Execução Eficiente do Algoritmo de Leilão nas Novas Arquiteturas Multicore

Alexandre C. Sena[1], Aline Nascimento[2], Cristina Vasconcelos[2] e Leandro A. J. Marzulo[1]

[1]Instituto de Matemática e Estatística
Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, Brasil

[2]Instituto de Computação
Universidade Federal Fluminense (UFF), Rio de Janeiro, Brasil

{asena,leandro}@ime.uerj.br,{aline,crisnv}@ic.uff.br

Resumo. *O algoritmo de leilão tem sido amplamente utilizado para resolver o problema de emparelhamento de grafos bipartidos e sua implementação paralela é empregada para encontrar soluções ótimas em um tempo computacional aceitável. Além disso, as novas arquiteturas* multicore*, além de seus vários núcleos de processamento, possuem um conjunto de instruções SIMD que pode aumentar o desempenho da aplicação quando exatamente as mesmas operações necessitam ser realizadas em múltiplos dados. Nesse contexto, o objetivo deste trabalho é explorar todo o potencial dessas arquiteturas na execução do algoritmo de leilão. Para alcançar este objetivo, versões vetorizadas foram implementadas e avaliadas. Em seguida, essas versões foram executadas em paralelo utilizando a biblioteca OpenMP. Os resultados mostram que a versão vetorizada consegue, em média, um desempenho dez vezes melhor que a versão sequencial, enquanto a versão vetorizada paralela é capaz de aproveitar todo o potencial das novas arquiteturas* multicore*, atingindo um desempenho até* 200 *vezes melhor do que a versão sequencial.*

1. Introdução

Existem diversos problemas onde são recebidos como entrada dois conjuntos distintos, a partir dos quais um emparelhamento ótimo deve ser encontrado. Esses problemas podem ser modelados através de grafos bipartidos, cujos nós representam os elementos a serem emparelhados e as arestas podem ser associadas com pesos que representam os custos para emparelhar os nós correspondentes. A combinação (emparelhamento) ótima é o subconjunto final de pares, em que cada nó está presente em no máximo um único par e a soma das arestas escolhidas é mínima/máxima.

O problema de emparelhamento de grafos bipartidos é explorado em diversas áreas tais como controle distribuído e alocação de instalações (*distributed control and facility allocation*), na bioinformática para verificar as interações e semelhanças entre proteínas [Kollias et al. 2013] e na Visão Computacional, onde o objetivo pode ser, por exemplo, o emparelhamento de duas imagens ou reconhecimento de objetos, avaliando as semelhanças entre seus pontos [Shokoufandeh and Dickinson 1999, Vasconcelos and Rosenhahn 2009].

O algoritmo de leilão [Bertsekas 1979] é amplamente utilizado para resolver o problema de emparelhamento de grafos bipartidos. Originalmente, ele foi proposto para

atribuir m pessoas a n objetos distintos, onde cada par (pessoa, objeto) possui um custo associado que representa sua afinidade. O objetivo principal é atribuir uma pessoa ao objeto com maior afinidade, maximizando a soma total [Bertsekas 1992, Carpaneto et al. 1988].

A análise de imagens pode requerer uma grande quantidade de processamento, uma vez que imagens densas podem ter milhares de pontos a serem considerados ou, mesmo para imagens menores, pode ser necessário comparar uma sequência grande de imagens. Portanto, para resolver o problema de emparelhamento de duas imagens em um tempo computacional razoável, a computação paralela tem sido amplamente utilizada [Bertsekas and Castañon 1991, Kollias et al. 2012, Sathe et al. 2012]. Uma opção para aumentar o desempenho do algoritmo de leilão são as novas arquiteturas *multicore* disponíveis, que além de múltiplas unidades de processamento, possuem também instruções SIMD que, quando utilizadas eficientemente, são capazes de aumentar consideravelmente o desempenho das aplicações sequenciais.

Nesse contexto, o principal objetivo deste trabalho é executar o algoritmo de leilão eficientemente nas máquinas *multicore* modernas. Para isso, foram implementadas e analisadas versões sequenciais e paralelas (em OpenMP) do algoritmo de leilão, utilizando vetorização com instruções *intrinsics*. Resultados mostram que foi possível atingir um ganho de mais 10 vezes com a versão sequencial vetorizada e de até 200 vezes com a versão paralela vetorizada.

Este trabalho está dividido da seguinte maneira: a Seção 2 apresenta o algoritmo de leilão. Na Seção 3, a versão vetorizada proposta é descrita e avaliada. A implementação paralela utilizando a biblioteca OpenMP é apresentada e avaliada na Seção 4. Alguns trabalhos relacionados são descritos na Seção 5. Por fim, a Seção 6 apresenta as conclusões e possíveis trabalhos futuros.

2. Algoritmo de Leilão

Esta seção apresenta uma breve descrição sobre o Algoritmo de Leilão e sua implementação. O algoritmo de leilão clássico é definido como o problema de emparelhar m pessoas com n objetos supondo que há um benefício a_{ij} associado ao emparelhamento da pessoa i com o objeto j. Além disso, cada objeto tem um preço p_j associado e a pessoa que receber o objeto deve pagar seu preço p_j [Bertsekas 1979]. O valor associado ao objeto j pela pessoa i é $a_{ij} - p_j$ e cada pessoa i deseja ser associada ao objeto j_i de maior valor l_i (maior lance), conforme a Equação 1:

$$l_i = \max_{j=1,\ldots,n} \{a_{ij} - p_j\} \tag{1}$$

O Algoritmo de Leilão possui duas etapas principais: (1) OFERTA (Figura 1(a)), na qual as pessoas dão lances dinamicamente pelos objetos que desejam se associar, e (2) ASSOCIAÇÃO (Figura 1(b)), na qual o melhor lance recebido para cada objeto é selecionado individualmente, determinando suas associações e novos preços. O preço de cada objeto é aumentado de acordo com o valor do melhor lance por ele recebido.

