Resolvendo o Problema Quadrático de Alocação (QAP) em ambientes heterogêneos (CPUs-GPUs) com aplicação da Técnica de Reformulação e Linearização de nível 2 (RLT2)

Alexandre Domingues Gonçalves
Campus São Gonçalo - Instituto Federal do Rio de Janeiro - IFRJ
Rua José Augusto Pereira dos Santos s/n - Neves - São Gonçalo - RJ
alexandre.domingues@ifrj.edu.br

Artur Alves Pessoa
Engenharia de Produção - Universidade Federal Fluminense - UFF
Rua Passo da Pátria 156 - Bloco E - 4º andar - São Domingos - Niterói - RJ
artur@producao.uff.br

Lúcia Maria de Assumpção Drummond
Instituto de Computação - Universidade Federal Fluminense - UFF
Rua Passo da Pátria 156 - Instituto de computação - sala 525 - São Domingos - Niterói - RJ
lucia@ic.uff.br

Cristiana Bentes
Departamento de Geomática - Universidade Estadual do Rio de Janeiro - UERJ
Rua São Francisco Xavier 524 - 5o Andar - Bloco D - Maracanã - Rio de Janeiro - RJ
cris@eng.uerj.com

Ricardo Farias
COPPE Sistemas - Universidade Federal do Rio de Janeiro - UFRJ
Centro de Tecnologia - Bloco H - Sala 319 - Ilha do Fundão - Rio de Janeiro - RJ
rfarias@cos.ufrj.br

RESUMO

O Problema Quadrático de Alocação (QAP) é um clássico problema de otimização combinatória, classificado como NP-difícil e amplamente estudado. Consiste em atribuir N facilidades a N locais obedecendo a relação de 1 para 1 e com o objetivo de minimizar os custos obtidos pela soma dos produtos distância-fluxo. A aplicação da técnica de Reformulação e Linearização (RLT) ao QAP leva a uma relaxação linear justa porém de grandes dimensões e difícil resolução. Trabalhos anteriores baseado no RLT de nível 3 mostram que é necessário uma RAM na ordem de 700GB para processar uma instância considerada grande ($N = 30$). Nossa proposta apresenta uma versão modificada do algoritmo de Adams et al para execução em ambientes heterogêneos (CPUs-GPUs) baseada na RLT de nível 2. Os resultados obtidos, de acordo com a instância, chegam a ser 40 vezes mais rápidos e ocupam 95% menos memória do que a versão com RLT de nível 3.

PALAVRAS CHAVE. QAP, RLT, GPU.

Área Principal: Otimização Combinatória

ABSTRACT

The Quadratic Assignment Problem (QAP) is a classic combinatorial optimization problem, classified as NP-hard and thoroughly studied. This problem consists in assigning N facilities to N locations obeying the relation of 1 to 1, aiming to minimize costs of the displacement between the facilities. The application of reformulation and linearization technique (RLT) to the QAP leads to a tight linear relaxation but large and difficult to solve. Previous works based on level 3 RLT need a total of working memory of about 700GB to process large instances (N = 30 facilities). Our proposal presents a modified version of algorithm of Adams et.al. which executes on heterogeneous systems (CPUs, GPUs) based on level 2 RLT. Depending on the instance, our results are up to 40 times faster and occupy 95 % less memory than the version with level 3 RLT.

KEYWORDS. QAP. RLT. GPU.

Main Area: Combinatorial Optimization

1. Introdução

O Problema Quadrático de Alocação, ou QAP (sigla da literatura internacional utilizada aqui neste trabalho), é um dos mais difíceis e mais estudados problemas de otimização combinatória da literatura. Consiste em encontrar uma alocação de N facilidades a N locais obedecendo a relação de 1 para 1 e objetivando minimizar os custos de deslocamento entre facilidades, custos estes obtidos pela soma dos produtos distância-fluxo. Sua formulação foi apresentada inicialmente por Koopmans e Beckmann (1957) e tem aplicações práticas em alocação de objetos em departamentos, design de placas de circuitos eletrônicos, problemas de layouts, planejamento de construções, entre outros.

Dentre os estudos relacionados às aplicações do QAP temos: Heffley (1980) em problemas econômicos, Francis *et al*. (1992) com um *framework* para a atribuição de facilidades à locações, Hubert (1986) em análises estatísticas, Dickey e Hopkins (1972) em alocações em um campus universitário e Elshafei (1977) em um planejamento hospitalar. Para um estudo mais aprofundado sobre o QAP sugerimos as seguintes referências: Pardalos *et al*. (1994), Padberg e Rijal (1996), Burkard *et al*. (1998) e Cela (1997).

Métodos exatos para a solução do QAP requerem alto poder computacional, por exemplo, para uma instância de 30 facilidades (nug30), Hahn *et al*. (2013) levou 8 dias em um supercomputador composto de 32 máquinas Intel Xeon 2,2GHz e cerca de 700Gb de memória principal compartilhada. Uma estratégia para alcançar a solução exata de problemas muito complexos é dividi-lo em subproblemas menores na tentativa de resolvê-los de forma exata. Os algoritmos do tipo *branch-and-bound* são os mais conhecidos e utilizados como método exato para a solução do QAP.

Os limites inferiores são componentes essenciais para os procedimentos de *branch-and-bound*, pois permitem descartar um maior número de soluções alternativas na busca pela solução ótima. Os limites alcançados por Burer e Vandenbussche (2006), Adams *et al*. (2007) e Zhu *et al*. (2012) destacam-se como os mais justos para o QAP. A proposta de Burer e Vandenbussche (2006) consiste em relaxações *lift-and-project* de problemas binários inteiros, Adams *et al*. (2007) apresenta um algoritmo dual de subida baseado na técnica de reformulação e linearização de nível 2 e Zhu *et al*. (2012) também em um algoritmo dual de subida mas baseado na técnica de reformulação e linearização de nível 3.

