *workload balance*.

Como a função `partition()` sempre divide o vetor a ser ordenado em duas partes, o *cutoff* para o *Quicksort* pode ser definido como $n/2^c$. Então, para escolher o melhor *cutoff*, basta escolher o valor de c que maximiza o *speedup*.

Para escolher esse valor *empiricamente*, geramos 300 permutações aleatórias da sequência $\langle 1, 2, 3, \dots, 2^{16} \rangle$ e calculamos o *speedup* médio obtido por `POqs()`, em relação a `SSqs()`, para ordenar todas elas, para cada c entre 1 e 16. Depois, analisando os dados obtidos, exibidos na Figura 18, concluímos que o *speedup* é máximo para $c = 4$.

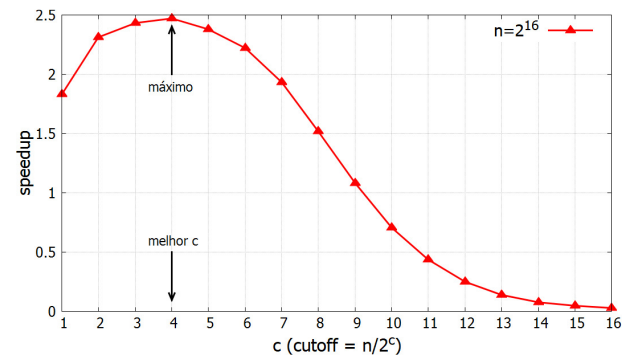


Figura 18 – Melhor *cutoff* para um vetor de tamanho $n = 2^{16}$.

Repetindo o experimento para vetores contendo permutações aleatórias da sequência $\langle 1, 2, 3, \dots, n \rangle$, para $n = 2^k$ e $k \in [16..23]$, observamos que o valor de c que maximiza o *speedup* aumenta uma unidade, à medida que o valor de n é *quadruplicado* (os dados obtidos estão na Figura 19). Então, definimos o melhor valor de c como $\lfloor \log_4 n - 4 \rfloor$.

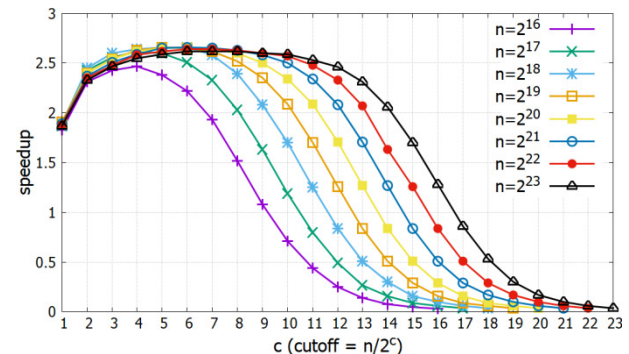


Figura 19 – Melhores valores de c para vetores grandes.

Finalmente, repetindo o experimento para vetores contendo permutações aleatórias da sequência $\langle 1, 2, 3, \dots, n \rangle$, para $n=2^k$ e $k \in [10..15]$, observamos que, para vetores com 2^{12} itens, o *speedup* é muito baixo e, para vetores menores ainda, temos *speeddown*. Então, ajustamos a fórmula que dá o melhor valor de c para $\max(\lfloor \log_4 n - 4 \rfloor, 1)$. Os dados obtidos são apresentados na Figura 20.

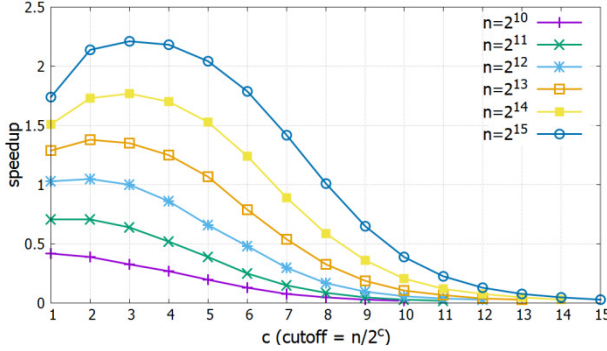


Figura 20 – Melhores valores de c para vetores pequenos.