O fato de os preços não serem reduzidos em nenhum momento do algoritmo garante que mesmo em casos de disputas de um conjunto de pessoas por um objeto específico, em algum momento o encarecimento de seu preço com o passar das interações

torna outros objetos mais e mais atrativos, de maneira que em algum momento alguma pessoa fará o lance final ao objeto disputado e a ele será associado enquanto que os demais terão perdido o interesse e passarão a investir em outros objetos [Bertsekas 1979].

O algoritmo itera em rodadas repetindo as duas etapas principais. Ao fim de cada rodada é produzido um conjunto de preços e associações (Figura 1(b), *linhas 5 e 6*). Se todas as pessoas estão satisfeitas com isso, ou seja, cada pessoa está associada a um objeto, o algoritmo termina. Caso contrário, cada pessoa livre i ofertará um novo lance para um objeto j, onde os lances são calculados como a diferença entre o objeto de maior e segundo maior interesse para uma determinada pessoa (Figura 1(a), *linhas 8 a 18*). É importante ressaltar que uma constante infinitesimal ϵ é adicionada ao valor do maior lance para tratar a convergência em casos de empate. Após a oferta de lances, um objeto j será então associado a pessoa i que ofereceu o maior valor associado (Figura 1(b), *linhas 9 a 16*).

Desta forma, enquanto existirem pessoas sem objetos associados, o leilão continua trocando as pessoas associadas aos objetos, e ajustando o preço de cada objeto j com os valores dos maiores lances atribuídos a eles (Figura 1(b), *linha 5*).

3. Vetorização do Algoritmo de Leilão

As máquinas *multicore*, além de vários núcleos de processamento, possuem instruções SIMD especializadas que podem aumentar consideravelmente o desempenho da aplicação quando uma mesma operação pode ser realizada em um conjunto de dados. O processo de conversão de uma implementação escalar de um programa (que realiza uma operação em um par de operandos por vez) para um processo vetorial (em que uma única instrução pode se referir a um vetor) é chamado de vetorização [Mark-Sabahi 2012]. Embora para alguns casos o próprio compilador seja capaz de vetorizar o programa do usuário, em geral, ele só é capaz de vetorizar códigos simples ou que estejam bem otimizados. Essa dificuldade acontece pois, para vetorizar um programa, o compilador tem que ser capaz de identificar a ausência de dependência de dados nas operações e que o acesso a memória seja contíguo [Mark-Sabahi 2012].

Assim, em muitos casos, o próprio programador tem que ser capaz de vetorizar o seu programa e, com isso, extrair todo o potencial de desempenho das instruções SIMD.

```
01 | ofertaLance(A, p, l, m, n) {
02 |   for (i = 0 ; i < m ; i++)
03 |     if pessoa i não possui objeto associado
04 |       (l[i].obj,l[i].val) = maiorSegMaior(A[i],p,n);
05 |     else
06 |       (l[i].obj,l[i].val) = (-1,-1);
07 | }
08 | maiorSegMaior(Ai, p, n) {
09 |   (mInd,mValor,smVal) = (0,(Ai[0]-p[0]),-1);
10 |   for (j = 1 ; j < n ; j++) {
11 |     vaux = Ai[j]-p[j];
12 |     if ( mVal < vaux )
13 |       (smVal,mVal,mInd) = (mVal,vaux,j);
14 |     else if ( vaux > smVal )
15 |       smVal = vaux;
16 |   }
17 |   return ( mInd, (mVal - smVal + ε) );
18 | }
```

(a) Oferta

```
01 | associacao(l, p, m, n) {
02 |   for ( j = 0 ; j < n ; j++ ){
03 |     (mVal,mInd) = maiorValParaObj(l,j,m);
04 |     if Existe um novo vencedor para o objeto j
05 |       p[j] = p[j] + mVal;
06 |       atualizaAssociacao(mVal,mInd);
07 |   }
08 | }
09 | maiorValParaObj(l, j, m) {
10 |   (mVal,mObj) = (-1,-1);
11 |   for ( i = 0; i < m ; i++ )
12 |     if ( l[i].obj == j )
13 |       if ( l[i].val > mVal )
14 |         (mVal,mInd) = (l[i].val,i);
15 |   return (mVal, mInd);
16 | }
```

(b) Associação

Figura 1. Algoritmo de leilão.

Esta seção descreve três versões vetorizadas para o algoritmo de leilão. Elas foram desenvolvidas utilizando funções *Intrinsics*, que permitem o acesso a diversas instruções SIMD específicas dos processadores Intel, sem a necessidade de escrever código *assembly* [Intel 2007, Intel 2017].

A Figura 2 provê uma visão simplificada da estratégia de vetorização utilizada para o algoritmo de leilão. Neste exemplo, o algoritmo original para obter o maior valor de um vetor e seu índice é apresentado na Figura 2(a). Uma vetorização inicial para esse algoritmo é detalhada na Figura 2(b), onde as operações de desvios são trocadas por operações lógicas e todas as operações são realizadas em um bloco de elementos simultaneamente (4, neste exemplo). Por fim, o algoritmo de uma versão vetorizada otimizada, que evita executar um conjunto de instruções quando todos os elementos do bloco são menores ou iguais aos maiores valores vistos até o momento pode ser visto na Figura 2(c).