A Técnica de reformulação e linearização (aqui também utilizada a sigla RLT da literatura internacional), foi inicialmente desenvolvida por Adams e Sherali (1990), Sherali e Adams (1999), com o objetivo de gerar relaxações lineares com limites inferiores justos para uma classe de problemas de programação inteira mista 0-1. Na literatura, aplicado ao QAP, encontramos o RLT de nível 1, ou RLT1, em Hahn e Grant (1998), o RLT de nível 2, ou RLT2, em Adams *et al*. (2007) e ainda o RLT de nível 3, ou RLT3, em Zhu *et al*. (2012). Versões paralelas do RLT3 são apresentadas em

Hahn *et al*. (2013) e Gonçalves *et al*. (2013), implementados para um cluster com memória compartilhada e sistemas distribuídos respectivamente. Os trabalhos citados têm mostrado que quanto maior o nível do RLT utilizado, mais justo é o limite inferior obtido, entretanto, a memória de trabalho (RAM) necessária para o processamento destes limites aumenta exponencialmente conforme cresce o nível do RLT.

Não encontramos na literatura trabalhos que abordem métodos exatos para a solução do QAP utilizando uma estrutura computacional composta de CPUs e GPUs, apenas trabalhos baseados em heurísticas, como o do Tsutsui e Fujimoto (2011) baseado no algoritmo de otimização de colônia de formigas com busca Tabu e utilizando algoritmos genéticos em Tsutsui e Fujimoto (2009).

A nossa proposta consiste em um algoritmo do tipo *branch-and-bound* que utiliza para cálculo do limite inferior uma versão modificada do algoritmo dual de subida de Adams *et al*. (2007) para ser executado em um ambiente heterogêneo (CPUs-GPUs). O algoritmo tira proveito dos bons limites alcançados pelo dual de subida RLT2 e do poder computacional das GPUs com baixo uso de memória de trabalho quando comparamos com algoritmos dual de subida RLT3 cujos limites são melhores que o anterior mas com a exigência de uma quantidade de memória de trabalho proibitiva para computadores comerciais.

Este trabalho está organizado da seguinte forma: Na seção 2 temos a formulação do QAP. Na seção 3 descrevemos a técnica de reformulação e linearização e a aplicação deste método na formulação do QAP. Na Seção 4 apresentamos o procedimento dual de subida baseado no RLT2 e suas operações. Na Seção 5 descrevemos as considerações sobre GPUs. Na Seção 6 detalhamos a implementação do *branch-and-bound* para ser executado em um ambiente heterogêneo (CPUs - GPUs). Na Seção 7 reportamos os resultados experimentais, e, finalmente, as considerações e propostas de novos trabalhos na Seção 8.

2. A formulação do Problema Quadrático de Alocação - QAP

Dados N facilidades, N locais, um fluxo f_{ik} de cada facilidade i para cada facilidade k, $k \neq i$, e uma distância d_{jn} de cada local j para cada local n, $n \neq j$, o QAP consiste em atribuir cada facilidade i à exatamente um local distinto j de modo a encontrar:

$$min \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{\substack{k=1 \\ k \neq i}}^{N} \sum_{\substack{n=1 \\ n \neq j}}^{N} f_{ik} d_{jn} x_{ij} x_{kn} \tag{1}$$

$$\text{s.a.} \sum_{i=1}^{N} x_{ij} = 1 \ \ \forall \ \ \ j = 1, ..., N \tag{2}$$

$$\sum_{j=1}^{N} x_{ij} = 1 \ \ \forall \ \ i = 1, ..., N \tag{3}$$

$$x_{ij} \in \{0, 1\} \ \ \forall \ \ i = 1, ..., N; \ \ \ j = 1, ..., N \tag{4}$$

3. Técnica de reformulação-linearização - RLT

Para problemas envolvendo n variáveis, a técnica de reformulação e linearização (RLT) estabelece n níveis hierárquicos de relaxação para um polígono convexo de soluções inteiras para o problema original. Para um dado nível k, $k \in \{1, ..., n\}$, também denominado como RLTk, ele utiliza todos os fatores polinomiais de grau k envolvendo k variáveis binárias x ou seus complementares $(1 - x)$.

A linearização consiste em adicionar variáveis auxiliares, cada uma representando um possível produto de variáveis originais ou complementares, e relacioná-los aos originais por novas

restrições. Cada nova restrição corresponde a uma restrição original multiplicada por um produto de variáveis originais ou complementares.

Assumindo $c_{ijkn} = f_{ik}d_{jn}$ e $d_{ijknpq} = 0$ para todo $i, k, p = 1, ..., N$ com i, k, p distintos, e $j, n, q = 1, ..., N$, também com j, n, q distintos. A seguir a formulação obtida quando aplicado o método RLT2 à formulação do QAP (Equações (1-4)), conforme Adams *et al*. (2007);

$$min \left\{ \sum_{i=1}^{N}\sum_{j=1}^{N} b_{ij}x_{ij} + \sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{\substack{k=1\\k\neq i}}^{N}\sum_{\substack{n=1\\n\neq j}}^{N} c_{ijkn}x'_{ijkn} + \sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{\substack{k=1\\k\neq i}}^{N}\sum_{\substack{n=1\\n\neq j}}^{N}\sum_{\substack{p=1\\p\neq i,k}}^{N}\sum_{\substack{q=1\\q\neq j,n}}^{N} d_{ijknpq}x''_{ijknpq} \right\} \tag{5}$$

