ABORDAGENS PARALELAS PARA O MÉTODO HÍBRIDO CLUSTERING SEARCH

Felipe Mendes Miranda
Universidade Federal de São Paulo
São José dos Campos - SP
felipe_mendes76@hotmail.com

Álvaro Luiz Fazenda
Universidade Federal de São Paulo
São José dos Campos - SP
alvaro.fazenda@unifesp.br

Antônio Augusto Chaves
Universidade Federal de São Paulo
São José dos Campos - SP
antonio.chaves@unifesp.br

RESUMO

O *Clustering Search* (CS) é um método híbrido que procura combinar metaheurísticas e heurísticas de busca local, de tal forma que a busca seja intensificada somente em regiões promissoras do espaço de soluções. Neste trabalho propõe-se duas versões paralelas do CS para resolver o Problema de Agrupamento Centrado Capacitado (CCCP). O CCCP consiste em particionar um conjunto de n pontos em p grupos disjuntos com capacidade limitada. A cada ponto está associado um valor de demanda e o objetivo é minimizar a soma das distâncias euclidianas entre os pontos e os seus respectivos centros geométricos. A primeira abordagem paralela faz uso de OpenMP e a segunda de MPI. Ambas as abordagens fazem a paralelização do componente de busca local do CS. Os resultados computacionais mostram que a paralelização do CS é uma estratégia eficiente em termos de tempo computacional e eficiência.

PALAVARAS CHAVE. Metaheurística. OpenMP. MPI.

Área principal: Metaheurísticas

ABSTRACT

The Clustering Search (CS) is a hybrid method that tries to combine metaheuristics and local search heuristics, so that the search is intensified only in promising regions of the solution space. In this paper we propose two parallel versions of CS to solve the Capacitated Centered Clustering Problem (CCCP). The CCCP is to partition a set of n points into p disjoint groups with limited capacity. Each point is associated with a demand value and the objective is to minimize the sum of the Euclidean distances between points and their respective geometric centers. The first parallel approach makes use of OpenMP and the second of MPI. Both approaches make parallelization of the local search component of CS. The computational results show that the CS parallelization is an effective strategy in terms of computational time and efficiency.

KEYWORDS. Metaheuristic. OpenMP. MPI.

Main area: Metaheuristics

1. Introdução

Nas últimas décadas, as meta-heurísticas têm-se apresentado como ferramentas interessantes para produzir, em tempo razoável, soluções de boa qualidade para problemas de otimização combinatória. Sendo que, o conceito de meta-heurísticas híbridas vem ganhado cada vez mais importância como alternativa para melhorar o desempenho das meta-heurísticas. Vários tipos de integrações são possíveis, dando origem a algoritmos eficazes e eficientes.

Uma combinação muito utilizada é a aplicação de heurísticas de buscas locais específicas para o problema abordado para intensificar a busca nas soluções encontradas pelas meta-heurísticas. Entretanto, heurísticas de busca local geralmente possuem um custo computacional muito alto, o que inviabiliza a sua utilização de forma indiscriminada.

Sendo assim, uma técnica interessante é a aplicação de busca local somente em regiões consideradas promissoras. Uma estratégia para identificar o potencial positivo de subespaços de busca é por meio da quantidade de soluções geradas nessas regiões, uma vez que as meta-heurísticas tendem a privilegiar as melhores soluções. Portanto, estabelece-se que regiões nas quais muitas soluções são geradas tendem a serem regiões promissoras, isto é, regiões que contenham as melhores soluções.

Neste contexto, Oliveira *et al*. (2013) propõe o método híbrido *Clustering Search* (CS). No CS uma meta-heurística é responsável pela exploração do espaço de busca, gerando soluções representativas de regiões promissoras. Em conjunto com a meta-heurística, heurísticas de buscas locais específicas ficam responsáveis por realizar uma intensificação da busca em tais regiões, assim que possível.

O CS tem obtido sucesso quando aplicado a diversos problemas de otimização combinatória encontrados na literatura: Nagano *et al*. (2006), Chaves *et al*. (2007,2009), Biajoli e Lorena (2007), Chaves e Lorena (2010), Ribeiro *et al*. (2011), Oliveira *et al*. (2012), entre outros.

Estudos realizados em Chaves (2009) mostram que a busca local é o componente mais custoso do CS em termos de tempo computacional. Apesar de heurísticas de buscas locais serem aplicadas com mais critério neste método, estas ainda representam de 80 a 95% do tempo total de execução do CS. Por outro lado, o CS é composto por procedimentos que podem ser divididos em blocos que possuem pouca dependência entre si, principalmente na busca local.