Uma explicação detalhada de como funciona a vetorização é apresentada na Figura 2(d). Do lado esquerdo da figura se encontra uma simplificação do algoritmo vetorizado detalhado na Figura 2(b), para achar o maior valor e seu índice, considerando um vetor de 8 elementos inteiros de 4 bits. O lado direito da figura mostra o valor (em hexadecimal) das variáveis ao longo da execução do algoritmo. Para facilitar foi destacado em azul o que está sendo lido, em vermelho o que está sendo atribuído e em cinza escuro o que foi lido e atribuído para a mesma variável. A linha 1 do algoritmo carrega as 4 primeiras posições do vetor v na variável vetorizada `mVal`, enquanto que a linha 2 carrega os índices dessas posições na variável `mInd`. A linha 3 carrega as próximas 4 posições do vetor na variável `auxV` e a linha 5 os índices dessas posições na variável `auxI`. Por sua vez, a linha 4 compara `auxV` com `mVal` colocando na variável vetorizada `mask` o resultado dessa comparação que é F (15) quando os valores de `auxV` são maiores que os correspondentes em `maiorV` e 0 caso contrário. Na linha 6, `mVal` recebe os maiores valores, comparando cada posição de `auxV` com `mVal` através da função `MAX`. As linhas de 7 a 9 descrevem os passos para pegar os índices das variáveis `mInd` e `auxI`, de acordo os maiores valores de cada uma das variáveis `auxV` e `maiorV`. Por fim, a redução encontra o maior valor e seu índice percorrendo todos os elementos da variável vetorizada. É importante observar que a redução só é executada uma vez ao final do algoritmo, enquanto as etapas anteriores acontecem $m/chunk$ vezes onde m é o numero de objetos (colunas da matriz) e $chunk$ o número de elementos na variável vetorizada (4, neste exemplo).

Neste trabalho foram elaboradas 3 versões vetorizadas para o algoritmo de leilão:

- **simples**: Nesta versão foram vetorizadas as funções `maiorSegMaior` e `maiorValorParaObj` (Figura 1), eliminando todos os comandos `if` com operações lógicas e comparação com máscara, conforme o exemplo da Figura 2(b). Desta forma, é necessário executar sempre todas as instruções para os elementos do vetor.
- **cAssociação**: Esta versão é baseada na versão **simples**, onde é feita uma otimização utilizando instruções condicionais na função `maiorValorParaObj` (Figura 1(b)), conforme o exemplo da Figura 2(c). Desta forma, evita-se a execução de instruções quando o objeto j não for encontrado em um bloco de l (linha 12 da Figura 1(b)).
- **cCompleto**: Esta versão é baseada na versão **cAssociação**, onde é feita uma otimização utilizando instruções condicionais na função `maiorSegMaior` (Fi-

```
01 | mVal = Ai[0];
02 | mInd = 0;
03 | for (j=1 ; j<m ; j++) {
04 |   if (Ai[j] > mVal) {
05 |     mVal = Ai[j];
06 |     mInd = j;
07 |   }
08 | }
```

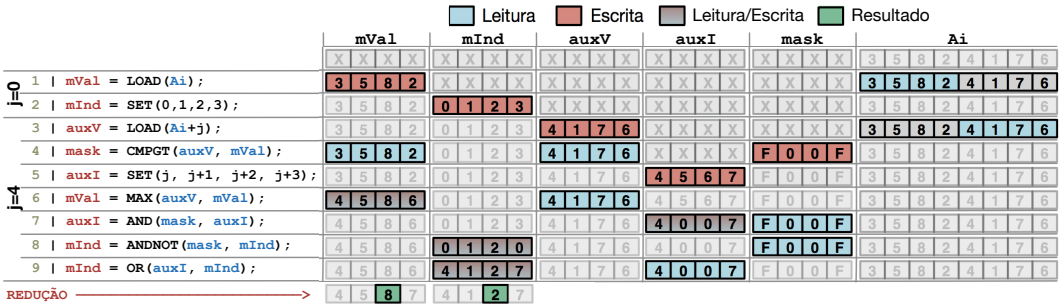
(a) Código original

```
01 | mVal = LOAD(Ai);
02 | mInd = SET(0,1,2,3);
03 | for (j=4 ; j<m ; j+=4) {
04 |   auxV = LOAD(Ai+j);
05 |   mask = CMPGT(auxV, mVal);
06 |   auxI = SET(i, j+1, j+2, j+3);
07 |   mVal = MAX(auxV, mVal);
08 |   auxI = AND(mask, auxI);
09 |   mInd = ANDNOT(mask, mIind);
10 |   mInd = OR(auxI, mInd);
11 | }
12 | REDUÇÃO
```

(b) Código vetorizado

```
01 | mVal = LOAD(Ai);
02 | mInd = SET(0,1,2,3);
03 | auxif = SET1(-1);
04 | for (j=4 ; j<m ; j+=4) {
05 |   auxV = LOAD(Ai+j);
06 |   mask = CMPGT(auxV, mVal);
07 |   cond = TESTZ(mask, auxif);
08 |   if (!cond) {
09 |     auxI = SET(i, j+1, j+2, j+3);
10 |     mVal = MAX(auxV, mVal);
11 |     auxI = AND(mask, auxI);
12 |     mInd = ANDNOT(mask, mIind);
13 |     mInd = OR(auxI, mInd);
14 |   }
15 | }
16 | REDUÇÃO
```

(c) Código vetorizado otimizado



(d) Execução vetorizada - baseada no código de (b)

Figura 2. Exemplo de vetorização para achar o maior elemento de um vetor e o seu índice.

gura 1(a)), conforme o exemplo da Figura 2(c). Neste caso são usadas duas instruções condicionais para evitar a execução de instruções quando nenhum valor de $vaux$ no bloco é maior do que os correspondentes em $mVal$, ou quando todos os valores de $vaux$ no bloco são maiores do que os correspondentes em $mVal$ (linha 12 da Figura 1(a)).