Com base na fórmula obtida, criamos a função que escolhe o melhor *cutoff*, para cada n , exibida na Figura 21.

```
1 double lg(double x, double b) { return log(x)/log(b); }
2 int best_c(int n) { return max(floor(lg(n,4)-4),1); }
3 int best_cutoff(int n) { return max(n>>best_c(n),1); }
```

Figura 21 – *Best-cutoff* em função do tamanho do vetor.

A função *BPSqs()* – *Best-grained Parallel Standard Quicksort* – na Figura 22, é um *wrapper* para a função *psqs()*, que define um *cutoff* que maximiza o *speedup*.

```
1 void BPSqs(int v[], int n) {
2     #pragma omp parallel
3     #pragma omp single nowait
4     psqs(v, 0, n-1, best_cutoff(n));
5 }
```

Figura 22 – *Best-grained Parallel Standard Quicksort*.

A função *BPOqs()* – *Best-grained Parallel Optimized Quicksort* – na Figura 23, é um *wrapper* para a função *poqs()*, que define um *cutoff* que maximiza o *speedup*.

```
1 void BPOqs(int v[], int n) {
2     #pragma omp parallel
3     #pragma omp single nowait
4     poqs(v, 0, n-1, best_cutoff(n));
5 }
```

Figura 23 – *Best-grained Parallel Optimized Quicksort*.

5. Resultados Empíricos

As diversas versões seriais e paralelas do *Quicksort*, propostas neste artigo, foram implementadas com o compilador *Pelles C*, versão 8.00.60, 32 bits, rodando em uma máquina *Intel(R) Core(TM) i7-5500U @ 2.40GHz*, com 2 núcleos físicos (4 processadores lógicos) e 4GB de memória RAM *DDR3*, no sistema operacional *Windows 10*.

Os experimentos foram feitos com vetores contendo permutações aleatórias da sequência $\langle 1, 2, 3, \dots, n \rangle$, gerados pela função na Figura 24. Os tempos reportados são a média dos tempos medidos para 300 permutações, para cada tamanho de vetor, pela função definida na Figura 25.

```
1 void permutation(int v[], int n) {
2     for(int i=0; i<n; i++) v[i] = i+1;
3     while( n-- ) swap(v, n, rand()%n+1);
4 }
```

Figura 24 – Geração de permutações aleatórias.

```
1 double etime(void *f, int *v, int n, int *w, int cutoff) {
2     memcpy(w, v, n*sizeof(int));
3     double start = omp_get_wtime();
4     if( cutoff <= 0 ) ((void (*)(int *, int))f)(w, n);
5     else ((void (*)(int *, int, int))f)(w, n, cutoff);
6     return omp_get_wtime() - start;
7 }
```

Figura 25 – Tempo de execução de uma função de ordenação.

Note que *etime()* não altera v , permitindo que os mesmos vetores sejam usados pelos algoritmos comparados.

5.1. Influência da Otimização

Os tempos de execução das versões seriais, padrão e otimizada, são dados na Figura 26. Ao contrário do esperado, a versão otimizada (em relação a espaço) consome mais tempo que a versão padrão (embora a diferença seja em *ms*). Esse tempo adicional pode ser devido ao *if*, em *SOqs()*, que decide que parte do vetor sendo ordenado será tratada com recursão ou com iteração.

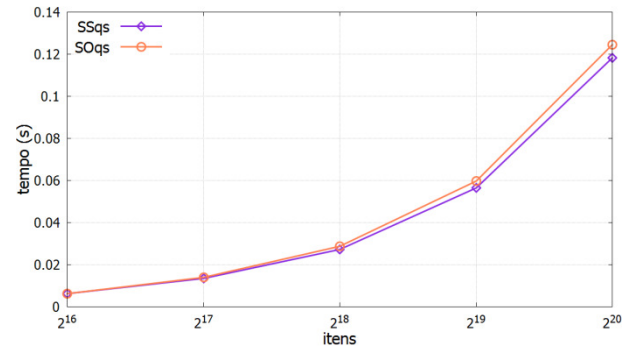


Figura 26 – Tempos das versões padrão e otimizada.