Sendo assim, este trabalho propõe duas abordagens paralelas para o CS visando diminuir seu tempo de execução sem que haja perda de qualidade das soluções. Nestas versões também busca-se manter a eficiência do processamento em relação ao tempo de execução. Para realizar a paralelização do CS utiliza-se duas técnicas de programação paralela: *Open Multi-Processing* (OpenMP) e *Message Passing Interface* (MPI). OpenMP é uma interface para programação *multi-threading* de memória compartilhada e MPI é um padrão para comunicação de dados em programação paralela que utiliza uma arquitetura de memória distribuída. Para avaliar o desempenho do CS paralelo realiza-se um estudo de caso no qual é utilizado o método CS implementado para resolver o Problema de Agrupamento Centrado Capacitado (CCCP, do inglês *Capacited Centered Clustering Problem*) (ver Chaves e Lorena (2010) e Oliveira *et al*. (2013)). Os testes computacionais apresentam uma melhoria significativa no tempo computacional nas abordagens paralelas em relação à abordagem sequencial, mantendo a eficiência do método CS.

O restante do trabalho está organizado como se segue. A seção 2 apresenta o método CS e as técnicas de programação paralela OpenMP e MPI. Na seção 3 faz-se uma descrição do CCCP e do CS aplicado ao CCCP. Na seção 4 são discutidas as abordagens paralelas propostas para o CS aplicado ao CCCP. A seção 5 explana os resultados computacionais e a seção 6 as considerações finais.

2. Material e métodos

Nesta seção são apresentados o método híbrido CS e as técnicas de programação paralela OpenMP e MPI.

2.1 *Clustering Search*

O *Clustering Search* (CS) (Oliveira *et al*., 2013) é um método híbrido que busca combinar metaheurísticas e heurísticas de busca local, em que a busca é intensificada somente em regiões do espaço de busca que merecem atenção especial (regiões promissoras). O CS introduz uma inteligência e prioridade para a escolha de soluções para aplicar a busca local, em vez de escolher aleatoriamente ou aplicar busca local em todas as soluções. Consequentemente é esperado uma melhora no processo de convergência com uma diminuição no esforço computacional, pois há uma aplicação mais racional das heurísticas.

O CS se atém a localizar áreas de busca promissora construindo-as em *clusters*. Um *cluster* é definido por um centro, *c*, que é, geralmente, inicializado aleatoriamente e, posteriormente, tende progressivamente a pontos promissores no espaço de busca. O número de *clusters*, *NC*, deve ser fixado a priori.

O CS pode ser explicitado em quatro partes conceitualmente independentes: a metaheurística (SM), o componente de agrupamento (IC), um módulo de análise (AM) e a busca local (LS). A Figura 1 apresenta o pseudocódigo do CS.

```
algoritmo CS
    crie os clusters iniciais
    { meta-heurística - SM }
    enquanto ( critério de parada não for satisfeito ) faça
        gere uma solução (s_k) pela meta-heurística
        { processo de agrupamento - IC }
        encontre o cluster mais similar a s_k ( C_j | d(s_k,c_j) ≡ min { d(s_k,c) } )
        agrupe s_k no cluster mais similar C_j  ( δ_j ← δ_j + 1 )
        atualize o centro do cluster ( c_j ← assimilação (c_j, s_k) )
        { modulo de análise - AM }
        se (δ_j ≥ λ ) então
            reduza o volume δ_j ← 0
            { heurística de busca local - LS }
            encontre o melhor vizinho ( ĉ_j )  de  c_j
            se ( f ( ĉ_j ) < f ( c_j ) ) então
                atualize o centro  c_j ← ĉ_j
            fim-se
        fim-se
    fim-enquanto
fim-algoritmo
```

Figura 1 - Pseudocódigo do CS
Fonte: Adaptado de Oliveira *et al*. (2013)

O componente SM pode ser implementado por qualquer algoritmo de otimização que gera soluções diversificadas do espaço de busca. Ela trabalha como um gerador de soluções, explorando o espaço de busca através de uma manipulação de um conjunto de soluções, de acordo com sua estratégia de busca específica.

O componente IC procura reunir soluções similares dentro de grupos, mantendo um centro de *cluster* representativo delas. Uma métrica de distância, Δ, é definida, a priori, permitindo uma medida de similaridade para o processo de agrupamento. Por exemplo, em um problema de otimização combinatória, a similaridade pode ser definida como o número de movimentos necessários para alterar uma solução até o centro do *cluster* (Oliveira e Lorena, 2007).

O processo de assimilação é aplicado no centro mais próximo c_j, considerando a nova solução gerada s_k. O método *Path-relinking* (Glover *et al.*, 2000) pode ser usado para gerar uma série de pontos, memorizando o melhor ponto avaliado para ser o novo centro.

O componente AM examina cada *cluster*, em intervalos regulares, indicando um provável *cluster* promissor. Uma densidade de *cluster*, δ_j, é uma medida que indica o nível de atividade dentro do *clusterj*. Para simplificar, δ_j pode contar o número de soluções geradas pelo SM e agrupada em c_j. Quando δ_j atinge um limitante λ, indicando que certo padrão de soluções estão sendo gerados por SM, a região deste *cluster* é melhor investigada para acelerar o processo de convergência.

Por último, o componente LS é um módulo de pesquisa interno que provém a exploração de uma região supostamente promissora, representada por um *cluster*, intensificando a busca nessa região, aplicando uma busca local no centro desse *cluster*.