3.1. Análise Experimental

Todos os experimentos deste trabalho foram realizados em uma máquina *multicore* NUMA (*Non-Uniform Memory Access*) com 36 núcleos de processamento (2 *chips* Intel® Xeon® CPU E5-2699 v3 @ 2.30GHz) com 128 GB de RAM. Esse processador conta com instruções SIMD do tipo AVX2 de 256 bits. Todos os programas foram compilados utilizando o compilador icc da Intel com otimização -O3. No primeiro experimento foram utilizadas 35 matrizes reais distintas para o problema de emparelhamento de duas imagens. Para este problema os pontos da primeira imagem são representados nas linhas, enquanto que os pontos da segunda imagem nas colunas. Cada elemento da matriz representa a afinidade de um ponto da primeira imagem com um ponto da segunda imagem.

É importante ressaltar que, apesar do tempo para calcular o emparelhamento de algumas das matrizes da Tabela 1 ser muito pequeno, ele corresponde apenas ao tempo necessário para se emparelhar duas imagens. Para emparelhar uma sequência de imagens de um filme, por exemplo, o tempo pode aumentar consideravelmente. Neste caso, mesmo para matrizes pequenas, o uso do algoritmo de leilão vetorizado proposto é fundamental para se conseguir emparelhar sequências de imagens com um tempo de execução substancialmente menor. Como, para a maioria das instâncias, o tempo de execução é muito

pequeno, cada versão do algoritmo de leilão implementada para cada uma das matrizes foi executada 100 vezes e o coeficiente de variação foi calculado.

Tabela 1. Tempo de execução, coeficiente de variação e *speedup* das versões vetorizadas

Imagem	Características			Tempos (segundos)				Coef. de Variação (%)				*Speedup*		
	#Lin	#Col	#Iter	Orig	simp	cA	cC	Orig	simp	cA	cC	simp	cA	cC
alamo13	795	1241	404	0.6383	0.2362	0.0822	0.0851	8	16	16	13	2.70	**7.77**	7.50
alamo14	1241	1544	998	2.6583	0.8690	0.2807	0.2828	1	2	3	5	3.06	**9.47**	9.40
alamo16	1020	1241	964	1.7339	0.5902	0.2190	0.2145	3	5	13	14	2.94	7.92	**8.08**
alamo17	966	1241	925	1.5775	0.5303	0.1889	0.1903	5	7	16	15	2.97	**8.35**	8.29
alamo18	1069	3306	3097	15.3270	4.9661	1.5891	1.5219	2	10	1	0	3.09	9.65	**10.07**
bank	654	1418	134	0.2511	0.0737	0.0255	0.0057	14	13	12	11	3.41	**9.85**	9.77
bdom	1237	1470	841	2.1164	0.6814	0.2113	0.2109	0	3	3	3	3.11	10.01	**10.03**
cars	187	730	110	0.0368	0.0118	0.0059	0.0059	7	10	11	11	3.13	**6.19**	**6.19**
chinesebuilding	861	1683	224	0.4964	0.1806	0.0652	0.0647	7	14	14	14	2.75	7.62	**7.67**
eifell	387	1917	121	0.1882	0.0561	0.0239	0.0231	13	14	10	11	3.35	7.87	**8.14**
essighaus	951	898	3698	4.8937	1.6273	0.5866	0.5734	5	3	1	3	3.01	8.34	**8.53**
grafitti	1559	1641	1775	6.1955	1.9848	0.5845	0.5812	0	0	0	0	3.12	10.60	**10.66**
londonbridge	1219	1332	2472	5.4977	1.7815	0.5552	0.5432	0	1	1	2	3.09	9.90	**10.12**
madrid	1159	1360	671	1.4950	0.4964	0.1679	0.1696	2	3	9	9	3.01	**8.91**	8.82
metz	1286	1773	460	1.4388	0.4594	0.1378	0.1367	3	0	0	0	3.13	10.44	**10.52**
miduomo	719	1143	1374	1.6831	0.5829	0.2168	0.2145	3	9	16	16	2.89	7.76	**7.85**
miduomo02	2932	4369	1081	18.8940	6.0137	1.6779	1.6414	0	0	0	0	3.14	11.26	**11.51**
montreal	991	1193	1173	1.9659	0.6661	0.2406	0.2437	3	6	14	13	2.95	**8.17**	8.07
neubrandeburg	2142	3526	2380	24.4941	7.8014	2.2193	2.1722	0	0	0	0	3.14	11.04	**11.28**
notredame	3026	3297	2087	28.2164	8.9569	2.4323	2.4014	0	0	0	0	3.15	11.60	**11.75**
notredame12	806	1246	176	0.3255	0.1020	0.0327	0.0330	15	15	14	14	3.19	**9.94**	9.85
notredame16	845	1246	235	0.4233	0.1433	0.0469	0.0469	10	15	11	12	2.95	**9.02**	**9.02**
pantheon	1339	3306	288	1.7659	0.5711	0.1802	0.1736	2	0	5	2	3.09	9.80	**10.17**
pantheon2	2083	4195	1093	12.9919	4.1978	1.1913	1.1644	0	3	0	0	3.09	10.90	**11.16**
portcullis	907	1411	211	0.4371	0.1563	0.0512	0.0513	9	15	13	13	2.80	**8.54**	8.52
postoffice	837	1189	468	0.7249	0.2729	0.0952	0.0908	7	15	13	15	2.66	7.62	**7.98**
riga	963	1683	642	1.4520	0.4891	0.1732	0.1721	1	3	9	8	2.97	8.38	**8.44**
sanmarco	851	934	2014	2.3241	0.7918	0.3084	0.3060	3	7	13	13	2.94	7.54	**7.60**
sanmarco2	2454	3213	501	5.3515	1.7000	0.4690	0.4654	1	1	0	0	3.15	11.41	**11.50**
startgarder	3044	4339	786	14.0710	4.4684	1.2225	1.2126	0	0	0	2	3.15	11.51	**11.60**
startgarder3	4028	4484	3004	73.6040	23.4472	6.3750	6.2465	0	1	0	0	3.14	11.54	**11.78**
taj	528	1122	546	0.6012	0.2000	0.0731	0.0723	8	11	9	11	3.01	8.22	**8.32**
tavern	1286	2719	134	0.6443	0.2080	0.0637	0.0633	0	3	0	1	3.10	10.11	**10.19**
townsquare	1004	1112	1479	2.3336	0.7959	0.2914	0.2871	2	4	11	12	2.93	8.01	**8.13**
worldbuilding	1268	2067	438	1.5835	0.5017	0.1501	0.1484	2	0	0	1	3.16	10.55	**10.67**