5.2. Influência da Granularidade Fina

Os tempos de execução das versões paralelas, padrão e otimizada, com granularidade fina, são dados na Figura 27. Apesar de ambas as versões exibirem *speeddown*, curiosamente, a versão paralela otimizada é cerca de 2 vezes mais rápida que a versão paralela padrão. Isso pode ser devido ao fato de a versão otimizada disparar cerca de metade das tarefas que são disparadas pela versão padrão, pois a outra metade ela trata de forma iterativa.

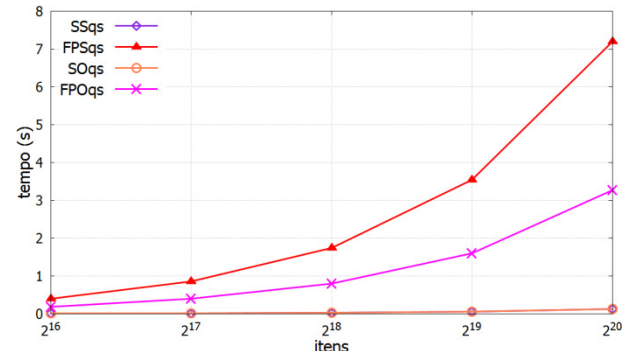


Figura 27 – *Speeddown* devido à granularidade fina.

5.2. Influência da Granularidade Grossa

Os tempos de execução das versões paralelas, padrão e otimizada, com granularidade grossa, são dados na Figura 28. Ambas as versões exibem aproximadamente o mesmo *speedup*. Isso pode ser devido ao fato de o *cutoff* ser muito alto ($= n/2$), impedindo que a versão padrão dispare muito mais tarefas que a versão otimizada.

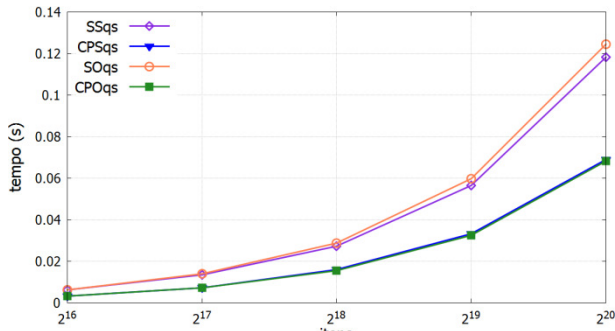


Figura 28 – Speedup devido à granularidade grossa.

5.3. Influência da Melhor Granularidade

Os tempos de execução das versões paralelas, padrão e otimizada, com a melhor granularidade, são dados na Figura 29. Com a escolha do melhor *cutoff*, o *speedup* médio para a versão padrão foi 2.1, enquanto para a otimizada foi 2.5. Considerando que a máquina usada nos experimentos tem apenas dois núcleos físicos, o *speedup* obtido com a versão otimizada foi bastante alto.

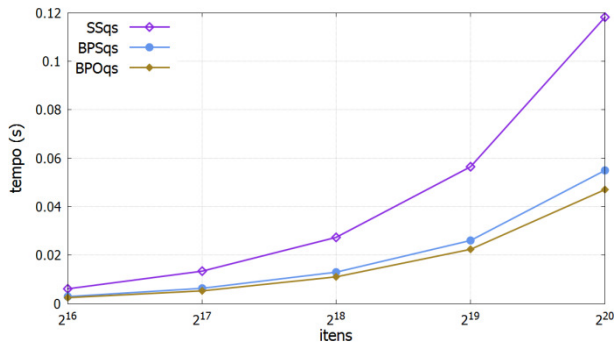


Figura 29 – A melhor granularidade aumenta o *speedup*.

Para evidenciar a diferença de desempenho entre todas as versões de *Quicksort* propostas neste artigo, todos os *speedups* são apresentados na Figura 30. Como pode ser observado, BPOqs () é a função com melhor desempenho.

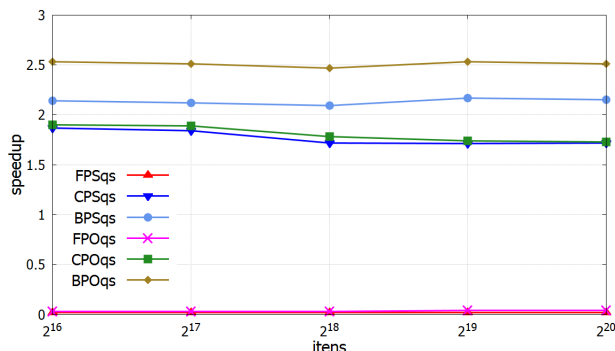


Figura 30 – Comparação geral das versões do *Quicksort*.

