Otimização com *threads* do desempenho de algoritmos de ordenação em máquinas com processamento *multicore*

Carlos E. Buarque C. Pimentel, Arthur F. M. de Sousa, Carlos H. Maciel S. Timóteo e Karina R. Pereira
Centro de Informática
Universidade Federal de Pernambuco
Recife, PE, Brasil
Email: {cebcp, afms, chmst, krp}@cin.ufpe.br

***Resumo*—Este artigo propõe novas implementações de alguns dos algoritmos de ordenação mais utilizados para obter maior proveito do poder de processamento de máquinas *multicore* na solução de problemas de ordenação. Portanto, algoritmos de ordenação de natureza de dividir para conquistar foram implementados para fazer o uso explícito de *threads*. Os resultados obtidos mostraram que, na maioria dos casos, o uso de *threads* nos algoritmos de ordenação analisados surte um efeito positivo, fazendo com que eles tenham um melhor desempenho em uma máquina com um processador de múltiplos núcleos.**

Além disso, segundo os modelos em GSPN, o desempenho do algoritmo é limitado, dessa forma, o cenário com um processador de 8 *threads* é um resultado bastante viável.

I. Introdução

O padrão para os computadores atuais, mesmo os pessoais, são os processadores de múltiplos núcleos. Conhecidos popularmente como processadores *multicore*, eles foram a forma que a indústria de componentes de *hardware* encontrou para superar os limites físicos dos semicondutores e continuar a evolução do poder de processamento dos computadores. Com os processadores *multicore* é possível realizar um processamento paralelo, uma vez que os núcleos que os compõem são independentes.

Com os processadores de único núcleo, um *clock* maior ou um maior número de transistores já eram o suficiente para que os *softwares* passassem a ter um desempenho melhor em termos de tempo de execução. Entretanto, se o número de núcleos de um processador dobrar ou mesmo quadruplicar, isso não implicará em um desempenho duas ou quatro vezes mais rápido e pode implicar até em nenhuma melhora se o algoritmo não for otimizado para esse tipo de processamento.

Os sistemas operacionais mais modernos procuram dividir, de forma transparente e autônoma, a carga de processamento dos *softwares* em execução entre os diversos núcleos de um processador *multicore* de um computador. Entretanto, ganhos mais significativos só são obtidos quando, ao escrever um *software*, o desenvolvedor procura explicitamente usar técnicas que explorem esse tipo de processamento como, por exemplo, *threads*.

Uma classe especial de algoritmos que pode se beneficiar bastante desse novo paradigma de processamento são os algoritmos de ordenação. Como algoritmos de ordenação eficientes são importantes para a construção de uma série de *softwares*, como os que realizam buscas, por exemplo, eles sempre atraíram a atenção da comunidade científica de computação. Apesar da ideia por trás dos algoritmos de ordenação ser simples, a complexidade de implementá-los de forma eficiente faz com que muitos estudos sejam conduzidos nessa área há vários anos.

Neste artigo, novas implementações de alguns dos algoritmos de ordenação mais utilizados são apresentadas. Essas novas implementações fazem o uso explícito de *threads* com a finalidade de obter um melhor desempenho na execução desses algoritmos em máquinas com múltiplos núcleos de processamento quando comparadas com as implementações comuns, ou seja, sem *threads*, nas mesmas máquinas. Além disso, é realizada uma avaliação do desempenho dos algoritmos através de um modelo em redes de petri estocásticas generalizadas (GSPN).

As características dos algoritmos e a abordagem para a otimização através do uso de *threads* serão introduzidas na segunda seção do artigo. A quarta seção mostra a metodologia adotada para condução do estudo empírico realizado a partir da proposta apresentada, bem como as limitações e as ameaças à validade desse estudo. Na quinta, os resultados das novas implementações serão apresentados. Em seguida, são apresentados os modelos dos algoritmos, os resultados da avaliação de desempenho e uma previsão desempenho dos algoritmos caso fossem utilizadas 8, 16, 32 e 64 *threads*. Por último, os resultados obtidos são comentados e direções para trabalhos futuros são apontadas.

II. Algoritmos de ordenação utilizando *threads*

Desde os primórdios da computação, os algoritmos de ordenação vêm sendo estudados e desenvolvidos, sendo o estudo realizado acerca do *Bubble sort* em 1956 [1] um exemplo. Apesar de muitos já considerarem o problema da ordenação como resolvido, até hoje, diversas novas abordagens são apresentadas incluindo as que utilizam a técnica de dividir para conquistar. A partir da técnica de dividir para

conquistar, dois dos mais populares algoritmos de ordenação foram desenvolvidos: o *Quicksort* e o *Merge sort*.