$$\text{s.a.} \sum_{\substack{k=1\\k\neq i}}^{N} x'_{ijkn} = x_{ij} \ : \ (i, j, n = 1, .., N) \ ; \ n \neq j \tag{6}$$

$$\sum_{\substack{q=1\\j\neq n}}^{N} x'_{ijkn} = x_{ij} \ : \ (i, j, k = 1, .., N) \ ; \ i \neq k \tag{7}$$

$$\sum_{\substack{p=1\\p\neq i,k}}^{N} x''_{ijknpq} = x'_{ijkn} \ : \ (i, j, k, n, q = 1, .., N) \ ; \ k \neq i; \ \ j, q, n \ distintos \tag{8}$$

$$\sum_{\substack{q=1\\q\neq j,n}}^{N} x''_{ijknpq} = x'_{ijkn} \ : \ (i, j, k, n, p = 1, .., N) \ ; \ i, k, p \ distintos; \ n \neq j \tag{9}$$

$$x'_{ijkn} = x'_{knij} \ (2 \ complementares) \ : \ (i, j, k, n = 1, .., N) \ ; \ i < k \ ; \ j \neq n \tag{10}$$

$$\begin{gathered} x''_{ijknpq} = x''_{ijpqkn} = x''_{knijpq} = x''_{knpqij} = x''_{pqijkn} = x''_{pqknij} \ (6 \ complementares) : \\ (i, j, k, n, p, q = 1, .., N) \ ; \ i < k < p \ ; \ j, n, q \ distintos \end{gathered} \tag{11}$$

$$x_{ij} \geq 0 \ : \ (i, j = 1, .., N) \ ; \tag{12}$$

$$x'_{ijkn} \geq 0 \ : \ (i, j, k, n = 1, .., N) \ ; \ i < k \ ; \ j \neq, n \tag{13}$$

$$x''_{ijknpq} \geq 0 \ : \ (i, j, k, n, p, q = 1, .., N) \ ; \ i < k < p \ ; \ j, n, qdistintos \tag{14}$$

e equações (2) e (3);

Para obter a formulação (5 - 14), aplicamos inicialmente os passos do RLT1: São adicionadas $(N \times (N-1))^2$ novas restrições (6) e (7) pela multiplicação das restrições (2) e (3) por cada uma das N^2 variáveis binárias x_{kn}, , com $i \neq k$ e com $j \neq n$, (e trocando os índices i e k e os índices j e n). Logo após, substituímos cada produto $x_{ij}x_{kn}$ pela variável binária correspondente x'_{ijkn} e cada produto $x_{ij}x_{ij}$ por x_{ij}. Devido a propriedade comutativa dos produtos, a restrição adicional (10) é acrescentada. As x'_{ijkn} e x'_{knij} são chamadas de coeficientes complementares.

A seguir, foram aplicados os passos do RLT2: São adicionadas $(N \times (N-1) \times (N-2))^2$ novas restrições (8) e (9) obtidas pela multiplicação das restrições (6) e (7) por cada uma das N^2 variáveis binárias x_{pq}, com i, k, p distintos e j, n, q também distintos, (substituindo os índices i, k e p, e os índices j, n e q). Então, cada produto $x'_{ijkn}x_{pq}$ é substituído por cada variável binária x''_{ijknpq}. Similar a reformulação do RLT1, a restrição (11) deve também ser imposta. Os coeficientes $x''_{ijknpq}, x''_{ijpqkn}, x''_{knijpq}, x''_{knpqij}, x''_{pqijkn}$, e x''_{pqknij} são também chamados de complementares.

Nós representamos a solução dual corrente por um conjunto de matrizes contendo os coeficientes de custos modificados (custos reduzidos) que são mantidos não negativos durante a execução do algoritmo. Os coeficientes $b_{ij}\ \forall\ (i, j = 1, .., N)$ são armazenados em uma matriz B de dimensão $N \times N$, os coeficientes de custo $c_{ijkn}\ \forall\ (i, j, k, n = 1, .., N)$, com $i \neq k$ e $j \neq n$ são armazenados na matriz C de dimensão $N(N-1) \times N(N-1)$, por fim, os coeficientes de custo $d_{ijknpq}\ \forall\ (i, j, k, n, p, q = 1, .., N)$, com i, k, p distintos e j, n, q também distintos, são armazenados na matriz D de dimensão $N(N-1)(N-2) \times N(N-1)(N-2)$

4. O Algoritmo dual de subida para cálculo do limite inferior

O LB (Lower Bound) é o valor resultante da maximização da função objetivo obtida da dualização da formulação do RLT2 aplicado ao QAP (Equações (2), (3) e (5 - 14)). O algoritmo dual de subida implementado neste trabalho consiste em modificar o valor do LB e dos elementos das matrizes B, C e D de modo que nenhum valor se torne negativo e o custo de qualquer solução viável para o QAP permaneça inalterado após a modificação conforme as Equações (2), (3) e (5 - 14). Como consequência desta propriedade, o valor do LB, em qualquer instante da execução do algoritmo, é um limite inferior válido para o custo da solução ótima.

Nossa abordagem para maximizar o LB é transferir custos de D para C, a seguir, de C para B, e por último , de B para LB. Para tal, utilizaremos uma versão modificada do algoritmo dual de subida RLT2 de Adams *et al*. (2007). O Algoritmo 1 consiste basicamente de um loop com 3 operações: Espalhamento de custos, transferência de custos entre complementares e concentração de custos.