5.4. Função de Ordenação da Biblioteca de C

A comparação entre BPOqs () e CSLqs (), a função de ordenação da biblioteca padrão de C, é apresentada na Figura 31. Embora essa comparação não seja muito justa, pois a função *qsort* () usada em CSLqs () é *serial* e mais *genérica*, ela mostra que o uso da versão paralela proposta nesse artigo pode ser muito vantajoso. De fato, a análise dos tempos de execução na Figura 31 mostra que a função BPOqs () é cerca de 8 vezes mais rápida que CSLqs ().

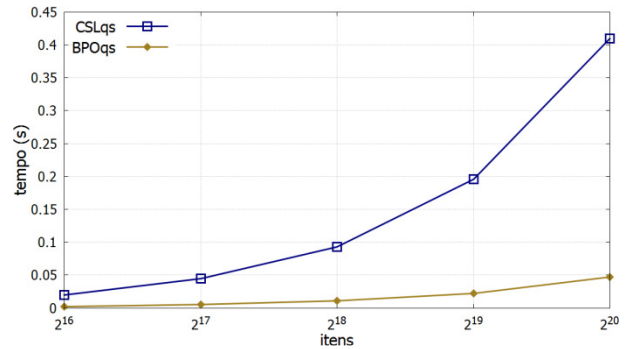


Figura 31 – Vantagem sobre a biblioteca padrão de C.

6. Conclusões

Neste artigo, mostramos como o algoritmo *Quicksort* pode ser paralelizado, usando recursos disponíveis na biblioteca `omp.h` do compilador *Pelles C*. Como pôde ser constatado, a paralelização de um algoritmo com essa biblioteca é relativamente simples. O que não é tão simples é a obtenção de uma versão paralela que seja eficiente.

Várias versões seriais e paralelas do *Quicksort* foram implementadas usando *C* e *OpenMP*. Mais precisamente, foram implementadas 2 versões seriais e 6 paralelas, com diferentes granularidades (*fine-grained*, *coarse-grained* e *best-grained*)*. A comparação empírica entre os tempos de execução dos algoritmos mostrou que as versões *best-grained*, tanto a versão padrão quanto a versão otimizada com relação ao uso de espaço de memória, foram aquelas que tiveram melhor desempenho (i.e., maiores *speedups*). Portanto, acreditamos que a contribuição mais importante desse trabalho foi o estudo feito para determinação do *cutoff* que maximiza o *speedup* das versões paralelas.

Em trabalho futuro, pretendemos estender o estudo feito para escolha do melhor *cutoff* considerando vetores cujos tamanhos não sejam potências de 2. Também pretendemos investigar como a fórmula proposta para definir o melhor *cutoff* (i.e., $\max(n/2^{\max(\lfloor \log_4 n - 4, 1 \rfloor)}, 1)$) é afetada pelo número de processadores existentes na máquina usada para execução dos algoritmos implementados.

Referências Bibliográficas

- [1] T. H. Cormen et al. **Introduction to Algorithms**, 3rd Edition, MIT Press, Cambridge, 2010.
- [2] H. Shi; J. Schaeffer. **Parallel Sorting by Regular Sampling**. Journal of Parallel and Distributed Computing, v. 14:4, p. 361-372, 1992.
- [3] P. Srivastava. **Hyper Quick Sort (Parallel Quicksort)**, The State University of New York, 2014.
- [4] A. Maus. **A Full Parallel Quicksort Algorithm for Multicore Processors**, University of Oslo, 2015.
- [5] R.Chandra et al. **Parallel programming in OpenMP**. Morgan Kaufmann, CA, USA, 2001.
- [6] B. Chapman et al. **Using OpenMP: Portable Shared Memory Parallel Programming**. MIT Press, 2008.
- [7] B. W. Kernighan; D. M. Ritchie. **The C Programming Language**, 2nd Edition, Englewood Cliffs, Prentice Hall, New Jersey, 1988.

* Todos os algoritmos implementados estão disponíveis em www.ime.usp.br/~slago/parallel_quicksort.zip.