Devido a popularidade e ao fato de serem algoritmos do tipo dividir para conquistar, o que possibilita que as subdivisões possam ser processadas por *threads*, o *Quicksort* [2] e o *Merge sort* [3] foram os algoritmos escolhidos para utilização neste artigo. A seguir, o funcionamento de cada um deles e detalhado.

A. Quicksort

Esse algoritmo escolhe um dos elementos de uma coleção de dados, o qual é chamado de pivô. A partir disso, todos os outros elementos da coleção serão comparados ao pivô e os menores ou iguais a eles passam a vir antes dele e os maiores vêm depois. Ao final desse processo, o pivô estará no lugar certo. O processo descrito até aqui vai então ser aplicado recursivamente tanto ao grupo de elementos que vem antes do pivô, bem como ao que vem depois [4], [5].

O que é relevante para a eficiência dele é a escolha do pivô. Quando se escolhe o menor ou o maior elemento tem-se o pior caso, cuja complexidade computacional é de $O(n^2)$, podendo gerar um estouro de pilha devido às inúmeras chamadas recursivas. O melhor caso é quando se escolhe o elemento mediano, criando uma divisão possivelmente simétrica, o que resulta em uma complexidade de $O(nlogn)$, igual ao caso médio.

Como o custo para encontrar o valor que gera o melhor caso é alto, geralmente utiliza-se a estratégia de usar o item de índice médio da coleção como pivô.

B. Merge sort

O *Merge sort*, [4], [5], divide a coleção de dados em duas coleções a partir da metade do tamanho da inicial. Isso é feito recursivamente em cada coleção resultante até que sejam obtidas coleções com uma quantidade mínima de elementos pré-determinada.

Uma vez que a divisão das coleções tenha sido finalizada, um algoritmo de ordenação simples é então utilizado nas coleções resultantes. Após cada parte estar ordenada, concatena-se uma a uma de forma que a ordenação seja mantida e até toda coleção inicial ordenada seja obtida.

Em todos os casos, a complexidade computacional do *Merge sort* é $O(nlogn)$.

C. Novas implementações

Para que os algoritmos de ordenação mostrados anteriormente possam aproveitar melhor o poder de processamento dos atuais processadores *multicore*, novas implementações foram desenvolvidas nas quais o conceito de *threads* foi explicitamente utilizado.

Threads são as menores unidades de processamento que um sistema operacional pode escalonar [6]. Geralmente elas estão contidas em um processo e compartilham muitos de seus recursos, como memória, por exemplo, e são executadas de forma concorrente. Apesar disso, existe uma sobrecarga associada à criação de cada *thread* e isso pode acabar degradando o desempenho de execução de um *software*.

Em máquinas com processadores de único núcleo, essa concorrência não é real. O que acontece é que o processador intercala a execução de diferentes *threads*, ao longo do tempo, de forma tão rápida que o usuário tem a impressão de que elas tão sendo executadas em paralelo. Já em máquinas de múltiplo núcleos, essa concorrência de fato existe com cada *thread* sendo executada por um núcleo.

Nas implementações clássicas do *Quicksort* e do *Merge sort*, a parte do algoritmo que realiza a ordenação é chamada recursivamente até que a coleção esteja ordenada. Já nas novas implementações, uma *thread* é criada para cada chamada recursiva do trecho do algoritmo responsável pela ordenação e o número de *threads* é limitado pela quantidade de núcleos do processador.

O fato desses algoritmos serem do tipo dividir para conquistar facilita o uso de *threads*, o que também contribuiu para a escolha deles para o desenvolvimento deste artigo.

A seguir, são mostrados os trechos de código comparando os dois tipos de implementação, em Java, tanto para o *Quicksort* quanto para o *Merge sort*.

Quicksort clássico

```
...
public static void quick(Comparable a[], int
    start, int end){
...
        if (start < j) {
                quick(a,start,j);
        }
        if (i <end) {
                quick(a,i,end);
        }
...
}
```

Quicksort com threads

```
...
if (Thread.activeCount() <= NUM_NUCLEOS) {
        QuickSortThread thread1 = null,
            thread2 = null;
        if (start < j) {
                thread1 = new QuickSortThread(
                    a, start, j);
                thread1.start();
        }
        if (i < end) {
                thread2 = new QuickSortThread(
                    a, i, end);
                thread2.start();
        }
        try {
                if (start < j) {
                        thread1.join();
                }
                if (i < end) {
                        thread2.join();
                }
        } catch (InterruptedException e) {
                e.printStackTrace();
        }
```

```
}else{
        if (start < j) {
                QuickSort.quick(a, start ,j);
        }
        if (i < end) {
                QuickSort.quick(a,i,end);
        }
}
...
```

Merge sort clássico

```
...
public static void sort(Comparable dados[]) {
...
        sort(a);
        sort(b);
        merge(a,b,dados);
}
...
```