Algorithm 1:

inicio
  $LB \leftarrow 0$
  $UB \leftarrow$ *melhor solução conhecida alcançada por uma heurística*
  $K \leftarrow$ *limite mínimo do progresso do* $LB (0,00001 \leq K \leq 1,0)$
  $b_{ij} \leftarrow$ *(se houver, custo de atribuição de* i *em* $j\ \forall\ (i, j)$
  $c_{ijkn} \leftarrow f_{ik} \times d_{jn}\ \forall\ (i, j, k, n)$ *com* $i \neq k$ *e* $j \neq n$
  $d_{ijknpq} \leftarrow 0\ \forall\ (i, j, k, n, p, q)$ *com* i, k, p *distintos e* j, n, q *também distintos*
  $progress \leftarrow 1$
  loop *Enquanto* $(progress >= K)$ *e* $(LB < UB)$
    Distribuição de custos de B *para* C
    Distribuição de custos de C *para* D
    Transferência de custos entre coeficientes complementares de D
    Concentração de custos de D *para* C $(c_{ijkn} \leftarrow Concentrar(d_{ijkn}))$
    Transferência de custos entre coeficientes complementares de C
    Concentração de custos de C *para* B $(b_{ij} \leftarrow Concentrar(c_{ij}))$
    Concentração de custos de B *to* LB' $(LB' \leftarrow Concentrar(B))$
    LB $\leftarrow$ *LB + LB'*
    $progresso \leftarrow (LB'/UB)$
  fim
fin

O Parâmetro K é fornecido no início da execução da aplicação e serve para interromper o loop do dual de subida. O K corresponde ao percentual mínimo que o progresso do LB deve ter. O valor de K não tem um valor definido, depende da instância, dimensão e do histórico de testes, mas está entre 0,01% ($K = 0,00001$) e $100\%(K = 1)$.

4.1. Concentração de custos

A operação de concentração de custos da matriz consiste na transferência de custos da matriz D para a matriz C, da matriz C para a matriz B e da matriz B para o LB, obedecendo as restrições (2 - 3) e (6 - 9). A operação de concentração de custos é tratada como um problema de atribuição linear. Para solucionar cada problema de atribuição linear Adams *et al*. (2007) adota o Algoritmo Húngaro Munkres (1957). O Algoritmo Húngaro tem o objetivo de minimizar o custo total S de uma designação. Tomemos como exemplo uma matriz de custos M de dimensão N^2, onde cada elemento M_{rs} corresponde ao custo de atribuir um recurso r a uma atividade s. A função objetivo fica, então: $min\ S = \sum_r^N \sum_s^N M_{rs}x_{rs}$ onde $x_{rs} \in \{0, 1\}$, $\sum_{r=1}^N x_{rs} = 1\ \forall\ s \in \{1, .., N\}$, $\sum_{s=1}^N x_{rs} = 1\ \forall\ r \in \{1, .., N\}$.

Para a concentração de D para C, considere M uma matriz de dimensão $(N-2)^2$. Para cada (i, j, k, n), M recebe os $(N-2)^2$ elementos de custos da submatriz D_{ijkn}. Para cada $(r, s = 1, .., N-2)$, M_{rs} recebe $d_{(ijkn)pq}\ \forall\ (i, j, k, n, p, q)$, com i, p, k distintos e j, n, q distintos. Obtém-se S a partir da aplicação do algoritmo húngaro em M e adiciona S em c_{ijkn}. Os elementos de custos de $d_{(ijkn)pq}$, $\forall(i, j, k, n, p, q)$, com i, p, k distintos e j, n, q distintos são substituídos pelos correspondentes coeficientes residuais de M. Representamos esta transferência de custos como: $c_{ijkn} \leftarrow Concentrar(D_{ijkn})$. O mesmo será feito para a operação de concentração de C para B, $b_{ij} \leftarrow Concentrar(C_{ij})$, M com a dimensão $(N-1)^2$ e a operação de concentração de B para LB, representado por $LB \leftarrow Concentrar(B)$ e M com a dimensão N^2.

4.2. Espalhamento de custos

A operação de espalhamento de custos é o inverso da concentração de custos e é feita de B para C e de C para D. A operação de espalhamento de custos de B para C consiste em: Para cada (i, j), o elemento de custo b_{ij} é espalhado nas $(N-1)$ linhas da submatriz C_{ij}, ou seja, cada elemento de custo c_{ijkn} recebe um acréscimo de $b_{ij}/(N-1)$, $\forall\ (i, j, k, n)$, com $k \neq i$ e $n \neq j$. Após a atualização de C, $b_{ij} = 0\ \forall\ (i, j)$.

Similar a operação anterior, o espalhamento de custos de C to D consiste em: para cada (i, j, k, n), o elemento de custo c_{ijkn} é espalhado nas $(N-2)$ linhas da submatriz D_{ijkn}, ou seja, cada elemento de custo d_{ijknpq} recebe um acréscimo de $c_{ijkn}/(N-2)$, $\forall\ (i, j, k, n, p, q)$, com i, k, p distintos e j, n, q também distintos. Após a atualização de D, $c_{ijkn} = 0\ \forall(i, j, k, n)$, com $k \neq i$ e $n \neq j$.

4.3. Transferência de custos entre coeficientes complementares

Os coeficientes complementares provem a comunicação de custos entre as submatrizes. Esta operação é essencial para alcançar um bom LB pela aplicação do método RLT. Uma boa escolha da estratégia de transferência de custos permite alcançar LB justos com baixo tempo de execução em algoritmos dual de subida. A operação de transferência de custo entre complementares consiste de incrementar um ou mais coeficientes e decrementar outro(s), mantendo o total de decrementos e incrementos entre complementares sempre igual a 0.