2.2 OpenMP

O OpenMP (do inglês *Open Multi-Processing*) (Chapman *et al*., 2007) é uma API para programação paralela explícita que faz uso de sistemas de memória compartilhada. OpenMP pode ser usado tanto em computadores comerciais quanto em supercomputadores, sendo constituído por diretivas de compilação, biblioteca de funções e variáveis de ambiente.

O uso do OpenMP possibilita desenvolver versões de algoritmos clássicos, ou mesmo novas propostas, usando programação paralela e explorando quantos processadores estiverem disponíveis.

A estrutura básica do OpenMP é a definição de uma seção paralela do programa. Nesta seção pode-se utilizar mais de um processador para realizar a computação. Assim, as tarefas da seção são distribuídas em processos (*threads*) que executam suas operações. Uma seção paralela chega ao fim quando todos os *threads* que estão sendo utilizados terminam suas tarefas, e posteriormente o programa continua sua execução de forma sequencial.

Uma vez que o programador tenha identificado a parte do código que deseja paralelizar, com ênfase na paralelização de ciclos, o trabalho é dividido por *threads* de execução que podem partilhar ou duplicar variáveis. A comunicação entre *threads* é realizada através de variáveis partilhadas.

O OpenMP baseia-se em um modelo de execução tipo *fork-join*. Neste modelo inicia-se a execução com um processo (*master thread*) e no início do construtor paralelo é criado um "time de *threads*". Ao completar as tarefas este time de *threads* é sincronizado em uma barreira implícita e apenas o *master thread* continua a execução. A Figura 2 ilustra o modelo de execução *fork-join* do OpenMP.

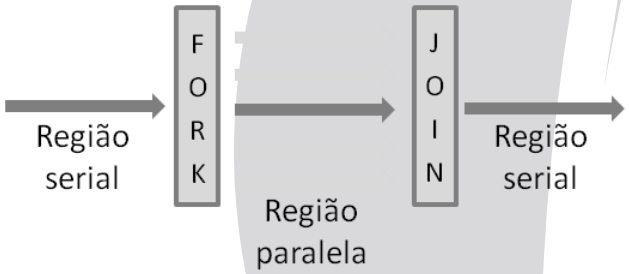

Figura 2 - Modelo de execução *fork-join* do OpenMP

2.3 MPI

O MPI (do inglês *Message Passing Interface*) (MPI Forum, 2012) é um padrão para comunicação de dados em programação paralela que faz uso de uma arquitetura de memória distribuída. O MPI permite que um mesmo programa seja executado simultaneamente em computadores interconectados, utilizando para isto o paradigma de troca de mensagens.

A Figura 3 ilustra o modelo de troca de mensagem do MPI. Neste modelo assume-se que cada *hardware* é uma coleção de processadores, cada um com sua própria memória interna, e uma rede de interconexão que possibilita a troca de mensagens entre os processadores.

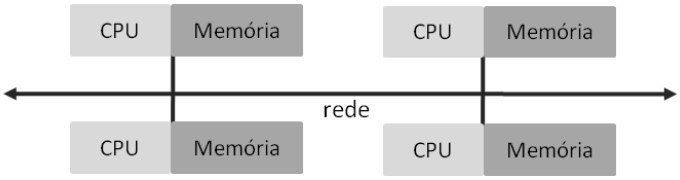

Figura 3 - Modelo de troca de mensagem do MPI

Apesar de todas as facilidades do padrão, o desenvolvedor é o responsável pela identificação e definição das seções paralelas e pelo dimensionamento de carga entre os computadores participantes. Com o objetivo de aumentar o desempenho deve-se realizar o menor número de trocas de mensagens possível, visando diminuir o impacto do tempo despendido na distribuição e consolidação dos dados no tempo total de execução.

3. CS aplicado ao CCCP

Problemas de localização de facilidades possuem várias aplicações práticas, tais como, localização de escolas, postos de saúde, pontos de ônibus, corpo de bombeiros, redes de transmissão, depósitos, entre outros.

Um dos problemas de localização de facilidades mais estudados é o problema de *p*-medianas (Hakimi, 1964). Esse problema consiste em localizar *p* facilidades (medianas) em um dado espaço, satisfazendo *n* pontos de demanda de forma em que a soma das distâncias entre cada ponto de demanda e sua mediana mais próxima seja minimizada.

O problema de *p*-medianas pode ser generalizado definindo-se uma capacidade fixa para cada mediana candidata e uma demanda para cada ponto de atendimento, sendo que, a soma das demandas de todos os pontos alocados a uma mediana não pode exceder sua capacidade. Essa generalização é conhecida como Problema de *p*-Medianas Capacitado (CPMP, do inglês *Capacited p-Median Problem*) (Mulvey e Beck, 1984).

O Problema de Agrupamento Centrado Capacitado (CCCP, do inglês *Capacited Centered Clustering Problem*), proposto por Negreiros e Palhano (2006), é similar ao CPMP. A diferença desse problema está no fato de existir, em vez de medianas, um centróide para cada agrupamento de pontos de demanda. Os centróides são calculados a partir da média das coordenadas dos pontos de demanda alocados no agrupamento.