Merge sort com threads

```
...
if (MergeSortThread.activeCount()<=
    NUM_NUCLEOS) {
        MergeSortThread thread1 = null,
            thread2 = null;
        thread1 = new MergeSortThread(a);
        thread1.start();
        thread2 = new MergeSortThread(b);
        thread2.start();
        try {
                thread1.join();
                thread2.join();
        } catch (InterruptedException e) {
                e.printStackTrace();
        }
}else{
        MergeSort.sort(a);
        MergeSort.sort(b);
}
MergeSortThreads.merge(a,b,dados);
...
```

III. Redes de Petri Estocásticas Generalizadas

GSPN é uma extensão do formalismo das redes de Petri, a qual generaliza o uso conjunto de transições temporizadas e imediatas para representação do mesmo problema. Com isso, aumenta-se o poder de expressão de uma rede de Petri, mantendo tanto a possibilidade de uma análise comportamental dos sistemas quanto uma análise sob a óptica da avaliação do desempenho, através de especicações temporais que permitem a criação de modelos probabilísticos [7]. Da mesma forma que uma rede de Petri não temporizada, uma GSPN consiste na definição de lugares, transições e arcos que os interligam. Todavia, nesse caso existem dois tipos de transições. Em GSPN, uma transição temporizada consiste em um retângulo não preenchido, enquanto a transição imediata continua sendo representada por um retângulo preenchido.

Observa-se que a GSPN conserva um aspecto não determinístico e consequentemente esse modelo de análise remete a representação de problemas reais com tal comportamento. Portanto, essa característica estocástica surge como uma alternativa para a representação de problemas reais nos quais uma abordagem determinística não mapeia seu comportamento.

IV. Metodologia

Esta pesquisa, cujo tema é otimização do desempenho de algoritmos de ordenação em máquinas com processamento *multicore*, propõe o uso de *threads* para obter maior proveito do poder de processamento de máquinas *multicore* na solução de problemas de ordenação. Portanto, a hipótese desta proposta é que determinar explicitamente o ponto de divisão de *threads* torna algoritmos de ordenação mais eficientes do que quando esta divisão é feita automaticamente pelo sistema operacional.

Para guiar e facilitar a realização do estudo, foram identificadas algumas variáveis e suas classificações:

- **Tempo** - variável que expressa o tempo de execução do algoritmo, sendo portanto a variável dependente do estudo, que foi obtida através de *timestamps* apurados antes e após a execução do algoritmo;
- **Tamanho da amostra** - variável moderadora do estudo e também um fator de teste, que indica o número de elementos do *array* a ser ordenado, a qual foram atribuídos os valores 10.000, 100.000 e 1.000.000;
- **Ordem inicial dos elementos** - variável moderadora e também um fator de teste, que indica a ordem inicial na qual os elementos do *array* a ser ordenado se encontra, no estudo foi atribuído a esta variável os valores crescente, decrescente e aleatório;
- **Quantidade de *threads*** - variável independente do estudo que expressa o número máximo de *threads* que será criada durante a execução do algoritmo. A esta variável foram atribuídos os valores 0 (nenhuma *thread*), 2 e 4;
- **Algoritmo** - variável moderadora do estudo e também um fator de teste, que indica qual algoritmo foi utilizado para ordenar o *array*;
- **Processos de SO e de Usuário, *hardware*** - variável de controle do estudo.

O estudo realizado está inserido num contexto que pode ser classificado como *offline*, pois os dados usados nas ordenações são fictícios (gerados).

Nesta pesquisa, tanto os códigos tradicionais dos algoritmos de ordenação quanto os novos com *threads* foram implementados em Java. Todos os experimentos foram realizados numa máquina composta de um processador Intel Core2Quad Q6600 com quatro núcleos (cada um com frequência de *clock* de 2,4 GHz) [1] e 8 GB de memória RAM, executando o sistema operacional Ubuntu 10.04 de 64 *bits*. Além disso, para garantir a validade do estudo, os *softwares* executados durante os experimentos foram limitados a: processos que são inicializados por padrão pelo sistema operacional, a IDE de programação Eclipse e a máquina virtual da linguagem Java, a JVM.

[1]Como o número de núcleos do processador é 4, o número limite de *threads* adotado para o estudo realizado aqui é também 4. Tal limite é para que, no máximo, cada núcleo execute uma única *thread*

Neste trabalho foram realizados 36 experimentos do tipo 1 fator com 2 tratamentos, com o fator sendo a variável independente, ou seja, a quantidade de *threads*, e com os tratamentos , por sua vez, sendo os valores que essa variável pode assumir, 0 e 2 ou 0 e 4 [2]. Do total dos experimentos, 18 foram feitos aplicando-se o algoritmo *Merge sort* e, a outra metade, o algoritmo *Quicksort*. Dentro desses 18 experimentos realizados para cada algoritmo, foram consideradas amostras ordenadas aleatoriamente, em ordem (ou crescente) e ordenadas decrescentemente.