Os resultados para as três versões vetorizadas, descritas na Seção 3, assim como o algoritmo de leilão sequencial original (Seção 2) podem ser vistos na Tabela 1. Para cada uma das matrizes (cada linha da tabela) são apresentados o nome da matriz e a sua quantidade de linhas, colunas e iterações, assim como, a média do tempo de execução, o coeficiente de variação e o *speedup* das versões vetorizadas em relação a versão sequencial original. Os maiores *speedups* para cada uma das linhas da tabela estão destacados em negrito.

O tempo para executar cada imagem é diretamente proporcional ao tamanho da matriz de pontos e a quantidade de iterações para achar o emparelhamento ótimo, uma vez que o algoritmo de leilão, a cada iteração, percorre toda a matriz. As versões vetorizadas tiveram um desempenho muito superior a versão original, aproveitando o grande potencial das instruções SIMD. A média dos tempos de execução da versão **simples** (**simp**), considerando todas as matrizes, foi aproximadamente três vezes menor do que da versão original (**Orig**), enquanto que as médias das versões **cAssociação** (**cA**) e **cCompleto** (**cC**) foram mais de 9 vezes menor. Para aproximadamente metade das matrizes, os coeficientes

de variação foram bem pequenos (abaixo de 10%), para outra metade, onde os tempos de execução são significativamente menores, o coeficiente de variação foi um pouco maior (abaixo de 16%). É importante ressaltar que para as maiores instâncias (onde o tempo de execução do algoritmo original foi maior que 5 segundos) o maior coeficiente de variação encontrado foi 3%, sendo que a maior parte ficou menor ou igual a 1%.

Analisando os melhores *speedups*, que estão destacados em negrito, fica claro que, para a maioria das matrizes, a versão **cCompleto** produziu os melhores desempenhos. Praticamente em todos os casos que **cAssociação** apresentou o maior *speedup*, a diferença foi muito pequena em relação a **cCompleto** e o coeficiente de variação muito alto, o que mostra uma grande variação nos tempos de execução para essas matrizes. Desse modo, a próxima seção irá analisar o desempenho apenas das implementações paralelas baseadas no algoritmo original e na versão vetorizada **cCompleto**.

4. Paralelização do Algoritmo de Leilão para Arquiteturas Multicore

O presente trabalho tem o objetivo de aproveitar todo potencial das arquiteturas *multicore*. Assim, além da vetorização que maximiza o uso das instruções SIMD, é necessário utilizar os múltiplos núcleos de processamento de maneira eficiente, através de técnicas de paralelização baseadas em memória compartilhada. Para isso, foi utilizado o modelo de programação paralela com memória compartilhada *OpenMP* [Chandra et al. 2000].

Como descrito na Seção 2, o algoritmo de leilão é basicamente uma sucessão de iterações, onde em cada uma dessas iterações são executadas as fases de OFERTA e ASSOCIAÇÃO. Na fase de OFERTA, o cálculo do lance dado por cada pessoa para um objeto não depende do lance de outra pessoa. Assim, cada lance pode ser realizado em paralelo, bastando para isso distribuir as m iterações do laço da Linha 02 da função `ofertaLance` (Figura 1(a)) entre as p *threads* disponíveis, através de uma diretiva `parallel for` do openMP.

Por sua vez, na fase de ASSOCIAÇÃO cada objeto seleciona o melhor lance recebido e determina sua associação e novo preço, sem depender das seleções dos outros objetos. Assim, o código OpenMP da fase de ASSOCIAÇÃO é simplesmente distribuir as n iterações do laço da Linha 02 da função `associacao` (Figura 1(b)) entre as p *threads* disponíveis, através de uma diretiva `parallel for` do openMP.