Chaves e Lorena (2010) propuseram utilizar o método híbrido CS para resolver o CCCP, usando a metaheurística Algoritmo Genético (Holland, 1975) como gerada de soluções. Entretanto, devido às dimensões das instâncias disponíveis na literatura e a complexidade de solução do problema, o método proposto apresenta tempos computacionais altos. Sendo assim, torna-se interessante o uso de programação paralela para obter a diminuição destes tempos.

Uma solução do CCCP pode ser representada por um vetor de valores inteiros com os *n* pontos de demanda. O valor de cada posição *i* do vetor representa o agrupamento ao qual o ponto *i* está alocado. A Figura 4 ilustra um exemplo de uma solução com 7 pontos de demanda e 3 agrupamentos. Os agrupamentos neste exemplo são {(1,2), {3,4,6}, (5,7)}.

Pontos	*1*	*2*	*3*	*4*	*5*	*6*	*7*
	1	1	2	2	3	2	3

Figura 4 - Exemplo da representação de uma solução do CCCP

No cálculo da função objetivo do CCCP é preciso determinara localização dos centróides, calculando a média das coordenadas dos pontos de demanda atribuídos a cada agrupamento. E então, calcula-se a distância euclidiana entre os centróides e os pontos de

demanda alocados a cada agrupamento e a demanda atendida dos agrupamentos. Essa característica faz com que a avaliação da função objetivo tenha um custo computacional muito alto. Caso haja soluções inviáveis, isto é, soluções que extrapolem a capacidade dos agrupamentos. A função objetivo precisa ser penalizada nestas soluções.

O método híbrido CS precisa de uma métrica de distância para calcular a similaridade entre soluções. Neste trabalho define-se a medida de distância entre duas soluções do CCCP como sendo o número de pontos de demanda atribuídos para agrupamentos diferentes nestas soluções.

O CS utiliza o Algoritmo Genético clássico como componente SM para gerar soluções para o processo de agrupamento. O Algoritmo Genético possui três operadores: seleção, cruzamento e mutação. A população inicial é gerada com soluções aleatórias, na qual cada ponto de demanda é alocado aleatoriamente a um agrupamento. O operador de seleção define os pais que participarão do processo de cruzamento por meio do método do torneio, privilegiando os pais mais bem adaptados. O cruzamento uniforme (Syswerda, 1989) é utilizado para combinar pares de pais e gerar a população de filhos. Por fim, o operador de mutação realiza uma troca aleatória de agrupamento para um ponto demanda também selecionado aleatoriamente. A Figura 5 mostra um exemplo destes operadores.

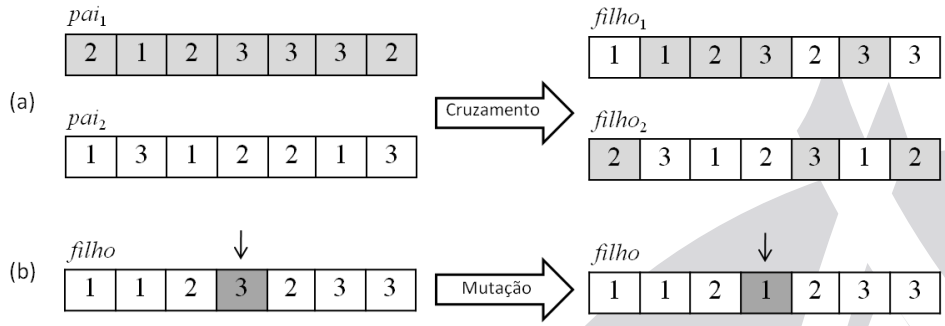

Figura 5 - (a) Exemplo de um cruzamento uniforme para o CCCP; (b) Exemplo de uma mutação para o CCCP

A cada iteração do CS, um filho s_k é agrupado ao *cluster j* mais próximo (*cluster* com a menor distância entre seu centro e o filho s_k). O volume deste *cluster* é incrementado em uma unidade e o centro do *cluster* precisa ser atualizado com novos atributos de s_k. Neste processo de assimilação é utilizado o método *Path-Relinking* (Glover, 1996). Sendo que, o novo centro será a melhor solução encontrada no caminho entre o centro c_j e a solução s_k. Para o CCCP, um movimento no *Path-Relinking* consiste em trocar o agrupamento no qual um ponto de demanda está alocado. A Figura 6 apresenta um exemplo do *Path-Relinking* aplicado ao CCCP.