Cada experimento foi executado 500 vezes e os seus resultados armazenados em arquivos. Com os resultados nos arquivos, gráficos foram gerados para efeito de comparação entre as implementações sem e com *threads*, depois, para dar significância estatística à análise sendo conduzida, testes de hipótese foram aplicados. Diante do número de dados colhidos para cada experimento e do desconhecimento da variância populacional, foi aplicado o teste-t para a inferência da diferença das médias de duas amostras. Para todos os testes realizados, foram considerados os dados resultantes dos experimentos que não usaram *threads* como amostra 1, o nível de significância, α, de 0,05, variâncias populacionais diferentes e as seguintes hipóteses:

$$H_0 : \mu_1 = \mu_2$$

$$H_a : \mu 1 > \mu_2.$$

Dessa forma, a *hipótese nula* diz que a diferença das médias entre os experimentos sem e com *threads* é igual a 0, ou seja, não há melhora nem piora em aplicar *threads*; enquanto a *hipótese alternativa* diz que a diferença das médias entre o tempo sem e com *threads* é positiva, ou seja, usar *threads* otimiza o processamento dos algoritmos. Para a realização do teste t, foi utilizado o MATLAB [8].

Após obter os resultados dos algoritmos modificados, foi realizada a análise exploratória dos dados para verificar que tipo de distribuição de probabilidade eles obedeciam. Em seguida, foi desenvolvido um modelo em GSPN para avaliar o desempenho dos algoritmos quanto às métricas de utilização dos recursos de processamento e *throughput* de dados ordenados por unidade de tempo.

Posteriormente à validação dos resultados obtidos pelo modelo através do teste de hipótese com os dados coletados no experimento anterior, foi possível realizar a previsão do desempenho do sistema para 8, 16, 32 e 64 *threads*.

A. Ameaças e limitações

A seguir são apresentadas possíveis ameças à validade do trabalho realizado:

- **Ameaças internas** - cada linguagem implementa as *threads* de uma maneira internamente, o que pode gerar mais *overhead*, ou menos, afetando o tempo de execução; e bibliotecas podem não funcionar como esperado.
- **Ameaças externas** - os benefícios esperados podem não ser os mesmos em máquinas de configurações diferentes; a escolha do número de *threads* pode influenciar nos resultados.
- **Ameaças de construção** - utilização errada das *threads*; implementação ineficiente do algoritmo de ordenação; e possíveis simplificações exageradas no experimento. Para mitigar estas ameaças, as implementações foram feitas sempre em pares de desenvolvedores.
- **Ameaças de conclusão** - fazer interpretações que não sejam estatisticamente significativas. Esta ameaça foi mitigada por meio da utilização do teste t.

Como limitação do estudo conduzido, podem ser destacadas as diversas simplificações feitas ao arranjo experimental e à análise estatística empregada nos resultados. Tais simplificações foram realizadas por questões de tempo impostas à proposta aqui desenvolvida.

V. Resultados das novas implementações

Nesta seção são apresentados os resultados dos experimentos, bem como suas respectivas análises. Nos gráficos e tabelas apresentados, a ordenação inical dos elementos é representada da seguintes forma: aleatória, A, crescente, C e decrescente, D.

Como resultado do uso do MATLAB para análise estatística entre os experimentos com 0 e os com 2 e 4 *threads*, tem-se o seguinte:

- ***h*** - que indica se a hipótese nula é válida (0) ou não (1);
- ***p*** - indica o *p-value*, ou seja, indica o menor nível de significância com o qual a hipótese nula é rejeitada probabilidade de *outliers*. Se esse valor for menor do que α, que para o estudo é de 0,05, então a hipótese nula é rejeitada e a alternativa é imediatamente aceita.

Nos gráficos das Figuras 1, 2, 3, são apresentados os resultados dos experimentos realizados usando o algoritmo *Quicksort* sem *threads* e com 2 e 4 *threads*. De maneira geral, os gráficos mostram que, quanto maior a quantidade de elementos, maior o ganho que se tem utilizando a nova implementação do *Quicksort* que faz uso de *threads*. Também de se notar são o ganho maior que se tem ao usar o nova implementação para ordenar elementos que já estejam ordenados crescente ou decrescentemente; e a pouca diferença que, nesses mesmos casos, usar 4 no lugar de 2 *threads* faz.

Para dar melhor entendimento aos gráficos, o estudo estatístico deles tem seus resultados apresentados na Tabela I. Através deles, pode-se verificar que a hipótese nula não foi válida em nenhum caso onde 2 *threads* foram usadas, indicando assim, com 95% de confiança, que o uso dessa quantidade de *threads* melhorou o tempo de execução em relação à implementação tradicional.