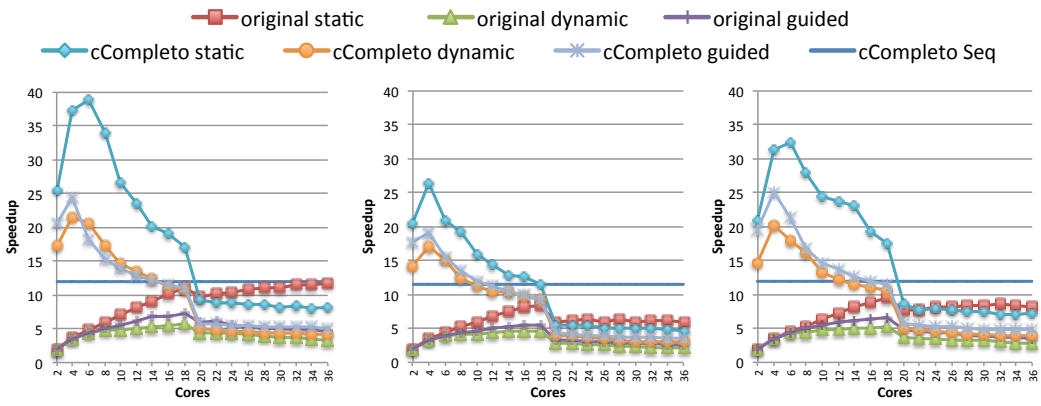
4.1. Análise Experimental

Os experimentos para avaliar as versão paralelas utilizaram o mesmo ambiente descrito na Subseção 3.1. Inicialmente, foram executadas as três matrizes com os maiores tempo de execução da Tabela 1 (*startgarder3*, *neubrandeburg* e *notredame*), variando a quantidade de *threads* OpenMP de 2 até 36. Além disso, dois outros parâmetros de execução foram avaliados: **escalonamento de laços** e **afinidade com o processador**. O **escalonamento de laços** do OpenMP permite uma série de opções de como dividir as tarefas do laço paralelizado entre as *threads*. Foram avaliadas as opções: *static*, *dynamic* e *guided*. A opção *static* divide todas as iterações do laço em frações de mesmo tamanho entre as *threads*. Já, as opções *dynamic* e *guided* dinamicamente vão atribuindo um conjunto de iterações para as *threads*. Enquanto a opção *dynamic* distribui uma iteração por vez, a opção *guided* inicia distribuindo frações grandes para cada *thread* que vão diminuindo durante a execução. Por sua vez, a **afinidade com o processador** pode ser habilitada para

determinar o conjunto de *cores* onde as *threads* do programa poderão ser escalonadas. O uso da afinidade pode ser interessante para evitar o uso de *Hyper-threading* e para mitigar custos com acesso à memória (NUMA).

O *speedup* para cada um dos cenários avaliados, calculado a partir da média de 5 execuções (o coeficiente de variação foi na média abaixo de 2% e não ultrapassou 5%), pode ser visto na Figura 3. Foram exibidos apenas os resultados com afinidade, visto que o desempenho das versões sem afinidade foi de aproximadamente metade do que o da versão com afinidade. A razão para tal comportamento se deve principalmente ao fato de que entradas pequenas tiveram uma grande redução do tempo de execução para as versões vetorizadas, resultando em baixa granularidade. Sendo assim, os custos de acesso à memória da arquitetura NUMA se tornaram mais evidentes, ao usar desnecessariamente *cores* dos dois *chips*. Para os demais experimentos, optamos por não executar os experimentos sem afinidade.

Com relação as políticas de escalonamento de laços, a opção *static* foi a que produziu os melhores resultados tanto para versão paralela original como também para a versão paralela vetorizada (*cCompleto*). Apesar da execução não ser uniforme, ou seja, a oferta de lances das pessoas serem espalhadas ao longo das linhas da matriz e a associação dos objetos as pessoas também serem espalhadas, isso não causa um desbalanceamento suficiente para que as políticas de escalonamento dinâmicas tirem proveito. Assim, a baixa sobrecarga da política estática produziu os melhores resultados.



(a) startgarder3 com afinidade (b) neubrandenburg com afinidade (c) notredame com afinidade

Figura 3. *Speedup* das versões paralelas vetorizada e original

Ao se analisar a escalabilidade é possível verificar que a versão paralela original apresentou ganho de desempenho a medida que a quantidade de *cores* aumenta até o limite de 18 cores. Especificamente para a matriz *startgarder3*, a maior das três, também houve melhora de desempenho com mais de 30 cores. Porém, para todos os cenários, o desempenho da versão paralela original foi pior do que a versão vetorizada sequencial (*cCompleto seq*). Por sua vez, a versão paralela vetorizada só escalou até 6 cores, sendo que para a menor matriz (*neubrandeburg*) somente até 4 cores. A razão para baixa escalabilidade é a granularidade fina das tarefas em função do excelente desempenho da versão vetorizada sequencial. Por exemplo, o tempo total de execução da versão original sequencial para a matriz *startgarder3* é de 73,6 segundos, enquanto que o tempo da versão

vetorizada sequencial é de apenas $6,2$. Por sua vez, tempo da execução paralela com 6 cores foi de apenas $1,92$ segundos para executar as 3004 iterações, ou seja, um tempo de aproximadamente $0,00063$ segundos por iteração. Assim, como o paralelismo ocorre a cada iteração do algoritmo, fica evidente a granularidade fina das tarefas nesse ponto, limitando sua escalabilidade. Mesmo assim, é importante destacar que para essa matriz a versão paralela vetorizada foi quase 40 vezes mais rápida que a versão original sequencial, utilizando apenas 6 *cores*. Para a menor matriz (*neubrandeburg*) o desempenho foi 26 vezes melhor utilizando apenas 4 *cores*.

É possível observar uma queda brusca de desempenho de 18 para 20 cores em todos os cenários apresentados na Figura 3. A explicação para tal comportamento é a sobrecarga no acesso a memória em função da arquitetura NUMA. Para até 18 cores, apenas um *chip* (processador) da máquina é utilizado, o que garante o acesso a memória próxima a ele. Por sua vez, com 20 ou mais cores pelo menos uma das *threads* vai ter que acessar a memória mais distante (acesso mais custoso), aumentando a sobrecarga.