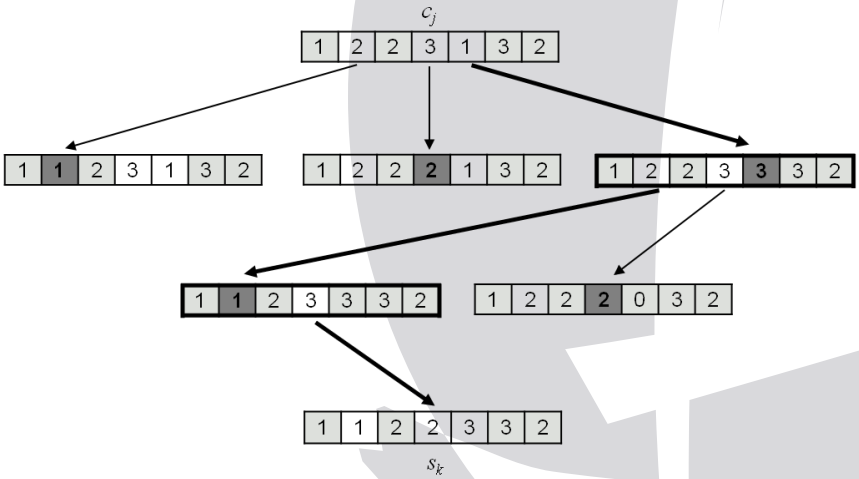

Figura 6 - Exemplo de um *Path-Relinking* aplicado ao CCCP

A busca local é acionada sempre que um *cluster* for considerado promissor, intensificando a busca na região do centro c_j. Chaves e Lorena (2010) utilizaram o método Descida em Vizinhança Variável (VND, do inglês *Variable Neighborhood Descent*) (Mladenovic e Hansen, 1997), no qual são utilizadas duas heurísticas de descida: Transferir-Ponto e Trocar-Pontos. A heurística Transferir-Ponto examina todos os movimentos de transferir um ponto de um agrupamento para outro agrupamento. Enquanto que a heurística Trocar-Pontos examina todos os movimentos de trocar dois pontos de agrupamentos diferentes. Se uma solução melhor for encontrada retorna-se para a primeira heurística e continua-se a busca a partir da nova solução. O VND se encerra quando nenhuma melhora na solução corrente puder ser obtida por meio destas heurísticas.

4. Abordagens paralelas para o CS

Nesta seção apresenta-se as duas abordagens paralelas propostas para o CS. A primeira abordagem consiste em paralelizar a busca local (componente LS) utilizando OpenMP, visto que esta tarefa consome um tempo de processamento significativo no CS. Na segunda abordagem utiliza-se MPI para fazer a comunicação entre diferentes processadores e propõe-se uma modificação na estrutura do CS para que os componentes SM, IC e AM sejam executados em um processador e o componente LS seja executado em paralelo por outros processadores.

4.1 Paralelização usando OpenMP no componente de busca local

Nesta abordagem a execução do método CS continua serial, apenas o procedimento de busca local (componente LS) é paralelizado. Sendo assim, quando a busca local for realizada em um *cluster* promissor, cria-se uma região paralela e as tarefas da busca local são divididas entre os processos (*threads*).

As heurísticas implementadas no componente LS não possuem dependências de dados. Desta forma, as tarefas de gerar vizinhos e calcular a função objetivo das heurísticas são particionadas e distribuídas para os processos disponíveis.

As Figuras 7 e 8 apresentam os pseudo-códigos das heurísticas Trocar-Alocação e Trocar-Pontos com a paralelização utilizando OpenMP, respectivamente. Observa-se que apesar de não haver dependência de dados, há uma região de seção crítica em que apenas um processo pode ser executado por vez. Esta seção é gerada no momento de verificar se uma solução vizinha é a melhor solução obtida até o momento.

```
#pragma omp parallel
algoritmo Transferir-Ponto ( s )
    #pragma omp for private (i, s', j)
    para ( i de 1 até n ) faça
        para ( j de 1 até p) faça
            s' ←s
            se ( ponto i não é atendido pelo agrupamento j ) então
                s'[i] ←j
                f (s') = CalcularFO(s' );
                #pragma omp critical
                se( f (s') <f (s) ) então
                    s←s'
            fim-se
        fim-para
    fim-para
retornar s
```

Figura 7 - Pseudo-código paralelo da heurística Transferir-Ponto

```
#pragma omp parallel
algoritmoTrocar-Pontos ( s )
    #pragma omp for private (i, s', j)
    para ( i de 1 até n ) faça
        para ( j de 1 até n) faça
            s' ←s
            se ( pontos i e j estão em agrupamentos diferentes ) então
                Trocar s'[i] e s'[j]
                f (s') = CalcularFO(s' );
                #pragma omp critical
                se( f (s') <f (s) ) então
                    s←s'
            fim-se
        fim-para
    fim-para
retornar s
```

Figura 8 - Pseudo-código paralelo da heurística Trocar-Pontos

4.2 CS paralelo utilizando MPI

Nesta abordagem utilizou padrão de comunicação MPI para tornar o método CS paralelo. O objetivo é que a metaheurística (componente SM) e o processo de agrupamento (componentes IC e AM) sejam executados de forma serial e apenas quando um *cluster* se tornar promissor e a busca local (componente LS) é ativada seleciona-se outro processador para executar a intensificação da busca na região do centro deste *cluster*. A execução dos demais componentes do CS continua, mesmo sem o resultado da busca local. Desta forma, utiliza-se o termo "Mestre" para o processador que executa a parte serial do CS e "Escravos" para os processadores que executarem a busca local.