Aumentando o número de *threads* para 4, o estudo estatístico mostra que a hipótese nula foi válida para todos os casos onde o tamanho da amostra foi de 10.000, ou seja, com 95% de confiança, não fez diferença usar 4 *threads* ou nenhuma. Isso aconteceu devido à sobrecarga gerada para criar

[2]Um experimento com 0 *threads* significa a execução da implementação tradicional de um algoritmo de ordenação, enquanto que experimentos com 2 ou 4 *threads* significam as novas implementação do algoritmo.

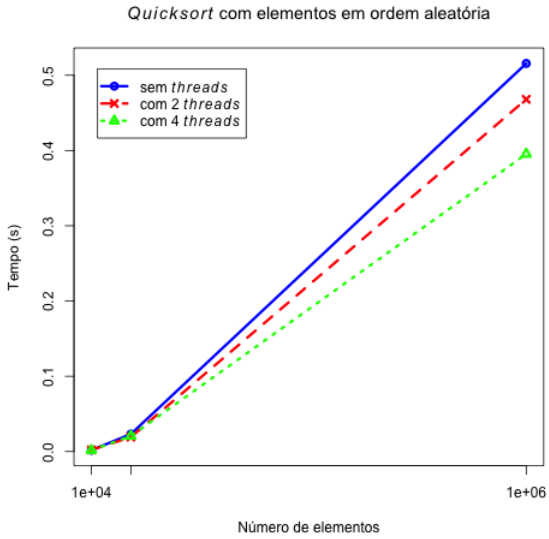

Figura 1. Médias dos tempos de execução para o *Quicksort* com e sem *threads* para um conjunto de elementos ordenados de forma aleatória.

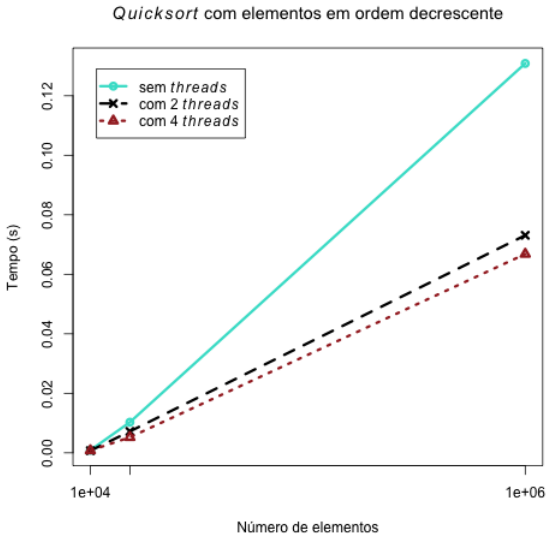

Figura 3. Médias dos tempos de execução para o *Quicksort* com e sem *threads* para um conjunto de elementos ordenados de forma decrescente.

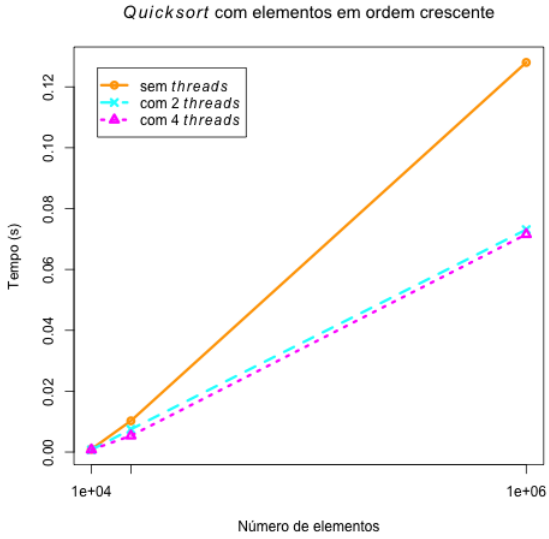

Figura 2. Médias dos tempos de execução para o *Quicksort* com e sem *threads* para um conjunto de elementos ordenados de forma crescente.

as *threads*, o que anulou qualquer melhoria de desempenho. Entretanto, para os demais conjuntos, a nova implementação, com 4 *threads*, foi melhor do que a tradicional.

Tabela I
RESULTADOS DOS EXPERIMENTOS UTILIZANDO O ALGORITMO *Quicksort* SEM NENHUMA *thread* E COM 2 E 4 *threads*.

Amostra	**Ordem inicial**	***h* (2 *threads*)**	***p* (2 *threads*)**	***h* (4 *threads*)**	***p* (4 *threads*)**
10.000	A	1	3,2720e-293	0	0,3221
	C	1	0	0	0,1920
	D	1	5,6605e-05	0	0,3673
100.000	A	1	0	1	1,7491e-116
	C	1	0	1	5.7504e-267
	D	1	0	1	1,6309e-302
1.000.000	A	1	2,9234e-260	1	0
	C	1	0	1	0
	D	1	0	1	0

Em tratando-se do *Merge sort*, os resultados exibidos nos gráficos das Figuras 4, 5 e 6 geram conclusões similares as que foram feitas para o *Quicksort*: o uso de *threads* melhorou o desempenho, mas aumentar a quantidade delas fez pouca diferença. Além disso, é facilmente observado que o ganho no *Merge sort* é maior do que o obtido no *Quicksort* .