O próximo experimento avalia as versões paralelas para matrizes maiores. Para isso foi utilizada uma aplicação que simula o movimento de partículas, ao longo do tempo, dentro de um ambiente 3D com diferentes velocidades, perturbadas por ruídos aleatórios. Essa aplicação permite criar casos de estudo com maior concorrência ou maior esparsidade pela escolha do número de partículas e definição da caixa envolvente. Dessa forma, permite criar matrizes de custo/benefício com diferentes características de alocação. Foram geradas matrizes de custo/benefício com diferentes tamanhos e quantidades de iterações e o emparelhamento dessas partículas, entre dois instantes de tempo, foi realizado utilizando as versões paralela original e vetorizada.

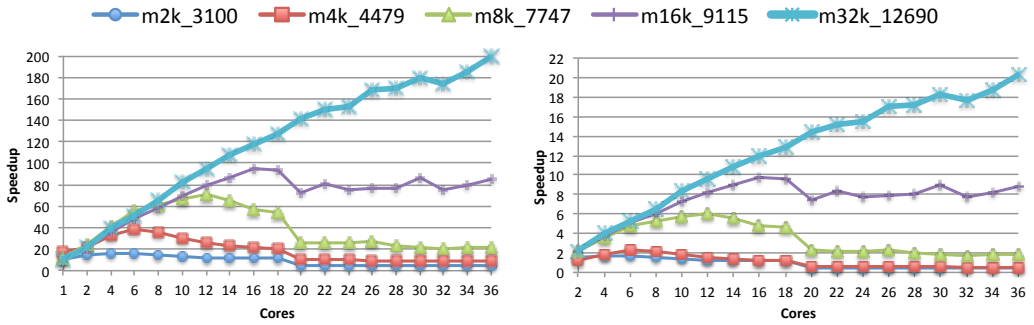
A Figura 4 apresenta os gráficos das execuções da versão paralela **cCompleto** com políticas de escalonamento de laços *static*, para as seguintes matrizes de entrada com diferentes tamanhos e iterações: 2000×2000 elementos e 3100 iterações (m2k_3100); 4000×4000 elementos e 3100 iterações (m4k_4479); 8000×8000 elementos e 7747 iterações (m8k_7747); 16000×16000 elementos e 9115 iterações (m16k_9115); 32000×32000 elementos e 12690 iterações (m32k_12690). As Figuras 4(a) e 4(b) mostram os *speedups* relativos a versão sequencial original e sequencial vetorizada, respectivamente.

As duas figuras mostram claramente o aumento da escalabilidade com o aumento do tamanho das matrizes de entrada. Por exemplo, como mostra a Figura 4(b), a entrada m32k_12690 atinge *speedup* de $20,2$ para 32 *cores*, enquanto que a entrada m2k_3100 apresenta *speedup* de apenas $1,2$ para o mesmo cenário. Como já destacado anteriormente, o motivo da baixa escalabilidade das matrizes pequenas é a granularidade fina das tarefas, principalmente em função do excelente desempenho da versão vetorizada sequencial. Assim, é esperado que para matrizes grandes todo o potencial das arquiteturas *multicore* possa ser aproveitado, mesmo considerando a sobrecarga de acesso a memória das arquiteturas NUMA.

Já, a Figura 4(a) destaca o excelente desempenho alcançado, em função da combinação da vetorização, que permite o uso eficiente das instruções SIMD, com o paralelismo, que permite uso dos múltiplos núcleos de processamento disponíveis nas arquiteturas *multicore*, atingindo o patamar de 200 de *speedup* para a entrada m32k_12690, executando em 36 *cores*, o que significou uma redução de mais 5 horas de execução com

a versão sequencial original para apenas $90,7$ segundos, com a versão vetorizada paralela.

Uma característica interessante é que, de maneira geral, a maioria das associações ocorrem nas primeiras iterações (por exemplo, para todas as matrizes apresentadas mais de 90% das associações ocorreram antes de atingir 500 iterações). Isso faz com que as iterações intermediárias e mais próximas do fim executem menos processamento, pois a maioria dos objetos já está associada, restando poucas pessoas disputando poucos objetos. Experimentos iniciais mostraram que é possível ajustar durante a execução a quantidade de *threads* para tirar proveito deste comportamento. Mais especificamente, foi verificado que, para a matriz m16k_9115, ao se iniciar a execução com 18 *threads* e, a partir do momento que mais de 90% dos objetos forem associados, reduzir a quantidade de *threads* para 16, diminui o tempo de execução.



(a) *speedups* em relação ao sequencial original (b) *speedups* em relação ao sequencial vetorizado

Figura 4. Execução Paralela para Matrizes de Emparelhamento de Partículas

5. Trabalhos Relacionados

O algoritmo de leilão para o problema de emparelhamento já foi cuidadosamente analisado em [Bertsekas 1979, Bertsekas 1992], onde foi demonstrado a importância do incremento ϵ-scaling que garante a otimalidade do algoritmo. Além disso, implementações paralelas síncronas e assíncronas do algoritmo de leilão foram propostas e executadas em uma máquina paralela real em [Bertsekas and Castañon 1991].

O trabalho em [Buš and Tvrdík 2009] introduziu a estratégia, chamada de *look-back*, que estende a implementação clássica com a habilidade de reutilizar informação de lances anteriores. O artigo mostra que é possível utlizar lances anteriores eficientemente, embora o algoritmo falhe em quase 30% para um dos tipos das entradas testadas.

Um algoritmo de leilão paralelo capaz de emparelhar grandes grafos bipartidos densos e esparsos foi apresentado em [Sathe et al. 2012], propondo uma nova estratégia ϵ-scaling. Resultados experimentais em um supercomputador Cray XE6 mostram que a implementação híbrida MPI−OpenMP reduziu drasticamente o tempo de execução, embora nenhuma análise do desempenho tenha sido apresentada.