5. Considerações sobre a GPU

A GPU funciona como um co-processador para a CPU executando massivamente cálculos matemáticos. As GPUs são compostas de vários multiprocessadores (ou SM) do tipo *Single Instruction, Multiple Data* (SIMD) e possibilitam a execução de operações em paralelo. Cada SM possui um grupo de de unidades de processamento (ou SP). A NVidia C2070, por exemplo, possui 14 SMs com 32 SPs cada.

O modelo de programação *Compute Unified Device Architecture* (CUDA) permite o desenvolvimento de programas direcionados a aproveitar o potencial das GPUs. As tarefas são submetidas pela CPU (*host*) para a GPU (*device*) através de chamadas com sinalizações definidas pelo

modelo de programação. Essas tarefas são chamadas de *kernel*. Em CUDA cada *kernel* é executado por uma *thread*. Estas por sua vez são agrupadas em Blocos, e estes em Grades. Quando um programa CUDA chama uma grade para ser executado na GPU, cada um dos blocos que compõe essa grade é direcionado para um SM disponível. Ao receber um bloco para executar, o SM divide o bloco em conjuntos de 32 *threads* consecutivas (*warps*). Cada *warp* executa uma única instrução de cada vez. Quando um bloco é finalizado, um novo é atribuído ao SM.

As memória da GPU é organizada em um forma hierárquica composta por 4 níveis distintos: a memória *local* presente em cada SP, a memória *compartilhada* que pode ser acessada por qualquer SP de um mesmo SM, a memória *global* que é visível por qualquer SP e, finalmente, a memória *constante* que é acessível apenas para leitura por todos os SPs. Tais níveis possuem tempo de resposta de 2, 2, 600 e 600 ciclos de clock, respectivamente.

6. Branch-and-bound com o dual de subida RLT2 adaptado para a GPU

O algoritmo *branch-and-bound* baseia-se em uma enumeração implícita de todas as soluções para o problema a ser resolvido. O espaço de potenciais soluções é explorado por construção dinâmica de uma árvore cujo nó raiz representa o problema inicial. Os nós folha são as soluções possíveis e os nós internos são subespaços do espaço de busca total. Cada nó interno da árvore representa um subproblema do problema a ser resolvido. As operações de *Branch* e *Bound* definem uma nova divisão do problema ou o encerramento da sondagem do nó, respectivamente. Na sondagem do cada nó é utilizado como limite inferior o obtido na execução do algoritmo dual de subida apresentado na seção 4, e com limite superior, a melhor solução conhecida da literatura obtida por uma heurística.

A aplicação contendo o *Branch-and-bound* é executada na CPU (*host*), criando *cpu_thread* (utilizamos essa denominação para *threads* triviais com o propósito de diferenciar das *threads* da GPU). A quantidade de *cpu-thread* acompanha o total de GPUs disponíveis. O nó raiz da árvore de *Branch-and-bound* é sondado pela *cpu_thread* de id = 0. Após o 1o. *Branch*, cada *cpu_thread* assume para si um ramo (ou nó) do nível 1 da árvore e faz busca em profundidade. Ao terminar o seu ramo, a *cpu_thread* assume outro ramo que ainda não tenha sido sondado.

As matrizes B, C e D são armazenadas na memória *global* da GPU. Para cada nó sondado a *cpu_thread* executa o algoritmo dual de subida da Seção 4. As operações de concentração de custos, espalhamento de custos e transferência de custos são executadas pela GPU, submetidas para a GPU na forma de *kernels*.

A concentração de custos deverá fazer uso de um algoritmo de atribuição linear, que poderia ser o algoritmo Húngaro, Munkres (1957), amplamente usado devido a sua eficiência. Este algoritmo não permite uma boa paralelização em função das diversas cadeias de decisão em seu código. Optamos neste trabalho em utilizar o Algoritmo do Leilão de Bertsekas e Castanon (1989), também utilizado como algoritmo de atribuição linear e é mais eficiente na paralelização para GPU. cada pessoa (participante) e objeto do algoritmo de Leilão corresponderá, aqui neste trabalho, a facilidade e local do problema original do QAP, respectivamente. A estratégia na paralelização do algoritmo de Leilão está em deixar cada *thread* com a função de uma pessoa do algoritmo.

Na concentração de D para C, a matriz M de dimensão $(N-2)^2$ é posicionada na memória *compartilhada* de cada SM. Para cada (i, j, k, n), os coeficientes de custos da submatriz D_{ijkn} são transferidos para M e um algoritmo de leilão é executado. É designado 1 *warp* (32 *threads*) para cada execução do algoritmo de Leilão. Ao final da execução, os coeficientes resultantes de M, na memória *compartilhada*, são transferidos para D_{ijkn} na memória *global*. Essa disposição é repetida na concentração de C para B e de B para LB.

Optamos em não implementar o compartilhamento da mesma posição de memória para coeficientes complementares, permitido pelas restrições (10) e (11) e utilizado em Adams *et al.* (2007), o que poderia reduzir a memória necessária para armazenar a matriz D. A justificativa está na perda de desempenho da aplicação, já que os coeficientes das matrizes não estariam em blocos contíguos necessitando, assim, de vários ciclos de leitura para transferir da memória *global* para a

matriz M na memória *compartilhada*. Outro motivo, que também ocasiona perda de desempenho, é termos várias *threads*, em *warps* distintos, acessando de forma concorrente a mesma posição na memória, e assim, serializando o acesso a memória *global*.