Analisando a Tabela II com os resultados do estudo estatístico para o *Merge sort*, pode ser observado que em todos os casos, o uso de *threads* melhorou o desempenho do algortimo.

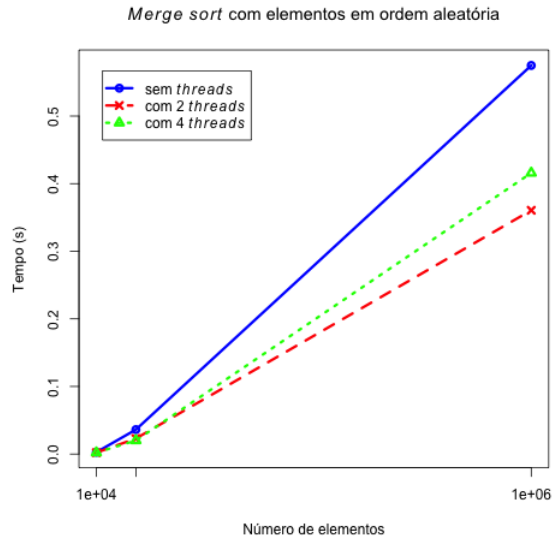

Figura 4. Médias dos tempos de execução para o *Merge sort* com e sem *threads* para um conjunto de elementos ordenados de forma aleatória.

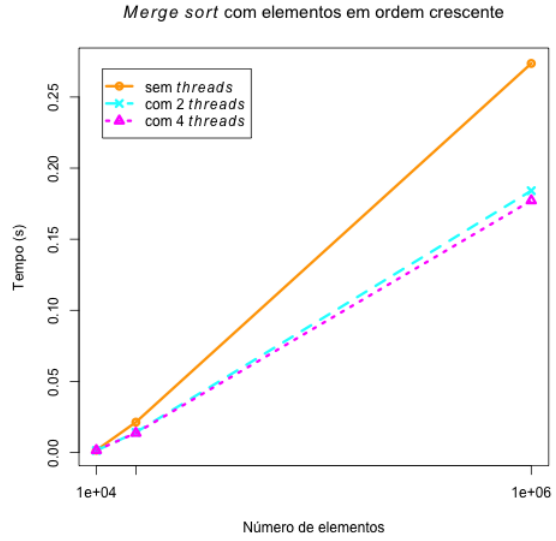

Figura 5. Médias dos tempos de execução para o *Merge sort* com e sem *threads* para um conjunto de elementos ordenados de forma crescente.

Tabela II
RESULTADOS DOS EXPERIMENTOS UTILIZANDO O ALGORITMO *Merge sort* SEM NENHUMA *thread* E COM 2 E 4 *threads*.

Amostra	**Ordem inicial**	***h* (2 *threads*)**	***p* (2 *threads*)**	***h* (4 *threads*)**	***p* (4 *threads*)**
10.000	A	1	0	1	5,6660e-11
	C	1	6,6603e-12	1	0,0346
	D	1	0	1	0
100.000	A	1	0	1	0
	C	1	6,1656e-272	1	7,8653e-201
	D	1	0	1	2,8806e-241
1.000.000	A	1	0	1	0
	C	1	0	1	7,0057e-279
	D	1	0	1	2,5452e-275

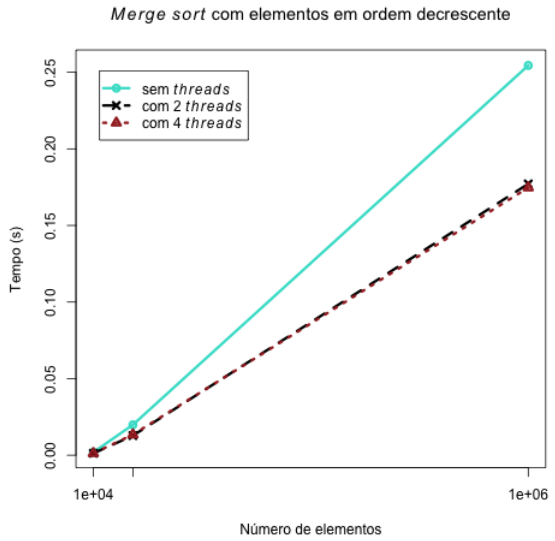

Figura 6. Médias dos tempos de execução para o *Merge sort* com e sem *threads* para um conjunto de elementos ordenados de forma decrescente.