O problema de alinhamento de grafo global em *clusters* de alto desempenho foi apresentado em [Kollias et al. 2014]. Este trabalho não só acha os vértices similares através do uso do algoritmo de leilão para emparelhamento de grafos bipartidos, mas, diferentemente dos outros trabalhos apresentados nesta seção, previamente computa a

matriz de similaridades. Matrizes contendo de 5.000 até $1.500.000$ vértices foram executadas em um supercomputador Cray XE6 eficientemente, especialmente para as matrizes grandes.

O trabalho apresentado em [Nascimento et al. 2016] analisa como o tamanho da matriz e quantidade de iterações influenciam no tempo de execução de uma versão híbrida (OpenMP/MPI) do algoritmo de leilão. Uma importante contribuição foi mostrar o custo de comunicação entre os nós do *cluster*, a cada iteração, e como isso afeta o desempenho do algoritmo de leilão.

Diferentemente de todos os trabalhos descritos nesta seção, este trabalho implementa e avalia versões sequenciais vetorizadas que tiram proveito das instruções SIMD disponíveis nas novas arquiteturas. Além disso, também implementa e avalia versões paralelas com a biblioteca OpenMP que mostram que o uso eficiente das instruções SIMD em conjunto com os processadores (*cores*) potencializam o ganho de desempenho nas novas arquiteturas *multicore*.

6. Conclusões e Trabalhos Futuros

A motivação prática para a adoção do algoritmo de leilão na resolução do problema de emparelhamento de grafos bipartidos é a existência de cenários onde o tamanho do problema é altamente proibitivo, tornando sua natureza distributiva uma formulação valiosa para o uso do paralelismo.

Este trabalho implementou e avaliou versões vetorizadas do algoritmo de leilão, assim como, implementações paralelas utilizando memória compartilhada. Os resultados mostram que é possível aproveitar todo potencial das novas arquiteturas *multicore*. Enquanto que a versão vetorizada foi em média 10 vezes mais rápida que a versão sequencial, sendo suficiente para resolver problemas pequenos muito rapidamente, a versão paralela vetorizada atingiu 200 de *speedup*, se mostrando como uma excelente alternativa para execução de matrizes médias e grandes em arquiteturas *multicore*.

Trabalhos futuros irão avaliar técnicas para variar a quantidade de unidades de processamento a serem utilizadas ao longo da execução, uma vez que a quantidade de trabalho a ser realizada diminui com o número de iterações. Além disso, otimizações para melhorar a sobrecarga de acesso a memória NUMA serão investigadas com o objetivo de aumentar ainda mais a escalabilidade do algoritmo de leilão nas máquinas *multicore*.

Agradecimentos

À FAPERJ, ao CNPq e à CAPES pelo apoio dado aos autores deste trabalho. Os autores também agradecem o uso dos recursos computacionais *manycore* mantidos e operados pelo Núcleo de Computação Científica da Universidade Estadual Paulista (NCC/UNESP), financiado parcialmente pela Intel, no contexto do projeto Intel/UNESP Modern Code.

Referências

Bertsekas, D. P. (1979). A distributed algorithm for the assignment problem. Technical report, Lab. for Information and Decision Systems, M.I.T., Cambridge, MA.

Bertsekas, D. P. (1992). Auction algorithms for network flow problems: A tutorial introduction. *Computational Optimization and Applications*, 1(1):7–66.

Bertsekas, D. P. and Castañon, D. A. (1991). Parallel synchronous and asynchronous implementations of the auction algorithm. *Parallel Comput.*, 17(6-7):707–732.

Buš, L. and Tvrdík, P. (2009). Towards auction algorithms for large dense assignment problems. *Computational Optimization and Applications*, 43(3):411–436.

Carpaneto, G., Martello, S., and Toth, P. (1988). Algorithms and codes for the assignment problem. *Annals of Operations Research*, 13(1):191–223.

Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., and McDonald, J. (2000). *Parallel Programming in OpenMP*. Morgan Kaufmann, 1st edition.

Intel (2007). Intel c++ intrinsic reference. Technical report, Intel.

Intel (2017). Intel intrinsics guide. Technical report, Intel.

Kollias, G., Sathe, M., Mohammadi, S., and Grama, A. (2013). A fast approach to global alignment of protein-protein interaction networks. *BMC Research Notes*, 6(1):1–11.

Kollias, G., Sathe, M., Schenk, O., and Grama, A. (2012). Fast parallel algorithms for graph similarity and matching. Technical report RR12-010, Department of Computer Science, Purdue University.

Kollias, G., Sathe, M., Schenk, O., and Grama, A. (2014). Fast parallel algorithms for graph similarity and matching. *Journal of Parallel and Distributed Computing*, 74(5):2400 – 2410.

Mark-Sabahi (2012). A guide to auto-vectorization with intel c++ compilers. Technical report, Intel.

Nascimento, A. P., Vasconcelos, C. N., Jamel, F. S., and Sena, A. C. (2016). A hybrid parallel algorithm for the auction algorithm in multicore systems. In *Inter. Symp. on Computer Architecture and High Perf. Comp. Workshops (SBAC-PADW)*, pages 73–78.

Sathe, M., Schenk, O., and Burkhart, H. (2012). An auction-based weighted matching implementation on massively parallel architectures. *Parallel Computing*, 38(12):595 – 614.

Shokoufandeh, A. and Dickinson, S. (1999). Applications of bipartite matching to problems in object recognition. In *In Proceedings, ICCV Workshop on Graph Algorithms and Computer Vision*, page http://www.cs.cornel.

Vasconcelos, C. N. and Rosenhahn, B. (2009). *Bipartite Graph Matching Computation on GPU*. Springer Berlin Heidelberg, Berlin, Heidelberg.