Nós observamos que cada coeficiente da submatriz $(d_{ijkn})_{pq}$ corresponde ao complementar da submatriz $(d_{knij})_{pq}$. Então, nós introduzimos o conceito de submatrizes complementares onde: para todo $(i, k = i, ..., N)$, e $(j, n = 1, ..., N)$, com $i < k$, as submatrizes $(d_{ijkn})_{pq}$ e $(d_{knij})_{pq}$ são complementares. Utilizando este conceito, nós podemos reduzir o tamanho da matriz D pela metade, como também reduzir pela metade a quantidade de operações de concentração de custos de D para C impactando no tempo de execução da aplicação.

A estratégia de *branch* consiste de um *strong branch* onde é feita uma avaliação prévia de um conjunto de atribuições e concentrações das facilidades ainda não atribuídas a cada local, também não atribuído, e em seguida, é selecionada a unidade (linha ou coluna) que obtiver o maior LB. A concentração utilizada no *strong branch* segue o algoritmo dual de subida RLT1 similar ao da Seção 4, excetuando pelas operações que transferem custos relacionados a matriz D.

7. Implementação e Resultados

O algoritmo proposto, aqui chamado de GPU RLT2, foi implementado usando a linguagem C++ e o modelo de programação CUDA. Os experimentos, exceto pela instância que está em negrito na Tabela 1, foram executados em uma máquina de uso não exclusivo com 2 CPUs Intel Xeon Hexcore (total de 12 núcleos) com 24 GB de RAM e 4 GPUs NVidia Tesla C2070. Cada GPU possui 448 cores de 1,15GHz (14 SMs com 32 SPs cada) e 6 GB DDR5 de memória *global*.

A Tabela 1 expõe os resultados alcançados para diferentes instâncias e tamanhos do QAP. Utilizamos como referência, na avaliação de nossos experimentos, os obtidos pelo algoritmo sequencial dual de subida RLT2 de Adams *et al*. (2007) e pelo algoritmo paralelo RLT1/2/3 Parallel C de Hahn *et al*. (2013).

Novos testes foram feitos com o algoritmo dual de subida RLT2 e seus resultados foram apresentados em Zhu *et al*. (2012), sendo deste último os que constam na Tabela 1. Os experimentos do dual de subida RLT2 de Zhu *et al*. (2012) foram feitos em 2 ambientes computacionais: em 1 CPU Sun Fire E6900 de 1,9GHz, computador este utilizado no processamento das instâncias Nug20, 22, 24 e 25, e em 1 CPU Itanium de 733MHz utilizado nas instâncias Nug28 e 30. O algoritmo dual de subida RLT1/2/3 Parallel C de Hahn *et al*. (2013) empregou 30 dos 64 nós do *Palmetto Supercomputing Cluster* da Universidade de Clemson - EUA, cluster com 2TB de memória compartilhada e cada nó com 1 CPU Intel Xeon de 2,2GHz.

Nossa proposta foi avaliada executando instâncias encontradas no site do QAPLIB (Burkard *et al*. (1997)). Além das instâncias utilizadas em Hahn *et al*. (2013), adicionamos à bateria de testes as instâncias: Tai20a, 20b, 25a, 25b, 30b, Kra30a, Kra30b e Tho30.

A Tabela 1 está assim organizada: Nas 3 primeiras colunas, as informações sobre a instância como: a identificação, o total de facilidades (N) e o valor ótimo encontrado na literatura. As demais colunas apresentam os resultados dos algoritmos já citados. Na 4a, 5a e 6a coluna, a quantidade de nós do *Branch-and-bound*, quantidade de memória principal e o tempo de execução, respectivamente, referentes ao algoritmo dual de subida RLT2. Na 7a, 8a, 9a e 10a coluna, a quantidade de nós do *Branch-and-bound*, o total de nós na etapa do RLT3, a quantidade de memória principal e o tempo de execução, respectivamente. E, finalmente, os resultados obtidos em nosso algoritmo (GPU RLT2), em sequência: o total de nós sondados no *Branch-and-bound*, a quantidade de memória RAM utilizada no *host*, o total de memória *global* utilizada nas GPUs (somadas as memórias individuais das 4 GPUs), e o tempo de relógio da execução.

Comparando os resultados alcançados pelo dual de subida RLT2 e pelo GPU RLT2 observamos que o *speedup* varia de 50 a 300 conforme a instância. Algumas instâncias possuem ramos do B&B bastante desbalanceados fazendo com que GPUs encerrem seus ramos mais cedo que outras. Procedimentos de balanceamento de carga foram implementados, mas os testes não foram finalizados a tempo para esse artigo.

Instância	N	Ótimo	dual subida RLT 2			RLT1/2/3 Parallel C				GPU RLT2			
			Nós	Memória	tempo	Nós		Memória	tempo	Nós	Mem. (GB)		tempo
			B&B	(GB)	(s)	B&B	RLT3	(GB)	(s)	B&B	*Host*	*Device*	(s)
nug20	20	2570	1.407	0,4	2.978	39	9	18,4	736	312	1	1,2	19
nug22	22	3596	1.450	0,8	3.361	52	16	40,7	1.578	130	1	2,4	34
nug24	24	3488	37.099	0,8	5.781	102	16	85,0	5.097	676	1,5	4,8	104
nug25	25	3744	15.497	1,7	124.702	267	31	120,3	23.072	6.151	2	6	551
nug27	27	5234	-	-	-	359	46	231,5	53.227	8.120	3	10,5	1.028
nug28	28	5166	202.295	3,6	2.856.392	1.538	115	321,4	130.527	7.056	4	14	6.083
nug30	30	6124	543.061	6	19.735.563	3.383	251	722,8	733.812	103.380	5	22	88.867
tai20a	20	703482	-	-		-	-	-	-	1.553	1	1,2	123
tai20b	20	122455319	-	-		-	-	-	-	429	1	1,2	29
tai25a	25	1167256	-	-	-	-	-	-	-	324.394	2	6	109.011
tai25b	25	344355646	-	-	-	-	-	-	-	1328	2	6	311
tai30b	30	637117113	-	-	-	sem inf	sem inf	753	315.584	20.205	5	22	19.885
tai35b	**30**	**283315445***	-	-	-					**3.632.310**	**5**	**22**	**792.216**
kra30a	30	88900	-	-	-	-	-	-	-	5.191	5	22	1.104
kra30b	30	91420	-	-	-	-	-	-	-	24.082	5	22	16.552
tho30	30	149936	-	-	-	-	-	-	-	61.841	5	22	51.896