Essa proposta implica em uma modificação na versão serial do CS. Neste, o programa só prossegue após a conclusão da busca local e atualização do centro do *cluster*. No CS usando MPI, o programa continua a execução após delegar a tarefa de executar a heurística de busca local para outro processador. Desta forma, podem acontecer buscas em centros "antigos", ou seja, centros que ainda não foram atualizados pelos resultados das heurísticas de busca local. Entretanto, não há perdas de dados, uma vez que o centro sempre é atualizado com a melhor solução encontrada no *cluster* independente do processador que executou a busca local.

A cada iteração o Mestre analisa se algum processo Escravo terminou sua tarefa. Caso um ou mais tenham terminado, o Mestre recebe a(s) solução(es) ótima(s) local(is) encontradas pelos Escravos e atualiza os centros dos *clusters*.

Se um *cluster* se torna promissor e todos os Escravos estão ocupados, o Mestre espera o término da tarefa de, pelo menos, um Escravo. Este procedimento é utilizado para que as informações armazenadas nos centros dos *clusters* não fiquem muito tempo desatualizado.

A Figura 9 mostra uma representação do modelo Mestre-Escravo para o CS paralelo usando MPI. Nesta abordagem, sempre que um *cluster* se tornar promissor ($\delta_j > \lambda$) o processo Mestre envia o centro deste *cluster* para um processo Escravo realizar a busca local nesta região. O processo Mestre continua sua execução após o envio dos dados e verifica se algum Escravo retornou o resultado da busca local para atualizar os centros dos *clusters*.

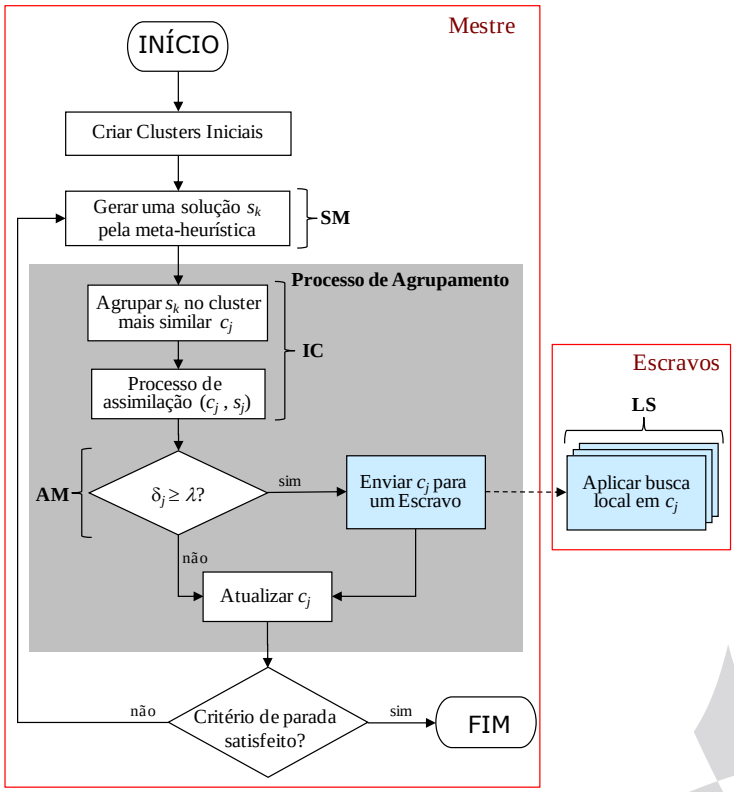

Figura 9 – Representação Mestre-Escravo para o CS usando MPI

5. Experimentos computacionais

O CS para o CCCP foi codificado em C++ e os experimentos foram conduzidos em um PC com processador Intel *core 2 quad* Q8400, cache L2 de 4 MB e memória de 2 GB de RAM sob plataforma Ubuntu. Para verificar a eficiência do CS utiliza-se três instâncias disponíveis na literatura (SJC3a – 300 pontos e 25 agrupamentos, SJC4b – 400 pontos e 40 agrupamentos, e Doni2 – 2000 pontos e 6 agrupamentos) e faz-se uma comparação das soluções obtidas com as versões serial e paralela com as melhores conhecidas para estas instâncias.

Foram utilizadas duas métricas para medir o desempenho do CS paralelo:

$$Speedup(P) = T\,(1 \text{ processador}) \,/\, T\,(P \text{ processadores}) \quad (1)$$

$$\text{Eficiência}(P) = Speedup(P) \,/\, P \quad (2)$$

sendo, T o tempo de execução do CS e P o número de processadores

As Tabelas 1 e 2 apresentam os resultados dos tempos computacionais para as versões de CS paralelas usando OpenMP e MPI. As colunas *T* representam o tempo de execução total, em minutos, do CS, as colunas *Sp* (*speedup*) indicam quantas vezes o tempo de execução da versão paralela foi mais rápido que a versão serial, as colunas *Ef* (eficiência) apresentam a eficiência na redução de tempo em função do número de processadores utilizados.