VI. AVALIAÇÃO E PREVISÃO DO DESEMPENHO DOS ALGORITMOS

De posse dos resultados dos experimentos realizados, foi possível proceder com um resumo estatístico e testes de aderência para verificar quais as funções que melhor descrevem o comportamento dos algoritmos. Para todos os experimentos foi verificado que a função de distribuição de probabilidade (fdp) exponencial é um representante aceitável.

Conhecendo o comportamento estocástico referente ao tempo de processamento do algoritmo e a respectiva utilização dos recursos da máquina e inviabilidade de realizar o experimento em máquinas com grande diversidade de configurações, foi elaborado um modelo em GSPN que representa o algoritmo como uma caixa-preta, conforme a Figura 7.

No modelo da Figura 7, a constante "TAM" representa o tamanho da amostra de acordo com o experimento anterior; "NS" representa o número de *threads* reais do processador da máquina; "TOrd" representa o tempo médio de processamento de uma *thread* por amostra; "*Utilization*" é a métrica de desempenho que representa a taxa de utilização das *threads* do processador; "*Throughput*" representa a vazão de dados ordenados por unidade de tempo produzido pelo algoritmo. O lugar "*In*" é um acumulador de *tokens*, que representam as amostras, "CN" representa a quantidade de *threads* reais do processador, "*Busy*" representa o estado em que a thread está em execução; "*Out*" acumula os *tokens* que representam as amostras ordenadas. A transição "T1" representa o evento de alocação de tarefas para uma *thread* e "T0" a fdp exponencial que representa o tempo de processamento de uma *thread* por amostra.

Após a análise transiente em que todas as amostras são ordenadas, foram obtidos os valores para utilização e *throughput* apresentados nas Figuras 8 e 9 e nas Tabelas III e IV para 2, 4,

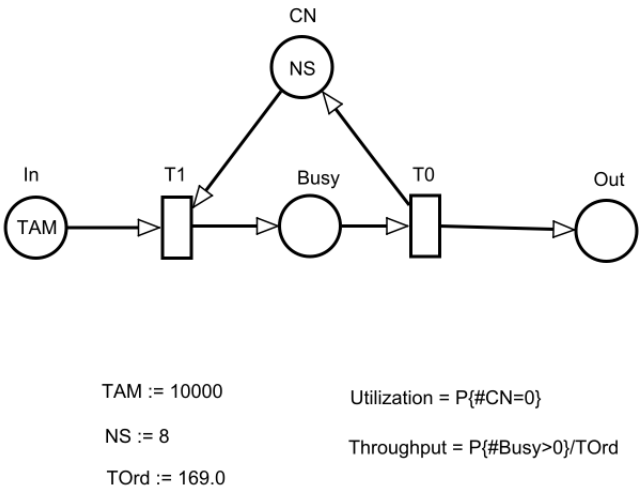

Figura 7. Modelo parametrizado em GSPN do algoritmo de ordenação caixa-preta.

8, 16, 32 e 64 *threads*. Os resultados foram validados através do teste de hipótese entre os resultados obtidos pelo modelo e pelo experimento para 2 e 4 threads.

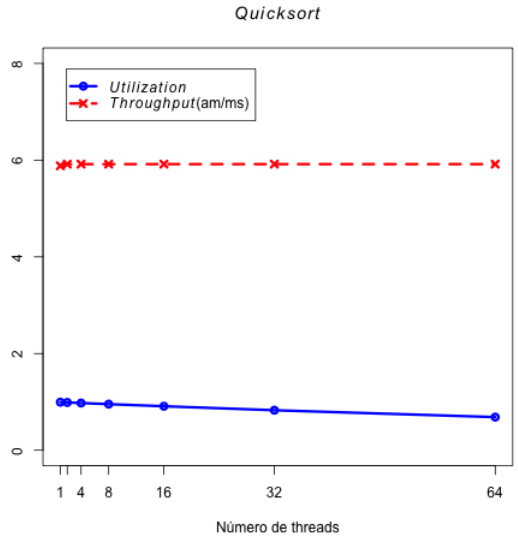

Figura 8. Valores das métricas *Throughput* e *Utilization* para diferente números de *threads* utilizando o *Quicksort*.

Tabela III
RESULTADOS DA ANÁLISE DE DESEMPENHO DO ALGORITMO *Quicksort*.

Threads	***Utilization***	***Throughput (am/ms)***
1	0,994117647	5,88235294
2	0,988235701	5,91675029
4	0,976473059	5,91715959
8	0,95389872	5,91715976
16	0,909922767	5,9171598
32	0,827959443	5,9171598
64	0,685516839	5,9171598

Diante dos resultados obtidos da avaliação do desempenho do *Quicksort*, Figura 8 e Tabela III, é possível verificar que não há uma melhoria acentuada na vazão, ou *throughput*, com relação ao aumento de *threads* dos processadores, mas a utilização dos recursos cai rapidamente como esperado.