*** Instância aberta na literatura e resolvida de forma exata pela primeira vez**

Tabela 1: Comparação entre os resultados do algoritmo dual subida RLT2 dos testes de Zhu *et al*. (2012), RLT2 1/2/3 Parallel C de Hahn *et al*. (2013) e GPU RLT2 proposto nesse trabalho

Ao avaliar o tempo de execução de cada operação constante do algoritmo dual de subida apresentado na Seção 4, observamos que a operação de *transferência de custos entre complementares da matriz D* registra um tempo de execução comparável à operação de *concentração de custos da matriz D*. Isto é justificável já que a operação de concentração de custos é realizada na memória compartilhada, cujo tempo de resposta é de cerca de 1 a 2 ciclos de clock e o algoritmo do leilão foi implementado buscando aproveitar a transferência coalescente de dados, onde um grupo de *threads* acessa um igual grupo contíguo de dados em um mesmo ciclo de leitura/escrita de memória. A operação de transferência de custos entre complementares consiste basicamente na leitura e escrita na memória global, cujo tempo de resposta é de cerca de 600 ciclos de clock, bem superior ao da memória compartilhada, e tem como agravante a disposição não contígua dos coeficientes complementares, gerando acessos não coalescentes das *threads*, prejudicando a paralelização das operações.

O algoritmo dual de subida RLT1/2/3 Parallel C estende a relaxação até o RLT de nível 3, alcançando limites inferiores mais justos que o RLT de nível 2, entretanto há a necessidade de uma extensa memória de trabalho. A implementação da técnica RLT3 exige, além das matrizes B, C e D presentes no RLT2, uma matriz E de dimensão $(N \times (N-1) \times (N-2) \times (N-3))^2$ impondo, assim, a necessidade de uma elevada quantidade de memória de trabalho para o armazenamento e processamento da matriz E. Com o limite inferior mais justo, as árvores do *branch-and-bound* do RLT1/2/3 Parallel C passam a ser podadas mais cedo e, portanto, um menor número de nós são sondados. Ao compararmos os resultados da nossa proposta com os do RLT1/2/3 Parallel C, comprovamos em números na Tabela 1 o que era evidente, a baixa quantidade de memória de trabalho e um maior número de nós sondados em nossa proposta (GPU RLT2).

Ainda comparando os resultados do GPU RLT2 com os alcançados pelo RLT1/2/3 Parallel C, observamos que o tempo de execução é cerca de 40x menor nas instâncias Nug20, 22, 24, 25 e 27, cerca de 22x menor na Nug28 e cerca de 8,5x menor na Nug30. A redução de desempenho do RLT2 em relação ao RLT3 com o aumento da dimensão do problema do QAP é apontado em Zhu *et al*. (2012). Eles mostram que para instâncias com dimensões superiores a 28 facilidades, as aplicações baseadas na relaxação RLT2 passaram a ter um tempo de execução superior às aplicações com relaxação RLT3.

A Instância *Tai35b*, em negrito na Tabela 1, foi resolvida de forma exata pela primeira vez pelo nosso algoritmo confirmando o valor ótimo alcançado por heurísticas. Para resolver a instância *Tai35b*, o algoritmo GPU RLT2 foi executado em 2 máquinas conectadas a redes distintas, sem exclusividade e com interrupções. Cada máquina possui 2 CPU Xeon com 6 núcleos cada. A primeira máquina hospeda uma GPU NVidia Titan de 16 SMs com 192 SPs cada (total de 3072 SPs de 1088MHz) e 12GB DDR de memória *global*. A segunda hospeda uma GPU NVidia Tesla K40c de 15 SMs com 192 SPs cada (total de 2880 SPs de 745MHz) e também 12GB DDR5 de memória *global*. O desempenho da segunda máquina, durante a execução da aplicação, é cerca de 3 a 4 vezes menor que a primeira. A execução esteve sujeita a interrupções devido a falta de energia e queda da rede, impondo a necessidade de implementação de um procedimento de gravação de *checkpoints* em intervalos de tempo pré-definidos, com o objetivo de armazenar o caminho percorrido da raiz até o nó do B&B no instante da gravação. O tempo apresentado na Tabela 1 corresponde a soma dos tempos individuais de relógio de ambas as máquinas. Na execução da Tai35b, o *gap* alcançado no nó raiz foi de 4,92%, bem inferior ao melhor conhecido constante do site do QAPLIB Burkard *et al*. (1997) que é de 14,52 %.

8. Considerações e trabalhos futuros

As GPUs mais recentes do mercado tendem a ter um número cada vez maior de SPs, no entanto, sua memória não tem acompanhado o mesmo crescimento. Quanto maior a quantidade de SPs, maior o poder computacional e consequente redução no tempo de execução de uma aplicação, este cenário propicia às aplicações com relaxação RLT2 vantagens sobre as aplicações com RLT3, já que uma GPU com 12GB poderá, com auxílio de nosso algoritmo, processar uma instância

com 40 facilidades, mas uma aplicação baseada no RLT3 vai precisar, para a mesma instância de 40 facilidades, não menos do que 2TB de memória principal, memória RAM proibitiva a uma máquina comercial.