Tabela1 – Resultados do CS usando OpenMP

	Número de processadores											
	1			2			3			4		
Instância	*T*	*Sp*	*Ef*	*T*	*Sp*	*Ef*	*T*	*Sp*	*Ef*	*T*	*Sp*	*Ef*
SJC3a	4,29	1x	100%	2,50	1,5x	75%	2,22	1,8x	60%	1,58	2,2x	55%
SJC4b	17,09	1x	100%	11,51	1,4x	70%	9,45	1,7x	56%	8,47	1,9x	47%
Doni2	35,47	1x	100%	30,44	1,1x	55%	26,38	1,3x	43%	19,23	1,8x	45%
média	18,95			14,81		66%	12,68		53%	9,76		49%

Observando os resultados obtidos na Tabela 1, nota-se que o uso da ferramenta OpenMP gera uma diminuição no tempo computacional do CS, principalmente na maior instância Doni2. É importante salientar que esta ferramenta faz uso de memória compartilhada e utiliza apenas os processadores de uma mesma máquina.

Nos testes com MPI foram utilizados computadores com a mesma configuração descrita anteriormente e para a comunicação entre os computadores uma Rede Gigabit Ethernet. Os resultados destes testes com até 8 computadores (32 processadores) são apresentados na Tabela 2.

Tabela 2 – Resultados do CS usando MPI

	Número de computadores / processadores											
	1/1			2 / 8			4 / 16			8 / 32		
Instância	*T*	*Sp*	*Ef*	*T*	*Sp*	*Ef*	*T*	*Sp*	*Ef*	*T*	*Sp*	*Ef*
SJC3a	4,29	1x	100%	0,46	5,8x	72%	0,28	9,6x	60%	0,19	14,1x	44%
SJC4b	17,09	1x	100%	2,58	5,7x	71%	2,33	6,7x	42%	2,24	7,1x	22%
Doni2	35,47	1x	100%	4,06	8,7x	100%	2,19	15,4x	96%	1,13	29,4x	92%
média	18,95			2,36		81%	1,60		66%	1,18		52%

Observa-se na Tabela 2 que o CS usando MPI apresenta uma diminuição no tempo computacional ainda mais significativa. Para a instância Doni2 o tempo de execução diminui de 35 minutos para 1 minuto (utilizando 32 processadores), ou seja, uma redução de 97% do tempo necessário para executar todo o processo de busca do CS.

A diferença nas taxas de eficiência entre as instâncias se deve às porcentagens do tempo de execução das buscas locais, que são as regiões paralelizadas do código, em cada instância. Por exemplo, na instância SJC4b (com capacidade menos apertada) gasta-se menos tempo computacional de busca local que na instância SJC3a (com capacidade mais apertada). Assim, quanto maior for o tempo de busca local em relação ao restante do programa, mais tempo será destinado ao processamento paralelo gerando uma eficiência maior nos resultados.

As abordagens com OpenMP e MPI se diferem pelo momento no qual é realizado o paralelismo do código. No OpenMP realiza-se paralelismo dentro dos métodos de busca local. Assim, quando uma busca local é executada divide-se o processamento desta em vários *threads*. Enquanto que, no MPI paraleliza-se a chamada ao método de busca local. Desta forma, várias buscas podem ser realizadas em paralelo (uma em cada processo).

Esta diferença entre as abordagens explica o baixo desempenho usando OpenMP. Uma vez que, em OpenMP realiza-se muitas aberturas de seções paralelas de curta duração e pouca computação. O tempo gasto para criar os *threads* interfere na escalabilidade do método.

Além da eficiência no tempo computacional, espera-se que a versão paralela do CS seja capaz de obter boas soluções para o CCCP. Desta forma, a Tabela 3 compara a média das soluções obtidas pelo CS paralelo (versão usando MPI e 32 processadores) em 10 execuções do algoritmo com as melhores soluções conhecidas na literatura para estas instâncias (Muritiba *et al*. 2012) e a versão serial do CS.

Tabela 3 – Comparação das soluções médias usando CS serial e paralelo

Instância	Melhor solução	CS (versão serial)	CS (versão paralela com 32 processadores)
SJC3a	45356,35	45383,55	45445,91
SJC4b	52202,48	52258,67	52317,21
Doni2	6080,70	6374,40	6374,27
média	34546,51	34672,21	34712,46

Observa-se na Tabela 3 que a qualidade das soluções médias obtidas na versão paralela do CS está bem próxima da versão serial do CS. Ambas as versões encontraram a melhor solução conhecida na literatura ao menos uma vez em 30 execuções. Além disso, o CS se mostra robusto encontrando soluções médias próximas as melhores soluções conhecidas na literatura.

6. Conclusão

Neste trabalho foram propostas duas versões paralelas para o método *Clustering Search* (CS) para resolver o Problema de Agrupamento Centrado Capacitado (CCCP). A primeira versão paralela do CS faz uso de OpenMP (utilizando processadores com memória compartilhada) e a segunda versão faz uso de MPI (utilizando uma rede de computadores sem memória compartilhada).

A ideia do CS é evitar a aplicação de heurísticas de busca local em todas as soluções geradas por uma meta-heurística. Assim, o método busca detectar regiões promissoras no espaço de busca e aplica busca local somente nessas regiões. Ainda assim, o tempo computacional gasto nestas buscas locais representa uma parte considerável do tempo total de execução do CS. Desta forma, neste trabalho optou-se por paralelizar apenas o componente LS do método CS.