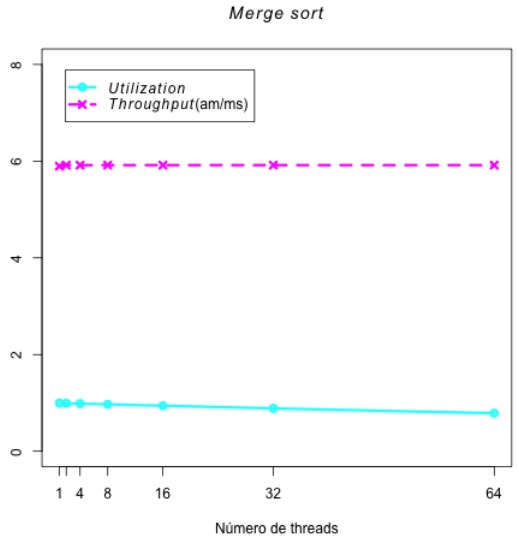

Figura 9. Valores das métricas *Throughput* e *Utilization* para diferente números de *threads* utilizando o *Merge sort*.

Tabela IV
RESULTADOS DA ANÁLISE DE DESEMPENHO DO ALGORITMO *Merge sort*.

Threads	***Utilization***	***Throughput (am/ms)***
1	0,996254682	5,8949981
2	0,992523391	5,9170768
4	0,985102681	5,9171598
8	0,970427292	5,9171598
16	0,94172913	5,9171598
32	0,886853754	5,9171598
64	0,78650958	5,9171598

Diante dos resultados obtidos para o *Merge sort*, Figura 9 e Tabela IV, é possível notar que a vazão alcança o seu teto superior já a partir de 4 *threads*. Já a utilização, decai de forma mais discreta que para o *Quicksort*.

VII. CONCLUSÃO

Na maioria dos casos, o uso de *threads* nos algoritmos de ordenação analisados surte um efeito positivo, fazendo com que esses algoritmos tenham um melhor desempenho em uma máquina com um processador de múltiplos núcleos.

Entretanto, no *Quicksort*, em casos com 10.000 elementos e uso de 4 *threads*, a sobrecarga de usar *threads* acabada anulando qualquer ganho de desempenho do qual o algoritmo poderia se beneficiar.

O *Merge sort*, por sua vez, sempre consegue melhores resultados, independente dos parâmetros do experimento em questão. O algoritmo *Merge sort* também apresenta maiores

ganhos com a utilização de *threads*, em relação ao *Quicksort*, possivelmente devido a sua natureza e ao modo como foram implementadas as divisões em *threads*.

No algoritmo *Quicksort* a divisão em *threads* pode resultar em uma *thread* com poucos elementos e outra com muitos elementos. Já no *Merge sort* a divisão em *threads* sempre produz *threads* simétricas, ou seja, com mesmo número de elementos, levando, portanto a um melhor aproveitamento do paralelismo do processador, já que a carga neste caso é dividida por igual entre os núcleos. No entanto, o modelo em GSPN dos algoritmos mostra que a medida que aumentam o numero de *threads*, o desempenho dos algoritmos tornam-se os mesmos, já que são algoritmos da classe dividir para conquistar. Mas o*Quicksort* utilizaria menos recursos da CPU.

Como trabalhos futuros, sugere-se realizar experimentos em máquinas com configurações diferentes e fazer implementações em outras linguagens de programação, bem como de outros algoritmos de ordenação.

Além disso, sugere-se também investigar casos de experimentos nos quais são permitidas a criação de um número de *threads* maior do que o número de núcleos do processador.

REFERÊNCIAS

[1] H. Demuth, *Electronic Data Sorting.PhD thesis* Stanford University, 1956.

[2] Hoare, C. A. R., *Quicksort*. Computer Journal 5 (1): 10-15, 1962.

[3] Katajainen, J., Pasanen, T. e Teuhola, J., *Practical in-place mergesort*. Nordic Journal of Computing 3: pp. 2740, 1996.

[4] Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[5] Harvey M. Deitel and Paul J. Deitel, *Java How to Program*. Prentice Hall; 5 edition, 2003.

[6] "Documentação de Threads," Disponível em: http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Thread.html Último Acesso: 19 de julho de 2011.

[7] MARSAN, M. A.; BALBO, G.; CONTE, G.; DONATELLI, S.; FRANCESCHINIS, *G. Modelling with Generalized Stochastic Petri Nets*. New York: Wiley Series in Parallel Computing - John Wiley and Sons, 1995.

[8] "Ttest no MATLAB," Disponível em: http://www.mathworks.com/help/toolbox/stats/ttest2.html Último Acesso: 19 de julho de 2011.