A código do algoritmo GPU RLT2 ainda está em processo de revisão, e novas alterações já permitem um desempenho melhor, em especial na execução de instâncias com dimensões superiores a 30 facilidades. Uma versão distribuída está sendo avaliada prometendo bons resultados. Outras contribuições, como uma técnica de economia de memória ampliando a execução de dimensões maiores que 35 facilidades utilizando as mesmas GPUs, também está em teste. Os resultados e as novas contribuições serão apresentadas em artigo futuro.

Agradecimento

Agradecemos ao Professor Esteban Clua do Departamento de Ciência da Computação da Universidade Federal Fluminense e ao Pesquisador Douglas Mota Dias do Departamento de Engenharia Elétrica da PUC-RJ, por disponibilizarem diversas de suas máquinas para executarmos vários dos nossos experimentos, inclusive as que auxiliaram na solução da instância Tai35b.

Referências

Adams, W. P., Guignard, M., Hahn, P. M. e Hightower, W. L. (2007), A level-2 reformulation-linearization technique bound for the quadratic assignment problem. *European Journal of Operational Research*, v. 180, n. 3, p. 983–996.

Adams, W. P. e Sherali, H. D. (1990), Linearization strategies for a class of zero-one mixed integer programming problems. *Operations Research*, v. 38, n. 2, p. 217–226.

Bertsekas, D. e Castanon, D. (1989), The auction algorithm for the transportation problem. *Annals of Operations Research*, v. 20, n. 1, p. 67–96.

Burer, S. e Vandenbussche, D. (2006), Solving lift-and-project relaxations of binary integer programs. *SIAM Journal on Optimization*, v. 16, p. 726–750.

Burkard, R. E., Karisch, S. E. e Rendl, F. (1997), Qaplib - a quadratic assignment problem library. *J. of Global Optimization*, v. 10, n. 4, p. 391–403.

Burkard, R. E., Çela, E., Pardalos, P. M. e Pitsoulis, L. S. The quadratic assignment problem, 1998.

Cela, E. *The Quadratic Assignment Problem: Theory and Algorithms*. Combinatorial Optimization. Springer. ISBN 9780792348788, 1997.

Dickey, J. e Hopkins, J. (1972), Campus building arrangement using topaz. *Transportation Research*, v. 6, p. 59–68.

Elshafei, A. N. (1977), Hospital layout as a quadratic assignment problem. *Journal of The Operational Research Society*, v. 28, n. 1, p. 167–179.

Francis, R. L., McGinnis, L. F. e White, J. A. *Facility layout and location: an analytical approach*. Prentice Hall, Englewood Cliffs, NJ, 1992.

Gonçalves, A., Drummond, L., Pessoa, A. e Hahn, P. (2013), Improving lower bounds for the quadratic assignment problem by applying a distributed dual ascent algorithm. v. .

Hahn, P., Roth, A., Saltzman, M. e Guignard, M. (2013), Memory-aware parallelized rlt3 for solving quadratic assignment problems.

Hahn, P. M. e Grant, T. (1998), Lower bounds for the quadratic assignment problem based upon a dual formulation. *Operations Research*, v. 46, n. 6, p. 912–922.

Heffley, D. R. (1980), Decomposition of the koopmans-beckmann problem. *Regional Science and Urban Economics*, v. 10, n. 4, p. 571–580.

Hubert, L. *Assignment methods in combinatorial data analysis*. M. Dekker, New York, N.Y. ISBN 0824776178 9780824776176, 1986.

Koopmans, T. C. e Beckmann, M. (1957), Assignment problems and the location of economic activities. *Econometrica*, v. 25, n. 1, p. 53–76.

Munkres, J. (1957), Algorithms for the assignment and transportation problems. *Journal of the Society of Industrial and Applied Mathematics*, v. 5, n. 1, p. 32–38.

Padberg, W. e Rijal, P. *Location, Scheduling, Design and Integer Programming*. Kluwer Academic Publishers. ISBN 978-1-4612-8596-0, 1996.

Pardalos, P. M., Rendl, F. e Wolkowicz, H. The quadratic assignment problem: A survey and recent developments. *In Proceedings of the DIMACS Workshop on Quadratic Assignment Problems, volume 16 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, p. 1–42. American Mathematical Society, 1994.

Sherali, H. e Adams, W. Reformulation-linearization techniques for discrete optimization problems. Du, D.-Z. e Pardalos, P. (Eds.), *Handbook of Combinatorial Optimization*, p. 479–532. Springer US, 1999.

Tsutsui, S. e Fujimoto, N. Solving quadratic assignment problems by genetic algorithms with gpu computation: A case study. *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, GECCO '09, p. 2523–2530, New York, NY, USA. ACM. ISBN 978-1-60558-505-5, 2009.

Tsutsui, S. e Fujimoto, N. Aco with tabu search on a gpu for solving qaps using move-cost adjusted thread assignment. *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, p. 1547–1554, New York, NY, USA. ACM. ISBN 978-1-4503-0557-0, 2011.

Zhu, Y.-R., Hahn, P. M., Guignard, M., Hightower, W. L. e Saltzman, M. J. (2012), A level-3 reformulation-linearization technique-based bound for the quadratic assignment problem. *INFORMS J. on Computing*, v. 24, n. 2, p. 202–209.