Este trabalho reporta os resultados encontrados pelo CS (versões serial e paralela) sobre instâncias do CCCP disponíveis na literatura. As versões paralelas do CS foram capazes de diminuir significativamente o tempo computacional do método, apresentando boas medidas de *speedup* e eficiência de acordo com o aumento no número de processadores. Além disso, as soluções obtidas pelas versões paralelas do CS são comparáveis com as soluções da versão serial e também com as melhores soluções conhecidas na literatura.

Como trabalhos futuros propõe-se melhorar a eficiência computacional da versão do CS com OpenMP e integrá-la à versão com MPI. Para melhorar a qualidade dos resultados pretende-se testar outras heurísticas de busca local para o CCCP. Além de realizar testes com instâncias maiores.

Agradecimentos

Este trabalho é financiado pela Fundação de Amparo à Pesquisa do Estado de São Paulo - FAPESP (Processo nº 2012/17523-3) e pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq (Processo nº 482170/2013-1, Processo nº 304979/2012-0).

Referências bibliográficas

Biajoli, F. L. e Lorena L.A.N. (2007), Clustering Search Approach for the Traveling Tournament Problem. ***LNAI***, 4827: 83–93.

Chapman, B., Jost, G., Van Der Pas, R. *Using OpenMP: Portable Shared Memory Parallel Programming.* The MIT Press, 384 pp., 2007.

Chaves, A.A. (2009). Uma meta-heurística híbrida com busca por agrupamentos aplicada a problemas de otimização combinatória. *Tese de doutorado em Computação Aplicada*, INPE, São José dos Campos, Brasil.

Chaves, A.A.; Correa, F.A.; Lorena, L.A.N. (2007), Clustering Search Heuristic for the Capacitated p-median Problem. ***Springer Advances in Software Computing Series***, 44: 136–143.

Chaves, A.A.; Lorena, L.A.N; Miralles, C. (2009). Hybrid metaheuristic for the Assembly Line Worker Assignment and Balancing Problem. ***Lecture Notes in Computer Science***, 5818: 1–14.

Chaves, A.A; Lorena, L.A.N. (2010). Clustering Search Algorithm for the Capacitated Centered Clustering Problem. ***Computers and Operations Research***, 37(3): 552–558.

Glover, F; Laguna, M.; Marti, R. (2000). Fundamentals of scatter search and path relinking. ***Control and Cybernetics***, 39: 653–684.

Glover, F. (1996). Tabu search and adaptive memory programing. ***Interfaces in computer science and operations research***: 1–75.

Hakimi, S. (1964). Optimum location of switching centers and the absolute centers and the medians of a graph. ***Operations Research***, v. 12, p. 450–459.

Holland, J.H. (1975). Adaptation in Natural and Artificial Systems. ***Technical Report***. University of Michigan Press, Michigan, USA.

Mladenovic, N.; Hansen, P. (1997). Variable neighborhood search. ***Computers and Operations Research***, v. 24, n. 11, p. 1097–1100.

Message Passing Interface Forum. ***MPI: A Message-Passing Interface Standard***, Version 3.0. High Performance Computing Center Stuttgart (HLRS). 852 pp., 2012.

Mulvey, J.; Beck, P. (1984). Solving capacitated clustering problems. ***European Journal of Operational Research***, 18(3): 339–348.

Muritiba, A.E.F; Negreiros, M.; Oriá, H.L.G.; Souza, M.F. (2012), A Tabu Search Algorithm for the Capacitated Centred Clustering Problem. Simpósio Brasileiro de Pesquisa Operacional, ***Anais do SBPO/CLAIO 2012***.

Nagano, M.S.; Ribeiro Filho, G.; Lorena, L.A.N. (2006). Metaheurística Híbrida Algoritmo Genético-Clustering Search para Otimização em Sistemas de Produção Flow Shop Permutacional. ***Learning and Nonlinear Models***, 4(1): 32–42.

Negreiros, M.J.; Palhano, A.W. (2006). The capacitated centred clustering problem. ***Computers and Operations Research***, 33(6): 1639–1663.

Oliveira, R.M.; Mauri, G.R.; Lorena, L.A.N. (2012). Clustering Search for the Berth Allocation Problem. ***Expert Systems with Applications***, 39: 5499–5505.

Oliveira, A.C.M.; Chaves, A.A.; Lorena, L.A.N. (2013), Clustering Search. ***Pesquisa Operacional***, v.33 (1), 105-121.

Oliveira, A.C.M.; Lorena, L.A.N. (2007). Hybrid Evolutionary Algorithms and Clustering Search. ***Hybrid Evolutionary Systems: Studies in Computational Intelligence***, 75: 81–102.

Ribeiro, G.M.; Laporte, G.; Mauri, G.R. (2011). A comparison of three metaheuristics for the workover rig routing problem. ***European Journal of Operational Research***: <http://dx.doi.org/10.1016/j.ejor.2012.01.031>

Syswerda, G. (1989). Uniform crossover in genetic algorithms. In: ***Proceedings of International conference on genetic algorithms:*** 2–9.